μPD7281
μPD9305
IMAGE PIPELINED PROCESSOR

USER'S MANUAL

μPD7281
μPD9305
IMAGE PIPELINED PROCESSOR

USER'S MANUAL

A) μPD7281 IMAGE PIPELINED PROCESSOR (IMPP)
- PRODUCT DESCRIPTION
- ELECTRICAL SPECIFICATION

B) μPD9305 MEMORY ACCESS AND GENERAL BUS INTERFACE CHIP (MAGIC) / SUPPORT CHIP FOR μPD7281
- PRODUCT DESCRIPTION
- ELECTRICAL SPECIFICATION

C) μPD7281
- APPLICATION LIBRARY FOR IMAGE PIPELINED PROCESSOR

D) EBIBM-7281GS EVALUATION BOARD
- USER'S MANUAL

E) FAXXX-XXXX-7281 ASSEMBLER AND SIMULATOR SOFTWARE
- USER'S MANUAL

TABLE OF CONTENTS

1. PRODUCT DESCRIPTION uPD7281

Chapter 1 Overview 1

1.1 What is an Image Pipelined Processor? 1
1.2 What is the Data Flow Method? 1
1.3 Features 6
1.4 Pin Connections 8
1.5 Pin Functions 9
1.6 uPD7281 Block Diagram 11
1.7 Functional Outline of Each Block 11

Chapter 2 Functions of Each Block 13

2.1 Input Controller (IC) 13
2.2 Link Table (LT) 14
2.2.1 LT Field Format 15
2.2.2 LT Input/Output Token Formats 16
2.3 Function Table (FT) 18
2.3.1 FT Field Format 19
2.4 Address Generator and Flow Controller (AG & FC) .. 21
2.5 Data Memory (DM) 21
2.6 Queue (Q) 22
2.7 Processing Unit (PU) 23
2.8 Output Queue (OQ) 24
2.9 Output Controller (OC) 25
2.10 Refresh Controller (RC) 26

Chapter 3 uPD7281 Operating Procedure 27

3.1 Reset Operation 28
3.1.1 Module Number Set 28
3.1.2 Internal Pipeline Initialization 29
3.1.3 IACK/ and OREQ/ Initialization 29
3.1.4 Internal Mode Initialization 29
3.1.5 Internal Counter Initialization 29
3.2 Program Load 29
3.2.1 DM Set 30
3.2.2 FT Set 30
3.2.3 LT Set 30
3.3 Starting the uPD7281 Program 30
3.4 Terminating the uPD7281 Program 31
3.4.1 Status Polling 31
3.4.2 Other Processing 31
3.5 uPD7281 Token Transformations 31
3.5.1 Basic Internal Data Transitions 32
3.5.2 Effective Pipeline Processing 37

Chapter 4 Input/Output Tokens 39

4.1 Execution Tokens 41
4.2 Object Program Set 41
4.3 Object Program Read 44
4.4 Command Reset 46
4.5 Internal Mode Transitions 47
4.5.1 NORMAL Mode 47
4.5.2 TEST Mode 47
4.5.3 BREAK Mode 47
4.5.4 Input Restriction Mode 52
4.6 Other Input/Output Tokens 54
4.6.1 Pass Tokens (PASS and PASSD) 54
4.6.2 Vanish Tokens (VAN) 54

Chapter 5 Instructions 55

5.1 AG & FC Instructions 56
5.1.1 QUEUE 60
5.1.2 RDCYCS (Read Cyclic Short) 65
5.1.3 RDCYCL (Read Cyclic Long) 69
5.1.4 WRCYCS (Write Cyclic Short) 71
5.1.5 WRCYCL (Write Cyclic Long) 74
5.1.6 RDWR (Read/Write) 75
5.1.7 RDIDX (Read Index) 80
5.1.8 PICKUP 83
5.1.9 COUNT 87
5.1.10 CUT 91
5.1.11 DIVCYC (Divide Cyclic) 95
5.1.12 DIV (Divide) 98
5.1.13 DIST (Distribution) 101
5.1.14 CONVO (Convolution) 103
5.1.15 SAVE 106
5.1.16 CNTGE (Count Generation) 110
5.2 PU (Processing Unit) Instructions 114
5.2.1 Logical Operation Instructions 134
5.2.2 Arithmetic Operation Instructions 135
5.2.3 Shift Instructions 138
5.2.4 Comparison Instructions 141
5.2.5 Bit Manipulation Instructions 143
5.2.6 Bit Check Instructions 145
5.2.7 Data Conversion Instructions 147
5.2.8 Double Precision Adjustment 148
5.2.9 Cumulative Addition 149
5.2.10 C Bit Copy 153
5.3 GE Instructions 154
5.3.1 COPYBK (Copy Block) 157
5.3.2 COPYM (Copy Multiple) 159
5.3.3 SETCTL (Set Control Field Data) 161
5.4 OUT Instructions 172
5.4.1 OUT1 172
5.4.2 OUT2 173

Chapter 6 Multiprocessor Operation 176

6.1 Hardware Configuration 176
6.2 Software Configuration 176
6.3 Multiprocessing and "Bus Neck" (Memory Access Bottleneck) 177
6.4 Number of Processors Used and Performance 180
6.5 Synchronization 180

Chapter 7 Interfacing with the Host 182

7.1 I/O Ports 184
7.2 Function Control 185

Chapter 8 Accessing the Image Memory (IM) 190

8.1 IM Access Tokens 191
8.2 IM Access Procedures 192
8.3 IM Access in a Multiprocessor Configuration 193

Appendix A Flowgraphs 195

A.1 Flowgraph Explanation (outline) 195
A.2 Flowgraph Functions 196
A.3 Flowgraphs and the uPD7281 Assembler 198
A.3.1 Arcs and LINK Statements 199
A.3.2 Nodes and FUNCTION Statements 199
A.3.3 MEMORY Statements 199
A.4 Flowgraph Examples 206

Appendix B uPD7281 Development Support Tools 209

B.1 Assembler 213
B.2 Software Simulator 214
B.3 Object Converter 215

2. ELECTRICAL SPECIFICATION 217

This document introduces a new product, still under development, and its functions. The descriptions are therefore subject to change without advance notice.

TABLE OF CONTENTS

1. PRODUCT DESCRIPTION μPD9305R

Introduction i

Chapter 1 Overview 1

1.1 General 1
1.2 Features 1
1.3 Pin Configuration 2
1.4 uPD9305 Block Diagram 4
1.5 System Configuration Example 5

Chapter 2 Pin Functions 7

2.1 CLK (Clock) 8
2.2 RESET/ 8
2.3 RD/ (Read) 8
2.4 WR/ (Write) 8
2.5 A1, A0 (Address) 9
2.6 D15 - D0 (Data Bus) 9
2.7 CS/ (Chip Select) 9
2.8 OREQ/ (Output Request) 9
2.9 OACK/ (Output Acknowledge) 10
2.10 ODB15 - ODB0 (Output Data Bus) 10
2.11 IREQ/ (Input Request) 10
2.12 IACK/ (Input Acknowledge) 10
2.13 IDB15 - IDB0 (Input Data Bus) 10
2.14 MN3 - MN0 (Module Number) 10
2.15 IPPRST/ (Image Pipelined Processor Reset) 11
2.16 IMA23 - IMA0 (Image Memory Address) 11
2.17 IMD17 - IMD0 (Image Memory Data) 11
2.18 IMRD (Image Memory Read) 11
2.19 IMWR (Image Memory Write) 11
2.20 IMRF (Image Memory Refresh) 11
2.21 IMAK/ (Image Memory Acknowledge) 12
2.22 DMAAEN (DMA Address Enable) 12
2.23 DMARQ1/ (DMA Request 1) 12
2.24 DMARQ2/ (DMA Request 2) 12
2.25 DMAAK1/ (DMA Acknowledge 1) 12
2.26 DMAAK2/ (DMA Acknowledge 2) 12
2.27 CPURQ (CPU Request) 12
2.28 INBUSY (Input Busy) 13
2.29 SOLBSY (Self Object Load Busy) 13
2.30 ERR (Error) 13
2.31 VDD (Power) 13
2.32 GND (Ground) 13

Chapter 3 Internal Block Functions 15

3.1 System Bus Interface 15
3.2 Image Memory Interface 16
3.3 uPD7281 Output Bus Interface 18
3.4 uPD7281 Input Bus Interface 19

Chapter 4 Host/uPD7281 Interfacing 21

4.1 Input/Output Ports 23
4.1.1 Status 25
4.2 Function Controls 27
4.2.1 Mode registers 27
4.3 Host/uPD7281 I/O Procedures 33
4.4 uPD7281 Output Control 35
4.5 uPD7281 Input Control 37
4.6 Module Number Setting 40

Chapter 5 Image Memory/uPD7281 Interfacing 43

5.1 Image Memory Access Tokens 45
5.1.1 Image memory read/write procedures 47
5.1.2 Image memory access request 48
5.2 Register Files 54
5.3 Output Control (Read Data --> uPD7281) 57
5.4 Self Object Load 61
5.5 Refresh Control 65
5.6 Read/Modify/Write 67
5.7 C and S Bits of Data Read from the Image Memory . 70

Chapter 6 DMA Request Function 71

6.1 DMA1 72
6.2 DMA2 74

Appendix A System Circuit Diagram Example 83

2. ELECTRICAL SPECIFICATION 85

TABLE OF CONTENTS

APPLICATION NOTE μPD7281 - Volume I (Binary Image Processing)

Introduction 1

Chapter 1: System Configuration 3

Chapter 2: Block Transfer 5
2.1 Word Boundary Transfer 5
2.1.1 Processing Explained 5
2.1.2 Algorithm 5
2.1.3 Parameters and Their Applicable Ranges 6
2.1.4 Flow Graph Explained 7
2.1.5 Tips on Writing Flow Graphs 9
2.1.6 Assembler Source Listing 15

Chapter 3: Logical Operation 13
3.1 NOT (Single-Operand Operation) 13
3.1.1 Processing Explained 13
3.1.2 Algorithm 13
3.1.3 Parameters and Their Applicable Ranges 13
3.1.4 Flow Graphs Explained 14
3.1.5 Tips on Writing Flow Graphs 16
3.1.6 Assembler Source Listing 16
3.2 AND, OR, Exclusive OR (Double-Operand Operations) . 18
3.2.1 Processing Explained 18
3.2.2 Algorithm 18
3.2.3 Parameters and Their Applicable Ranges 18
3.2.4 Flow Graph Explained 19
3.2.5 Tips on Writing Flow Graphs 21
3.2.6 Assembler Source Listing 21

Chapter 4: Enlargement and Shrinking 23
4.1 Simple One-Half Shrinking 23
4.1.1 Processing Explained 23
4.1.2 Algorithm 23
4.1.3 Parameters and Their Applicable Ranges 24
4.1.4 Flow Graph Explained 26
4.1.5 Tips on Writing Flow Graphs 30
4.1.6 Assembler Source Listing 30
4.2 Four-Point OR One-Half Shrinking 33
4.2.1 Processing Explained 33
4.2.2 Algorithm 33
4.2.3 Parameters and Their Applicable Ranges 34
4.2.4 Flow Graph Explained 35
4.2.5 Tips on Writing Flow Graphs 40
4.2.6 Assembler Source Listing 42
4.3 Neighboring 16-Point Addition One-Quarter Shrinking 45
4.3.1 Processing Explained 45
4.3.2 Algorithm 45
4.3.3 Parameters and Their Applicable Ranges 47
4.3.4 Flow Graph Explained 48
4.3.5 Assembler Source Listing 51

4.4 Simple Double Enlargement 55
4.4.1 Processing Explained 55
4.4.2 Algorithm 55
4.4.3 Parameters and Their Applicable Ranges 56
4.4.4 Flow Graph Explained 57
4.4.5 Assembler Source Listing 60
4.5 Simple Quadruple Enlargement 64
4.5.1 Processing Explained 64
4.5.2 Algorithm 64
4.5.3 Parameters and Their Applicable Ranges 65
4.5.4 Flow Graph Explained 66
4.5.5 Tips on Writing Flow Graphs 68
4.5.6 Assembler Source Listing 68

Chapter 5: Affine Transformation 71
5.1 Processing Explained 71
5.2 Algorithm 71
5.3 Parameters and Their Applicable Ranges 75
5.4 Flow Graph Explained 77
5.5 Tips on Writing Flow Graphs 79
5.6 Assembler Source Listing 79

Chapter 6: Profiling 83
6.1 Horizontal Profiling 83
6.1.1 Processing Explained 83
6.1.2 Algorithm 83
6.1.3 Parameters and Their Applicable Ranges 85
6.1.4 Flow Graph Explained 86
6.1.5 Assembler Source Listing 89
6.2 Vertical Profiling 91
6.2.1 Processing Explained 91
6.2.2 Algorithm 91
6.2.3 Parameters and Their Applicable Ranges 93
6.2.4 Flow Graph Explained 94
6.2.5 Assember Source Listing 97

Chapter 7: 3 x 3 Masking 99
7.1 Common Processing (Image Memory Address Generation) 99
7.1.1 Processing Explained 99
7.1.2 Algorithm 100
7.1.3 Parameters and Their Applicable Ranges 102
7.1.4 Flow Graph Explained 103
7.1.5 Tips on Writing Flow Graphs 106
7.1.6 Assembler Source Listing 106
7.2 Mask Operations 106
7.2.1 Smoothing 107
7.2.1.1 Processing Explained 107
7.2.1.2 Algorithm 108
7.2.1.3 Flow Graph Explained 111
7.2.1.4 Assembler Source Listing 114
7.2.2 Thinning 118
7.2.2.1 Processing Explained 118
7.2.2.2 Algorithm 118
7.2.2.3 Flow Graph Explained 121
7.2.2.4 Tips on Writing Flow Graphs 124

7.2.2.5 Assembler Source Listing 124
7.2.3 Edge Detection 129
7.2.3.1 Processing Explained 129
7.2.3.2 Algorithm 129
7.2.3.3 Flow Graph Explained 131
7.2.3.4 Assembler Source Listing 133

Appendix A: Image Memory Read/Write 137

Table of Contents

Application Note uPD7281 - Volume II (Gray Scale Image Processing)

Chapter

1 System Configuration 1
Image Memory Organization 2
Image Memory Organization for use in Gray Scale Image Display 4

2 Binarization 9
Algorithm 9
Parameters 11
Assembler-Coded Parameters 11
Start-Up Token Defined Parameters 11
Flow Graph Explanation 11
Tips on Preparing Flow Graphs 15

3 Continuous-Tone Transforamtion 21
Algorithm 21
Parameters 23
Assembler-Coded Parameters 23
Start-Up Token Defined Parameters 23
Flow Graph Expalanation 23

4 Dither Transformation 33
Algorithm 33
Parameters 36
Assembler-Coded Parameters 36
Start-Up Token Defined Parameters 36
Mask Tables 36
Flow Graph Explanation 37
Typical Dither Matrices 41

5 Mask Processing 47
Algorithm 47
Computation Precision 50
Parameters 51
Assembler-Coded Parameters 51
Start-Up Token Defined Parameters 51
Flow Graph Explanation 52
Using a Mask Table 60
Averaging 60
Gradient 60
Laplacian 61
Noise Removal 62
Image Emphasis 62

6 Affine Transformation 714
Algorithm 71
Processing Flow 73
Computation Precision 73
Parameters 75
Assembler-Coded Parameters 75
Start-Up Token Defined Parameters 76
Flow Graph Explanation 77

TABLE OF CONTENTS

APPLICATION NOTE μPD7281 - Volume III (Numerical Calculations)

Introduction 1

Chapter 1: System Configuration 2
1.1 System Configuration 2
1.2 Data Input Methods 3
1.2.1 Setting up the Input Circuit 3
1.2.2 Generating Memory Address 4

Chapter 2: Matrix-Vector Product 5
2.1 Processing Explained 5
2.2 Algorithm 5
2.3 17 bit x 17 bit = 17 bit 6
2.3.1 Algorithm 6
2.3.2 Range of Allowable Parameter Values 7
2.3.3 The Flow Graph Explained 9
2.4 17 bit x 17 bit = 33 bit 14
2.4.1 Algorithm 14
2.4.2 Range of Allowable Parameter Values 14
2.4.3 Explanation of the Flow Graph 16
2.5 33 bit x 33 bit = 33 bit 22
2.5.1 Algorithm 22
2.5.2 Range of Allowable Parameter Values 23
2.5.3 The Flow Graph Explained 25

Chapter 3: Evaluation of Polynomials 33
3.1 Processing Explained 33
3.2 Algorithm 33
3.3 17 bit x 17 bit = 17 bit 34
3.3.1 Algorithm 34
3.3.2 Range of Allowable Parameter Values 34
3.3.3 Explanations of the Flow Graph 35
3.4 33 bit x 33 bit = 33 bit 40
3.4.1 Algorithm 40
3.4.2 Range of Allowable Parameter Values 41
3.4.3 Explanations of the Flow Graph 42
3.5 Computation of Cosine by Taylor Series Expansion . 45

Chapter 4: Multiplication 51
4.1 Processing Explained 51
4.2 33 bit x 33 bit = 65 bit 51
4.2.1 Algorithm 51
4.2.2 The Allowable Range of Parameter Values ... 52
4.2.3 Explanations of the Flow Graph 53
4.3 49 bit x 49 bit = 97 bit 58
4.3.1 Algorithm 58
4.3.2 Allowable Range of Parameter Values 60
4.3.3 Explanations of the Flow Graph 61

Chapter 5: Floating-Point Computations 69
5.1 Processing Explained 69
5.2 Floating-Point Multiplication 70
5.2.1 Algorithm 70
5.2.2 Allowable Range of Parameter Values 71
5.2.3 Explanations of the Flow Graph 71
5.3 Floating-Point Addition 76
5.3.1 Algorithm 76
5.3.2 Allowable Range of Parameter Values 77
5.3.3 Explanations of the Flow Graph 77

Chapter 6: Distance Squared Calculations 85
6.1 Processing Explained 85
6.2 Algorithm 85
6.3 Range of Allowable Parameter Values 86
6.4 Explanations of the Flow Graph 87
6.5 Tips on Preparing Flow Graphs 90

Chapter 7: Fast Fourier Transform (FFT) 93
7.1 Processing Explained 93
7.2 Algorithm 93
7.3 Range of Allowable Parameter Values 99
7.4 Flow Graph Explained 100

Table of Contents

User's Manual EBIBM-7281GS

Chapter 1 Introduction
1.1 Introduction 1
1.2 General Presentation 2

Chapter 2 The Software Organization
2.1 The Software Organization 3
2.2 A page screen organization 6
2.3 Human interface 9
2.4 Help procedure 11

Chapter 3 Image Library
3.1 The library objects 13
3.2 Image management utilities 15
3.3 LUT management utilities 17
3.4 Grey level images 19
3.4.1 Image display 19
3.4.2 Image enhancement 21
3.4.3 Logical operations between an image and a constant 23
3.4.4 Logical operations between two images 23
3.4.5 Arithmetical operations between an image and a constant .23
3.4.6 Arithmetical operations between two images 24
3.4.7 Filtering 25
3.4.8 Utilities 26
3.5 Binary images 28
3.5.1 Logical operations 28
3.5.2 Processings 30

Appendix A List of error messages 31

TABLE OF CONTENTS

USER'S MANUAL FAXXX-XXXX-7281

PREFACE

PART I OUTLINE OF THE SOFTWARE PACKAGE.....1-1

INTRODUCTION.....1-2

USING THE SOFTWARE PACKAGE.....1-3

CHAPTER 1 OUTLINE.....1-6

1.1 uPD7281 Assembler.....1-6

1.2 uPD7281 Simulator.....1-6

1.3 uPD7281 Object-code Conversion Program.....1-6

1.4 Development of programs using the Software Package.....1-6

CHAPTER 2 PROGRAM DEVELOPMENT PROCEDURES.....1-9

2.1 Creating source module file.....1-9

2.2 Execution of Assembler (creation of object module file).....1-9

2.3 Simulation.....1-9

2.4 Conversion to HEX-format or ASCII-data-format object module.....1-12

PART II USAGE OF THE ASSEMBLER.....2-1

CHAPTER 1 OUTLINE OF SOFTWARE.....2-2

1.1 Outline of the assembler's functions.....2-2

1.2 Files handled by the assembler and their relationships.....2-4

1.3 Assembler's features.....2-5

CHAPTER 2 BASIC USAGE 2-7

2.1 Flowgraph.....2-7

2.2 General organization of source modules.....2-9

2.3 List of statements.....2-10

2.4 Starting the sssembler.....2-11

CHAPTER 3 SOURCE MODULE FORMAT.....2-12

3.1 Organization of the source module.....2-12
3.1.1 Locations where statements can be used.....2-12
3.1.2 Organization of the Declaration Section....2-13
3.1.3 Organization of the Execution Section.....2-14
3.1.4 Organization of the Data Section.....2-16

3.2 Statements.....2-17
3.2.1 Statement field.....2-17
3.2.2 Symbol field.....2-18
3.2.3 Operand field.....2-18
3.2.4 Comments.....2-18
3.2.5 Tab function.....2-18

3.3 Character set.....2-19
3.3.1 Alphanumeric characters.....2-20
3.3.2 Numeric digits.....2-20
3.3.3 Use of special characters.....2-20
3.3.4 Use of control characters.....2-21

3.4 Symbols.....2-21
3.4.1 Rules on using symbols.....2-22
3.4.2 Remarks on coding symbols.....2-23

3.5 Coding format for the operand field.....2-24
3.5.1 Reserved words.....2-24
3.5.2 Symbols.....2-25
3.5.3 Numeric constants.....2-25
3.5.4 Expressions.....2-27
3.5.5 Character strings.....2-27

3.6 Operators....2-27
3.6.1 Order of precedence of operators....2-28
3.6.2 Arithmetic operators....2-29
3.6.3 Logical operators....2-30
3.6.4 Shift operators....2-30
3.6.5 Other operators....2-31
3.6.6 Restriction on the operators and their operands....2-31

CHAPTER 4 STATEMENTS....2-34

4.1 Program structuring statements....2-34
4.1.1 MODULE....2-34
4.1.2 START....2-35

4.2 Name definition statements....2-37
4.2.1 NAME....2-37
4.2.2 LITERAL....2-37
4.2.3 EQUATE....2-38
4.2.4 ADDRESS....2-39

4.3 Input/Output statements....2-41
4.3.1 INPUT....2-41
4.3.2 OUTPUT....2-42

4.4 Memory allocation statements....2-44
4.4.1 LOCATE....2-44
4.4.2 MEMORY....2-45
4.4.3 DMSETLT....2-47
4.4.4 DMSETFT....2-48

4.5 Execution statements....2-49
4.5.1 LINK....2-49
4.5.2 FUNCTION....2-54
4.5.3 DEFINE....2-59

4.6 Data definition statements....2-61
4.6.1 DATA....2-61

4.7 Assembly end statement....2-66
4.7.1 END....2-66

CHAPTER 5 INPUT/OUTPUT FILES....2-67

5.1 Input/Output files handled by the Assembler....2-67
5.1.1 File types....2-68
5.1.2 Input/Output classification and media....2-70

5.2 Output lists.....2-71
5.2.1 Assembly list.....2-72
5.2.2 Error list.....2-73
5.2.3 Mnemonic-format object code list.....2-74
5.2.4 HEX-format object code list.....2-78
5.2.5 Symbol cross-reference list.....2-79

CHAPTER 6 THE ASSEMBLER OPERATION PROCEDURES.....2-81

6.1 Sequence of operations.....2-81

6.2 Controls.....2-84
6.2.1 Basic controls.....2-84
6.2.2 General controls.....2-97

6.3 Rules on file names.....2-103

CHAPTER 7 EXECUTION EXAMPLE.....2-105

APPENDIX 1 LIST OF RESERVED WORDS.....A2-113

APPENDIX 2 COMMAND NAMES AND PARAMETER LIST FORMAT FOR DATA STATEMENT.....A2-114

APPENDIX 3 ERROR MESSAGES.....A2-115

APPENDIX 4 CODING FORMAT OF INSTRUCTIONS.....A2-125

PART III USAGE OF THE SIMULATOR.....3-1

CHAPTER 1 OUTLINE OF THE SOFTWARE.....3-2

1.1 Outline of Simulator functions.....3-2
1.1.1 Simulation Model.....3-2
1.1.2 Symbolic debug.....3-5

1.2 Organization of files handled by the Simulator....3-5
1.2.1 Work file.....3-6

CHAPTER 2 DETAILS OF SIMULATION MODELS.....3-7

2.1 Interaction between blocks.....3-8
2.1.1 Input Control Block.....3-8
2.1.2 Output Control Block.....3-9
2.1.3 IM Block.....3-11
2.1.4 uPD7281 Block.....3-11

2.2 uPD7281 simulation model.....3-13
2.2.1 IC Block (Input Controller).....3-13
2.2.2 LTT Block (Link Table Transfer).....3-14
2.2.3 OC Block (Output Controller).....3-15
2.2.4 RC Block (Refresh Controller).....3-16
2.2.5 IOC Block (Input/Output Controller).....3-17

2.3 IM Block model.....3-18
2.3.1 Model of IM Block in MAGIC mode.....3-18
2.3.2 Model of IM Block in NON-MAGIC mode.....3-21

CHAPTER 3 INPUT/OUTPUT FILE FORMAT.....3-23

3.1 File name.....3-23

3.2 Rules on creating files.....3-24
3.2.1 Character code.....3-24
3.2.2 Number.....3-24

3.3 Input files.....3-24
3.3.1 Object module file.....3-24
3.3.2 Input data file.....3-24
3.3.3 Macro file.....3-26
3.3.4 Simulation resource load file.....3-27
3.3.5 Include file.....3-27

3.4 Output files.....3-28
3.4.1 Macro file.....3-28
3.4.3 Simulation resource save file.....3-28
3.4.4 Output data file.....3-28
3.4.5 Trace-log keeping file.....3-28
3.4.6 Trace-log display format.....3-30

CHAPTER 4 BASIC USAGE OF THE SIMULATOR.....3-35

4.1 Character set.....3-35
4.1.1 Use of special characters.....3-35
4.1.2 Use of control characters.....3-37

4.2 Expressions and Operators.....3-37
4.2.1 Order of precedence of operators.....3-38
4.2.2 Arithmetic operators.....3-39
4.2.3 Logical operators.....3-39
4.2.4 Shift operators.....3-40
4.2.5 Relational operators.....3-41
4.2.6 Other operators.....3-42

4.3 The Simulator Operation Sequence.....3-43

4.4 The Simulator Command Format.....3-45

4.5 A Set of Simulator Commands.....3-48

CHAPTER 5 THE SIMULATOR COMMANDS.....3-52

5.1 Set/Display Simulation Environment commands.....3-52
5.1.1 RESET.....3-53
5.1.2 ENVIRON.....3-54
5.1.3 STATUS.....3-65

5.2 Set/Display Input Timing commands.....3-66
5.2.1 TIMING=timing specification.....3-67
5.2.2 TIMING.....3-68

5.3 Load Object Module command.....3-69
5.3.1 LOAD OBJECT.....3-70

5.4 SimulationInput/OutputDataFileSpecification commands.....3-74
5.4.1 DATA INM.....3-75
5.4.2 DATA OTM.....3-76

5.5 Trace-log commands.....3-77
5.5.1 MAP.....3-78
5.5.2 PRINT.....3-79

5.6 Host Data Input command.....3-81
5.6.1 HOST.....3-82

5.7 Simulation Execution commands.....3-83
5.7.1 GO.....3-84
5.7.2 STEP.....3-91
5.7.3 CONTINUE.....3-92

5.8 Set/Display/Cancel GR Break Condition commands....3-94
5.8.1 GR=GR break condition.....3-95
5.8.2 GR.....3-96
5.8.3 NOGR.....3-97

5.9 Set/Display/Cancel Conditional Breakpoint Register commands.....3-98
5.9.1 BRn=break condition.....3-99
5.9.2 BR[n].....3-100
5.9.3 NOBR[n].....3-101

5.10 Set/Display/Cancel Unconditional Breakpoint Register commands.....3-102
5.10.1 BPn=break condition.....3-103
5.10.2 BP[n].....3-104
5.10.3 NOBP[n].....3-105

5.11 Set/Display/Cancel Trace Item commands.....3-106
5.11.1 DEFINE TRACE.....3-107
5.11.2 TRACE.....3-110
5.11.3 NOTRACE.....3-111
5.11.4 TON=trace condition.....3-112
5.11.5 TON.....3-114
5.11.6 NOTON.....3-115
5.11.7 TOFF=trace condition.....3-116
5.11.8 TOFF.....3-117
5.11.9 NOTOFF.....3-118

5.12 Symbolic Debug commands.....3-119
5.12.1 CALCULATE.....3-120
5.12.2 DEFINE SYMBOL.....3-121
5.12.3 NOSYMBOL.....3-125
5.12.4 .symbol=.....3-126
5.12.5 address expression=.....3-127
5.12.6 SYMBOLS.....3-129
5.12.7 .symbol.....3-130
5.12.8 address expression.....3-131

5.13 Assemble commands.....3-133
5.13.1 ASSEMBLE.....3-134
5.13.2 DISASSEMBLE.....3-136

5.14 Macro Function commands.....3-138
5.14.1 REPEAT.....3-139
5.14.2 COUNT.....3-140
5.14.3 IF.....3-142
5.14.4 MACRO.....3-145
5.14.5 INCLUDE.....3-152
5.14.6 WRITE.....3-154
5.14.7 ECHO.....3-155
5.14.8 NOECHO.....3-156

5.15 Display Pipeline Step command.....3-157
5.15.1 CLOCK.....3-158

5.16 Set/Display Current Module Number commands.....3-159
5.16.1 #=.....3-160
5.16.2 #.....3-161

5.17 Evaluation commands.....3-162
5.17.1 LTEVAL.....3-163
5.17.2 PUEVAL.....3-164
5.17.3 IMEVAL.....3-165
5.17.4 NOEVAL.....3-166

5.18 Save/Load Simulation Resource commands.....3-167
5.18.1 SAVE RESOURCE.....3-168
5.18.2 LOAD RESOURCE.....3-169

5.19 Set/Cancel Listing Function commands.....3-171
5.19.1 LIST.....3-172
5.19.2 NOLIST.....3-173

5.20 DUMP ASCII File command.....3-174
5.20.1 TYPE.....3-175

5.21 End Simulation command.....3-176
5.21.1 EXIT.....3-177

CHAPTER 6 EXECUTION EXAMPLE.....3-178

APPENDIX 1 LIST OF RESERVED WORDS.....A3-186

APPENDIX 2 LIST OF SYMBOL TYPES.....A3-188

APPENDIX 3 SPECIFICATION/DISPLAY FORMAT OF ASSEMBLY CODE.....A3-191

APPENDIX 4 LIST OF ERROR MESSAGES OF THE SIMULATOR.....A3-198

PART IV USAGE OF THE OBJECT-CODE CONVERSION PROGRAM.....4-1

CHAPTER 1 OUTLINE OF THE SOFTWARE.....4-2

1.1 Outline of the Functions of the Object-code Conversion Program.....4-2

1.2 Files handled by the Object-code Conversion Program and their relationshipS.....4-3

CHAPTER 2 INPUT/OUTPUT FILES.....4-4

2.1 Types of files.....4-4

2.2 File formats.....4-4
2.2.1 Object module file.....4-4
2.2.2 HEX-format object module file.....4-4
2.2.3 ASCII-data-format object module files.....4-11
2.2.4 Symbol table module file.....4-14

CHAPTER 3 OPERATION PROCEDURES OF THE OBJECT-CODE CONVERSION PROGRAM.....4-16

3.1 Operation Procedures of the Object-code Conversion Program.....4-16
3.1.1 How to invoke the Object-code Conversion Program.....4-16
3.1.2 The termination of the Object-code Conversion Program.....4-18

3.2 Control Specification.....4-18
3.2.1 Module information output specification.....4-19
3.2.2 Data information output specification.....4-20
3.2.3 Symbol table module output specification...4-20
3.2.4 Object module output format specification..4-21
3.2.5 HEX-format object module start address specification.....4-21
3.2.6 MAGIC information adding specification.....4-22

3.3 Rules on file names.....4-23

3.4 Execution examples.....4-25

APPENDIX 1 LIST OF ERROR MESSAGES.....A4-27

PART V VAX/UNIX BASED SOFTWARE.....5-1

A

μPD7281

IMAGE PIPELINED PROCESSOR

(IMPP)

PRODUCT DESCRIPTION

Chapter 1

1.1 What is an Image Pipelined Processor?

In recent years, with improvements in image processing technology, there has been a wide range of attempts to put image processing to practical use. Until now, however, these applications have been limited to large scale systems such as those used for remote sensing, medical images (scanning), industrial robots (product sorting, etc.), and signature verification. This has been because the massive quantities of image data require correspondingly huge amounts of processing time and the adoption of large scale mainframe computers or dedicated image processors (on a scale of several boards).

In the work place, the continued development of office automation has brought about greater availability of word processors and graphic controllers, popularizing character and graphic editing. Image work stations, capable of processing photographs and pictures, however, have lagged behind.

In this background, the advent of low cost, compact LSI capable of performing high speed image processing has been much awaited. The Image Pipelined Processor, employing a novel data flow architecture and pipeline configuration, is an LSI device developed to meet these demands. In addition to realizing high speed processing, the adoption of RAM for program memory area permits the Image Pipelined Processor to be used for a wide variety of applications.

1.2 What is the Data Flow Method?

Almost all of the computers currently on the market, including microcomputers, are based on traditional 'von Neumann' architecture. In this architecture, the object program is set in memory, and the program counter is set to point to the starting address of the program. Then the instructions in memory are fetched one after another from the location indicated by the program counter, decoded and executed.

However, in a computer adopting the data flow method, there is no (or rather, no need for) program counter. Let's consider then how a data flow method computer executes instructions.

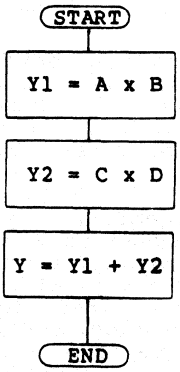
As the first example, let's look at how the following simple calculation is executed:

$$Y = A \times B + C \times D \qquad (1)$$

In a von Neumann computer, the calculation would be performed as shown in the flowchart in Figure 1.1. However, if we consider the calculation carefully, we see that there

is no reason why the two operations, Y1 = A x B and Y2 = C x D, cannot be executed in parallel (concurrently). Data flow computers enable this type of parallel operation. A data flow computer can have a number of data-driven arithmetic units that begin operation when data arrives. This feature allows asynchronous operation of each of the arithmetic units. In addition to arithmetic unit(s), a data flow computer includes a flow controller that functions to determine which data is to be sent where. The data itself includes an ID that indicates where the data originated.

Figure 1-1
Y = A x B + B x C Flowchart



Therefore, in a data flow computer program, the description indicates where the data is to be sent for processing. For this reason, unlike von Neumann computers, there is no need for the instructions to be described in the program in the order in which they are to be executed. Instead, the flow of the data to be processed must be described.

Because the flowcharts used for von Neumann computers are ill suited to the expression of data flows and parallel execution within a program, 'flowgraphs*' are used to graphically depict these programs.

* Flowgraphs and tokens
Flowgraphs consist of 'nodes' that correspond to the operators in an expression (the processing to be performed) and oriented segments, or 'arcs', that indicate the data flow. Note that flowgraphs do not express processing sequentially but are drawn with the objective of making explicit the parallel nature of certain processes. Further, data flowing between the nodes is referred to as 'tokens' and the information required for flow control is added to the tokens as they pass through the different stages of the hardware.

The flowgraph for the calculation of the expression in equation (1) is shown in Figure 1-3. Note that this flowgraph makes it clear that the two operations, Y1 = A x B and Y2 = C x D, can be executed in parallel. The differences with a von Neumann computer become even more pronounced when you actually execute instructions on a data flow computer.

Figure 1-2
Data Flow Method

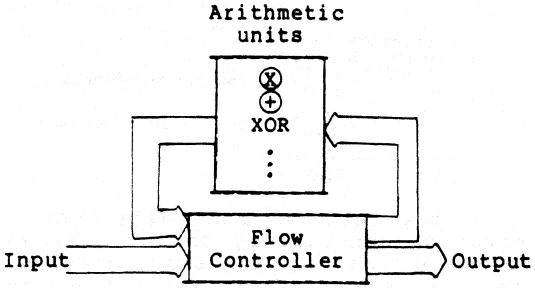
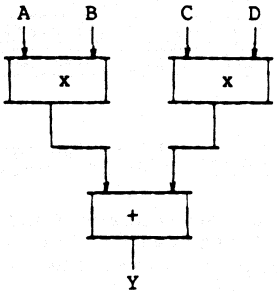


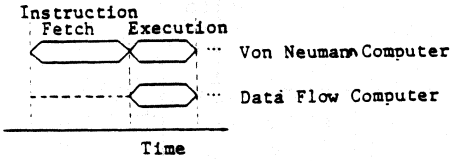
Figure 1-3
Flowgraph for Y = A x B + C x D



To execute an instruction using a von Neumann computer, the object program must first be read from the location where it is set in memory; the instruction read is then decoded and the data required for execution assembled. A large proportion of the time required for instruction execution is thus spent for memory access. Traditional computer architecture can therefore not be considered optimum for high-speed processing. In data flow computers, however, once the object program has been set, the instructions execute as soon as the necessary data are present.

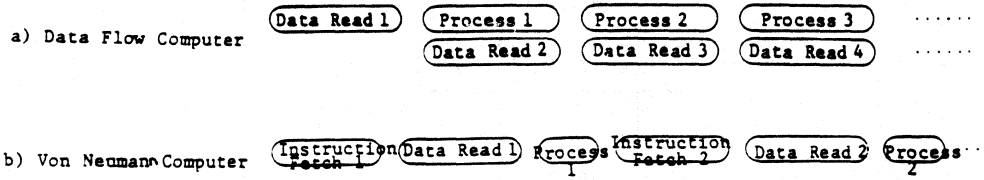
As another example, let's consider an operation that logically ANDs data A and data B. Here, it is assumed that the execution time for this operation (once the necessary data has been assembled) is the same for the conventional processor and the data flow processor. As can be seen in Figure 1-4, the high speed processing of the data flow computer far exceeds that of the conventional computer.

Figure 1-4
Timing Chart for Von Neumann and Data Flow Computers



Another benefit of computers using the data flow method is that a high degree of concurrency can be achieved. For instance, while one process is executing, another process can be assembling the data necessary for its own execution. If these data are in memory, it is possible to reduce apparent memory access time almost to zero.

Figure 1-5
Processing Procedures



To sum up, compared to a von Neumann computer, a data flow computer can reduce drastically the number of memory accesses, and make highly efficient use of the access time.

Next, let's explore how the data flow techniques have been implemented in the uPD7281. In the data flow computer shown in Figure 1-2, the device includes multiple arithmetic units. If this many units were to be mounted on a single device, a remarkable processing speed would certainly be realized. However, the complexity required of the flow controller in such a device would be prohibitive and since each unit would only perform a specific operation, a large number of units would be required when performing complex operations. Another defect in this system is that even if a large number of units were used, there is a strong risk that too much data could become concentrated in a single

arithmetic unit, causing processing to come to a halt. For these reasons we have elected to use a single arithmetic unit for the uPD7281. Since this single unit will have to perform a wide variety of operations, it must be a general purpose arithmetic unit.

In the computer described above, since each unit performs a unique task, only an ID indicating the destination arithmetic unit within the computer need be included in each token. In the case of the uPD7281, however, data that specifies the type of operation to be performed must be included as part of the token. The token input to the arithmetic unit therefore contains an Op Code. Also, the flow controller shown in Figure 1-2 can be broken down into the following four blocks:

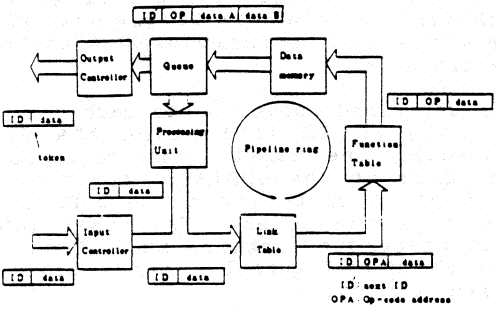
Link Table	: Checks the IDs of arriving tokens; passes the next ID value and Op Code address to the token.
Function Table	: Adds an Op Code to the token.
Data Memory	: Controls the gathering of two tokens needed for operations.
Queue	: Controls the input of tokens to the processing unit.

As the tokens pass through these blocks, data are appended to them and some tokens are merged into single tokens. Then the tokens are sent to the processing unit or output controller.

The format of tokens output from the uPD7281 via the output controller, or from the processing unit to the Link Table, is the same as those input to the Input Controller. These concepts are all presented in Figure 1-6. Because the flow between blocks is performed by a pipeline, passing of tokens between blocks is simplified. Each of these blocks - Link Table, Function Table and Data Memory - are configured in RAM so that programs can be modified simply by rewriting the contents of these memories. This is known as the programmable pipeline method. The uPD7281 executes programs using the data flow and programmable pipeline methods.

Although the uPD7281 has only a single arithmetic unit, multiple processes can progress concurrently and since external memory access, unlike that of von Neumann computers, consists only of the read and write of data, the total number of memory access can be significantly reduced. Also, because processing is performed concurrently, memory access time is almost invisible as a part of the processing time. All of these factors clearly indicate that the uPD7281 is ideally suited for use as a high speed processor.

Figure 1-6



1.3 Features

1. High speed data processing

   The pipeline architecture adopted for the Image Pipelined Processor is particularly effective for high speed handling of the kinds of iterative operations that figure so prominently in image processing. Because the modules that configured earlier pipelines were set in hardware, the types of operations that could be performed were also fixed. In the uPD7281, however, the operations performed by the pipeline can be changed by user program and multiple processes can be performed concurrently. To realize this, the uPD7281 has been configured of programmable pipeline modules and the data flow method, wherein the information that specifies the type of operation to be performed is included as part of the processing data, has been adopted.

2. High speed arithmetic operations

   The uPD7281 is capable of performing a variety of operations at high speeds. Multiplication operations in particular are greatly enhanced by the 16 x 16 bit on-chip multiplier with a 200 ns execution time (at 10 MHz), the same as for all operations. In additon, the following image processing instructions are available: multi-bit shift, bit manipulation (set, clear, get), bit check and data conversion.

3. Easy multiprocessor configuration

   The high speed processing realized by the use of a single uPD7281 can be further enhanced by using several Image Pipelined Processors in a multiprocessor configuration. This is facilitated by a simple wiring connection between uPD7281 processors.

4. High speed data I/O

   In a multiprocessor configuration, the I/O data bus is often the bottleneck limiting performance. To avoid this, the uPD7281 adopted separate input and output lines, achieving improved transmission rates. Also, to reduce the need for external circuits, an asynchronous two-line handshaking method has been adopted.

5. Wide range of applications

   Because the on-chip program memory area of the uPD7281 is configured in RAM, the user can create programs and load and execute them at will. Although the uPD7281 was designed first and foremost to perform high speed image processing, as the above features indicate, it is also suited for use as a powerful general purpose arithmetic processor. The pipelined architecture of the uPD7281 shows to maximum advantage when processing large amounts of image data where the operations are performed in succession: matrix operations, floating point arithmetic operations and calculations for all types of digital filters can all be performed at high speeds.

   As the above clearly indicates, the uPD7281 is the ideal processor for applications requiring high speed arithmetic processing or image processing of bi- or polynomial image data.

6. On-chip memories

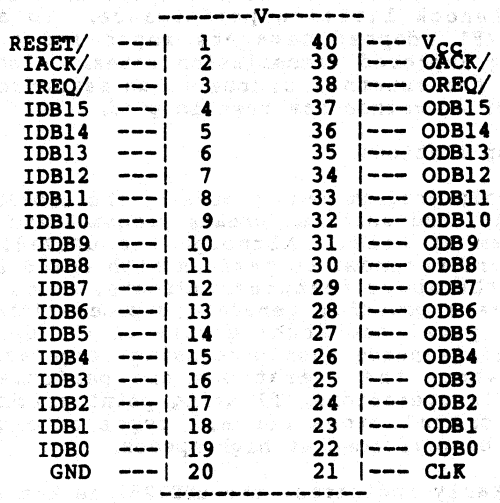
LT (Link Table)	:	128 x 16 bits
FT (Function Table)	:	64 x 40 bits
DM (Data Memory)	:	512 x 18 bits
Q (Queue)	:	48 x 60 bits
OQ (Output Queue)	:	8 x 32 bits

7, N-channel MOS

8. Single +5 volt power supply

9. 40 pin ceramic DIP

1.4 Pin connections

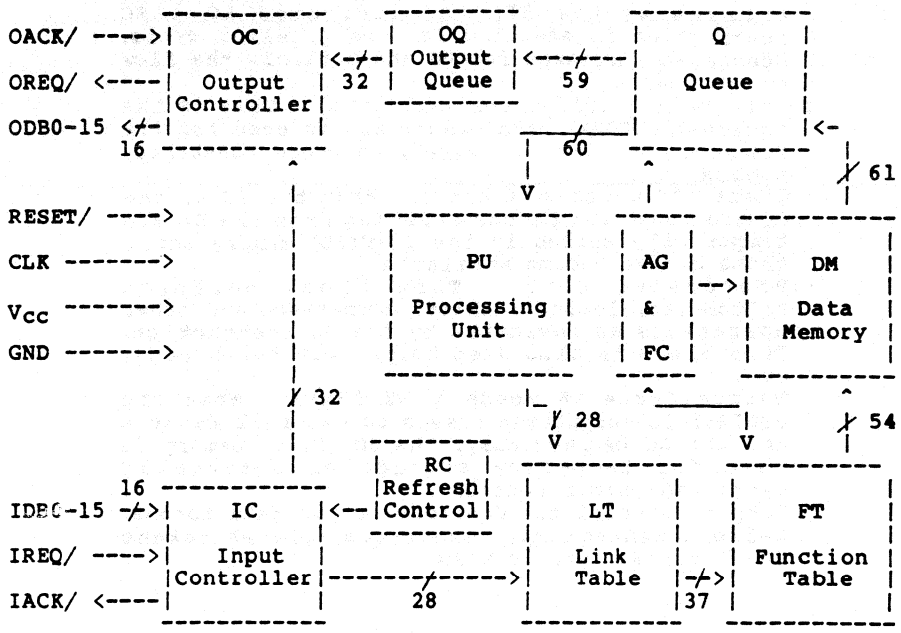


1.5 Terminal Functions

Name	I/O	Function	At reset
IDB0-IDB15	In	16-bit input data bus. 32-bit data are fetched twice in the order of the upper-order 16 bits followed by the low-order 16-bits. Data I/O control is performed by the IREQ/ and IACK/ signals.	
IREQ/	In	When data is to be input to the uPD7281 via the IDB bus, this signal goes low to inform the uPD7281. After confirming that the uPD7281 has received this signal and responded with an IACK/, IREQ/ should return to a high level.	
IACK/	Out	After the IREQ/ signal goes low, when the uPD7281 has completed preparations to accept data input, this signal goes low to inform the external data source. When IREQ/ returns to a high level, this signal also goes high.	High
ODB0-ODB15	Out	16-bit output data bus. Since all output data are configured as 32-bit words, output is performed in two steps in the order of the upper-order 16 bits followed by the lower order 16 bits. Control of data output is performed by the OREQ/ and OACK/ signals.	Hi-Z
OREQ/	Out	When data is to be output via the ODB bus, this signal goes low to inform the external destination. This signal will then return to a high level after the external destination receives this signal and responds with OACK/ brought low.	High

OACK/	In	When the external data destination has completed preparations to receive the data output from the uPD7281, the signal at this terminal should go low to inform the uPD7281. When OREQ/ goes high, this signal should also return to a high level.
RESET/	In	This is the reset input used to initialize the uPD7281.
CLK	In	System clock input
V_{CC}		+5 volt power supply
GND		Ground potential

1.6 Block Diagram



1.7 Functional Outline of Each Block

The uPD7281 consists of 10 function blocks each with its own independent function. The pipeline ring is formed by the LT, FT, AG & FC, DM, Q and PU blocks.

IC : Input Controller. Controls input data tokens and determines whether or not an input data token should be sent to the circular pipeline for processing.

LT : Link Table (128 words x 16 bits). The link data for the user program is written to this program area and corresponds to flowgraph arcs.

FT : Function Table (64 words x 40 bits). This is the program area where the instruction code data of the user program are written. The instruction code that will be executed during a single circulation of the pipeline ring is written to this table. (One instruction corresponds to a single flowgraph node.)

AG & FC : Address Generator and Flow Controller. When a token that executes an AG & FC instruction arrives at the FT, the indicated AG & FC instruction is executed for one pipeline cycle. Generates addresses for DM and controls the flow of tokens.

DM : Data Memory (512 words x 18 bits). This is the internal uPD7281 data memory and is used for the temporary storage of values, to store constants, tables, etc.

Q : Queue (48 words x 60 bits). When the PU or the OQ are busy, the tokens arriving from the DM are temporarily stored in the Q (FIFO) memory until those blocks become available.

PU : Processing Unit. This block performs arithmetic, logical, shift, comparison and other operations as indicated by the PU instruction. This block is also used to perform token copy, etc.

OQ : Output Queue (8 words x 32 bits). When the uPD7281 is outputting tokens to external devices and the OC becomes busy, the OQ (FIFO) memory is used for temporary storage of internally generated output tokens.

OC : Output Controller. Controls output data tokens.

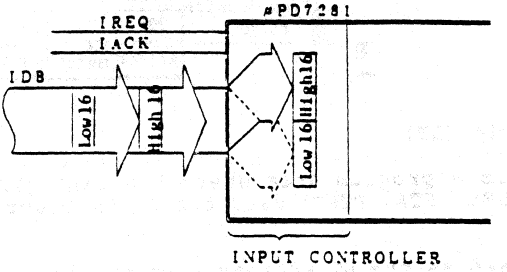
RC : Refresh Controller. Generates refresh tokens for internal dynamic RAMs.

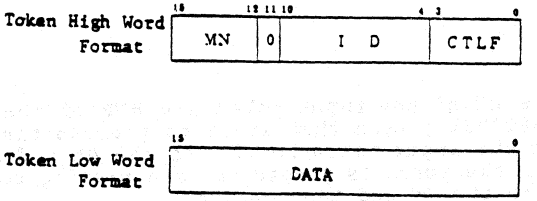
Chapter 2

FUNCTIONS OF EACH BLOCK

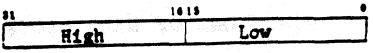
2.1 Input Controller (IC)

All tokens input to the uPD7281 must pass the IC block. All input/output tokens of the uPD7281 are 32-bits long (see the description following). However, since there are only 16 input pins, the data must be divided into high and low words. The uPD7281 does not have an input signal to identify the high or low word. This is performed by the order in which the 16-bit words are input to the uPD7281.

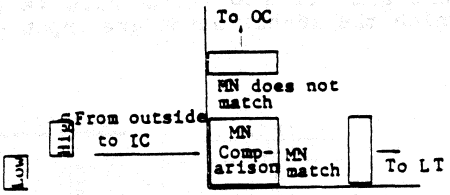




When the high and low words are input to the IC, these two 16-bit data are merged to form a 32-bit input token



Once the input token has been formed, the judgement is made whether to take the token into the uPD7281 or to output it again as a pass token. This judgment is decided by whether the module number (MN) of the input token and the MNR* value of the IC coincide.



2.2 Link Table (LT)

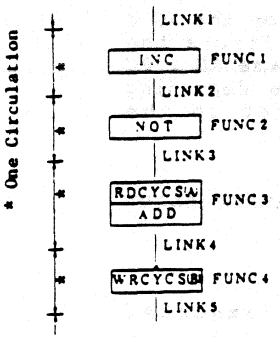
The LT is a program area where the link (ID) data and the codes (SEL, FTA, FTRC) used for FT instruction select are stored.

ID is used as the LT address when the token completes one circulation of the pipeline and returns to the LT again. The FT instruction select data (SEL, FTA, FTRC) sets the type of instruction to be performed by the token while it circulates through the pipeline ring.

* If the MN of the input token and MNR of the IC match, the uPD7821 judges that it is to process the token and sends the input token to LT. If the MN and MNR do not match, the token is judged to be a passing token and is sent to the OC and then output.

Figure 2-1
Link Table

(a) Flowgraph



(b) Data stored in LT

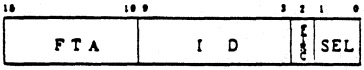
	ID	FTA.SEL.FTRC
LINK1:	LINK2	FUNC1 *
LINK2:	LINK3	FUNC2
LINK3:	LINK4	FUNC3
LINK4:	LINK5	FUNC4

* FUNC indicates the FT address that stores the instruction to be processed during the pipeline circulation.

2.2.1 LT field format

The LT is a 128-word x 16-bit memory configured in dynamic RAM. The internal fields are as shown in the following figure:

LT field format



- FTA (function table address)
This field stores the FT address that the token will access when it arrives at the FT, which is the next block of the pipeline ring.

- ID (identifier)
The token arriving at the LT uses this ID data to access the LT. The token output to the FT has a new ID defined

by the LT. This new ID is thus the LT address that will be accessed by the token after it has made a complete circulation through the pipeline ring. When the OUT instruction is used to generate an output token, however, this value will be the ID of the output token.

- FTRC (function table right control)
This field is an auxiliary instruction code for the AG & FC instructions. The value of the field affects the execution of the AG & FC instruction; the same AG & FC instruction will perform different operations depending on the field's value. For details, refer to the descriptions of the AG & FC instructions

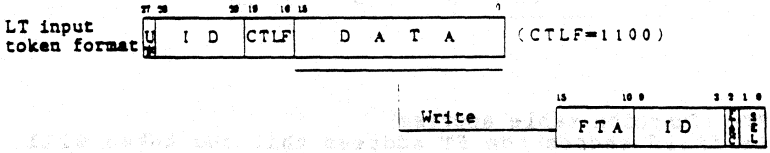
- SEL (select)
The bits of this field indicate the type of instruction stored in the FT field addressed by FTA.
When SEL = 00, OUT instruction
When SEL = 01, PU instruction
When SEL = 10, GE instruction
When SEL = 11, AG & FC instruction (executed independently*)

* AG & FC instructions, in addition to being executable by themselves, can be used in combination with other instructions. See later sections for details.

2.2.2 LT input/output token formats

When a token arrives at the LT block from the IC (input controller) or the PU (processing unit), the processing specified by the CTLF field of the token is performed. The processing performed can be divided into four types: write, read, execution and passing.

(1) Write
Data write to the LT is performed when the object program is loaded from the host. During program execution, only the GE instruction SETCTL can be used to write to the contents of the LT.

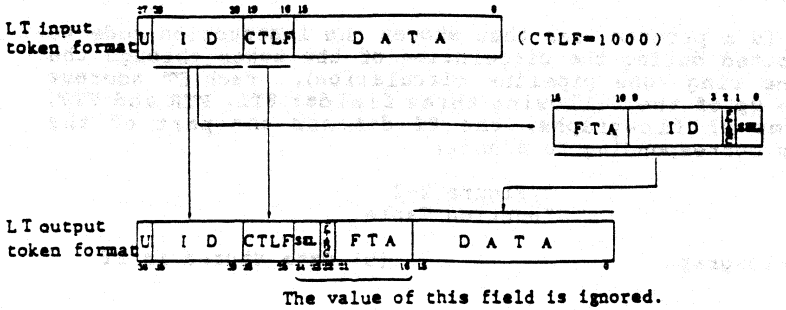


* : UNUSE bit (Specify the token is valid or not)

When a CTLF=1100B token arrives at the LT, the 16-bit data field of the token is set in the LT. After performing this data set, the token is deleted.

(2) Read

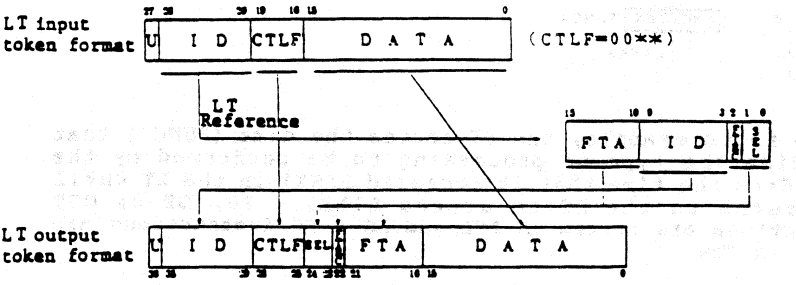
The contents of the LT are read when a token with CTLF=1000B arrives at the LT block. When this type of token arrives, the 16-bit contents of the LT location addressed by the ID are read. This data then becomes the data field for the new token output from the LT. This token is then automatically output from the μPD7281 after passing through FT and DM as NOP (no operation).



The value of this field is ignored.

(3) Execute

During a program execution, when a token that executes a PU, AG & FC, OUT or GE instruction* arrives at the LT, the contents of the LT address indicated by ID are read out and added to the token. The LT output token thus formed is sent to the FT.



* For details on instructions, see Chapter 5

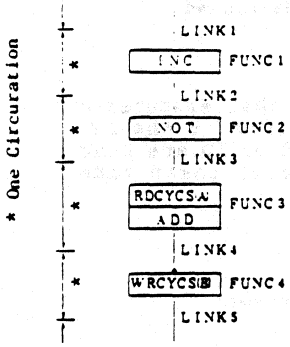
(4) Pass
Tokens whose CTLF specifies read or write of the FT pass through the LT as NOP. The contents of the LT address indicated by the ID of such a token is not read and the token proceeds to FT. However, as the bit lengths of the input and output tokens are different, the token is padded with null data and output.

2.3 Function Table (FT)

The FT is a program area that stores the instruction code to be executed during the circulation of the token through the pipeline ring (one pipeline circulation). Each FT address is made up of the following three fields: FTL, FTR and FTT. In terms of flowgraphs, the FT defines the part of the program corresponding to a node.

Figure 2-2
Function Table

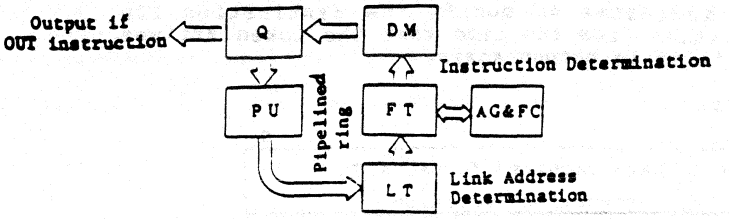
(a) Flowgraph



(b) Data stored in FT

	FTL	FTR	FTT
FUNC1:	INC		
FUNC2:	NOT		
FUNC3:	ADD	RDCYCS	RC
FUNC4:		WRCYCS	WC

In the above example, the FT stores the data (FUNC1) that specifies the type of processing to be performed by the token from the time that it accessed LINK1 in the LT until it returns to the LT to access LINK2. PU, GE or OUT instructions are stored in FTL and AG & FC instructions are stored in FTR.



2.3.1 FT field format

The FT is a 64-word x 40-bit memory configured in dynamic RAM. The internal fields are shown in the following figure.

FT field format

- The FTL field stores the Op Codes for the PU, GE or OUT instruction.
- The FTR field stores the Op Codes for the AG & FC instructions.
- The FTT stores the counters and registers used by the AG & FC instructions

(1) FTL field format
This field is significant for the PU, GE or OUT instruction processed at blocks subsequent to the FT. The meaning of this field is different for each instruction. Instruction group identification is performed by the SEL bits of the LT.

SEL

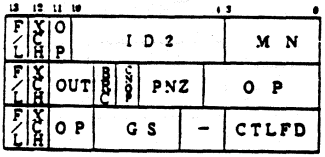
00 OUT instruction

01 PU instruction

10 GE instruction

11 AG & FC instruction

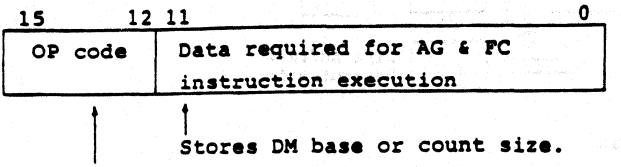
FTL field



(2) FTR field format
This field is significant to AG & FC instructions that

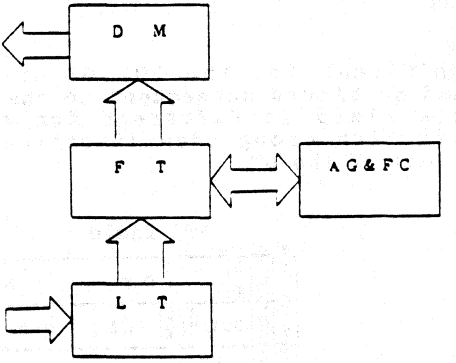
generate DM addresses or modify the destination IDs during circulation from the time that the token arrives at the FT until it is output again.

FTR field format



When a token that executes an AG & FC instruction arrives at the FT block, the AG & FC instruction stored in the FTR field is executed by the AG & FC block. The contents of the FTT field are referenced as the processing is performed. As a result, execution of an AG & FC instruction uses both the FTR and FTT fields.

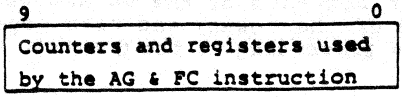
AG & FC instructions are executed during the circulation from when the token arrives at the FT until it is output.



(3) FTT field format

This field is used as a temporary field when an AG & FC instruction executes and stores the counters and registers required by the AG & FC instruction. The actual contents of this field vary depending on the type of AG & FC instruction executed.

FTT field format



2.4 Address Generator and Flow Controller (AG & FC)

AG & FC instructions execute either when the token arriving at the FT block indicates AG & FC instruction execution (SEL=11) or when the F/L bit of the FTL specified by FTA is '1'. AG & FC instructions complete execution within one pipeline cycle (200 ns at 10 MHz) from the time the token arrives at the FT until it is output to the DM block.

The AG & FC instructions can be roughly broken into the following two types:

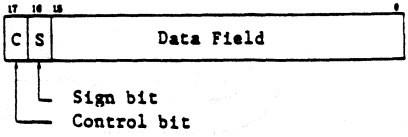
(1) DM address generate instructions
These instructions generate the DMA (data memory addresses) used to access the DM. Example instructions are RDCYC, WRCYC and RDIDX. These instructions calculate the DM addresses based on the value of FTR (DM base), the counter stored in FTT and the data field of the arriving token.

(2) Flow (ID) control
These instructions modify the destination ID of the arriving tokens. Example instructions are PICKUP, DIST and CONVO. This processing is performed by comparing the values of the FTR (count size) and that of the FTT (counter). The ID is then modified based on the results of this comparison.

2.5 Data Memory (DM)

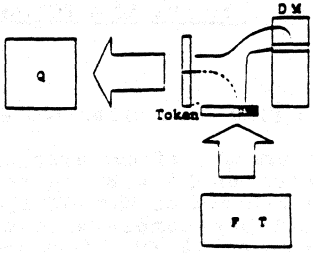
The DM is a 512-word x 18-bit dynamic RAM used for temporary storage of data. Each 18-bit word of the DM consists of a control bit (C), a sign bit (S) and a 16-bit data field.

DM field format



When a token arrives at the DM from the FT, it reads out the

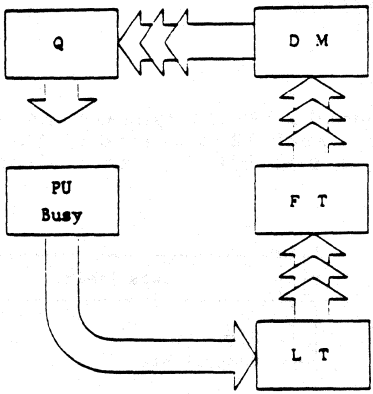
18-bit contents of the DM address (specified by the DMA* field of the token) and then appends this data to itself before proceeding to the Q block.



* The value of DMA is determined by the execution of the AG & FC instruction. If no AG & FC instruction is executed, the value of ID is used to access the DM.

2.6 Queue (Q)

Of the five blocks of the pipelined ring, only the PU, which performs the actual processing, has a variable processing time. Therefore, even if tokens arrive at the PU each pipeline cycle, if the PU is busy (because processing for the previously input token has not completed), the next token cannot be processed. The QUEUE, which acts as a buffer memory to temporarily store waiting tokens arriving from the DM, has therefore been provided. As the PU completes processing, the QUEUE sends tokens to the PU in first in, first out (FIFO) order.



The internal configuration of the queue is a 32-level data queue (DQ) and a 16-level generator queue (GQ).

2.6.1 Data Queue (DQ)

The DQ is a 32-word x 60-bit dynamic RAM used to temporarily store all arriving tokens except for those with SEL bits indicating a GE instruction. DQ is a 32-level FIFO memory.

2.6.2 Generator Queue (GQ)

The GQ is a 16-word x 60-bit dynamic RAM used to temporarily store all arriving GE instruction tokens. Because the GQ is limited to 16 levels, care must be exercised when designing the user program so that no more than 16 GE instruction tokens are generated at a time.

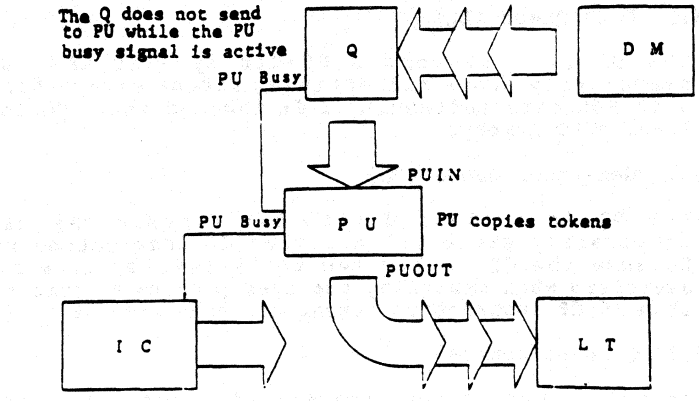
2.6.3 Q level control

Normally, when tokens are stored in both the GQ and DQ, token output alternates between the contents of the two queues. However, when more than eight levels of tokens are stored in the DQ, output of GQ tokens is halted and only DQ tokens are output to the PU or OQ. This function prevents the DQ from overflowing with tokens generated by the GE instruction.

A DQ overflow may occur if the internal processing speed is not fast enough to keep up with the input of tokens. In this case, the input restrict/prohibit mode may be used to prevent an overflow. The input restrict/prohibit mode is enforced when the number of levels in the DQ exceeds 24. (This mode is described in detail in a later section.)

2.7 Processing Unit (PU)

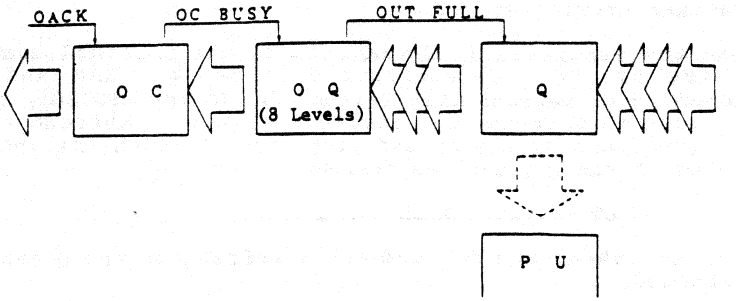
The PU functions to execute PU and GE instructions. Among the PU instructions are logical, arithmetic and shift instructions. The GE instructions include instructions that copy and generate tokens, set the control field (SETCTL instruction), etc. There are also PU instructions (copy instructions or instructions that output a binomial result) that process a single input token to output multiple tokens. These instructions output a token for each pipeline cycle with the result that the PU operation time varies with the number of output tokens. While the PU is operation, it sends a PU busy signal to the Q and IC to give priority to the processing under way.



The internal configuration of the PU consists of two pipelines. The input tokens pass through these two pipelines and are then output to the LT. This pipeline configuration enables tokens to be output from the PU at intervals of 200 ns (at 10 MHz).

2.8 Output Queue (OQ)

The OQ is an 8-word x 32-bit static RAM that functions as an 8-level FIFO memory to temporarily store tokens when the number of output tokens is greater that can actually be output. When there are no more output tokens in OC (OC non-busy), the tokens in the OQ are sent to the OC. When the OQ becomes full, it sends a OUTFULL signal to the Q and temporarily halts the output of tokens from the Q.



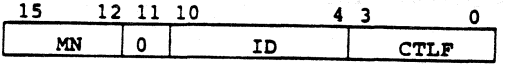
2.9 Output Controller (OC)

All tokens output from the uPD7281 (including pass tokens) are output via the OC. The output tokens are 32 bits long; the uPD7281, however, only has 16 output pins which requires that each token be partitioned into a low and high word. Control of this operation is performed by the OC.

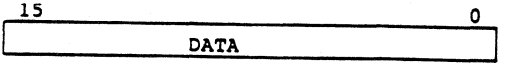
The following four types of tokens are output from the OC:
(1) Pass tokens from the IC
(2) OQ tokens (processing results)
(3) Error tokens from internal blocks (error or BREAK mode)
(4) Dump tokens (contents of the internal blocks)

The OC also controls priority among these four types of tokens. The order of priority is as follows:
In NORMAL mode: ERROR* > PASS > OQ
In BREAK mode: PASS = DUMP
DUMP tokens are output only when the processor is in BREAK mode and, for this reason, no other tokens are output. The format of output tokens is exactly the same as input tokens.

Token high word format



Token low word format



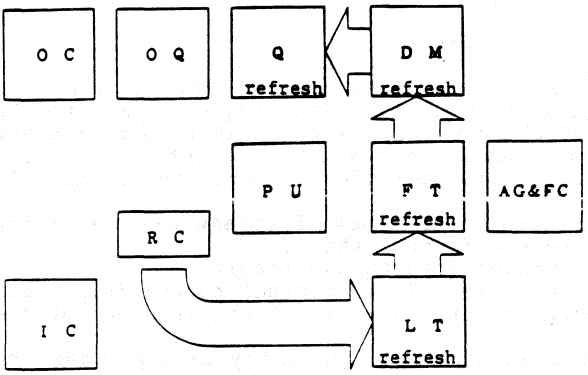
* The generation of ERROR data indicates the transition from NORMAL to BREAK mode. ERROR data is output only during the period immediately after this transition. For this reason, even if there is NORMAL mode data waiting to be output, the ERROR data will have priority.

2.10 Refresh Controller (RC)

With the exception of the OQ, all of the internal memories of the uPD7281 are configured in dynamic RAM and must therefore be periodically refreshed. In the uPD7281, tokens are used to perform the refresh operation. Refresh tokens are generated by the RC and propagated through the internal blocks of the pipeline as follows:

LT --> FT --> DM --> Q

Refresh tokens are deleted after refresh of the Q has been performed.

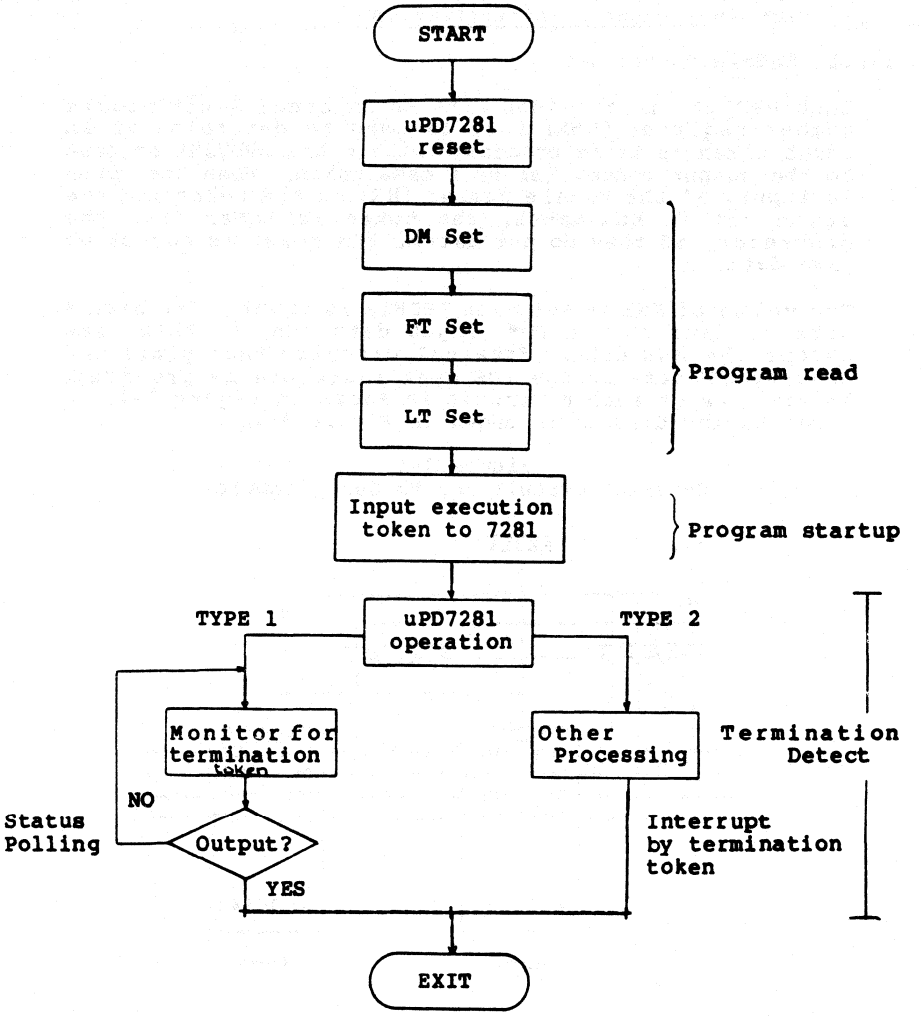


Because the refresh tokens are generated internally, when the same program has been executed a number of times, variations may appear in the order of processing. In other words, when the uPD7281 repeats a process, even if the same results are obtained, the order of the processing is not reproducible.

Chapter 3

μPD7281 Operating Procedure

The procedure shown in the following flowchart is used to operate the μPD7281. The process flow here is as seen from the host.



3.1 Reset Operation

Reset of the uPD7281 is performed by inputting a low level signal to the RESET/ pin. The main operations performed at this time are the following:

(1) Module number set
(2) Internal pipeline initialization
(3) IACK/ and OREQ/ initialization
(4) Internal mode initialization
(5) Internal counter initialization

3.1.1 Module number set

Each uPD7281 is provided with an internal 4-bit module number register (MNR) that is used to determine if an input token is to be processed within the uPD7281 or sent to the output controller as a pass token. When the token is input, if the module number (MN) of the token and the value set in MNR match, the token is taken into the processor; if they do not match, the token is output as pass data.

The value of MNR is set when RESET/ is input. The high 4 bits of data on the IDB (input data bus) at this time become the MNR data. External circuits that place the desired MN data on the IDB must therefore be provided. An example of such a circuit is shown in Figure 3-1. A reset timing diagram is shown in Figure 3-2.

Figure 3-1
External Circuit for MN Set (Example)

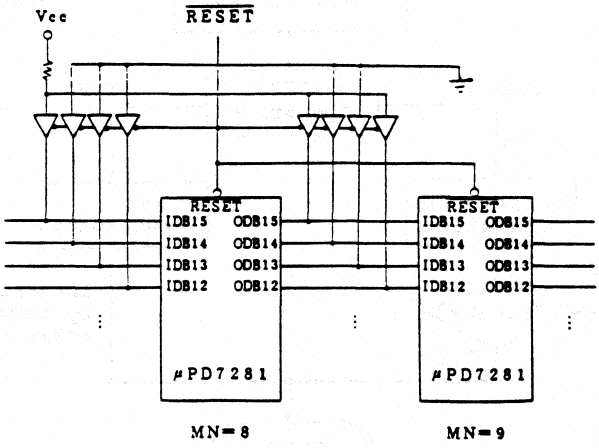
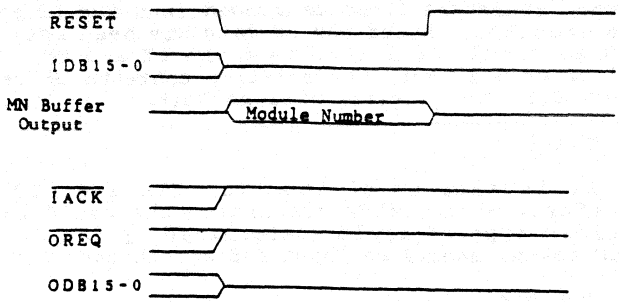


Figure 3-2
RESET/ Input Timing



3.1.2 Internal pipeline initialization

While the uPD7281 operates, tokens flow through the internal pipeline. However, when a RESET/ signal is asserted, all tokens in the pipeline are deleted.

3.1.3 IACK/ and OREQ/ initialization

The output of the uPD7281 consists of the IACK/ and OREQ/ signals and output bus ODB0-15. When RESET/ is input, IACK/ and OREQ/ become high level and ODB0-15 become high impedance.

3.1.4 Internal mode initialization

When RESET/ is input, any input restrict/prohibit mode or internal BREAK mode that has been set is released.

3.1.5 Internal counter initialization

When RESET/ is input, the refresh address counter and the refresh timing counter are reset to '0'. Internal refresh is not performed during RESET/ operation. Therefore, if the RESET/ signal is held at the active level too long, the contents of the internal RAM memories will be lost.

3.2 Program Load

Programs should be loaded into the uPD7281 internal blocks in the order of DM, then FT and then LT. (For details, refer to Chapter 4.)

3.2.1 DM set

In contrast to the LT and FT, there are no special tokens that can be used to set the DM. A program that sets the DM must therefore first be loaded into the LT and FT and then executed. After the DM data has been set, the user program should be loaded into the LT and FT. When the user program is set, the existing contents of the LT and FT (DM data set program) will be lost.

3.2.2 FT set

The FT is made up of the FTL, FTR and FTT fields. Therefore, data fields containing the set data for the specific type of field (determined by the CTLF of the input token) should be input and set in each field.

(1) FTL set
To set the FTL, input a token with CTLF=1110B to the uPD7281. The lower-order 14 bits of the data field of this token are used as the FTL set data. The FTL address to be set is specified by the ID of the input token. After the FTL data has been set, this token is internally deleted.

(2) FTR set
To set the FTR, input a token with CTLF=1101B to the uPD7281. The entire 16 bits of this token's data field are used as the FTR set data. The FTR address to be set is specified by the ID of the input token. After the FTR data has been set, this token is internally deleted.

(3) FTT set
To set the FTT, input a token with CTLF=1111B to the uPD7281. The lower-order 10 bits of the data field of this token are used as the FTT set data. The FTT address to be set is specified by the ID of the input token. After the FTT data has been set, this token is internally deleted.

3.2.3 LT set

To set the LT, input a token with CTLF=1100B to the uPD7281. The entire 16 bits of this token's data field are used as the LT set data. The LT address to be set is specified by the ID of the input token. After the LT data has been set, this token is internally deleted.

3.3 Starting the uPD7281 Program

The uPD7281 cannot be operated simply by resetting it and loading the user program. To execute the user program after

it has been loaded, an execution token must be input to the START section of the program (flowgraph arc marked START). When the program is started in this way, the input token executes the program and processing begins.

3.4 Terminating the uPD7281 Program

Operation of the uPD7281 during program execution can be one of two types: either the host is performing status polling or the host is performing other processing.

3.4.1 Status polling

In this case, the uPD7281 operates normally and the host constantly monitors the tokens output from the uPD7281 to determine if a termination token has been output. If a termination token* is output from the uPD7281, the host judges that the uPD7281 has completed processing. The host then moves to the next operation

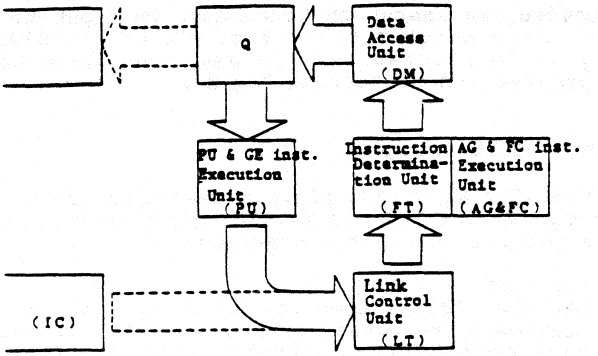
3.4.2 Other processing

In this case, the uPD7281 is performing operations completely independent of the host. However, by designing the system so that the termination token output by the uPD7281 will generate an interrupt to the host, the host can detect uPD7281 operation completion.

3.5 uPD7281 Token Transformations

As can be seen from the following figure, the internal processing of the uPD7281 is divided among the different blocks.

* The contents fo this token can be freely defined by the user program. For example: High-order token = 0000H, Low-order token = EEEEH



The LT can be thought of as the starting point for the token's circulation through the pipelined ring. The token passes through the blocks of the ring in the following order:

LT --> FT --> DM --> Q --> PU

The LT and FT determine the instructions to be exectued during the circulation of the token through the ring. The AG & FC block executes the specified AG & FC instruction simultaneously with the FT block's determination of the instruction Op Code. At the DM, the data necessary to the current pipeline cycle is read or written. Because there is some variation in the length of the processes performed by the PU, the Q is provided to act as a temporary buffer for tokens output in rapid succession from the DM.

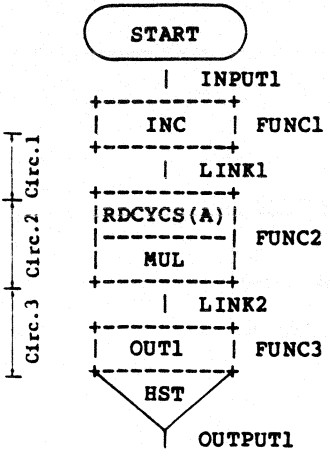
Of the instructions selected for execution by the LT and FT, the PU processes the PU and GE instructions. When a token has completed a circulation of the ring and has returned to the LT, it uses the ID defined during the previous cycle to access the LT. The process then repeats.

3.5.1 Basic internal data transitions

In this section, simple examples are given to explain the changes that a token (data) undergoes within the uPD7281. The operation actually performed here is the calculation of the following expression:

$$(X + 1) \times A$$

where X: input data
A: constant data already set in the DM



The flowgraph for this expression is shown at left. During the first calculation, data is input and the result of (X+1) is calculated; during the second circulation, data A is read from the data memory and multiplication is performed; in the third circulation, the result of the expression are output.

The two tables below present the operation at each block of the processing flow. (Detailed descriptions of each instruction are given in Chapter 5.)

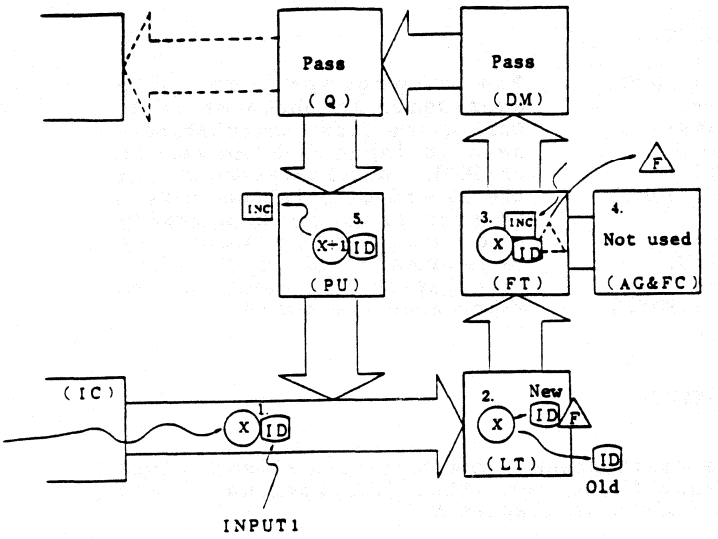
LT contents

	ID	Instruction	
INPUT1	LINK1	FUNC1	
LINK1	LINK2	FUNC2	
LINK2	OUTPUT1	FUNC3	

FT contents

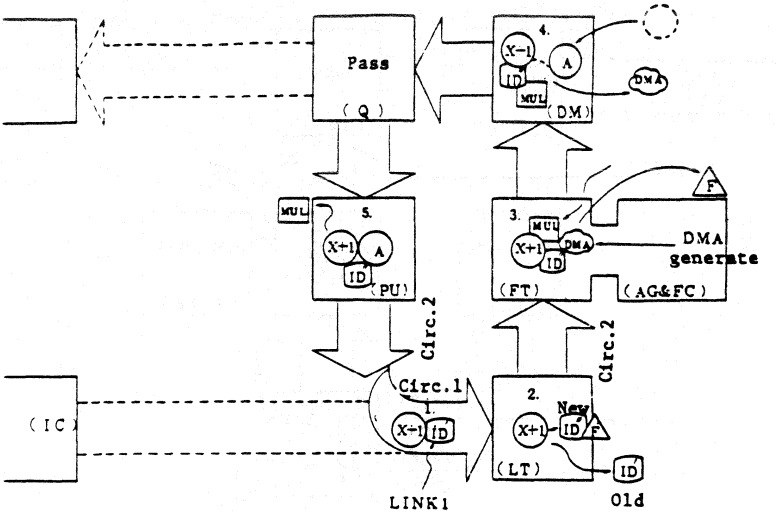
	FTL	FTR	FTT
FUNC1	INC		
FUNC2	MUL	RDCYCS(A)	RC
FUNC3	OUT1		

Token transitions during circulation 1



(1) A token with data X and the ID for arc INPUT1 passes through the IC.
(2) When this token arrives at the LT, the current ID of the token is used to access the LT. The FT instruction address data F(FUNC1) for this circulation is read, along with the new ID data. These data are incorporated into the token and the token then proceeds to the FT.
(3) When the token arrives at the FT, the FT address data F(FUNC1) is used to access the FT to read the instruction code to be executed during the current circulation.
(4) Because the instruction (INC) to be executed is an independent PU instruction that does not require any AG & FC or DM processing, the token passes through these blocks as NOP. If there are no tokens remaining in the Q from the previous processing, the token passes through the Q and proceeds directly to the PU. If there are tokens in the Q, the token is stacked on the Q to await PU processing.
(5) When the token arrives at the PU, the PU decodes and executes the instruction code carried by the token. The PU processing ends and this completes one circulation through the pipeline. The new ID' (LINK1) of the token output from the PU is then used to access the LT and circulation 2 begins.

Token transitions during circulation 2



(1) The token with the data (X+1) generated during circulation 1 arrives at the LT.

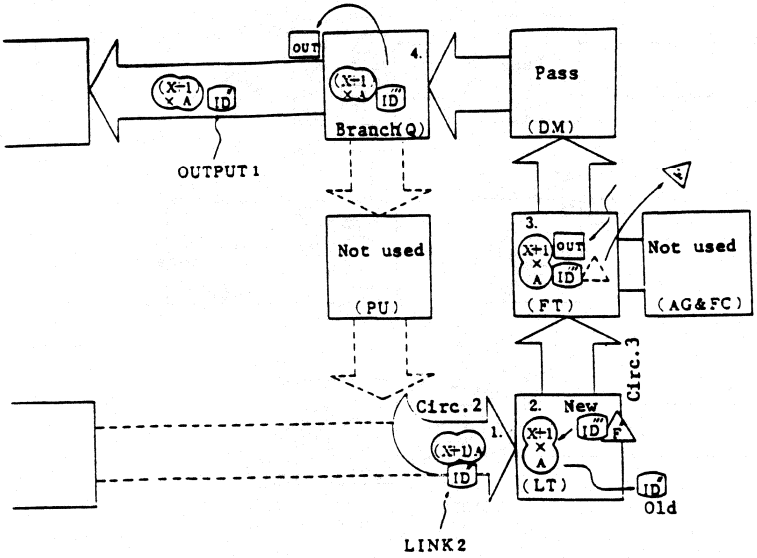
(2) When the token arrives at the LT, the ID' (LINK1) of the token is used to access the LT. The new ID'' (LINK2) and instruction address data F(FUNC2) are read from the LT. These data are incorporated into the token and the token then proceeds to the FT.

(3) When the token arrives at the FT, the FT address data F(FUNC2) is used to access the FT to read the instruction code (MUL) to be executed during circulation 2. The instruction data includes AG & FC instruction RDCYCS (to read constant data A from the DM). This instruction therefore executes during the current pipeline circulation and generates the DM address (DMA) where constant A is stored.

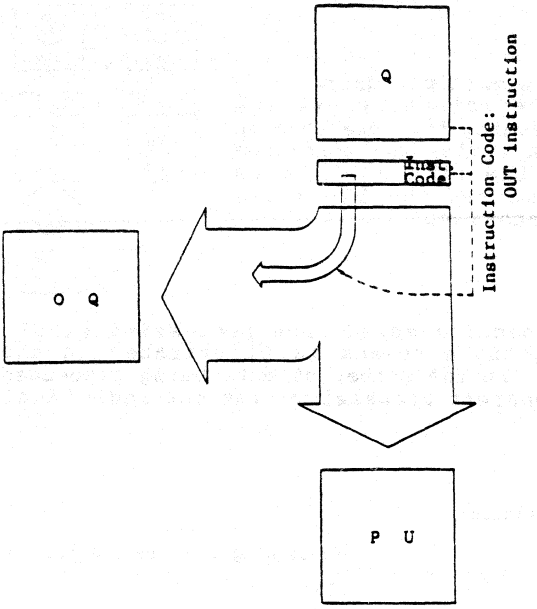
(4) The token, which now contains data (X+1), the instruction data (MUL), the new ID'', and DM address leaves the FT and arrives at the DM. The contents of the DM (constant A) are read from the indicated DM address and the token proceeds to the next block. At this point the token contains these four data: ID'', data (X+1), data (A) and instruction code (MUL).

(5) When the token with these four data arrives at the PU, the PU decodes and executes the instruction code. When the PU processing ends, this completes circulation 2. The ID'' (LINK2) of the token output from the PU is then used to access the LT and circulation 3 begins.

Token transitions during circulation 3



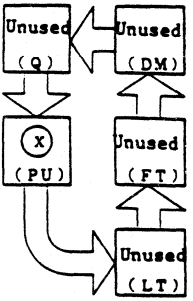
(1) The token with the data (X+1)·A generated during circulation 2 arrives at LT.
(2) When the token arrives at the LT block, the ID'' (LINK2) of the token is used to access the LT. The new ID''' (OUTPUT1) and instruction address data F(FUNC3) is read from the LT. This data is incorporated into the token and the token then proceeds to the FT.
(3) When the token arrives at the FT, the FT address data F(FUNC3) is used to access the FT to read the instruction code (OUT1) to be executed during circulation 3. The token then proceeds to the DM.
(4) When this token arrives at Q, the Q evaluates the instruction code (in this case, the OUT instruction) and sends the token to the appropriate block, either PU or OQ (in this case, OQ). The destination (PU or OQ) of the token is determined when it is read from the queue memory.



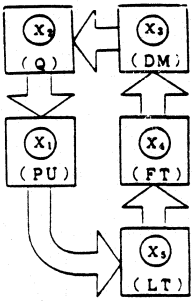
3.5.2 Effective pipeline processing

The example given on the preceding pages was probably sufficient to convey the basic concepts of data flow in the uPD7281. As mentioned earlier, the uPD7281 consists of five function blocks organized in a pipelined ring configuration and the benefits of this configuration are realized when performing a series of operations involving many separate processes. To illustrate this point, let's compare the processing speed when performing multiple operations (e.g., for data X_1 - X_5) against that when performing only a single operation (X).

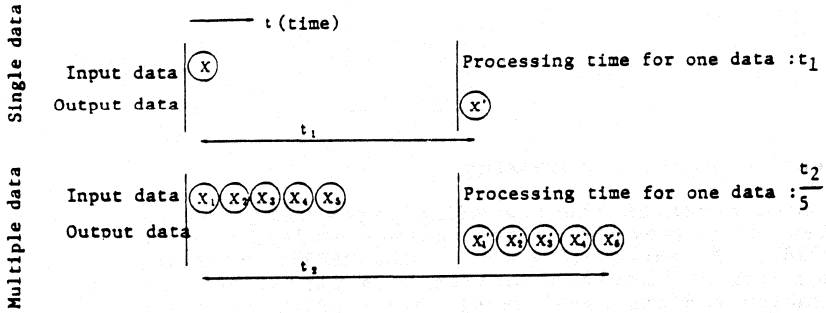
(1) When processing a single data:
In this case, there is only one token in the pipelined ring at a given moment. Therfore, of the five blocks of the pipeline, only one is operational at a time. The figure at the right shows the condition of the pipeline ring when the input token is being processed by the PU.



(2) **When processing multiple data:**
When multiple data X_1 to X_5 are processed, assuming that one data is input for each cycle, the condition of the pipeline ring will be as shown at right and the block use rate will be enhanced.



The apparent processing speed when processing single tokens versus multiple tokens is illustrated in the following figure. As the number of data being processed increases, the apparent processing time for individual data decreases.



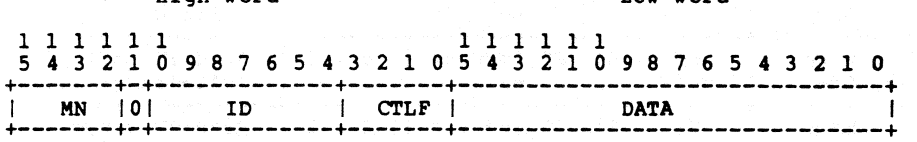
Chapter 4

Input/Output Tokens

Each input or output token consists of 32 bits. However, since the width of both the input and the output data bus is 16 bits, all tokens must be divided into two 16-bit words for input or output. The first 16 bits (which include the address and control data) are referred to as the high word, and the next 16 bits (data) are referred to as the low word.

The general format of input tokens is the same as that of output tokens (see Fig. 4-1). The detailed format of each type of input and output token is shown in Tables 4-1 and 4-2.

Figure 4-1
Input/Output Token Format



MN:	Module Number	Selects the uPD7281
ID:	Identifier	Address for LT access
CTLF:	Control Field	Data that determines the operation to be performed

Table 4-1
Input Token Format

Input Token	Hi/Lo Word	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Remarks
SETLT	High	MN				0	LT Address							1	1	0	0	Set LT
	Low	Data to be set in FT																
SETFTR	High	MN				0	FT Address							1	1	0	1	Set FT right field
	Low	Data to be set in FTR																
SETFTL	High	MN				0	FT Address							1	1	1	0	Set FT left field
	Low	Data to be set in FTL																
SETFTT	High	MN				0	FT Address							1	1	1	1	Set FT temporary field
	Low	Data to be set in FTT																
RDLT	High	MN				0	LT Address							1	0	0	0	Read LT
	Low	--																
RDFTR	High	MN				0	FT Address							1	0	0	1	Read FT right field
	Low	--																
RDFTL	High	MN				0	FT Address							1	0	1	0	Read FT left field
	Low	--																
RDFTT	High	MN				0	FT Address							1	0	1	1	Read FT temporary field
	Low	--																
CRESET	High	MN				0	--							0	1	0	0	Command reset
	Low	--																
SETMD	High	MN				0	--							0	1	0	1	Set operation mode
	Low	0	0	Refresh Count										I	I	R	SD	
SETBRK	High	MN				0	ID							0	1	1	0	Set break condition
	Low	M	Count															
DUMP	High	MN				0	D	U	M	P				0	1	1	1	Dump
	Low	--																
CBRK	High	0	0	0	0	0	--							0	1	0	0	Command break
	Low	--																
VAN	High	1	1	1	1	0	--							-				Vanish Data
	Low	--																
PASS	High	MN*				0	--							-				Pass Data
	Low	--																
EXEC	High	MN				0	ID							0	0	C	S	Normal execution data
	Low	Data																

*: When MN is not the current module number

Table 4-2
Output Token Format

Input Token	Hi/Lo Word	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	Remarks
LTRDD	High	0 0 0 0 | 0 | LT Address | 1 0 0 0	LT read data
	Low	Data read from LT	
FTRRDD	High	0 0 0 0 | 0 | FT Address | 1 0 0 1	FT right fld. read data
	Low	Data read from FTR	
FTLRDD	High	0 0 0 0 | 0 | FT Address | 1 0 1 0	FT left field read data
	Low	Data read from FTL	
FTTRDD	High	0 0 0 0 | 0 | FT Address | 1 0 1 1	FT temporary fld read data
	Low	Data read from FTT	
PASSD	High	MN | 0 | ID | CTLFD	Pass data
	Low	Data	
ERR	High	0 0 0 0 | 0 | 0 0 0 0 0 0 0 | 0 1 0 0	Error data
	Low	MN MODE 0 0 0 Status	
DUMPD	High	0 0 0 0 | 0 | 0 0 0 0 Dump | 0 1 1 1	Dumped data
	Low	DUMP DATA	
OUTD	High	MN | 0 | ID | 0 0 C S	Output data
	Low	Data	

4.1 Execution Tokens

Of the tokens whose module number (MN) is the same as that assigned to the current uPD7281, those with '00' in the high order 2 bits of the CTRL field are processed within the chip according to the program.

Tokens output by execution of an OUT instruction are accepted and processed by the uPD7281 whose module number corresponds to the MN field of the token.

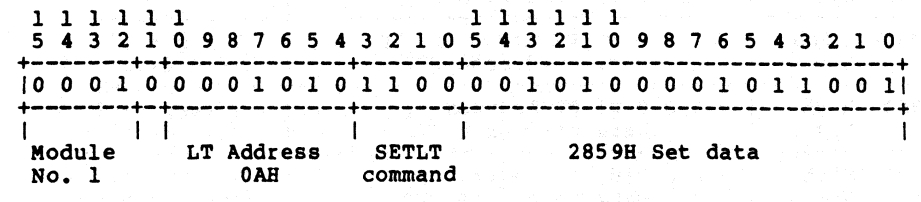
Tokens that are objects for execution within the uPD7281 are called execution (EXEC) tokens and tokens output by execution of an OUT instruction are called OUTD tokens.

4.2 Object Program Set

Because the internal program areas of the uPD7281 are configured in RAM, the program must be set in this area before processing is performed. The SETLT, SETFTR, SETFTL

and SETFTT input tokens shown in Figure 4-1 specify arbitrary data to be set in any address of the LT, FTR, FTL or FTT respectively.

For example, the SETLT token shown below sets data 2859H in address 0AH of the LT of the uPD7281 chip with module number 1. (For the meaning of this data, refer to the descriptions of the instruction set.)



This token will be input as a high word (10ACH) and a low word (2859H). Input tokens to set data (object programs) in the FTR, FTL or FTT can also be created in the same manner.

The range of LT address values that can be specified by a SETLT token is 00 to 7FH (0 to 127). In other words, all 7 bits of the ID field are valid.

However, in the case of the SETFTR, SETFTL and SETFTT tokens, only the lower 6 bits are valid and all higher bits are ignored. For example, if data 7FH is set in the ID field of a SETFTR token, its effective value will be 3FH. As for the data specified by the low word of the token, only the number of bits corresponding to the bit width of the memory where the data is to be set will be valid; all higher order bits are ignored. By using these tokens, object programs can be set in the LT, FTR, FTL and FTT. Input tokens cannot, however, be used to directly set data in the data memory (DM). To do this, a program that writes to the DM must be set in the LT and FT and then executed. The token used to execute this program (EXEC type execution token, so called because it starts the operation of the uPD7281) should contain the data to be written to the DM. Figure 4-2 shows a program that sets data in the DM. Refer to later chapters for flowgraphs and details of each instruction.

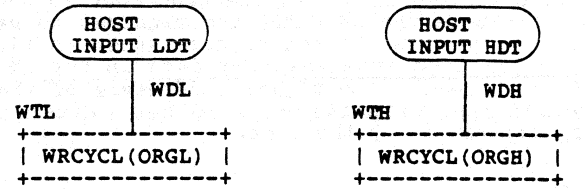
The four input tokens described above (SETLT, SETFTR, SETFTL, SETFTT) are used to set this program in the respective fields.

The procedure for programming the uPD7281 is therefore to set a program in the LT and FT that will write data to the DM and then to execute this program. The LT and FT portions of the uPD7281 program are then set in the LT and FT fields.

The input tokens used to set object programs are internally

deleted after the data have been set and they therefore produce no corresponding output tokens. Execution of the program shown in Figure 4-2 will produce no output tokens.

Figure 4-2
DM Set Program Example (MN:8)




```
EQUATE    HOST = 0;
MODULE    DMWT = 8;
INPUT     WDL, WDH;
LINK           = WTL(WDL)
LINK           = WTH(WDH)
FUNCTION  WTL = WRCYCL (ORGL, 256);
FUNCTION  WTH = WRCYCL (ORGH, 256);
MEMORY    ORGL = AREA (256) AT 0;
MEMORY    ORGH = AREA (256) AT 256;
END;
```

When the range of DM addresses to be set is 00 to FFH (0 to 255), the desired DM address should be set in the WTL address of the FTT, the write data set in the EXEC token, and the WDL address input as the ID field. When the range of DM addresses is 100H to 1FFH (256 to 511), the desired DM address minus 256 should be set in the WTH address of FTT, and the EXEC token should be input with the WDH address set in the ID field.

To set a single DM data, it is therefore necessary to input two tokens. However, when data is set in contiguous addresses, FTT need not be set after the first EXEC token has been input (with the exception of cases where the data set straddles the two memory areas (0 to 255 and 256 to 511).

The object programs output by the uPD7281 assembler use this same method to set programs in the uPD7281.

When the object program for DM set shown in Figure 4-2 is assembled, a file containing the object program, EXEC tokens (with the DM data), and tokens to set the LT and FT are all output. All that is required, then, to set your application program in the uPD7281 is to input in order the tokens that have been written in the file output by the assembler.

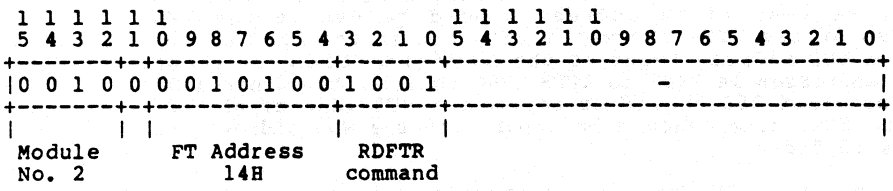
Note the following differences between the method shown in

Figure 4-2 and that actually used by the assembler to set data in the DM. In the method shown in Figure 4-2, the address that specifies the WRCYCL base address is fixed and the offset changes to generate the address. In the method used by the assembler, the offset is fixed and the base address in changed to generate addresses. With this method, two tokens (a SETFTR token to change the base address and an EXEC token that includes the DM data) are required to set a single DM data. In contrast to the program shown in Figure 4-2, however, the instructions that set data in the LT and FT require only one token. The highest addresses of the LT and FT are 7FH and 3FH, respectively. Other values than these, however, can be set by the user.

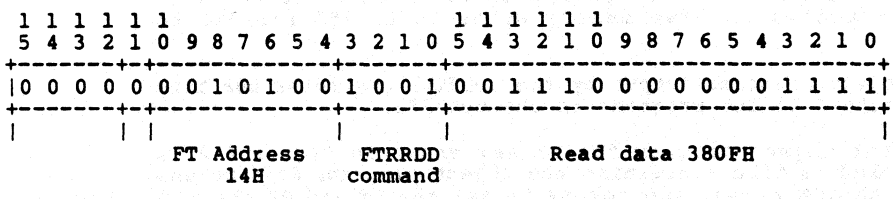
4.3 Object Program Read

The RDLT, RDFTR, RDFTL and RDFTT input tokens read the contents of arbitrary addresses in the LT, FTR, FTL and FTT respectively. The data read from these blocks is output from the uPD7281 as the LTRDD, FTRRDD, FTLRDD and FTTRDD tokens, respectively.

For example, assume that data 380FH has been set in FTR address 14H of a uPD7281 with module number 2. (For the meaning of this data, refer to the descriptions of the instruction set.) The contents of this address are read by inputting the RDFTR input token shown below.



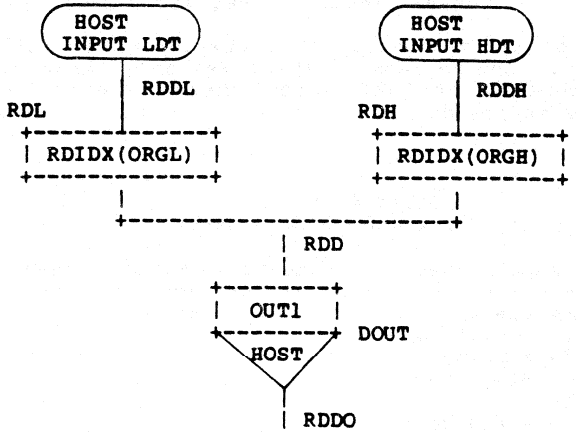
This will cause the following token to be output by the uPD7281:



Note that, regardless of the module number of the uPD7281 from which the data is read, the value of the MN field of the output token is always 0. The output token is therefore always received by the chip assigned module number 0 (usually the host) and this must be taken into consideration

when configuring a multiprocessor system. As with the object program set operations described in the preceding section, there are no tokens that can directly read the contents of the DM. For this reason, a program that reads the contents of the DM must be set in the LT and FT and an execution token input to execute the program. Figure 4-3 shows an example of a DM read program.

Figure 4-3
DM read program example (MN:8)




```
EQUATE     HOST = 0;
MODULE     DMRD = 8;
INPUT      RDDL,RDDH;
OUTPUT     RDDO;
LINK       RDDO = DOUT (RDD);
LINK       RDD = RDL (RDDL);
LINK       RDD = RDH (RDDH);
FUNCTION   RDL = RDIDX (ORGL, 256);
FUNCTION   RDH = RDIDX (ORGH, 256);
FUNCTION   DOUT = OUT1 (HOST);
MEMORY     ORGL = AREA (256) AT 0;
MEMORY     ORGH = AREA (256) AT 256;
END;
```

When the range of DM addresses is 00 to FFH (0 to 255), the address should be set in the low word of the EXEC token and the RDDL address of the LT should be input as the ID field 00 to FFH (0 to 255). When the range of DM address is 100H to 1FFH (256 to 511), the address-256 should be set in the low word and an EXEC token whose ID field contains the RDDH address of the LT should be input. An OUTD token is output

for each EXEC token input. This program requires one token to read one DM data.

4.4 Command Reset

Two methods are available to initialize the uPD7281. One is an external (hardware) reset in which a RESET/ signal is applied to the RESET/ input. This method is used to initialized the uPD7281 immediately after power is applied. The other method, command (software) reset, is performed by inputting a CRESET* token and is used mainly for resetting the chip prior to program execution (see Table 4-3).

Table 4-3 compares the operation of the two reset methods. As can be seen, command reset initializes only those blocks related to control of token flow where external reset intializes the entire device.

Table 4-3
External Reset and Command Reset

External (hardware) reset	Command (software) reset
- synchronizes the clock with the reset signal and sets the module number	--
- Clears the IACK/ and OREQ/ signals	--
- Initializes the registers and counters used for refresh	--
- Sets the mode to NORMAL, DATA: no input restrictions	- Sets the mode to NORMAL
- Sets all pipelines to "non-use" status	- Sets all pipelines to "non-use" status
- Initializes the queue	- Initializes the queue
- Aborts instruction execution in the PU and clears the internal registers	- Aborts instruction execution in the PU and clears the internal registers

* If a SETCTL instruction is used to generate a CRESET token within the uPD7281 pipelined ring, that token is deleted at the LT block and reset operation is not performed. In the same manner, if SETMD, SETBRK or DUMP tokens are generated internally, they are deleted at the LT block and no operation is performed.

4.5 Internal Mode Transitions

During operation, the uPD7281 is always in one of the following modes:

- NORMAL
- TEST
- BREAK

In addition, and independent of the operating mode, this mode can also be set:

- Input restriction mode

4.5.1 NORMAL mode

This is the mode after the RESET/ signal or the CRESET token has been input. In this mode the uPD7281 performs normal operation.

4.5.2 TEST mode

The TEST mode is used to facilitate program debug. When a SETBRK token (see Table 4-1) is input to a uPD7281 when in NORMAL mode, a count is set in the counter that specifies the interval from the point in time specified by the low word of the SETBRK token until the BREAK mode is entered. This count functions as a break condition. Once this data has been set, the uPD7281 enters the TEST mode.

These two types of operation are available for the TEST mode. Selection between the two is made by the most signification bit of the low word of the SETBRK token, the M bit.

M=0 BREAK mode is entered when 'count value+1' number of valid tokens arrives at the LT address specified by the ID field of the SETBRK token.

M=1 BREAK mode is entered when program execution has progressed 'count value+1' pipeline cycles after the first token arrives at the LT address specified by the ID field of the SETBRK token.

Operation in the TEST mode is identical to that performed in the NORMAL mode with the sole difference that the break conditions are tested. New break conditions can be set by inputting a second SETBRK token when the processor is in the TEST mode.

4.5.3 BREAK mode

Execution enters the BREAK mode when any one of these three conditions is met when the uPD7281 is in either the NORMAL or TEST mode:

(1) When an overflow occurs in the DM queue, data queue or generator queue
(2) When CBRK token (see Table 4-1) in input
(3) When the break condition is met (TEST mode only)

When one of the above conditions is fulfilled, execution of the current pipeline cycle operation completes. The tokens latched in the LT and QUEUE input latches are saved for later output by the DUMP command. The pipeline is disabled from the next pipeline cycle and the processor enters the BREAK mode.

Once a token is saved, it can't be returned to the former state. However, the state of the pipeline at the time break mode was entered can be confirmed by a DUMP command.

When the processor enters the BREAK mode as a result of (1) or (3) above, the uPD7281 outputs an ERR token (see Table 4-2).

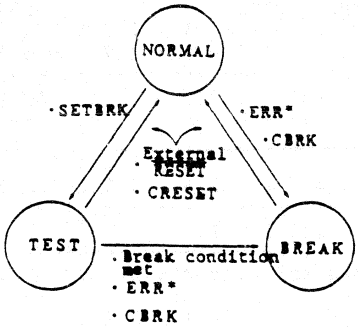
When a CBRK token is input to the uPD7281, causing the BREAK mode to be entered, the uPD7281 outputs this token as a PASSD token. As can be seen in Tables 4-1 and 4-2, the CBRK token includes an ERR token (the CTLF field is the same). Therefore, when multiple uPD7281 processors are connected together in cascade, if any one of the uPD7281s outputs an ERR token, all subsequent uPD7281s will enter the BREAK mode. The host processor can cuase all of the uPD7281s to enter the BREAK mode simply by inputting a CBRK token to the first processor of the cascade.

When the uPD7281 enters the BREAK mode, each pipeline within the processor enters the non-use mode and normal operation ceases. The only operations performed in this mode are:

- To pass all tokens (PASS and CBRK). CBRK tokens are passed with no processing performed.
- To input DUMP and CRESET tokens bearing the module number of the current processor. All other tokens are deleted.

When a CRESET token is input, the uPD7281 leaves the BREAK mode and enters the NORMAL mode. At the same time, command reset is performed as specified by the CRESET toekn. Transitions between the NORMAL, TEST and BREAK modes are shown in Figure 4-5.

Figure 4-5
Mode Transitions



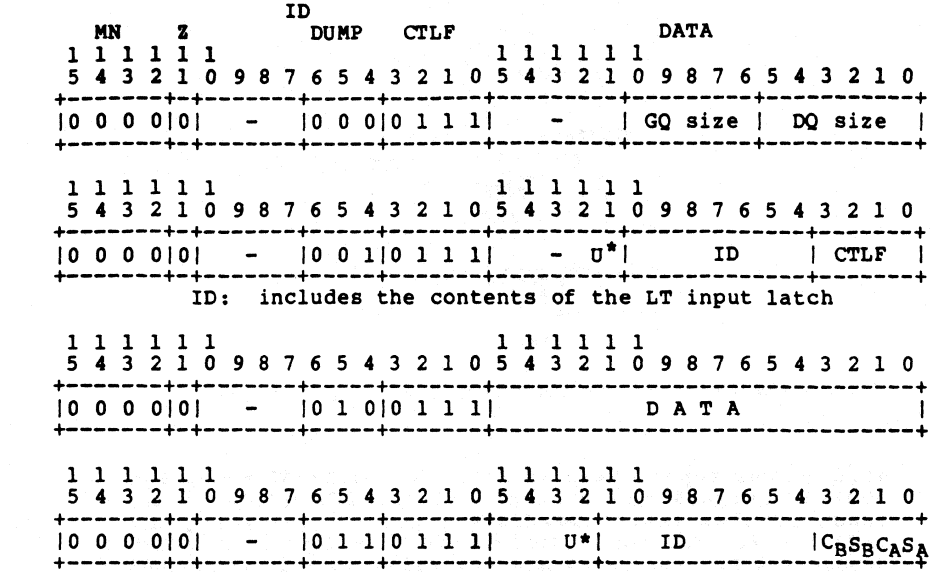
* When the DM, data, or generator queue has overflown.

When a DUMP input token is input in the BREAK mode, the uPD7281 outputs the contents of the LT, QUEUE, etc., input latches saved immediately prior to entering the BREAK mode. The token containing this data is a DUMPD output token. (DUMPD tokens are valid only in the BREAK mode and are deleted if input in the NORMAL or TEST mode.) Figures 4-6 and 4-7 show the format of the DUMP and DUMPD tokens.

Figure 4-6
DUMP Token (Input) Format

	High word															Low word															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	MN				0		-				DUMP						0	1	1	1												

Figure 4-7
DUMPD Output Token Format



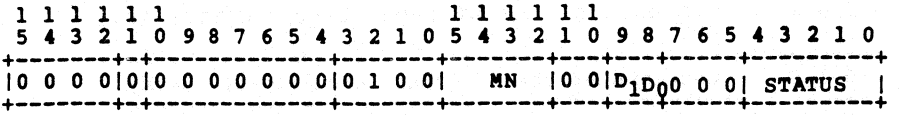
* ID: ID of the FT token when it entered the LT
UNUSE bit (bit indicates whether LT token is valid or invalid) U=0 : Valid U=1 : Invalid

DUMP	Dump specification data (see Figure 4-7)
0 0 0	Size of DQ and GQ being used
0 0 1	Contents of the LT input latch (UNUSE, ID, CTLF)
0 1 0	Contents of the LT input latch (16-bit data)
0 1 1	Contents of the QUEUE input latch (UNUSE, ID, SEL, C_A, C_B, S_A, S_B)
1 0 0	Contents of the QUEUE input latch (lower 12 bits of FTL)
1 0 1	Contents of the QUEUE input latch ($DATA_A$)
1 1 0	Contents of the QUEUE input latch ($DATA_B$)
1 1 1	ID (at the LT block) of the token that accessed the FT* immediately prior to entering the BREAK mode

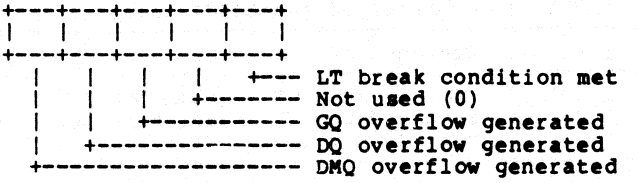
By analyzing the contents of the output ERR token, the cause for the uPD7281 to enter the BREAK mode can be determined. Figure 4-8 shows the ERR token format

* Inputting a DUMP token whose DUMP field is 111 returns the LT location that was accessed by the token that caused the DM queue error, causing the BREAK mode to be entered. This token is used to improve the efficiency of program debug.

Figure 4-8
ERR Token Output Format



MN : Module Number
D_1D_0 : Input restriction setting
Each bit of the STATUS field indicates one of the following causes for the BREAK mode when '1'.

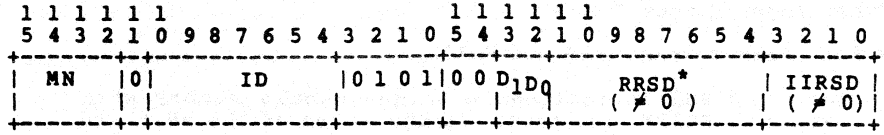


DMQ overflows are generated when tokens are queued in the DM for the purpose of performing two-operand operations and the tokens cannot be paired and processed. As a result, to many of one kind of token (FTRC=1 or FTRC=0) are waiting on the queue and the specified queue size is exceeded. GQ and DQ overflows are generated when GE instruction tokens are concentrated in the QUEUE and execution of the GE instructions causes the number of tokens to increase too rapidly.

4.5.4 Input Restriction Mode

The SETDM token (see Table 4-1) is used to set the input restriction mode for each uPD7281. The input restriction mode is used when multiple uPD7281s are connected in a cascade and helps to prevent overflow of the internal queues of the later-stage uPD7281s caused by excessive concentration of input tokens.

The following three types of input restriction mode operation are available:



RRSD : Refresh Register Set Data

IIRSD : Input Inhibit Register Set Data

D_1 D_0 : D_1=D_0=1 use prohibited

Type 1 : Input prohibited D_1D_0=01
Prohibits input to the LT while 24 or more levels of the data queue are being used. PASS tokens are passed to the output bus.

Type 2 : Input restricted D_1D_0=10
Restricts input to the LT while 24 or more levels of the data queue are being used. This restriction of input is performed by means of two timers: the 3-bit base counter and 4-bit input inhibit counter. The base counter is incremented (+1) for each pipeline cycle and the input inhibit counter (IIC) is incremented (+1) every eight counts. The contents of the IIRSD field of the SETMD token are set in the input inhibit register. Input inhibit is temporarily released and a token input to the LT when this value and the value of IIC match. If, for example, IIRSD = 4, input inhibit will be released every 32 pipeline cycles.

Type 3 : Input restriction released D_1D_0=00
No input restriction. This mode is set when an external RESET/ signal is input.

*: RRSD is used to generate the refresh cycle for the internal RAMs of the uPD7281. The refresh cycle is generated in much the same manner as the input inhibit cycle using a base and a refresh counter. The refresh cycle therefore equals RRSD x 8 pipeline cycles. After an external RESET/ has been input, if no SETMD token is input, the refresh register will be set to 10 (0AH) and refresh of the internal RAMs of the uPD7281 performed every 80 pipeline cycles. When there is no particular need to change the timing of the refresh cycle, set data 0AH in RRSD.

4.6 Other Input/Output Tokens

4.6.1 Pass Tokens (PASS and PASSD)

Each uPD7281 is assigned a unique module number when external RESET/ is input. If the value of the MN field of the input token does not match this module number, the input is evaluated as PASS and output as a PASSD token without any processing.

4.6.2 Vanish Tokens (VAN)

An input token whose module number is 15 (1111B) is evaluated as VAN and is deleted (vanishes) regardless of the contents of remaining bits of the token. This token is effective for vanishing useless tokens during program debugging. It is prohibited to assign a module number 15 to any module.

Chapter 5

Instructions

The instructions of the uPD7281 can be broadly divided into four types as shown in Table 5-1. Specification of the instruction group is performed by the SEL bits of the LT (Link Table).

Table 5-1

Instruction Types

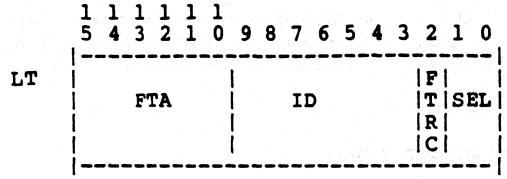
SEL bits of LT	Instruction Type	Function
1 1	AG & FC	Executes the instruction indicated by the FT right field (FTR) while monitoring the FT temporary field (FTT). Instructions of this group perform such operations as generating the DM address and modifying the ID (changing the next LT address to be referenced). The codes for these instructions are stored in the FTR field.
0 1	PU	These instructions use the PU to perform arithmetic and logical operations. The codes for these instructions are stored in the FT left field (FTL).
1 0	GE	For each token input to the PU, these instructions generate multiple tokens and send them to the LT. These instructions not only copy tokens but also perform increment and decrement operations for the data fields of the copied tokens. The codes for these instructions are stored in the FTL field.
0 0	OUT	These instructions output the data that has been processed within the uPD7281. The codes for these instructions are stored in the FTL field.

Note that the AG & FC instruction can be used in combination with the other three types of instructions. When used in this way, the instruction code of the PU, GE or OUT instruction takes precedence in the SEL field of the LT. The result is a combination instruction and the F/L bit of the FTL field is set to 1.

In the instruction descriptions that follow, flowgraphs and assembler lists are used. Consult Appendixes A and B as necessary for a more complete explanation. Appendix A explains

flowgraphs, and Appendix B is on program development support tools. For details of the assembler, refer to the uPD7281 Assembler Manual, available separately.

LT field format



FTA (Function Table Address) 6 bits
This is the address used to access the FT.

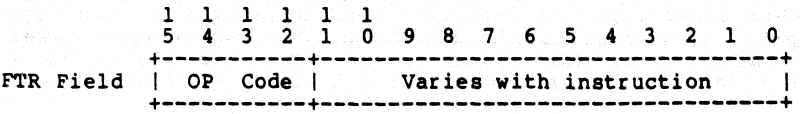
ID (Identifier) 7 bits
This data becomes the new ID data, that is, the ID of the token after it passes the LT.

FTRC (FT right field control) 1 bit
When two types of data are sent to the FT or AG & FC block, this bit is used to distinquish between them.

SEL (Selection) 2 bits
This field determines the processing to be performed to the data after the token accesses the LT.

5.1 AG & FC Instructions

When SEL=11 in the LT location accessed by the token, the operation specified by the FT right field (FTR) will be performed. There are a 16 different AG & FC instructions that can be specified by the Op Code stored in FTR. These include instructions that read the DM, queue binomial data and perform flow control. Because the operations specified by these tokens terminate at the AG & FC or DM block, they either pass through the PU as NOP or are deleted at the FT or DM block. The instruction to be executed is determined by the OP Code (higher 4 bits) of the FTR field.



The AG & FC instructions can be divided into those that

generate addresses and those that control token flow. There is also a third type of instruction that combines these two functions.

These six control instructions generate DM addresses or determine the type of read/write operation to be performed: RDCYCS, RDCYCL, WRCYCS, WRCYCL, RDWR, RDIDX.

The flow control instructions rewrite the ID field of the tokens to be sent to the FT block. In this way they correspond to the instructions of a conventional processor that control looping and related functions. There are nine such instructions: PICKUP, CONT, CUT, DIVCYC, DIV, DIST, CONVO, SAVE, CNTGE.

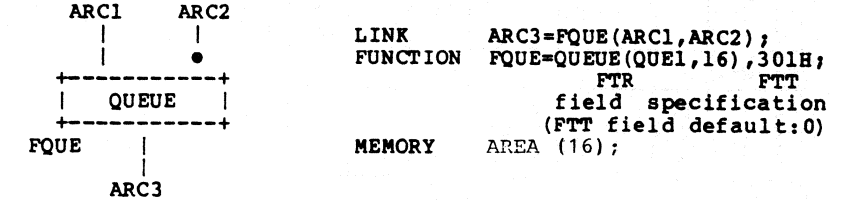
The QUEUE instruction combines the functions of the above two types of instructions. This instruction controls the queuing of two-operand data (in two different tokens).

Table 5-2 lists the AG & FC instruction. For details, refer to the description for each instruction.

- AG & FC Instructions Assembler description

Because the AG & FC instructions use the FTR and FTT fields, the values for these fields must be set by the user. When the user inputs a constant value, that data is automatically written to the FTT field.

Example:



To specify the FTR field, the constants required by the AG & FC instructions are described within parenthesis. In this example, QUE1 is the DM base address and 16 indicates the queue size. Normally, the value of the FTT field is 0. In this case, however, an initial value of 301H is set in the FTT. This value represents the state of the FTT field when one token has arrived from ARC2.

Table 5-2 AG & FC Instructions

Mnemonic	FTR 15 14 13 12	FTR 11 10 9 8 7 6 5 4	FTR 3 2 1 0	FTT 9	FTT 8	FTT 7 6 5 4	FTT 3 2 1 0	FTRC	Operation
QUEUE	0 0 1 1	DM Base ($x2^1$)	Queue Size	A/B	W/R	Read Counter	Write Counter		Synchronization of two-operand data
RDCYCS	0 0 0 0	DM Base ($x2^1$) (DMB)	Buffer Size (BS)			--	Read Counter (RC)	0	DATA ← (DMB+RC), RC ← RC+1
								1	DATA ← (DMB+RC), RC ← RC+1, when BS=RC, copy
RDCYCL	1 0 0 0	DM Base ($x2^5$)	Buffer Size	--		Read Counter		0	DATA ← (DMB+RC), RC ← RC+1
								1	DATA ← (DMB+RC), RC ← RC+1, when BS=RC, copy
WRCYCS	0 0 0 1	DM Base ($x2^1$)	Buffer Size			--	Write Counter (WC)	0	(DMB+WC) ← DATA, WC ← WC+1, delete data
								1	(DMB+WC) ← DATA, WC ← WC+1, when BS=WC, data not deleted
WRCYCL	1 0 0 1	DM Base ($x2^5$)	Buffer Size	--		write Counter		0	(DMB+WC) ← DATA, WC ← WC+1, delete data
								1	(DMB+WC)←DATA,WC-WC+1,whenBS=WC,data not deleted
RDWR	0 1 0 0	DM Base ($x2^1$)	--	--		Address Register (AR)		0	DATA ← (DMB+AR+DATA), AR ← AR+DATA
								1	(DMB+AR) ← DATA, AR ← 0, data not deleted
RDIDX	0 1 0 1	DM Base ($x2^1$)	--	--		Address Register		0	DATA ← (DMB+AR+DATA), AR ← 0
								1	AR ← AR+DATA, data not deleted

Table 5-2 AG & FC Instructions (continued)

Mnemonic	FTR 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	FTT 9 8 7 6 5 4 3 2 1 0	FTRC	Operation
PICKUP	1 1 0 0 -- Count Size (CS)	- - Counter (C)	0	When CS≠C, C ← C+1, When CS=C, distribute, C ← 0
			1	C ← C+DATA, data deleted
COUNT	1 1 0 1 -- Count Size	- - Counter	0	When CS≠C, C←C+1, When CS=C, copy, C←0
			1	C ← C+DATA, data deleted
CUT	0 1 1 1 -- Count Size	S/F - Counter	0	When S/F=0 and C≤CS, C←C+1, data deleted; when S/F=0 and C>CS, or when S/F=1, C ← C+1, data not deleted
			1	S/F ← 0, C ← 0, data deleted
DIVCYC	1 0 1 0 -- Count Size / Divide Size	- - Counter / Counter	0	When C≤CS, C ← C+1; when C>CS, branch, C ← C+1, When C=DS, C ← 0
			1	C ← C+DATA, data deleted
DIV	1 0 1 1 -- Count Size	S/F - Counter	0	When S/F=0 and C≤CS, C ← C+1; when S/F=0 and C>CS, or when S/F=1 distribute, C ← C+1
			1	S/F ← 0, C ← 0, data deleted
DIST	0 0 1 0 -- / Δ ID Size	-- / Δ ID	0	ID ← (ID+ΔID) modulo ΔID size
			1	When ΔID≠ΔID size, ID ← (ID+ΔID) modulo ΔID size, ΔID ← ΔID+1 When ΔID=ΔID size, Δ ID ← 0
CONVO	1 1 1 1 -- Count Size	- - Counter	-	When CS≠C, ID ← ID+C (modulo2), C ← C+1 When CS=C, ID ← ID+2, C ← 0
SAVE	0 1 1 0 --	- - ID Stack Register (IDSR)	0	IDSR ← Lower 8-bit of DATA
			1	ID ← IDSR
CNTGE	1 1 1 0 -- Count Size	W/D - Counter	0	When dead, final processing (ID+2); when wait, warm start
			1	When dead, initialization; when wait, delete

5.1.1 QUEUE Instruction

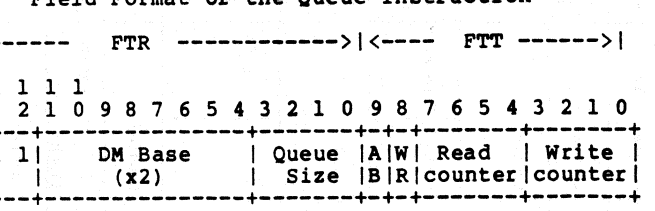
This instruction is used for such purposes as performing two-operand operations at the PU or for synchronizing program timing.

This instruction operates in the following manner. When the first token (FTRC=1/0) arrives at this instruction node, it is saved in the DM until the corresponding token (FTRC=0/1) arrives at the node. When the corresponding token arrives, the token that was first written in the DM is read and execution proceeds to the next cycle.

Note: The QUEUE instruction forms pairs of FTRC=0 and FTRC=1 tokens. For this reason, if too many of only one type of token arrive at the queue, the queue will overflow causing the BREAK mode to be entered.

The flowgraphs in Figures 5-2 and 5-3 show examples of using the QUEUE instruction. The example in Figure 5-2 shows token queuing to perform two-operand operations at the PU and that in Figure 5-3 shows the use of queuing to regulate the timing of the program flow.

Field Format of the Queue Instruction



The FTR field stores the fixed data portion of this instruction, FTT the dynamic data portion.

1. The OP Code of the QUEUE instruction is stored in the high order 4 bits of FTR.
2. DM base indicates the starting address of the queue memory (FIFO) within the DM. The 8-bit field of the DM base, however, is too small to point to all 512 addresses of the DM. The value of this field is therefore shifted left 1 bit and then used as the DM base. In other words, only even addresses within the DM can be used as the DM base.

   Note: The assembler automatically specifies the even addresses of data memory from 0 to 511. Therefore, there is no need for the user to take this into consideration when programming. However, when operations such as a direct dump of the contents of FTR are performed, the user must be aware of the way in which the DM base in

generated.

3. QUEUE SIZE indicates the size of the FIFO buffer to be used for the queue. This size can be set by the user in the range from 1 to 16 levels. The value set in this field, however, is the actual size -1.
4. The A/B bit indicates the state of the FTRC bit of the data written to the DM.
   When FTRC=0, A/B=0
   When FTRC=1, A/B=1.
5. The W/R bit indicates whether the operation performed when the previous token arrived at the queue was a read (a token with a complementary FTRC bit was waiting in the queue) or a write (there was no partner token and the arriving token was stored in the DM). If the last operation was a read, W/R=0 and, if it was a write, W/R=1. The initial state of is W/R=0.
6. The READ COUNTER (RC) and WRITE COUNTER (WC) are used to generate the addresses of the tokens stored in the DM. When a token is written to the DM, WC is incremented, and when a token is read, RC is incremented. The following expression shows the interrelation between these values. If M is the incremented value of the read or write counter, QS the value specified as QUEUE SIZE, and C the value specified for the read or write counter, then

$$C = M \bmod QS$$

Here C is the remainder of the integer division of M by QS. The queue level where tokens will be read or written is calculated using these values.

For example,

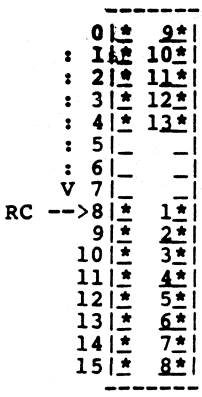
(a) If R/W=0, A/B=undefined, and RC=WC=0 as the intial condition, this indicates that the queue is empty.
(b) When R/W=1, A/B=1, RC=8, WC=5 and QS=16*, the state of the queue is as shown in the Figure 5-1. The available queue areas are 5, 6, and 7. There are currently 13 FTRC=1 tokens stored in the queue area.

* In this case, if WC and RC counters were not modulo counters, when the data written to the DM were read, the relation between the values of the two counters would become:

$$WC \geq RC$$

with the equal size valid when R/W=0. However, since modulo calculation of the counters is used, the result is as shown in the example. In other words, WC < RC indicates that there are still data stored in the DM that have not yet been read. Also note that when WC=RC, R/W=0 indicates that the queue is empty and R/W=1 indicates that the queue is full.

Figure 5-1
RC and WC



The starred (*) portions indicate where tokens waiting for a partner are stored.

The numbers stored between the stars (*) indicate the order in which the tokens are actually read out.

(c) When R/W=1 and RC=WC, that is, when the queue buffer is full, if a token arrives whose FTRC value matches that of the A/B bit, the token will be written to the DM and an overflow will occur.

Calculating the DM address:

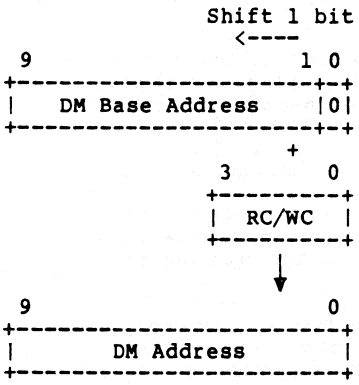
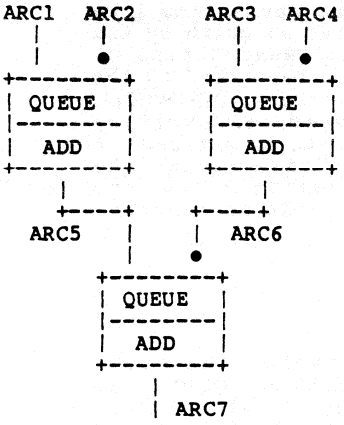
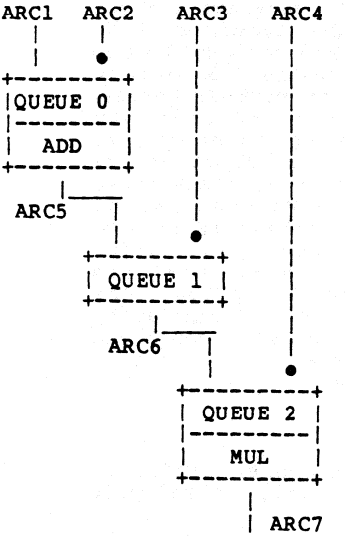


Figure 5-2



The tokens carrying the necessary data are asynchronously input to arcs ARC1 through ARC4. The QUEUE instructions queue ARC1 with ARC2 and ARC3 with ARC4. It then adds these values. The results of these operations are output to ARC5 and ARC6. These are queued in turn and added. The result is output to ARC7.

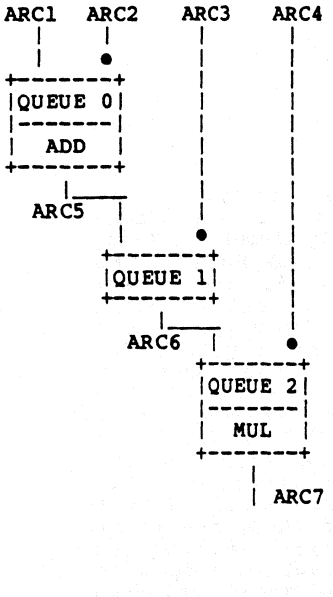
Figure 5-3



The tokens carrying the necessary data are asynchronously input to ARC1, ARC2, ARC3 and ARC4. The result of the addition of ARC1 and ARC2 is output to ARC5. The token input to ARC4 (FTRC=1) waits at QUEUE2 for the arrival of the token from ARC6. The result of the additon that has been output to ARC5 (FTRC=0) waits at QUEUE1 for the arrival of the execution token from ARC3. When the execution token arrives from ARC3, the token that has been waiting at QUEUE1 (FTRC=0) is output to ARC6 (because FTRC=0 indicates the A side). At this point, the data necessary for a multiplication operation (the data output to ARC6 and the data from ARC4) are present in QUEUE2. The result of the operation is then output to ARC7.

Among the parameters of the QUEUE instruction, QUEUE SIZE can be specified. This is the queue buffer size and can be set by the user to any value in the range from 1 to 16. FTRC=1 and FTRC=0 tokens are handled as pairs by the QUEUE instruction. Therefore, if too many tokens of FTRC=1(0) arrive at the queue, they will be stored in the DM to wait for the arrival of partner tokens (FTRC=0(1)). If the specified queue size is exceeded, an overflow error will occur causing the BREAK mode to be entered. A large enough queue size must be specified taking the token flow into consideration. Programming of the sort shown in example Figure 5-4 will result in an overflow error.

Figure 5-4



In this example, the queue size for QUEUE0 and QUEUE2 are both 16 (the actual value is 15; hereafter, actual values are shown in parentheses). The input method adopted for this example is to repeatedly input one token to ARC1, ARC2 and ARC4 every cycle and to input to ARC3 once every three cycles. FTRC=0 tokens will be output to ARC5 each cycle and will wait (on the FTRC=0 side of QUEUE1) for partner tokens. Since ARC3 inputs only one such token every three cycles, tokens will accumulate at the rate of two tokens every three cycles. The FTRC=0 side of QUEUE1 will exceed 16 levels and an overflow will occur when this operation has continued for 25 cycles. One solution is to temporarily stop input to ARC1, ARC2 and ARC4 (using input restrictions).

QUEUE instruction assembler description
(For detailed specifications of the assembler, refer to the uPD7281 Assembler Operating Manual, available separately.)

1. FUNCTION FQUE1=QUEUE(base,size)
   base : This parameter specifies the DM base. The memory size indicated by 'size' from the

location specified by this parameter is defined as the QUEUE (FIFO buffer) memory. The absolute DM address is automatically set to an even address by the assembler and must therefore be defined by an assembler MEMORY statement.

size : This parameter specifies the queue size, in this case the size of FQUE1. The size of the queue may be freely specified by the user within the range of from 1 to 16. This is the maximum number of data words that can accumulate (be queued) in the DM. Since the field includes 0, the value of this field will be that specified in the assembler description minus 1.

Field	:	0 to 15
Assembler	:	1 to 16

2. FUNCTION FQUE1=QUEUE(base,size),temp

temp : Specifies the FTT field of the instruction. Any 10-bit value for FTT can be directly specified here. A full knowledge of the contents of the FTT is necessary before this specification can be performed.

For example, when temp=301H:

This sets the FTT state when one FTRC=1 token has arrived at the queue. When an FTRC==0 token arrives, the 18-bit data stored in the DM base address is read out and execution proceeds to the next cycle.

3. FUNCTION FQUE1=ADD(XY,EX),QUEUE(QUE1,16)

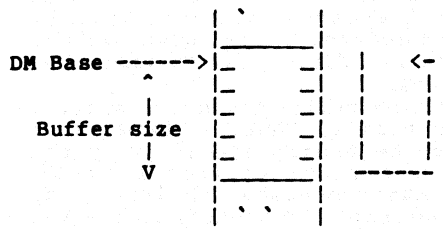
PU, OUT or GE instruction — AG & FC instruction

This is an example of using the QUEUE instruction in combination with a PU instruction. The PU (or OUT or GE) instruction is described on the left side and the AG & FC instruction on the right. The format of the AG & FC instruction is the same as when it is used alone.

5.1.2 RDCYCS (Read cycle short) Instruction

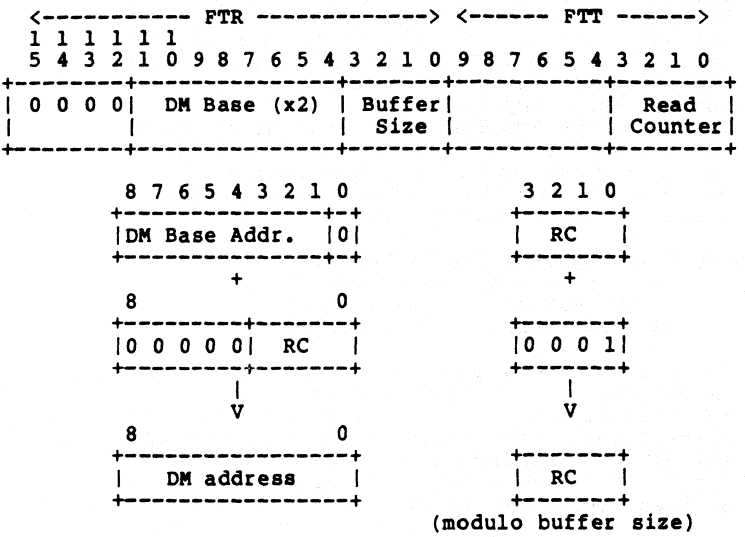
This instruction is used mainly to read the 18-bit contents of the DM. RDCYCS performs cyclic read of the DM address range specified as the buffer size. The function of this command can be easily envisaged by referring to Figure 5-5.

Figure 5.5



The maximum buffer size that can be specified is 16 levels. When FTRC=1 and RC (read counter)=BS (buffer size), the token is copied by the PU.

RDCYCS instruction field format



The FTR field stores the fixed data portion of this instruction, FTT the dynamic data.

1. The OP Code of the RDCYCS instruction is stored in the high order 4 bits of the FTR.
2. DM Base indicates the starting address of the read data stored in the DM. The 8 bit field of DM Base, however, is not large enough to point to all 512 addresses of the DM and is therefore shifted 1 bit and then used. In other words, only even addresses within

the DM can be used as the DM base.

Note: The assembler automatically specifies the even addresses of data memory addresses 0 to 511 and there is no need for the user to take this into consideration when programming. However, when operations such as direct dump of the contents of FTR are performed, the user must be aware of the way in which the DM base is generated.

3. Buffer Size (BS) expresses the size of the buffer that stores the data to be read by this instruction. Since the BS field of this instruction is 4 bits long, the user can specify any size in the range of from 1 to 16 (0 to 15).
4. The Read Counter (RC) is added to the DM base x 2 to generate the address of the DM that is actually read. In other words, RC points to the address (from the DM base) to be read. The value of RC never exceeds that of BS. The value of RC never exceeds that of BS. After the DM address has been generated, the value of RC is incremented. Note that RC is a modulo counter.

Execution of this instruction reads 18 bits of data consisting of the 16-bit data, plus C and S bits. The address to be read is determined by doubling the value of DM base and adding the value of the read counter (RC). When RC=BS and the arriving token is FTRC=1, the data will be copied by the PU. (Copying here means to increment the value of ID to ID+1 to create a new token; the data field of the new token is identical to that of the original token.) Also note that for this instruction, since BS is a 4-bit field, a 16-level read cycle is the maximum that can be set. any value in the range 1 to 16 (0 to 15*) may be selected by the user for the BS level. To perform cyclic read operations of greater length, use the RDCYCL (read cyclic long) instruction.

* Values in the parentheses mean real setting data. The same thing will be applied in the following pages.

The following examples with flowgraphs illustrate the use of the RDCYCS instruction.

Figure 5-6 shows an example of the read instruction used by itself. The data read from the DM becomes the A data* and is output to ARC2.

* 'A data' and 'B data' are the names given to data when they are input to the PU.

Figure 5-6

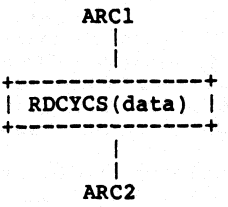


Figure 5-7 shows the read instruction used in combination with a PU instruction. Data is read from the DM by the token arriving from ARC1. When a PU instruction is used in combination with an AG & FC instruction, the data read from the DM is B data and is sent to the PU. (Note that A and B data can be exchanged by means of the XCH bit in the FTL field of the PU instructions; details are given in a later section.) At the PU, addition of the A data (the data of the token input from ARC1) and the B data (the data read from the DM) is performed and the result is output to ARC2.

Figure 5-7

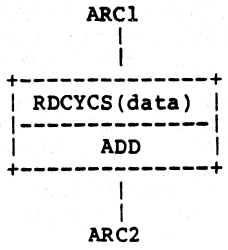
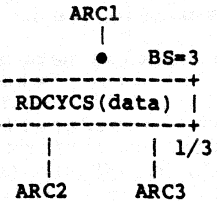


Figure 5-8 shows the use of an independent read instruction. In this example, FTRC=1 tokens are input. When RC (read counter)=BS (buffer size), the token is copied by the PU. Here a value of 3 has been specified for BS so the data read by the first and second read operations are output to ARC2. When the third read is performed, the condition RC=BS is fulfilled and the token is copied. The input token is output to ARC2 and the copied token to ARC3.

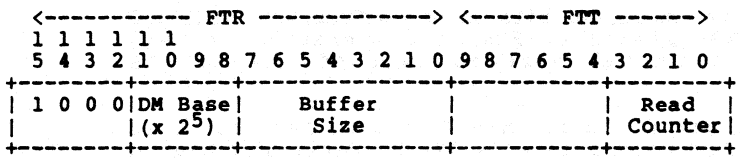
Figure 5-8

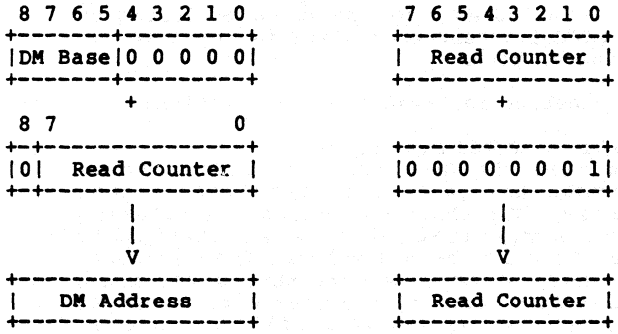


5.1.3 RDCYCL (Read Cyclic Long) Instruction

The use of this instruction is essentially the same as that of the RDCYCS instruction. The bit lengths of the DM base and buffer size (BS) for the two instructions are different.

RDCYCL instruction field format





The FTR field stores the fixed data portion of this instruction and FTT the dynamic data.

1. The OP Code of the RDCYCL instruction is stored in the high order 4 bits of FTR.
2. The significance of DM base in this instruction is the same as in the RDCYCS instruction. However, as this

field is only 4 bits wide, it is shifted 5 bits to the left and then used as the DM base. In other words, DM base can only be addresses within the DM (0 to 511) that are multiples of 32.

Note: The assembler automatically sets addresses that are multiples of 32 from data memory addresses 0 to 511 and there is no need for the user to take this into consideration when programming. However, when operations such as direct dump of the contents of FTR are performed, the user must be aware of the way in which the DM base is generated.

3. Buffer Size (BS) is used in the same manner as in the RDCYCS instruction. In the case of the RDCYCL instruction, this field is 8 bits long. The user can therefore specify sizes in the range of from 1 to 256 (0 to 255).
4. The Read Counter (RC) is added to the DM base x 2^5 to generate the address of the DM to be read. In other words, RC expresses which number data from the DM base is to be read. The value of RC never exceeds that of BS.

Except for the number of bits of the buffer size (BS) and DM base (DMB) fields of FTR, the operation of this instruction is the same as that of the RDCYCS instruction.

Because the DMB field is only 4 bits in length, it is shifted 5 bits to the left (x 32) to determine the DM base address. Therefore, only those addresses within the DM (addresses 0 to 511) that are multiples of 32 can be expresses by DMB. The token copy operation performed by the PU for input FTRC=1 tokens when RC=BS is the same as with the RDCYCS instruction.

RDCYCS and RDCYCL instruction assembler description:

1. FUNCTION FRED1 = RDCYCS(base,size)

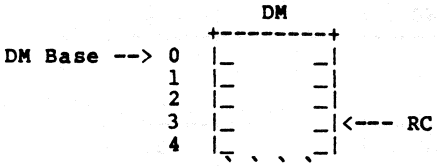
   base : Specifies the DM base. The buffer whose contents are to be read starts from this address. The absolute address in relation to this specification is automatically determined by the assembler. The assembler does not, however, secure the buffer area. This must be done by the user with a MEMORY statement. The DM base will be an even address in the case of the RDCYCS instruction and a multiple of 32 in the base of the RDCYCL instruction.

   size : Specifies the buffer size. Indicates the size of the buffer area from which cyclic read is to be performed. This value is from 1 to 16 (0 to 15; 4 bits) in the case of the RDCYCS instruction and 1 to 256 (0 to 255; 8

bits) in the case of the RDCYCL instruction.

2. FUNCTION FRED1 = RDCYCS(base,size),temp
   temp : Specifies FTT of the command field. Any 10-bit value for the FTT can be directly specified here. A full knowledge of the contents of FTT is necessary to determine this value, however. Refer to the FTT field format when setting the data for temp.

   When temp=3:
   The state existing after three read operations have been performed is set in FTT. (The value of BS, however, must be greater that 3.) The value of the read counter (RC) is stored in FTT indicates the point to which the read operation has progressed in the buffer.



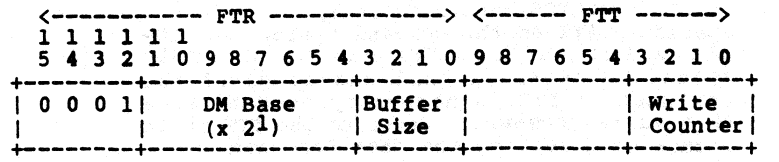
3. FUNCTION FRED1 = ADD, RDCYCS(base,size)

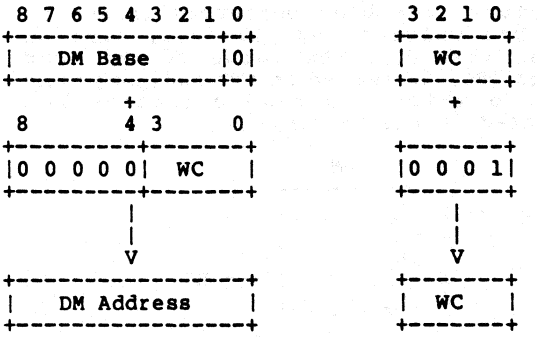
This is an example of using the RDCYCS instruction in combination with another instruction (in this case a PU instruction). A PU, OUT or GE instruction should be described on the left and the AG & FC instruction on the right. The description format is otherwise the same as when the AG & FC instruction is used alone.

5.1.4 WRCYCS (Write Cyclic Short) Instruction

This instruction is used to write 18-bit data words to the DM. This instruction performs cyclic write to the DM from the DM base for the range indicated by the buffer size (BS). Both of these are user-specified. The operation of this instruction differs from that of the RDCYCS instruction only in that it performs write instead of read and a write counter is used instead of a read counter.

WRCYCS instruction field format





The FTR field stores the fixed data portion of this instruction, FTT the dynamic data.

1. The OP Code of the WRCYCS instruction is stored in the high order 4 bits of FTR.
2. The use of DM base and buffer size (BS) is exactly the same as for the RDCYCS instruction.
4. The write counter (WC) functions in much the same manner as the read counter of the RDCYCS instruction.

The 18 bits of data (Data, C, S) carried by the token arriving at this instruction are written to the DM address indicated by the sum of the value of DM base after it has been shifted left (x 2) and the value of the write counter (WC). After the DM address has been generated, the value of WC is incremented and rewritten to FTT. The modulo (remainder of integer division) is used to control the increment operation. That is, when WC (before incrementing) becomes equal to the value of BS, data 0000B will be written to the WC field of FTT. Normally, after the 18 bits consisting of Data, C and S have been written to the DM, the token is deleted. When WC (before incrementing)=BS and S=1, however, this data is not deleted but is output to Q.

Also, since the BS field is 4 bits wide, the maximum number of cyclic write operations that can be performed

is 16 levels. The value for BS can be set by the user in the range 1 to 16 (0 to 15). To perform longer cyclic write operations, use the WRCYCL (Write Cyclic Long) instruction.

The following are actual use examples with flowgraphs. Figure 5-9 shows an example of the single write instruction. The 18-bit data of the token input from ARC1 are written to the DM and then the tokens are deleted.

Figure 5-9

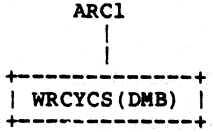


Figure 5-10 shows an example of inputting an FTRC=1 token to execute a WRCYCS instruction. After the instruction executes, when buffer size (BS)=write counter (WC), the input token is not deleted but is output to ARC2. In the example, BS=WC will be true for every third token.

Figure 5-10

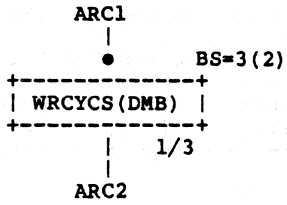
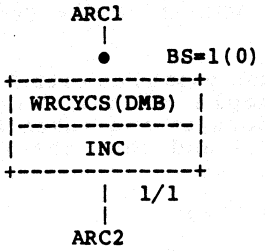


Figure 5-11 shows an example of using WRCYCS linked to a PU instruction. In this case, special attention must be paid to the buffer size (BS). If BS=1(0), BS=WC will be true for each write operation and all of the tokens will be output to the PU (in this case, INC) instruction. The result is then output to ARC2.

Figure 5-11

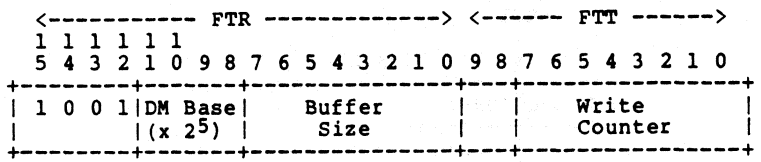


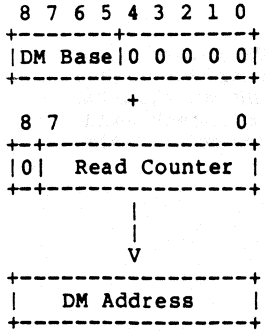
5.1.5 WRCYCL (Write Cyclic Long) Instruction

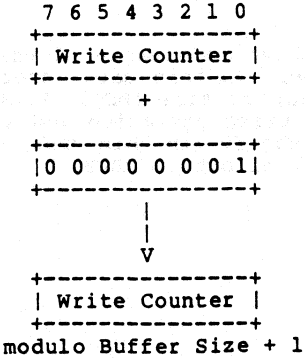
The method for using this instruction is the same as that for the WRCYCS instruction.

The operation of this instruction differs from the RDCYCL instruction only in that in performs write instead of read and that a write counter is used instead of a read counter.

WRCYCL instruction field format:







The FTR field stores the fixed data portion of this instruction and FTT the dynamic data.

1. The Op Code of the WRCYCL instruction is stored in the high order 4 bits of FTR.
2. The significance of the DM base in this instruction is exactly the same as in the RDCYCL instruction.
3. The significance of buffer size in this instruction is exactly the same as in the RDCYCL instruction.
4. The function performed by the write counter (WC) is much the same as that of the read counter in the RDCYCL instruction.

Except for the number of bits of the buffer size (BS) and DM base fields of FTR, the operation of this instruction is the same as that of WRCYCS instruction. Because the DM base field is only 4 bits in length, it is shifted 5 bits to the left (x 32) to generate the DM base address. Therefore, only those addresses within the DM (address 0 to 511) that are multiples of 32 can be used as DM base. These addresses are automatically set by the assembler. Another difference with the WRCYCS instruction is that the write counter is an 8-bit field.

WRCYCS and WRCYCL instruction assembler description:

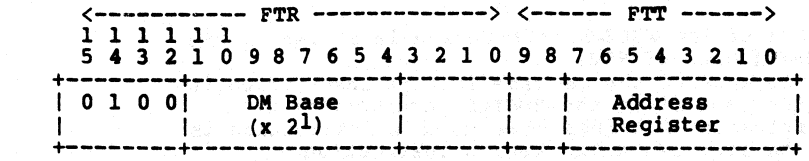
The assembler descriptions for these instructions are essentially the same as those for the RDCYCS and RDCYCL instructions. Refer to the descriptions of those instructions.

5.1.6 RDWR (Read/Write) Instruction

This instructions performs read (FTRC=0 token) and write (FTRC=1 token) of the contents of the DM. In other words, this instruction can use pairs of tokens consisting of an address setting token (FTRC=0) and a data token (FTRC=1) to rewrite the contents of the DM. (If the only operation to be performed is read of the DM contents, use of the RDIDX instruction is recommended.) This instruction can also be used to perform read/modify/write operation to the contents of the DM. In this case, the time from the read operation until the write operation completes is used as the basic timing cycle. No other token may use the FTR at this location during this period.

Note: The same node (the same FTR) is used to set both the write and the read address. Refer to Figure 5-13.

RDWR instruction field format:



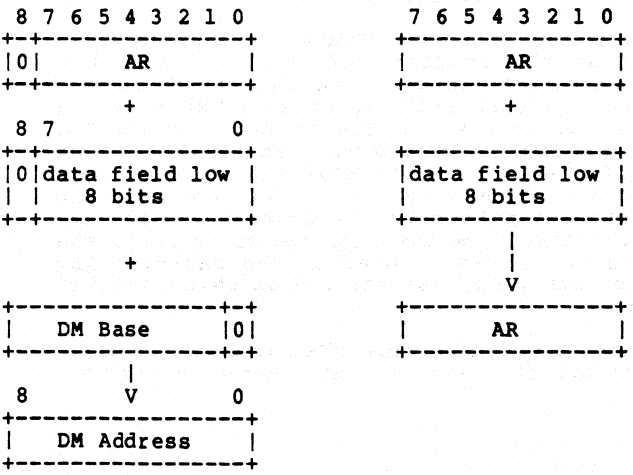
The FTR field stores the fixed data portion of this instruction and FTT the dynamic data.

1. The Op Code of the RDWR instruction is stored in the high order 4 bits of FTR.
2. The use of DM base is exactly the same as for the RDCYCS instruction; i.e. to indicate the starting address of the DM area to be accessed (refer to the description of the RDCYCS instruction).
3. The address register (AR) field stores the contents of the FTRC=0 token data field. The read/write operation is performed to the DM address generated by adding this value to the DM base x 2.

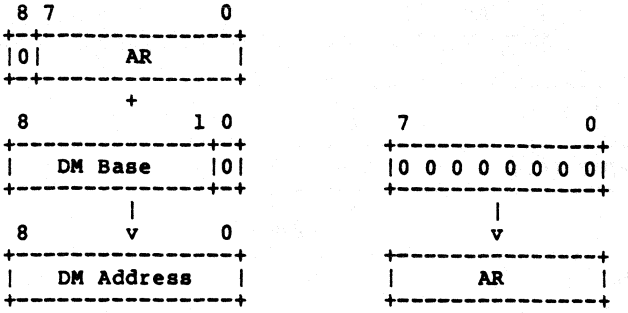
For FTRC = 0 tokens:

Reads the contents of the DM address found as the sum of these three values: the contents of the address register (AR) (initial value: 0), the low order 8 bits of the token data field and the DM base.

The sum of the contents of AR and the low order 8 bits of the data field are then rewritten to AR. Any overflow (more than 8 bits) generated is ignored.



FTRC=1 tokens:
The contents of the address register (AR) and DM base x 2 are added to generate the DM write address. The 18-bit data of the token is then written to this address. When the data has been written to the DM, AR in the instruction token FTT field is reset to 0.



e following are actual examples with flowgraphs.
the example shown in Figure 5-12, the relative address ffset) from the DM base is input from arc WA and the ite data from arc WD. The reason a QUEUE instruction used in the first node is that the write data must be itten to the DM before the next address setting token rives and sets a new address. Here the QUEUE ıtructions acts to synchronize the tokens and for this ıson the tokens will be output successively to execute RDWR instructions read and write. Also, note that contents of the DM indicated by the address setting en (WA') are read after the address has been set. In s example, however, the read data is not used but ply deleted.

Figure 5-12

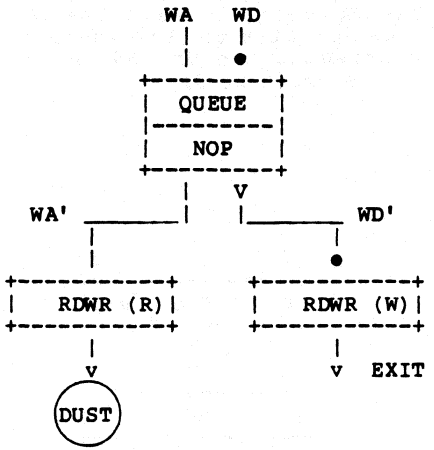
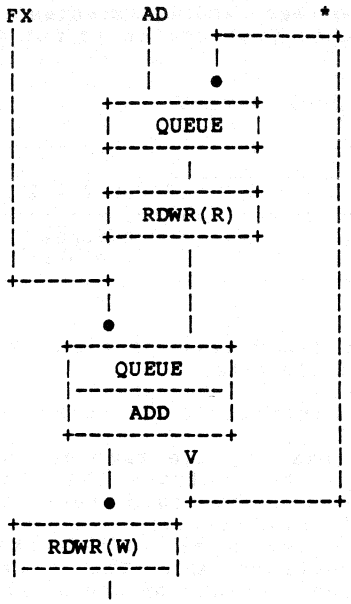


Figure 5-13 shows an example of using the RDWR instruction to perform a read/modify/write operation. Here the contents of the DM address specified by the token input to AD are queued with and added to (modified by) the token input to FX. Then a RDWR(W) instruction is used to write the result to the address read earlier in the operation. When this process completes, the next modify operation is performed.

Figure 5-13



* The data must already be waiting at the QUEUE instruction when the token arrives from the AD arc.

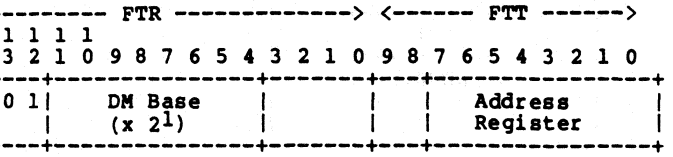
RDWR instruction assembler description:

1. FUNCTION FRDWR1 = RDWR(base),temp
   base : Specifies the DM base address to be used by this instruction. This specified value is automatically converted and set in the FTR field by the assembler. A MEMORY statement, however, must still be used to secure the memory area.
   temp : This value is used to set the contents of the FTT. The value specified as temp is directly written in the FTT field.

5.1.7 RDIDX (Read Index) instruction

This instruction is used to read random addresses within the DM. The main use of this instruction is for table look-up operations.

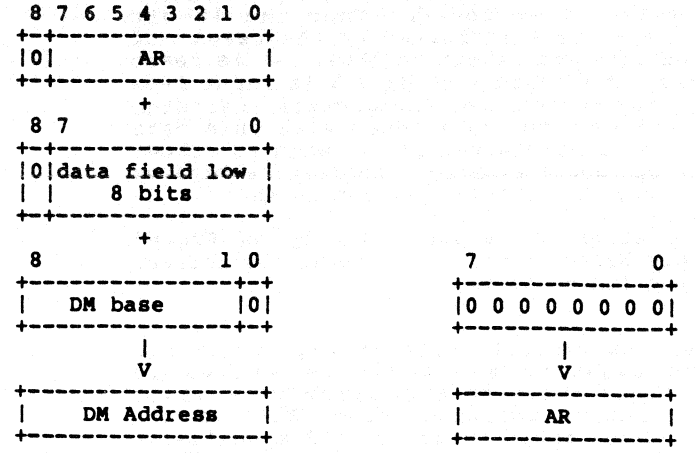
RDIDX instruction field format:



The FTR field stores the fixed data portion of this instruction and FTT the dynamic data.

1. The Op Code of the RDIDX instruction is stored in the high order 4 bits of FTR.
2. The use of DM base is exactly the same as for the RDCYCS instruction, i.e. to indicate the starting address of the DM area to be accessed (refer to the description of the RDCYCS instruction).
3. The contents of the data field of the FTRC=1 token are stored in the address register (AR). The arrival of an FTRC=0 token reads the contents of the DM address specified by the sum of this register, the data field of the FTRC=0 token and the DM base (x 2). After the read operation is performed, AR is automatically reset to 0.

- When FTRC=0:

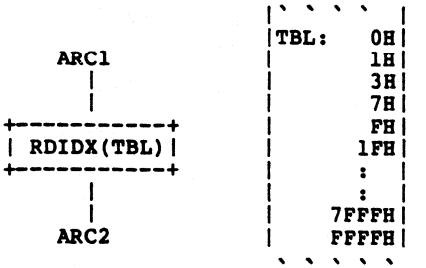


The contents of the address register (AR) are added to the low order 8 bits of the data field of the arriving token. DM base x 2 is then added to the result to generate the DM address. The 18 bits at this address (consisting of 16-bit DATA, C, S) are then read.

After the DM address has been generated, AR is reset to 0. Even if the addition of AR and the lower 8 bits of the data field generate an overflow bit, this bit does not participate in the DM address calculation.

In the example in Figure 5-14, standard look-up is performed for mask data. Here it is assumed that DM base x 2 points to TBL: (first address of the table) and that the values indicated in the figure are stored in the table.

Figure 5-14

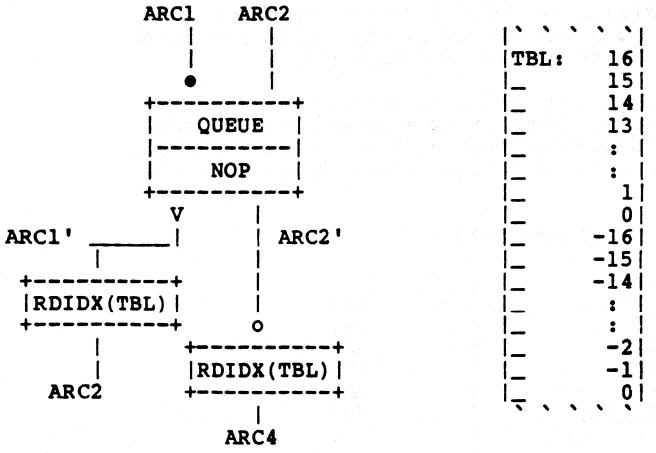


When an FTRC=0 token is input, the sum of the lower 8 bits of the data of the token input from ARC1, DM base x 2, and AR (default: 0) is found. Then the 18-bit contents of the DM address indicated by the result of this addition are read and output to ARC2. AR is reset to 0. For example, if a token with data 5 is input from ARC1 (when AR=0), the contents of the address indicated by TBL+5 (1FH) will be read and a token with this data output to ARC2. AR is maintained at 0. When an FTRC=1 token is input in the above example, the lower 8 bits of the data field of the FTRC=1 token are set in AR.

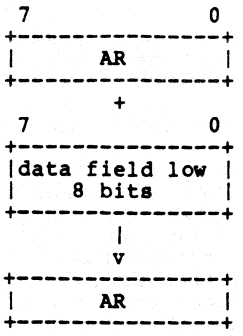
AR is reset to 0 after it is used once by an FTRC=0 token. The example below (Figure 5-15) shows the correct usage of FTRC=0 and FTRC=1 tokens.

In this example, non-periodic referencing of a table containing both negative and positive values is performed. The token input to ARC1 contains a value indicating the relative position (from TBL:) of the reference data and the token input to ARC2 specifies a value for TBL:. In other words, to reference positive values within the table, 0 should be set in the data field of the ARC2 token (set TBL: as the starting address); to reference negative values, the data field of the input token should be set to 17.

Figure 5-15



- For FTRC=1 tokens:
  FTRC=1 tokens are input to set the user specified value in the address register (AR). The lower 8 bits of the data field of the input token are added to the current value of AR and the result is rewritten to AR. Any overflow generated, however, will be ignored. The data field of the token is not deleted.



RDIDX instruction assembler description:

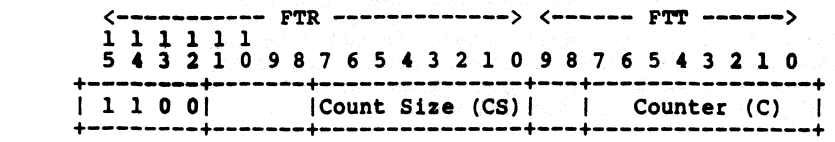
1. FUNCTION FIDX1 = RDIDX(base),temp
   base : Specifies the DM base address to be used by this instruction. This value is automatically set in the FTR field by the assembler.
   temp : This value is used to set the contents of the FTT. The value specified as temp is directly set in the FTT field.

5.1.8 PICKUP Instruction

This instruction counts the number of arriving tokens and compares this count against the count size (CS) specified by the user. When these two values coincide, the ID of that token is incremented by 1 and output as ID+1. At the same time, the contents of the counter are reset to 0.

The difference between the COUNT instruction and the PICKUP instruction is in the processing that is performed when the count size in FTR and the counter (C) in FTT coincide. The PICKUP instruction outputs only a single token (ID+1; token flow branch), and the COUNT instruction outputs two tokens (ID and ID+1; token copy).

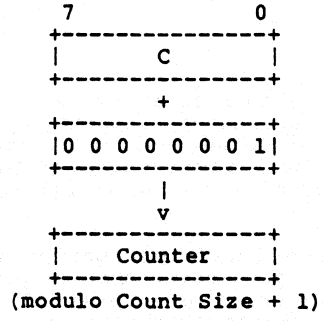
PICKUP instruction field format:



The FTR field stores the fixed data portion of this instruction and FTT the dynamic data.

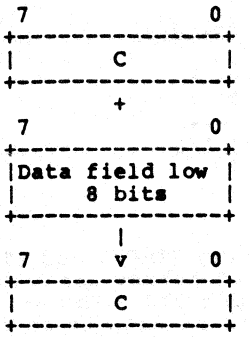
1. The Op Code of the PICKUP instruction is stored in the high order 4 bits of FTR.
2. The count specified by the user is set in the count size field. The value actually stored, however, is the value specified in the assembler minus 1.
3. The counter field is used to count the number of tokens arriving at this instruction. Each time a token arrives, the value of this field is incremented. When the value of the counter and that of the count size match, token flow branches (an ID+1 token is generated) and the counter is clear to 0).

- FTRC = 0 tokens
  The counter (C) is incremented each time a token arrives at this instruction. The modulo of the count size (CS) +1 is used to control the counter (C). In other words, when C (before it is incremented) = CS, C is reset to 0. The ID field will be incremented (ID+1) to branch execution flow and the token output with the next cycle.



- FTRC = 1 tokens
  The low order 8 bits of the data field of the arriving tokens are added to the contents of the counter field of the instruction. The token is then deleted. This

addition does not use the modulo of the counter size and any overflow generated is ignored.



In the example shown in figure 5-16, every fourth token arriving from IN is incremented (ID+1) and output to OT2.

Figure 5-16

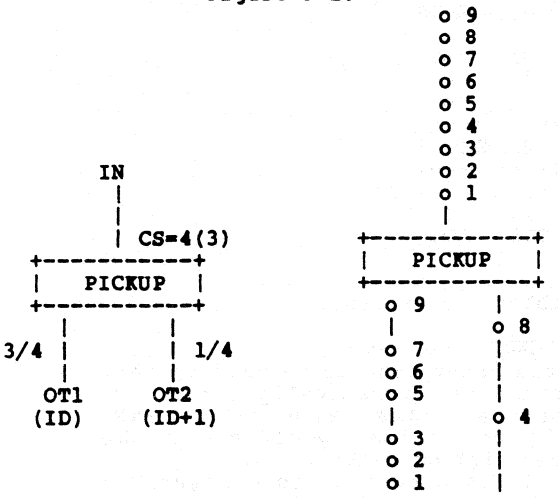


Figure 5-17 shows an example of using the PICKUP instruction to perform delay loop processing for arriving tokens. The tokens loop nine times and then on the tenth loop are output to OT with ID+1.

Figure 5-17

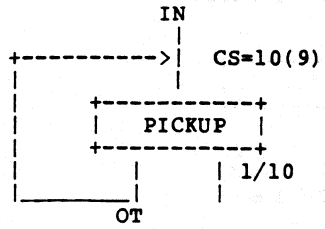
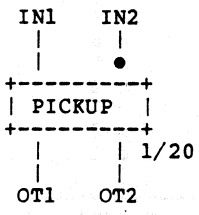


Figure 5-18 shows an example of using FTRC=0 and FTRC=1 tokens in combination. The flow of tokens arriving at IN1 is divided and output to OT1 and OT2. The arrival from IN2 of an FTRC=1 token containing data to be force set as the FTT count value causes the low order 8 bits of the data to be added to the contents of the counter. The result is then written to FTT. The token that brought this data is then deleted. Thus when the next FTRC=0 token arrives from IN1, a new operation based on the new value set in the FTT counter begins.

Figure 5-18



PICKUP instruction assembler description:

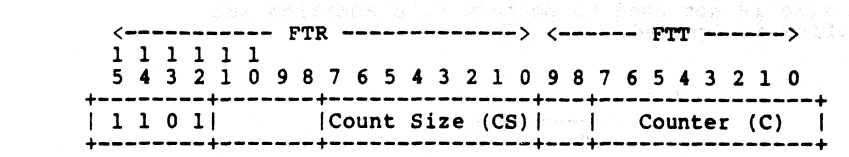
1. FUNCTION FPIC1 = PICKUP(size),temp
   - size : Indicates the count size to be used by this instruction and is automatically set in FTR by the assembler. The value set in FTR, however, is the value specified in the assembler description minus 1.
   - temp : This value is used to set the contents of FTT. In this instruction, the counter is stored in FTT. The assembler sets the value of temp directly (without performing -1 subtraction) into FTT. This subtraction must therefore be done by the user before specifying temp.

5.1.9 COUNT Instruction

This instruction counts the number of arriving tokens and compares this count with the count size (CS) specified by the user. When these two values coincide, the instruction copies and outputs two tokens, one with ID and one with ID+1. At the same time, the contents of the counter are reset to 0 (initial value).

The difference between the COUNT instruction and the PICKUP instruction is in the processing that is performed when the count size (CS) in the FTR and the counter (C) in the FTT coincide. In the case of the PICKUP instruction, a single token with ID+1 is output, and in the case of the COUNT instruction two tokens, one with ID and one with ID+1, are output.

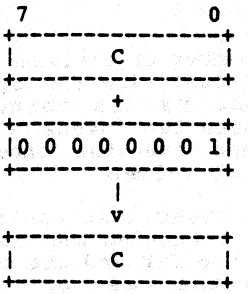
COUNT instruction field format:



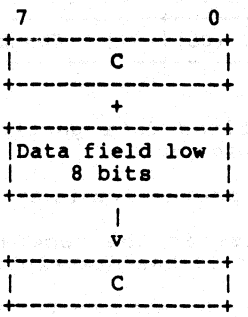
The FTR field stores the fixed data portion of this instruction and FTT the dynamic data.

1. The Op Code of the COUNT instruction is stored in the high order 4 bits of FTR.
2. The use of the count size is the same as for the PICKUP instruction (refer to the description of the PICKUP instruction).
3. The use of the counter is the same as for the PICKUP instruction (refer to the description of the PICKUP instruction).

- FTRC = 0 tokens
  The counter (C) is incremented each time a token arrives at this instruction. The modulo of the count size (CS) +1 is used to control the counter (C). In other words, when C (before it is incremented) = CS, C is reset to 0. Also, when C=CS, the token is copied by the PU (the ID field used is ID+1).



- FTRC = 1 tokens:
  The lower order 8 bits of the data field of the arriving token are added to C and the result is set in C. The token is then deleted. The modulo of the counter size is not used to perform this addition and any overflow is ignored.



In the example shown in figure 5-19, the COUNT instructions copies and outputs an ID and an ID+1 token for every fourth token arriving from arc IN. This operation results in all tokens that are multiples of 4 being output to both OUT1 (ID) and OUT2 (ID + 1).

Figure 5-19

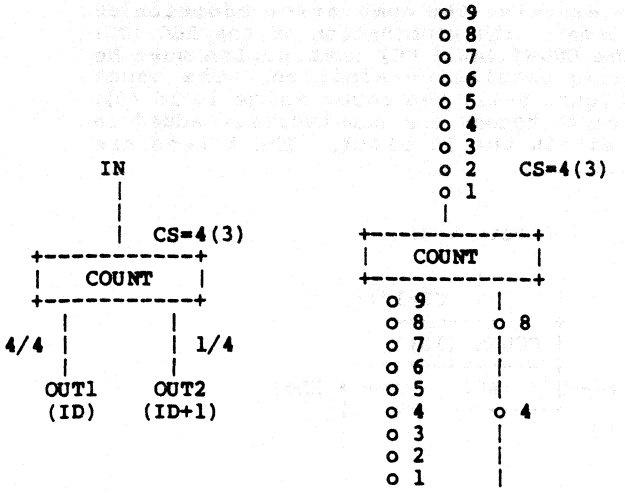
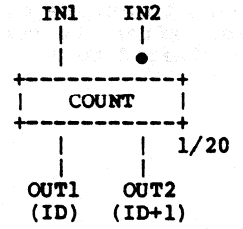


Figure 5-20 shows an example of using FTRC=0 and FTRC=1 tokens in combination. Normally, if there is no input to IN2, the tokens arriving at IN1 are processed and output to OUT1 (ID) and OUT2 (ID+1). However, if an FTRC=1 token containing data to be force set as the count value of the FTT arrives from IN2, the low order 8 bits of the data field will be added to the value of the counter and the result written to the FTT counter field. The token that brought this data is not output but deleted. When the next token arrives from IN1, a new operation based on the value just set in the FTT counter begins

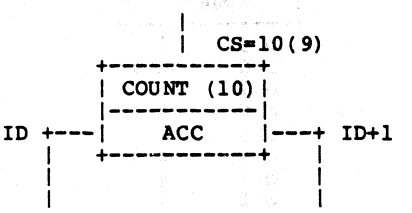
Figure 5-20



The COUNT instruction can also be used as a pair with the PU instruction ACC (accumulate). The operation performed

by this combination is illustrated in figure 5-21. (The ACC instruction is described in detail in a later section.) In this example, the cumulative addition of ten tokens is performed. The combination of the ACC (PU) instruction and the COUNT (AG & FC) instruction must be used when performing cumulative addition. The count value is 10. In Figure 5-21, the count value is 10 (9). The first through ninth tokens are cumulatively added in the ACC register within the PU block. The tokens are then deleted.

Figure 5-21



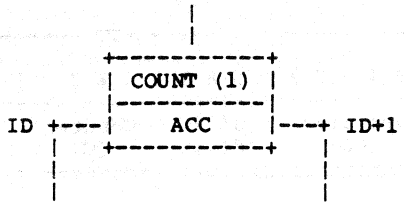
After the contents of the tenth token have been added, the ACC register is read and the contents output to the LT. The ID field of this token, however, may be either ID or ID+1 as explained below:

ID : This is the ID field when all ten tokens are cumulatively added without causing an overflow.

ID+1 : This is the ID field when one or more overflows has occured. The number of overflows, however, cannot be known and the overflow bit is not included in the result.

If the user wishes, for example, to terminate operation and read out the contents of the ACC register, a different node (Figure 5-22) combining a COUNT(1) and an ACC instruction should be used. The data of the token input to this node must be set to 0 to assure the result of addition will not be affected. After the contents of the ACC register are read, it is cleared to 0.

Figure 5-22



Note: When using the ACC instruction, BRC=1 and PNZ=000 must be set at the X output. This, however, is done by the assembler and there is no need for the user to be aware of the states of these bits.

COUNT instruction assembler description:

1. FUNCTION FCNT1 = COUNT(size),temp

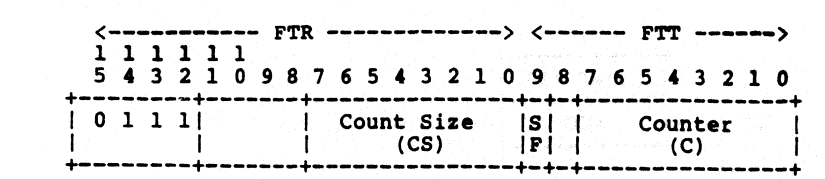
   size : Indicates the count size to be used by this instruction and is automatically set in FTR by the assembler. The value actually set in FTR, however, is the value specified in the assembler description minus one.

   temp : This value is used to set the contents of FTT. In this instruction, the counter is stored in FTT. The assembler sets the value of temp directly (without performing minus 1 subtraction) and this subtraction must therefore be done by the user before specifying temp.

5.1.10 CUT Instruction

This instruction is mainly used to delete unnecessary tokens from a series of tokens. The CUT instruction counts and deletes tokens up to the number specified by the user. When the number of arriving tokens matches the number specified by the user, the tokens from that point are not deleted but output (the token matching the user token number specification is deleted). The instruction is initialized by the arrival of an FTRC=1 token.

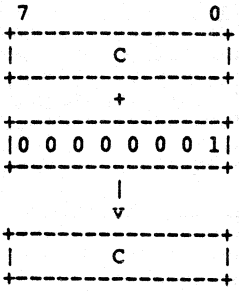
CUT instruction field format:



The FTR field stores the fixed portion of this instruction and FTT the dynamic data.

1. The Op Code of the CUT instruction is stored in the high order 4 bits of FTR.
2. The count size indicates the number of arriving tokens to be deleted. All tokens from the first arriving token to the token number indicated by this parameter are deleted.
3. The initial value of the S/F bit is 0; it is set to 1 when count size = counter. This token is deleted. All tokens thereafter are not deleted but output to the next cycle. However, when an FTRC=1 token is input to execute this instruction, the S/F bit and the counter are both cleared to 0 and the instruction returns to the initial state.
4. The counter is used to count the number of tokens arriving at this instruction. Each time a token arrives, the value of the counter is incremented. When an FTRC=1 token is input to execute this instruction, the counter is cleared to 0. (The FTRC=1 token input at this time is deleted.)

- FTRC=0 tokens:
  While S/F=0 and $C \leq CS$, C is incremented and arriving tokens are deleted. When C=CS, S/F is set to 1 and the token arriving at that time is deleted. When S/F=0 and $C > CS$, or when S/F=1, the tokens are not deleted but are sent to the next block. The S/F bit is not cleared until an FTRC=1 tokens arrives.



- FTRC=1 tokens:
  The S/F bit and C are both cleared to 0 and the delete operation repeats.

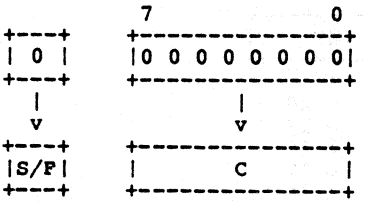
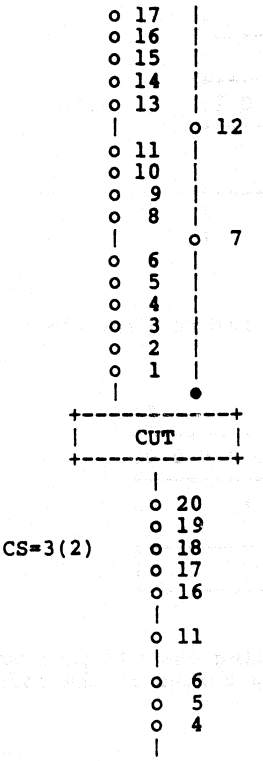


Figure 5-23 shows an example of using the CUT instruction to delete the first three arriving tokens at the node.

Figure 5-23



In this example, a count size of 3 has been set and the first three tokens arriving after initialization are deleted. Tokens after that point are not deleted but are output to the next cycle.

When FTRC=1 tokens (tokens 7 & 12 in Figure 23) arrive, the instruction is initialized and again the first three tokens arriving after initialization (tokens 8, 9, 10, 13, 14, 15) are deleted. The FTRC=1 token itself is also deleted.

CUT instruction assembler description

1. FUNCTION FCUT1 = CUT(size),temp
   size : Indicates the count size to be used by this instruction and is automatically set in FTR

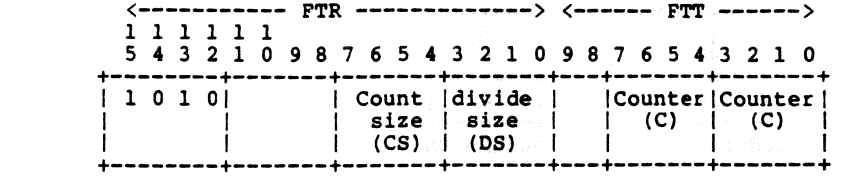
by the assembler. The value set in FTR, however, is the value specified in the assembler description minus 1.

temp : This value is used to set the contents (count) of FTT. The value specified as temp is set as is in FTT.

5.1.11 DIVCYC (Divide Cyclic) Instruction

This instruction divides the arriving tokens in a fixed ratio, into ID and ID+1 token flow.

DIVCYC instruction field format:

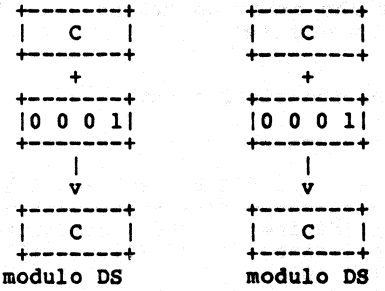


The FTR field stores the fixed data portion of this instruction and FTT the dynamic data.

1. The Op Code of the DIVCYC instruction is stored in the high order 4 bits of FTR.
2. The count size indicates the number of tokens that will arrive at the instruction before the destination ID of the output token is changed from ID to ID+1. In other words, if a count size of 3 is specified, the destination ID of the first three tokens arriving at the instruction will be ID and those from the fourth token to the number specified as the divide size will be ID+1.
3. The divide size indicates the number of tokens from the first token that arrives at the instruction until the destination ID field changes back from ID+1 to ID. For example, if a divide size of 5 is specified, the value of the destination ID from the first token to the token number specified in the count size will be ID; that from count size +1 to the fifth token will be ID+1. At this point the basic execution cycle of the DIVCYC instruction completes and the counter is clear to 0. The divide size therefore indicates the number of tokens from the first token until the completion of the basic cycle.
4. There are two 4-bit counters (C). The same data are to be set in these two counters. When rewriting the count by an FTRC=1 token, set the same count value in

each of the two counters.

- FTRC=0 tokens:
  As long as C ≤ CS, the ID field of the of the arriving tokens are output unchanged. If C > CS, the ID field of the arriving token is incremented and ID+1 is set in the ID field of the leaving token. The values of C and CS are compared and then C is incremented. When C = DS, C is clear to 0.



- FTRC=1 tokens:
  The lower order 8 bits of the data field of the arriving token are added to the value of C and the result is written to C. The data field of the arriving token must be configured so that the same value will be set in the two 4-bit counters. After this data has been written to C, the token is deleted.

  Note: The same value must be written to the two counters within FTT.

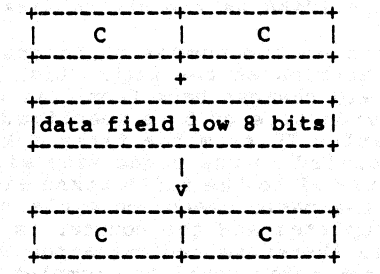
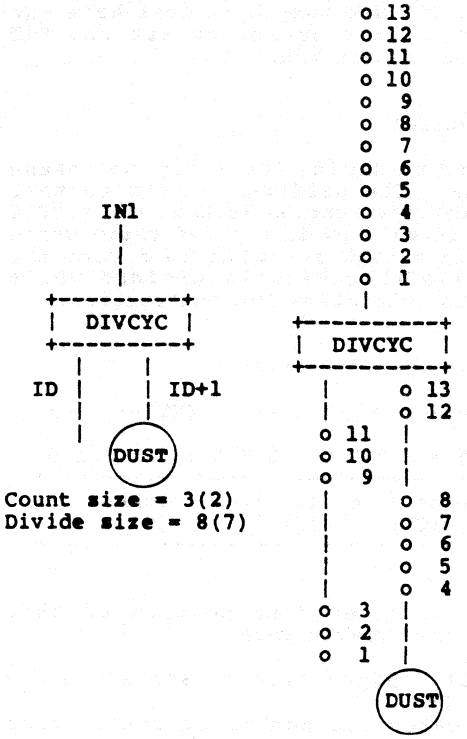


Figure 5-24 shows the use of the DIVCYC instruction to divide tokens arriving in a ratio of three valid tokens to five null tokens. The null tokens are deleted. To

specify this ratio, a count size of 3 and a divide size of 8 (3+5) must be specified in the assembler description.

Figure 5-24



In this example, tokens are successively input from IN1. The DIVCYC instruction (CS = 3(2), DS = 8(7)) outputs the first three tokens to ID and the next five tokens to ID+1. This operation repeats, dividing the tokens in a fixed ratio of 3:5. The tokens output to ID+1 after flow division are deleted in this example.

DIVCYC instruction assembler description:

1. FUNCTION FDIVC = DIVCYC(Csize,Dsize),temp
   Csize : Value of the count size used by this instruction.
   Dsize : Value of the divide size used by this instruction. Both of these values must be

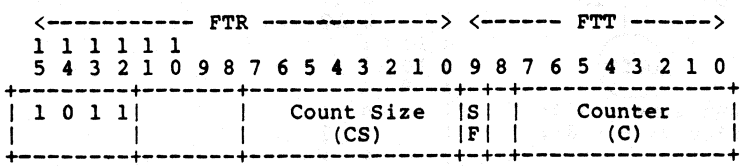
specified by the user when describing the DIVCYC instruction. The value of these parameters is then set by the assembler in FTR as count size = Csize -1; divide size = Dsize -1.

temp : This value is used to set the contents of FTT. In this instruction, there are two counters in FTT and they both must have the same value. For example, to set the FTT counter to 5, temp = 55H.

5.1.12 DIV (Divide) Instruction

This instruction is used to divide the arriving tokens into ID and ID+1 tokens. The difference between this instruction and the DIVCYC instruction is that the DIVCYC instruction divides the token flow in a fixed ratio while the DIV instruction uses a counter value to divide the tokens. The flow of FTRC=0 tokens is divided while FTRC=1 tokens are used to initialize the instruction.

DIVCYC instruction field format:



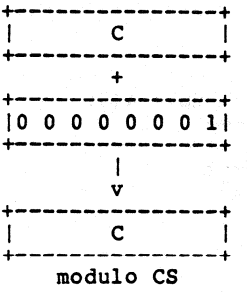
The FTR field stores the fixed data portion of this instruction and the FTT the dynamic data.

1. The Op Code of the DIV instruction is stored in the high order 4 bits of FTR.
2. The count size indicates the number of tokens that will arrive at the instruction before the destination ID of the output token is changed from ID to ID+1. In other words, if a count size of twenty is specified, the destination ID of the first through twentieth token will be ID and those from the twenty-first token will be ID+1.
3. The S/F bit is 0 as the initial condition and becomes 1 when count size = counter. From this point the destination ID of the output tokens become ID+1. The S/F bit is not reset even after the counter is cleared and the destination ID continues to be ID+1. An FTRC=1 token, however, can be used to initialize the instruction, resetting this bit as well as the counter. From this point, the operation of the DIV instruction is performed again from the beginning.

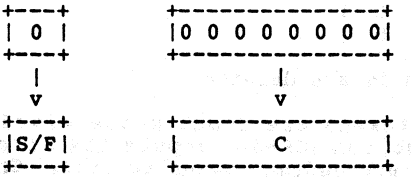
4. The counter counts the number of tokens arriving at the instruction. Each time a token arrives, the value of the counter is incremented.

- For FTRC=0 tokens:
  The initial condition is S/F = 0 and while C ≤ CS, the ID of the arriving token is output unchanged as the destination ID of the token and then C in incremented. When C = CS, the S/F bit is set to 1.

  When S/F = 0 and C > CS, or when S/F = 1, the ID field of the arriving token is incremented and output as ID+1. The S/F bit is not reset until an FTRC=1 token is input.

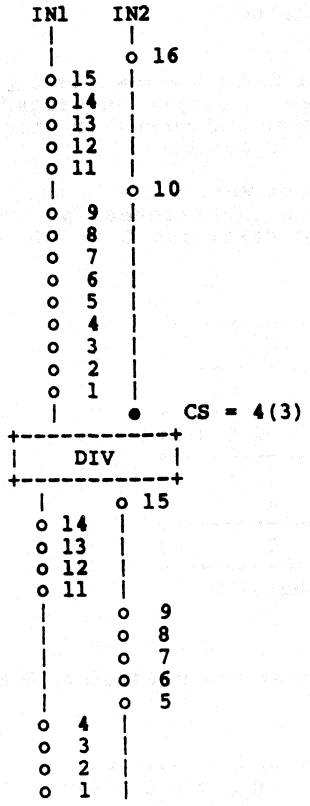


- FTRC=1 tokens:
  The S/F bit and the counter are cleared to 0 and then the token is deleted.



In the example shown in Figure 5-25, only the first four tokens are output to ID, then all tokens thereafter to ID+1. When an FTRC=1 token (tokens 10 and 16) is input, the instruction returns to the initial state and again the first four FTRC=0 tokens to arrive at the instruction (tokens 11, 12, 13, 14) are output to ID with subsequent tokens output to ID+1.

Figure 5-25



Tokens 10 and 16 are deleted

In this example, FTRC=0 tokens are input successively from IN1. The DIV instruction (CS=4) outputs the first four tokens to ID and all succeeding tokens to ID+1. The tenth token (token 10), however, is an FTRC=1 token that initializes the instruction and is then deleted. As the FTRC=0 tokens continue to arrive, the first four tokens (tokens 11, 12, 13, 14) are output to ID; those thereafter to ID+1.

DIV instruction assembler description:

1. FUNCTION FDIV1 = DIV(size),temp
   size : Indicates the count size to be used by this

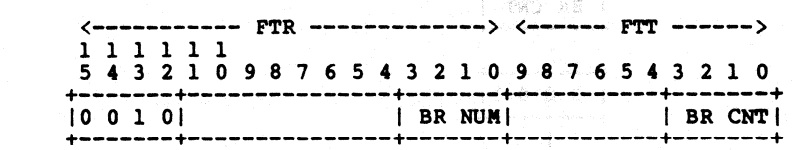
instruction and is automatically set in FTR by the assembler. The value set in FTR, however, is the value specified in the assembler description minus 1.

temp : This value is used to set the contents of FTT. Note that in addition to the counter, the S/F bit is stored in the MSB of FTT. This value specified as temp is set as is in FTT.

5.1.13 DIST (Distribution) Instruction

This instruction is used to divide the flow of the arriving tokens into several different IDs. There are also two kinds of arriving tokens: normal data tokens (FTRC=0) and ID control tokens (FTRC=1).

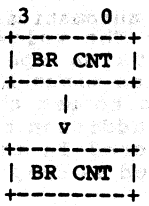
DIST instruction field format:



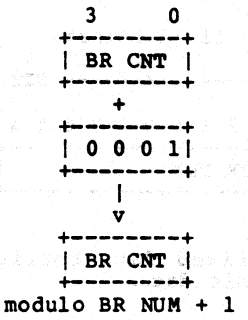
The FTR field stores the fixed data portion of this instruction and FTT the dynamic data.

1. The Op Code of the DIST instruction is stored in the high order four bits of FTR.
2. BR NUM indicates the maximum value that the original ID value can be increased. The values of the data token IDs are modified within the range ID to BR NUM.
3. BR CNT is the value added to the input token ID to generate the destination ID. BR CNT is incremented each time an FTRC=1 token arrives. When BR CNT = BR NUM, the arrival of the next FTRC=1 token will cause BR CNT to be reset to 0. The output ID is calculated and the token is output.

- For FTRC=0 tokens:
  The ID field of the output tokens are calculated as ID + BR CNT (where ID is the field of the arriving token); the BR CNT field is not changed.



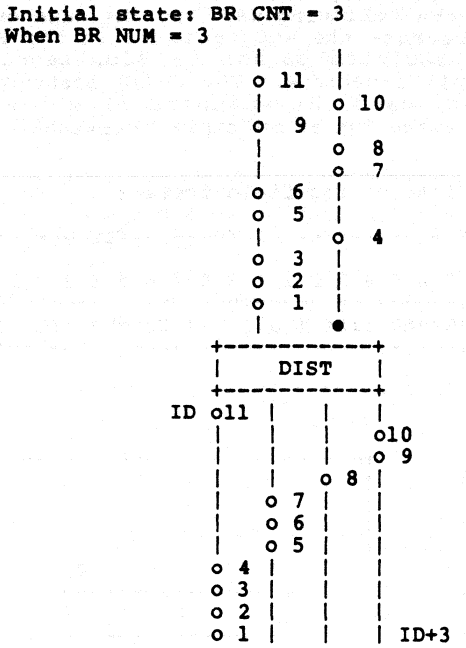
- FTRC=1 tokens:
  The ID field of the leaving token is calculated as ID + BR CNT. After the destination ID has been generated in this way, BR CNT is incremented. If, however, the value of BR CNT before it is incremented equals BR NUM, BR CNT is reset to 0.



In the following example, the arriving tokens are divided into four different IDs (ID, ID+1, ID+2, ID+3). The first event in Figure 5-26 is the arrival of three FTRC=0 tokens (tokens 1, 2, 3). The fourth token (token 4) is an FTRC=1 token. This token causes the next two (token 5 and 6) tokens to be output to ID+1. The seventh token (token 7) is also an FTRC=1 token and causes the eighth token to be output to ID+2. In other words, each time an FTRC=1 token arrives, it causes the ID of the output tokens to be incremented by 1. The FTRC=1 token itself, however, is output to the unincremented ID.

In this example, since BR NUM is 3. When FTRC=1 token 10 is input, the condition BR CNT = BR NUM is fulfilled and BR CNT is reset to 0. The next FTRC=0 token input (token 11) is therefore output to ID.

Figure 5-26



DIST instruction assembler description:

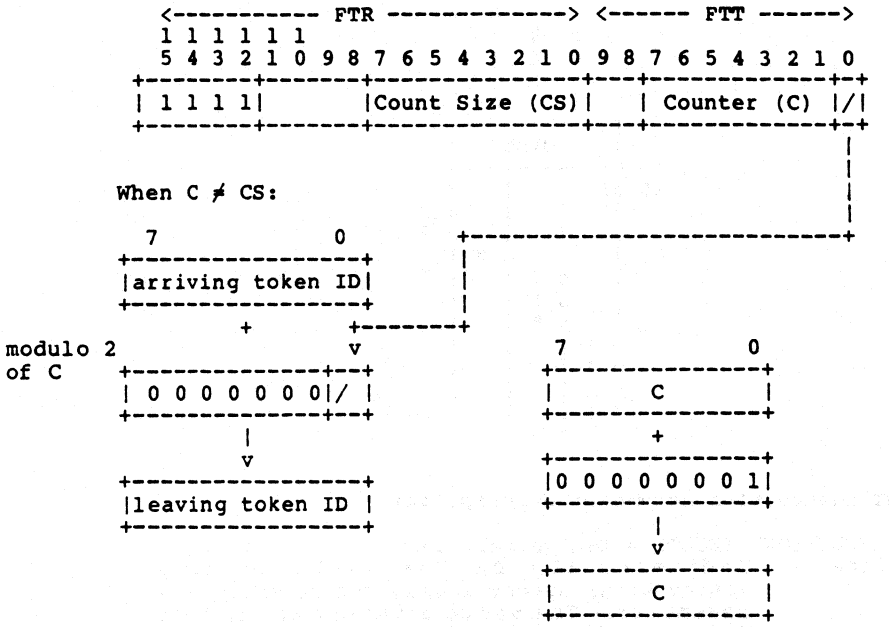
1. FUNCTION FDIST1 = DIST(size), temp
   size : Indicates the BR NUM field of this instruction; automatically set in FTR by the assembler. The value actually set in this field is size minus 1.
   temp : This value is used to set the contents of FTT. As the default value, 0 is set in this field. This parameter specifies the value of BR CNT and is set in FTT exactly as it is specified by the user.

5.1.14 CONVO (Convolution) Instruction

This instruction is used primarily with a PU instruction to perform cumulative operations such as the sum of a_i or product of a_i. The CONVO instruction can also be used effectively for special flow control.

The function of the ACC instruction resembles that of the CONVO instruction. However, the CONVO differs from the ACC instruction in the following way. In the case of the ACC instruction, because the ACC register in the PU is used, cumulative addition cannot be simultaneously performed at multiple locations. The CONVO instruction, on the other hand, uses the standard flow control technique enabling execution at multiple locations.

CONVO instruction field format:



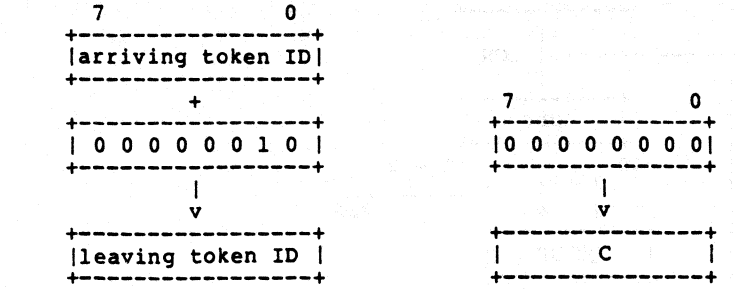
The FTR field stores the fixed data portion of this instruction and FTT the dynamic data.

1. The Op Code of the CONVO instruction is stored in the high order 4 bits of FTR.
2. The counter (C) counts the number of tokens arriving at this instruction. The value of C is incremented each time a token arrives. The least significant bit of this field is used to determine the ID field of the token sent out from this instruction.

The ID field of the token leaving this instruction is

generated by adding the ID field of the arriving token to the modulo 2 of the counter. In other words, if the initial value of the counter is 0 (even), the ID fields of the odd number tokens arriving at the node will be sent out unchanged and those of the even number tokens will be sent out as ID+1. This is because the value of the counter is incremented after each ID field is calculated.

- When CS = C:



The value 2 is added to the ID field of the arriving token to generate the ID field of the leaving token, in this case ID+2. The counter (C) is then cleared to 0.

In the following example, the operation $(a_i \cdot b_i)$ is performed. In figure 5-27, the series $a_1, a_2, \ldots, a_n$ is input from IN1 while the series $b_1, b_2, \ldots, b_n$ is input from IN2. The corresponding pairs of a_i and b_i are multiplied and the result is output to CON.

The tokens arriving at the CONVO instruction are divided into ID and ID+1 tokens. The ID tokens are output to the queue and added, then sent back to the CONVO instruction. In the case of

$$\sum_{i=1}^{n} (a_i \cdot b_i)$$

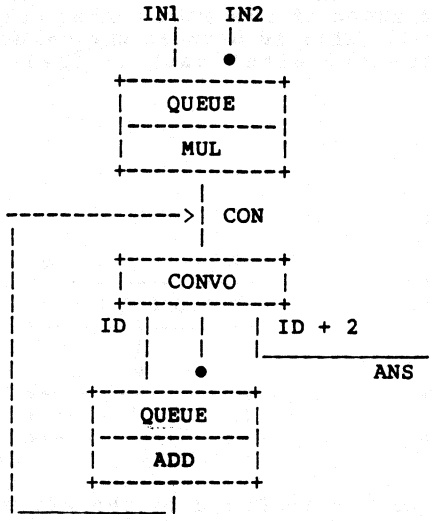
where n=11, the operation performed will be as follows. Here the product of a_i x b_i is expressed as c_i. In other words,

$$\sum_{i=1}^{n} c_i = c_1 + c_2 + c_3 + \ldots + c_n$$

or $((((((c_1 + c_2) + c_3) + c_4) + c_5) + \ldots) + c_{11})$. The relation between the count size (CS) and n is:

CS = 2n - 1

Figure 5-27



CONVO instruction assembler description:

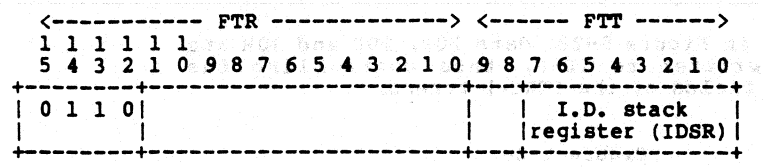
1. FUNCTION FCNV1 = CONVO(size),temp
   - size : Indicates the count size to be used by this instruction and is automatically set in FTR by the assembler. The value set in FTR, however, is the value specified in the assembler description minus 1.
   - temp : This value is used to set the contents of FTT. In this instruction, the counter is stored in FTT. The assembler sets the value of temp directly (without performing minus 1 subtraction) and this subtraction must therefore be done by the user before specifying temp.

5.1.15 SAVE (Save) Instruction

This instruction is used to set the physical address specified by the user as the destination ID. For example, to set address 30H as the destination ID (link table address) of the arriving tokens, first set 30H in

the ID stack register (IDSR) of FTT. This is done by inputting an FTRC=0 token with data 30H. Then when FTRC=1 tokens are input, the destination IDs of these tokens will be set to address 30H. This data remains in IDSR of FTT until it is rewritten by another FTRC=0 token.

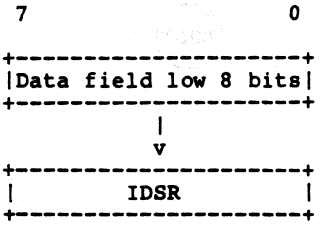
SAVE instruction field format:



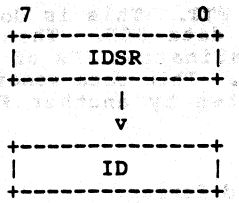
The FTR field stores the fixed data portion of this instruction and FTT the dynamic data.

1. The Op Code of the SAVE instruction is stored in the high order 4 bits of FTR.
2. The ID stack register (IDSR) is a register that stores the destination address data that replaces the destination ID of the arriving token. The value of this register is set by the input of an FTRC=0 token and the data in this register replaces the ID field of the arriving FTRC=1 tokens.

- For FTRC=0 tokens:
  The lower 8 bits of the data field of the arriving token are set in IDSR.

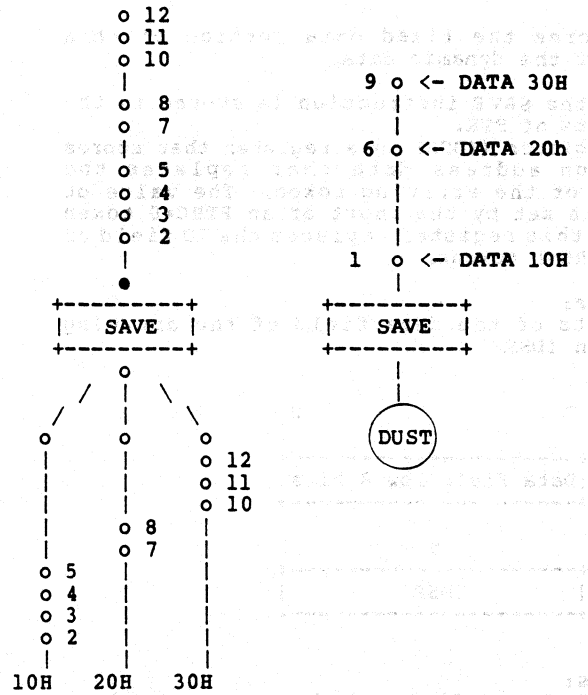


- For FTRC=1 tokens:
  The contents of the ID field of the arriving tokens are replaced by the contents of IDSR to generate the ID field of the token when it leaves the node.



In the example in Figure 5-28, data 10H, 20H and 30H are successively written to IDSR, thus controlling the destination ID fields of the FTRC=1 tokens.

Figure 5-28



First, an FTRC=0 token (token 1) with data 10H is input to the SAVE instruction to set IDSR. Data 10H is then set in the ID field of the FTRC=1 tokens (tokens 2 through 5) arriving at the instruction with the result

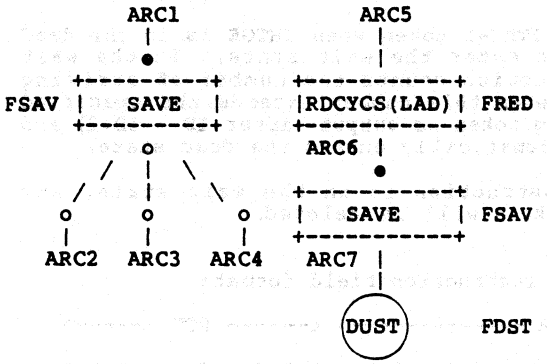
that these tokens are output to LT address 10H. The sixth input token (token 6) is FTRC=0 and carries data 20H. This data is set in IDSR. Note that the same address in the FT must be used for each of these input tokens because IDSR is stored in FTT.

Subsequent tokens (tokens 7 and 8) are output with data 20H as the destination ID field. In the same manner, an FTRC=0 token (token 9) sets data 30H in IDSR and causes tokens 10 through 12 to be output with ID destination data 30H. The FTRC=0 tokens that set the address (ID) data are deleted.

SAVE instruction assembler description:

1. FUNCTION FSAV = SAVE
   The operation of this instruction is controlled by two tokens: an FTRC=0 token and an FTRC=1 token. Unlike other instructions, however, the operation of SAVE is influenced by the address specification of the LINK statement. This relation is shown in Figure 5-29.

Figure 5-29



DUST uses the FTRC=1 COUNT token.

In this example, the flow of tokens arriving from ARC1 is directed to ARC2, ARC3 or ARC4. The arc to which the token will flow is determined by the data set beforehand by the token arriving from ARC6.

```
LINK      ARC2 = FSAV (ARC1);
LINK      ARC3 = FSAV (ARC1);
LINK      ARC4 = FSAV (ARC1);
LINK      ARC6 = FRED (ARC5);
```

```
LINK      ARC7 = FSAV (,ARC6);
LINK           = FDST (,ARC7);

FUNCTION  FSAV = SAVE, ARC2;
FUNCTION  FRED = RDCYCS (LAD, 3);
FUNCTION  FDST = COUNT (1);

MEMORY    LAD = ARC2, ARC3, ARC4
```

5.1.16 CNTGE (Count Generation) Instruction

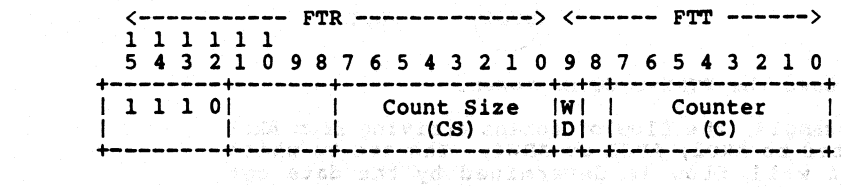
This instruction is normally used in combination with the GE instruction COPYBK (described in a later section). The maximum number to tokens that can be copied by the COPYBK instruction is 16. Therefore, when more tokens than this are to be generated, the CNTGE instruction is combined with the COPYBK instruction.

A feature of this instruction is that it has a 'dead' state and a 'wait' state. The dead state can be thought of as the initial state and in this state, the instruction performs no operation. The wait state indicates that the instruction is performing an operation.

The arrival of an FTRC=1 token when CNTGE is in the dead state causes it to enter the wait state. In the wait state, this instruction counts the number of arriving FTRC=0 tokens and when this number exceeds the specified value, the arriving token is output (after ID - ID+2) and the instruction automatically enters the dead state.

When the CNTGE instruction is in the wait state, any arriving FTRC=1 tokens will be deleted.

CNTGE instruction field format:



The FTR field stores the fixed data portion of this instruction and FTT the dynamic data.

1. The Op Code of the CNTGE instruction is stored in the high order 4 bits of FTR.
2. The count specified by the user is set in the count

size field. The value actually stored, however, is the value specified in the assembler minus 1.

3. The W/D bit indicates whether the instruction is operating (wait state) or not operating (dead state).
   W/D = 0 : dead state
   W/D = 1 : wait state
4. The counter field is used to count the number of tokens arriving at this instruction. Each time a token arrives, the value of this field is incremented by one.

The processing performed by this instruction varies depending on the state of the W/D bit, the counter (C), and the FTRC bit of the arriving token. When W/D=0 (dead state), the arrival of an FTRC=1 token causes the W/D bit to be set to 1 and the instruction to enter the wait state. Then C is incremented. Thereafter, C is incremented each time an FTRC=0 token arrives. When C = CS, the instruction enters the dead state (W/D = 0) and C is cleared to 0.

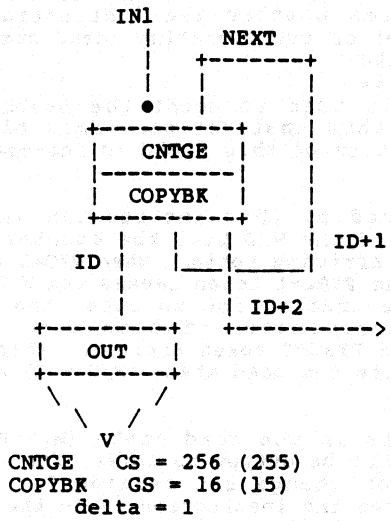
When the instruction is in the dead state (W/D=0) an arriving FTRC=0 token will be output to ID+2. The value of the W/D bit does not change but remains 0. If a FTRC=1 tokens arrives when the instruction is in the wait state (W/D=1), that token is deleted. The value of the W/D bit does not change. The transitions between these states is shown in the following table.

Table 5-2

Arrival				Leaving		
W/D	C=CS ?	FTRC bit arriving token	ID of arriving token	W/D	C	ID of arriving token
0	Yes	1	ID	0	+0	
	No			1	+1	ID
	don't care	0	ID	0	+0	ID+2
1	don't care	1	ID	1	+0	delete
	Yes	0	ID	0	cleared	ID
	No	0	ID	1	+1	ID

Figure 5-30 shows the CNTGE and COPYBK instructions used in combination to generate external memory addresses.

Figure 5-30



The first operation that is performed here is to input an FTRC=1 token to IN1. The CNTGE instruction (which had been in the dead state) enters the wait state and operation begins).

Assume that an FTRC=1 token with data 0 arrives from IN1. The CNTGE instruction enters the wait state and sends a token with data 0 to the COPYBK instruction. At the COPYBK instruction, the 0 token is copied 16 times with delta = 1. The 17th token is then output (with ID+1) to arc NEXT. Because delta = 1, the data of the tokens output to ID will be 0, 1, 2, 3, ..., 14, 15, and that of the token output to ID + 1 will be 16. This token is an FTRC=0 token and is input again from ID+1 to CNTGE.

Since an FTRC=1 token has already been input to CNTGE, it is in the wait state and once again sends a token with data 16 to the COPYBK instruction. The COPYBK instruction then again copies this token (data field = 16) 16 times with delta = 1 and outputs a token with a data field of 32 to ID+1. In this way, tokens with incremental (by 1) data fields can be generated in units of 16.

When a data field of 4096 (16 x 256) is output to ID+1, and the last token specified by CNTGE (in this case the 256th token) arrives, the instruction outputs an ID+2 token instruction instead of an ID token. In other

words, in the example shown above, where CS CNTGE = 256, GS of COPYBK = 16, and delta = 1, and where the data field of the start up token is 0, tokens with data fields from 0 to 4095 will be generated.

CNTGE instruction assembler description:

1. FUNCTION FGE1 = CNTGE(size),temp
   - size : Indicates the count size to be used by this instruction and is automatically set in FTR by the assembler. The value set in FTR, however, is the value specified in the assembler description minus 1.
   - temp : This value is used to set the contents of FTT. Note that in addition to the counter, the W/D bit is stored in the MSB of FTT.

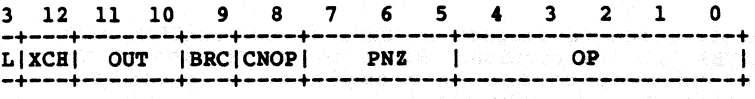
This instruction is frequently used in combination with the COPYBK (GE) instruction. Refer to the section on the COPYBK instruction for an example of an assembler description of this combination.

5.2 PU (Processing Unit) Instructions

These instructions use the PU (processing unit) to perform operations on the data field of arriving tokens. Among the types of operations that can be performed are arithmetic and logical operations, bit manipulations, comparisons, etc.

PU instructions are selected by the SEL bits of LT (SEL=01) and are stored in the FTL field of the FT indicated by FTA (function table address) of LT. When a PU instruction arrives at the FT block, it read bits 0 to 11 of FTL, incorporates this data as part of its own token and proceeds to the PU block.

FTL Field



F/L : Full/Left
XCH : Exchange
OUT : Output
BRC : Branch Control
CNOP : C-bit NOP
PNZ : Positive Negative Zero
OP : Operation

Bits 12 and 13 (XCH and F/L) are deleted before this token arrives at the PU block.

- F/L bit:
  This bit determines whether the PU instruction is to be executed independently or in combination with an AG & FC instruction.

  When F/L=0, the PU instruction is executed independently to perform one-operand operations; only FTL is significant.

  When F/L=1, it indicates that a linked PU instruction is to be executed to perform two-operand operations; both FTL and FTR are significant. The instruction written in FTR (the instruction to be linked) must be an AG & FC instruction.

  For example, when the PU instruction INC is used, the operation performed is a single-operand type and the PU instruction is executed independently (only the instruction indicated by FTL is executed). At this time, the SEL bits of the LT are '01' and the F/L bit is '0' (Left). To perform two-operand operations (such as ADD or MUL instructions) at the PU, an AG & FC instruction is used in combination with the PU instruction and both FTR

and FTL are significant. At this time SEL=01 and F/L=1 (full).

Table 5-3

LT SEL Field	(FTL)	(FTR)	(FTT)
SEL=00	OUT Instruction	When F/L = 1, AG & FC instruction; When F/L=0, ignored*	Used as temporary field by AG & FC instruction
SEL=01	PU Instruction	Same as above	Same as above
SEL=10	GE Instruction	Same as above	Same as above
SEL=11	Ignored* Instruction	AG & FC	Same as above

* Because this setting is ignored, the operations of FTL and FTR are completely independent of one another and it is possible to execute two instructions independently (provided that F/L = 0). The result is that the FTL memory area can be used more efficiently. The uPD7281 assembler automatically compresses these fields when independent FTL (PU, GE or OUT) instructions exist together in the same module (same uPD7281) with independent FTR (AG & FC) instructions. There is normally no need for the user to take this operation into consideration. The user must, however, be aware of this fact when performing operations such as program area dump.

- Control performed by F/L bit and example flowgraphs:
  In the example in Figure 5-31, the data arriving from ARC1 (Data 1) is incremented and the result is output to ARC2 (Data 2). Here, because the PU instruction is executed independently, F/L = 0.

Figure 5-31

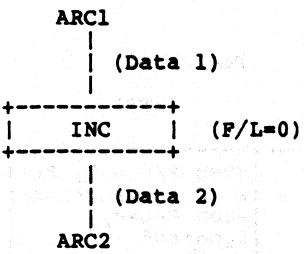
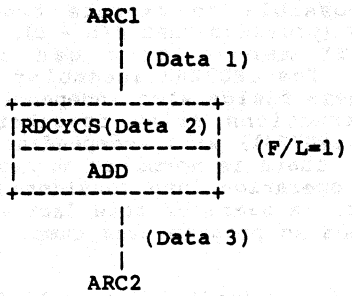


Figure 5-32 shows a two-operand operation in which the token input from ARC1 (Data 1) reads data from the DM (Data 2) and performs a two-operand operation (addition) at the PU. Because the AG & FC instruction RDCYCS is used as the linking instruction, F/L = 1 and both the PU instruction stored in FTL and the AG & FC instruction stored in FTR are valid.

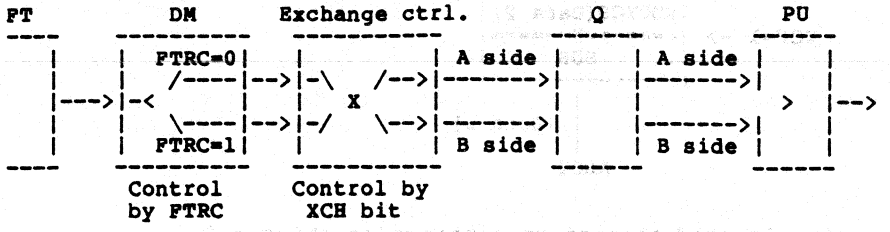
Figure 5-32



- XCH bit:
  This bit controls the exchange operation that can be performed at the exit of the DM between the memory data (read from DM) and the bus data (brought by the tokens coming from the LT and FT). Normally, at the entry to the PU the data of the tokens coming from the LT and FT are handled as A side data and that read from the DM as B side data. To exchange the A and B side data before sending them to the PU, the control performed by this bit is used. (Subtraction and shift operation are examples of when this exchange would be desirable.) When exchange is performed in conjunction with a read instruction, the operation performed is a described above. Note, however, that when a QUEUE instruction is used, the output will

also be affected by the condition of the FTRC bits of the tokens. The concepts involved here are illustrated in Figure 5-33.

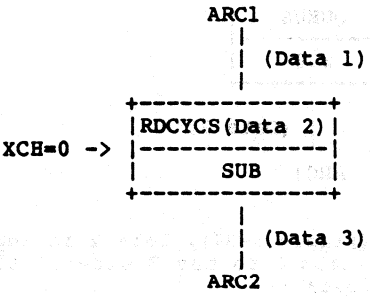
Figure 5-33



(The FTRC control and the XCH control are actually performed simultaneously)

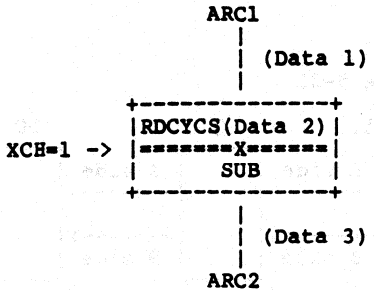
- Control performed by XCH bit and example flowgraph:
  In the example in Figure 5-34, the XCH bit is not set. The relation here between the data arriving from ARC1 (Data 1), that read from the DM (Data 2) and the output data (Data 3) is as follows: Data 1 is input to the A side of the PU and Data 2 to the B side. The results is: (Data 1) - (Data 2) = (Data 3)

Figure 5-34



When the XCH bit is set (Figure 5-35), Data 2 is input to the A side of the PU and Data 1 to the B side. The result is: (Data 2) - (Data 1) = (Data 3)

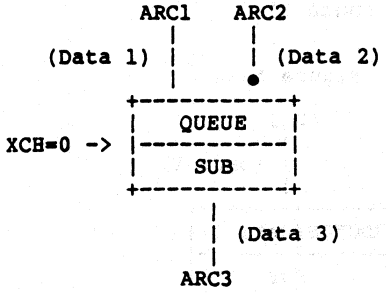
Figure 5-35



When queuing is used to perform subtraction (Figure 5-36), the FTRC bit is used to determine which data is from the bus side and which data is from the memory side. In other words, Data 1 is input to the A side of the PU (because FTRC=0) and Data 2 is input to the B side (because FTRC=1). Here the result will be:

(Data 1) - (Data 2) = (Data 3)

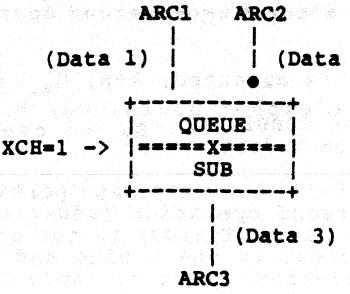
Figure 5-36



When the XCH bit is set (Figure 5-37), Data 2 is input to the A side of the PU and Data 1 to the B side of the PU.

(Data 2) - (Data 1) = (Data 3)

Figure 5-37



- OUT:
  These two bits control the output of the result calculated in the PU. The format of the token output from the PU will be one of the following: X output, Y output, XY output or XX output.

  The data output from X and Y will vary depending on the type of operation performed. For example, when addition (ADD) is performed, the 16 bit result is output to the X side and the carry bit is output to the Y side. The relation between the OUT bits and the output format is shown below.

Table 5-4
OUT Bits and Output Format

OUT bits	No. of outputs	First Output ID	First Output DATA,C,S	Second output ID	Second output DATA,C,S
00	1	ID	X (note 1)	-	-
01	1	ID	Y (note 2)	-	-
10	2	ID	X	ID+1	X
11	2	ID	X	ID+1	Y

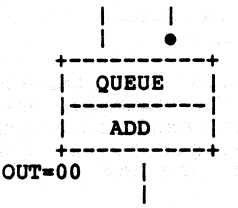
Notes:
1. This is the 18 bit result of the operation output to the X side. It includes the C_X and S_X bits.
2. This is the 18 bit results of the operation output to the Y side. It includes the C_Y and S_Y bits. The output format is also affected by the state of the BRC bit (described in a later section).

- Control performed by OUT bits and example flowgraphs:
  The following examples illustrate the operation performed when the result is output after a two-operand operation (ADD).

  When the ADD instruction is executed, A+B, C_x, S_x are output as the X data and either 0001H, C_y, S_y (A+B generated an overflow) or 0000H, C_y, S_y (no overflow generated) are output as the Y data.

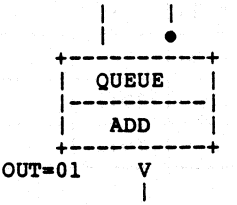
  In the example in Figure 5-38, only the data portion of the result of the two-operand operation (addition) is output to the X side; the carry (Y side) is not output. (For the types of data output to the X side and the Y side as the result of operation, refer to Table 5-8 PU instructions.)

Figure 5-38



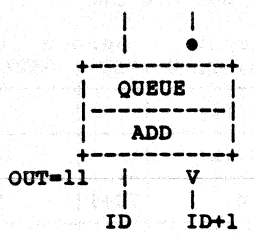
In the example in Figure 5-39, only the carry portion of the result of the operation is output.

Figure 5-39



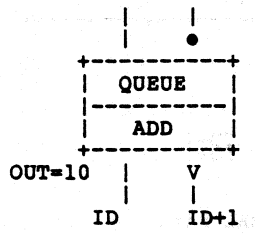
In the example in Figure 5-40, both X side (data) and the Y side (carry) of the result of the operation are output (the first output is to the X side and the second to the Y side).

Figure 5-40



In the example shown in Figure 5-41, the result of the operation is output twice to the X (data) side. The destination IDs used are ID and ID+1.

Figure 5-41



- BRC:
  This bit causes the output after PU operation to be controlled by the condition of the C bit. (Whether the C_X or C_Y bit is used is determined by the OUT bits.) If the C bit of the output data is set (to 1) by the operation, the destination ID will be ID+1, and if C=0, the destination ID will be ID.

  When this bit is used (BRC=1), a single ouput (either X or Y) must be specified by the OUT field. If the OUT field does not specify a single output, the operation by the BRC bit (division of the data flow into ID and ID+1 arcs) cannot be performed.

  The output control by the OUT field and the BRC bit is shown in the following table. The setting of the C bit can be specified by the PNZ field so that the operation result can be tested and used to control the flow of program execution.

Table 5-5
Output Control by OUT and BRC

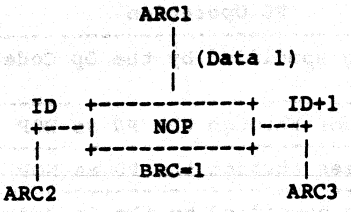
OUT	BRC	No. of outputs	First Output ID	First Output DATA, C, S	Second Output ID	Second Output DATA, C, S
00	0	1	ID	X (1)	-	-
01	0	1	ID	Y (2)	-	-
10	0	2	ID	X	ID+1	X
10	1	Use Prohibited				
11	0	2	ID	X	ID+1	Y
11	1	Use Prohibited				
00	1	1	(3)	X	-	-
01	1	1	(4)	Y	-	-

Notes:
1. X: $DATA_X$, C_X, S_X
2. Y: $DATA_Y$, C_Y, S_Y
3. When C_X=0, ID; when C_X=1, ID+1
4. When C_Y=0, ID; when C_Y=1, ID+1

- Control of PU data flow by the BRC bit:
  1. The operation specified by the Op Code is performed.
  2. The C bit is set by PNZ (default: 000) to reflect the result. The function of the PNZ field is described in detail in a later section.
  3. When the BRC bit is 1, if the C bit is 1, the result is output to ID+1 (second output); if the C bit is 0, the result is output to ID (first output).

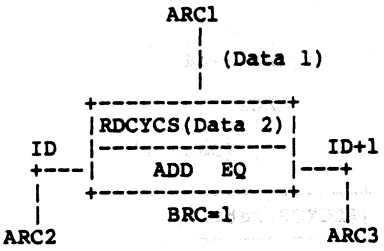
- Control performed by the BRC bit and example flowgraphs:
  In the example in Figure 5-42, if the C bit of the data arriving from ARC1 is 1, the token flow will branch to ARC3 and if it is 0, flow will branch to ARC2. The condition of the C bit output from this node is the same as that input.

Figure 5-42



In the example in Figure 5-43, the data arriving from ARC1 (Data 1) is added to that read from the DM (Data 2) and then PNZ judgment is performed. Here PNZ has been set to EQ (equal) so that if the result of the addition equals 0, the C bit will be set to 1. Next, the BRC bit tests the C bit and determines the output destination (ID or ID+1) of the result.

Figure 5-43



- CNOP:
  This bit controls whether or not the PU instruction is to be executed. This determination is based on the coincidence (=) or noncoincidence (≠) of the C bits of the A and B side data (C_A and C_B).

- When CNOP=1:
  The operation specified by the Op Code is performed when $C_A=C_B$ and the operation is not performed (the token passes PU as NOP) when $C_A \neq C_B$.

- When CNOP=0:
  The operation specified by the Op Code is performed irrespective of the condition of the C bits.

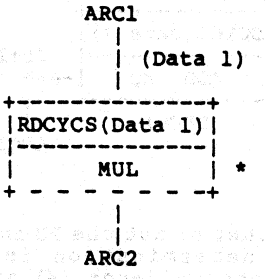
The operation of the PU as controlled by C_A and C_B is shown in Table 5-6.

Table 5-6

CNOP	CA	CB	PU Operation
1	0	0	Processing specified by the Op Code is performed.
	0	1	Token passes through the PU as NOP
	1	0	Token passes through the PU as NOP
	1	1	Processing specified by the Op Code is performed
0	x	x	Processing specified by the Op Code is performed

- Control performed by the CNOP bit and example flowgraphs:
  In the example shown in Figure 5-44, the multiplication operation (MUL) is performed when the C bits of the data arriving from ARC1 (Data 1) and that read from DM (Data 2) are the same.

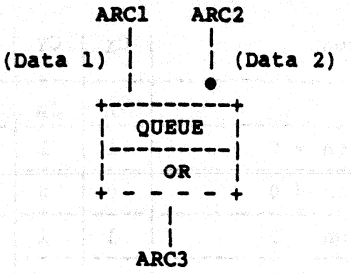
Figure 5-44



In the same manner, in the example in Figure 5-45, the OR logical operation is performed (or not) according to the coincidence/noncoincidence of the C bits of Data 1 and Data 2.

* Nodes enclosed in dotted lines are CNOP=1. (See Appendix A)

Figure 5-45



- PNZ (Positive, Negative, Zero)
  The PNZ field is used to set conditions that judge whether the result of the operation specified by the Op Code is positive, negative or zero, or if an overflow has occurred. As a result of this judgement, the C bit field of the PU output data is set/reset.

Table 5-7
Control by PNZ

P N Z	Condition	Cx	Cy	Assembler Description	
0 0 0	No condition set	C_A	C_B		
0 0 1	Result of operation = 0	1	1	EQ	True
	Result of operation ≠ 0	0	0		False
0 1 0	Result of operation < 0	1	1	LT	True
	Result of operation ≥ 0	0	0		False
0 1 1	Result of operation ≤ 0	1	1	LE	True
	Result of operation > 0	0	0		False
1 0 0	Result of operation > 0	1	1	GT	True
	Result of operation ≤ 0	0	0		False
1 0 1	Result of operation ≥ 0	1	1	GE	True
	Result of operation < 0	0	0		False
1 1 0	Result of operation ≠ 0	1	1	NE	True
	Result of operation = 0	0	0		False
1 1 1	Overflow generated	1	1	OVF	True
	No overflow generated	0	0		False

PNZ is also used with comparison instructions (CMPNOM, CMP and CMPXCH) in which case PNZ expresses the comparison condition. The contents of output data X and Y are determined by the result of this judgement. PNZ=000 and PNZ=111 may not be specified for comparison instructions. Therefore, the user may not specify PNZ for such instructions as COPYC, ACC, CVT2AB, CVTAB2, ORMSK and ANDMSK. PNZ judgement is normally performed in relation to the X side data. In the case of multiplication or adjustment of double precision values, however, it is performed for 32 bits of data plus the sign bit.

- OP

These five bits are the Op Code for the operation to be performed within the PU. Tables 5-8 through 5-13 shows the Op Codes and the corresponding mnemonics.

Table 5-8 PU Instruction (1/6)

Item	Mnemonic	OP code	Input						Output						Notes
			CA	SA	DATAA	CB	SB	DATAB	CX	SX	DATAX	CY	SY	DATAY	
Logical Operations	OR	00000	CA	SA	A	CB	SB	B	CX	SA	AVB	CY	0	0000H	
	AND	00001	CA	SA	A	CB	SB	B	CX	SA	A∧B	CY	0	0000H	
	XOR	00010	CA	SA	A	CB	SB	B	CX	SA	A∨B	CY	0	0000H	
	ANDNOT	00011	CA	SA	A	CB	SB	B	CX	SA	$\bar{A}$∧B	CY	0	0000H	
	NOT	01100	CA	0	A	-	-	-	CX	SA	$\bar{A}$	CY	0	0000H	
Arithmetic Operations	ADD	11000	CA	0	A	CB	0	B	CX	0	A+B	CY	0	*	
			CA	0	A	CB	1	B	CX	0	A-B	CY	0	0000H	When A>B,SX=0
										1	B-A		1		When A<B,SX=1
			CA	1	A	CB	0	B	CX	0	B-A	CY	0	0000H	When A<B,SX=0
										1	A-B		1		When A>B,SX=1
			CA	1	A	CB	1	B	CX	1	A+B	CY	1	*	
	ADDSC	11100	CA	0	A	CB	0	B	CX	0	A+B	CY	Ss	No. of shifts†	
			CA	0	A	CB	1	B	CX	0	A-B	CY	Ss	*	When A>B,SX=0
										1	B-A			No. of shifts†	When A<B,SX=1
			CA	1	A	CB	0	B	CX	0	B-A	CY	Ss	*	When A<B,SX=0
										1	A-B			No. of shifts†	When A>B,SX=1
			CA	1	A	CB	1	B	CX	1	A+B	CY	Ss	No. of shifts†	

* If an overflow occurs as the result of A+B, DATAY=0001H and if no overflow, DATAY=0000H.

† This indicates the number of consecutive zeros from the MSB of DATAX. This number is used to calculate the number of shifts to be performed by subsequent processing.

Table 5-9 PU Instruction (2/6)

Item	Mnemonic	OP code	Input						Output						Notes
			CA	SA	DATAA	CB	SB	DATAB	CX	SX	DATAX	CY	SY	DATAY	
Arithmetic operations	SUB	11001	CA	0	A	CB	0	B	CX	0	A-B	CY	0	0000H	When A>B,SX=0
										1	B-A		1		When A<B,SX=1
			CA	0	A	CB	1	B	CX	0	A+B	CY	0	*	
			CA	1	A	CB	0	B	CX	1	A+B	CY	1	*	
			CA	1	A	CB	1	B	CX	0	B-A	CY	0	0000H	When A<B,SX=0
										1	A-B		1		When A>B,SX=1
	SUBSC	11101	CA	0	A	CB	0	B	CX	0	A-B	CY	SS	No. of shifts†	When A>B,SX=0
										1	B-A				When A<B,SX=1
			CA	0	A	CB	1	B	CX	0	A+B	CY	SS	No.of shifts†	
			CA	1	A	CB	0	B	CX	1	A+B	CY	SS	No.of shifts†	
			CA	1	A	CB	1	B	CX	0	B-A	CY	SS	No. of shifts†	When A<B,SX=0
										1	A-B				When A>B,SX=1
	MUL	11010	CA	SA	A	CB	SB	B	CX	SX	A B High	CY	SX	AXB Low	SX=SA V SB
	MULSC	11110	CA	SA	A	CB	SB	B	CX	SX	A B High	CY	SS	No. of shifts†	SX=SA V SB
	NOP	11011	CA	SA	A	CB	SB	B	CX	SA	A	CY	SB	B	
	NOPSC	11111	CA	SA	A	CB	SB	B	CX	SA	A	CY	SS	No. of shifts†	

* If an overflow occurs as the result of A+B, $DATA_Y$=0001H and if no overflow, $DATA_Y$=0000H.

† This indicates the number of consecutive zeros from the MSB of DATAX. This number is used to calculate the shifts to be performed by subsequent processing.

Table 5-10 PU Instruction (3/6)

Item	Mnemonic	OP code	Input						Output						Notes
			CA	SA	DATAA	CB	SB	DATAB	CX	SX	DATAX	CY	SY	DATAY	
Arithmetic operations	INC	01010	CA	0	A	-	-	-	CX	0	A+1	CY	0	*	
			CA	1	A	-	-	-	CX	0	1	CY	0	0000H	When A=0, SX=0
										1	A-1		1		When A>1, SX=1
	DEC	01011	CA	0	A	-	-	-	CX	0	A-1	CY	0	0000H	When A>1, SX=0
										1	1		1		When A=0, SX=1
			CA	1	A	-	-	-	CX	1	A+1	CY	1	*	
Shift	SHL	00100	CA	SA	A	CB	0	No.of shifts	CX	SA	Shift A right	CY	SA	Shift A right	
			CA	SA	A	CB	1	No.of shifts	CX	SA	Shift A right	CY	SA	Shift A left	
	SHLBRV	00101	CA	SA	A	CB	0	No.of shifts	CX	SA	Invert A shift left	CY	SA	Invert A Shift right	
			CA	SA	A	CB	1	No.of shifts	CX	SA	Invert A shift right	CY	SA	Invert A shift left	
	SHR	00110	CA	SA	A	CB	0	No.of shifts	CX	SA	Shift A right	CY	SA	Shift A left	
			CA	SA	A	CB	1	No.of shifts	CX	SA	Shift A left	CY	SA	Shift A right	
	SHRBRV	00111	CA	SA	A	CB	0	No.of shifts	CX	SA	Invert A shift right	CY	SA	Invert A shift left	
			CA	SA	A	CB	1	No.of shifts	CX	SA	Invert A shift left	CY	SA	Invert A shift right	

* When the result of A+1 generates an overflow, DATAY=0001H and when no overflow is generated, DATAY=0000H.

Table 5-11 PU Instruction (4/6)

Item	Mnemonic	OP code	Input						Output						Notes
			CA	SA	DATAA	CB	SB	DATAB	CX	SX	DATAX	CY	SY	DATAY	
Comparison	CMPNOM	01000	CA	SA	A	CB	SB	B	0	0	0000H	0	0	0000H	When PNZ is false
			CA	SA	A	CB	SB	B	1	0	0001H	1	0	0000H	When PNZ is true
	CMP	01001	CA	SA	A	CB	SB	B	0	SA	A	0	SB	B	When PNZ is false
			CA	SA	A	CB	SB	B	1	SA	A	1	SB	B	When PNZ is true
	CMPXCH	10001	CA	SA	A	CB	SB	B	CA	SA	A	CB	SB	B	When PNZ is true
			CA	SA	A	CB	SB	B	CB	SB	B	CA	SA	A	When PNZ is false

Use of PNZ with Comparison Instructions

PNZ	Condition	True/False	Note
001	SA, DATAA = SB, DATAB	True	Equal
	SA, DATAA ≠ SB, DATAB	False	Not Equal
010	SA, DATAA < SB, DATAB	True	Less Than
	SA, DATAA ≥ SB, DATAB	False	Greater or Equal
011	SA, DATAA ≤ SB, DATAB	True	Less or Equal
	SA, DATAA > SB, DATAB	False	Greater Than
100	SA, DATAA > SB, DATAB	True	Greater Than
	SA, DATAA ≤ SB, DATAB	False	Less or Equal
101	SA, DATAA ≥ SB, DATAB	True	Greater or Equal
	SA, DATAA < SB, DATAB	False	Less Than
110	SA, DATAA ≠ SB, DATAB	True	Not Equal
	SA, DATAA = SB, DATAB	False	Equal

Use of PNZ=000 or PNZ=111 for comparison instructions is prohibited.

Table 5-12 PU Instruction (5/6)

Item	Mnemonic	OP code	Input						Output						Notes
			CA	SA	DATAA	CB	SB	DATAB	CX	SX	DATAX	CY	SY	DATAY	
	ACC	10010	CA	SA	A	CB	SB	B	CX	SX	A	--	--	--	Used as a pair with AG & FC instruction COUNT
	COPYC	10011	CA	SA	A	CB	SB	B	CA	SA	A	CA	SB	B	
Bit operation	GET1	10101	CA	SA	A	CB	SB	Bit position	CX	SA	0000H	CY	0	0000H	When Abit=0, specified by lower 4 bits of B
											0001H				When Abit=1, specified by lower 4 bits of B
	SET1	10110	CA	SA	A	CB	SB	Bit position	CX	SA	Set Abit	CY	0	0000H	Bit specification by lower 4 bits of B
	CLR1	10111	CA	SA	A	CB	SB	Bit position	CX	SA	Clear Abit	CY	0	0000H	Bit specification by lower 4 bits of B
Bit check	ANDMSK	01101	CA	SA	A	CB	SB	B	0	SA	A	0	SB	B	$\bar{A} \wedge B = 0$
									1	SA	A	1	SB	B	$\bar{A} \wedge B = 0$
	ORMSK	10000	CA	SA	A	CB	SB	B	0	SA	A	0	SB	B	$A \wedge B = 0$
									1	SA	A	1	SB	B	$A \wedge B = 0$
Data conversion	CVT2AB	01110	CA	SA	A	CB	SB	B	CX	SX	Converted A data	CY	0	0000H	Absolute value-Twos complement
	CVTAB2	01111	CA	SA	A	CB	SB	B	CX	SX	Converted A data	CY	0	0000H	Twos complement-absolute value

Table 5-13 PU Instruction (6/6)

Item	Mnemonic	OP code	Input						Output						Notes
			CA	SA	DATAA	CB	SB	DATAB	CX	SX	DATAX	CY	SY	DATAY	
Adjustment of double precision numbers	ADJL	10100	CA	0	A	CB	1	B	CX	0	A-1	CY	0	0000H-B	A ≠ 0 and B ≠ 0
			CA	1	A	CB	0	B	CX	1	A-1	CY	1	0000H-B	A ≠ 0 and B ≠ 0
			CA	0	A	CB	1	0000H	CX	0	A	CY	0	0000H	
			CA	0	0000H	CB	1	B	CX	1	0000H	CY	1	B	B ≠ 0
			CA	1	A	CB	0	0000H	CX	1	A	CY	1	0000H	
			CA	1	0000H	CB	0	B	CX	0	0000H	CY	0	B	B ≠ 0
			CA	0	A	CB	0	B	CX	0	A	CY	0	B	
			CA	1	A	CB	1	B	CX	1	B	CY	1	B	

- Mnemonics and the uPD7281 Assembler

As the descriptions thus far should make evident, the PU instructions, in addition to performing the operations indicated by their respective Op Codes (FTL field), can also be used to perform such operations as branching and PNZ processing. The parameters for these operations are specified in the manner shown below.

```
FUNCTION  FADD1 = ADD( ), QUEUE(QUE1,1);
                      ↑
```

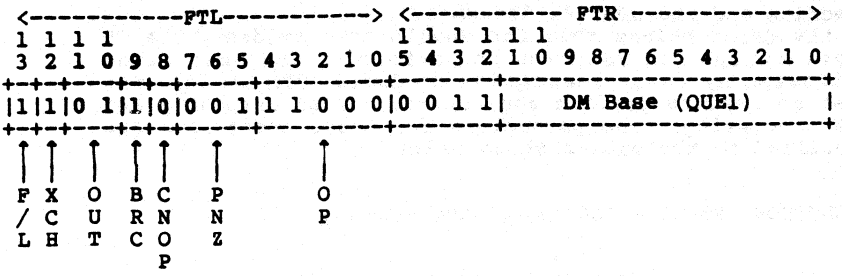
Specify the parameters within these parenthesis (order fixed)

Table 5-14
Parameter List

Bits to be specified		Mnemonic	Default	Notes
XCH bit		XCH	XCH = 0	
BRC bit		BRC	BRC = 0	
P	Omit	no specificat.	No condition specified PNZ = 000	The conditions vary for comparison instructions. Cannot be specified for some instructions. Refer chapter on instructions
	= 0	EQ		
N	< 0	LT		
	≤ 0	LE		
Z	> 0	GT		
	≥ 0	GE		
	≠	NE		
	overflow	OVF		
CNOP bit		CNOP	CNOP = 0	
O	X output	X	OUT = 00 X output	
U	Y output	Y		
T	XX output	XX		
	XY output	XY		

When the specification shown is made, for example, the assembler sets the contents of the FTL field as shown in the figure.

```
FUNCTION  FADD1 = ADD(XCH, BRC, Y, EQ), QUEUE(QUE1,1)
```



- PU instruction functions
  The PU instructions define operations to be performed using the PU. This is not limited, however, to operations specified by the Op Code, but includes such operations as branching and output control based on the results of the operation. Details of each PU instruction are given on the following pages.

5.2.1 Logical operation instructions

These instructions perform 16-bit logical operation on the data fields of tokens arriving at the PU. These operations do not affect the C and S bits.

(1) OR, AND, XOR Instructions

OR Instruction
Finds the logical OR of the data fields of the tokens input to the A and B sides of the PU. The result (18 bits) is output to the X side.

AND Instruction
Same function as the OR instruction except that logical AND is found.

XOR Instruction
Same function as the OR instruction except that logical XOR (exclusive or) is found.

(2) ANDNOT Instruction
This instruction first finds the complement (NOT) of the data input from the A side (A) and then logically ANDs A with the data input from the B side. The result (18 bits) is then output to the X side.

(3) NOT Instruction
NOT is a one-operand instruction that inverts each bit of (finds the complement of) the data input to the A side. The result is output to the X side.

5.2.2 Arithmetic Operation Instructions

These instructions perform arithmetic operations for the 17-bit data fields (with S bit) of the tokens arriving at the PU.

(1) ADD, MUL, SUB and NOP Instructions

ADD Instruction
This instruction adds the data fields (sign bit and 16 bits of data) of the tokens input to the A and B sides of the PU. The result is output to the X side. The Y output becomes 0001H when the addition operation generates an overflow, -1 (actually, 0001H with S_Y=1) when an underflow has been generated, and 0 when neither of the above has occurred.
When PNZ has been specified, the C bits of output data X and Y reflect this setting. When there is no PNZ setting (PNZ=000) the value of the output C bits will be as follows:

C_A --> CX
C_B --> CY

SUB Instruction
This instruction subtracts the data of the B side token (sign plus 16 bits) from that of the A side token (same configuration). Note that the sign bit participates in the subtraction operation.
The result of the subtraction operation is output to the X side. The data output to the Y side will be 0001H if an overflow was generated by the subtraction operation and -1 (0001H with S_Y=1) if an underflow was generated. When PNZ has been specified, the C bits of output data X an Y reflect this setting. When there is no PNZ setting (PNZ=000) the value of the output C bits will be as follows:

C_A --> C_X
C_B --> C_Y

MUL Instruction
This instruction multiplies the data fields (sign bit plus 16 bits of data) of the tokens input to the A and B sides of the PU. The result (sign bit and 32 bits of data) is output in the following manner:

X side output: Sign bit and high 16 bits of data
Y side output: Sign bit and low 16 bits of data.

The sign bit of the result is thus output to both the X and Y side. When PNZ is specified, the C bit of output data X and Y reflect this setting. When there is no PNZ setting (PNZ=000) the value of the output C bits will be as follows:

C_A --> C_X
C_B --> C_Y

When PNZ processing is performed, the state of the C bit is determined by the result of the operation (sign

bit and 32 bits of data). For example, when the multiplication of 2 x 3 is performed, the output at the X side is 0 (that at the Y side side is 0006H). The PNZ judgement, however, is based on the 32 bit result (even for PNZ=001, $C_{X=0}$).

NOP Instruction
This instruction specifies that no operation is to be performed. The tokens arriving from the A and B sides are therefore output unchanged to the X and Y sides. If any control other than that of the Op Code (such as PNZ processing, BRC control, etc.) has been specified, the output complies with that control.

(2) ADDSC, MULSC, SUBSC and NOPSC Instructions
These SC (shift and count) instructions first perform the normal operation of the ADD, MUL, SUB or NOP instruction. Then the number of 0s from the MSB of the X side output are counted and the count is output to the Y side.

ADDSC Instruction
Performs SC processing after first adding the data fields of the tokens input to the PU from the A and B sides. At this time, even if the addition result includes a carry bit, it is not output to the Y side. However, if an over- or underflow has been generated, this is indicated by the sign bit of the Y side output enabling detection of this condition.

SUBSC Instruction
Performs SC processing after first performing subtraction using the data fields of the tokens input to the PU from the A and B sides. At this time, even if the subtraction result includes a carry bit, it is not output to the Y side. However, when an over- or underflow has been generated, it is indicated by the sign bit of the Y side output enabling detection of this condition.

MULSC Instructioon
Performs SC processing after first multiplying the data fields of the tokens input to the PU from the A and B sides. At this time, the SC processing is performed for the higher bits of the result output (X side) and the lower bits of the multiplication result are not output to the Y side.

NOPSC Instruction
Outputs to the Y side the result of the SC processing performed to the A side token. The A side data itself is output unchanged to the X side. The B side data is deleted.

Table 5-15

X side data after processing																SRC output (Y data)		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	S_Y	Y side data	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0010 H	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	000F H	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	x	0	000E H	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	x	0	000D H	
0	0	0	0	0	0	0	0	0	0	0	0	0	1	x	x	0	000C H	
0	0	0	0	0	0	0	0	0	0	0	0	1	x	x	x	0	000B H	
0	0	0	0	0	0	0	0	0	0	0	1	x	x	x	x	0	000A H	
0	0	0	0	0	0	0	0	0	0	1	x	x	x	x	x	0	0009 H	
0	0	0	0	0	0	0	0	0	1	x	x	x	x	x	x	0	0008 H	
0	0	0	0	0	0	0	0	1	x	x	x	x	x	x	x	0	0007 H	
0	0	0	0	0	0	0	1	x	x	x	x	x	x	x	x	0	0006 H	
0	0	0	0	0	0	1	x	x	x	x	x	x	x	x	x	0	0005 H	
0	0	0	0	0	1	x	x	x	x	x	x	x	x	x	x	0	0004 H	
0	0	0	0	1	x	x	x	x	x	x	x	x	x	x	x	0	0003 H	
0	0	0	1	x	x	x	x	x	x	x	x	x	x	x	x	0	0002 H	
0	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0001 H	
0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0000 H	
1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0000 H	
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	0001 H	<-

x = don't care

When an overflow or underflow has occurred

(3) INC and DEC Instructions

INC Instruction
Increments by 1 the data field (sign bit and 16 bits of data) of the A side token and outputs the result to the X side. The Y side output is 1 only if the addition operation generates an overflow. At all other times, the Y output is 0.
Normally, the value of S_A is output to S_X and S_Y. However, when S_A=1 and $DATA_A$=0, S_X=S_Y=0. If PNZ has been specified, the output reflects this specification; if PNZ is omitted, C_A is output unchanged to C_X and C_Y.

DEC Instruction
Decrements by 1 the data field (sign bit and 16 bits of data) of the A side token and outputs the result to the X side. The Y side output is 1 (the Y data is 0001H with S_Y=1) only when the subtraction operation generates an underflow. At all other times, the Y output is 0.
Normally, the value of S_A is output to S_X and S_Y. However, when S_A=1 and $DATA_A$=0, S_X=S_Y=1. The handling of the C bits is the same as for the INC instruction.

5.2.3 Shift Instructions

The shift instructions treat the data field of the A side token as shift data and use the sign bit and the lower 5 bits of the B token data to specify the number of shifts to be performed.

The shift operations that can be performed are from -16 to +16. The 32-bit result of the shift operation is output to X and Y.

(1) SHL and SHR instructions

SHL Instruction
Shifts the A side data the number of bits to the left indicated by the B side data. Shifted bit positions are padded with 0s (refer to Table 5-16).
The direction of the shift operation is determined by the S_B bit of the B data. When S_B=0, left shift is indicated and when S_B=1, right shift (in the reverse direction) is performed.

SHR Instruction
Shifts the A side data the number of bits to the right indicated by the B side data. Shifted bit positions are padded with 0s (refer to Table 5-17).
The direction of the shift operation is determined by the S_B bit of the B data. When S_B=0, right shift is indicated and when S_B=1, left shift (in the reverse direction) is performed.

(2) SHLBRV and SHRBRV Instructions

These two instructions exchange the position of the MSB and LSB of the data before performng left (SHLBRV) or right (SHRBRV) shift..

Before exchange:

(MSB) A_{15} A_{14} A_{13} A_{12} . . . . . A_3 A_2 A_1 A_0 (LSB)

After exchange:

(MSB) A_0 A_1 A_2 A_3 . . . . . A_{12} A_{13} A_{14} A_{15} (LSB)

Therefore, when these instructions are executed, the 16-bit A data is reversed and then a shift is performed. The shift is performed in the same manner as the SHL and SHR instructions.

Table 5-16 Left Shift (SHL execution)

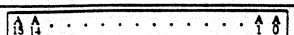
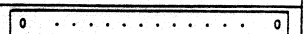
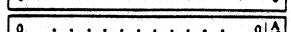
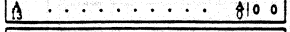
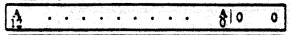
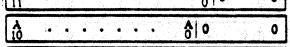
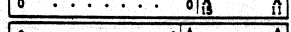
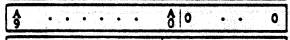
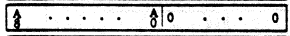
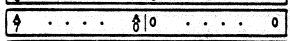
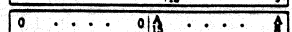
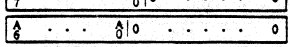
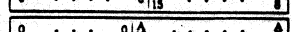
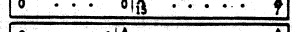
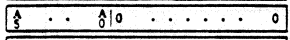
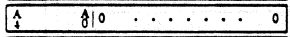
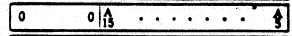
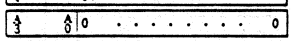
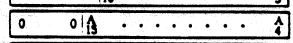
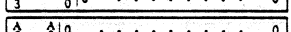
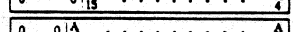
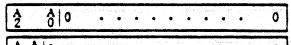
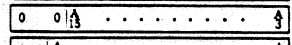
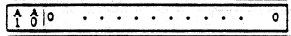
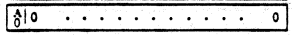
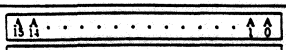
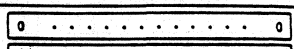
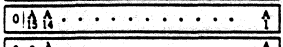
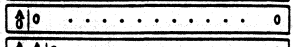
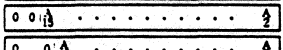
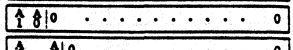
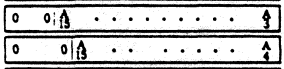
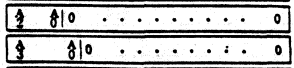
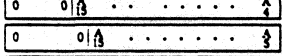
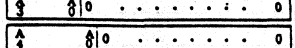
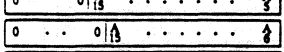
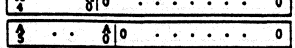
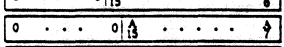
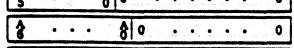
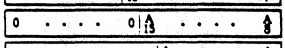
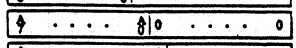
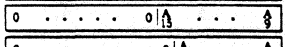
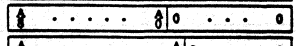
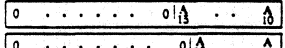
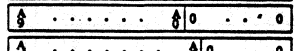
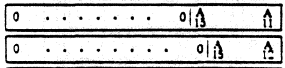
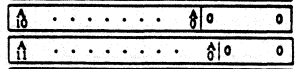
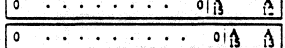
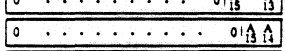
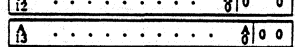
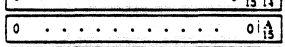
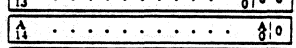
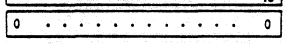
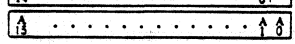
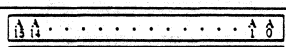
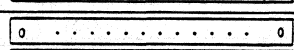
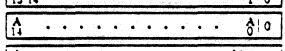
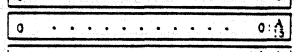
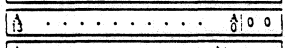
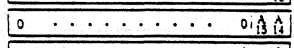
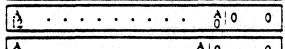
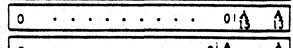
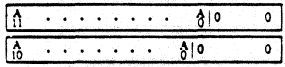
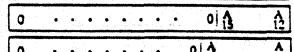
SB	DATAB Lower 5 bit (shift)	DATAX	DATAY
0	0 0 0 0 0		
0	0 0 0 0 1		
0	0 0 0 1 0		
0	0 0 0 1 1		
0	0 0 1 0 0		
0	0 0 1 0 1		
0	0 0 1 1 0		
0	0 0 1 1 1		
0	0 1 0 0 0		
0	0 1 0 0 1		
0	0 1 0 1 0		
0	0 1 0 1 1		
0	0 1 1 0 0		
0	0 1 1 0 1		
0	0 1 1 1 0		
0	0 1 1 1 1		
0	1 × × × ×		
1	0 0 0 0 0		
1	0 0 0 0 1		
1	0 0 0 1 0		
1	0 0 0 1 1		
1	0 0 1 0 0		
1	0 0 1 0 1		
1	0 0 1 1 0		
1	0 0 1 1 1		
1	0 1 0 0 0		
1	0 1 0 0 1		
1	0 1 0 1 0		
1	0 1 0 1 1		
1	0 1 1 0 0		
1	0 1 1 0 1		
1	0 1 1 1 0		
1	0 1 1 1 1		
1	1 × × × ×		

Table 5-17 Right Shift (SHR execution)

S_B	DATA$_B$ Lower 5 bit (shift)	DATA$_X$	DATA$_Y$
0	0 0 0 0 0		
0	0 0 0 0 1		
0	0 0 0 1 0		
0	0 0 0 1 1		
0	0 0 1 0 0		
0	0 0 1 0 1		
0	0 0 1 1 0		
0	0 0 1 1 1		
0	0 1 0 0 0		
0	0 1 0 0 1		
0	0 1 0 1 0		
0	0 1 0 1 1		
0	0 1 1 0 0		
0	0 1 1 0 1		
0	0 1 1 1 0		
0	0 1 1 1 1		
0	1 × × × ×		
1	0 0 0 0 0		
1	0 0 0 0 1		
1	0 0 0 1 0		
1	0 0 0 1 1		
1	0 0 1 0 0		
1	0 0 1 0 1		
1	0 0 1 1 0		
1	0 0 1 1 1		
1	0 1 0 0 0		
1	0 1 0 0 1		
1	0 1 0 1 0		
1	0 1 0 1 1		
1	0 1 1 0 0		
1	0 1 1 0 1		
1	0 1 1 1 0		
1	0 1 1 1 1		
1	1 × × × ×		

5.2.4 Comparison Instructions

These instructions compare, by performing subtraction, the data (sign bit and 16 bits) of the tokens arriving at the PU from the A side and the B side. PNZ judgement is then performed against the result and the result of the judgement (true or false) is used to generate the output data. PNZ must therefore always be specified when executing a comparison instruction. PNZ=000 and PNZ=111, however, may not be specified.

> Note that the PNZ conditions for a comparison instruction differ from those for a normal PU instruction.

(1) CMPNOM, CMP and CMPXCH instructions

CMPNOM Instruction
Performs PNZ judgement after first comparing the data of the tokens input to the PU from the A and B sides. If the result of the judgment is 'true', the C bits of both the X and Y sides are set to 1; the X side data field is 1 and that of the Y side is 0. The S bits, both S_X and S_Y are 0. If the result of the judgement is 'false', all C_X, C_Y, S_X, S_Y, and output data X, Y becomes 0.

CMP Instruction
Performs PNZ judgement after first comparing the data of the tokens input to the PU from the A and B sides. Except for the condition of the C bits, the output data is the unchanged input data; the A data and S_A are output to the X side and the B data and S_Y are output to the Y side. If the result of the PNZ judgement is false, the C bits C_X and C_Y are set to 0; if true, 1. This instruction is normally used in conjunction with the BRC bit.

CMPXCH Instruction
Performs PNZ judgement after first comparing the data of the tokens input to the PU from the A and B sides. If the result of the PNZ judgement is 'false', the A and B data are exchanged in the PU, that is, the B data (including the S and C bits) is output to the X and the A data (with S and C bits) to Y. When the result is 'true', the exchange operation is not performed.

Table 5-18 shows the state of the PU I/O data during comparison operations and the meaning of the PNZ bits when used with a comparison instruction.

Table 5-18

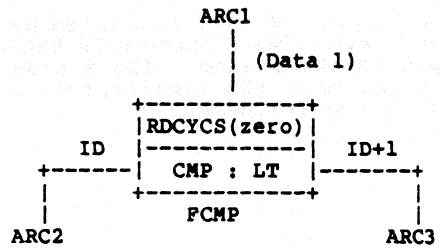
Mnemonic	Input						Output						Notes
	CA	SA	DATAA	CB	SB	DATAB	CX	SX	DATAX	CY	SY	DATAY	
CMPNOM	CA	SA	A	CB	SB	B	0	0	0000H	0	0	0000H	When PNZ is false
	CA	SA	A	CB	SB	B	1	0	0001H	1	0	0000H	When PNZ is true
CMP	CA	SA	A	CB	SB	B	0	SA	A	0	SB	B	When PNZ is false
	CA	SA	A	CB	SB	B	1	SA	A	1	SB	B	When PNZ is true
CMPXCH	CA	SA	A	CB	SB	B	CA	SA	A	CB	SB	B	When PNZ is true
	CA	SA	A	CB	SB	B	CB	SB	B	CA	SA	A	When PNZ is false

PNZ	Condition	True/False	Note	Mnemonic
001	SA, DATAA = SB, DATAB	True	Equal	EQ
	SA, DATAA ≠ SB, DATAB	False	Not equal	
010	SA, DATAA < SB, DATAB	True	Less than	LT
	SA, DATAA ≥ SB, DATAB	False	Greater or equal	
011	SA, DATAA ≤ SB, DATAB	True	Less or equal	LE
	SA, DATAA > SB, DATAB	False	Greater than	
100	SA, DATAA > SB, DATAB	True	Greater than	GT
	SA, DATAA ≤ SB, DATAB	False	Less or equal	
101	SA, DATAA ≥ SB, DATAB	True	Greater or equal	GE
	SA, DATAA < SB, DATAB	False	Less than	
110	SA, DATAA ≠ SB, DATAB	True	Not equal	NE
	SA, DATAA = SB, DATAB	False	Equal	

The significance of the PNZ bits when comparison instructions are executed differ from that for other PU instructions. PNZ=111 and PNZ=000 cannot be specified for comparison instructions.

Comparison instruction assembler description:
The following is a (partial) flowgraph for execution of the CMP comparison instruction. In the example in Figure 5-46, the flow of tokens to ARC2 or ARC3 is determined by whether Data 1 input from ARC1 is larger or smaller than 0.

Figure 5-46



The assembler description for this operation is shown below.

```
LINK          ARC2, ARC3 = FCMP(ARC1);
FUNCTION      FCMP = CMP(LT,BRC), RDCYCS(ZERO,1);
MEMORY        ZERO = 0H;
```

First, the token input from ARC1 (Data 1) reads data 0 from the DM. In this way, Data 1 becomes the A side data and 0 the B side data when input to the PU. After the comparison has been performed, the C bit is set in accordance with the PNZ specification. Because the PNZ specification is 'less than' (010), BRC=1, Data 1 will be output to ARC3 (ID+1) if it is less than 0 and to ARC2 (ID) if it is greater.

5.2.5 Bit Manipulation Instruction

These instructions are used to get (read), set (to 1), or clear (to 0) single bits within the 16-bit A data. The location of the bit to be manipulated is indicated by the lower 4 bits of the B data.

(1) GET1, SET1 and CLR1 Instructions

GET1 Instruction
Reads the bit at the A data bit position specified by the lower 4 bits of the B data. This 1-bit data is then output as the LSB of the X data.
In other words, if the bit at the indicated position is 1, the X output will be 0001H and if it is 0, X

will be 0000H.
The Y side data is always = 0. If there is no PNZ specification, the C and S bits are handled as follows:

C_A --> C_X
C_X --> C_Y
S_A --> S_X
S_Y = 0

SET1 Instruction
Sets the bit at the A data bit position specified by the lower 4 bits of the B data. The A data, with this bit set, is then output to the X side. The Y side data is always 0. If there is no PNZ specification, the C and S bits are handled as follows:

C_A --> C_X
C_B --> C_Y
S_A --> S_X
S_Y = 0

CLR1 Instruction
Clears (reset to 0) the bit at the A data bit position specified by the lower 4 bits of the B data. The A data, with this bit cleared, is then output to the X side.
The Y side is always 0. If there is no PNZ specification, the C and S bits are handled as follows:

C_A --> C_X
C_B --> C_Y
S_A --> S_X
S_Y = 0

The following shows the relation between A and B data:

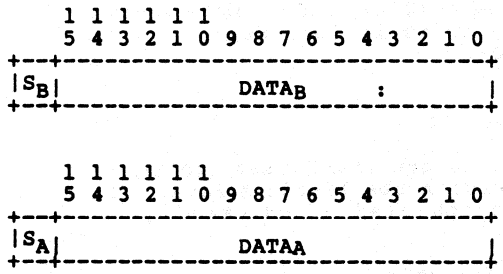


Table 5-18
A Data Bit Specification

B Data (LSB) 3 2 1 0	A Data Bit Position
0 0 0 0	A_0
0 0 0 1	A_1
0 0 1 0	A_2
0 0 1 1	A_3
0 1 0 0	A_4
0 1 0 1	A_5
0 1 1 0	A_6
0 1 1 1	A_7
1 0 0 0	A_8
1 0 0 1	A_9
1 0 1 0	A_{10}
1 0 1 1	A_{11}
1 1 0 0	A_{12}
1 1 0 1	A_{13}
1 1 1 0	A_{14}
1 1 1 1	A_{15}

5.2.6 Bit Check Instructions

These instructions test the bits of the A data in the positions corresponding to the '1' bits of the B data. The C bits of the output data (C_X and C_Y) are then set in accordance with the result. When the instruction is assembled, the assembler automatically sets the necessary data in PNZ. This setting cannot be performed by the user.

1. ANDMSK and ORMSK Instructions

ANDMSK Instruction
This instruction sets the C bits of the output data (C_X and C_Y) if all the bits of the A data in the positions

corresponding to the '1' bits of the B data are set.

Example 1:

A Data	0 0 1 1 0 1 1 1 0 0 0 1 0 1 1 1
B Data	0 1 0 1 0 1 0 1 0 1 0 1 1 1 0 1
Output C bits	C bits of output tokens are '0'

Example 2:

A Data	1 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1
B Data	1 0 0 1 0 0 1 1 0 0 1 1 0 0 0 1
Output C bits	C bits of output tokens are '1'

The C bits of both the X output and the Y output are set. The sign (S) bits and 16-bit data fields of the A and B input data are output as is to the corresponding X and Y sites.

S_A --> S_X
S_B --> S_Y
A data --> X data
B data --> Y data

ORMSK Instruction
This instruction sets the C bits of any one of the bits of the A data in the bit positions corresponding to the '1' bits of the B data is set.

Example 1:

A Data	0 0 1 1 1 0 1 0 0 1 1 0 0 0 1 0
B Data	1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1
Output C bits	C bits of output tokens are '1'

Example 2:

A Data	0 0 0 0 1 0 0 1 1 1 0 1 0 0 0 1
B Data	0 1 1 0 0 1 0 0 0 0 0 0 1 1 0 0
Output C bits	C bits of output tokens are '0'

The C bits of both the X output and the Y output are set. The sign (S) bits and the 16-bit data fields of the input A and B data are output as is to the corresponding X and Y sides.

S_A --> S_X
S_B --> S_Y
A data --> X data
B data --> Y data

5.2.7 Data Conversion Instructions

These instructions convert between numeric values expressed in two's complement format and those expressed as an absolute value (sign bit plus 16 bits of data). Since the latter is the format used by the uPD7281, this command is used to convert external two's complement data for internal use. These instructions can be used for a single data in which case data 0 is output to the Y side and $S_{Y=0}$.

1. CVT2AB and CVTAB2 Instruction

CVT2AB Instruction
This instruction converts 16-bit numeric values expressed in two's complement format to 'sign bit plus 16-bit absolute value' format.

CVTAB2 Instruction
This instruction converts sign bit plus 16-bit absolute value format data to two's complement format. Note that this operation entails the risk of generating an overflow or underflow. Should this happen, it is indicated by the C bit of the X output data. In other words, when C_X=1, it means that an overflow or underflow has been generated.

When this instruction is assembled, the assembler automatically sets the necessary data in PNZ and this setting can therefore not be performed by the user.

5.2.8 Double-Precision Adjustment Instruction

1. ADJL Instruction

To handle sign bit plus 32-bit data, the uPD7281 first divides this data into a high and low word (each in the format of sign bit plus 16 bits) and then processes each word independently. As a result, there may be cases in which the sign bits of the high and low words of the result will be different. The ADJL instruction adjusts these two independent 16-bit data so that the sign bits of the high and low word agree. This instruction can be executed for 32-bit data irrespective of whether sign bits agree.

The different types of processing are performed according to the condition of the sign bits. When the sign bits of the high and low words are equal, the tokens pass through the PU as NOP. When the sign bits of the high and low words do not agree, the sign bit of the low word is first changed to that of the high word. The two's complement of the low word is then found and this becomes the low word of the 32-bit data. However, should the value of the high word be 0, the sign bit of the high word is changed to that of the low word.

Input/Output		Sign	Data
Input	High (A data)	0	1234H
	Low (B data)	0	5678H
Output	High (X data)	0	1234H
	Low (Y data)	0	5678H

Input/Output		Sign	Data
Input	High (A data)	0	1234H
	Low (B data)	1	5678H
Output	High (X data)	0	1233H
	Low (Y data)	0	A988H

Input/Output		Sign	Data
Input	High (A data)	1	1234H
	Low (B data)	0	5678H
Output	High (X data)	1	1233H
	Low (Y data)	1	A988H

The PNZ judgement by this instruction is determined by the result of the operation (sign bit plus 32-bit data).

5.2.9 Cumulative Addition

1. ACC (Accumulation) Instruction
This instruction uses the ACC register of the PU (18 bits with C and S) to perform cumulative addition of the data fields of the arriving tokens. When using this instruction, the tokens input to the PU must be divided into those that vanish after the operation and those that read out the contents of the ACC register. The following three types of tokens read out the contents of the ACC register:
   a. The FTRC=1 token output (when BS=RC) from an RDCYCS instruction linked to the ACC instruction.
   b. The FTRC=1 token output (when BS=RC) from an RDCYCL instruction linked to the ACC instruction.
   c. The FTRC=0 token output (when CS=C) from a COUNT instruction linked to the ACC instruction. The ACC instruction is most commonly used with the COUNT instruction.

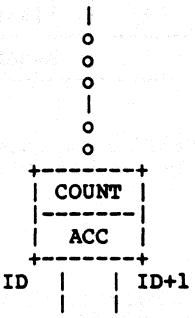
   Any type of token other that the three listed above will be deleted after the cumulative addition operation is performed.

   - Overflow (underflow) generated by the ACC instruction
   Since the ACC register stores numeric values as a sign bit plus 16 bits, overflow may be generated in the course of cumulative addition of values. If an overflow has occurred, the ID of the token that reads the contents of the ACC register will be changed to ID+1 and the C bit to 1. (This contrasts with normal tokens, which are output to ID.) Determination of whether or not an overflow has occurred can therefore be made according to the destination (ID or ID+1) of the token that reads the ACC register.

   In the example in Figure 5-47, an overflow has occurred during the cumulative addition operation performed by the linked ACC and COUNT instructions. The token that reads the contents of the ACC register

when CS=C is output to ID+1. If an overflow had not occurred, this token would have been output to ID.

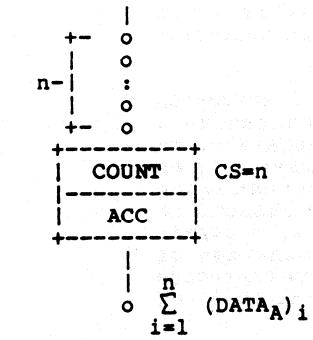
Figure 5-47



Methods for using the ACC instruction are shown next. Examples are shown here both for when the number of data to be totaled is known and for when it is not known. An example of handling overflows is also given.

1. When the number of data to be totaled is known
   In this example, the ACC instruction is linked to a COUNT instruction (AG & FC). The count of the COUNT instruction (CS) is set to the number of data to be totaled

Figure 5-48

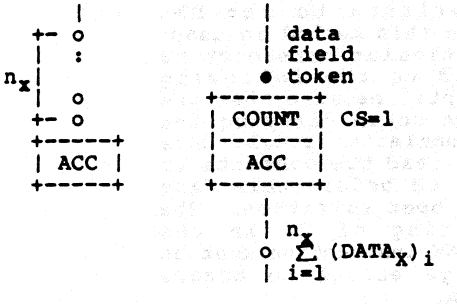


When this instruction is executed, CS is incremented and the A data of the arriving tokens totaled. When the condition C=CS is fulfilled in the COUNT instruction, the data of the current token is totaled and then the contents of the ACC register are read out. After the token containing the contents of the ACC register is output, the ACC register is cleared to 0.

2. When the number of data to be totaled is not known
   In this case, two ACC instructions should be used. The first ACC instruction is an independent

instruction used exclusively to perform the cumulative addition operation. When this instruction is executed, the A side tokens are accumulated in the ACC register after which they are deleted. Any number of tokens can be accumulated in the ACC register using this instruction. The other instruction is used to read the contents of the ACC register after the cumulative addition has completed.

Figure 5-49



This is a linked instruction consisting of a COUNT and an ACC instruction. Since the ACC register read condition must be fulfilled, the CS parameter of the COUNT instruction is set to 1. Note that since the ACC register is read after it has been added to the contents of the data field of the read token, this data must be 0 to avoid changing the results. (If the user is aware of this fact, the data field need not be 0.)

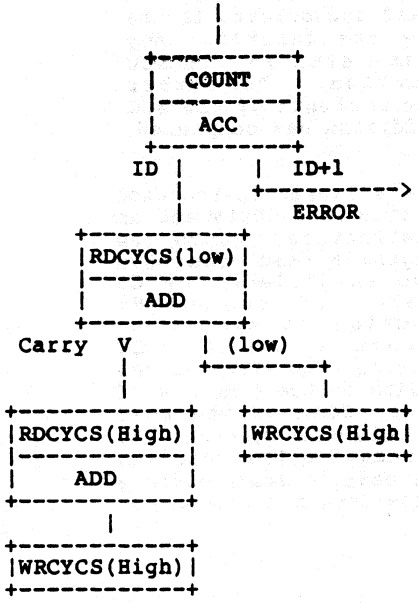
3. To handle overflows

Since the ACC register stores data as a sign bit plus 16 bits of data, the method shown below can be used to avoid overflows (within limits).

First, the CS parameter of the COUNT instruction is set to the estimated largest number of data that can be accumulated in the ACC register without causing an overflow. This is so that the contents of the ACC register can be read before an overflow occurs. ADD instructions are used to perform additions that are likely to generate an overflow. Figure 5-50 shows an example of this.

Here the result of the totaling operation is output from the ACC instruction and then added to a low word of data (initial value: 0) read from the DM.

Figure 5-50



If an overflow occurs at this time, the carry data of the Y output will be set to 1 and added to the high word of the DM location (initial value: 0). The result of this addition is then rewritten to the DM. When this method is used, particular care must be paid to the following point: be sure that the next token carrying the accumulation result does not read the contents of the DM before this data has been rewritten. The setting of CS in the COUNT instruction must be large enough to assure this.

Cautions when using the ACC instruction:

a. The contents of the ACC register of the PU block are cleared to 0 either by input of an external reset or a command reset. The ACC register is also cleared by the program after the contents have been read.
b. There is only one ACC register within the PU. For this reason, two or more accumulative addition operations cannot be performed at the same time.
c. If an over- or underflow occurs during the accumulative addition operation, the token that read the result is output to ID+1. The C bit of this token is set to 1.
d. The PNZ and BRC settings required by this instruction are automatically set by the assembler and can therefore not be set by the user.

5.2.10 C Bit Copy

This instruction copies the C bits of the A and B data of the tokens input to the PU. The user cannot set the PNZ condition when this instruction is used.

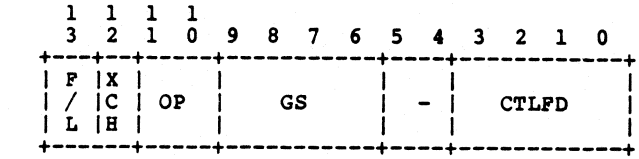
COPYC Instruction
This instruction copies the C bit of the A data of the tokens input to the PU to the C bit of the B data.

5.3 GE Instructions

The GE Instructions use the PU to generate and to copy data. These instructions generate multiple tokens from a single token input to the PU (processing unit). They are defined by the SEL bits of LT (SEL=10). The contents of the GE instruction operation to be performed are stored in the FTL field of the FT address indicated by the FTA (function table address) field of the LT token. To execute a GE instruction, the token arriving at the FT read bits 0 to 11 of the FTL (refer to the figure below) and incorporates this data before proceeding to the PU.

When F/L=0, the GE instruction can be used independently, and when F/L=1, it can be linked to an AG & FC instruction. Of the AG & FC instructions, the CNTGE instruction is most commonly linked to a GE instruction.

FTL Field for GE Instruction



F/L : Full/Left
XCH : Exchange
OP : Operation
GS : Generation Size
CTLFD : Control Field Data

F/L
This bit indicates whether the GE instruction is to be used alone or linked to an AG & FC instruction. F/L=0 indicates that the GE instruction is used alone and F/L=1 indicates that it is linked and used in combination with an AG & FC instruction. There are many possible combinations for GE and AG & FC instructions. However, the most common combined usage is that of COPYBK and the CNTGE instruction.

XCH
This bit indicates whether the relation of the A and B sides of two-operand data at the exit of the DM is to be exchanged before it is input to the PU. Normally, at the entrance to the PU, the tokens arriving from the LT via the FT (bus data) become the A data and that those read from the DM (memory data) the B data. The control performed by this bit exchanges the data before it is input to the PU. Refer to the description of the PU instructions.

OP
These two bits select from among the three types of GE instructions.

OP=00 : COPYBK (Copy Block)
OP=01 : COPYM (Copy Multiple)
OP=11 : SETCTL (Set Control Field Data)
OP=10 : Use prohibited

GS
These four bits determine the number of tokens to be generated by the PU.

CTLFD
This field is set as the CTLF data (of the LT token) when the GE instruction SETCTL is executed. The data that can be set by the user as the CTLFD field are shown in Table 5-19. If any other value is used, the token output from the PU will be deleted when it reaches the LT.

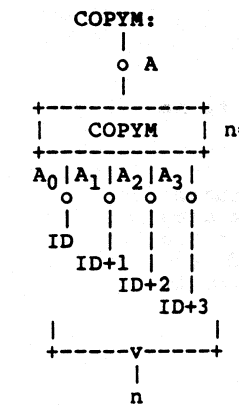
Table 5-19

CTLFD C_3	C_2	C_1	C_0	Description
0	0	x	x*	Normal execution data
1	1	0	0	Sets the contents of LT; token deleted
1	1	0	1	Sets the ID address of FTR; token deleted
1	1	1	0	Sets the ID address of FTL; token deleted
1	1	1	1	Sets the ID address of FTT; token deleted
1	0	0	0	Reads the ID address of LT; outputs this externally
1	0	0	1	Reads the ID address of FTR; outputs this externally
1	0	1	0	Reads the ID address of FTL; outputs this externally
1	0	1	1	Reads the ID address of FTT; outputs this externally

* Determines the sign and control bits of the execution token.

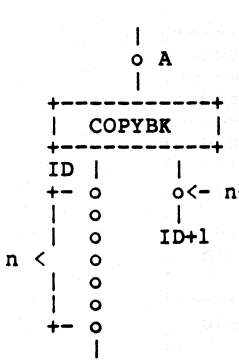
There are three kinds of GE instructions. These are COPYBK, COPYM and SETCTL.

COPYM Instruction



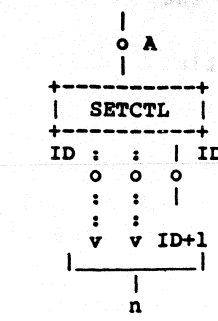
When this instruction is executed, the token arriving from the input arc is copied the number of times specified by the user ($2 \leq n \leq 17$). The IDs of the copied tokens are the ID of the input token incremented each time copy token is performed (see figure at left).

COPYBK Instruction



This instruction causes the token arriving on the input arc to be copied the number of times specified by the user +1. The values that can be specified by the user for this instruction are $1 \leq n \leq 16$. The IDs of the tokens generated by this instruction are the same as that of the input token. However, the ID of the last (n+1) token is ID+1 (see the figure at left)

SETCTL Instruction



When a token arrives at this instruction from the input arc, it is copied the number of times specified by the user ($2 \leq n \leq 17$). The ID field of the input token is incremented each time token copy is performed to generate the IDs of the new tokens (see the figure at left)

Note that the lower four bits of the FTL of the input token (the CTLFD field) replace the CTLF of the copied tokens and determine the operation to be performed by these tokens. The CTLF field of the n^{th} token, however, is the same as that of the execution token.

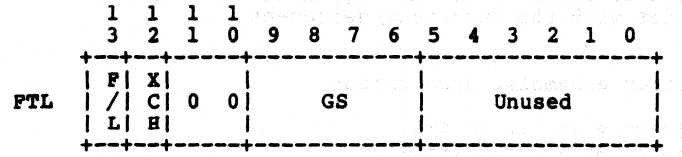
Functions of GE instructions

The GE instructions use the PU to generate tokens. These instructions do more, however, than simply generate tokens. They can also generate arithmetical progressions since they can increment/decrement the data.

The following are detailed descriptions of each of the GE instructions.

5.3.1 COPYBK (Copy Block) Instruction

COPYBK instruction field format

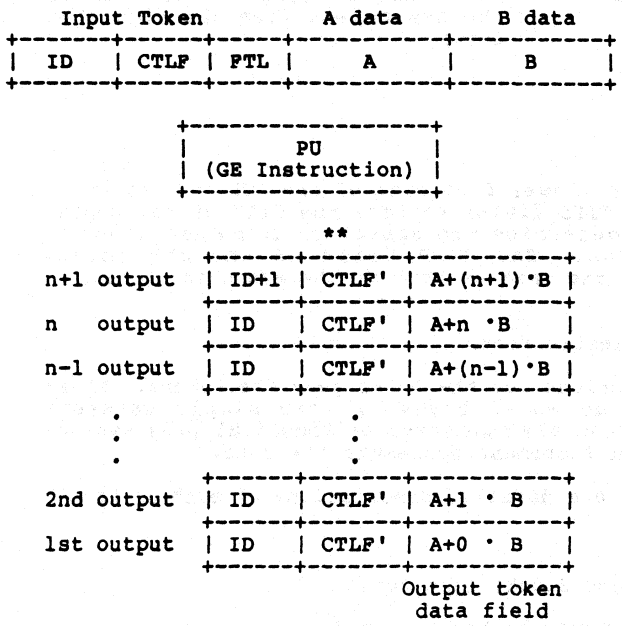


This instruction uses the PU to generate n+1 tokens for the value n specified by the user in the assembler description. The IDs of the first n tokens are the same as the ID of the input token; that of the n+1 token is ID+1. Also, when the specified number of tokens is being generated, the data fields of the tokens can be

incremented or decremented at a fixed rate specified as the increment/decrement value*. The relation between this value, the data fields of the output tokens and the ID fields is defined in Figure 5-51.

* The increment/decrement is calculated using the sign bit

Figure 5-51



** The sign bit in the CTLF of the output token changes in accordance with the increment/decrement

COPYBK instruction assembler description

1. For the example in Figure 5-52

```
LINK      ARC2, ARC3 = FCBK(ARC1);
FUNCTION  FCBK = COPYBK(size, delta{,XCH});
```

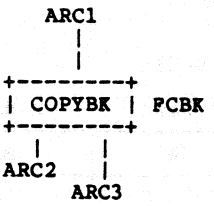
size : Indicates the value to be set in the GS field of this instruction and is automatically set in FTL by the assembler. The value set in FTL, however, is size -1.

delta : Indicates the value of the data input to the B side of the PU. This is the increment/decrement values used when tokens

are generated by the PU.

XCH : This parameter is specified to exchange the value read from the DM and the bus data coming from the LT via the FT. The exchange operation is performed at the exit of the DM. The default is XCH=0 (exchange not performed).

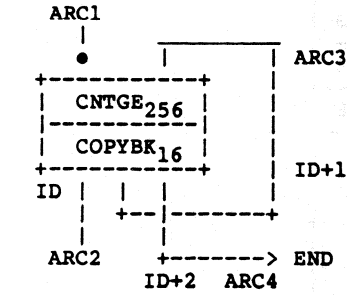
FIgure 5-52



2. For the example in Figure 5-53

```
LINK       ARC2, ARC3, ARC4 = FCBK(ARC3,ARC1);
FUNCTION   FCBK = COPYBK(16,1), CNTGE(256);
```

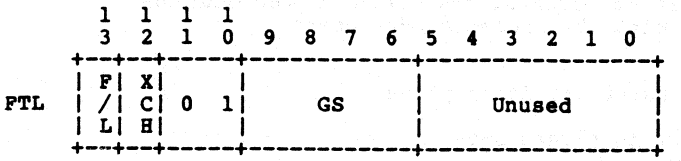
Figure 5-53



This is an example showing the most common method for generating 17 or more tokens. Here 16 x 256 tokens are generated for each token arriving from ARC1. The data fields of the copied tokens are incremented by 1 each time a token is generated.

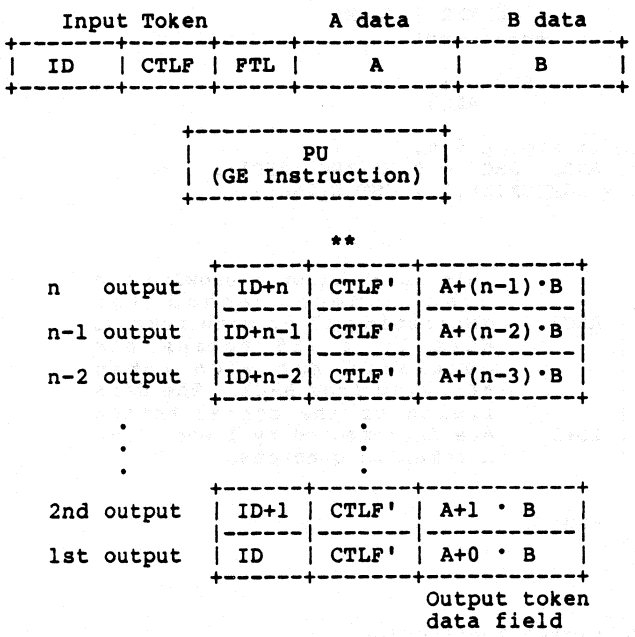
5.3.2 COPYM (Copy Multiple) Instruction

COPYM instruction field format



This instruction uses the PU to generate n tokens as specified by the user in the assembler description. The IDs of the n tokens generated by this instruction differ from those of the COPYBK instruction in that they are incremented by 1 each time a token is generated. Also, at the same time as the specified number of tokens is being generated, the data fields of the tokens can be incremented or decremented at a fixed rate specified as the increment/decrement value*. The relation between this value, the data fields of the output tokens and the ID fields is shown in Figure 5-54.

Figure 5-54



* The increment/decrement is calculated using the sign bit

** The sign bit in the CTLF of the output token changes in accordance with the increment/decrement

COPYM instruction assembler description

1. For the example in Figure 5-55

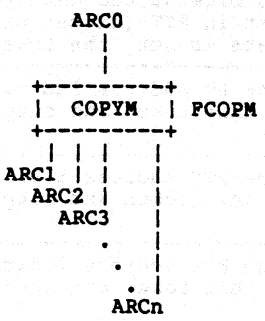
LINK ARC1, ARC2, ARC3, ... ARCn = FCOPM(ARC0);
FUNCTION FCOPM = COPYM (size,delta{,XCH});

size : Indicates the value to be set in the GS (generation size) field of this instruction and is automatically set in FTL by the assembler. The value set in FTL, however, is size -2.

delta : Indicates the value of the data input to the B side of the PU. This is the increment/decrement value used when tokens are generated by the PU.

XCH : This parameter is specified to exchange the value read from the DM and the bus data coming from the LT via the FT. The exchange operation is performed at the exit of the DM. The default is XCH=0 (exchange not performed).

Figure 5-55



5.3.3 SETCTL (Set Control Field Data) Instruction

This instruction is used to read and rewrite the contents of the LT and FT. This instruction can be used to create a program and rewrite itself. A thorough understanding of the functions of these fields is therefore required to use this instruction (refer to Table 5-20).

Table 5-20

CTLFD C_3	C_2	C_1	C_0	Operation
0	0	x	x*	Normal execution data. Operation is exactly the same as COPYM
1	1	0	0	The data field of this token is set in the LT. After the data is set, the token is deleted (C and S bits are not included)
1	1	0	1	The data field of this token is set in FTR. After the data is set, the token is deleted (C and S bits are not included)
1	1	1	0	The lower 14 bits of the data field of this token are set in FTL (higher bits ignored). After the data is set, the token is deleted
1	1	1	1	The lower 10 bits of the data field of this token are set in FTT (higher bits ignored). After the data is set, the token is deleted
1	0	0	0	Reads out the LT address indicated by the ID field of this token and outputs the contents.
1	0	0	1	Reads out the FTR address indicated by the ID field of this token and outputs the contents.
1	0	1	0	Reads out the FTL address indicated by the ID field of this token and outputs the contents.
1	0	1	1	Reads out the FTT address indicated by the ID field of this token and outputs the contents.

The set or write operation is performed at the address indicated by the ID field of the token.

* Determines the sign and control bits of the execution token.

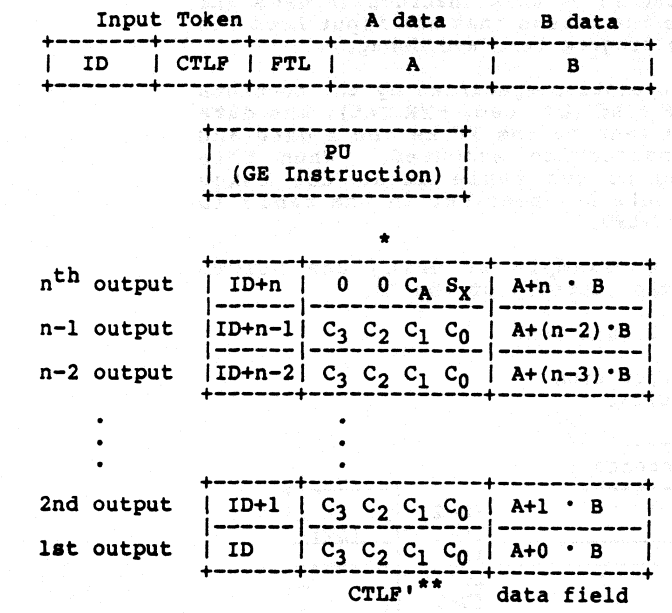
CTLF of the output token is defined by the CTLFD in the FTL of the input token.

PTL field format of the input token

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PTL	F/L	XCH	1	1	GS								CTLFD ($C_3C_2C_1C_0$)			

The relation between the input token, and the ID and CTLF fields of the tokens output by the SETCTL instruction is shown in Figure 5-56.

Figure 5-56



* The C_A bit of CTLF of the n^{th} output is the C_A bit of the input token. The S_X bit changes with the increment/decrement of the data field.

** CTLF' is determined by the CTLFD field of the FTL of the input token.

SETCTL instruction operation

This instruction generates the number of tokens (n) specified by the user in the assembler description. The IDs of the tokens increment each time a new token is generated. Also, at the same time the specified number of tokens is being generated, the data fields of the tokens can be incremented or decremented at a fixed rate (the increment/decrement value).

This instruction differs from the COPYM instruction described earlier in that values specified by the user can be set in the CTLF field of the output tokens. Tokens whose CTLF field is set in this way can thus be used to perform a wide range of operations, such as writing the data field of the token to the LT or FT, reading out the contents of the LT or FT, adding data to the data field of the token, etc. The set and read tokens output from the PU by this instruction have the same field format as the tokens that are input from the outside world during program load and debug.

For example, if the operation specified by the contents of the CTLF is table read (LT read, FTR set), the data to be set should be sent to the PU as the A data and then the SETCTL instruction executed. When this instruction is used to set table data, the token vanishes after the data has been set in the table ID addres specified by CTLFD.

Figure 5-57 shows an example of using the SETCTL instruction to read the contents of the LT.

Figure 5-57

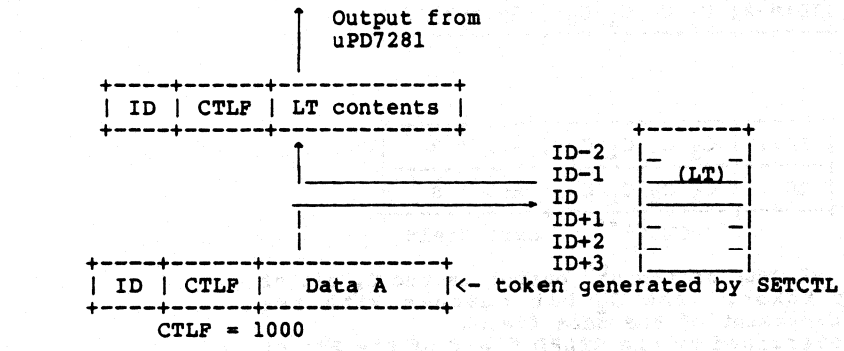
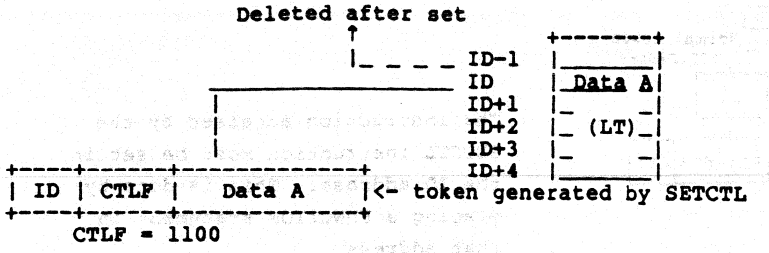


Figure 5-58 shows an example of using the SETCTL instruction to set the LT.

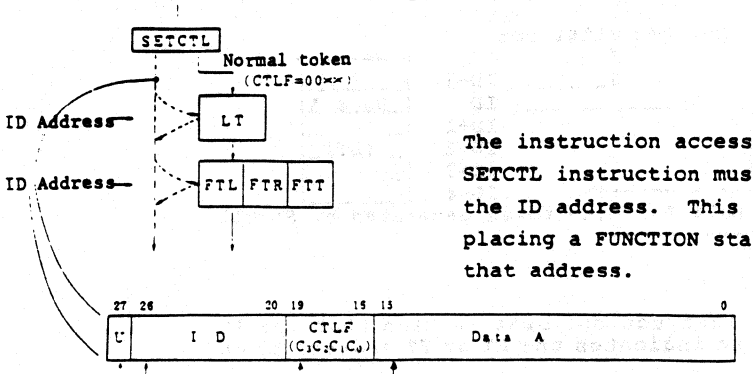
Figure 5-58



When using this instruction, bear in mind that the ID field of the token indicates the LT or FT address to be accessed.

As shown in Figure 5-59, the table location indicated by the ID of the token output after execution is the address that has been accessed. The token passes through (as NOP) any table other than the one specified by CTLF.

Figure 5-59



The instruction accessed by the SETCTL instruction must be set in the ID address. This is done by placing a FUNCTION statement in that address.

If CTLF indicates a set instruction, the set data is stored here. If a read instruction is indicated, the contents of this field are undefined.

This field determines which table is to be accessed and whether that access is read or write.

The address indicated by this ID field is accessed according to the operation indicated by the CTLF field. To set this ID field, place the assembler LINK statement in an absolute address as specified by the user.

The basic function of this instruction is the same as the COPYM instruction, and for this reason, it can be used to read multiple locations within a table (the ID addresses must, however, be contiguous). It would also be possible to write (set) data to contiguous ID addresses of the LT or FT table although in most cases there would be no reason to set the same data or data that must change at a fixed rate in multiple table locations.

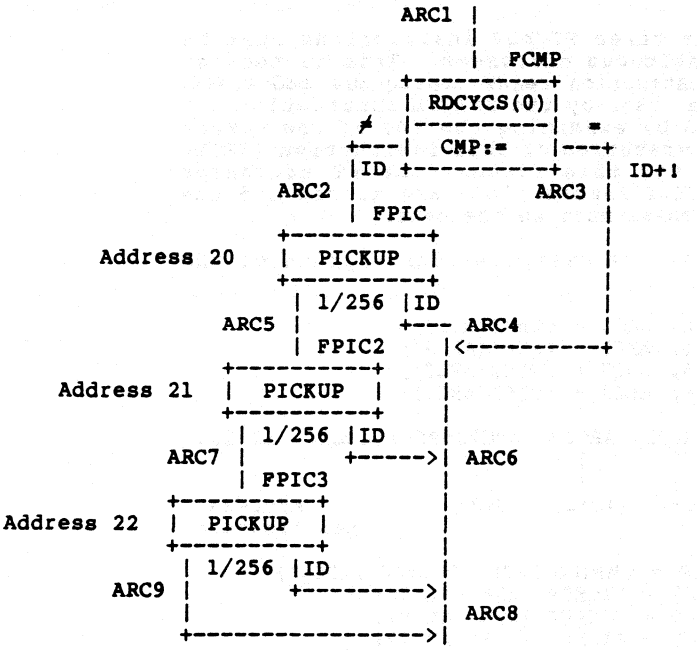
Using the SETCTL instruction (1)

In this example, the SETCTL instruction is used to perform FTT read (CTLF=1011) as part of a program that reads screen data in word units and counts the number of words that contain data other than 0.

In the example in Figure 5-60, tokens containing the

data to be processed are successively input to ARC1. These high-volume data are compared against 0 and the flow of tokens containing data other than 0 is diverted to the PICKUP instruction. When the number of non-zero data exceeds 255, the tokens go to the next PICKUP instruction. Thus, in this example, the data (up to FFFFFFH) can be counted using the three 8-bit counters in the FTT fields of three PICKUP instructions.

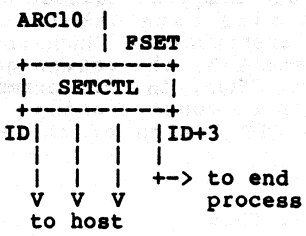
Figure 5-60



DUST

Next, the SETCTL instruction is used to read the contents of the counter (FTT field) and output this data to the host (Figure 5-61). When data are output to the host in this way, use of an OUT instruction is not required. The host can then determine the number of non-zero data by examing the three 8-bit data values.

Figure 5-61



To host

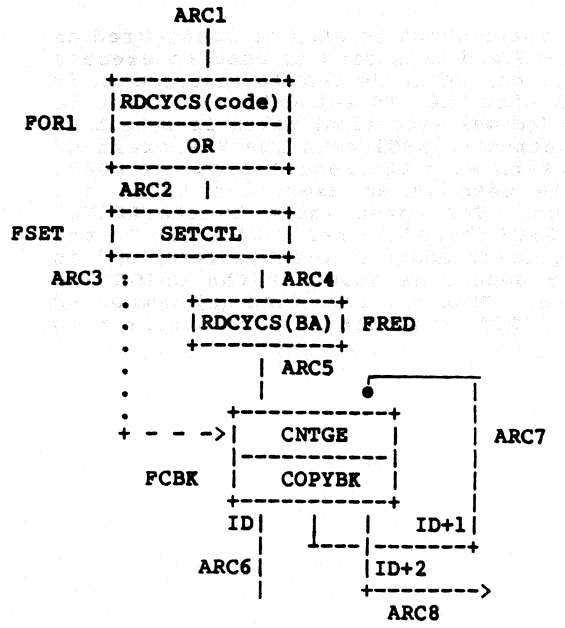
Note that the three PICKUP instructions must be stored in contiguous addresses. This is because the SETCTL instruction reads contiguous addresses. The addresses read by the SETCTL instruction can be determined by examining the IDs of the tokens output after execution of this instruction (ARC20 addresses). For this reason, the FT addresses where the PICKUP instructions are stored and the ARC20 LT addresses must be the same.

The following is the assembler description of the above example.

```
LINK        ARC2, ARC3 = FCMP(ARC1);
LINK        ARC4, ARC5 = FPIC1(ARC2);
LINK        ARC6, ARC7 = FPIC2(ARC5);
LINK        ARC8, ARC9 = FPIC3(ARC7);

LINK        ARC20, ARC21, ARC22, ARC23=FSET(ARC10) AT 20;
              (ID)  (ID+1) (ID+2) (ID+3)            Address
                                                  Specification

FUNCTION    FCMP = CMP(EQ,BRC), RDCYCS(ZERO,1);
FUNCTION    FPIC1 = PICKUP (256) AT 20;
FUNCTION    FPIC2 = PICKUP (256) AT 21;
FUNCTION    FPIC3 = PICKUP (256) AT 22;

FUNCTION    FSET = SETCTL (4, 0, 1011B)
                           size |  CTLF
                    increment/decrement value

MEMORY      ZERO=0;
```

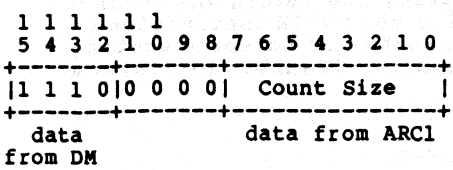
Using the SETCTL instruction (2)
This is an example of using the SETCTL instruction to rewrite the GS field of the CNTGE instruction to generate the specified number of tokens (counting from the base address). (Number of tokens generated n: $1 \leq n \leq 256$).

Figure 5-62



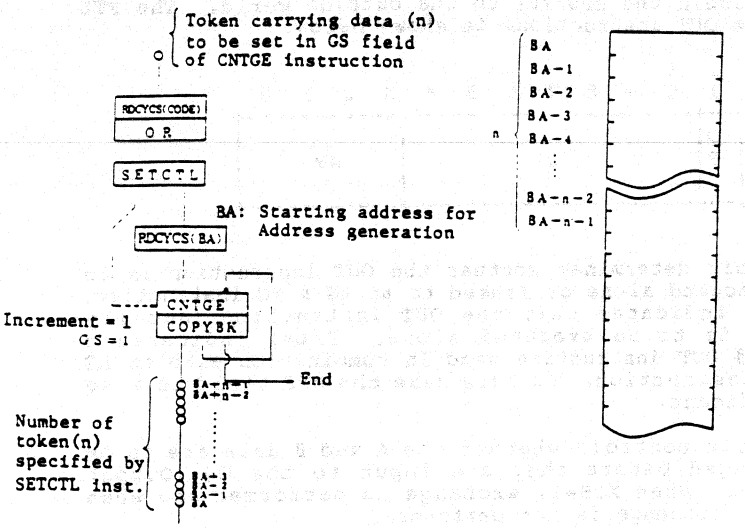
In the example in Figure 5-62 the number of addresses to be generated (count size) is first input from ARC1. (This number is set directly, so n-1 should not be specified.) Next the code of the CNTGE instruction (E000H) is read and these two values are logically ORed. The result completes the data to be written to the FTR field of the CNTGE instruction.

Figure 5-63



The A data of the token ouput to ARC2 is configured as shown in Figure 5-63 and this data is used to execute the SETCTL instruction. When the SETCTL instruction is executed, a token with the FTR set data is output to ARC3 and a CTLF=0 (normal execution) token is output to ARC4. The token output to ARC3 sets the FTR field of the CNTGE instruction with the result that the token output to ARC4 is used as an execution token for address generation. The token input to the RDCYCS instruction from ARC4 therefore reads from the DM the address generating start address (base address) and is then input to the node consisting of the CNTGE and COPYBK instructions. Thus the specified (n) number of tokens set by the SETCTL instructions are output from ARC6.

Figure 5-64

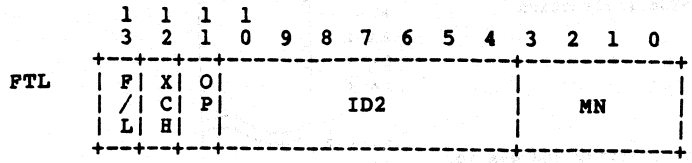


The assembler description is shown below.

```
LINK        ARC2=FOR1(ARC1);
LINK        ARC3, ARC4=FSET(ARC2) AT 30;
LINK        ARC5=FRED(ARC4);
LINK        ARC6, ARC7, ARC8=FCBK(ARC5,ARC7);

FUNCTION    FOR1=OR, RDCYCS(CODE,1);
FUNCTION    FSET=SETCTL(1, 1, 1101B);
FUNCTION    FRED=RDCYCS(BA,1);
FUNCTION    FCBK=COPYBK(1, 1), CNTGE(1) AT 30;

MEMORY      CODE=E000H;
MEMORY      BA=0;
```

5.4 OUT Instruction

These two instruction (OUT1 and OUT2) output the data processed within the uPD7281 to the outside world. The FTL field of the OUT instructions is shown below.



F/L: This bit determines whether the OUT instruction is to be executed alone or linked to an AG & FC instruction. F/L=0 indicates that the OUT instruction (usually OUT1) is to be executed alone. F/L=1 indicates a linked OUT instruction used in combination with an AG & FC instruction. In this case the FTR field is also significant.

XCH: This bit controls whether the A and B data are to be exchanged before they are input to the OQ (Output Queue). When XCH=1, exchange is performed and when XCH=0, exchange is not performed.

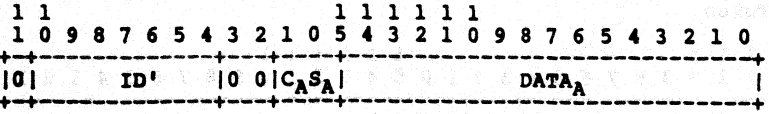
OP: This bit serves to distinquish the OUT1 and OUT2 instructions. OP=0 indicates an OUT1 instruction and OP=1 an OUT2 instruction.

ID2: This field is used by the OUT2 instruction when two tokens are output in succession to specify the ID of the second token.

MN: This field indicates the destination of the output token. The module number can be any value in the range 0 to FH so that 16 different destinations can be specified (module number 0 is usually the host). When the OUT2 instruction is executed to output the two tokens, they both take the module number specified by this field.

5.4.1 OUT1 Instruction

This instruction outputs a 32-bit token to the outside world. The configuration of the output token is shown in the following figure.



High word (first output) Low word (second output)

MN: The contents of this field are determined by the lower 4 bits of the FTL field.
ID': This is the ID field that is read by the LT reference.
C_A,C_A: These two bits are the C_A and S_A bits of the A data.
$DATA_A$: This is the first 16 bits of data of the token that executes this instruction.

OUT1 instruction assembler description

Figure 5-65

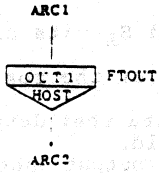


Figure 5-65 shows an example of using the OUT1 instruction to output the data of the tokens arriving from ARC1 to the host.

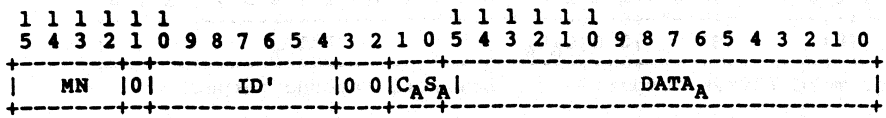
```
LINK        ARC2=FTOUT(ARC1);
FUNCTION    FTOUT=OUT1(mn, ident);
OUTPUT      ARC2
```

mn : The module number of the output destination should be described here. (The module number of the host is usually 0.)
ident : The ID of the output token should be described here. The specified value is then stored in the LT. The arc that is used by the OUT instruction to output tokens must first be defined by an OUTPUT statement.

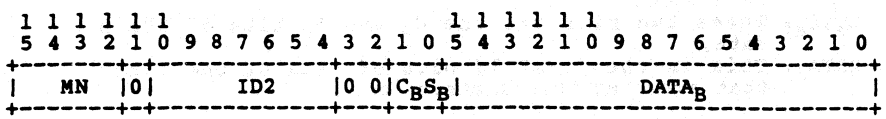
5.4.2 OUT2 Instruction

This instruction outputs two 32-bit tokens. The format of the output tokens is shown below.

First Token



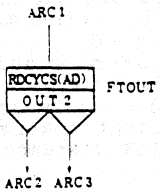
Second Token



MN: The contents of this field are determined by the lower 4 bits of the FTL field.
ID': The ID read by the LT reference is stored directly in this field.
C_A, S_A: These two bits are the C_A and S_A bits of the A data.
$DATA_A$: This is the 16 bits of A data of the token that executes this instruction.
ID2: This field stores the 7-bit data that determines the contents of the FTL ID2 field.
C_B, S_B: The C_B and S_B bits of $DATA_B$ are output unchanged.
$DATA_B$: This is the 16 bits of B data of the token that executes this instruction.

OUT2 instruction assembler description

Figure 5-66



In Figure 5-66, the A data from ARC1 and the data read from the DM are divided into two tokens and output to the output data bus (ODB). The first token contains the A data and the second token the B data.

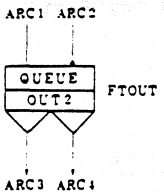
```
LINK        ARC2, ARC3=FTOUT(ARC1);
FUNCTION    FTOUT=OUT2(mn, id1, id2), RDCYCS(AD, 1);
MEMORY      AD=0;
OUTPUT      ARC2, ARC3;
```

mn : The MN for the tokens to be output both to ARC2 and ARC3 are described here. In other words, ARC2 and ARC3 both have the same MN.

id1 : This is the ID of the token output to ARC2. The value of id is stored in the ID field of the LT.

id2 : This is the ID of the token output to ARC3.

Figure 5-67



In the example in Figure 5-67, the tokens input from ARC1 and ARC2 are queued by the QUEUE instruction and then output. The first token that is output is an FTRC=0 token and is output with the A data from ARC1. The second token output is an FTRC=1 token and is output with the data of the ARC2 token.

Assembler description:

```
LINK        ARC3, ARC4=FTOUT(ARC1, ARC2);
FUNCTION    FTOUT=OUT2(mn, id1, id2), QUEUE(QUE1, 16);
MEMORY      QUE1=AREA(16);
OUTPUT      ARC3, ARC4;
```

The use of the mn, id1, and id2 parameters is as described in the preceding example.

Chapter 6

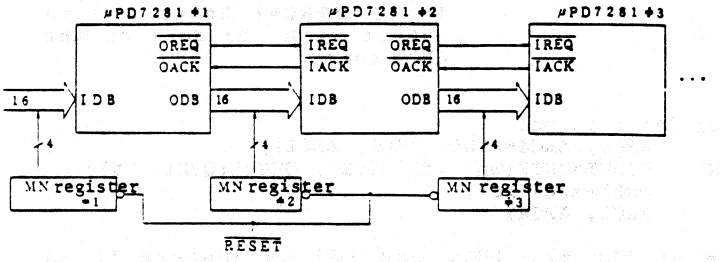
Multiprocessor Operation

The high speed processing realized by the adoption of just a single uPD7281 can be further enhanced by the use of several processors in a multiprocessor configuration.

6.1 Hardware Configuration

As shown in Figure 6-1, a multiprocessor configuration is realized simply by wiring the output lines of one uPD7281 to the input lines of another uPD7281. Note, however, that a multiprocessor configuration like this requires MN registers to set the module number of each processor.

Figure 6-1
Multiprocessor Configuration



Note: The outputs of the MN registers are active only when RESET is input.

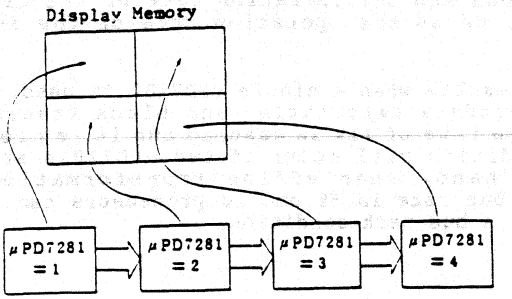
6.2 Software Configuration

The following two approaches are available when writing a program for multiple uPD7281s:

1. To partition the data areas to be processed
2. To partition the processing routines

Method (1) is effective when memory data is to be processed. In this method, all of the uPD7281s are loaded with the same program and the memory area to be processed is divided among them. Figure 6-2 shows the use of this method. Memory area division is effective for such operations as enlargement and reduction of the image as well as rotation or logical operations between images, filtering, etc.

Figure 6-2
Partitioning Data Areas



Method (2), on the other hand, is used when the data to be processed cannot be easily divided or when the entire program cannot be stored in a single processor. If we consider method (1) to be a parallel approach in which multiple data is processed simultaneously, method (2) is a parallel operation approach in which multiple processing is performed in pipeline fashion for a single data.

6.3 Multiprocessing and 'Bus Neck' (Memory Access Bottleneck)

In either of the two approaches described above, processing time decreases in a nearly inverse proportion to the number of uPD7281 processors employed. There is, however, one limit on the improvement in processing speed that can be achieved. This is the 'bus neck'. Although the uPD7281 can perform input/output of tokens at a high speed, if multiple processors are outputting tokens to the I/O bus in rapid succession, the bus will eventually become saturated. In other words, a bus neck condition will occur, degrading the efficiency of the internal processing. Bus neck is most likely to occur when the internal processing performed is simple and requires a large number of accesses to external memories (block transfer of image data, inversion, etc.).

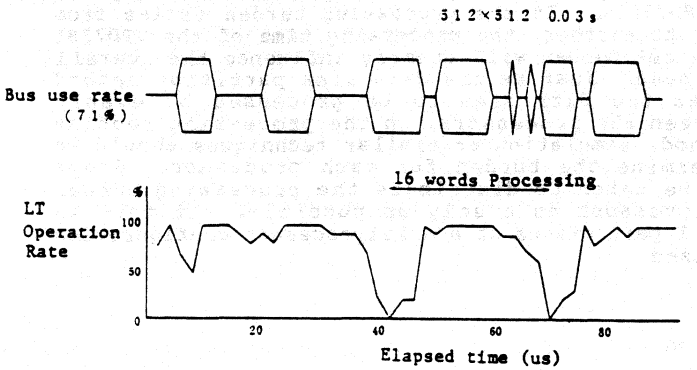
Conversely, if the internal processing is complicated and requires few external memory accesses (affine transformations, compression, extension, etc.), the use of the bus is reduced as is the chance that bus neck will occur. The number of processors that will cause a bus neck condition can be calculated by using simulation techniques to derive the use rate of the I/O bus under different conditions. Examples of this are shown in Figure 6-3.

Figure 6-3 shows the results of simulating three different types of operations: bit boundary calculation, block transfer, and affine transformations. Shown are the use rates for the bus and the operation rate of the LT (which can be thought of as the operation rate of the internal pipeline).

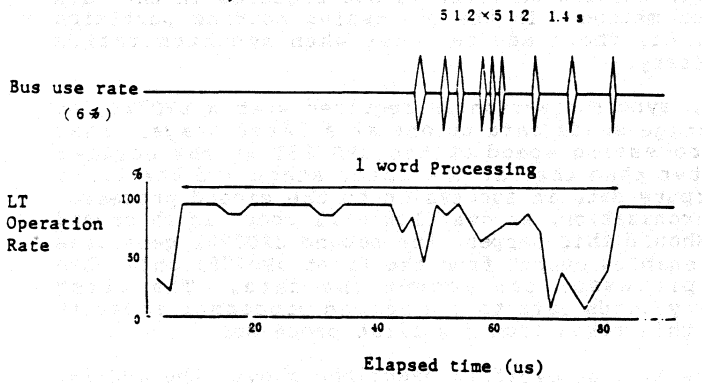
These are the results when a single uPD7281 is used. In the case of bit boundary calculation and block transfer, an existing bus use rate of 71% is assumed and it is clear that a bus neck condition will occur if two uPD7281s are used. On the other hand, when affine transformations are calcuated, the bus rate is 6% and 10 processors can be used without causing a bus neck condition.

Figure 6-3
Bus Use Rate (Using One uPD7281)

(Bit Boundary Block Transfer)



(Affine transformations)



6.4 Number of Processors Used and Performance

Provided that the number of uPD7281s used is restricted to prevent occurence of a bus neck condition, performance of a system adopting the data area partition method will improve in almost direct proportion to the number of processors employed. This is not, however, necessarily the case when division of processing routines has been opted for. This is because, depending on the program, cases may arise that require that timing of processoes be synchronized between individual uPD7281s. If the processing burden varies from one processor to another, the processing time of the uPD7281 with the heaviest burden will greatly influence the overall processing speed. Just as the data area partition method requires that the data area to be processed be evenly divided between the processors, in the processing routine division method, simulation or similar techniques should be used to determine the burden for each processor. Steps should then be taken to distribute the processing burden among the processors as evenly as possible. If this is done, the full performance of a multiprocessor configuration will be realized

6.5 Synchronization

Synchronization between uPD7281s is not required in the data area partition method. In the processing routine partition method, however, there may be cases when synchronization becomes necessary.

Specifically, synchronization is required when a uPD7281 at an earlier stage sends data to one at a later stage. That is, if the processing speed of the uPD7281 at the earlier stage is faster than that at the later stage and the first processor outputs data in succession to the second processor without synchronization, an overflow will occur in th second processor. Should this happen, the second uPD7281 generates a token that enables output from the first uPD7281 only when the second processor can accept the data. The first processor only sends data to the second processor after it has received this token from the first processor.

To prevent the kind of overflow described above, the uPD7281 is provided with an 'input restrict' function. If the mode set command is used to specify input restrict or input prohibit mode, input from outside the processor will be either restricted or prohibited when more than 24 levels accumulate in the DQ (data queue). This function effectively prevents DQ overflow. Note, however, that because the input restrict function operates only when there are more than 24 levels in the DQ. In other words, only DQ

overflow is prevented and there is no assurance against DMQ (DM queue) or GQ (Generator Queue) overflow. Therefore, synchronization between uPD7281s is always required in cases where the data input to the later stage processor is used directly to execute a QUEUE instruction.

Chapter 7

Interfacing with the Host

The uPD7281 is designed primarily as a peripheral processor for an 8- or 16-bit CPU and is normally accessed from the CPU by performing read/write operation to the uPD7281 as an I/O device. However, since the very concept of addresses is not employed in the uPD7281, some external circuits are required to support this access.

The functions that are required to interface the uPD7281 to the host computer are:

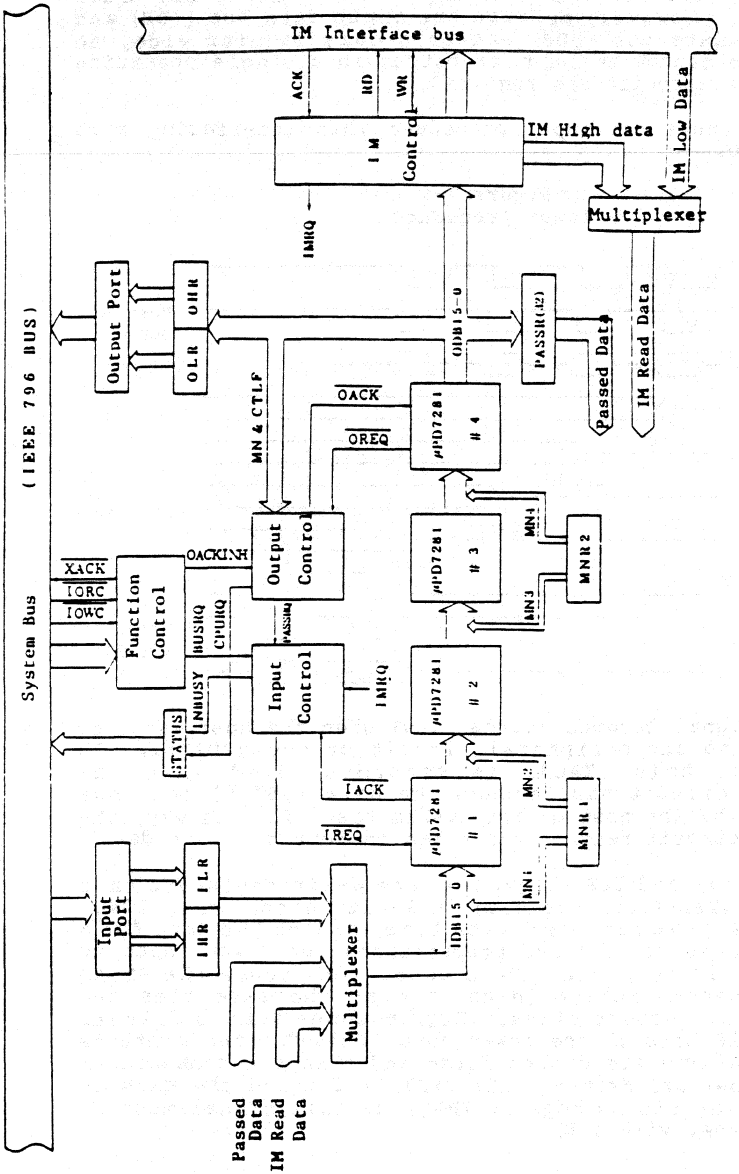
1. I/O ports
2. Function control (status, registers, MN registers, etc.)
3. uPD7281 input control
4. uPD7281 output control

A system configuration example that includes the above functions in addition to an image memory (IM) interface is shown in Figure 7-1.

In this interface configuration example, read/write of all I/O ports is performed by status polling. However, as required by the application, systems could also be configured that use the output data of the uPD7281 to generate interrupt requests to the host.

Figure 7-1

uPD7281 Peripheral Circuit Block Diagram

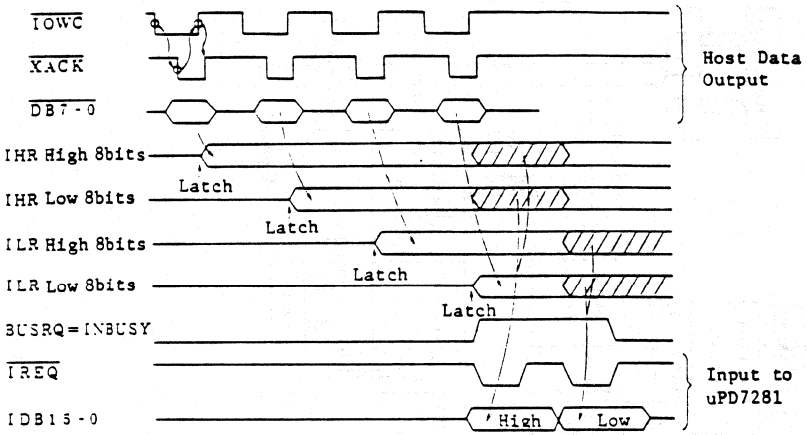


7.1 I/O Ports

The uPD7281 performs all data input/output in 32-bit units (one token). However, because the input data bus (IDB) and the output data bus (ODB) are both only 16 bits wide, an entire token cannot be input or output in a single operation and external circuits are required.

Figure 7-2 shows the input procedure when interfacing with an 8-bit CPU.

Figure 7-2
Input Procedure

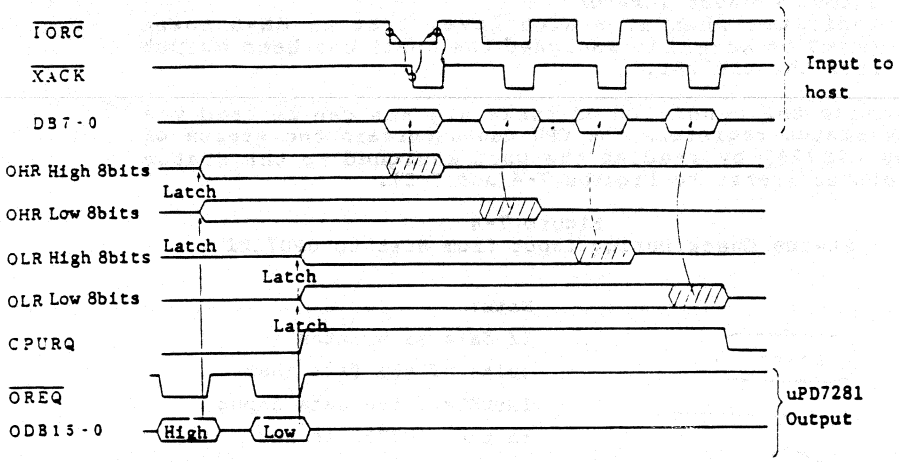


The IOWC/ signal becomes active (low) when the host wants to write data to the peripheral circuit of the uPD7281. In response to this, XACK/ is returned (low) when the peripheral circuit has fetched the data (DB7-0) that has been placed by the host on the system bus. In this way, the peripheral circuit fetches the token in four 8-bit reads.

IHR (high/low) and ILR (high/low) are 8-bit registers that latch the token (from the MSB) in 8-bit units. When IOWC/ has become active for the fourth time, all of the data for a single token is present and preparations for token input to the uPD7281 is complete. This condition causes the BUSRQ signal to become active (high) and at the same time the uPD7281 input request signal IREQ/ becomes active driving the higher 16 bits of the token onto IDB15-0. The contents of IHR (high/low) are driven first and then the contents of ILR (high/low) are driven. The uPD7281 fetches the data on IDB0-15 at the rising edge of IREQ/ so this signal must be driven together with IDB.

Figure 7-3 shows a procedure that can be used by the host to fetch data output by the uPD7281.

Figure 7-3
Output Procedure



When an MN=0 token is output from the uPD7281 (it is assumed here that MN=0 is assigned to the host), it is latched in the OHR and OLR registers. The low word of the token is latched in the OLR register and at the same time the CPURQ signal becomes active (high). At this point the output data from the uPD7281 is latched and can be read by the host at any time. Therefore, the host drives IORC/ active (low) after confirming that CPURQ is active. The peripheral circuit outputs the tokens in 8-bit units from the MSB. When the host has fetched the data four times, CPURQ becomes inactive (low) completing token fetch. The OHR and OLR registers latch the data from the uPD7281 at the rising edge of OREQ/.

7.2 Function Control

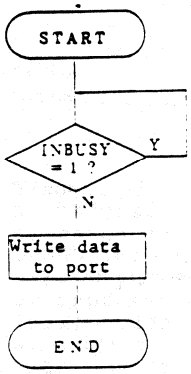
1. Status Check
   In this system, data input/output between uPD7281 and the host must be performed while monitoring the status of the processor. This is because the host must not set the next input data before the previous input completes and any data read operation performed when there is no output data from the uPD7281 would be meaningless.

These two status signals are related to input/output:

a. Input busy (INBUSY)
   Indicates when at a high level that input of the uPD7281 data set by the host in the input register (via the input port) has not completed.
b. Output request (CPURQ)
   Indicates when at a high level that an MN=0 token (MN=0 is normally assigned the host) has been output from the uPD7281.

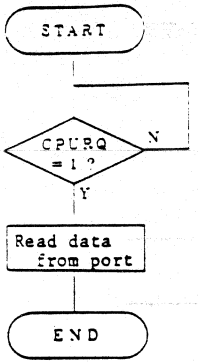
Because the condition of these signals can be read via the status register, the CPU can ascertain the status of the uPD7281 by reading the port assigned to the status register (refer to Figures 7-4 and 7-5).

Figure 7-4
Status Check During Input from Host to uPD7281



Note:
If data is written in spite of the fact that INBUSY=1, the data input to the uPD7281 is not assured.

Figure 7-5
Status Checking During Output from uPD7281 to Host



Note:
Data read when CPURQ=0 is meaningless.

2. MN Registers
Each uPD7281 can be assigned a unique module number (MN). The system must therefore provide MN registers (4 bits) to set the module number of each processor. The user has two choices here: (1) To provide the MN registers (times the number of uPD7281s) and then use a program to set, via the write ports, the module number data; or (2) to use DIP switches that put the module number data on the input data bus of each uPD7281 when RESET/ is input.

3. Input Control
The following three kinds of data can be input to uPD7281 No. 1 of a multiprocessor configuration shown in Figure 7-1. Control signal names are enclosed in parenthesis.
a. Data from the host (BUSRQ)
b. Data from the uPD7281 No. 4 (PASSRQ)
c. IM read data (IMRQ)
When any of the above types of data are present, the IREQ/ signal becomes active and the data is input to the uPD7281 (refer to Figure 7-6 and 7-7).

Figure 7-6
IREQ Generating Circuit Example

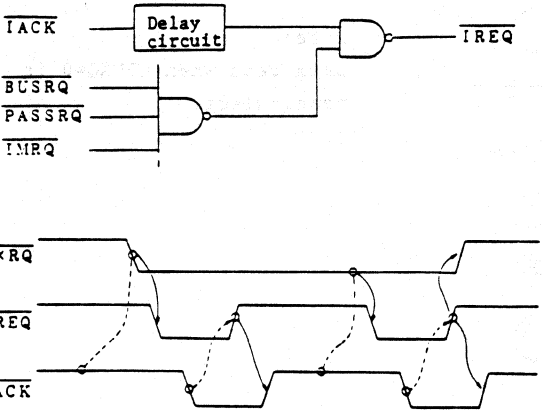
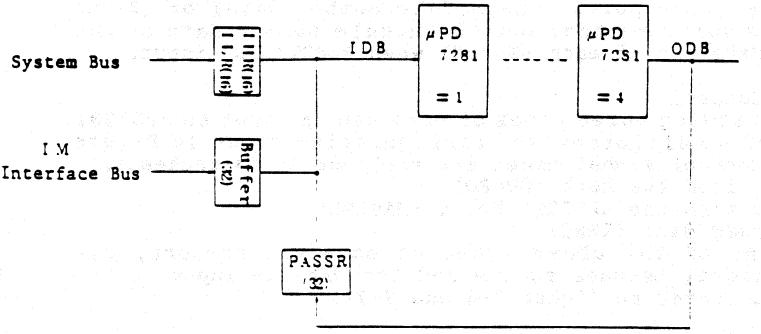
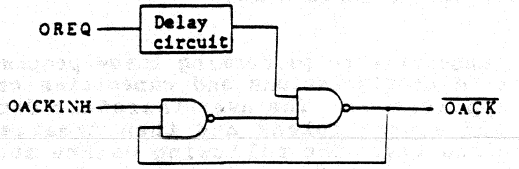


Figure 7-7
uPD7281 Input



4. Output Control
   The data output from uPD7281 No. 4 are divided into the following four types according to the destination MN.
   a. Output request data to host
   b. IM access data
   c. Pass data (reinput to uPD7281 No. 1)
   d. Vanish (delete) data
   The output request destination should be set so that it does not return OACK/ to the last-stage uPD7281 (uPD7281 No. 4 in the above example) while OACK INH is active (during output).

Figure 7-8
OACK/ Generating Circuit



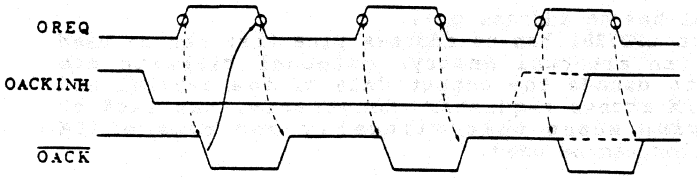
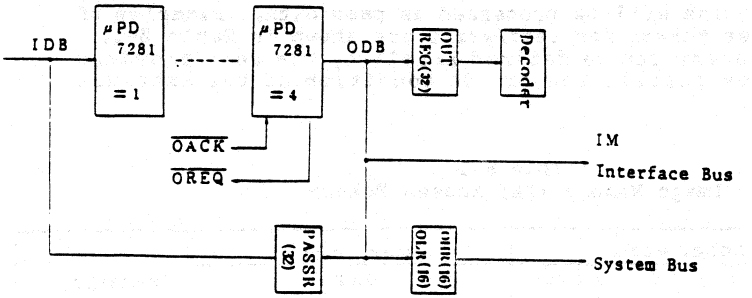


Figure 7-9
uPD7281 Output



Chapter 8

Accessing the Image Memory

An image memory (IM) is essential to performing image processing. Many different types of IM configurations and capacities can be used depending on the application. The user therefore defines the IM configuration and access tokens and then creates the appropriate external circuitry. The following points must be kept in mind when a system is configured that includes both uPD7281 processors and an IM.

1. The uPD7281 has no address pins
   Because the uPD7281 has no address pins that can be used to access an external memory, external circuits are required to decode the output data of the uPD7281 and generate IM access requests. Conversely, the lack of address pins means that virtually any type of IM configuration can be used.

2. IM access tokens must be defined
   In the same way that the MN (module number) defines the uPD7281, a MN should be assigned to the IM. External circuits should be provided so that when the MN of the token output from the uPD7281 is decoded, a token with the same MN as the IM will generate an IM request. Any other tokens will be processed as pass data. Examples of IM access tokens for read/write are shown in Table 8-1. These tokens can be defined freely by the user provided that they fulfill the decode condition of the external decoder.

Table 8-1
Image Memory (IM) Access Tokens

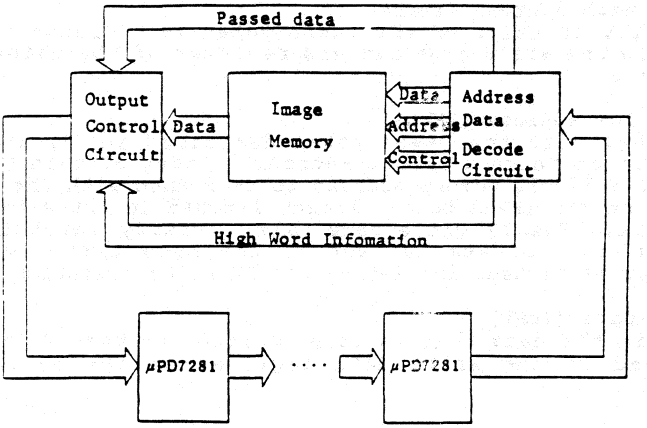
Token	Higher word				Lower word	Remarks
	MN		ID	CTLF	DATA	
	15 12	11	10 4	3 0	15 0	
IMRHA	0 1 0 0*	0	1 1 1 * * * *	* * * *	IM READ HIGH ADDRESS	Read instruction
IMRLA	0 1 0 0	0	←MN→ ←ID→	* * * *	IM READ LOW ADDRESS	
IMWHA	0 1 0 1	0	* 0 1 * * * *	* * * *	IM WRITE HIGH ADDRESS	
IMWLA	0 1 0 1	0	* 0 0 * * * *	* * * *	IM WRITE LOW ADDRESS	Write instruction
IMWD	0 1 0 1	0	* 1 0 * * * *	0 0 C S	IM WRITE DATA	
VAN	1 1 1 1	0	* * * * * * *	* * * *	* * * * * * * * * * * * * * * *	Data deleted
PASS	others	0	* * * * * * *	* * * *	* * * * * * * * * * * * * * * *	Data pass

*: In this example, memory module number is assumed to be 4 for read and 5 for write.

3. The IM and uPD7281s are connected in a ring configuration:
   Because the input and output buses of the uPD7281 are isolated, the processors must output IM access request tokens from the output side and input IM read data from the input side. Therefore, to realize efficient processing in a system that includes an IM, a ring configuration must be adopted (refer to Figure 8-1). This configuration allows the creation of a feedback loop from the later-stage processors to the earlier-stage processors and increases the flexibility with which programs can be written.

Figure 8-1 shows an example of a system using an IM that was designed taking the above points into consideration.

Figure 8-1
Image Memory Connection Example



8.1 IM Access Tokens

As mentioned earlier, as long as the IM access token fulfills the decode condition of the IM decoder, the token itself can be defined as desired by the user. These are descriptions of five different tokens from Table 8-1:

1. Read high address (IMRHA)
   If only the 16-bit data of the token output from the uPD7281 is used to address the memory, the size of the memory will be limited to 64K bytes. However, if the

address is divided between two tokens into low and high address words, 32-bit addresses can be generated. Further, the high address is defined by type as a read or write address. This is because there are many cases in which it is desirable to use separate banks (the plane defined by the high address) for the memory to be processed and for the memory used to store the processing results (refer to section 8.2, IM Read/Write Procedures). The only function of the IMRHA token is to latch input data in the read high address register of the external circuit.

2. Read low address (IMRLA)
   Together with the IMRHA token described in (1) above, this token forms the 32-bit address. When this command is input, a 32-bit address is output that combines the data of this token with that of the IMRHA token already latched in the external circuit. At the same time, a read request is output, initiating the read operation.

3. Write high address (IMWHA)
   The only function of the IMWHA token is to latch input data in the write high address register of the external circuit.

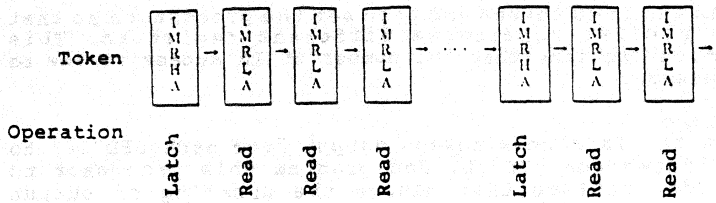
4. Write low address (IMWLA)
   Together with the IMWHA token described in (3) above, this token forms a 32-bit address. Input of this token causes a 32-bit memory address to be formed with the data field of the IMWHA token already latched in the external circuit. The IM write data (also already latched) is output to the data bus and at the same time a write request is issued, initiating the IM write operation

5. Write data (IMWD)
   This is the data that is to be written to the IM and is latched in the write data register of the external circuit.

8.2 IM Read/Write Procedure

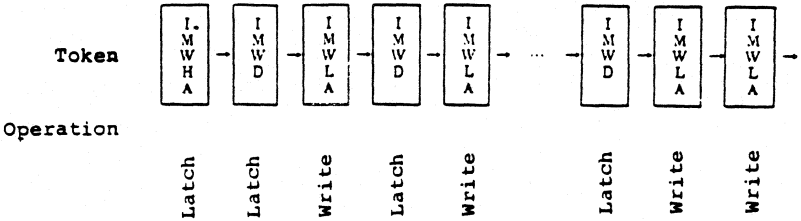
The procedure described here is based on the IM access tokens defined in section 8.1.

1. IM read
   Once an IMRHA token has been set, it remains valid until the next IMRHA token arrives; the high word of the address therefore remains unchanged.



Each time an IMRLA token arrives, the contents are read from the IM address formed by the high address set by IMRHA and the low address specified by the arriving IMRLA token.

2. IM write
Once the IMWHA and IMWD tokens have been set, they remain valid until the next IMWHA or IMWD token arrives; the high word of the address and the write data remain the same.



Each time an IMWLA token arrives, the data specified by IMWD is written to the IM address formed by the high address set by IMWHA and the low address set by the arriving IMWLA token.

8.3 IM Access in a Multiprocessor Configuration

Although there is no difficulty when a single uPD7281 accesses the IM, a problem does arise in a multiprocessor configuration if IM access tokens are output by more than one uPD7281. What happens is that a IMRHA token output by a different uPD7281 will be interposed between the IMRHA and IMRLA tokens output by the first uPD7281 processor. The same danger exists in the case of the IMWHA, IMWD, and IMWLA tokens. The following solution, one using hardware and the other software, are available.

1. Hardware
Provide multipe registers to store the data of the IMRHA,

IMWHA and IMWD tokens and then set the processors so that each uPD7281 references different registers. This solution requires that the number of IM access tokens be increased.

2. Software
   Input the IM access tokens output from each uPD7281 to the last-stage uPD7281 and program this processor to include routines that manage the ordering of output tokens.

Appendix A

Flowgraphs

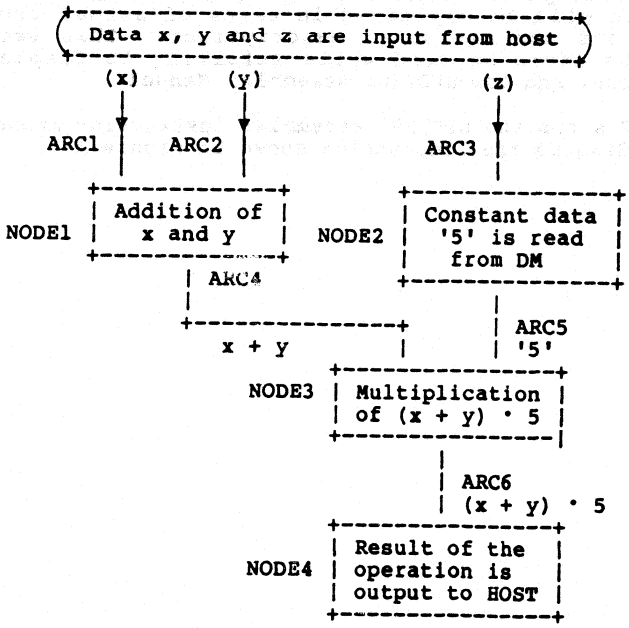
Flowgraphs are used when creating uPD7281 programs. These correspond to the flowcharts used writing for conventional processors.

As shown in Figure A-1, flowgraphs are composed of 'arcs' and 'nodes'. Arcs indicate the flow of data to be processed by the uPD7281 and nodes indicate the type of processing to be performed with the uPD7281.

A.1 Flowgraph Explanation (outline)

Figure A-1 shows the flowgraph when a uPD7281 processor is used to execute the expression (x + y) x 5.

Figure A-1



Each step of the flow is explained below:

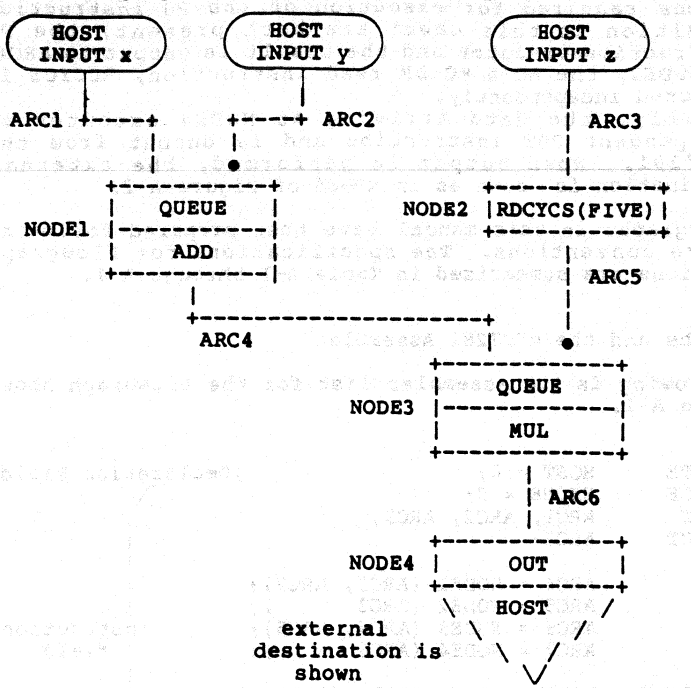
1. Data x, y and z are input from the host via the input terminal to ARC1, ARC2 and ARC3 respectively.
2. The input data proceeds along the arc to the intended node for the indicated processing. NODE1 performs the addition of x and y, and NODE2 reads constant data '5' from the data memory.
3. The results of the processing performed at NODE1 and NODE2 are connected to the next node by ARC4 and ARC5. The data that flows to ARC4 is the result of (x + y), and that flowing to ARC5 is the data read ('5') from the data memory.
4. ARC4 and ARC5 input to NODE3. When the two data (the result of (x + y) and constant data '5' read from the DM arrive at NODE3, the multiplication operation (x + y) x 5 is performed.
5. The result obtained at NODE3 is then output to ARC6.
6. From ARC6 the processing moves to NODE4 where the result of the operation is output from the uPD7281 to the outside world. Processing terminates when the result is output from this node.

A.2 Flowgraph Functions

In the paragraphs that follow, the operations described to this point will be explained in terms of actual uPD7281 instructions and assembler descriptions. This section should therefore be read while referring to Chapter 5, Instructions, and the uPD7281 Assembler Manual.

Figure A-2 shows the uPD7281 assembler instruction mnemonics corresponding to the computation shown in Figure A-1.

Figure A-2
Flowgraph with Assembler Descriptions



1. First, variables x, y and z are input from the host. To input data to a uPD7281, the format shown in Figure A-2 is used. In this example, the input is from the host. However, input from an image scanner or another uPD7281 connected in cascade is also possible.
2. The input variable data x, y and z then proceed to the nodes via the specified arcs. Note that a black circle at the input to a node (for example, from ARC2 to NODE1 or from ARC5 to NODE3) indicates an FTRC=1 token.
   The input to a node can therefore be either FTRC=0 or FTRC=1; the operation indicated by these two types of tokens is different.
3. The data arriving at a node executes the instruction indicated by the node. In a combination of two instructions (F/L=1; see NODE1 and NODE3), the AG & FC instruction appears in the upper half (input side) of the node.
   NODE2 and NODE4 are examples of instructions that are used independently (F/L=0, SEL=11). In this case, only that instruction appears in the node.

For example, tokens arriving at NODE1 or NODE3 first execute the AG & FC instruction QUEUE. When the two tokens required for execution of the PU instruction (addition in this case) are both present, the PU instruction executes and the result is output to ARC4. In NODE2, the AG & FC DM read instruction, RDCYCS is executed independently.

4. Finally, the data arriving at NODE4 executes an independent OUT instruction and is output from the uPD7281. When output is performed, the external destination is shown as in NODE4 of Figure A-2.

The flowgraphs in this manual have been prepared following the above conventions. The specifications for flowgraph descriptions are summarized in Table A-1 through A-4.

A.3 Flowgraphs and the uPD7281 Assembler

The following is the assembler list for the flowgraph shown in Figure A-2.

```
 1.  EQUATE     HOST = 0;                    )Declaration Field
 2.  MODULE     EXONE = 8;                   ⌐
 3.  INPUT      ARC1, ARC2, ARC3;            |
 4.  OUTPUT     ARC7;                        |
                                             |
 5.  LINK       ARC4 = NODE1 (ARC1, ARC2);   |
 6.  LINK       ARC5 = NODE2 (ARC3      );   |
 7.  LINK       ARC6 = NODE3 (ARC4, ARC5);   Instruction
 8.  LINK       ARC7 = NODE4 (ARC6      );     field
                                             |
 9.  FUNCTION   NODE1 = ADD, QUEUE (QUE1, 1);|
10.  FUNCTION   NODE2 = RDCYCS     (FIVE, 1);|
11.  FUNCTION   NODE3 = MUL (Y), QUEUE (QUE2, 1);
12.  FUNCTION   NODE4 = OUT1       (HOST, 0);|
                                             |
13.  MEMORY     QUE1 = AREA (1);             |
14.  MEMORY     QUE2 = AREA (1);             |
15.  MEMORY     FIVE = 5;                    ⌙
                                             ⌐
16.  START;                                  |
17.  DATA       EXEC (EXONE, ARC1, 2);       |
18.  DATA       EXEC (EXONE, ARC2, 3);       | Data
19.  DATA       EXEC (EXONE, ARC3, 0);       | Field
                                             ⌙
20.  END
```

First compare the assembler list against the example in Figure A-2. The first to fourth lines of the assembler description have no direct bearing on the flowgraph. The execution section of the list, which follows the declarations, is the part of the list most directly related

to the flowgraph. The data section is the part of the list that starts execution of the program.

A.3.1 Arcs and LINK Statements

The arcs described in the previous section correspond directly to the LINK statements.

```
LINK        ARC4 = NODE1 (ARC1, ARC2);
  |           |       |      |     |
  |           |       |      |     +- Arc input to NODE1
  |           |       |      |          (FTRC=1 side)
  |           |       |      +- Arc input to NODE1
  |           |       |           (FTRC=0 side)
  |           |       +- Node output to ARC4
  |           +- Output arc name
  +- Pseudo-instruction (secures link table (LT) area)
```

Note, however, that because ARC1, ARC2 and ARC3 are input from the host, they are not defined by LINK statements but are assigned input values by the DATA statements.

A.3.2 Nodes and FUNCTION statements

The nodes described in the previous section correspond directly to the FUNCTION statement.

```
FUNCTION    NODE1 = ADD, QUEUE (QUE1, 1);
  |           |       |      |     |
  |           |       |      |     +- Parameters for AG &
  |           |       |      |          FC instruction
  |           |       |      +- AG & FC instruction of node
  |           |       +- PU instruction of node
  |           +- Node name
  +- Psuedo-instruction (secures function table (FT)
       area)
```

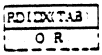
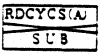
A.3.3 MEMORY Statements

The MEMORY statements do not appear directly in the flowgraphs in the previous section. Simply, they are used to define the data memory (DM).

```
MEMORY      QUE1 = AREA (1);
  |           |       |
  |           |       +- Size of area in DM equal to one
  |           +- Name (label) of area in DM
  +- Pseudo-instruction (secures and sets the contents
       of the DM)
```

Table A-1
Flowgraph Description Specifications (1)

Node (FUNCTION)

Node	
	Indicates an external input terminal. The numeric value in parentheses indicates the order of token input and is omitted when the input order doesn't matter. In addition to input from the host, input from another uPD7281 or from an image scanner, etc., is also possible.
	Indicates a data processing node which contains an independent instruction (F/L=0 or SEL=11), i.e., the processing performed by a single PU, GE, OUT, or AG & FC instruction. In the case of the OUT instruction, however, the description must conform to the specifications of the OUT instruction.
	Indicates a linked instruction (F/L=1), that is, when a token executes two instructions in a single circulation of the pipeline. The most common combinations are AG & FC instructions and PU, GE, or OUT instructions.
 (PU inst. only)	When CNOP=1, the node described in the dotted lines is either executed or is processed as NOP according to the condition of the C bits. If the CA and CB bits of the input data agree, the instruction is executed. If they do not, the instruction is processed as NOP.
	When the XCH bit is '1', the A data and B data input to the PU are exchanged. Special care should be taken when the exchange function is used with a SUB or shift instruction.

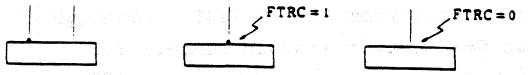
	Indicates an OUT1 instruction node. Used when the uPD7281 outputs a token to the outside world. Among the possible destinations are the host, the image memory, or another uPD7281. The output destination is described in the inverted triangle.
	In the case of the OUT2 instruction, two tokens are output from the uPD7281. Although the output destination (module number) for these two tokens (host, IM, another uPD7281, etc.) must be the same, the data that they contain is different and for this reason they are shown in two separate inverted triangles.

Table A-2 Flowgraph Description Specifications (2)

Node input/output format (junction of nodes and arcs)

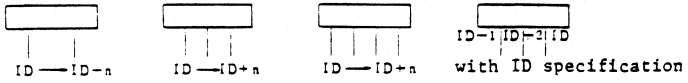
° Input

FTRC=1 data input to a node from an arc is indicated with a black dot as shown below.

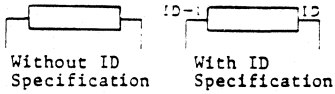


° Output

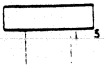
When there is more than one output from a node, they are (from left) ID, ID+1, ID+2, etc., unless otherwise specified.
If there is only one output, ID is assumed.



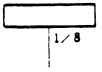
When BRC=1 (indicating branch control by the C bit), the output arcs are shown on the left and right sides of the node.
Unless otherwise specified, the output on the left is ID and that on the right is ID+1.



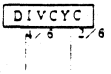
When an AG & FC instruction is used to change the number of output from a node or to modify the output ID, the flow should render these changes understandable.



When one out of every n tokens is output to ID+1 (in the example n=5). Example instructions are RDCYCS, COUNT, PICKUP, etc.



When one out of every n tokens is output to ID (in the example n=8). Example instructions are WRCYCL, WRCYCS, etc.

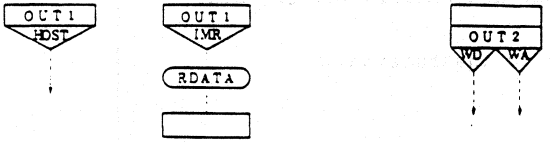


Divide cyclic instruction.

In other cases, the flowgraph description format is altered as necessary.

Table A-3 Flowgraph Description Specifications (3)

The OUT instruction is used to output tokens from the uPD7281 and for this reason is shown by a dashed line. For operations such as image memory read (IMR) where the token returns again to the uPD7281, the OUT node is connected to the next node with a dashed line.
When a termination or similar token that does not return to the processor is output, this is shown with a dashed line and an arrow.



When the SAVE instruction is used, the ID of the output token varies according to the value of IDSR (ID stack register). This is shown below.

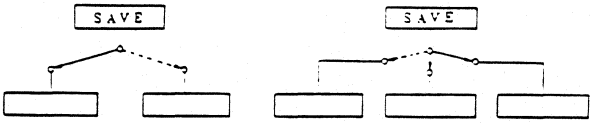
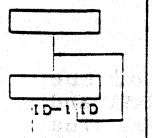
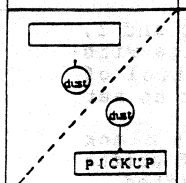


Table A-4 Flowgraph Description Specifications (3)

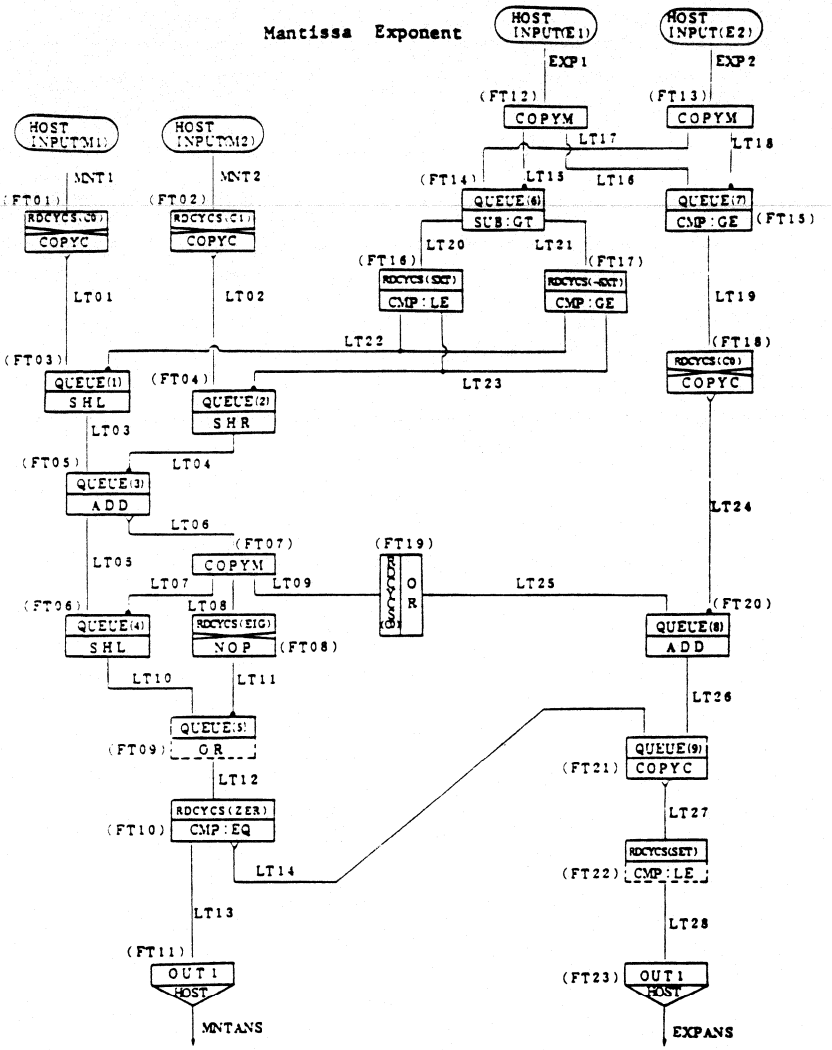
	When two arcs merge, one of the arcs is marked with an arrow.
	When tokens are to be deleted, normally either an FTRC=1 token input to a PICKUP or COUNT instruction is used.
SUB:EQ CMP:GT MUL:LE	When a PNZ condition is attached to the operation performed by a PU instruction, the name of the operation is shown on the left and the PNZ condition on the right.

A.4 Flowgraph Examples

Figure A-3 shows the flowgraph for floating-point arithmetic addition.

Explanation:

1. The input terminals are, for the mantissa of the addends, MNT1 and MNT2, and for the exponents, EXP1 and EXP2. These are defined by the INPUT statements. The order of token input in not specified.
2. The C bits of the mantissa tokens are set to 0 and 1, respectively. This is done by first writing data with the C bit set to the DM and then using the control of the combination of the XCH and OUT instructions to set the C bit of the arriving data and output it.
3. FTRC=1 tokens input to a node are marked with a black circle. The FTRC=1 token indicates the number of shifts and the FTRC=0 tokens contains the data to be shifted.
4. If no output arc is specified, the outputs are (from left) ID, ID+1 and ID+2. At FT07 the first output (ID) is to LT07, the second (ID+1) is to LT08, and the third (ID+2) to LT09.
5. When there is an XY output, the X output is always ID (first output) and Y is always ID+1 (second output). Therefore, in the case of an XY output, ID is assumed even if no output is specified.
6. When the C bit NOP condition is used, the PU instruction to be processed as NOP (if the C bits do not agree) is shown within dashed lines. Here, if the C bits do not agree, the CMP instruction is not executed.
7. When the BRC bit is used to control output, arcs are connected to both terminals of the node. In this case, if the result of subtracting the FTRC=1 data from the FTRC=0 data is greater than 0, the C bit will be '1' and only the ID+1 token will be output. If the result is less than 0, only the ID token will be output.
8. When the OUT instruction is used, the token destination is shown within the inverted triangle with the output terminal. The terminal name must have been defined by an OUTPUT statement.



Note:
Node names are described in parentheses.

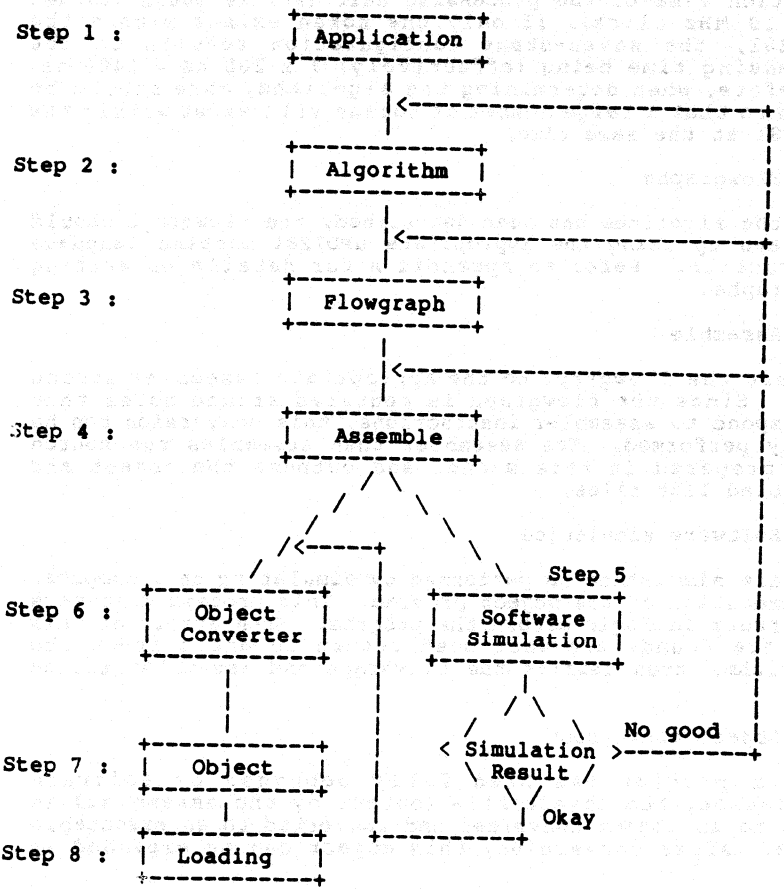
Fig. A-3 Floating-Point Addition Flowgraph

Appendix B

uPD7281 Development Support Tools

Because the architecture of the uPD7281 is completely different from that of a conventional microprocessor, new methods of program development must also be employed. These methods are presented here. Figure B-1 shows a flowchart for typical uPD7281 program development.

Figure B-1
Program Development Flow



Step 1 : Application

The application for which the uPD7281 is to be used is first determined.

Step 2 : Algorithm

Next the optimum algorithm for the application is settled upon. This process must be carried out taking the architecture of the uPD7281 into consideration. In particular, attention must be paid to the concurrency of the processing. Because of the seven-stage pipelined configuration of the uPD7281, if a large number of data tokens do not exist simultaneously in the processor, full concurrency is not realized. For example in the case of multiplying two 16-bit values, we see that since the execution time of the processing unit (PU) is 200ns (target with 10 MHz clock), if only one token exists within the uPD7281, the seven-stage configuration results in the processing time being (effectively) 7 x 200 ns = 1400 ns. Therefore, when determining the algorithm, care should be taken so that a large number of tokens will exist within the uPD7281 at the same time.

Step 3 : Flowgraphs

Once the algorithm has been determined, the flowgraph should be drawn up using the appropriate uPD7281 machine language instruction. Refer to Appendix A for details on writing flowgraphs.

Step 4 : Assemble

Convert the flowgraph to the appropriate assembler source code. Since the flowgraph is centered around nodes that correspond to assembler instructions, this conversion can be easily performed. The assembler then assembles the source code prepared in this manner and outputs the object and assembled list files.

Step 5 : Software simulation

Software simulation is performed by simulating on a computer the execution of the object program. This process discovers any errors in logic within the program. When errors of this type are found, the user must return to and correct the algorithm, then rewrite the flowgraph and assembler source code.

Step 6 : Object conversion

When a program has been fully debugged by software simulation, the object file (output by the assembler) is input to an object converter and converted to an executable image. After conversion, this object can be expanded in

RAM, written into ROM, or inserted in the host processor assembler or compiler source program.

Step 7 : Object

The output of the object converter is either in HEX* format or hexadecimal ASCII format. The former is used for expansion in ROM or RAM and the latter is used for insertion in the source code of an assembler or compiler

* HEX format object
The format for the HEX object code is shown below:

DR : Data record
EOFR : End of record

Date record format

: XX XXXX XX DD...DD SS
1 2 3 4 5 6

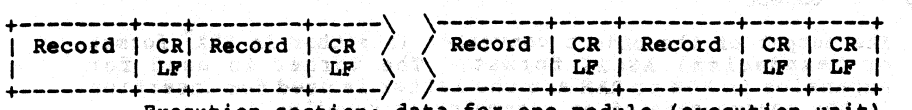
1: Record mark; indicates the start of a record.
2: Number of bytes of object data. It is '00' in the case of the last record.
3: Indicates the relative address within the current module of the first object data 5 of this record. It is '0000' in the case of the last record. The starting address of the current module is specified by the ADDRESS control.
4: Record type, '00' indicates a data record, '01' the last record.
5: Object data; this field is omitted in the case of the last record.
6: Checksum

Example:

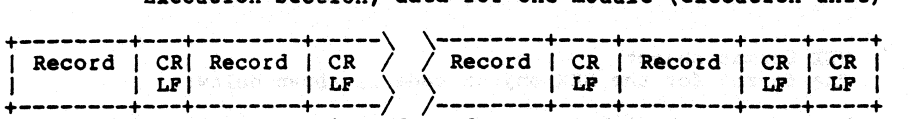
```
:10017000810000180D0000280E0000280F00002D7
:10018000810000280D0000380E0000380F00003C3
:10019000810000380D0000480E0000480F00004AF
:0401A00081000004D6
:0401A4002A540000D9
:00000001FF
```

ASCII format object:

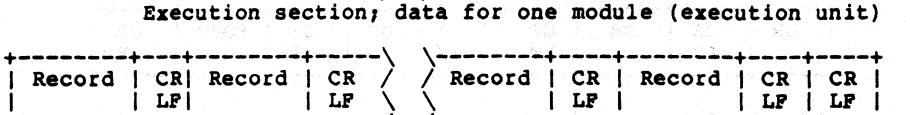
The format for the ASCII object code is shown below.



Execution section; data for one module (execution unit)

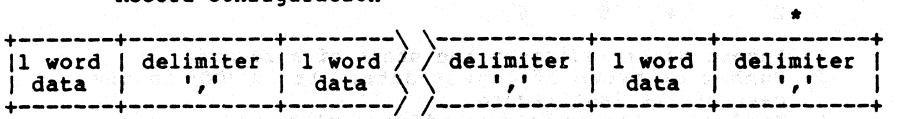


Execution section; data for one module (execution unit)



Data section

Record configuration



Maximum 16 words of data

1 word data : Indicates the format in which a single word (16 bits) of data is expressed 0XXXXH (Six ASCII characters). One word (16 bits) of data is four hexadecimal digits.

Delimiter (',') : Delimits words of data

* No delimiter is attached to the last word of the last record of the module.

Example:

```
08456H, 0F45DH, 08FF75H, 03201H,
08732H, 05ABCH,
09456H, 0934DH, 09724H, 00051H
```

Step 8 : Loading

Loading of the uPD7281 object is accomplished by loading the program either into ROM or inserting it into the host program. Execution of the program, however, is only enabled when is has been set in the program area of the uPD7281.

B.1 Assembler

The assembler description of the uPD7281 is completely different from that of a conventional assembler. The source programs that are assembled are function descriptions that reflect the data flow architecture of the uPD7281.

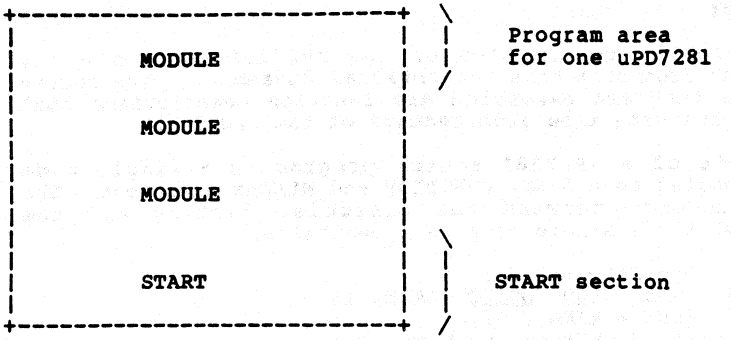
The basis of a uPD7281 source program is a single node corresponding to a LINK, FUNCTION and MEMORY statement. The correspondence between the assembler listing and the flowgraph for a single node is shown below.

```
LINK      C = FADD (A, B);
FUNCTION  FADD = ADD, QUEUE (BASE, 16);
MEMORY    BASE = AREA (16);
```
(The underlined strings are keywords)

This example shows the program for the expression C = A + B. The LINK statement defines the input and output for a single token. In this example, tokens A and B are input and token C is output. The FUNCTION statement defines the function of the node. AG & FC instructions may be combined with other instructions. FADD is a user-assigned node name. PU, GE, OUT or AG & FC instructions can be described in the FUNCTION statement. MEMORY statements are used to secure and define initial values for DM areas. In this example, the AG & FC instruction requires 16 words of DM memory area. The MEMORY statement, therefore, secures 16 words of memory.

In this way, each node of the flowgraph can be converted to LINK, FUNCTIION and MEMORY statements. It is also possible to write source programs for the uPD7281 that use multiple uPD7281s. In this case the program for a single uPD7281 begins with a MODULE statement. This statement is also used when overly techniques are used for a single processor in which case the overlay programs within the main program are each delimited with a MODULE statement.

uPD7281 programs cannot simply be loaded into the internal program area and executed; after the program has been loaded an execution token must be input to start execution. This token can be written in the START section of the assembler source program. (This section is not a part of the program but is used to define execution tokens and processing data; only one START section may be described per source program.) The token written in the START section is included in the object output by the assembler and can be sent from the host processor to the uPD7281 to initiate execution.

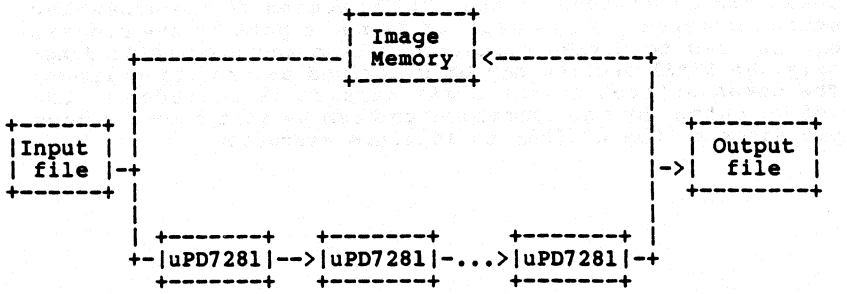


In this way, the uPD7281 assembler assembles the source program and outputs the object file and the assembler list. The output object file can be input to a software simulator or an object converter. The assembler list includes the source, error, disassemble, object and cross reference lists. The disassemble list is the object program in mnemonics listed in the order of physical address. This list is an important aid for program debug.

B.2 Software Simulator

The object program output by the uPD7281 assembler can be simulated on a computer using a software simulator. The software simulator is not only capable of simulating a system of multiple cascade-connected uPD7281 processors, but can also simulate uPD7281s connected in a ring configuration with an image memory. Figure B-3 diagrams a basic simulation system.

Figure B-3
Basic Simulation System



Modules starting with MODULE statements are loaded into each uPD7281 simulator. The module with the START statement is loaded into the input file. When the simulator starts operation, data is sent one at a time from the input file to the uPD7281s; the output tokens are written to the output file. When an image memory is used, read and write data can be sent to the memory to perform read and write operations.

Program debug using simulation is performed by dumping the contents of the output file and tracing the internal latches of the uPD7281s. Using the simulator allows the user to check the following:

1. Program logic errors.
2. Execution time
3. PU (Processing Unit) of LT (Link Table) operating rate.

Of these, the PU or LT operation rate is a particularly important part of program evaluation because a program with a low PU or LT operation rate still has room for improved performance.

Note: Internal operations of the uPD7281 are simulated closely by the simulator. Operations outside of the uPD7821, however, are controlled freely by the simulator and there is a danger that the timing will differ from that of an actual system. In this sense the execution time of the simulator must be regarded as only a reference value.

B.3 Object Converter

After its operation has been confirmed by the simulator, the object program output by the uPD7281 assembler is loaded into the system. However, because the object program must be written to the program area of the uPD7281, the object converter should be used. The following two formats are available for loading:

- ROM (RAM)
- Host program insert

The object converter outputs the following two kinds of files:

- HEX format object
- ASCII format code

Note that from the point of view of the host processor, the object program is considered data. For this reason, the memory image when the HEX format object is expanded contains not only the object code but also the code size data, attributes, etc., The host processor translates these values, and send the object code to the uPD7281 as tokens.

In the case of ASCII format code, the contents can be entered directly as source data in the application program of the host processor so that with minor changes it can be inserted in the assembler or compiler source.

A

μPD7281

IMAGE PIPELINED PROCESSOR

(IMPP)

ELECTRICAL SPECIFICATION

Electrical Specifications

Absolute Maximum Ratings (Ta = 25°C)

Parameter	Symbol	Test Conditions	Ratings	Unit
Power Supply Voltage	V_{DD}		-0.5 to +7.0	V
Input Voltage	V_I		-0.5 to +7.0	V
Output Voltage	V_O		-0.5 to +7.0	V
Operation Temperature	T_{opt1}	air flow: 2m/sec	0 to +70	°C
	T_{opt2}	no air flow	0 to +45	°C
Storage Temperature	T_{stg}		-65 to +150	°C

DC Characteristics (Ta = 0 to +70°C, V_{DD} = +5V + 10%)

Parameter	Symbol	Test Conditions	MIN.	TYP.	MAX.	Unit
Input Voltage Low	V_{IL1}	Except CLK Input	-0.5		0.7	V
	V_{IL2}	CLK Input	-0.5		0.45	V
Input Voltage High	V_{IH1}	Except CLK Input	2.2		V_{DD}+0.5	V
	V_{IH2}	CLK Input	3.5		V_{DD}+0.5	V
Output Voltage Low	V_{OL}	I_{OL}=2.0mA			0.45	V
Output Voltage High	V_{OH}	I_{OH}=-400uA	2.4			V
Input Leakage Current	I_{LI}	$0V \leq V_I \leq V_{DD}$			±10	uA
Output Leakage Current	I_{LO}	$0V \leq V_O \leq V_{DD}$			±20	uA
Power Supply Current	I_{DD}			320	500	mA

Capacitance (Ta = 25°C)

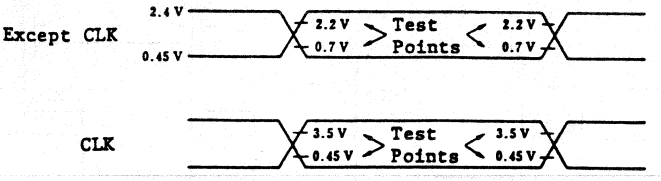
Parameter	Symbol	Test Conditions	MIN.	TYP.	MAX.	Unit
Clock Capacitance	C_K	f_C = 1MHz Unmeasured pins returned to 0V			20	pF
Input Capacitance	C_I				10	pF
Output Capacitance	C_O				20	pF

AC Characteristics (Ta = 0 to +70°C, V_{DD} = +5V ± 10%)

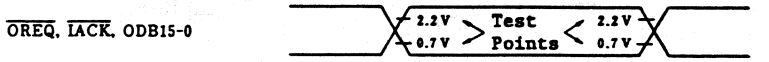
Parameter	Symbol	Test Conditions	MIN.	MAX.	Unit
Clock Cycle Time	t_{CYK}	Test points: 2V	100	500	ns
Clock High Level Width	t_{WKH}		40		ns
Clock Low Level Width	t_{WKL}		40		ns
Clock Rise Time	t_{KR}			10	ns
Clock Fall Time	t_{KF}			10	ns
$\overline{IACK}$ Active Delay Time from $\overline{IREQ}$	t_{DIAL1}	Higher 16-bit data	20	50	ns
$\overline{IACK}$ Inactive Delay Time from $\overline{IREQ}$	t_{DIAH1}		20	55	ns
$\overline{IACK}$ Active Delay Time from $\overline{IREQ}$	t_{DIAL2}	Lower 16-bit data	20	70	ns
$\overline{IACK}$ Inactive Delay Time from $\overline{IREQ}$	t_{DIAH2}		20	70	ns
$\overline{IREQ}$ Hold Time from $\overline{IACK}$	t_{HIQ}		15		ns
$\overline{IREQ}$ Rise Time	t_{IQR}			10	ns
$\overline{IREQ}$ Fall Time	t_{IQF}			10	ns
Input Data Set Time to $\overline{IREQ}$	t_{SID}		40		ns
Input Data Hold Time from $\overline{IREQ}$	t_{HID}		0		ns
$\overline{OREQ}$ Inactive Delay Time from $\overline{OACK}$	t_{DOQH}		15	35	ns
$\overline{OREQ}$ Active Delay Time from $\overline{OACK}$	t_{DOQL}		15	45	ns
$\overline{OACK}$ Delay Time from $\overline{OREQ}$	t_{DOA}		15		ns
$\overline{OACK}$ Rise Time	t_{OAR}			10	ns
$\overline{OACK}$ Fall Time	t_{OAF}			10	ns
Output Data Delay Time from $\overline{OREQ}$	t_{DOD}			25	ns
Output Data Float Delay Time from $\overline{OREQ}$	t_{FOD}		10	35	ns
$\overline{RESET}$ Low Level Width	t_{WRST}		$6t_{CYK}$		ns
$\overline{RESET}$ Recovery Time	t_{RVRST}		$2t_{CYK}$		ns
Module Number Delay Time from $\overline{RESET}$	t_{DMD}			$2t_{CYK}$	ns
Module Number Hold Time from $\overline{RESET}$	t_{HMD}		0		ns
$\overline{RESET}$ Delay Time from CLK (1)	t_{DRST}		0	$1/2t_{CYK}$	ns

Note1: This specification may be not satisfied when a uPD9305 is used. Resets should be repeated as the same times as the number of the uPD7281Ds which are connected to the uPD9305. This causes to set a correct module number.

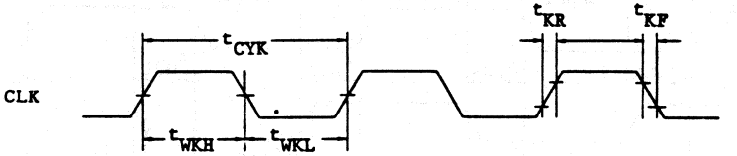
AC Test Input Waveforms



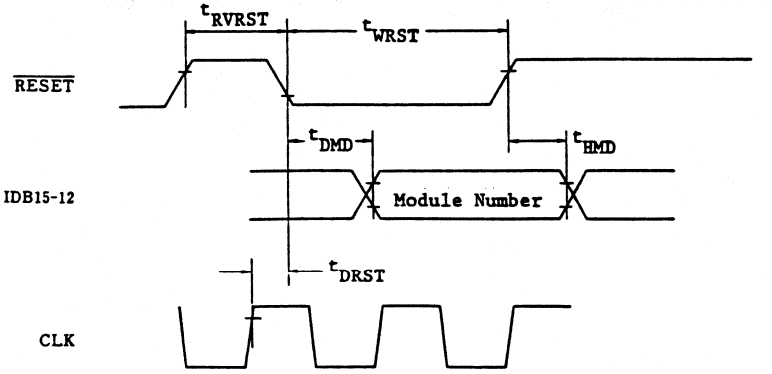
AC Test Output Waveform



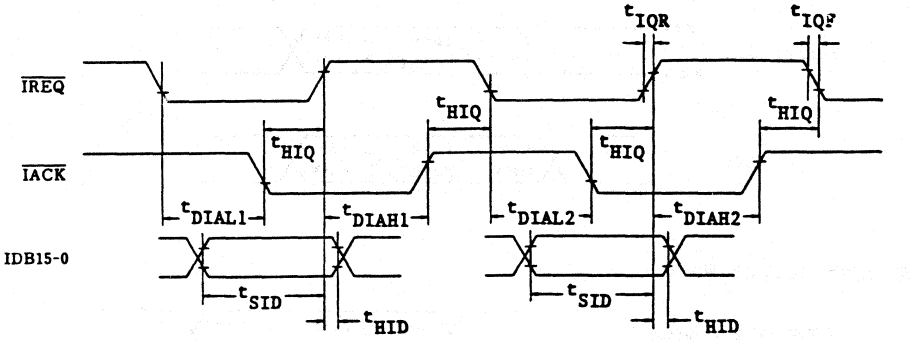
Clock Waveform



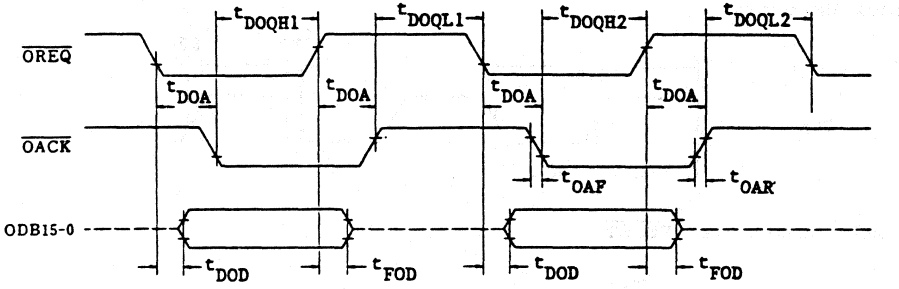
Module Number vs. $\overline{\text{RESET}}$



Input Handshake Waveform



Output Handshake Waveform



B

μPD9305
MEMORY ACCESS AND GENERAL BUS INTERFACE CHIP (MAGIC) / SUPPORT CHIP FOR μPD7281

PRODUCT DESCRIPTION

Introduction

The uPD9305, Memory Access and General bus-Interface Chip (MAGIC), is an LSI peripheral device for the uPD7281, Image Pipelined Processor (ImPP). Among the functions supported by the uPD9305 are: interfacing between multiple uPD7281s and a host computer, interfacing between uPD7281s and a system (image) memory, and DMA operations. Utilizing the uPD9305 permits the design of compact and efficient uPD7281 systems.

This manual describes the functions and operation of the uPD9305 only. For details on the uPD7281, please refer to "uPD7281 User's Manual". It is assumed throughout this manual that the reader is familiar with the operation of the uPD7281.

Chapter 1

OVERVIEW

1.1 General

The uPD9305, Memory Access and General Bus-Interface Chip (MAGIC), is a peripheral LSI device for use with the uPD7281, Image Pipelined Processor (ImPP). Because addresses are not used by the uPD7281, the memories used in a uPD7281 system are treated as processing modules to which module numbers have been assigned. To perform memory access, each uPD7281 must output memory access tokens containing the necessary address, data and control signals that it recognizes (by matching module numbers) as its own, perform the necessary processing, and output the result of the access as another memory access token.

External circuitry is therefore required between the uPD7281 and the memory to process the memory access tokens output by the uPD7281(s) and to organize the data output from the memory into tokens.

Circuitry is also required between the host processor and the uPD7281(s) to organize the data from the host into tokens and to return the data output from the uPD7281(s) into the form required by the host or to re-input the tokens to the uPD7281(s).

The uPD9305 incorporates these functions in a single integrated circuit, greatly simplifying the design of a uPD7281 system.

1.2 Features

- Reduces the number of external circuits
  The uPD9305 contains all of the basic circuits required to implement a uPD7281 system. This circuitry normally would comprise about 120 MSI/SSI circuits. Integrating these circuits into a single chip reduces the number of system components as well as the size of the mounting surface resulting in a simplified printed circuit board design and improved reliability.

- Simplified host intefacing
  Because the uPD7281 moves data in the form of tokens, conversion of data from conventional format to token format and from token format to conventional format is required whenever data is transferred between the host system and the uPD7281. Adoption of the uPD9305 not only simplifies interfacing between the host processor and the uPD7281, it also permits the use of either an 8-bit or 16-bit CPU for the host processor.

- High performance memory interface
  Because addresses are not used by the uPD7281, memories are treated as a collection of modules to which the appropriate address/data/control information is sent in the form of tokens.

Data from the memory is converted to token format before it is returned to the uPD7281.
The uPD9305 contains powerful interface functions that support, in addition to basic interface operations, uPD7281 object program load, read/modify/write of the image memory, multiple uPD7281 image memory accesses, etc.

- Internal DMA request generate
- CMOS technology
- 132-pin Pin Grid Array (PGA) package
- Single +5 volt power supply

1.3 Pin Configuration

Figure 1-1
Pin Configuration

(Bottom View)

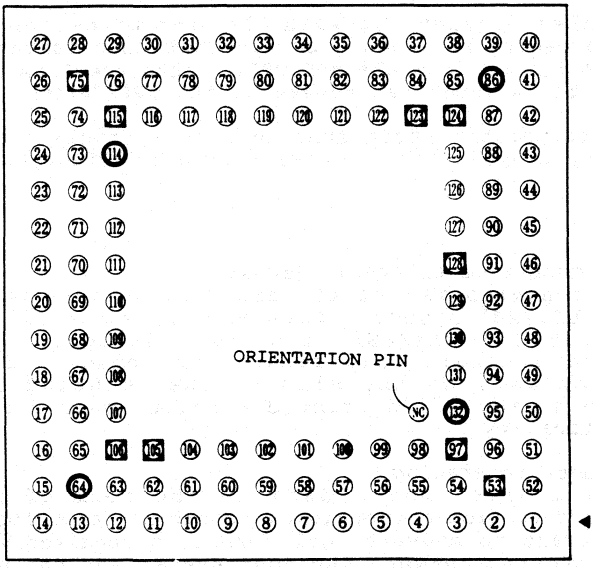
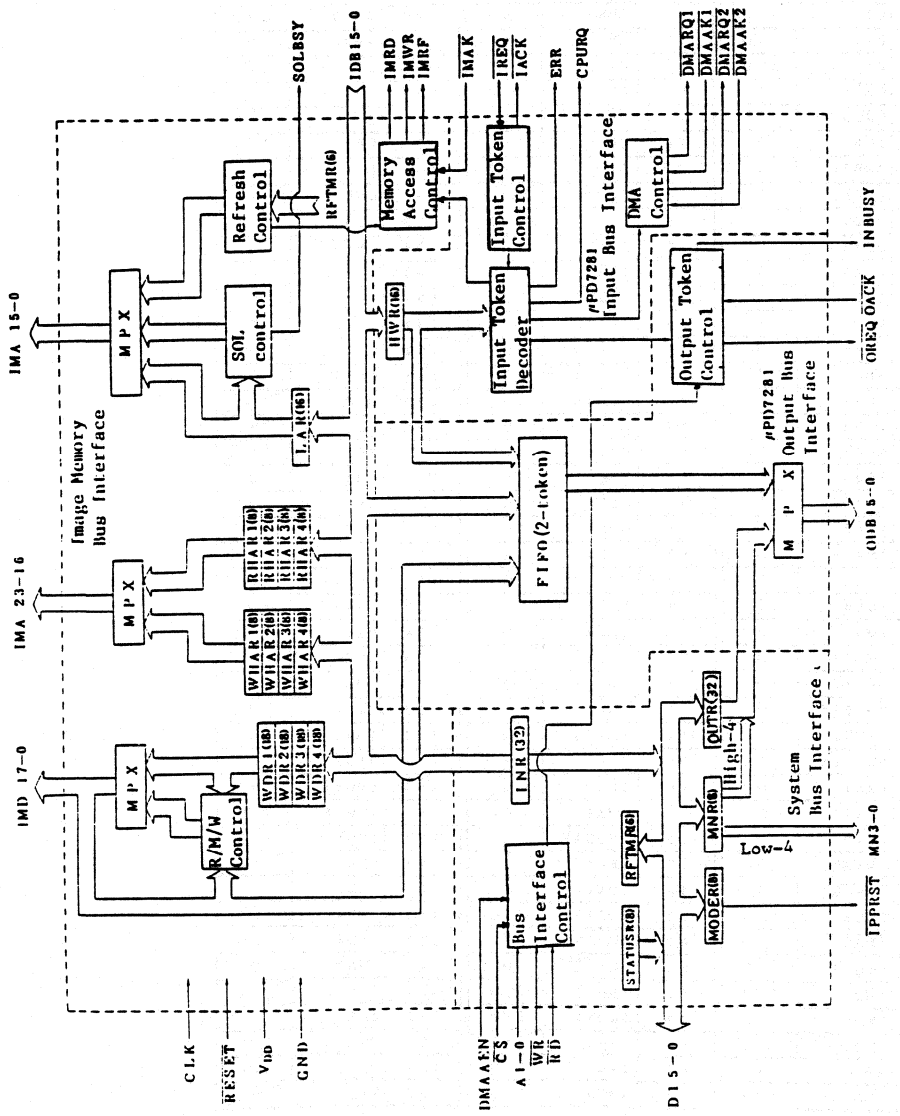


TABLE 1-1
PIN IDENTIFICATION

PIN #	SIGNAL	I/O	PIN #	SIGNAL	I/O	PIN #	SIGNAL	I/O
1	CLK	I	45	IMD1	I/O	89	A1	I
2	D10	I/O	46	IMWR	O	90	IMD0	I/O
3	D12	I/O	47	$\overline{WR}$	I	91	IMRF	O
4	D15	I/O	48	D2	I/O	92	D0	I/O
5	$\overline{OACK}$	I	49	D5	I/O	93	$\overline{RD}$	I
6	$\overline{OREQ}$	O	50	$\overline{CS}$	I	94	D4	I/O
7	IDB14	I	51	D8	I/O	95	D6	I/O
8	ODB14	O	52	D9	I/O	96	D7	I/O
9	IDB11	I	53	GND		97	GND	
10	ODB11	O	54	D11	I/O	98	D13	I/O
11	ODB8	O	55	D14	I/O	99	$\overline{IPPRST}$	O
12	IDB9	I	56	$\overline{IREQ}$	I	100	IDB15	I
13	ODB5	O	57	$\overline{IACK}$	O	101	ODB15	O
14	IDB8	I	58	IDB13	I	102	IDB12	I
15	ODB4	O	59	ODB13	O	103	ODB12	O
16	IDB7	I	60	IDB10	I	104	ODB9	O
17	ODB2	O	61	ODB10	O	105	GND	
18	IDB6	I	62	ODB7	O	106	GND	
19	MN2	O	63	ODB6	O	107	ODB0	O
20	IDB4	I	64	V_{DD}		108	MN3	O
21	IMA22	O	65	ODB3	O	109	MN0	O
22	IDB2	I	66	ODB1	O	110	IDB3	I
23	IMA18	O	67	IDB5	I	111	IMA20	O
24	IMA15	O	68	MN1	O	112	IMA19	O
25	IDB0	I	69	IMA23	O	113	IMA16	O
26	IMA12	O	70	IMA21	O	114	V_{DD}	
27	IMA11	O	71	IDB1	I	115	GND	
28	IMA10	O	72	IMA17	O	116	IMA7	O
29	SOLBSY	O	73	IMA14	O	117	IMA6	O
30	CPURQ	O	74	IMA13	O	118	IMA3	O
31	DMAAEN	I	75	GND		119	$\overline{RESET}$	I
32	IMA5	O	76	IMA9	O	120	IMD16	I/O
33	IMA2	O	77	IMA8	O	121	IMD15	I/O
34	IMA0	O	78	INBUSY	O	122	IMD14	I/O
35	$\overline{DMAAK1}$	I	79	IMA4	O	123	GND	
36	$\overline{DMARQ1}$	O	80	IMA1	O	124	GND	
37	IMD13	I/O	81	IMD17	I/O	125	IMD4	I/O
38	$\overline{IMAK}$	I	82	$\overline{DMAAK2}$	I	126	IMD2	I/O
39	IMD10	I/O	83	$\overline{DMARQ2}$	O	127	IMRD	O
40	IMD9	I/O	84	IMD12	I/O	128	GND	
41	IMD8	I/O	85	IMD11	I/O	129	ERR	O
42	IMD7	I/O	86	V_{DD}		130	D1	I/O
43	A0	I	87	IMD6	I/O	131	D3	I/O
44	IMD3	I/O	88	IMD5	I/O	132	V_{DD}	

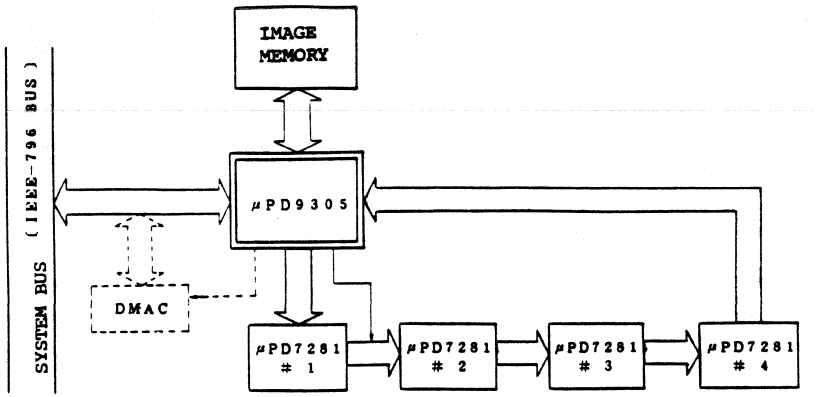
1.4 uPD9305 Block Diagram

Figure 1-2
uPD9305 Block Diagram



1.5 System Configuration Example

Figure 1-3
System Configuration Example



- The uPD9305 supports up to eight uPD7281s. As described later, however, when more that five uPD7281s are connected, care must be exercised to control the number of tokens output to the image memory.
- The image memory can have a maximum address width of 24 bits (16,777,216 addresses) and a maximum data width of 18 bits.
- A Direct Memory Access Controller (DMAC) can be incorporated into the system if desired.

Chapter 2

Pin Functions

The uPD9305 has 132 pins. This chapter describes the functions of these pins.

Table 2-1
uPD9305 Pinout by Functional Groups

I/O	Signal	Pin No.
I	CLK	1
	$\overline{\text{RESET}}$	119
Status		
O	ERR	129
	SOLBSY	29
	CPURQ	30
	INBUSY	78
Host Interface		
I	$\overline{\text{WR}}$	47
	$\overline{\text{RD}}$	93
	$\overline{\text{CS}}$	50
	A0	43
	A1	89
I/O	D0	92
	D1	130
	D2	48
	D3	131
	D4	94
	D5	49
	D6	95
	D7	96
	D8	51
	D9	52
	D10	2
	D11	54
	D12	3
	D13	98
	D14	55
	D15	4
DMA		
O	$\overline{\text{DMARQ1}}$	36
	$\overline{\text{DMARQ2}}$	83
I	$\overline{\text{DMAAK1}}$	35
	$\overline{\text{DMAAK2}}$	82
	DMAAEN	31

I/O	Signal	Pin No.
uPD7281 Interface		
O	MN0	109
	MN1	68
	MN2	19
	MN3	108
O	$\overline{\text{OREQ}}$	6
I	$\overline{\text{OACK}}$	5
	$\overline{\text{IREQ}}$	56
O	$\overline{\text{IACK}}$	57
	$\overline{\text{IFFRST}}$	99
O	ODB0	107
	ODB1	66
	ODB2	17
	ODB3	65
	ODB4	15
	ODB5	13
	ODB6	63
	ODB7	62
	ODB8	11
	ODB9	104
	ODB10	61
	ODB11	10
	ODB12	103
	ODB13	59
	ODB14	8
	ODB15	101
I	IDB0	25
	IDB1	71
	IDB2	22
	IDB3	110
	IDB4	20
	IDB5	67
	IDB6	18
	IDB7	16
	IDB8	14
	IDB9	12
	IDB10	60
	IDB11	9
	IDB12	102
	IDB13	58
	IDB14	7
	IDB15	100

I/O	Signal	Pin No.
Image Memory Interface		
I	IMAK	38
O	IMRD	127
	IMWR	46
	IMRF	91
I/O	IMD0	90
	IMD1	45
	IMD2	126
	IMD3	44
	IMD4	125
	IMD5	88
	IMD6	87
	IMD7	42
	IMD8	41
	IMD9	40
	IMD10	39
	IMD11	85
	IMD12	84
	IMD13	37
	IMD14	122
	IMD15	121
	IMD16	120
	IMD17	81
O	IMA0	34
	IMA1	80
	IMA2	33
	IMA3	118
	IMA4	79
	IMA5	32
	IMA6	117
	IMA7	116
	IMA8	77
	IMA9	76
	IMA10	29
	IMA11	27
	IMA12	26
	IMA13	74
	IMA14	73
	IMA15	24
	IMA16	113
	IMA17	72
	IMA18	23
	IMA19	112
	IMA20	111
	IMA21	70
	IMA22	21
	IMA23	69

2.1 CLK (Clock) ... Input

The CLK pin is used to input the system clock which regulates all operations of the uPD9305. The clock frequency can be set independently of that used for the uPD7281(s) or for the image memory.

2.2 RESET/ (Reset) ... Input

The uPD9305 is initialized by applying a low level signal to the RESET/ pin. At this time, OREQ/, IACK/, the token input/output flip-flop, and the image memory access request signals are set to their inactive levels. A command reset also initializes all of these except for the image memory access signals. The image memory refresh address counter, refresh timer counter and mode registers are also cleared (set to 0).

To initialize the uPD9305, the RESET/ signal must be held low for the equivalent of at least four clock cycles (of either the uPD9305 or uPD7281, whichever clock is slower). In addition to the reset operation performed by this signal, there is also a command reset that uses the mode register (described later).

2.3 RD/ (Read) ... Input

When the RD/ signal is low while the CS/ signal is low, the contents of the register addressed by A_1 and A_0 are output on the Data Bus (D_{15} - D_0). (See Table 2-2)

Table 2-2
Reading uPD9305 Internal Registers

CS/	RD/	A_1	A_0	Internal Register
0	0	0	0	uPD7281 input register (MN=0)
0	0	0	1	Status register
0	0	1	0	*
0	0	1	1	Use prohibited

* : This register has the same function as a command reset, however, the data read has no meaning.

2.4 WR/ (Write) ... Input

When the WR/ signal is low while the CS/ signal is low, the data on D_{15} - D_0 is written to the register addressed by A_1 and A_0. (See Table 2-3)

Table 2-3
Writing uPD9305 Internal Registers

CS/	WR/	A_1	A_0	Internal Register
0	0	0	0	uPD7281 output register (MN=0)
0	0	0	1	Mode register
0	0	1	0	Module Number register
0	0	1	1	Refresh cycle register

2.5 A_1, A_0 (Address) ... Input

The A_1 and A_0 signals are used to select the uPD9305 internal register to be read or written. The registers and the combinations of high and low signals on A_1 and A_0 that select each register are given in Tables 2-2 and 2-3.

2.6 D_{15} - D_0 (Data Bus) ... Input/Output

When the internal registers of the uPD9305 are read, the contents of the registers are output on D_{15} - D_0. When the uPD9305 registers are written, the data on D_{15} - D_0 is set in the registers.

2.7 CS/ (Chip Select) ... Input

The RD/ and WR/ signals are enabled when CS/ is low.

2.8 OREQ/ (Output Request) ... Output

The OREQ/ is connected to the IREQ/ pin of the first uPD7281. Together with the IACK/ signal output by the uPD7281, the OREQ/ signal performs the handshaking that controls token output to the uPD7281. If the uPD9305 has a token to be output to the uPD7281 (from the host, pass data from another uPD7281, or data form the image memory), this signal goes low to inform the uPD7281. At the same time, the uPD9305 puts half (16 bits) of the token on the Output Data Bus (ODB). The ODB is connected to the Input Data Bus (IDB) of the uPD7281. When the uPD9305 receives the token half and is able to input the data, it acknowledges the transfer by bringing the IACK/ pin to a low level. The process then repeats for the remaining half token.

2.9 OACK/ (Output Acknowledge) ... Input

OACK/ is connected to the IACK/ pin of the first uPD7281. The OACK/ pin receives from the first uPD7281 the input acknowedge (IACK/) signifying the the uPD7281 has input the token half sent by the uPD9305.

2.10 ODB_0 - ODB_{15} (Output Data Bus) ... Output

The Output Data Bus (ODB) connects to the IDB of the first uPD7281. Data is output to the uPD7281 from the uPD9305 via the ODB while the OREQ/ pin of the uPD9305 is low.

2.11 IREQ/ (Input Request) ... Input

The IREQ/ pin is connected to the OREQ/ of the last uPD7281. When the uPD7281 has a token to output, it sets its OREQ/ to a low level and outputs the upper half token on its ODB (connected to the IDB of the uPD9305). When the uPD9305 receives this signal and is able to input the half token, it acknowledges by bringing its IACK/ signal to a low level. The process repeats for the lower half token sent from the uPD7281 to the uPD9305.

2.12 IACK/ (Input Acknowledge) ... Output

The IACK/ pin is connected to the OACK/ pin of the last uPD7281. When the IREQ/ input goes low, if the uPD9305 has completed preparations to input the token half, it sets the IACK/ signal to a low level. Together with the IREQ/ signal, this signal performs the handshaking that controls the output of tokens from the uPD7281.

2.13 IDB_{15} - IDB_0 (Input Data Bus) ... Input

The Input Data Bus (IDB) connects to the ODB of the last uPD7281. The IDB is used to receive tokens from the uPD7281(s).

2.14 MN_3 - MN_0 (Module Number) ... Tristate output

The Module Number (MN) pins are used to specify the MN of a uPD7281. The uPD9305 has an 8-bit MN register that can store the module numbers for two uPD7281s. When a system reset (using RESET/) or a command reset is applied, the uPD7281 MN stored in the upper 4 bits of the MN register is output on the upper 4 bits of the ODB while the lower 4 bits are output on the MN_3 - MN_0 pins. The MN pins are at a high impedance level at all other times.

2.15 IPPRST/ (Image Pipelined Processor Reset) ... Output

The IPPRST/ connects to the RESET/ pin of the uPD7281(s). When the IPPRST/ signal is low, the uPD7281s are reset. IPPRST/ is brought low by a system reset or a command reset.

2.16 IMA_{23} - IMA_0 (Image Memory Address) ... Output

The Image Memory Address (IMA) pins supply addresses for the image memory. The higher 8 bits of the 24-bit IMA are read from the internal register file. There are two types of addresses generated by the uPD9305: one for normal read/write operations, the other for dynamic memory refresh operation.

2.17 IMD_{17} - IMD_0 (Image Memory Data) ... Input/Output

The Image Memory Data (IMD) pins are used as a bidirectional data bus for transferring data between the uPD9305 and the image memory.

2.18 IMRD (Image Memory Read) ... Output

When the uPD9305 receives an image memory read request token from a uPD7281, the IMRD signal is brought high once the address on the IMA bus becomes stable. After the contents of the image memory location have been read, the image memory responds by applying a low signal to the IMAK/ pin of the uPD9305. In this way, image memory read operations are controlled by the IMRD and IMAK/ signals.

2.19 IMWR (Image Memory Write) ... Output

When the uPD9305 receives an image memory write request token from a uPD7281, the IMWR signal is brought high once the data output on the IMD bus and address on the IMA bus have become stable. After the image memory has completed the write operation, it returns a low signal on IMAK/. In this way, write operations are controlled by the IMWR and IMAK/ signals.

2.20 IMRF (Image Memory Refresh) ... Output

The uPD9305 can be used to supply refresh addresses and refresh timing to the image memory. The IMRF signal is brought high (active) for the fixed refresh cycle determined by the contents of the refresh cycle register. The refresh address is output on the lower 10 bits of the IMA bus. When the image memory has received the refresh address and has completed the refresh operation, it returns by placing a low level on IMAK/. In this way, image memory refresh operations are controlled by the IMRF and IMAK/ signals.

2.21 IMAK/ (Image Memory Acknowledge) ... Input

The IMAK/ pin is used to indicate to the uPD9305 that the image memory operation (read, write, or refresh) has been completed. Together with IMRD, IMWR, or IMRF, IMAK/ forms the handshaking that controls the various operations on the image memory.

2.22 DMAAEN (DMA Address Enable) ... Input

The DMAAEN signal is used to indicate to the uPD9305 that an external DMA controller is putting DMA addresses on the address bus. During DMA operation, DMA addresses (system memory addresses) are input to the A1 and A0 pins of the uPD9305, but these have no meaning for the uPD9305 and could possibly alter register contents. Therefore, the uPD9305 operates as if A1 and A0 are both 0 during DMAAEN active (high level).

2.23 DMARQ1/ (DMA Request 1) ... Output

When a DMA1 token is input to the uPD9305 from a uPD7281, the uPD9305 brings DMARQ1/ to a low level to indicate a DMA request to the externally connected DMA controller. Note, however, that the direction of this transfer is limited to that from the host to the uPD9305 so the DMA controller must be set accordingly.

2.24 DMARQ2/ (DMA Request 2) ... Output

The DMARQ2/ signal is brought low by a DMA2 request token from the uPD7281 to the uPD9305. The DMARQ2/ pin indicates to an external DMA controller that the uPD9305 is requesting to transfer data from the uPD9305 to the host system memory by the DMA controller.

2.25 DMAAK1/ (DMA Acknowledge 1) ... Input

The DMAAK1/ is input from the DMA controller to indicate that the DMA controller has received a DMARQ1/ signal.

2.26 DMAAK2/ (DMA Acknowledge 2) ... Input

The DMAAK2/ is input from the DMA controler to indicate that the DMA controller has received a DMARQ2/ signal.

2.27 CPURQ (CPU Request) ... Output

When a uPD7281 outputs a CPU token (MN=0), the CPURQ signal is brought high to inform the host CPU that a CPU token exists. The procedure is therefore for the CPU to monitor either this signal or the internal status register and to perform a read operation

after confirming that a CPU token exists.

2.28 INBUSY (Input Busy) ... Output

A high level on the INBUSY pin indicates that the uPD9305 has input a token from the last uPD7281 and remains high until the uPD9305 outputs that token to the first uPD7281. The host CPU monitors either this signal or the internal status register to determine when to output a token to the uPD9305.

2.29 SOLBSY (Self Object Load Busy) ... Output

The SOLBSY pin is brought high when a SOL1 or SOL2 token is output by the uPD7281 and is used to inform the host system that self object load is being performed.

2.30 ERR (Error) ... Output

The uPD9305 has three different kinds of error status. The first is when an error token is output from the uPD7281; the second when the host reads an output token when CPURQ=0; and the third when the host outputs a token to the uPD9305 when INBUSY=1. The ERR signal becomes high when any one of these errors occur and returns to a low level when command reset or system reset is applied.

2.31 V_{DD} (Power)

Single +5 volt power supply.

2.32 GND (Ground)

Power supply ground.

Chapter 3

INTERNAL BLOCK FUNCTIONS

The uPD9305 is divided into four interface sections, each of which is connected to a different bus:

- System bus interface
- Image memory bus interface
- uPD7281 input bus interface
- uPD72831 output bus interface

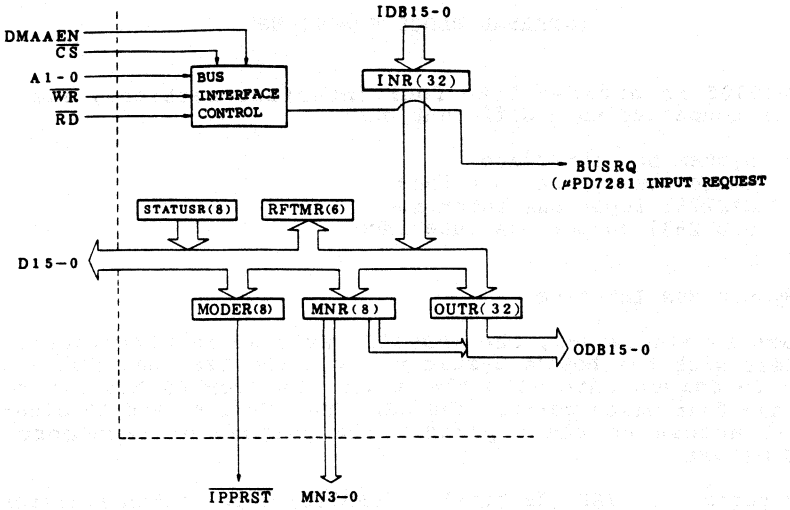
3.1 System Bus Inteface

As shown in Figure 3-1, the uPD9305 performs input/output data transfers with the host's system bus via the Data Bus (D15 - D0) pins. To communicate with the host, the uPD9305 has two read ports and four write ports. The CS/, RD/, WR/, A1 and A0 signals control access to these ports. The purpose of each port is listed below:

Read ports	:	INR (32 bits)	uPD7281 input token register
		STATUSR (8 bits)	Internal status flag register
Write ports	:	OUTR (32 bits)	uPD7281 output token register
		MODER (8 bits)	uPD9305 mode setting regsiter
		MNR (8 bits)	uPD7281 MN setting register
		RFTMR (16 bits)	Refresh timer setting register

The output from OUTR is connected to ODB15-ODB0 and the input to INR is connected to IDB15-IDB0.

Figure 3-1
System Bus Interface

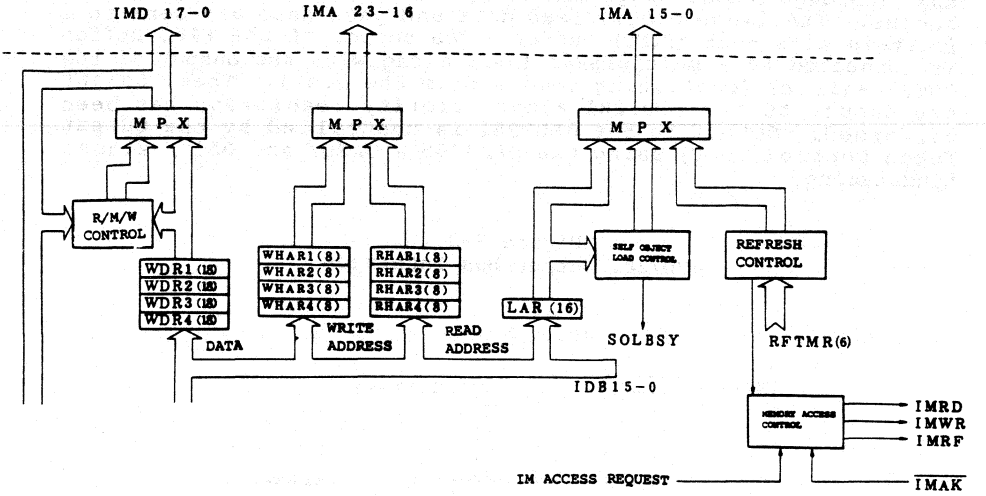


3.2 Image Memory Interface

The uPD9305 accesses the external image memory via a 24-bit address bus (IMA23-IMA0) and an 18-bit data bus (IMD17-IMD0). Handshaking is performed by the request signals (IMRD, IMWR and IMRF for read, write, and refresh, respectively) together with the response signal IMAK/. Address bus IMA is divided into the high-order 8 bits and the low-order 16 bits. Four 8-bit address registers are provided for read (RHAR4-RHAR1) and four for write (WHAR4-WHAR1). There are also four 18-bit Write Data Registers (WDR4-WDR1).

In the uPD9305, these groups of registers are referred to as register files. Use of register files is required because of the 16-bit configuration of the uPD7281 input and output buses. In other words, the information required by the uPD7281 to access the image memory must be divided. A problem would occur when a number of uPD7281s are accessing the memory at the same time. In this case, there is a danger that the address information and the corresponding memory write data will become separated. The register files (described in detail later) are provided to prevent this.

Figure 3-2
Image Memory Interface



In addition to basic memory access, the uPD9305 is also capable of three intelligent memory access funtions: read/modify/write (RMW), self object load (SOL) and refresh (RF).

To perform RMW, first the memory is read, then a logical operation (AND, OR, XOR) is performed between the data read from the memory and the contents of the selected Write Data Register (WDR). The result is then rewritten to the image memory address. In this way, read/modify/write can be performed at high speeds without involving the CPU or having the data pass through the uPD7281s.

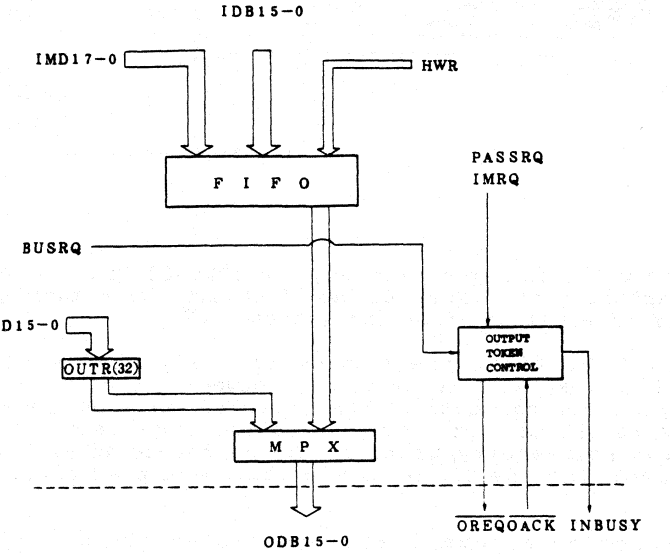
The SOL function automatically generates the appropriate addresses to read the object program stored in the image memory. This data is then sent to the uPD9305 and output to the uPD7281. SOL serves to reduce the overhead for program load operations.

In the RF operation, the uPD9305 generates the 10-bit address and the timing (programmable) to refresh the external image memory.

3.3 uPD7281 Output Bus Interface

Three kinds of tokens are output to the uPD7281: tokens from the host (through OUTR), image memory read data tokens, and pass data tokens. The image memory read data and pass data are sent to a first-in first-out (FIFO) buffer. The output of the FIFO buffer is connected to a multiplexer (MPX) along with the output of the OUTR register (containing tokens from the host). These tokens are output to the uPD7281 after priority resolution has been performed. Output to the uPD7281 is controlled by the Output Token Control (OTC) using the uPD9305's OREQ/ and OACK/ signal handshaking.

Figure 3-3
uPD7281 Output Bus Interface

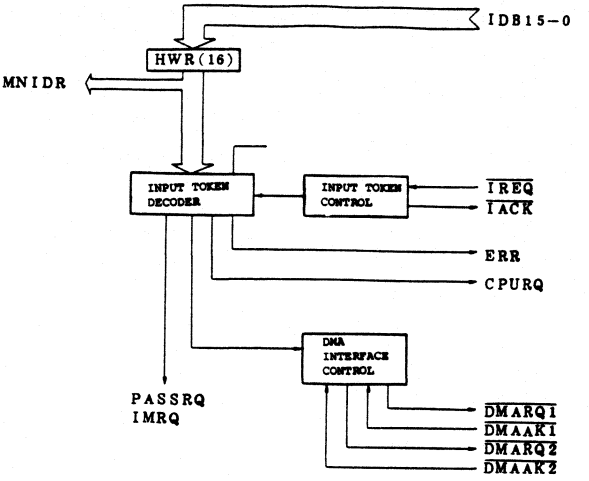


3.4 uPD7281 Input Bus Interface

The uPD9305 functions to perform such processes as check/decode of the MN and Identifier (ID) of the tokens input from the uPD7281. The higher-order 16 bits of the token input from the uPD7281 are latched in the 16-bit High Word Register (HWR) and then decoded (see Chapter 4, Host/uPD7281 Interfacing). The result is sent to the interface controls described in sections 3.1 to 3.3 where the necessary processing is performed.

The uPD9305 also has a DMA request function whereby the arrival of a DMA request token (see Chapter 6, DMA Functions) causes the output of a DMARQ1/ or DMARQ2/ signal.

Figure 3-4
uPD7281 Input Bus Interface



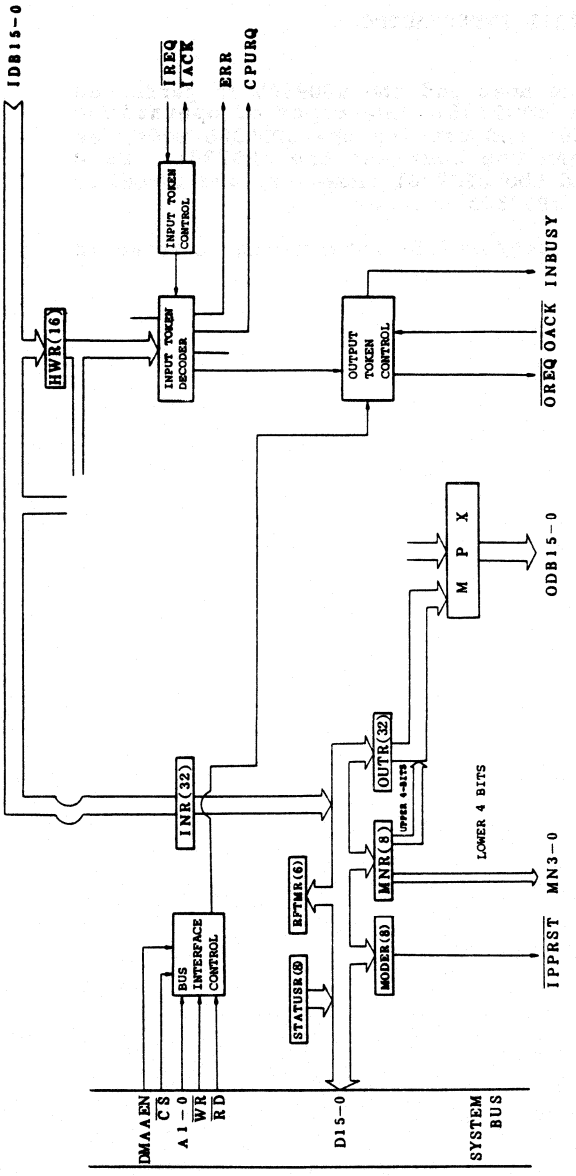
Chapter 4

HOST/uPD7281 INTERFACING

Data communication between the host and the uPD9305 is performed as I/O read and write to the uPD9305. The types of operations include reading uPD9305 status and setting the uPD9305 mode, as well as data transfer between the host and the uPD7281. Data transfers between the host and the uPD7281, however, are normally performed after checking the uPD9305 status.

An example block diagram for host/uPD7281 interfacing is shown in Figure 4-1.

Figure 4-1
Host/uPD7281 Interface Block Diagram



4.1 Input/Output Ports

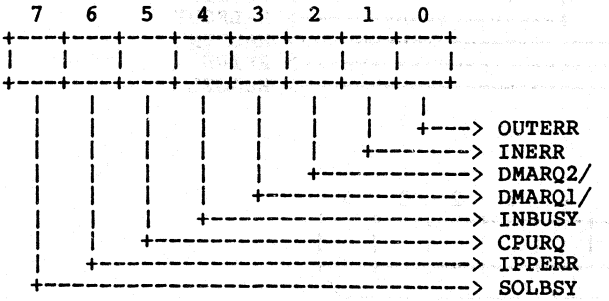
The uPD9305 internal ports include: status register, mode register, uPD7281 I/O data registers, etc. Access to these registers is controlled by the CS/, RD/, WR/, A1 and A0 signals. Data is transferred between the uPD9305 and the host over the Data Bus (D15-D0).

- Read

CS/	RD/	WR/	A1 A0	Internal Register
0	0	1	0 0	Data input from uPD7281 (INR)
0	0	1	0 1	Status Register
0	0	1	1 0	*
0	0	1	1 1	(Not used)

* : This has the same function as command reset, however the data read has no meaning.

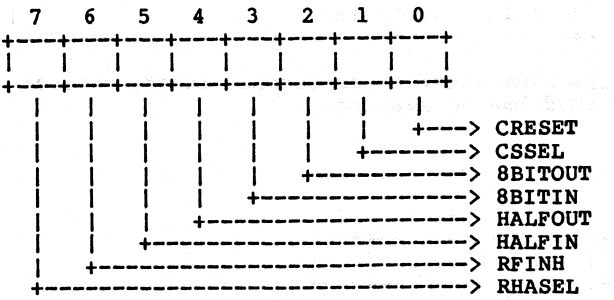
Status Register



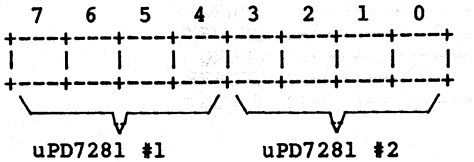
- Write

CS/	RD/	WR/	A1 A0	Internal Register	At RESET
0	1	0	0 0	uPD7281 Output Data Register (OUTR)	No change
0	1	0	0 1	Mode Register	00H
0	1	0	1 0	MN Register	No change
0	1	0	1 1	Refresh Timing Register	12H

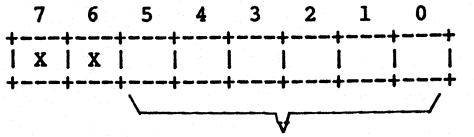
Mode Register



MN Register



Refresh Timing Register



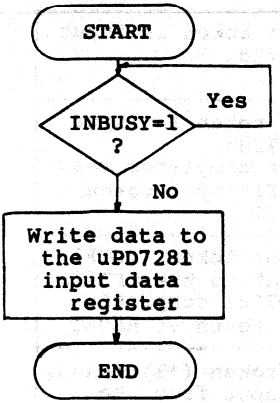
X = don't care

(The actual refresh cycle is (RC+1) x 8 CLK, RC = 0 to 63)

4.1.1 Status

Data transfers between the host and the uPD7281(s) should be performed after first checking the status. This is done by reading the contents of the status register. The status data can be output to the Data Bus of the uPD9305.

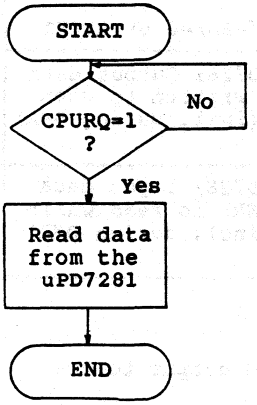
Host --> uPD7281



INBUSY : Bit 4 of status register

Note: If data is written to the register even though INBUSY=1, the data input to the uPD7281 is not assured and INERR will be set to 1.

uPD7281 --> Host



CPURQ : Bit 5 of status register

Note: Data read when CPURQ=0 is meaningless and will cause OUTERR to be set to 1.

Table 4-1
Status Register

Signal Name	Bit	Active	Set/Reset Conitions
SOLBSY	7	H	↑ When a SOL1 or SOL2 token(*4) is input from the uPD7281 ↓ When SOL terminates or RESET
IPPERR(*1,2)	6	H	↑ When an error token is input from the uPD7281 ↓ RESET
CPURQ (*2)	5	H	↑ When an MN=0 token is input from the uPD7281 ↓ When the host completes read of the uPD7281 input token (INR) or RESET
INBUSY	4	H	↑ When an output token (OUTR) is being input to the uPD7281 ↓ When the uPD7281 completes input of the token or RESET
DMARQ1/	3	L	↑ When a DMA1 token (*3) (MN=5, ID=6xH) is input from the uPD7281 ↓ When DMAAK1/+WR/=0 or RESET
DMARQ2	2	L	↑ When a DMA2 token (*3) (MN=5, ID=7xH) is input from the uPD7281 ↓ When DMAAK2/+RD/=0 or RESET
INERR (*1)	1	H	↑ When the uPD7281 Output data register is written to when INBUSY = 1 (incl. during DMA) ↓ RESET
OUTERR (*1)	0	H	↑ When the uPD7281 input data register (INR) is read while CPURQ = 0 (incl. during DMA) ↓ RESET

↑ : High (H) ↓ : Low (L)

*1 : IPPERR, INERR and OUTERR are ORed and output to the ERR pin.
*2 : When IPPERR becomes '1', CPURQ also becomes '1' at the same time.
*3 : Refer to Chapter 6, DMA Request Function
*4 : Refer to Chapter 5, Image Memory/uPD7281 Interfacing

4.2 Function Controls

The function blocks that monitor the mode and status of the uPD9305 and provide interfacing between the I/O controllers and the host computer are described in this section.

4.2.1 Mode Register

The Mode Register is the register that defines the operation of the uPD9305. The operation control performed by each bit of the Mode Register is summarized in Table 4-2.

- RHASEL

There are two kinds of tokens output by the uPD7281 that set the Read High Address (RHA) registers of the image memory interface. These are:

a.) Tokens with MN = 1 to 4 and ID = 70H to 7FH, and
b.) Tokens with MN = 5 and ID = 30H to 3FH.

The RHASEL mode selects whether or not the former type of token is valid. That is, when RHASEL = 1, the data of a token with MN = 1 to 4 and ID = 70H to 7FH will not be used to set the RHA but will function instead as an image memory Read Low Address command. After the image memory has been accessed, the MN of the read data token is set to '7'.

- RFINH

The RFINH mode selects whether or not to output a refresh control signal (IMRF) to the image memory. If the image memory provides its own refresh timing, RFINH should be set to '1'.

- CSSEL

Each word of the image memory is configured for 18 bits. This includes the 16-bit data and the C and S bits. When CSSEL = 0, the C and S bits are masked to '00'. Note, however, that data write is always performed as 18 bits words regardless of the setting of CSSEL.

- CRESET

Command reset can be applied to the system by setting and resetting this bit in the following sequence:

'0' --> '1' --> '0'

The differences between command reset and reset performed by applying a low level signal to the RESET/ pin are summarized in Table 4-3.

Table 4-2
Mode Register

Bit	Mode	State	Mode
7	RHASEL	0*	Read High Address valid for MN = 1 to 4 tokens
		1	Read High Address invalid for MN = 1 to 4 tokens (read oper.)
6	RFINH	0*	IMRF output
		1	IMRF not output
5	HALFIN	0*	High/low 32-bit input
		1	Low 16 bits input (high 16 bits fixed)
4	HALFOUT	0*	High/low 32-bit output
		1	Low 16 bits output (high 16 bits ignored)
3	8BITIN	0*	16-bit external data bus
		1	8-bit external data bus
2	8BITOUT	0*	16-bit external data bus
		1	8-bit external data bus
1	CSSEL	0*	Image memory read data C,S bits are '00'
		1	Image memory read data C,S bits are 'CS'
0	CRESET	0*	No operation
		1	Command reset

* : Default (value set when external reset is applied).

Table 4-3
Command and External Reset Differences

Item	RESET/ applied	Command Reset
- I/O data counter - Tokens in the uPD9305 - Image memory access requests (except refresh) - OREQ/, IACK/ - DMA Request	Cleared	Cleared
- Refresh timer - Refresh request - Refresh address - Mode register	Default values	No change
- IPPRST/ pin	0 (active)	0 (active)

- 8BITIN, HALFIN
Sets the write method for input data.

Host --> uPD7281 : Writing data

mode		Input data write sequence		Use Example
8BITIN	HALFIN	16-bit HH | HL	16-bit LH | LL	
0	0	①	②	16bit C P U normal use
0	1	Fixed data *	①	Scanner →uPD7281 →Image Memory
1	0	② | ①	④ | ③	8bit C P U normal use
1	1	Fixed data *	③ | ①	8-bit C P U memory →uPD7281 →Image Memory

The circled numbers in the above figure indicate the order in which the data are written.

* "Fixed data" should be defined beforehand by the user.

- 8BITOUT, HALFOUT
Sets the read method for output data.

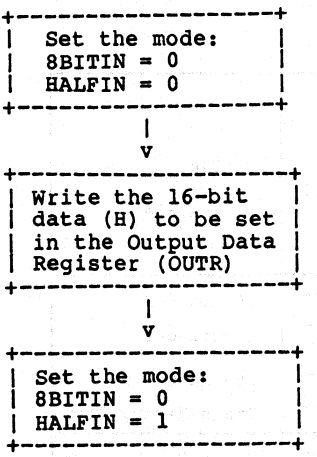
Host <-- uPD7281 : Reading data

mode		Output data read-out sequence		
8BITOUT	HALFOUT	16-bit (HH | HL)	16-bit (LH | LL)	Use / Example
0	0	①	②	16-bit C P U normal use
0	1		①	16-bit C P U Image Memory Read
1	0	② | ①	④ | ③	8-bit C P U normal use
1	1		② | ①	8-bit C P U Image Memory Read

The circled numbers in the above figure indicate the order in which the data are read.

Setting the fixed (16-bit) data

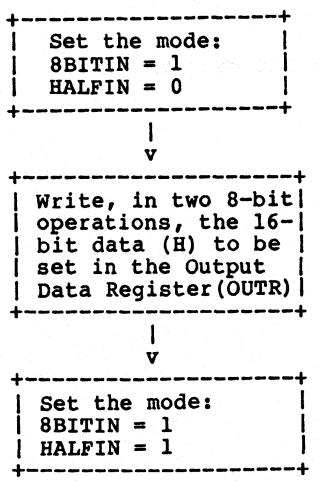
For 16-bit CPUs



Note: Low data is not written

Thereafter, each time the 16-bit Low data is written to the Output Data Register, the 16-bit High data set by the above procedure will be input with it to the uPD7281.

For 8-bit CPUs



Note: Low data is not written

Thereafter, each time the 16-bit Low data is written to the Output Data Register (in two write operations as LL (8 bits) and LH (8 bits), the 16-bit High data set by the above procedure will be input with it to the uPD7281.

4.3 Host/uPD7281 I/O Procedures

The uPD7281 performs all data input/output in 32-bit units (one token). However, because the Input Data Bus (IDB) and the Output Data Bus (ODB) of the uPD7281 are only 16 bits wide, this operation must be performed by dividing the token into two 16-bit halves, a high and a low word.

The I/O procedures shown below are examples of interfacing with an 8-bit CPU.

Figure 4-2
Input Timing (Host to uPD9305 to uPD7281)

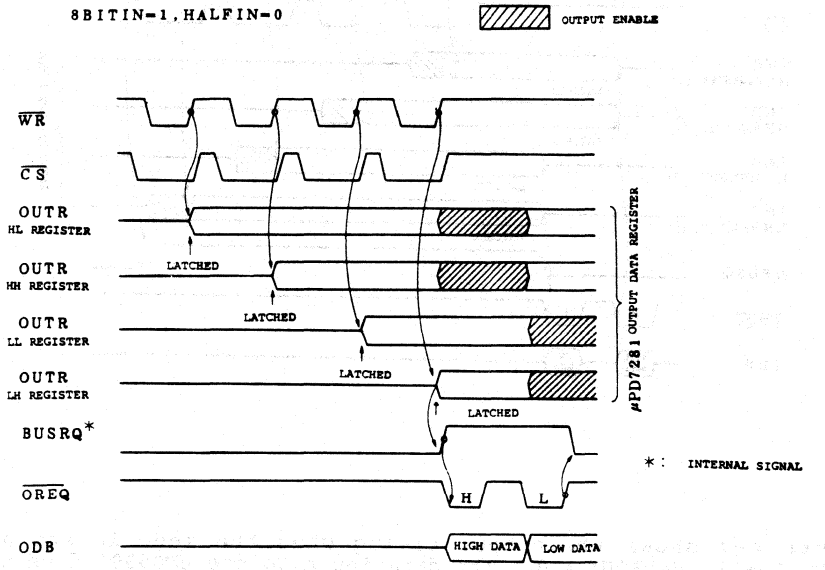


Figure 4-2 shows the input timing when the mode is set as 8BITIN = 1, HALFIN = 0. Once the data has been input to the uPD9305 (by four WR/ inputs), the uPD9305 internal BUSRQ signal is set to the active level (high) and data transfer to the uPD7281 is performed. The address for all four writes to the uPD9305 is A1,A0 = '00'. The only difference between this and other input modes is the timing with which data is latched in each register; in all cases, the BUSRQ signal is set to the active level when the LH register is latched, causing OREQ/ to be issued to the uPD7281's IREQ/ pin.

Figure 4-3
Output Timing (uPD7281 to uPD9305 to Host)

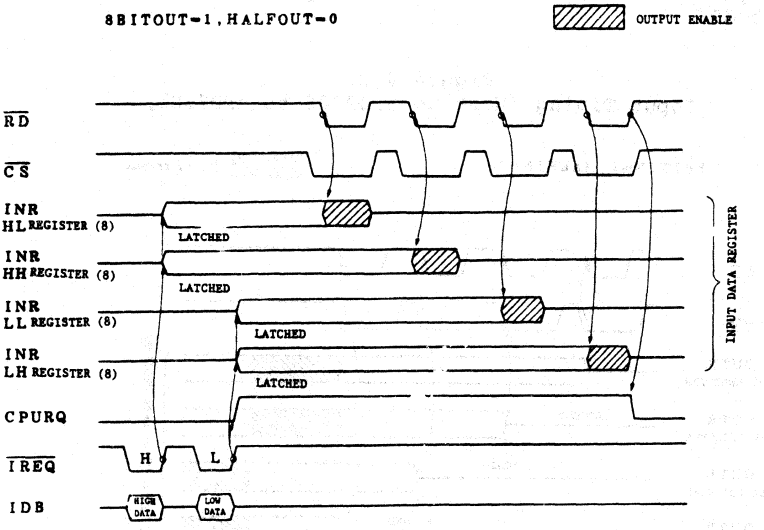


Figure 4-3 shows the output timing when the mode is set as: 8BITOUT = 1, HALFOUT = 0. The data input to the uPD9305 from the last uPD7281 is output to the host computer by four RD/ inputs. The address for all four reads of the uPD9305 by the host is A1,A0 = '00'. The only difference between this and other output modes is the output enable timing for each register; in all cases, CPURQ is set to the inactive level when output of the contents of the LH register has been completed.

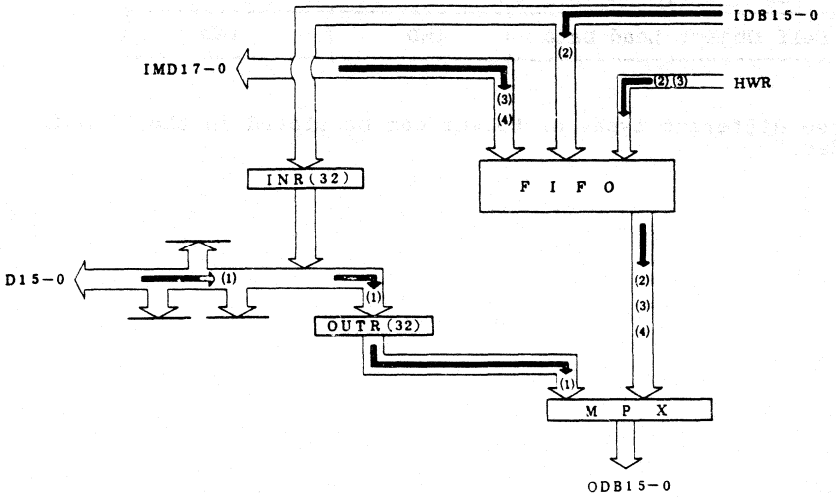
4.4 uPD7281 Output Control

In a uPD9305/uPD7281 system, the following four types of data can be output from the uPD9305 to the first uPD7281:

(1) Data sent from the host (including DMA)
(2) Data from the last uPD7281
(3) Image memory read data
(4) Self object load data

When any of the above types of data are present, the uPD9305 sets the OREQ/ signal to the active level (low) and outputs the data to the first uPD7281.

Figure 4-4
Output to uPD7281 Control Data Paths



- Priority Control

Of the four types of data, (1) through (4) described above, the priority of types (2) through (4) is controlled by the uPD7281 input controller to ensure that requests for two such data will not be simultaneously output. Because the lines for data from the host (1) are different from those for data types (2) through (4), the uPD7281 output controller must function to ensure that data from the host does not conflict with a request for another type of data.

The basic principle for priority setting by the output controller is that data types (2) through (4) have priority over data from the host.

Data actually being input from the host, however, has priority and any other data must wait until the input has been completed.

- FIFO

Data types (2) through (4) are stored in a two-level FIFO in the order of arrival. Three busses connect to the input of the FIFO and form the three types of tokens as follows:

	High side input	Low side input
Pass Data	HWR	IDB
Image Memory Read Data	HWR	IMD
Self Object Load Data	IMD	IMD

The three different types of tokens can be stored in the FIFO in any order.

4.5 uPD7281 Input Control

The data input from the last uPD7281 can be classified into one of the following five types, according to the values of the MN, ID and CTLF fields (see Figure 4-5 and Table 4-3):

(1) Output request data to the host (MN = 0)
(2) Image memory access data (MN = 1 to 6)
(3) DMA request data (MN = 5)
(4) Pass data (MN = 7 to 14)
(5) Delete data (MN = 15)

Figure 4-5
uPD7281 Input Control Data Flow

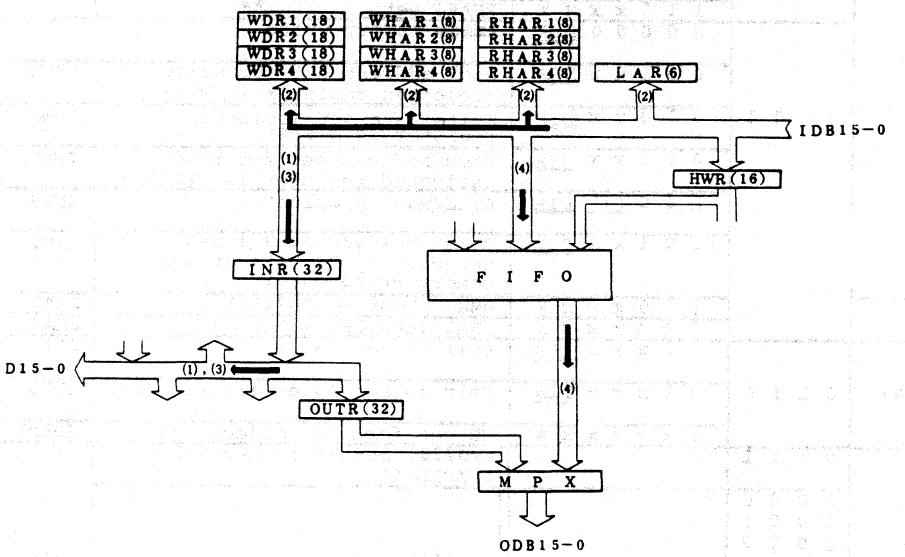


Table 4-3
MN Values and Token Types

	MN	ID	Function	Abbrev.
(1)	0 0 0 0	x x x x x x x	uPD7281 output data to host	CPU
(2)	0 0 0 1	MN' ID'*	Image memory read1 (RHAR1 select)	IMR
		1 1 1 x x x x	RHAR1 set **	
	0 0 1 0	MN' ID'*	Image memory read2 (RHAR2 select)	
		1 1 1 x x x x	RHAR2 set **	
	0 0 1 1	MN' ID'*	Image memory read3 (RHAR3 select)	
		1 1 1 x x x x	RHAR3 set **	
	0 1 0 0	MN' ID'*	Image memory read4 (RHAR4 select)	
		1 1 1 x x x x	RHAR4 set **	
	0 1 0 1	0 0 0 0 0 0 DIR	Image memory write	IMW
		0 0 1 x x DIR	High address set for write (selected register is DIR+1)	IMWHA
		0 1 0 x x DIR	Write data set (selected register is DIR+1)	IMWD
		0 1 1 x x DIR	High address set for read (selected register is DIR+1)	IMREA
		1 0 0 Mask DIR	Read/modify/write1	RMW1
		1 0 1 x x DIR	Read/modify/write2 (mask selected by CS bits of image memory write data)	RMW2
(3)		1 1 0 x x x x	DMA1 (host → uPD7281)	DMA1
		1 1 1 x x x x	DMA2 (uPD7281 → host)	DMA2
(2)	0 1 1 0	0 0 x x x DIR	Self object load1	SOL1
		0 1 x x x DIR	Self object load2 (rewrite MN)	SOL2
		1 x x x x x x	MN set for self object load	SOLMN
(4)	0 1 1 1		uPD7281 module number (when RHASEL=1)	PASS
	1 0 0 0 1 0 0 1 1 0 1 0 1 0 1 1 1 1 0 0 1 1 0 1 1 1 1 0		uPD7281 module numbers	
(5)	1 1 1 1		Deleted	VANISH

*: MN' is the MN used for the next circulation (MN'≠111).
ID' is the ID used for the next circulation.
**: When RHASEL bit in the mode register is="1"
these become image memory read token.

Module numbers MN = 1 through 6 are assigned to the image memory. The maximum number of uPD7281s that can be connected to the uPD9305 is therefore eight.

To control the amount of input data, the Input Controller on the uPD7821 uses the IACK/ signal. Similarly, to control the amount of input data to the uPD9305 from the uPD7281, the uPD9305 delays returning IACK/ to the uPD7281. Therefore, depending on the type of token (see Table 4-4) being processed by the uPD9305, the uPD9305 may cause tokens input from the uPD7281 to wait. It does this by delaying output of the IACK/ signal for the uPD7281 low half-token until the current processing has completed.

In Table 4-4, it is assumed that a token is input from the uPD7281 while the uPD9305 is processing another token. A '1' indicates that IACK/ is not returned for the low data of the token output by the uPD7281 and that data input to the uPD9305 is temporarily suspended.

Table 4-4
Input Wait to the uPD9305

uPD7281 Output Token \ Token Being Processed	CPU	IMR	IMW	RMW1	RMW2	DMA1	DMA2	SOL1	SOL2	PASS
CPU	1						1			
IMR		1	1	1	1	1		1	1	1
IMRHA		*		*	*			*	*	
IMW		1	1	1	1			1	1	
IMWHA			*	*	*					
IMWD			*	*	*					
RMW1		1	1	1	1			1	1	
RMW2		1	1	1	1			1	1	
DMA1		1				1		1	1	1
DMA2	1						1			
SOL1		1	1	1	1	1		1	1	1
SOL2		1	1	1	1	1		1	1	1
SOLMN									1	
PASS		1						1	1	**
VANISH										

* : "1" When the Registers used are the same

** : 2 Tokens may be input without a wait

4.6 Module Number (MN) Setting

The uPD9305 can set MNs for two uPD7281s. For three of more uPD7281s, an external buffer is required (See Figure 4-6). When RESET/ is applied, unique MN values must be assigned to each uPD7281. To do this, set the MN value of uPD7281 #1 in the higher four bits and that for uPD7281 #2 in the lower four bits of the 8-bit MN register. When the RESET/ input of the uPD9305 is set to a low level, causing IPPRST/ (connected to the uPD7281(s) RESET/ pin) to go low, the respective MN values are placed on the high-order 4 bits of the Input Data Bus (IDB) of the two uPD7281s (#1 and #2). This operation is performed only at reset. Note that the MN data for uPD7281 #1 is set on the high-oder 4 bits of the Output Data Bus (ODB) within the uPD9305. For this reason, no external pin connections are necessary for uPD7281 #1. The timing for setting MNs when RESET/ is applied is shown in Figure 4-7.

Figure 4-6
Module Number (MN) Setting

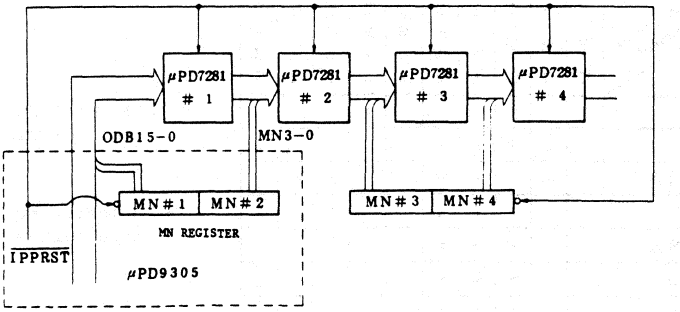
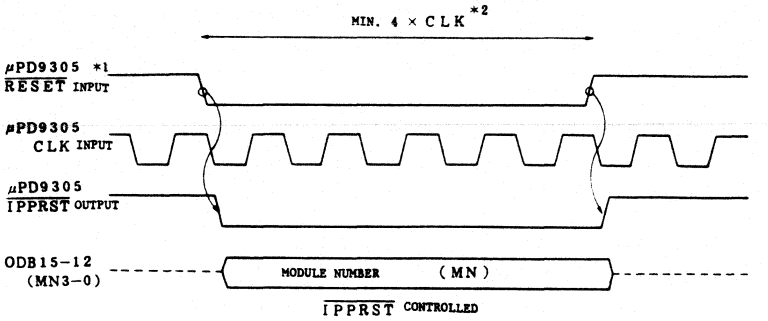


Figure 4-7
Module Number (MN) Setting Timing



*1 : IPPRST/ is also output by command reset to set the MNs

*2 : This CLK is either that of the uPD9305 or that of the uPD7281s, whichever is slower.

Chapter 5

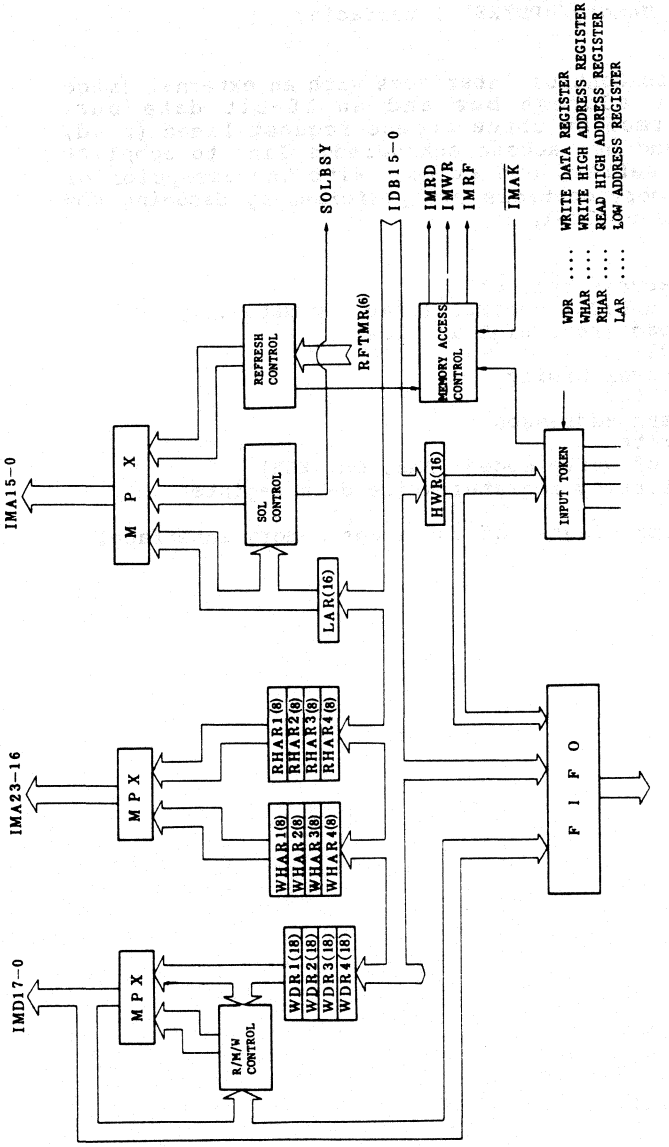
Image Memory/uPD7281 Interfacing

The Image Memory Access Control interfaces with an external image memory via a 24-bit address bus and an 18-bit data bus. Handshaking is performed by three access request lines (read, write, and refresh) and one access acknowledge line to complete the read, write, and refresh operation. With the exception of refresh, all image memory controls are performed by decoding the tokens output from the uPD7281.

Functions
- Image memory access request
- Read data --> uPD7281 control (read low data only)
- Self object load (read high and low)
- Refresh control
  - Refresh interval timing
  - Refresh inhibit
  - 10-bit refresh addresses
- Read/modify/write
  - Three types of write modes (AND, OR, XOR)
- High address setting register/write data register

Figure 5-1 shows a block diagram of the image memory interface.

Figure 5-1
Image Memory Interface



5.1 Image Memory Access Tokens

Because the uPD7281 does not have address ping to access an external memory special tokens output from the uPD7281 must be defined for each type of image memory access command. The image memory access tokens recognized by the uPD9305 are show in Table 5-1.

Because the data field of each token is 16 bits, in order to address the entire 24-bit memory space supported by the uPD9305, addressing must be performed by dividing the address into its high-order 8-bit and low-order 16-bit parts. In other words, the high-order bits of the address are first set in the high-order address register of the uPD9305 and with the arrival of the low-order address bits, the complete address is generated and then output to the address bus.

When performing write operations, in addition to the 24-bit address, 18-bit data to be written must be simultaneously provided. To do this, register setting tokens are first used to set appropriate values in the Write High Address Register (WHAR) and the Write Data Register (WDR). The write operation begins with the arrival of the write low-order address token.

The uPD9305 has a set of 12 internal registers. These are called registers files and provide four registers each for the Read High Address, Write High Address, and Write Data. These register files are used to improve the efficiency of image memory access and may be selected by the DIR field of the arriving image memory access tokens.

Table 5-1
Image Memory Access Tokens

- : don't care
S : Set, R : Read, W : Write

MN	Z	ID										C T L F	Data		Function	Op.
0001	-	MN'				ID'							Image memory read low address		Image memory read (RHAR1 reference)	R
		1	1	1	-	-	-	-	-			- - - -	- - - - - - - - -	Image memory read high address	Read high address register (RHAR1) set *	S
0010	-	MN'				ID'						- - - -	Image memory read low address		Image memory read (RHAR2 reference)	R
		1	1	1	-	-	-	-	-			- - - -	- - - - - - - - -	Image memory read high address	Read high address register (RHAR2) set *	S
0011	-	MN'				ID'						- - - -	Image memory read low address		Image memory read (RHAR3 reference)	R
		1	1	1	-	-	-	-	-			- - - -	- - - - - - - - -	Image memory read high address	Read high address register (RHAR3) set *	S
0100	-	MN'				ID'						- - - -	Image memory read low address		Image memory read (RHAR4 reference)	R
		1	1	1	-	-	-	-	-			- - - -	- - - - - - - - -	Image memory read high address	Read high address register (RHAR4) set *	S
0101	-	0	0	0	0	0	DIR					- - - -	Image memory write data		Image memory write (referencing WHAR and WDR selected by DIR)	W
		0	0	1	-	-	DIR					- - - -	- - - - - - - - -	Image memory write high address	Setting write high address register (WHAR) selected by DIR	S
		0	1	0	-	-	DIR					- - C,S	Image memory write low address		Setting write data register (WDR) selected by DIR	S
		0	1	1	-	-	DIR					- - - -	- - - - - - - - -	Image memory read high address	Setting read high address register (RHAR) selected by DIR	S
		1	0	0	MASK		DIR					- - - -	Read/write low address		Read/modify/write	RW
		1	0	1	-	-	DIR					- - - -	Read/write low address		Read/modify/write (write data CS selects mask)	RW
0110	-	0	0	-	-	-	DIR					- - - -	Load starting low address		Self object load	R
		0	1	-	-	-	DIR					- - - -	Load starting low address		Self object load (NM of output token is SOLMN)	R
		1	-	-	-	-	-					- - - -	- - - - - - - - - - - -	SOLMN	Setting SOLMN for self object load	S

ID' is the ID used for the next circulation.

MN' is the MN used for the next circulation (MN' ≠ 111).

DIR specifies the registers to be used for image memory access.

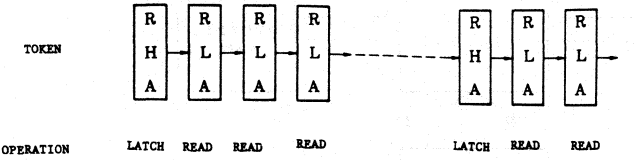
MASK specifies the modify mode (described later).

*:When the RHASEL bit of Mode Register is "1", the tokens become image memory read (request) token.

5.1.1 Image memory read/write procedures

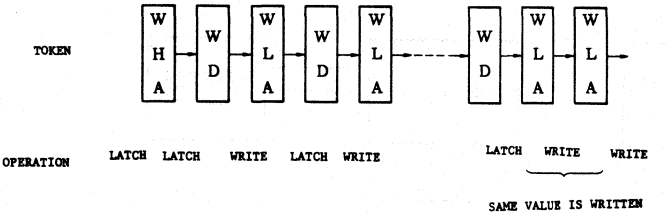
As shown in Table 5-1, tokens output from the μPD7281s with MN = 1 through 6 are used to access image memory.

(1) Read
Once the Read High Address (RHA) set token has been sent from the μPD7281s, there is no need to send this token again until the Read High Address is to be changed.



Each time a Read Low Address (RLA) token arrives, the data field of this token and that of the RHA token are used to generate the image memory address.

(2) Write
Once a Write High Address (WHA) register set token and a Write Data (WD) register set token have been sent from the μPD7281s, there is no need to send these tokens again until the write high address or data is to be changed.



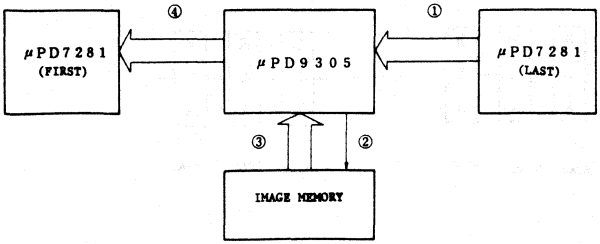
Each time a Write Low Address (WLA) token arrives, the data in the WD register specificed by the DIR field of the arriving token is written to the image memory addressed by WLA and the WHA specified by DIR.

Note: To write data to the image memory, output the WLA token after the WD token has been output. There is, however, no need to output the WD token when the same data is to be written (memory clear, etc.).

5.1.2 Image Memory Access Request

With the exception of refresh, all image memory accesses are initiated by the arrival of the appropriate input token from the uPD7281.

- Operation flow for image memory read



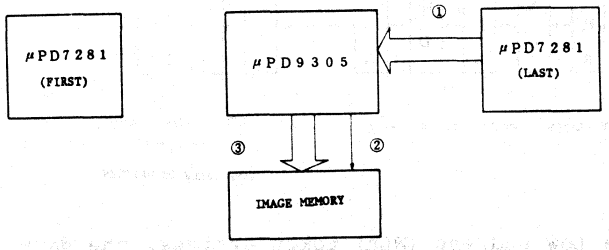
① IMAGE MEMORY READ REQUEST TOKEN

② READ REQUEST TO IMAGE MEMORY

③ READ DATA

④ IMAGE MEMORY READ TOKEN

- Operation flow for image memory write



① IMAGE MEMORY WRITE REQUEST TOKEN

② WRITE REQUEST TO IMAGE MEMORY

③ WRITE OPERATION

Figure 5-2
Operation Flow for Image Memory Read/Write

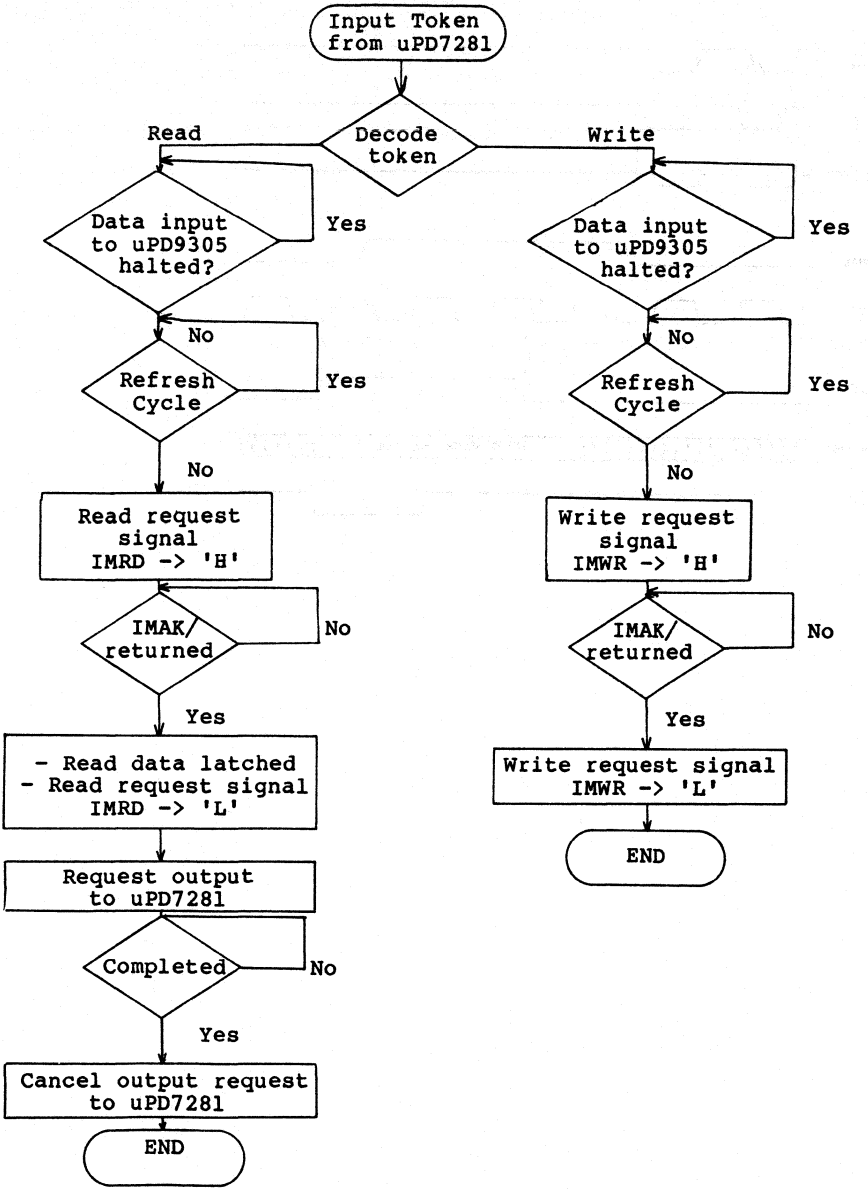


Figure 5-3
Image Memory Read Timing
(without refresh request)

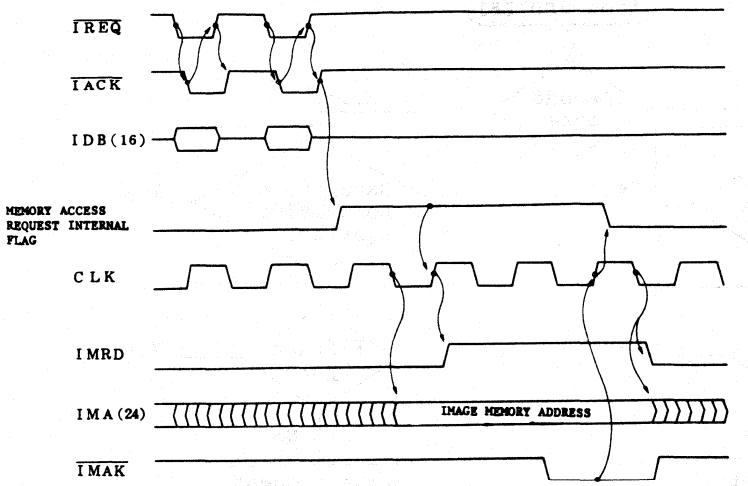
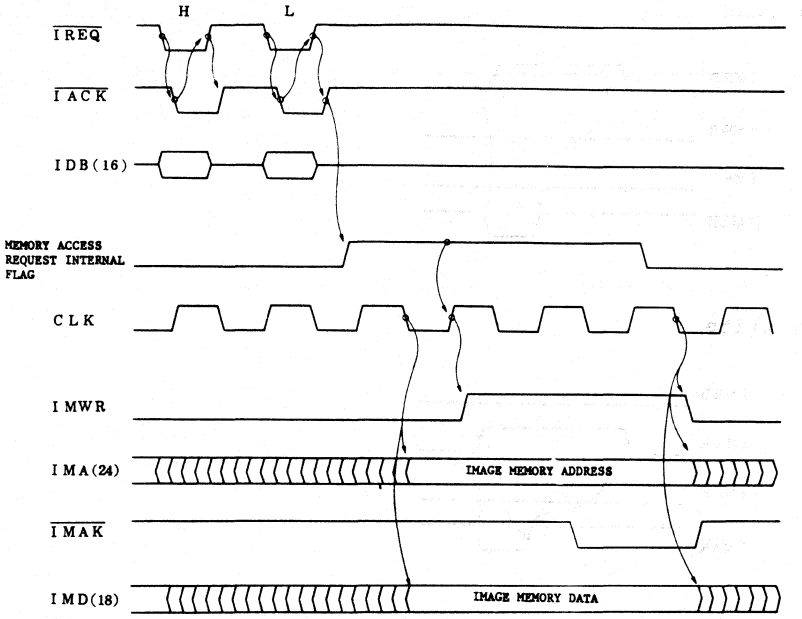


Figure 5-4
Image Memory Write Timing
(without refresh request)



- Four-line handshaking

Interfacing between the uPD9305 and external image memory is performed asynchronously using a four-line handshaking protocol.

Image memory read

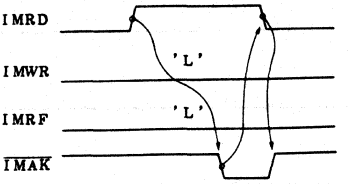


Image memory write

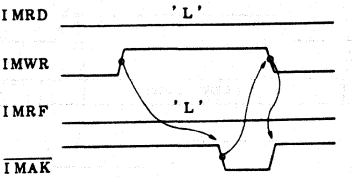
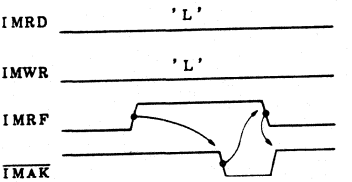


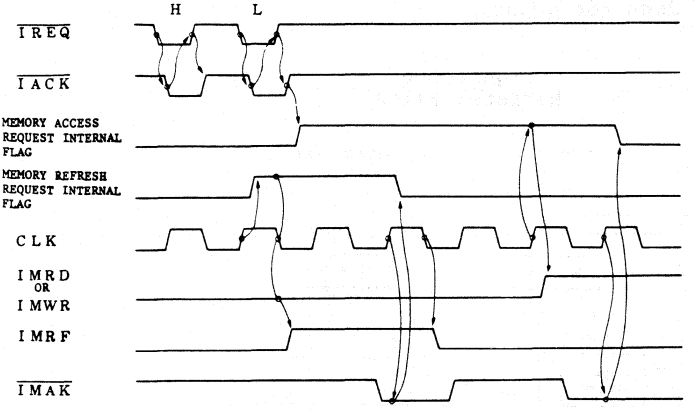
Image memory refresh



- Priority Control

When an image memory read or image memory write request token arrives at the uPD9305, the uPD9305 first exercises priority control to resolve any conflict with internally generated refresh requests. If no refresh request exists, the read request signal (IMRD) or write request signal (IMWR) is output. However, if a refresh request exists, the IMRD or IMWR signals are not output until the refresh operation finishes.

Figure 5-5
Image Memory Access Request Priority Control

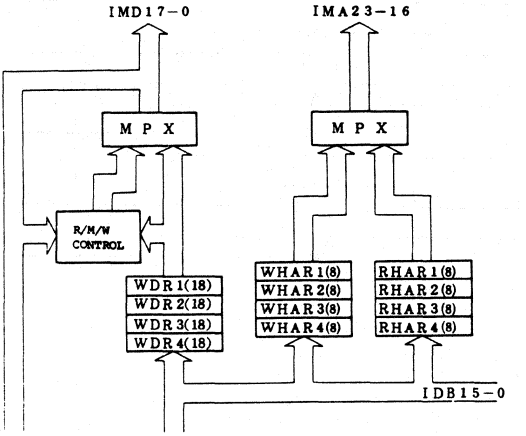


5.2 Register Files

The uPD9305 can support up to eight uPD7281s. However, one problem that occurs when multiple uPD7281s are connected in a system is that, if more than one uPD7281 are outputting image memory access tokens at the same time, there is a danger that a token output by one uPD7281 will be interposed between the high and low address tokens of another uPD7281. Should this happen, it would result in an erroneous operation.

As a countermeasure to this, the uPD9305 has four register files, each dedicated to a separate uPD7281. Each register file consists of High Address Registers (four for read and four for write) and four Write Data Registers. However, if five or more uPD7281s are used, care must be exercised so that the problem described above does not occur.

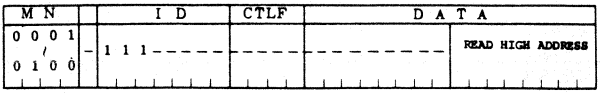
Figure 5-6
Register Files



(1) RHA Register set token

The following two types of tokens are used to set the Read High Address Register (RHAR). Note that (A) becomes a read request token (returning MN = 7) if RHASEL = 1.

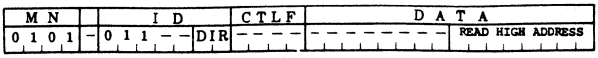
(A)



This token has the same MN as the read request token. Therefore, you can use the OUT2 instruction in the uPD7281 program. Both tokens can be sequentially output.

Note: MN = 0001 .... RHAR1
MN = 0010 .... RHAR2
MN = 0011 .... RHAR3
MN = 0100 .... RHAR4

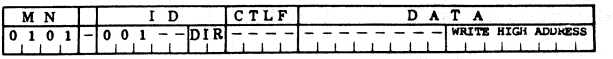
Ⓑ



If RHASEL = 1, this is the only token that can set RHA.

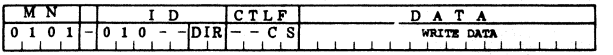
Note: DIR = 00 .... RHAR1
DIR = 01 .... RHAR2
DIR = 10 .... RHAR3
DIR = 11 .... RHAR4

(2) WHA Register Set Token



Note: DIR = 00 .... WHAR1
DIR = 01 .... WHAR2
DIR = 10 .... WHAR3
DIR = 11 .... WHAR4

(3) WD Register Set Token

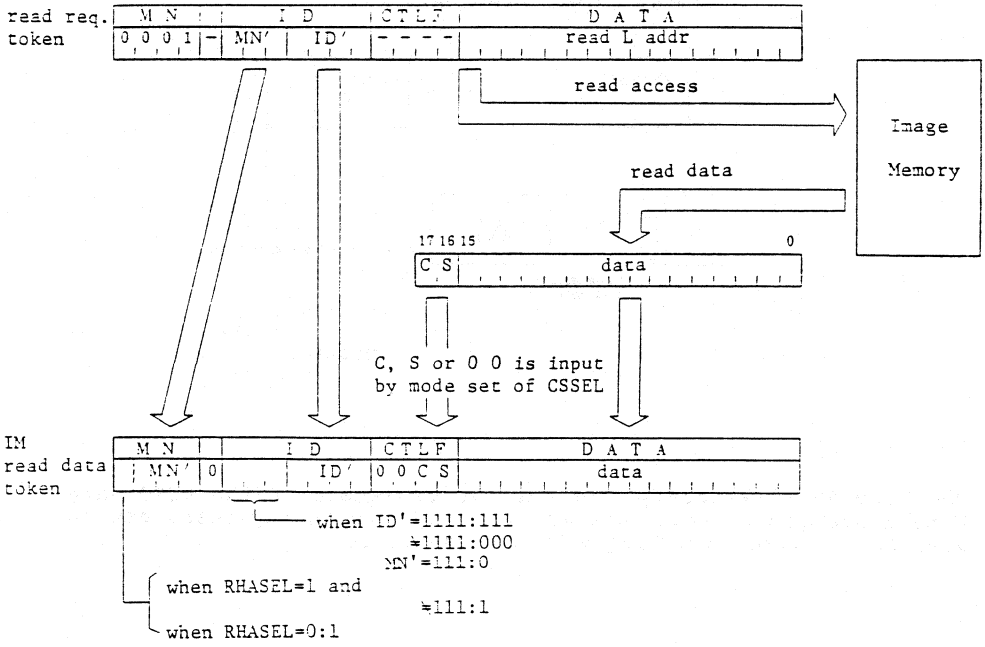


Note: DIR = 00 .... WDR1
DIR = 01 .... WDR2
DIR = 10 .... WDR3
DIR = 11 .... WDR4

5.3 Output Control (Read Data -> uPD7281)

The uPD9305 handles two types of data. The first is normal image data and the second is the data used for self object load. Therefore, the uPD9305 has two different methods for reading image memory. The method of reading self object load data is described later. The method of reading image data is described in this section.

When the uPD9305 receives an image memory read request token from a uPD7281, it decodes this token to perform access of the image memory. The data read from the image memory is an 18-bit word (16 bits of data and C and S bits). As shown in the following figure, this data is combined with parts of the read request token to configure the read data token.

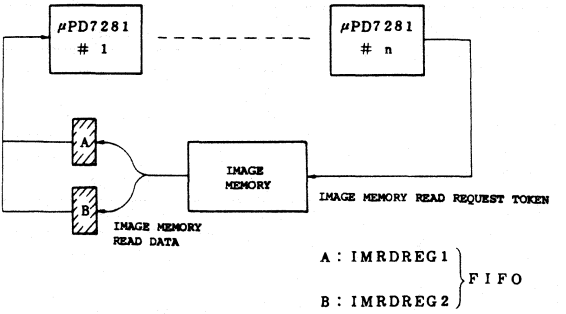


- Improving the speed of image memory read operations

In a uPD7281 system, image memory read operations start when a read request token is input from the last-stage processor (uPD7281 #n) and completes when the data read from the image memory is output to the first stage processor (uPD7281 #1). It is desirable to improve the speed of the image memory read operation as a means to improve the overall processing speed of the uPD7281 system.

To improve the speed, the uPD9305 has a memory read data FIFO (two-level register) to store the image data output to uPD7281 #1. This permits the next image memory read operation to be performed while the previous image memory read data is being output to uPD7281 #1. The hardware configuration for this FIFO is shown in Figure 5-7.

Figure 5-7
Image Memory Read Data FIFO



The uPD9305 alternately uses the two registers of the FIFO each time image memory read data arrives. When both registers are in use, image memory read requests are inhibited.

Figure 5-8
Read Data --> uPD7281 Output Timing
(Single Output)

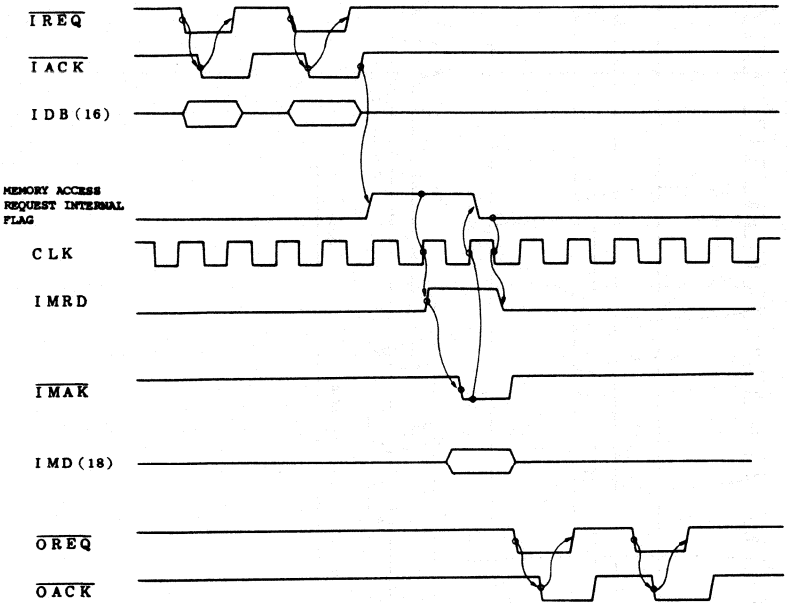
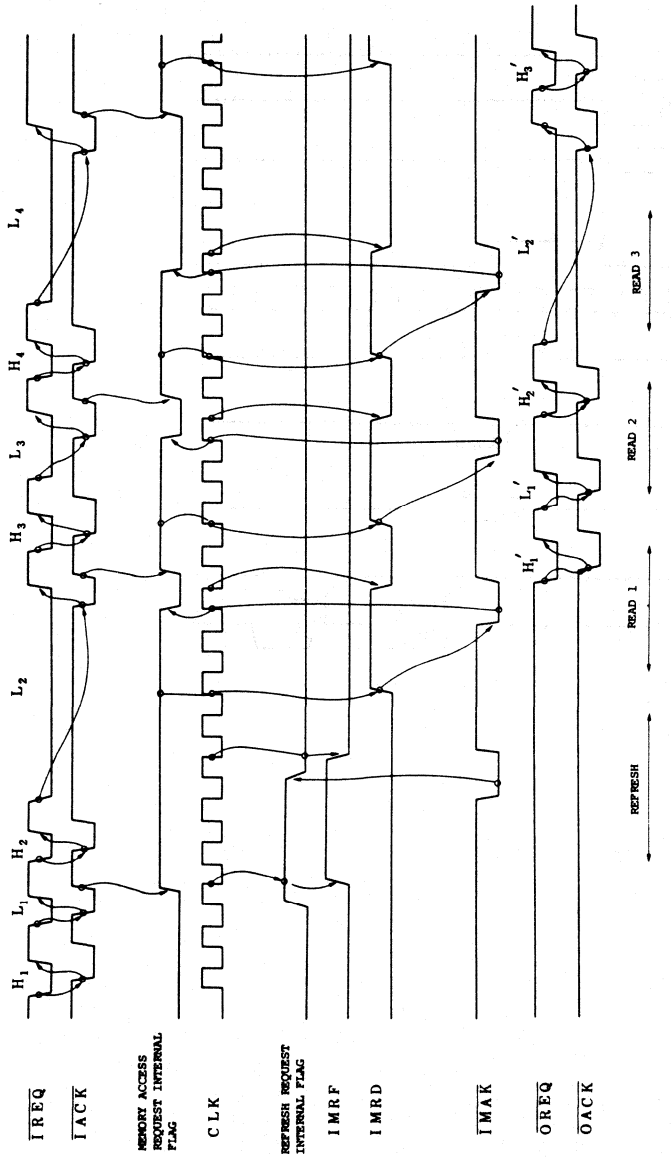


Figure 5-9
Read Data --> uPD7281 Output Timing
(Continuous Output)



5.4 Self Object Load

Before the uPD7281s can perform any processing, the processing program must first be downloaded into the uPD7281s. Normally, download is performed by the host processor with the host generating the tokens that set the program and outputting these to the uPD7281. As a result, in a case where downloading must be performed frequently, the overhead of the host computer is increased and operation efficiency suffers.

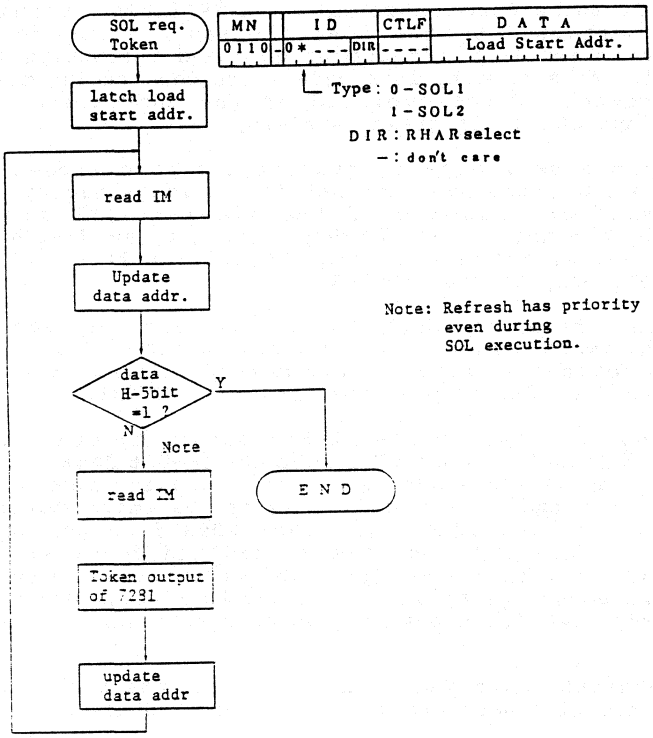
The uPD9305 has a self object load feature to overcome this problem. Once a program has been set in the image memory, this feature permits downloading of the program with minimal involvement by the host. This is done simply by sending a self object load request token from the uPD7281 (or from the host through the uPD7281s) to the uPD9305.

The image memory data read method described earlier reads the contents of the image memory. However, the ID and CTLF fields of the read data token are fixed in this method. Therefore, it is not appropriate for loading programs into the uPD7281.

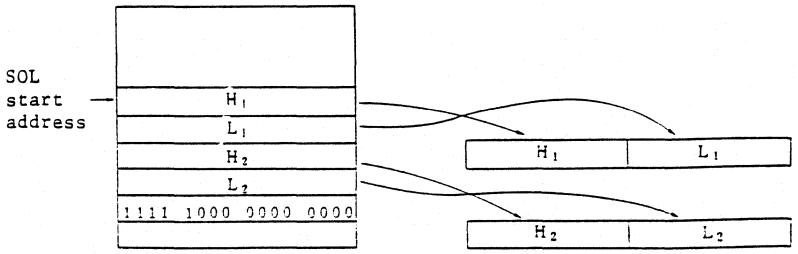
When a self object load request token output by the uPD7281 arrives at the uPD9305, the uPD9305 uses the 16-bit data field of the arriving token as the starting address for image memory read. In this case, only the low-order 16 bits of the image memory are read (C and S bits are ignored). The uPD9305 automatically increments the address to perform a read and continues this operation until read terminate data* is encountered. Note that the address increment operation is limited to 16 bits and values larger than this (64K locations) cannot be specified.

* : Read terminate data should be such that the high-order 5 bits of the high data are all '1' (i.e., F800H).

Figure 5-10
Operation Flow for Self Object Load

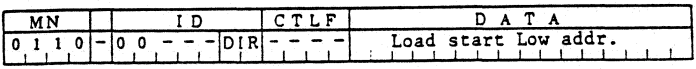


Read data is paired to create program setting tokens that are then sent to the uPD7281s.



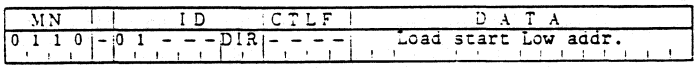
The following two types of tokens are used to start self object load:

1) SOL1 (Read All)



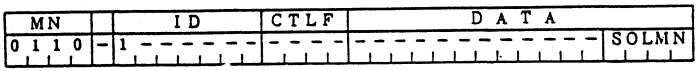
This token generates the starting address for self object load in image memory from the RHAR register referenced by the DIR field and the contents of the data field. The high-order 2 bits (C and S) of the 18-bit read data are ignored. The uPD9305 automatically increments the read address (lower 16 bits), reads the low word, and sets the OREQ/ signal to its active level to start output to the uPD7281. The uPD9305 continually repeats the above operation until terminate data is read. The termination code is all 5 high-order bits set to '1' in the high word (i.e., F800H). The SOL1 token is used when there is no need to change the MN to which the object is to be loaded. If the object program contains multiple MNs, this token must be used.

2) SOL2 (Replace MN field)



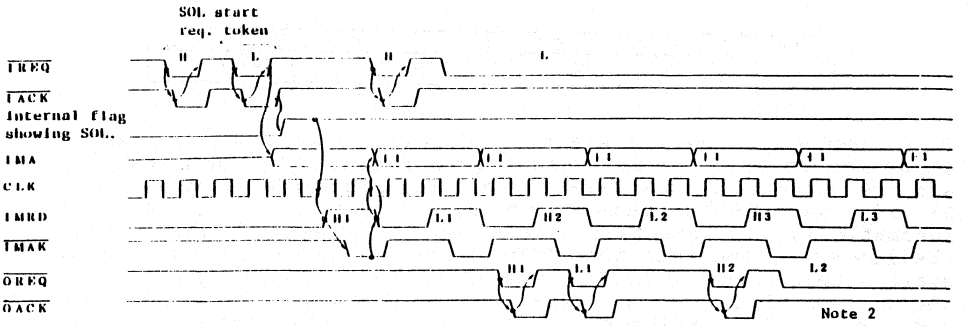
The processing sequence is almost identical to that for the SOL1 token. The difference is that the MN field of the tokens output to the uPD7281(s) are replaced by the contents of the SOLMN register. The value of the SOLMN register is set by the SOLMN token described next.

3) SOLMN (SOLMN Register Set)



This token is used when all of the MNs of the object program are the same. This token can be used effectively in combination with the SOL2 token described above to set the same object program in multiple uPD7281s, resulting in important savings in the amount of image memory used for object program storage.

Figure 5-11
Self Object Load Timing



Note: If a refresh request is generated during execution of self object load, refresh is given priority.

Note: When $\overline{\text{OACK}}$ is not returned, uPD9305 can retain up to two tokens in it. (In the Figure, (H2)-L2 and H3-L3 tokens). IMRD will not become active untill the token is output.

5.5 Refresh Control

The uPD9305 Refresh Control generates refresh addresses for the external image memory. The refresh cycle is programmable and may be set to any multiple of from 1 to 64 using 8 times the input clock as the basic unit.

Refresh control features:

- Ample refresh address length ... 10 bits (IMA9 - IMA0)
- Programmable refresh timing counter ... 6 bits
  (Refresh cycle: 8 - 512 CLK)
- Initialized by system reset only
  (Unaffected by command reset)

The timing and block diagrams for the refresh control are shown in Figures 5-12 and 5-13.

Figure 5-12
Refresh Timing

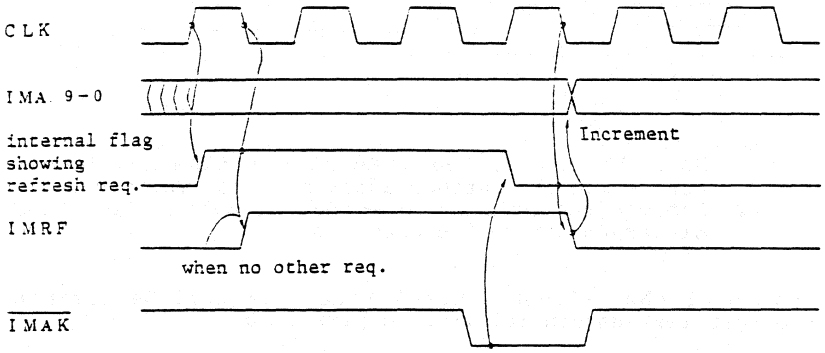
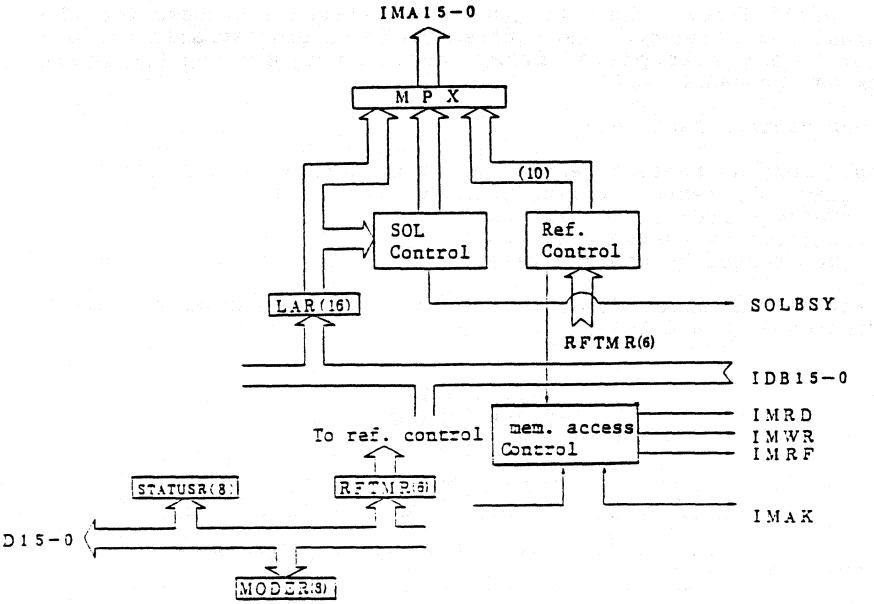


Figure 5-13
Refresh Control



Note:
1. When RFINH = 1, the IMRF signal does not become active and the refresh address is not incremented.
2. Refresh has priority even during self object load or read/modify/write execution.

RFINH (bit 6 of the Refresh Control Mode Register) determines whether or not refresh control will be performed.

Bit No.	Mode name	Value	Mode
6	RFINH	0	IMRF is output (refresh control)
		1	IMRF is not output

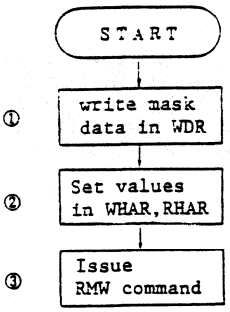
The refresh timing counter determines the length of the refresh cycle. By setting a value of from 0 to 63 in this counter, a refresh cycle of from 8 to 512 CLK may be set. The refresh timing counter is selected when A1,A0 = '11'. For details, refer to Section 4.1, Input/Output Ports. If RFINH is set to 1, the refresh timing counter and the refresh address retain the values held immediately prior to the bit being set.

5.6 Read/Modify/Write

The uPD9305 read/modify/write function performs the following series of operations: read data from the image memory, logical operation (AND, OR, XOR) between the data read and preset (mask) data, and writing the result back to the image memory. The 'preset data' referred to here is data that has been set in the Write Data Register (WDR). When this read/modify/write operation is executed, all other image memory access tokens are inhibited until the operation has finished.

- Logical function (modify mode): AND, OR, or XOR with mask

- Operating procedure:



- Different values may be written to each of the four WDRs and the DIR field used to select from among the register files.
- By specifying different values for WHAR and RHAR, it is possible to read and write the result to separate addresses. Note, however, that the low-order 16 bits of the read and write address are always the same.
- A refresh cycle may be executed between the read and write operations.

The order of procedures 1 and 2 may be reversed. Once done, they may be omitted unless it is desired to change the data in WDR, WHAR, or RHAR.

There are two different types of RMW commands that can be used to perform read/modify/write. The difference between these two is in the MASK data specification.

- RMW1 command

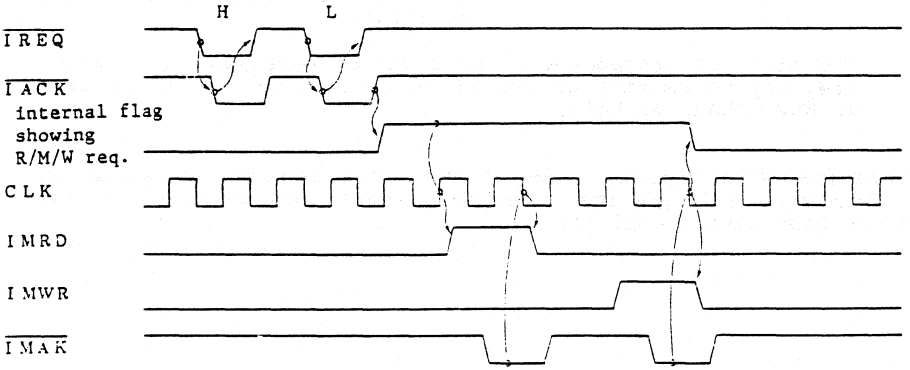
MN		ID				CTLF	DATA
0 1 0 1	-	1 0 0	MASK	DIR		- - - -	read/write address

DIR = 00 selects RHAR1, WHAR1, WDR1
DIR = 01 selects RHAR2, WHAR2, WDR2
DIR = 10 selects RHAR3, WHAR3, WDR3
DIR = 11 selects RHAR4, WHAR4, WDR4

MASK = 00 write the contents of WDR to the image memory
MASK = 01 AND image memory data with contents of WDR
MASK = 10 OR image memory data with contents of WDR
MASK = 11 XOR image memory data with contents of WDR

RMW1 is used to select a single mask (independent of the values of the C and S bits which indicate the uPD7281 operation result). However, the specifying of MASK is done by ID field data of the command, so multiple OUT instructions are needed to use different masks. The timing diagram for read/modify/write if shown in Figure 5-14.

Figure 5-14
Read/Modify/Write Timing Diagram



- RMW2 command

MN		ID	CTLF	DATA
0 1 0 1	-	1 0 1 - - DIR	- - - -	read/write address

DIR = 00 selects RHAR1, WHAR1, WDR1
DIR = 01 selects RHAR2, WHAR2, WDR2
DIR = 10 selects RHAR3, WHAR3, WDR3
DIR = 11 selects RHAR4, WHAR4, WDR4

MASK is specified by the upper two bits (C and S) of the selected WDR

C,S = 00 writes the contents of WDR to the image memory
C,S = 01 AND image memory data with contents of WDR
C,S = 10 OR image memory data with contents of WDR
C,S = 11 XOR image memory data with contents of WDR

The RMW2 command is used when it is desired to change the mask according to the C and S bits of the uPD7281 operation result. The MN and ID fields of the RMW2 command are common to every MASK selection (this is different from the RMW1 command) so that the same OUT instruction can be used in the program. The RMW2 command can be used to conditionally change a particular bit of a word to '1' or '0' (i.e., drawing of broken or dotted lines, rotation, etc.).

5.7 C and S Bits of Data Read from the Image Memory

In order to conform with the data format of the uPD7281, the data bus connecting the uPD9305 and the image memory has an 18-bit data width. There are many applications that may use 16-bit memory words. Writing these 16-bit words to the image memory presents no problem. However, when the image memory is read, the problem of how to handle the C and S bits arises.

Image memory read data token:

MN		ID	CTLF	DATA
MN'	0	ID'	0 0 C S	read data

The handling of the C and S bits is determined by the state of bit 1 (CSSEL) of the Mode Register, as shown below.

Bit No.	Mode name	Value	Mode
1	CSSEL	0	Image memory read data C,S bits are 00
		1	Image memory read data C,S bits as is

The CSSEL bit must be set to '0' for an image memory data width of 16 bits, and '1' for 18-bit data width. If CSSEL is set to '0' with an 18-bit image memory data width, the C and S bits will be both set to '0', regardless of the values read from the memory.

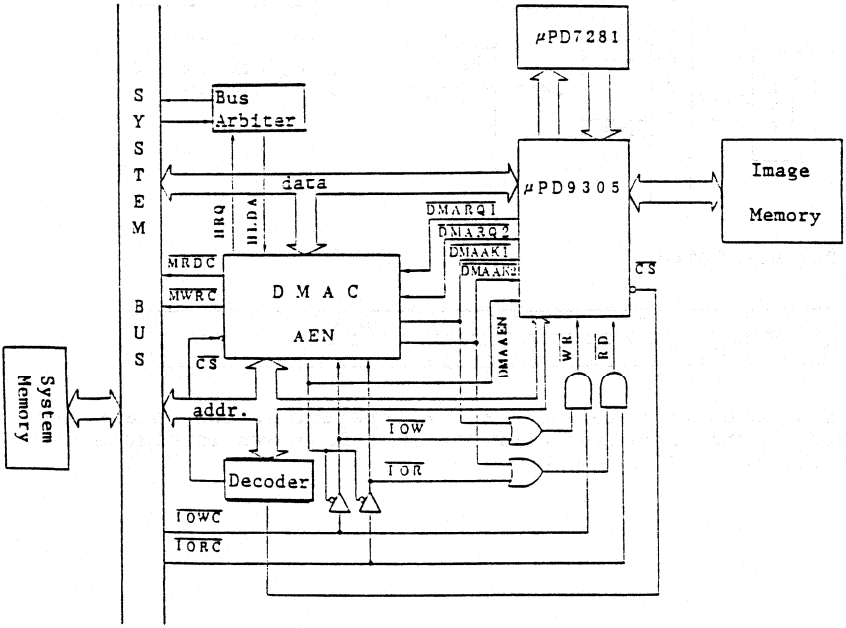
Chapter 6

DMA Request Operation

The uPD7281 handles large-volume data such as image data. There is no difficulty if the image memory incorporates a Direct Memory Access (DMA) transfer capability. However, if there is no DMA transfer function in the image memory, there is a danger that transfers will be unacceptably slow. To solve this problem, the uPD9305 used with a DMA Controller (DMAC) supports a DMA request operation enabling DMA transfers between the system memory and the image memory (Figure 6-1). The image memory addresses for DMA transfer are generated by the uPD7281 program and, in this sense, one could say that the DMA transfer actually is between the I/O (uPD9305) and memory (sysem memory).

In the uPD9305, transfers from the system memory to the uPD9305 (image memory) are referred to as DMA1, and those from the uPD9305 (image memory) to the system memory as DMA2.

Figure 6-1
Block Diagram of uPD9305 System Using DMAC

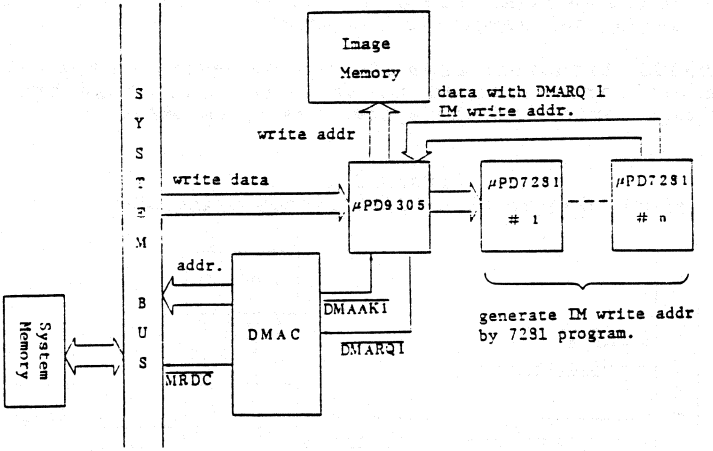


6.1 DMA1 (System Memory -> Image Memory)

Because the addresses for the system and image memories are generated by the external DMAC and by the uPD7281 program, respectively, the role of the uPD9305 is limited to performing control between the DMAC and the uPD7281s (Figure 6-2).

The DMARQ1/ signal output from the uPD9305 to the DMAC is set to the active level (low) when a DMA1 request token (MN = 5, ID = 6xH) arrives from the uPD7281. This signal then returns to the inactive level when the uPD9305 receives the DMAAK1/ signal from the DMAC.

Figure 6-2
DMA1 Schematic Diagram

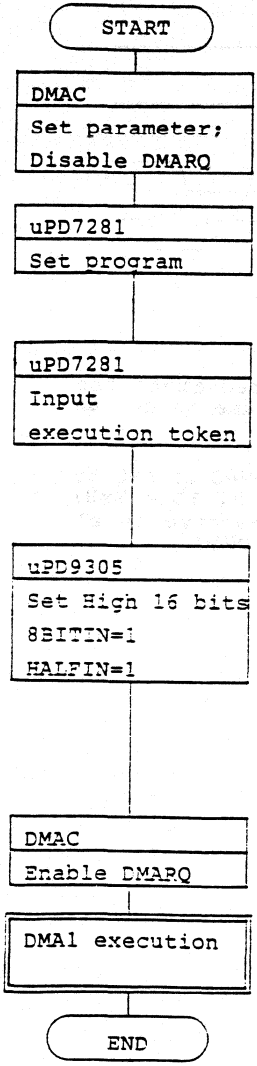


- DMA1 Request Token

MN		ID	CTLF	DATA
0 1 0 1	-	1 1 0 - - - -	- - - -	- - - - - - - - - - - - - - - -

Since this token only requests a DMA1 operation, the data field is ignored. The execution flow for DMA1 is shown in Figure 6-3 and the timing in Figure 6-4.

Figure 6-3
DMA1 Execution Flow



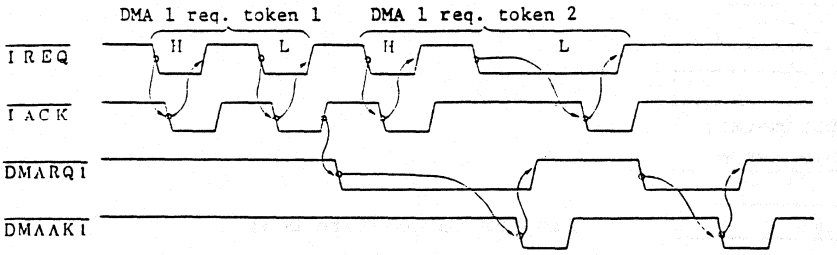
Program to generate DMA1 request token and image memory write addresses is set in uPD7281

Program is started by Execution token, then $\overline{\text{DMARQ1}}$ becomes active. However, DMARQ is disabled and not acknowledged by the DMAC.

Set the mode of uPD9305 :
-set high order 16-bit
-two 8-bit data from system memory is to be sent to uPD9305 and packed to 16-bit as low 16-bit of the token, then $\overline{\text{OREQ}}$ is output.

The uPD9305 A1,A0 inputs are ignored while DMAAEN=1; internal addresses set to 0.

Figure 6-4
DMA1 Timing

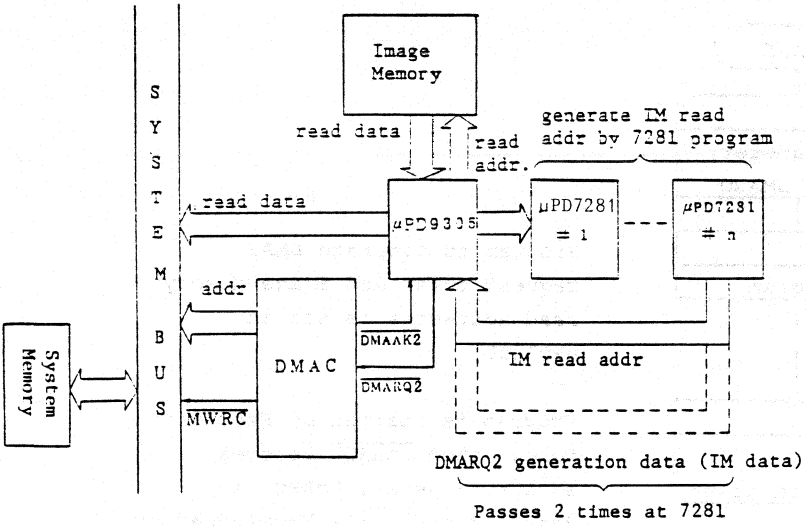


6.2 DMA2 (Image Memory -> System Memory)

For DMA2, the processing load is divided and shared among the three devices (DMAC, uPD7281, and uPD9305) in the same manner as DMA1 (Figure 6-5).

The DMARQ2/ signal output from the uPD9305 to the DMAC is set to the active level when a DMA2 request token (MN = 5, ID = 7xH) arrives from the uPD7281. This signal is set to inactive level when the uPD9305 receives a DMAAK2/ signal from the DMAC.

Figure 6-5
DMA2 Schematic Digram

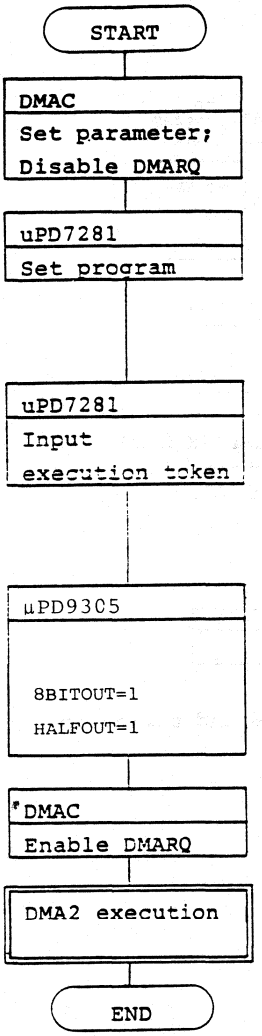


- DMA2 Request Token

MN		ID	CTLF	DATA
0 1 0 1	-	1 1 1 - - - -	- - - -	IM read data

The execution flow for DMA2 is shown in Figure 6-6 and timing in Figure 6-7.

Figure 6-6
DMA2 Execution Flow



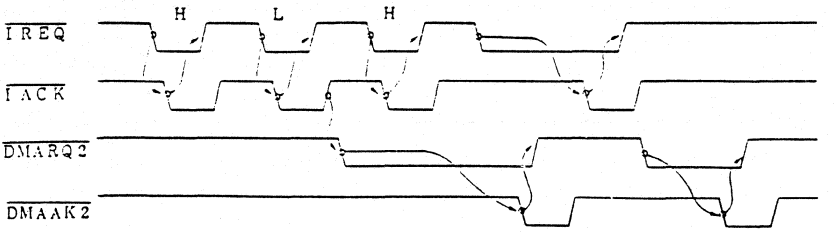
Program to generate DMA2 request token and image memory read addresses is set in uPD7281

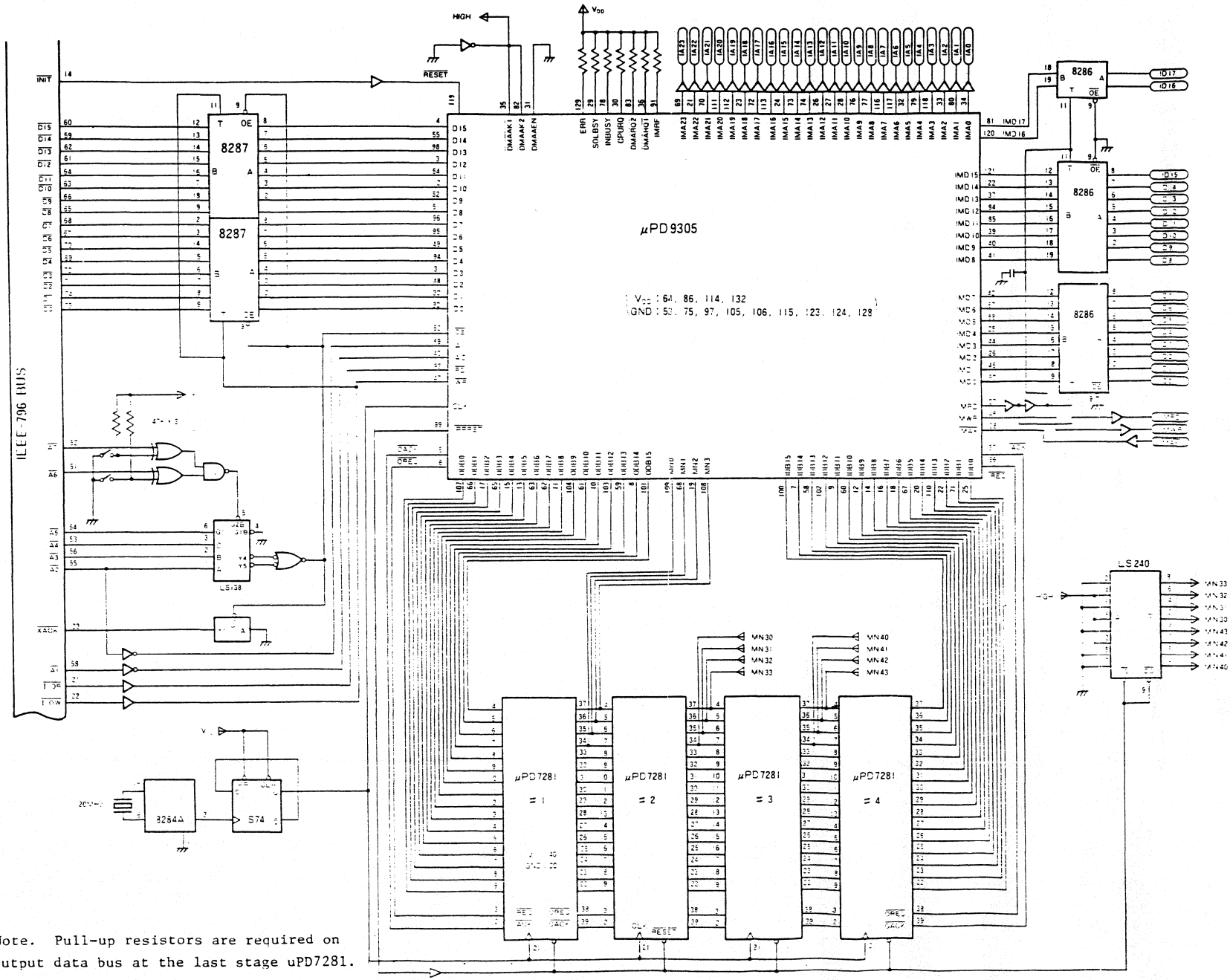
Program is started by Execution token, then $\overline{\text{DMARQ2}}$ becomes active. However, DMARQ is disabled and not acknowledged by the DMAC.

Because the 16-bit data of the image memory data is output twice in two 8-bit words, the single image memory data must be output as two DMA2 tokens.

The uPD9305 A1,A0 inputs are ignored while DMAAEN=1; internal addresses set to 0.

Figure 6-7
DMA2 Timing





Note. Pull-up resistors are required on output data bus at the last stage uPD7281.

Appendix A

System Circuit Diagram Example

The system configuration shown in Figure A-1 consists of the following elements:

(1) Four uPD7281s
(2) The system host and the uPD9305 are connected by an IEEE-796 Bus
(3) DMA is not used
(4) The uPD9305 refresh function is not used.

B

μPD9305
MEMORY ACCESS AND GENERAL BUS INTERFACE CHIP (MAGIC) / SUPPORT CHIP FOR μPD7281

ELECTRICAL SPECIFICATION

μPD9305 ELECTRICAL SPECIFICATION

ABSOLUTE MAXIMUM RATINGS (T_a = 25°C)

Parameter	Symbol	Test Conditions	Ratings	Units
Supply Voltage	V_{DD}		−0.5 to 7.0	V
Input Voltage	V_I		−0.5 to 7.0	V
Output Current	I_O		10	mA
Operating Temperature	T_{opt}		0 to 70	°C
Storage Temperature	T_{stg}		−65 to 150	°C

DC Characteristics (T_a = 0 to + 70°C, V_{DD} = 5V ± 10%)

Parameter	Symbol	Test Conditions	Min.	Typ.	Max.	Units
Input Low Voltage	V_{IL}		−0.5		0.8	V
Input High Voltage	V_{IH}		2.0		V_{DD}+0.5	V
Output Low Voltage	V_{OL}	I_{OL} = 2mA			0.4	V
Output High Volatge	V_{OH}	I_{OL} = −400 μA	V_{DD}−0.4			V
Input Leakage Current	I_{LI}	$0 \leqq V_I \leqq V_{DD}$			± 10	μA
Output Leakage Current	I_{LO}	$0 \leqq V_I \leqq V_{DD}$			± 10	μA
Supply Current	I_{DD}	10 MHz		10	100	mA

Capacitance (T_a = 25°C)

Paramater	Symbol	Test Conditions	Min.	Typ.	Max.	Units
Input Capacitance	C_I	fc = 1 MHz			10	p F
Output Capacitance	C_O	Unmeasured Pins			15	p F
I/O Capacitance	C_{IO}	return to ØV			15	p F

AC Characteristics (T_a = 0 to + 70°C, V_{DD} = 5V ± 10%)

Clock Timing

Paramater	Symbol	Test Conditions	Min.	Max.	Units
CLK Cycle Time	t_{CYK}		80		ns
Clock Pulse width high	t_{WKH}		30		ns
Clock Pulse width low	t_{WKL}		30		ns
Clock Rise Time	t_{KR}			10	ns
Clock Fall Time	t_{KF}			10	ns

Input Timing

Parameter	Symbol	Test Conditions	Min.	Max.	Units
Input Rise Time	t_{IR}		0	10	μs
Input Fall Time	t_{IF}		0	10	μs

RESET Timing

Parameter	Symbol	Test Conditions	Min.	Max.	Units
RESET/Pulse Width	t_{RST}	μPD9305 (only)	t_{CYK}		ns
RESET/Setup time to IPPRST/	t_{DRSPRL}			40	ns
IPPRST/Hold Time after RESET/up	t_{DRSPRH}			50	ns
IPPRST/Setup to MN0-3	t_{DMN}			60	ns
MN0-3 Float time after IPPRST/up	t_{FMN}			50	ns
IPPRST/Low Till ODB15-12 Active	t_{DPROD}			60	ns
ODB15-12 Float time after IPPRST/up	t_{FPROD}			50	ns

Host «---» μPD9305 Read/Write (Timing)

Parameter	Symbol	Test Conditions	Min.	Max.	Units
Address Setup to WR/down, RD/down	t_{SARW}		20		ns
Addres hold time after WR/up, RD/up	t_{HRWA}		20		ns
CS/setup to WR/down, RD/down	t_{SCRW}		0		ns
CS/Hold time after WR/up, RD/up	t_{HRWC}		0		ns
WR/, RD/Pulse Width	t_{WRWL}		100		ns
RD/Setup to Data	t_{DRD}			80	ns
Data Float time after RD/up	t_{FRD}			30	ns
Data setup to WR/	t_{SDW}		20		ns
Data Hold after WR/up	t_{HWD}		20		ns

I/O Request/Acknowledge Timing

Parameter	Symbol	Test Conditions	Min.	Max.	Units
IREQ/ setup time to IACK/	t_{DIQIAL}		15	60	ns
1st IACK/setup time to IREQ/up	t_{DIAIQH}		10		ns
1st IREQ/up setup time to IACK/up	t_{DIQIAH}		20	70	ns
1st IACK/up setup to IREQ/low	t_{DIAIQL}		10		ns
ID Bus setup time to IREQ/up	t_{SIDIQ}		20		ns
ID Bus Hold time from IREQ/up	t_{HIQID}		10		ns
OREQ/setup time to OACK/	t_{DOQOAL}		10		ns
OACK/setup time to OREQ/up	t_{DOAOQH}		20	70	ns
OREQ/up setup time to OACK/up	t_{DOQOAH}		10		ns
OACK/up setup time to OREQ/low	t_{DOAOQL}		15	60	ns
OREQ/setup time to ODB Valid	t_{DOQOD}			10	ns
ODB Float time after OREQ/up	t_{FOQOD}		10		ns

DMA Transfer Timing

Parameter	Symbol	Test Conditions	Min.	Max.	Units
DMARQ/low setup time to DMAAK/low	t_{DDQDA}		20		ns
DMARQ/Hold time from DMAAK/low	t_{DDADQ}			50	ns
DMARQ/Recovery Time DMAAK/high	t_{RVDQ}		50		ns
DMAAEN up setup time to (RD/, WR/) low	t_{SDERW}		30		ns
DMAAEN low Hold time after (RD/, WR/) high	t_{HRWDE}		30		ns
DMAAK/low setup time to (RD/, WR/) low	t_{SDARW}		0		ns
DMAAK hold time after (RD/, WR/) high	t_{HRWDA}		0		ns
DMAAK/pulse width (low to high)	t_{WDAL}		t_{CYK}		ns

DMARQ/ = DMARQ1/, DMARQ2/
DMAAK/ = DMAAK1/, DMAAK2/

Image Memory Read, Write, Refresh Timing

Parameter	Symbol	Test Conditions	Min.	Max.	Units
IMA Active time CLK	t_{DKMARF}	IM Refresh		100	ns
IMA Active time CLK	t_{DKMAMC}	IM Read/IM Write		60	ns
IMA Float time from IMWR/IMRD low	t_{FMCMA}		10		ns
IMWR/IMRD Recovery time	t_{RVMC}		1.5 t_{CYK}		ns
IMWR/IMRD/IMWR high delay time from CLK high	t_{DKMCH}			35	ns
IMWR/IMRD/IMWR low delay time from CLK low	t_{DKMCL}			40	ns
IMAK/setup time to CLK down	t_{SMKK}		10		ns
IMD setup time to CLK high	t_{SMDK}	Image Memory Read Timing	20		ns
IMD hold time from IMRD low	t_{HMRMD}	Image Memory Read Timing	0		ns
IMD delay time from CLK low	t_{DKMD}	Image Memory Write Timing		30	ns
IMD Float time from IMWR low	t_{FMWMD}	Image Memory Write Timing	20		ns
IMAK/Recovery time	t_{RVMK}		1.5 t_{CYK}		ns
IMAK/hold time from IMRD/IMWR low	t_{HMCMK}		0		ns

SOLBSY Timing

Parameter	Symbol	Test Conditions	Min.	Max.	Units
SOLBSY Delay time from IACK/up	t_{DIASB}			30	ns
SOLBSY delay time from CLK low	t_{DKSB}			60	ns

CPURQ Timing

Parameter	Symbol	Test Conditions	Min.	Max.	Units
CPURQ delay time from IACK/high	t_{DIAPQ}			30	ns
CPURQ delay time from RD/high	t_{DPRQ}			60	ns

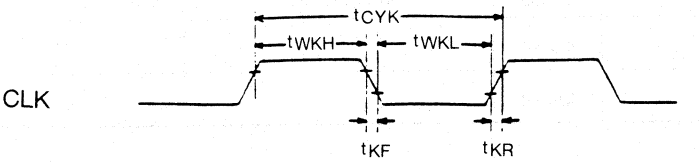
INBUSY Timing

Parameter	Symbol	Test Conditions	Min.	Max.	Units
INBUSY delay time from WR/high	t_{DWIB}			70	ns
INBUSY delay time from OREQ/high	t_{DOQIB}			40	ns

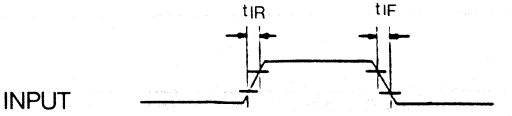
ERR Timing

Parameter	Symbol	Test Conditions	Min.	Max.	Units
ERR delay time from IACK/high	t_{DIAE}			30	ns
ERR delay time from WR/low	t_{DWE}	INBUSY = 1		60	ns
ERR delay time RD/low	t_{DRE}	CPURQ = 0		60	ns
INBUSY hold time from WR/low	t_{HWIB}			10	ns
CPURQ setup time to RD/low	t_{SPQR}			10	ns

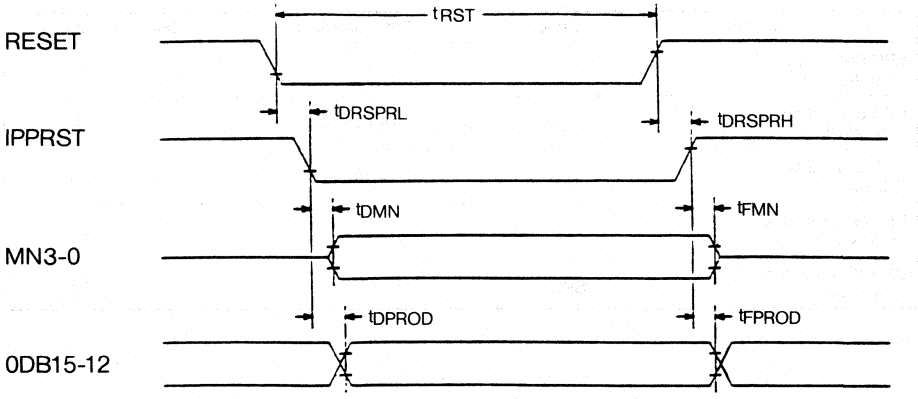
CLOCK TIMING



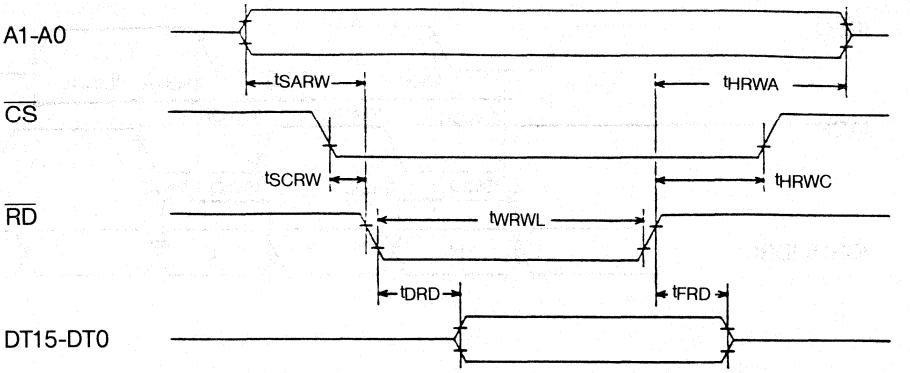
INPUT TIMING



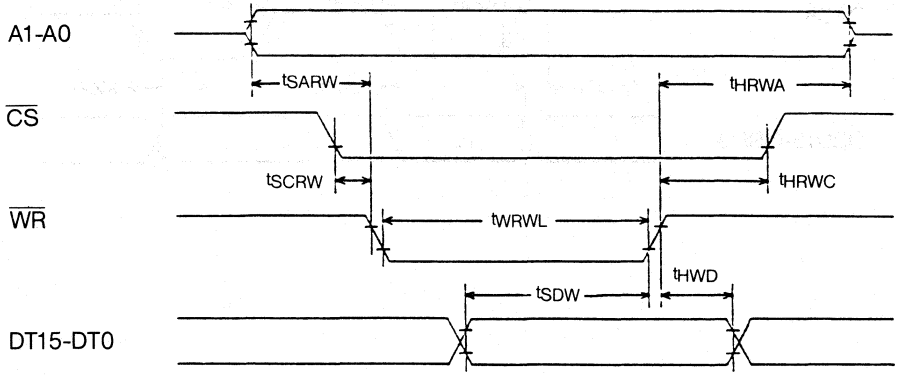
RESET TIMING



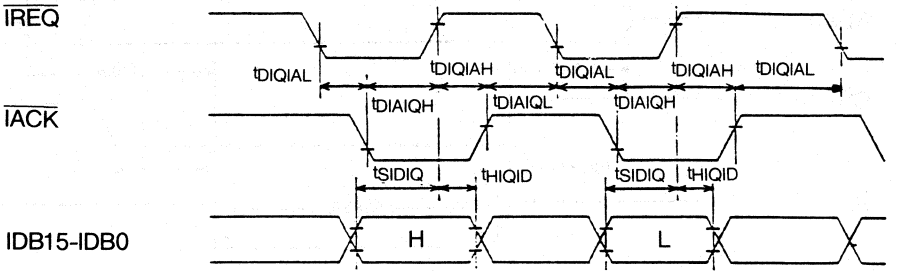
HOST CPU μPD9305 READ TIMING

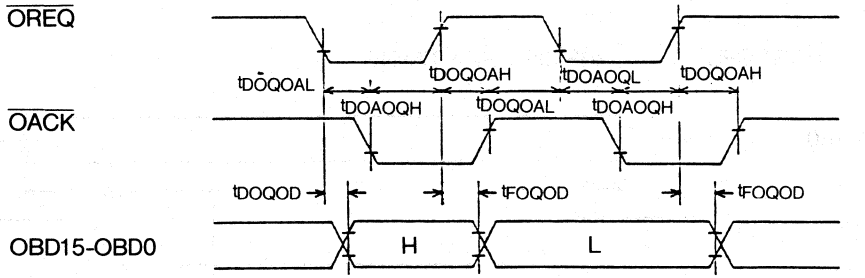


HOST CPU μPD9305 WRITE TIMING

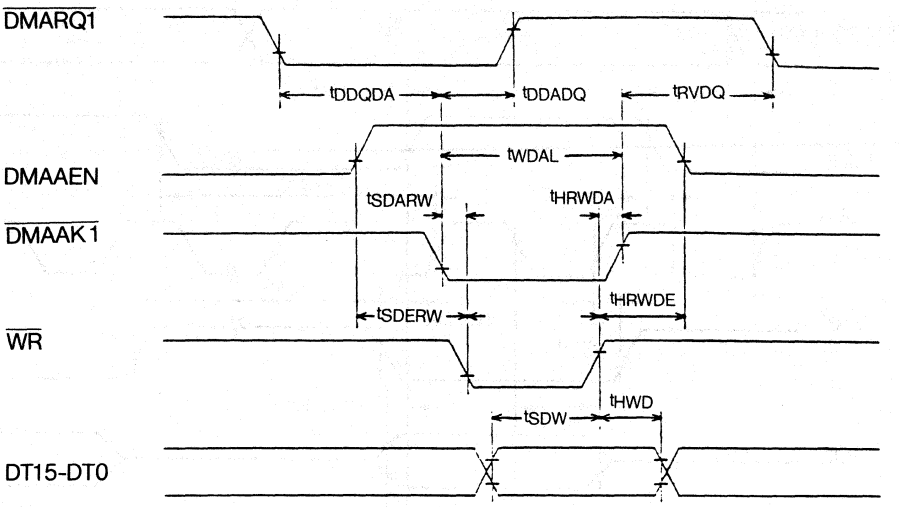


I/O DATA BUS HANDSHAKE TIMING





DMA1 TIMING



DMA2 TIMING

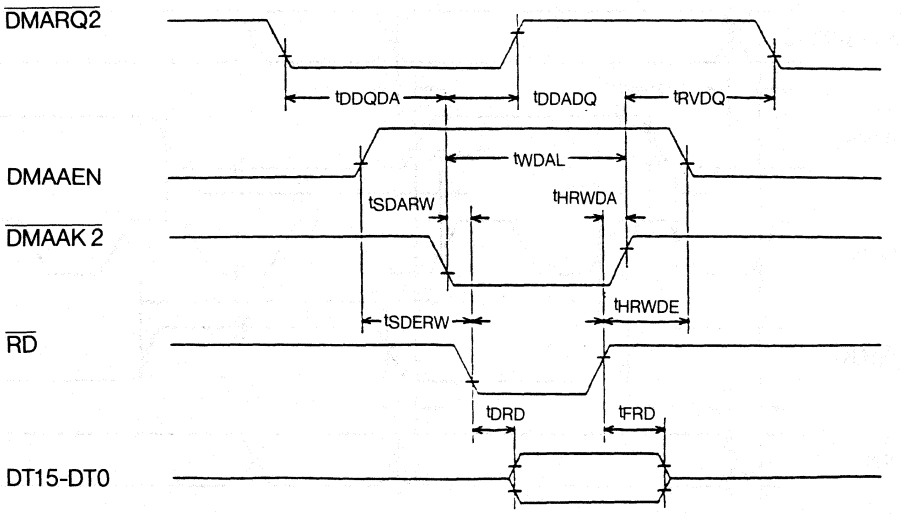


IMAGE MEMORY READ TIMING

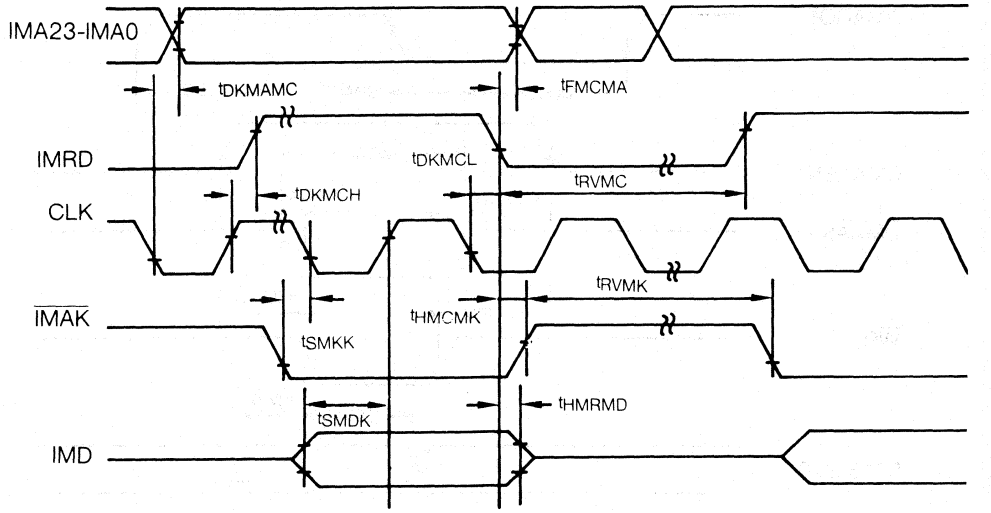


IMAGE MEMORY WRITE TIMING

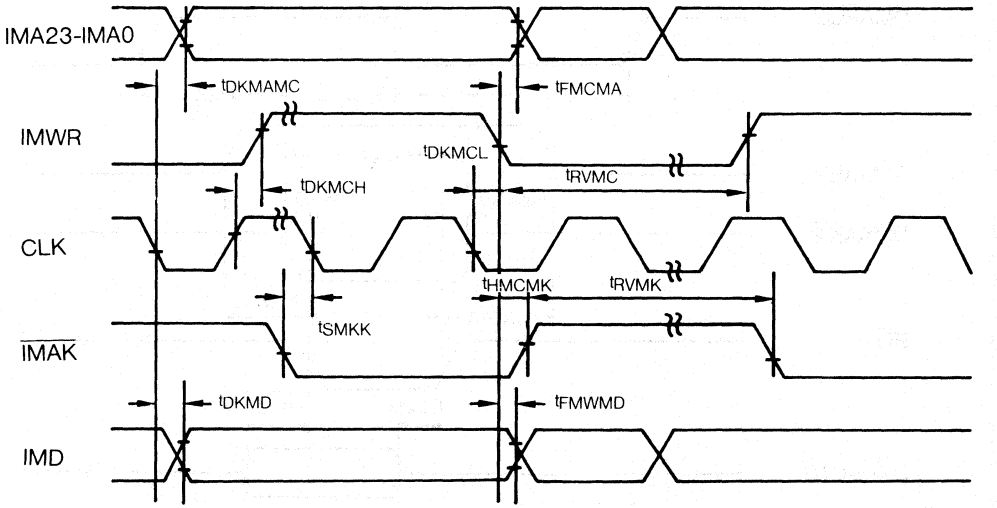
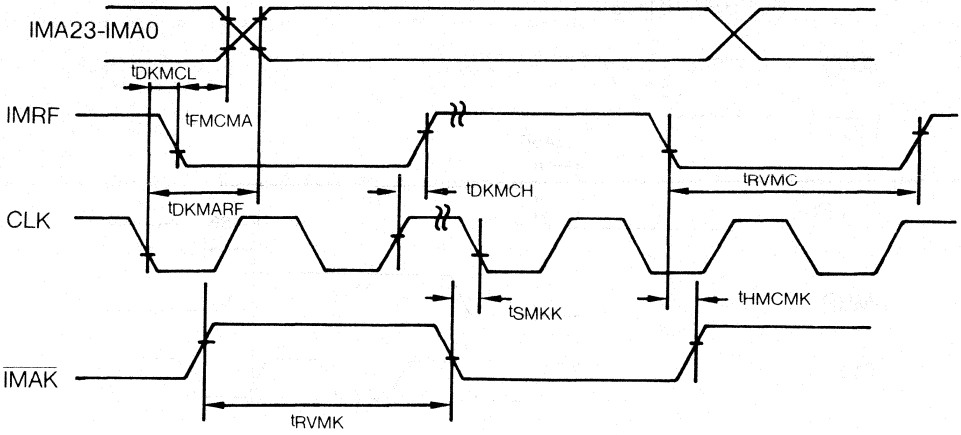
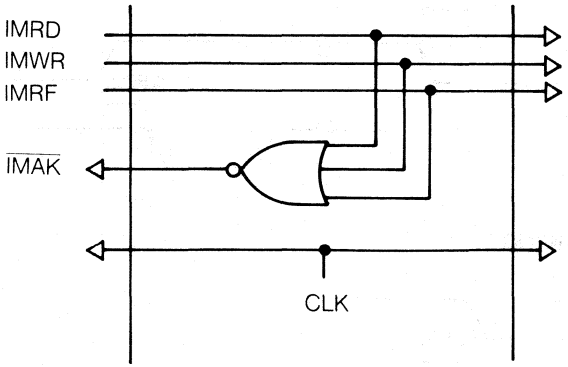


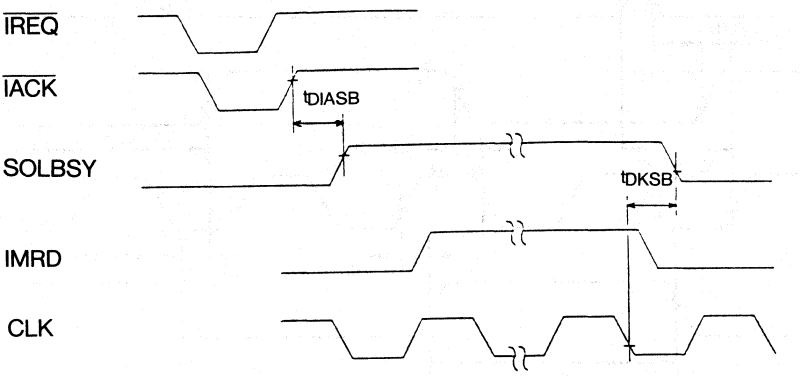
IMAGE MEMORY REFRESH TIMING



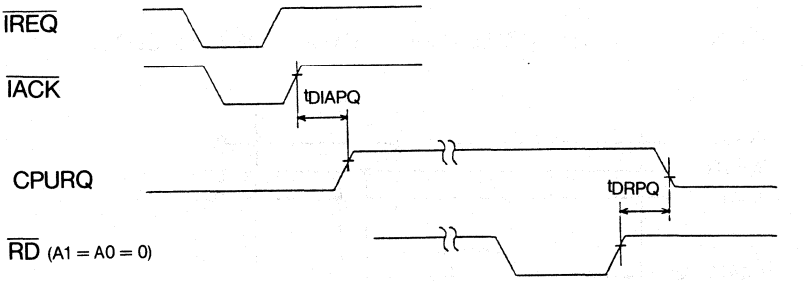
IMAK GENERATION EXAMPLE (CYCLETIME = 3 X CLOCK)



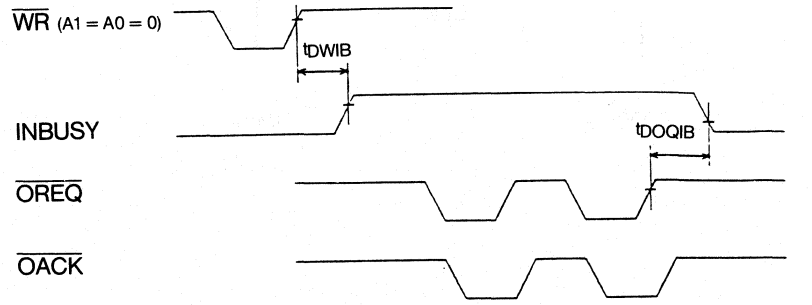
SOLBSY TIMING



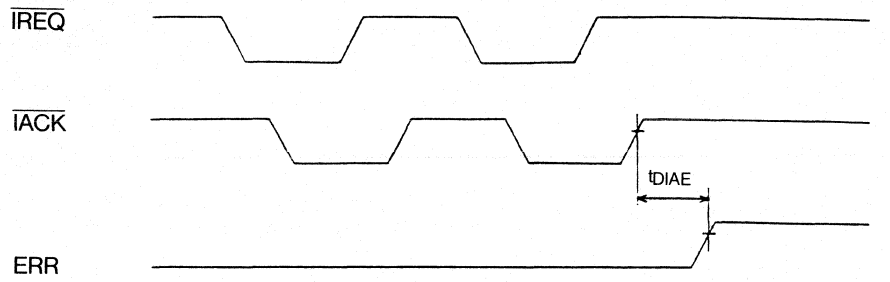
CPURQ TIMING



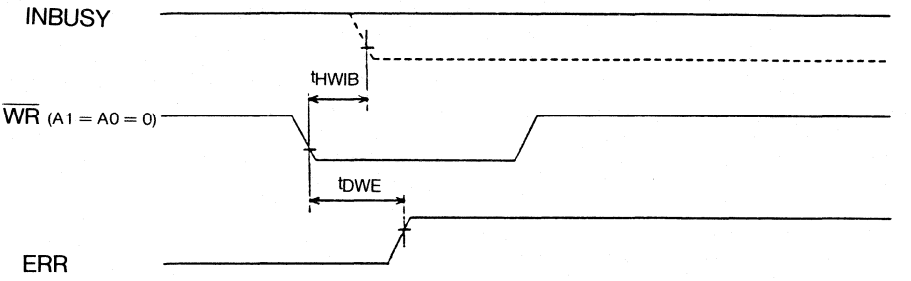
INBUSY TIMING



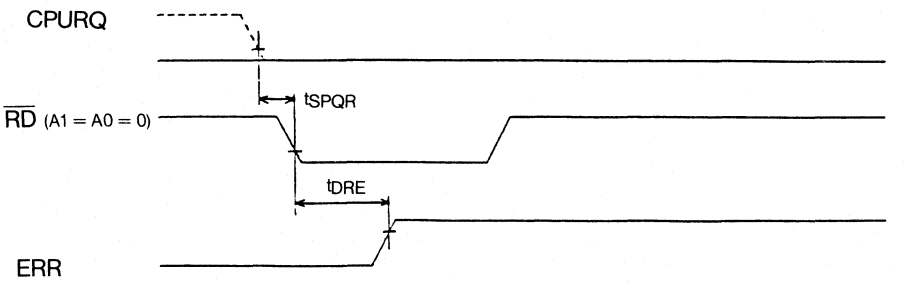
ERR TIMING (1)



ERR TIMING (2)



ERR TIMING (3)



C

ANμPD7281
APPLICATION LIBRARY FOR
IMAGE PIPELINED PROCESSOR

APPLICATION NOTE

The uPD7281 incorporates a configuration which is quite different from those of other microcomputers, and most readers may be unfamiliar with its program development facilities. The uPD7281 Application Library is a collection of programs that provide specific examples of programming on the uPD7281. The document is intended as a guide to learning uPD7281 programming methods and techniques. Three volumes are included:

I	Binary Image Processing	page 1-1
II	Gray-scale Image Processing	page 2-1
III	Numerical Calculations	page 3-1

This Application Library explains uPD7281 programming methods. The reader is advised to consult the following publications for information on the uPD7281 itself and the assembler:

(1) uPD7281 Product Description

(2) uPD7281 Software Package Operating Manual

Notice:

(1) The contents of this application library may not be copied wholly or in part.

(2) Information contained in this apllication library is subject to change without notice.

(3) This application library has been prepared with utmost care; however, if you have questions or find errors or omissions in it, please notify NEC Electronics. As the intention is to explain the uPD7281 programming methods, the examples provided have not fully been tested on actual systems, and no claim is made for their validity.

(4) Notwithstanding the statement in item (3) above, NEC Electronics is not responsible for any damage resulting from an execution of the contents of this document.

VOLUME I

BINARY IMAGE PROCESSING

Introduction

This Application Library (Volume I) addresses "Binary Image Processing," which is one of the application areas of the uPD7281. The applications presented in the following pages include:

* Block Transfer
  - Word Boundary Transfer
* Logical Operations
  - NOT (single-operand operations)
  - AND/OR/Exclusive OR (double-operand operations)
* Enlargement/Shrinking
  - Simple 1/2 Shrinking
  - 4-Point OR 1/2 Shrinking (Logical Addition)
  - Neighboring 16-Point Addition 1/4 Shrinking (the Majority Rule)
  - Simple Double Enlargement
  - Simple Quadruple Enlargement
* Affine Transformation
* Profiling
  - Horizontal
  - Vertical
* 3 x 3 Masking
  - Smoothing
  - Thinning
  - Edge Detection

Each example is discussed in terms of the following items:

(1) Processing
(2) Algorithm
(3) Parameters and their applicable ranges
(4) Flow graphs
(5) Tips on preparing flow graphs
(6) Assembler source listing

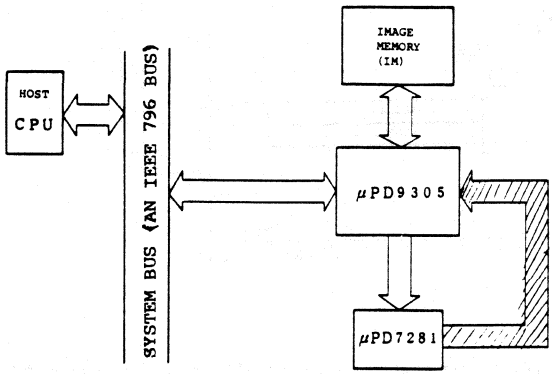
Chapter 1

System Configuration

The programs included in this Application Library have been written with the assumption that they will be executed on a system configured as shown in Figure 1-1. A characteristic of this system is the use of a uPD7281 peripheral LSI, the uPD9305. This means that a uPD9305-defined token is used in accessing the image memory (IM). The programs presented in this document are not directly applicable to those systems in which a uPD9305 is not employed.

Although the programs shown in this document are written for a single uPD7281, the uPD9305 permits the use of several uPD7281s. Therefore, it is possible to increase the processing speed by partitioning the memory area among several uPD7281s, each running the same program. In such a case, the input token into the uPD7281s should be set up in accordance with the memory area to which it is assigned.

Figure 1-1
System Configuration

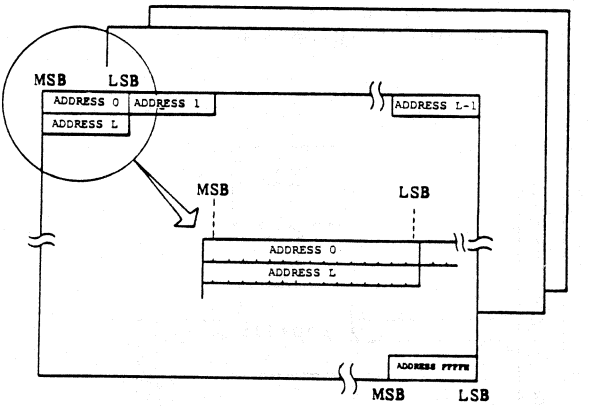


The following is a description of image memory configuration.

Shown in Figure 1-2 is an image memory configuration of the system addressed in this document. Viewing the memory in terms of screen display, the upper left corner of the screen corresponds to the most significant bit (MSB); this has the address of 0. The lower right corner the least significant bit (LSB); this has the address FFFFH*. Since there are 16 bits in a word, if there are L words in a horizontal line, the number of picture elements (pixels) present in that line is 16 x L dots.

In this system the addresses 0 through FFFFH comprise a screen image (one bank). However, since the 8 high order bits of the 24 image memory address bits of the uPD9305 are not set (i.e., neither the bank-defining read high address or write high address is set), it is necessary to set up a high address register in the uPD9305 to use multiple banks. For the same reason, the addressable memory space of these programs is addresses 0 through FFFFH, both for source and destination images.

Figure 1-2
Image Memory Configuration



* : This image memory configuration is different from that of the graphic display controllers uPD7220 and uPD7220A offered by NEC.

Chapter 2

Block Transfer

In this chapter we consider a program for transferring a specified rectangular area to another one. Although such a transfer can be accomplished with the same results by using a special case of the affine transformation (horizontal move), the method described below employs a different algorithm to reduce processing time. A discussion of word boundary transfer follows.

2.1 Word Boundary Transfer

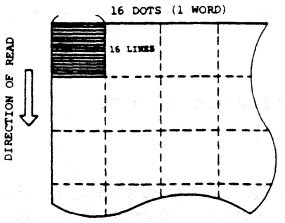
2.1.1 Processing Explained

A word boundary transfer is a block-oriented transfer in which a rectangular area defined by 16 horizontal dots (comprising a word specified by the image memory address) and 16 vertical dots are transferred as a block of data. Although the minimum size of the transfer block (16 x 16 dots) cannot be varied, the program permits variations in horizontal screen size L (number of words), and the number of horizontal blocks H and vertical blocks V to be transferred.

2.1.2 Algorithm

(1) First, the program and the constants are downloaded to the uPD7281. This is followed by the input from the host computer of two start-up tokens: the starting address (STARTS) of a source image area (SRC), and the starting address (STARTD) of the destination image area (DST).

Figure



(2) One block of data (SRC data) is read vertically, starting from the address indicated by STARTS.

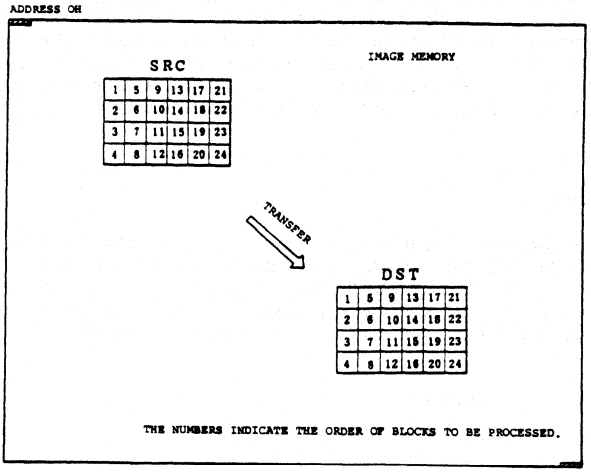
(3) Simultaneously, one block of memory addresses needed for the storage of SRC data is generated, starting at the address indicated by STARTD.

(4) The SRC data is then written into the DST of the image memory in accordance with the corresponding DST addresses.

(5) When all of V blocks have been transferred by repeating this operation vertically, the process is repeated in the horizontal direction by incrementing the horizontal address by 1.

(6) The transfer operation is terminated when step (5) is completed for all H blocks in the horizontal direction.

Figure 2-1
An Example of Data Transfer (for H=6, V=4)



2.1.3 Parameters and Their Applicable Ranges

<Assembler-coded parameters>

L ... Number of image memory words in horizontal direction
H ... Number of source image area words in horizontal direction

V ... Number of source image area blocks in vertical direction

<Start-up token-defined parameters>

STARTS: Starting address of the source image area
STARTD: Starting address of the destination image area

The allowable values of these parameters are indicated in the table below:

Parameter	Applicable range	(Value set in the example program)
L	0 - 65535	(64)
H	1 - 256	(32)
V	1 - 256	(32)
STARTS	0 - 65535	(0)*
STARTD	0 - 65535	(32)*

* : Although STARTS and STARTD are variables (i.e., addresses), they are given default values since they are used in the assembler DATA statement. When the uPD7281 is started up, the starting addresses of the SRC and DST areas are input as execution tokens.

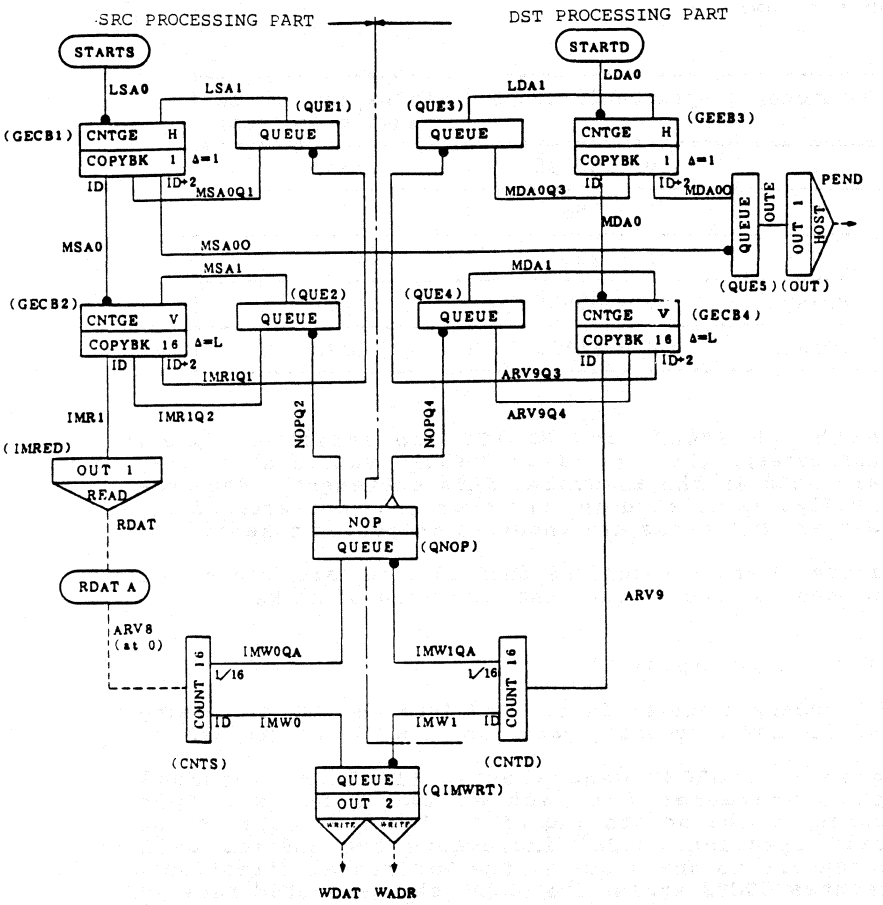
Furthermore, H and V should be defined with care, since this program does not accommodate the switching of banks.

2.1.4 Flow Graph Explained

A word boundary transfer is divided into the SRC processing part and the DST processing part, as shown in Figure 2-2.

In Figure 2-2, GECB2 generates 16 (1 block) vertical direction addresses for each of the V blocks. Upon completion of the processing of V blocks of data in the vertical direction, GECB1 increments the address by 1 (corresponding to one block in the horizontal direction), and executes GECB2 again. The GECB2 thus executed repeats the vertical direction processing in the same manner. This combination of vertical and horizontal processing creates a set of addresses comprising a rectangular area of H x V blocks.

Figure 2-2
Word Boundary Transfer Flow Graph



The image memory data (SRC data) indicated by these addresses are read on the SRC side and written to the output address generated on the DST side. Futher, since data are read on SRC and written to DST, SRC and DST are made to synchronize their actions so that the level of QIMWRT will not exceed 16.

<Explanation of Nodes>

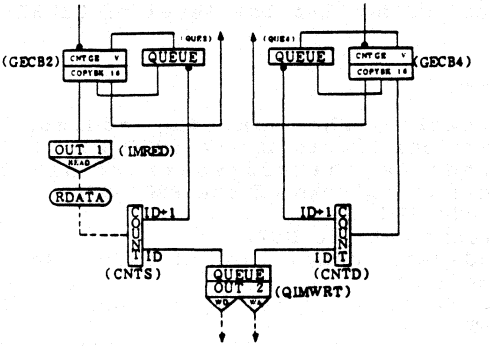
GECB1/GECB3	: Generate the starting address of V vertical blocks of the SRC/DST.
GECB2/GECB4	: Generate the addresses within the V vertical blocks of the SRC/DST.
IMRED	: Reads SRC data.
QIMWRT	: Writes data to DST.
OUT	: Indicates completion of data transfer to the host.
CNTS, CNTD, QNOP	: Synchronize processing between the SRC and DST.
QUE1/QUE3	: Synchronize the actions of GECB1 and GECB3 so that addresses are generated for V blocks at a time.
QUE2/QUE4	: Synchronize the actions of GEB2 and GEB4 so that addresses are generated for one block at a time.
QUE5	: Synchronizes SRC and DST to verify completion of their processing tasks.

2.1.5 Tips on Writing Flow Graphs

A program for word boundary data transfer is made up of an address generation part to read from the SRC side and an address generation part to write to the DST side. These parts are identical except for the portion that concerns the reading of SRC data.

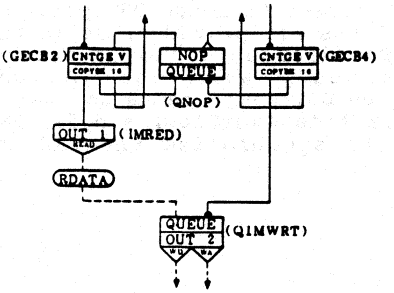
The SRC processing part generates addresses and reads SRC data from the image memory, whereas the DST processing part is involved only in the preparation of addresses. For this reason, the SRC part works slower than the DST part does. If a synchronizing node (QNOP) was not provided, as in the case of Figure 2-3(a), the DST addresses would be created one after another asynchronously with the SRC address generation and tokens sent by DST would cause an overflow in the QIMWRT node. To prevent this overflow, a node, QNOP, in Figure 2-2, is required to synchronize the SRC and DST parts.

Figure 2-3 (a)
Overflow Caused by Difference in Processing Speeds between SRC and DST



To synchronize these two processing parts, the scheme indicated in Figure 2-3 (b) might suffice and would also increase the processing speed. However, there are additional considerations: the amount of time required by the token issued to read the image memory and to make a return trip to the uPD7281, and situations where there may be some systems where the overflow occurs depending on the number of uPD7281s involved.

Figure 2-3 (b)
A Flow Graph in Which an Overflow May Occur Depending on the Time From When the Token is Issued to Read the Image Memory Until the Token Returns to the uPD7281



2.1.6 Assembler Source Listing

```
 1: ;****************************************
 2: ;
 3: ;       W O R D   B O U N D A R Y
 4: ;
 5: ;----------------------------------------
 6: ;
 7: MODULE  IPP     =       8               ;
 8: ;
 9: EQUATE  H       =       512/16          ;
10: EQUATE  V       =       512/16          ;
11: EQUATE  L       =       1024/16         ;
12: ;
13: EQUATE  HOST    =       0               ;
14: EQUATE  READ    =       4               ;
15: EQUATE  WRITE   =       5               ;
16: ;
17: EQUATE  STARTS  =       0               ;
18: EQUATE  STARTD  =       32              ;
19: ;
20: ;****************************************
21: ;
22: ;       INPUT-OUTPUT
23: ;
24: ;----------------------------------------
25: ;
26: INPUT   LSA0,   LDA0,   ARV8 AT 0       ;
27: ;
28: OUTPUT  RDAT,   WDAT,   WADR,   PEND    ;
29: ;
30: ;****************************************
31: ;
32: ;       LINK TABLE
33: ;
34: ;----------------------------------------
35: ;
36: LINK    MSA0,   MSA0Q1, MSA0O   =       GECB1   (LSA1,  LSA0    )       ;
37: LINK    LSA1                    =       QUE1    (MSA0Q1,IMR1Q1  )       ;
38: LINK    IMR1,   IMR1Q2, IMR1Q1  =       GECB2   (MSA1,  MSA0    )       ;
39: LINK    MSA1                    =       QUE2    (IMR1Q2,NOPQ2   )       ;
40: LINK    RDAT                    =       IMRED   (IMR1           )       ;
41: LINK    IMW0,   IMW0QA          =       CNTS    (ARV8           )       ;
42: LINK    MDA0,   MDA0Q3, MDA0O   =       GECB3   (LDA1,  LDA0    )       ;
43: LINK    LDA1                    =       QUE3    (MDA0Q3,ARV9Q3  )       ;
44: LINK    ARV9,   ARV9Q4, ARV9Q3  =       GECB4   (MDA1,  MDA0    )       ;
45: LINK    MDA1                    =       QUE4    (ARV9Q4,NOPQ4   )       ;
46: LINK    IMW1,   IMW1QA          =       CNTD    (ARV9           )       ;
47: LINK    NOPQ2,  NOPQ4           =       QNOP    (IMW0QA,IMW1QA  )       ;
48: LINK    WDAT,   WADR            =       QIMWRT  (IMW0,  IMW1    )       ;
49: LINK    OUTE                    =       QUE5    (MDA0O, MSA0O   )       ;
50: LINK    PEND                    =       OUT     (OUTE           )       ;
51: ;
```

```
52: ;*****************************************
53: ;
54: ;       FUNCTION TABLE
55: ;
56: ;-----------------------------------------
57: ;
58: FUNCTION   IMRED   =   OUT1    (READ.  0)                                  ;
59: FUNCTION   QIMWRT  =   OUT2    (WRITE. 20H. 0).QUEUE   (Q6.    16)        ;
60: FUNCTION   OUT     =   OUT1    (HOST.  0)                                  ;
61: FUNCTION   GECB1   =   COPYBK  (1.     1),     CNTGE   (H       )         ;
62: FUNCTION   GECB2   =   COPYBK  (16.    L),     CNTGE   (V       )         ;
63: FUNCTION   GECB3   =   COPYBK  (1.     1),     CNTGE   (H       )         ;
64: FUNCTION   GECB4   =   COPYBK  (16.    L),     CNTGE   (V       )         ;
65: FUNCTION   QUE1    =   QUEUE   (Q1.    1)                                  ;
66: FUNCTION   QUE2    =   QUEUE   (Q2.    1)                                  ;
67: FUNCTION   QUE3    =   QUEUE   (Q3.    1)                                  ;
68: FUNCTION   QUE4    =   QUEUE   (Q4.    1)                                  ;
69: FUNCTION   QUE5    =   QUEUE   (Q5.    1)                                  ;
70: FUNCTION   CNTS    =   COUNT   (16      )                                  ;
71: FUNCTION   CNTD    =   COUNT   (16      )                                  ;
72: FUNCTION   QNOP    =   NOP     (XY      ),     QUEUE   (QA.    1)         ;
73: ;
74: ;*****************************************
75: ;
76: ;       DATA MEMORY
77: ;
78: ;-----------------------------------------
79: ;
80: MEMORY  Q1      =       AREA    (1       )      ;
81: MEMORY  Q2      =       AREA    (1       )      ;
82: MEMORY  Q3      =       AREA    (1       )      ;
83: MEMORY  Q4      =       AREA    (1       )      ;
84: MEMORY  Q5      =       AREA    (1       )      ;
85: MEMORY  Q6      =       AREA    (16      )      ;
86: MEMORY  QA      =       AREA    (1       )      ;
87: ;
88: ;*****************************************
89: ;
90: ;       START
91: ;
92: ;-----------------------------------------
93: ;
94: START                                          ;
95: ;
96: DATA    EXEC    (IPP.   LSA0.   STARTS  )     ;
97: DATA    EXEC    (IPP.   LDA0.   STARTD  )     ;
98: ;
99: END                                            ;
```

Chapter 3

Logical Operations

3.1 NOT (Single-Operand Operation)

3.1.1 Processing Explained

The NOT operation is used in writing data, read from the image memory source area (SRC), to the destination area (DST) by performing bit inversions.

3.1.2 Algorithm

First the SRC starting address (STARTS) and the DST starting address (STARTD) are received from the host computer as input parameters, as occurs in a word boundary transfer. Then, based on STARTS, data is read from SRC, block by block, in the vertical direction. The data is then NOT-operated, and the inverted data is written to the DST after being combined with its corresponding DST addresses (as in the case of a word boundary transfer).

Details are provided in Section 2.1, "Word Boundary Transfer."

3.1.3 Parameters and Their Applicable Ranges

<Assembler-Coded Parameters>

L ... Number of image memory words in horizontal direction
H ... Number of source image area words in horizontal direction
V ... Number of destination image area blocks in vertical direction

<Start-up token-defined parameters>

STARTS ... Source image area (SRC) starting address
STARTD ... Destination image area (DST) starting address

The values that can be assigned to these parameters are as follows:

Parameter	Applicable range	(Value set in the example program)
L	0 - 65535	(64)
H	1 - 256	(32)
V	1 - 256	(32)
STARTS	0 - 65535	(0)*
STARTD	0 - 65535	(32)*

* : Although STARTS and STARTD are variables (i.e., addresses), they are given default values since they are used in the assembler DATA statement. When the uPD7281 is started up, the starting addresses of the SRC and DST areas are input as execution tokens.

Since this program has no provision for bank switching, care should be exercised in setting the values for the parameters H and L.

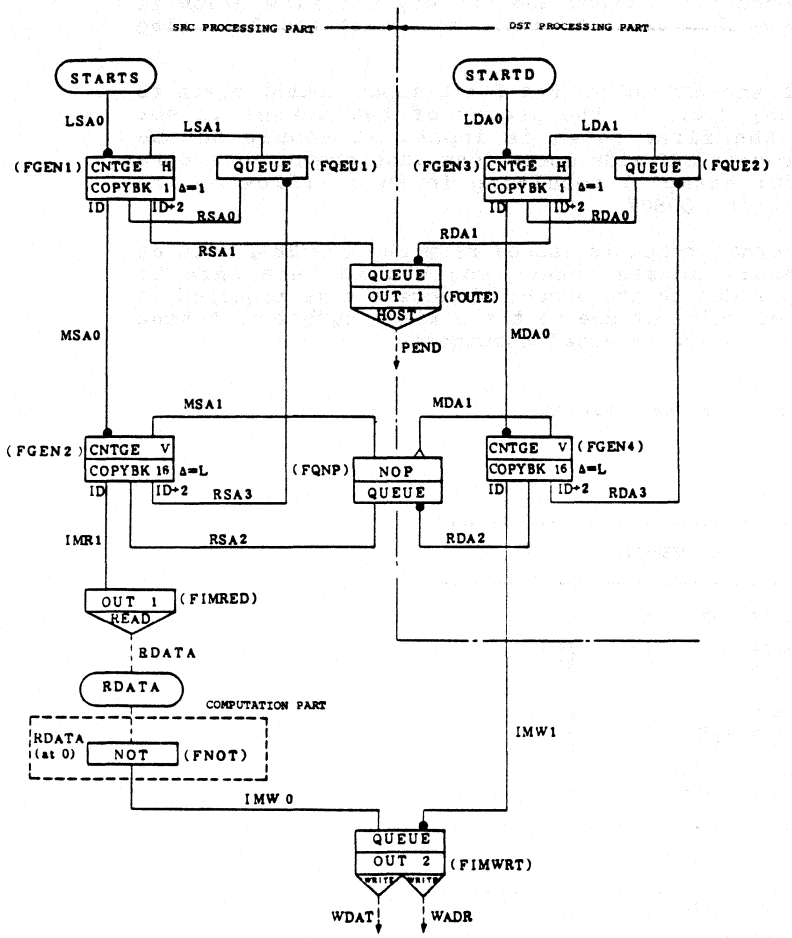
<Initial Values>

Initial values used in this program are the same as those used in the word boundary transfer program. (See 2.1.3).

3.1.4 Flow Graph Explained

The flow graph for the NOT operation, shown in Figure 3-1, represents the addition of SRC data NOT processing to the word boundary transfer flow graph given in Figure 2-1, except that the node for the COUNT instruction to ensure synchronization during a word boundary transfer has been removed. The method employed in the NOT operation for synchronizing SRC and DST processing is not an ideal one because there is a risk of an overflow occuring, depending on the reading time of SRC data. However, for the typical system configuration used in this example and for those cases where image memory access time is short, the synchronization method illustrated in the flow graph should be adequate and should yield improvements in processing speed.

Figure 3.1
A NOT (Single-Operand Operation) Flow Graph



3.1.5 Tips on Writing Flow Graphs

The SRC and DST processing must be synchronized for logical operations and transfer. In the absence of synchronization, some programs can run into a situation where an excess of data for either SRC processing or DST processing is created, resulting in a uPD7281 QUEUE overflow. The methods used in ensuring synchronization can vary, depending on the particular objective pursued and the way the flow graph is written. Any method used, however, must meet the following conditions:

- the work of the SRC token (the first input token) needs to be controlled; i.e., if the second of two tokens is not input after the first token is input, it should not be allowed to process SRC side operations independent of another side (DST, for example) resulting in an overflow on the image memory write QUEUE.

- when an external token is issued from the uPD7281, such as the image memory access token, you should take care to preclude errors due to the amount of access time required in the external circuits or due to the maximum number of tokens flowing around in the external components.

3.1.6 Assembler Source Listing

```
 1: ;****************************************
 2: ;
 3: ;       NOT OPERATION
 4: ;
 5: ;----------------------------------------
 6: ;
 7: MODULE  IPP     =       8                   ;
 8: ;
 9: EQUATE  L       =       64                  ;
10: EQUATE  H       =       32                  ;
11: EQUATE  V       =       32                  ;
12: ;
13: EQUATE  HOST    =       0                   ;
14: EQUATE  READ    =       4                   ;
15: EQUATE  WRITE   =       5                   ;
16: ;
17: EQUATE  START3  =       0                   ;
18: EQUATE  STARTD  =       32                  ;
19: ;
20: ;****************************************
21: ;
22: ;       INPUT-OUTPUT
23: ;
24: ;----------------------------------------
25: INPUT   LSA0.   LDA0.   RDATA AT 0          ;
26: ;
27: OUTPUT  RDAT.   WDAT.   WADR.   PEND        ;
28: ;
```

```
;****************************************
;
;       LINK TABLE
;
;----------------------------------------
;
LINK    MSA0,   RSA0,   RSA1    =       FGEN1   (LSA1,  LSA0    )       ;
LINK    IMR1,   RSA2,   RSA3    =       FGEN2   (MSA1,  MSA0    )       ;
LINK    LSA1                    =       FQUE1   (RSA0,  RSA3    )       ;
LINK    PEND                    =       FOUTE   (RSA1,  RDA1    )       ;
LINK    RDAT                    =       FIMRED  (IMR1           )       ;
LINK    MSA1,   MDA1            =       FQNP    (RSA2,  RDA2    )       ;
LINK    IMW0                    =       FNOT    (RDATA          )       ;
LINK    MDA0,   RDA0,   RDA1    =       FGEN3   (LDA1,  LDA0    )       ;
LINK    IMW1,   RDA2,   RDA3    =       FGEN4   (MDA1,  MDA0    )       ;
LINK    LDA1                    =       FQUE2   (RDA0,  RDA3    )       ;
LINK    WDAT,   WADR            =       FIMWRT  (IMW0,  IMW1    )       ;
;
;****************************************
;
;       FUNCTION TABLE
;
;----------------------------------------
;
FUNCTION    FIMRED  =   OUT1    (READ,  0)                              ;
FUNCTION    FIMWRT  =   OUT2    (WRTIE, 20H, 0),QUEUE   (QUEW,  16)     ;
FUNCTION    FOUTE   =   OUT1    (HOST,  0),     QUEUE   (QUEE,  1)      ;
FUNCTION    FGEN1   =   COPYBK  (1,     1),     CNTGE   (H      )       ;
FUNCTION    FGEN2   =   COPYBK  (16,    L),     CNTGE   (V      )       ;
FUNCTION    FGEN3   =   COPYBK  (1,     1),     CNTGE   (H      )       ;
FUNCTION    FGEN4   =   COPYBK  (16,    L),     CNTGE   (V      )       ;
FUNCTION    FQNP    =   NOP     (XY     ),      QUEUE   (QUEN,  1)      ;
FUNCTION    FNOT    =   NOT     (X      )
FUNCTION    FQUE1   =   QUEUE   (QUE1,  1)
FUNCTION    FQUE2   =   QUEUE   (QUE2,  1)
;
;****************************************
;
;       DATA MEMORY
;
;----------------------------------------
;
MEMORY  QUE1    =       AREA    (1      )       ;
MEMORY  QUE2    =       AREA    (1      )       ;
MEMORY  QUEN    =       AREA    (1      )       ;
MEMORY  QUEW    =       AREA    (16     )       ;
MEMORY  QUEE    =       AREA    (1      )       ;
;
;****************************************
;
;       START
;
;----------------------------------------
;
START                                           ;
;
DATA    EXEC    (IPP,   LSA0,   STARTS  )       ;
DATA    EXEC    (IPP,   LDA0,   STARTD  )       ;
;
END                                             ;
```

3.2 AND, OR, Exclusive OR (Double-Operand Operations)

3.2.1 Processing Explained

Given SRC1 and SRC2 as source image areas, these operations perform the specified operation (AND, OR, or Exclusive OR) between the data in these areas and outputs the resultant data to the destination image area. As in the case of the word boundary data transfer (discussed in 2.1), each of the areas, SRC1, SRC2, and DST, in this program are divided into word-aligned horizontal 16-dot (1 word) and vertical 16-dot (16 lines) rectangular areas (blocks) as units of data transfer. For the order of processing, refer to Section 2.1, "Word Boundary Transfer."

3.2.2 Algorithm

The starting read addresses S1ADR and S2ADR of source image areas SRC1 and SRC2, respectively, and the starting address DSTADR of destination image area DST are input from the host computer into the uPD7281 as part of a token. Then, one block of addresses are generated for each S1ADR and S2ADR, and the contents of the image memory at those addresses are read. Data in the two blocks just read are operated upon (AND, OR, or Exclusive OR) in conjunction with their corresponding data, resulting in the creation of DST data. The DST data are written out to the DST addresses generated on the basis of DSTADR.

3.2.3 Parameters and Their Applicable Ranges

Assembler-coded parameters:
L ... Number of image memory words in horizontal direction
H ... Number of source image area words in horizontal direction
V ... Number of source image area blocks in vertical direction

Start-up token-defined parameters:
S1ADR ... First operand source image area (SRC1) starting address
S2ADR ... Second operand source image area (SRC2) starting address
DSTADR .. Destination image area (DST) starting address

The allowable values of these parameters are as follows:

Parameter	Applicable range	(Value set in the example program)
L	0 - 65535	(64)
H	1 - 256	(16)
V	1 - 256	(16)
S1ADR	0 - 65535	(0)*
S2ADR	0 - 65535	(4000H)*
DSTADR	0 - 65535	(0)*

* : Although STARTS and STARTD are variables (i.e., addresses), they are given default values since they are used in the assembler DATA statement. When the uPD7281 is started up, the starting addresses of the SRC and DST areas are input as execution tokens.

Since no provision is made in this program for switching banks, you should exercise care in setting values for parameters H and V.

<Initial Values>

Most of the initial values used in this program are the same as those used in the word boundary transfer program, except that initial values need to be assigned to the two SRC addresses.

3.2.4 Flow Graph Explained

This program illustrates the use of the OR operation. To use AND or Exclusive OR, change "OR" to "AND" or "XOR" (XOR is the instruction code for the Exclusive OR operation) in the FQOR node in Figure 3-2.

<Explanation of Nodes>

FGEN1/FGEN3 : Creates H blocks of horizontal starting addresses on the basis of the starting addresses S1ADR/S2ADR for SRC1/SRC2.

FGEN2/FGEN4 : Creates V blocks of in-block addresses on the basis of addresses sent from FGEN1/FGEN3.

FIMRD1/FIMRD2 : Reads the contents of SRC1/SRC2 on the basis of addresses created in FGEN2/FGEN4.

FQOR : Creates DST data by performing the OR

between the data read in FIMRD1 and FIMRD2.

FGEN5 : Creates H blocks of DST horizontal starting addresses on the basis of the DST starting address DSTADR

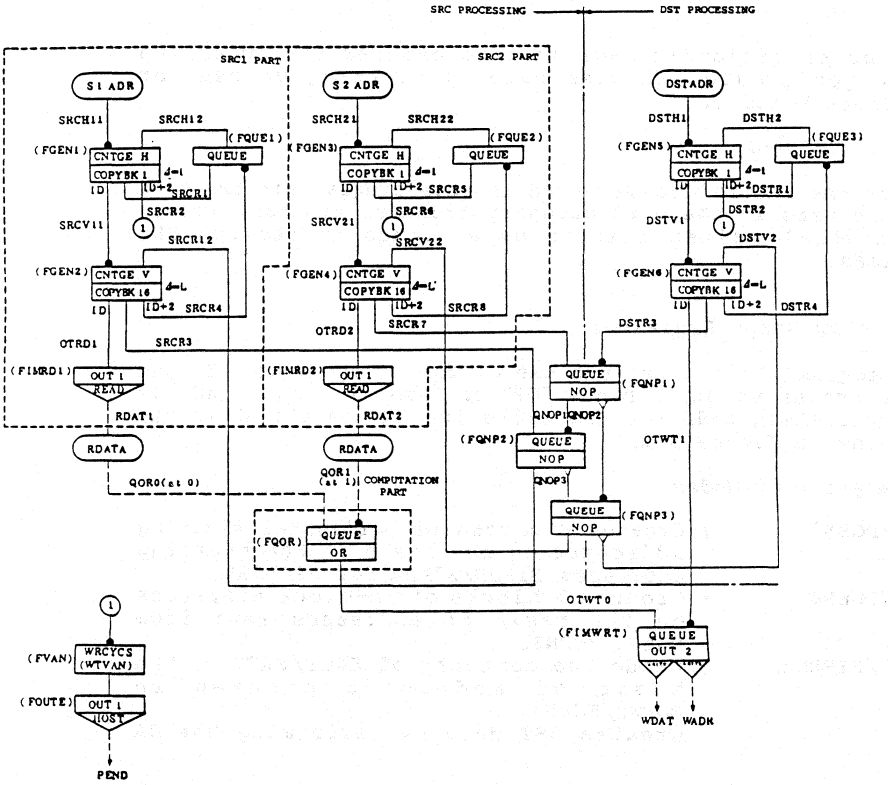
FGEN6 : Creates V blocks of in-block DST addresses on the basis of addresses generated in FGEN5.

FIMWRT : Writes DST data created in FQOR to the DST addresses generated in FGEN6.

FQUE1/FQUE2/FQUE3 : Synchronizes the actions of FGEN1/FGEN3/FGEN5 so that they make V blocks worth of addresses at a time.

FQNP1/FQNP2/FQNP3 : Synchronizes the actions of SRC and DST processing whenever one block of addresses are created in FGEN2, FGEN4, or FGEN6.

Figure 3-2
A Double-Operand Operation Flow Graph (OR)



3.2.5 Tips on Writing Flow Graphs

This program is basically the same as the word boundary transfer program in 2.1, except for minor differences that exist in the synchronization method because of the need to access three areas.

3.2.6 Assembler Source Listing

```
  1: ;****************************************
  2: ;
  3: ;       LOGICAL OPERATION
  4: ;
  5: ;----------------------------------------
  6: ;
  7: MODULE  IPP     =       8               ;
  8: ;
  9: EQUATE  L       =       64              ;
 10: EQUATE  H       =       16              ;
 11: EQUATE  V       =       16              ;
 12: ;
 13: EQUATE  HOST    =       0               ;
 14: EQUATE  READ    =       4               ;
 15: EQUATE  WRITE   =       5               ;
 16: ;
 17: EQUATE  S1ADR   =       0               ;
 18: EQUATE  S2ADR   =       256*L           ;
 19: EQUATE  DSTADR  =       0               ;
 20: ;
 21: ;
 22: ;****************************************
 23: ;
 24: ;       INPUT-OUTPUT
 25: ;
 26: ;----------------------------------------
 27: ;
 28: INPUT   SRCH11, SRCH21, DSTH1                    ;
 29: INPUT   QOR0 AT 0                               ;
 30: INPUT   QOR1 AT 1                               ;
 31: ;
 32: OUTPUT  RDAT1,  RDAT2,  WDAT,   WADR,   PEND    ;
 33: ;
 34: ;****************************************
 35: ;
 36: ;       LINK TABLE
 37: ;
 38: ;----------------------------------------
 39: ;
 40: LINK    SRCV11, SRCR1,  SRCR2   =       FGEN1   (SRCH12,SRCH11  )      ;
 41: LINK    OTRD1,  SRCR3,  SRCR4   =       FGEN2   (SRCV12,SRCV11  )      ;
 42: LINK    SRCH12                  =       FQUE1   (SRCR1, SRCR4   )      ;
 43: LINK    OUTH                    =       FVAN    (       ,SRCR2  )      ;
 44: LINK    RDAT1                   =       FIMRD1  (OTRD1          )      ;
 45: LINK    SRCV12, QNOP3           =       FQNP2   (SRCR3, QNOP1   )      ;
 46: LINK    SRCV21, SRCR5,  SRCR6   =       FGEN3   (SRCH22,SRCH21  )      ;
 47: LINK    OTRD2,  SRCR7,  SRCR8   =       FGEN4   (SRCV22,SRCV21  )      ;
 48: LINK    SRCH22                  =       FQUE2   (SRCR5, SRCR8   )      ;
 49: LINK    OUTH                    =       FVAN    (       ,SRCR6  )      ;
 50: LINK    RDAT2                   =       FIMRD2  (OTRD2          )      ;
 51: LINK    QNOP1,  QNOP2           =       FQNP1   (SRCR7, DSTR3   )      ;
 52: LINK    OTWT0                   =       FQOR    (QOR0,  QOR1    )      ;
 53: LINK    SRCV22, DSTV2           =       FQNP3   (QNOP3, QNOP2   )      ;
 54: LINK    DSTV1,  DSTR1,  DSTR2   =       FGEN5   (DSTH2, DSTH1   )      ;
 55: LINK    OTWT1,  DSTR3,  DSTR4   =       FGEN6   (DSTV2, DSTV1   )      ;
 56: LINK    DSTH2                   =       FQUE3   (DSTR1, DSTR4   )      ;
 57: LINK    OUTH                    =       FVAN    (       ,DSTR2  )      ;
 58: LINK    WDAT,   WADR            =       FIMWRT  (OTWT0, OTWT1   )      ;
 59: LINK    PEND                    =       FOUTE   (OUTH           )      ;
 60: ;
```

```
;*****************************************
;
;       FUNCTION TABLE
;
;-----------------------------------------
;
FUNCTION    FIMRD1  =   OUT1    (READ,    0)                                  ;
FUNCTION    FIMRD2  =   OUT1    (READ,    1)                                  ;
FUNCTION    FIMWRT  =   OUT2    (WRTIE, 20H, 0),QUEUE   (QUEW,  16)           ;
FUNCTION    FOUTE   =   OUT1    (HOST,    0)                                  ;
FUNCTION    FGEN1   =   COPYBK  (1,       1),   CNTGE   (H       )           ;
FUNCTION    FGEN2   =   COPYBK  (16,      L),   CNTGE   (V       )           ;
FUNCTION    FGEN3   =   COPYBK  (1,       1),   CNTGE   (H       )           ;
FUNCTION    FGEN4   =   COPYBK  (16,      L),   CNTGE   (V       )           ;
FUNCTION    FGEN5   =   COPYBK  (1,       1),   CNTGE   (H       )           ;
FUNCTION    FGEN6   =   COPYBK  (16,      L),   CNTGE   (V       )           ;
FUNCTION    FQOR    =   OR      (X         ),   QUEUE   (QUEOR, 16)           ;
FUNCTION    FQNP1   =   NOP     (XY        ),   QUEUE   (QUEN1, 1)            ;
FUNCTION    FQNP2   =   NOP     (XY        ),   QUEUE   (QUEN2, 1)            ;
FUNCTION    FQNP3   =   NOP     (XY        ),   QUEUE   (QUEN3, 1)            ;
FUNCTION    FQUE1   =   QUEUE   (QUE1,    1)                                  ;
FUNCTION    FQUE2   =   QUEUE   (QUE2,    1)                                  ;
FUNCTION    FQUE3   =   QUEUE   (QUE3,    1)                                  ;
FUNCTION    FVAN    =   WRCYCS  (WTVAN,   3)                                  ;
;
;*****************************************
;
;       DATA MEMORY
;
;-----------------------------------------
;
MEMORY  QUE1    =       AREA    (1      )       ;
MEMORY  QUE2    =       AREA    (1      )       ;
MEMORY  QUE3    =       AREA    (1      )       ;
MEMORY  QUEN1   =       AREA    (1      )       ;
MEMORY  QUEN2   =       AREA    (1      )       ;
MEMORY  QUEN3   =       AREA    (1      )       ;
MEMORY  QUEOR   =       AREA    (16     )       ;
MEMORY  QUEW    =       AREA    (16     )       ;
MEMORY  WTVAN   =       AREA    (3      )       ;
;
;*****************************************
;
;       START
;
;-----------------------------------------
;
START                                           ;
;
DATA    EXEC    (IPP,   SRCH11, S1ADR   )       ;
DATA    EXEC    (IPP,   SRCH21, S2ADR   )       ;
DATA    EXEC    (IPP,   DSTH1,  DSTADR  )       ;
;
END                                             ;
```

Chapter 4

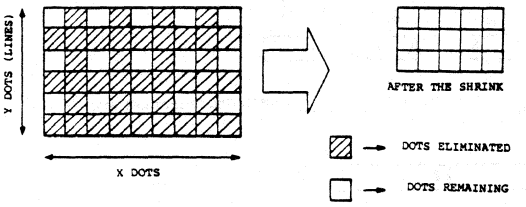
Enlargement and Shrinking

4.1 Simple One-Half Shrinking

4.1.1 Processing Explained

Simple one-half shrinking means a shrinking by simple elimination of source image area (SRC) data, as shown in the figure below.

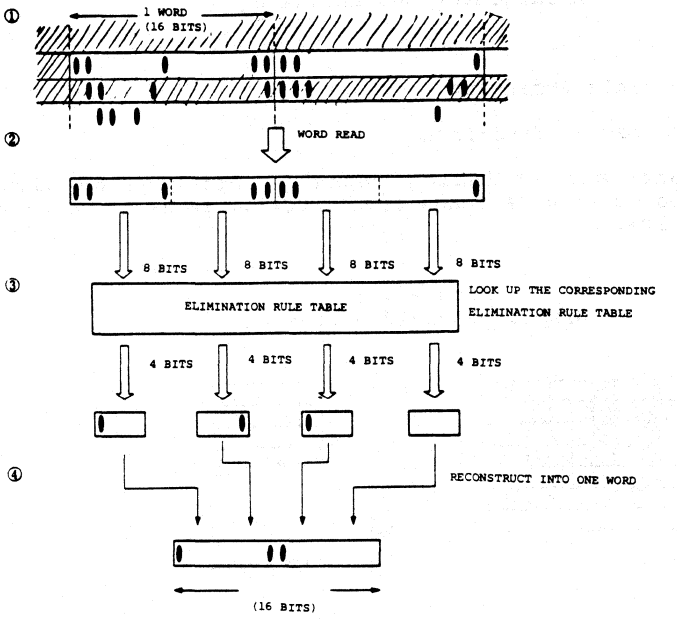
Figure 4-1
Simple One-Half Shrinking



4.1.2 Algorithm

Since all that is involved is a simple elimination, data are read vertically by skipping every other line. Horizontally, 16-bit data that are read are subjected to elimination by looking up an elimination rule table. To avoid the problem of handling a large 16-bit table, the data are divided into high-order and low-order 8-bit segments. A 256-word (8-bit address) elimination rule table is provided for each of these segments. The use of the 256-word table results in 4-bit data. Four pieces of such 4-bit data are written to the destination image area (DST) as one word (See Figure 4-2).

Figure 4-2
Algorithm for Simple One-Half Shrinking



4.1.3 Parameters and Their Applicable Ranges

<Assembler-coded parameters>

L ... Number of image memory words in horizontal direction
H ... Number of destination image area (DST) words in horizontal direction
M ... Number of destination image area (DST) lines in vertical direction

<Start-up token-defined parameters>

STARTS ... Source image area (SRC) starting address
STARTD ... Destination image area (DST) starting address

The allowable values of these parameters are indicated in the table below.

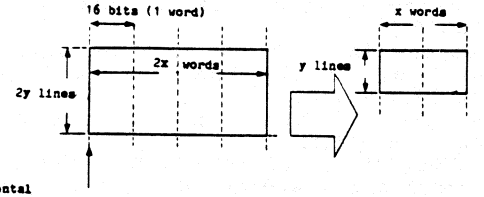
Parameter	Applicable range	(Value set in the example program)
L	0 - 32767	(64)
H	1 - 16	(16)
M	1 - 256	(256)
STARTS	0 - 65535	(0)*
STARTD	0 - 65535	(32)*

* : Although STARTS and STARTD are variables (i.e., addresses), they are given default values since they are used in the assembler DATA statement. When the uPD7281 is started up, the starting addresses of the SRC and DST areas are input as execution tokens.

Since no provision is made in this program for switching banks, you should exercise care in setting the values of parameters H and M. The maximum allowable values of these parameters in this program are H=16 and M=256; in other words, the maximum size of a destination image that can be reduced is 16 words horizontally and 256 lines vertically.

<Initial Values>

In setting initial values, the horizontal size of a SRC to be shrunk is defined in an even number of words, and the vertical size is defined in an even number of lines. Since the process involved is a simple one-half shrinking, once the size on the SRC side is determined, the size of the reduced image on the DST side is determined automatically.



The units in the horizontal direction must always be aligned with word boundaries.

Notice that, because DSTs are written word by word, the horizontal and vertical SRC definition numbers must always be multiples of 2.

4.1.4 Flow Graph Explained

This program consists of a SRC processing part (which reads every other line of the SRC data and creates shrunken data), and a DST part (which creates addresses necessary for writing the shrunk data). Shrunk data is generated on the basis of words of data at two contiguous addresses on the SRC side (the first word corresponds with the high-order 8 bits, and the second word with the low-order 8 bits). The SRC part receives from the host computer an SRC address which indicates the starting address of the shrinking processing, reads the SRC data horizontally every other line starting with the given SRC address, and separates them into high-order 8-bit and low-order 8-bit segments.

Each of the high-order 8-bit and low-order 8-bit segments is used as an address for a look-up table to generate high-order 4-bit and low-order 4-bit data generations. Four pieces of the 4-bit data thus created by two SRC data are placed in their respective bit positions to constitute DST (shrunk) data. That is, one DST data is created by two SRC data.

The DST processing part receives the starting address for writing the shrunk data from the host computer and creates write addresses on the basis of this address.

<Explanation of Nodes>

FNOP0	:	Synchronizes the actions of start-up tokens received from the host computer and indicating SRC (STARTS) and DST (STARTD) addresses, and sends them to the SRC and DST processing parts.
FGEN1	:	Creates line starting addresses for every other line in order to read SRC data. (Creates vertical addresses for every other line on the basis of the SRC address sent from FNOP0.)
FGEN2	:	Creates one line of SRC data addresses in two passes, on the basis of addresses generated in FGEN1.
FOUTR	:	Reads the contents of SRC addresses generated in FGEN2.
FARV1	:	Sends SRC data read in FOUTR alternately (high-order and low-order 8 bits) to the DST data generation node.
FCOP1/FCOP2	:	In the generation of high-order and low-order 8-bit data from the 16-bit data separated by FARV1, makes two copies of the data so that the high-order 4 bits of DST

data can be created from the high-order 8 bits of 16-bit SRC data, and the low-order 4 bits of DST data from the low-order 8 bits of the SRC data.

FRED1/FRED3 : Performs a right 8-bit shift to enable FTRN1/FTRN3 to look up the elimination rule data table on the basis of the high-order 8 bits of 16-bit SRC data.

FTRN1/FTRN3 : Looks up the elimination rule data table on the basis of the high-order 8 bits of 16-bit SRC data, and creates high-order 4-bit data for the high-order and low-order 8-bit DST data.

FTRN2/FTRN4 : Looks up the elimination rule table on the basis of the low-order 8 bits of a second copy of SRC data made in FCOP1/FCOP2, creates 4-bit elimination data for the low-order 8 bits, and assigns them as low-order 4-bit portions of the high-order and low-order 8-bit DST data. (4-bit data created in FTRN4 becomes the low-order 4-bit portion of the low-order 8-bit portion of the DST data.)

FRED12/FRED08 : Performs a 12/8 left shift on the 4-bit data created in FTRN1/FTRN2 and makes them into the high-order and low-order 4 bits of the high-order 8-bit DST data.

FRED04 : Performs a 4-bit left shift on the 4-bit data created in FTRN3 and makes it into high-order 4 bits of the low-order 8 bits of DST data.

FADD1/FADD2 : Adds the data created in FRED12/FRED04 and FRED08/FTRN4 together and creates high-order and low-order 8 bits of DST data.

FADD3 : Adds the data created in FADD1 and FADD2 together and makes them into DST data.

FARV2 : To enable FGEN2 to create SRC readout addresses each time 8 DST data are generated, notifies FQUE1 that the eighth data has been copied and that 8 DST data have been generated.

FGEN3 : Creates M lines of addresses for H horizontal words starting from the DST starting address STARTD sent from FNOP0.

FOUTW : Writes the reduced data generated by the SRC processing into the DST address created in FGEN3.

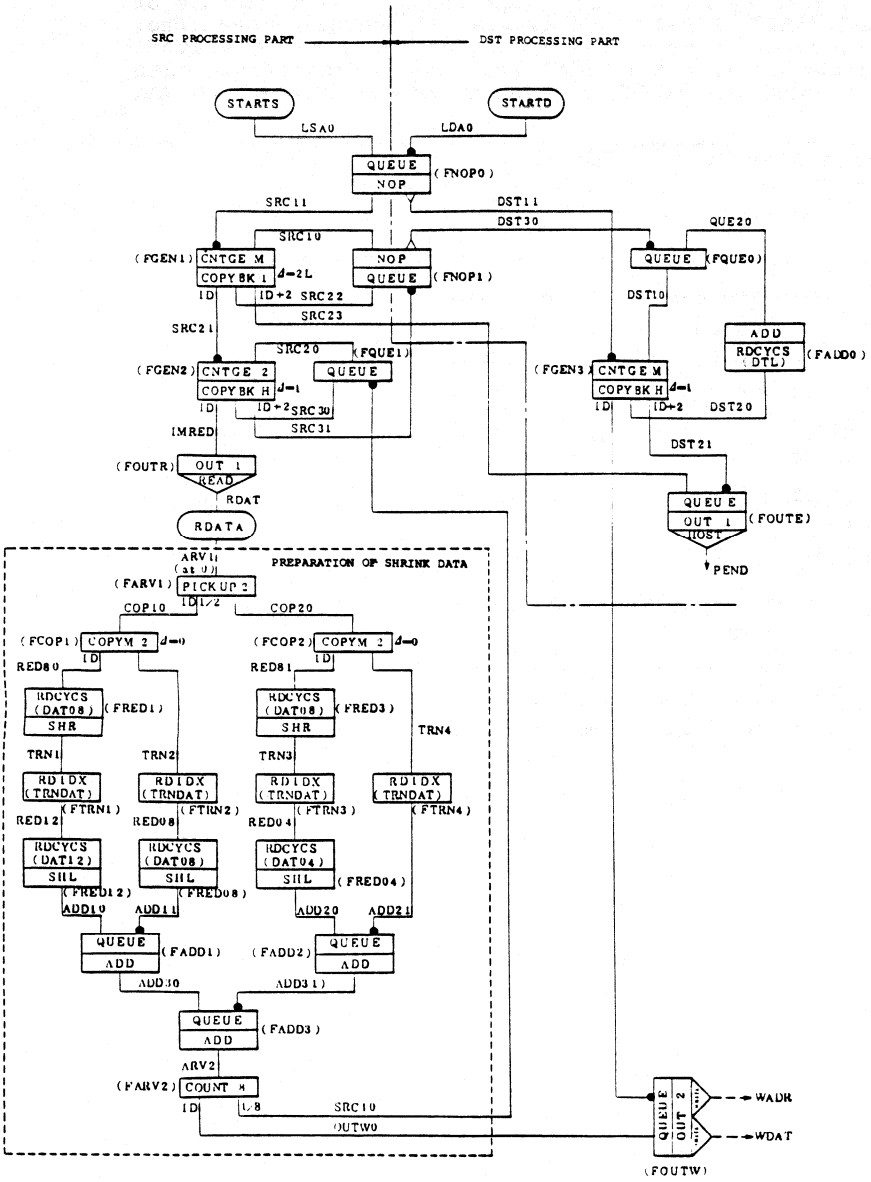
FADD0 : Upon completion of the generation of H words of horizontal addresses by FGEN3, modifies the addresses sent by FGEN3 in order to create addresses for the next line.

FNOP1 : Synchronizes the actions of FGEN1 and FGEN2 so that FGEN1 can create the starting address of a line each time FGEN2 generates addresses for one line of SRC data.

FQUE1 : Synchronizes the actions of FGEN2 and FARV2 so that SRC addresses will be created each time 8 DST data are generated.

FQUE2 : Synchronizes the actions of the SRC and DST processing parts so that the DST processing can create one line of addresses each time the SRC processing part generates one line of DST data.

Figure 4-3
A Flow Graph for Simple One-Half Shrinking



4.1.5 Tips on Writing Flow Graphs

In this program 16 SRC addresses are created for each 8 DST data generated by FGEN2, FQUE1, and FARV2. In the absence of FQUE1 and FARV2, FGEN2 would create SRC addresses one after another irrespective of the number of DST data generated, filling the GQ of the uPD7281 with the tokens of SRC data copied in FROP1 and FCOP2, leading to an overflow situation.

4.1.6 Assembler Source Listing

```
 1: ;****************************************
 2: ;
 3: ;       SHRINK 1/2
 4: ;
 5: ;----------------------------------------
 6: ;
 7: MODULE  IPP     =       8               ;
 8: ;
 9: EQUATE  L       =       64              ;
10: EQUATE  M       =       256             ;
11: EQUATE  H       =       16              ;
12: ;
13: EQUATE  HOST    =       0               ;
14: EQUATE  READ    =       4               ;
15: EQUATE  WRITE   =       5               ;
16: ;
17: EQUATE  STARTS  =       0               ;
18: EQUATE  STARTD  =       32              ;
19: ;
20: ;****************************************
21: ;
22: ;       INPUT-OUTPUT
23: ;
24: ;----------------------------------------
25: ;
26: INPUT   LSA0,   LDA0,   ARV1 AT 0       ;
27: ;
28: OUTPUT  RDAT,   WDAT,   WADR,   PEND    ;
29: ;
```

```
;****************************************
;
;       LINK TABLE
;
;----------------------------------------
;
LINK    SRC11,  DST11           =       FNOP0   (LSA0,  LDA0    )       ;
LINK    SRC21,  SRC22,  SRC23   =       FGEN1   (SRC10, SRC11   )       ;
LINK    OUTW1,  DST20,  DST21   =       FGEN3   (DST10, DST11   )       ;
LINK    IMRED,  SRC30,  SRC31   =       FGEN2   (SRC20, SRC21   )       ;
LINK    SRC10,  DST30           =       FNOP1   (SRC22, SRC31   )       ;
LINK    PEND                    =       FOUTE   (SRC23, DST21   )       ;
LINK    RDAT                    =       FOUTR   (IMRED          )       ;
LINK    SRC20                   =       FQUE1   (SRC30, SRC40   )       ;
LINK    COP10,  COP20           =       FARV1   (ARV1           )       ;
LINK    RED80,  TRN2            =       FCOP1   (COP10          )       ;
LINK    TRN1                    =       FRED1   (RED80          )       ;
LINK    RED12                   =       FTRN1   (TRN1           )       ;
LINK    ADD10                   =       FRED12  (RED12          )       ;
LINK    RED08                   =       FTRN2   (TRN2           )       ;
LINK    ADD11                   =       FRED08  (RED08          )       ;
LINK    ADD30                   =       FADD1   (ADD10, ADD11   )       ;
LINK    RED81,  TRN4            =       FCOP2   (COP20          )       ;
LINK    TRN3                    =       FRED3   (RED81          )       ;
LINK    RED04                   =       FTRN3   (TRN3           )       ;
LINK    ADD20                   =       FRED04  (RED04          )       ;
LINK    ADD21                   =       FTRN4   (TRN4           )       ;
LINK    ADD31                   =       FADD2   (ADD20, ADD21   )       ;
LINK    ARV2                    =       FADD3   (ADD30, ADD31   )       ;
LINK    OUTW0,  SRC40           =       FARV2   (ARV2           )       ;
LINK    WDAT,   WADR            =       FOUTW   (OUTW0, OUTW1   )       ;
LINK    QUE20                   =       FADD0   (DST20          )       ;
LINK    DST10                   =       FQUE0   (QUE20, DST30   )       ;
;
;****************************************
;
;       FUNCTION TABLE
;
;----------------------------------------
;
FUNCTION    FOUTR   =   OUT1    (READ,  0)                              ;
FUNCTION    FOUTW   =   OUT2    (WRTIE, 20H, 0),QUEUE   (QUEW, 16)      ;
FUNCTION    FOUTE   =   OUT1    (HOST,  0),     QUEUE   (QUEE,  1)      ;
FUNCTION    FGEN1   =   COPYBK  (1,   2*L),     CNTGE   (M      )       ;
FUNCTION    FGEN2   =   COPYBK  (H,     1),     CNTGE   (2      )       ;
FUNCTION    FGEN3   =   COPYBK  (H,     1),     CNTGE   (M      )       ;
FUNCTION    FARV1   =   PICKUP  (2      )                               ;
FUNCTION    FARV2   =   COUNT   (8      )                               ;
FUNCTION    FCOP1   =   COPYM   (2,     0)                              ;
FUNCTION    FCOP2   =   COPYM   (2,     0)                              ;
FUNCTION    FRED1   =   SHR     (X      ),      RDCYCS  (DAT08, 1)      ;
FUNCTION    FRED3   =   SHR     (X      ),      RDCYCS  (DAT08, 1)      ;
FUNCTION    FRED12  =   SHL     (X      ),      RDCYCS  (DAT12, 1)      ;
FUNCTION    FRED08  =   SHL     (X      ),      RDCYCS  (DAT08, 1)      ;
FUNCTION    FRED04  =   SHL     (X      ),      RDCYCS  (DAT04, 1)      ;
FUNCTION    FTRN1   =   RDIDX   (TRNDAT )                               ;
FUNCTION    FTRN2   =   RDIDX   (TRNDAT )                               ;
FUNCTION    FTRN3   =   RDIDX   (TRNDAT )                               ;
FUNCTION    FTRN4   =   RDIDX   (TRNDAT )                               ;
FUNCTION    FADD0   =   ADD     (X      ),      RDCYCS  (DTL,   1)      ;
FUNCTION    FADD1   =   ADD     (X      ),      QUEUE   (QUE4,  8)      ;
FUNCTION    FADD2   =   ADD     (X      ),      QUEUE   (QUE5,  8)      ;
FUNCTION    FADD3   =   ADD     (X      ),      QUEUE   (QUE6,  8)      ;
FUNCTION    FNOP0   =   NOP     (XY     ),      QUEUE   (QUE2,  1)      ;
FUNCTION    FNOP1   =   NOP     (XY     ),      QUEUE   (QUE3,  1)      ;
FUNCTION    FQUE0   =   QUEUE   (QUE0,  1)                              ;
FUNCTION    FQUE1   =   QUEUE   (QUE1,  1)                              ;
;
```

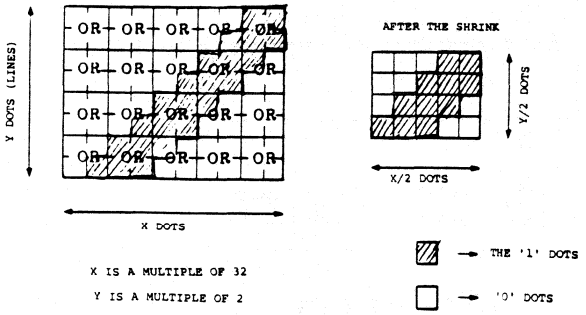
```
 98: ;****************************************
 99: ;
100: ;       DATA MEMORY
101: ;
102: ;----------------------------------------
103: ;
104: MEMORY  QUE0    =       AREA    (1      )       ;
105: MEMORY  QUE1    =       AREA    (1      )       ;
106: MEMORY  QUE2    =       AREA    (1      )       ;
107: MEMORY  QUE3    =       AREA    (1      )       ;
108: MEMORY  QUE4    =       AREA    (8      )       ;
109: MEMORY  QUE5    =       AREA    (8      )       ;
110: MEMORY  QUE6    =       AREA    (8      )       ;
111: MEMORY  QUEW    =       AREA    (16     )       ;
112: MEMORY  QUEE    =       AREA    (1      )       ;
113: MEMORY  DAT04   =       4                       ;
114: MEMORY  DAT08   =       8                       ;
115: MEMORY  DAT12   =       12                      ;
116: MEMORY  DTL     =       L-16                    ;
117: MEMORY  TRNDAT  =        0, 1, 0, 1, 2, 3, 2, 3, 0, 1, 0, 1, 2, 3, 2, 3,
118:                          4, 5, 4, 5, 6, 7, 6, 7, 4, 5, 4, 5, 6, 7, 6, 7,
119:                          0, 1, 0, 1, 2, 3, 2, 3, 0, 1, 0, 1, 2, 3, 2, 3,
120:                          4, 5, 4, 5, 6, 7, 6, 7, 4, 5, 4, 5, 6, 7, 6, 7,
121:                          8, 9, 8, 9,10,11,10,11, 8, 9, 8, 9,10,11,10,11,
122:                         12,13,12,13,14,15,14,15,12,13,12,13,14,15,14,15,
123:                          8, 9, 8, 9,10,11,10,11, 8, 9, 8, 9,10,11,10,11,
124:                         12,13,12,13,14,15,14,15,12,13,12,13,14,15,14,15,
125:                          0, 1, 0, 1, 2, 3, 2, 3, 0, 1, 0, 1, 2, 3, 2, 3,
126:                          4, 5, 4, 5, 6, 7, 6, 7, 4, 5, 4, 5, 6, 7, 6, 7,
127:                          0, 1, 0, 1, 2, 3, 2, 3, 0, 1, 0, 1, 2, 3, 2, 3,
128:                          4, 5, 4, 5, 6, 7, 6, 7, 4, 5, 4, 5, 6, 7, 6, 7,
129:                          8, 9, 8, 9,10,11,10,11, 8, 9, 8, 9,10,11,10,11,
130:                         12,13,12,13,14,15,14,15,12,13,12,13,14,15,14,15,
131:                          8, 9, 8, 9,10,11,10,11, 8, 9, 8, 9,10,11,10,11,
132:                         12,13,12,13,14,15,14,15,12,13,12,13,14,15,14,15 ;
133: ;
134: ;****************************************
135: ;
136: ;       START
137: ;
138: ;----------------------------------------
139: ;
140: START                                           ;
141: ;
142: DATA    EXEC    (IPP,   LSA0,   STARTS  )       ;
143: DATA    EXEC    (IPP,   LDA0,   STARTD  )       ;
144: ;
145: END                                             ;
```

4.2 Four-Point OR One-Half Shrinking

4.2.1 Processing Explained

In the four-point OR one-half shrinking method, the OR operation is performed on four adjacent points in a given source image area (SRC), and the resulting single dot is made into destination image (DST) data, as shown in Figure 4-4.

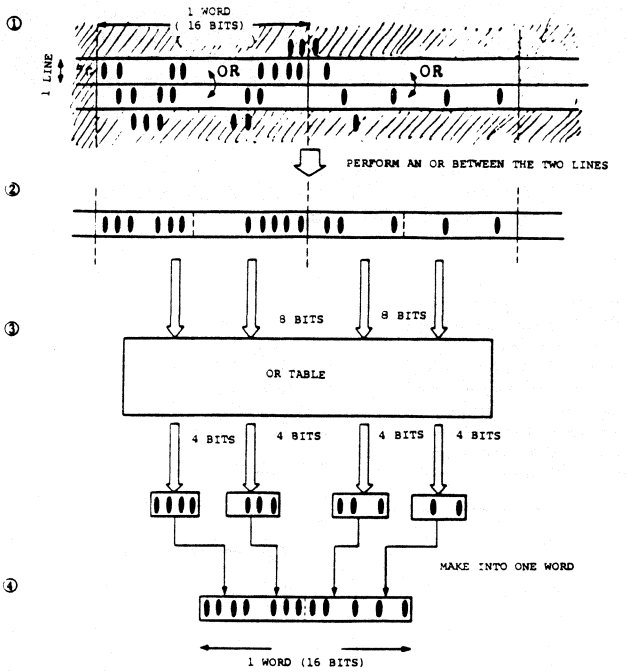
Figure 4-4
An Example of a Four-Point OR One-Half Shrinking
(45o straight line)



4.2.2 Algorithm

Since the four-point OR one-half shrinking method involves performing the OR operation on four adjacent points, the data that would not have been useful under the Simple One-Half Reduction method of Section 4.1 is required here. As shown in Figure 4-5, in this program a given line and the line following it are read one after another from the SRC in word units, the two lines are ORed, and the results are used to look up the OR table.

Figure 4.5
Algorithm for the Four-Point OR One-Half Shrinking Method



4.2.3 Parameters and Their Applicable Ranges

<Assembler-coded parameters>

L ... Number of image memory words in horizontal direction
H ... Number of destination image area (DST) words in horizontal direction
M ... Number of destination image area (DST) lines in vertical direction

<Start-up token-defined parameters>

STARTS ... Source image area (SRC) starting address
STARTD ... Destination image area (DST) starting address

The allowable values of these parameters are indicated in the table below.

Parameter	Applicable range	(Value set in the example program)
L	0 - 32767	(64)
H	1 - 16	(16)
M	1 - 256	(256)
STARTS	0 - 65535	(0)*
STARTD	0 - 65535	(32)*

* : Although STARTS and STARTD are variables (i.e., addresses), they are given default values since they are used in the assembler DATA statement. When the uPD7281 is started up, the starting addresses of the SRC and DST areas are input as execution tokens.

Since no provision is made in this program for switching banks, you should exercise care in setting the values of the parameters H and M. Maximum values that can be assigned to these parameters are H=16 and M=256; therefore, the maximum size destination image that can be shrunk by this program is 16 horizontal words and 256 vertical lines.

<Initial Values>

The initial values used in this program are basically the same as those employed in the Simple One-Half Shrinking Method described in Section 4.1, except that the lines that would be skipped in a simple one-half shrinking need to be read out from the SRC and initial values must be provided for the processing involved in the creation of these read addresses.

4.2.4 Flow Graph Explained

The four-point OR one-half shrinking method differs from the simple one-half shrinking method of Section 4.1 in the following respects:

(1) With an SRC starting address (STARTS) entered, the processing branches into two parts: A, which handles the creation of SRC addresses for every other line, as done during the simple one-half shrinking; and A', which creates those addresses which are not created by A above.
(2) These two processing steps are performed in parallel, each reading the SRC data.
(3) The two types of SRC data read in step (2) undergo an

OR operation, followed by a table lookup and the generation of shrunk data. The table used in this step is different from the one employed in the simple one-half shrinking method.

As indicated in Figure 4-6, this program consists of an SRC processing part (which reads two lines each (word unit) of SRC data and creates shrunk data) and an DST processing part (which is concerned with the generation of output addresses for the shrunk data). Each word of the shrunk data is made from four horizontally and vertically adjacent words.

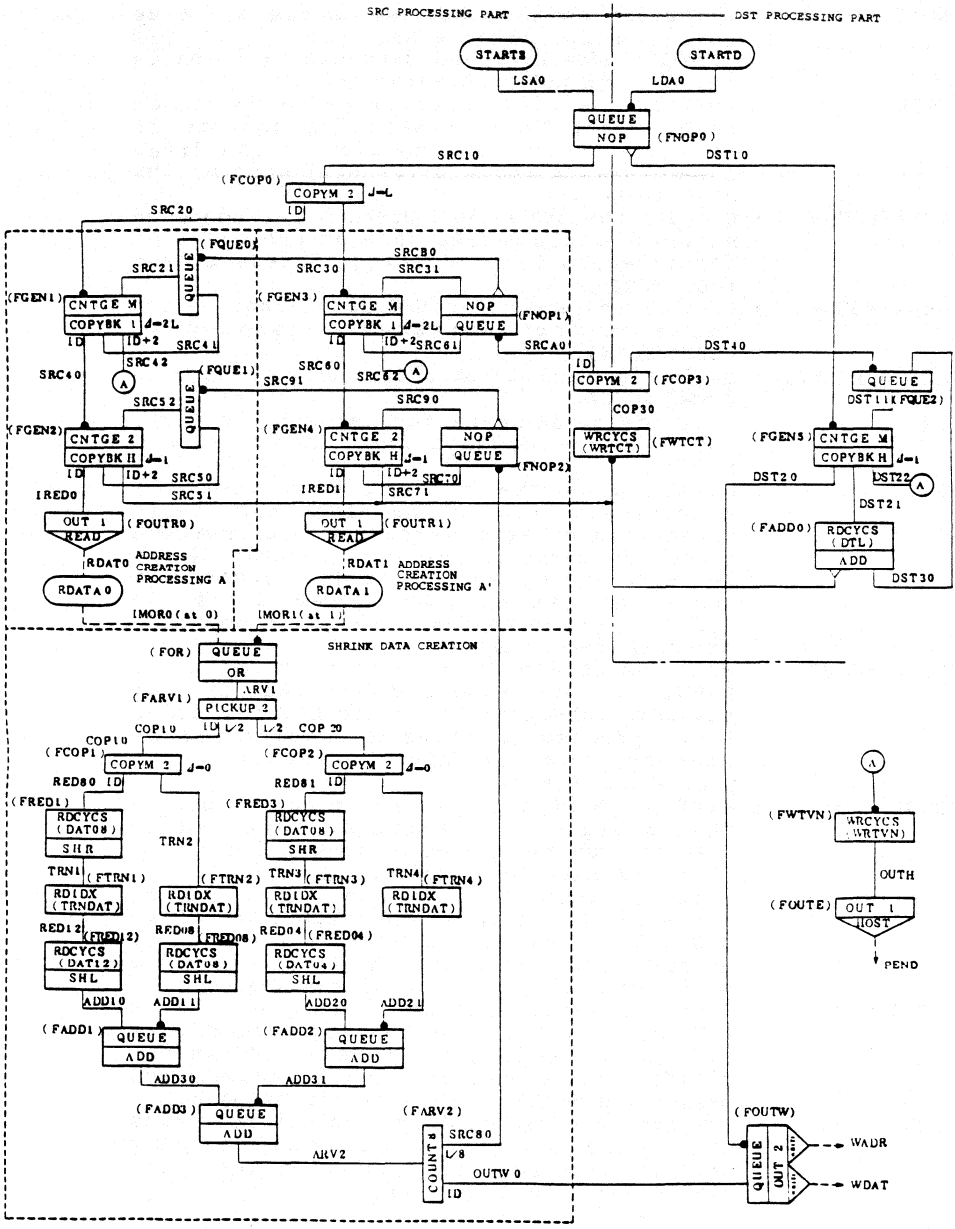
The SRC processing part receives an SRC address from the host computer that serves as the starting point of the shrinking processing. Starting with that address, the SRC processing part reads two lines each of word-unit data. An OR operation is performed between the first and second lines of data that have been read, and the resulting data is alternately distributed to the high-order 8-bit and the low-order 8-bit DST data creation processing tasks.

The high-order 8-bit and the low-order 8-bit creation processing tasks look up the OR table* on the basis of the high-order 8-bit and low-order 8-bit values received, respectively. Each then generates high-order 4-bit and low-order 4-bit data. This means 8-bit shrunk data is created by two DST words. The two 8-bit data thus generated are placed in their respective positions within the 16-bit word from which the shrunk data is generated.

The DST processing part receives a starting address from the host computer for writing the shrunk data. It generates the output addresses based on this address.

* : This is a 4-bit data table which is made by dividing 8-bit data into 2-bit segments and by performing an OR on each of the segments.

Figure 4-6
A Four-Point OR One-Half Shrinking Flow Graph



<Explanation of Nodes>

FNOP0 : Synchronizes the actions of the start-up tokens which indicate SRC (STARTS) and DST (STARTD) addresses and sends these token to the SRC and DST processing parts.

FCOP0 : So that two lines each of SRC data can be read, generates the starting address of second line data in addition to the first line starting address (STARTS) sent by the host computer.

FGEN1/FGEN3 : Creates the starting addresses for odd/even numbered lines of SRC data on the basis of first/second line starting addresses sent from FCOP0.

FGEN2/FGEN4 : Creates addresses for SRC data lines on the basis of the addresses generated in FGEN1/FGEN3.

FOUTR0/FOUTR1 : Reads the contents of addresses generated in FGEN2/FGEN4.

FOR : Performs an OR operation between the first and second line data read in FOUTR0 and FOUTR1.

FARV1 : Distributes the OR data sent from FOR alternately between the high-order 8-bit creation and the low-order 8-bit creation processing tasks so that a horizontal OR can be performed on two 16-bit data created by performing a vertical OR (between lines), in order to generate DST data (four-point OR one-half shrunk data).

FCOP1/FCOP2 : Make two copies of the data distributed by FARV1 to provide for the fact that each of the DST high-order 8-bit and DST low-order 8-bit processing steps are comprised of high-order 4-bit and low-order 4-bit processing tasks.

FRED1, FRED3 : Performs a right 8-bit shift on OR data so that a high-order 4-bit segment can be created from the high-order 8-bit segment of 16-bit OR data by using the OR table.

FTRN1/FTRN2 : Prepares 4-bit data by looking up the OR table on the basis of the high-order/low-order 8-bit values of OR data (created by dividing eight bits into 2-bit segments and performing an OR on each of the segments).

FTRN3/FTRN4 : Prepares 4-bit data by looking up the OR table on the basis of the high-order/low-order 8-bit values of OR data (created by dividing eight bits into 2-bit segments and performing an OR on each of the segments).

FRED12/FRED08 : Prepares the high-order/low-order four bits of the high-order 8-bit DST data by performing a left 12/8 shift on the 4-bit data created in FTRN1/FTRN2.

FRED04 : Creates the high-order four bits of the low-order 8-bit DST data by performing a left 4-bit shift on the 4-bit data created in FTRN3.

FADD1 : Creates the high-order eight bits of DST data by adding together the data generated by FRED12 and FRED08.

FADD2 : Creates the low-order eight bits of DST data by adding together the data generated by FRED04 and FTRN4.

FADD3 : Creates 16-bit DST data by adding together the data generated by FADD1 and FADD2.

FARV2 : For each 8 DST data sent from FADD3, makes two copies of the data and sends one copy to FQUE1 to notify completion of the generation of the eight DST data.

FGEN5 : Creates M lines of addresses for H horizontal words, starting from the DST address (STARTD) sent from FNOP0.

FOUTW : Writes the DST data generated by the SRC processing part into the DST address created in FGEN3.

FADD0 : Modifies the address sent from FGEN5 in order to address the starting address of the next line upon completion of address generation for H horizontal words by FGEN5.

FNOP1/FNOP0, FQUE0/FQUE1 : Regulates and synchronizes the address generation by FGEN1/FGEN2, and FGEN3/FGEN4.

FWTCT : Synchronizes the actions of the SRC and DST processing parts for each completion of SRC and DST addresses (i.e., first and second line addresses) for one line of DST data.

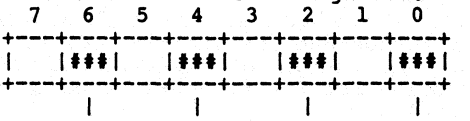
FCOP3 : Notifies the SRC and DST processing parts of the completion of addresses for one line of DST data by the SRC and DST processsing parts.

FWTVN : Synchronizes completion of all addresses by the SRC and DST processing parts.

PEND : Notifies the host computer of completion of generation for all addresses for a reduction processing task.

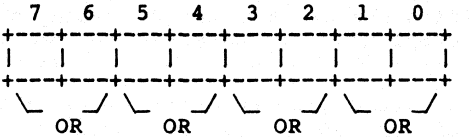
SRC Data	Simple One-Half Shrinking Table (Note 1)	Four-Point OR One-Half Shrinking Table (Note 2)
Bits 7 6 5 4 3 2 1 0	Table Data	Table Data
0 0 0 0 0 0 0 0	0 0 0 0	0 0 0 0
0 0 0 0 0 0 0 1	0 0 0 1	0 0 0 1
0 0 0 0 0 0 1 0	0 0 0 0	0 0 0 1
0 0 0 0 0 0 1 1	0 0 0 1	0 0 0 1
0 0 0 0 0 1 0 0	0 0 1 0	0 0 1 0
:	:	:
:	:	:
1 0 1 1 1 0 0 1	0 1 0 1	1 1 1 1
1 0 1 1 1 0 1 0	0 1 0 0	1 1 1 1
1 0 1 1 1 0 1 1	0 1 0 1	1 1 1 1
:	:	:
1 1 1 1 1 1 1 1	1 1 1 1	1 1 1 1

Note 1: The odd-numbered bits are ignored.



Only these bits are eliminated.

Note 2: An OR is performed between the odd-numbered and even-numbered bits.

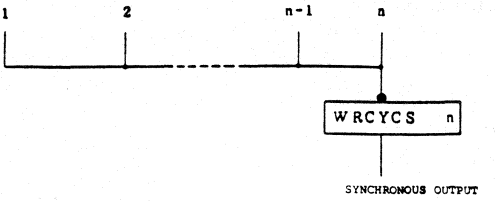


Each of these groups is ORed.

4.2.5 Tips on Writing Flow Graphs

Although the QUEUE instruction is normally employed in synchronizing various activities, in some cases the WRCYCS and WRCYCL instructions are used (as in the case of the FWTCT and FWTVN nodes in this program).

When BS (buffer size) = WC (write counter), the FTRC = 1 token does not vanish in either the WRCYCS or WRCYCL instruction. When there are many data senders and you wish to synchronize various actions including those of the senders, the flow graph should be constructed in the manner indicated in the figure below.



Notice that in this case n words in the DM (Data Memory) must be assigned as dummy data.

4.2.6 Assembler Source Listing

```
 1: ;*************************************
 2: ;
 3: ;       SHRINK 1/2  (4 POINT OR )
 4: ;
 5: ;-------------------------------------
 6: ;
 7: MODULE  IPP     =       8               ;
 8: ;
 9: EQUATE  L       =       64              ;
10: EQUATE  M       =       256             ;
11: EQUATE  H       =       16              ;
12: ;
13: EQUATE  HOST    =       0               ;
14: EQUATE  READ    =       4               ;
15: EQUATE  WRITE   =       5               ;
16: ;
17: EQUATE  STARTS  =       0               ;
18: EQUATE  STARTD  =       32              ;
19: ;
20: ;*************************************
21: ;
22: ;       INPUT-OUTPUT
23: ;
24: ;-------------------------------------
25: ;
26: INPUT   LSA0,   LDA0,   IMOR0 AT 0,     IMOR1 AT 1      ;
27: ;
28: OUTPUT  RDAT0,  RDAT1,  WDAT,   WADR,   PEND            ;
29: ;
30: ;*************************************
31: ;
32: ;       LINK TABLE
33: ;
34: ;-------------------------------------
35: ;
36: LINK    SRC10,  DST10           =       FNOP0   (LSA0,  LDA0    )       ;
37: LINK    SRC20,  SRC30           =       FCOP0   (SRC10          )       ;
38: LINK    DST20,  DST21,  DST22   =       FGEN5   (DST11, DST10   )       ;
39: LINK    SRC40,  SRC41,  SRC42   =       FGEN1   (SRC21, SRC20   )       ;
40: LINK    SRC60,  SRC61,  SRC62   =       FGEN3   (SRC31, SRC30   )       ;
41: LINK    IRED0,  SRC50,  SRC51   =       FGEN2   (SRC52, SRC40   )       ;
42: LINK    SRC21                   =       FQUE0   (SRC41, SRC80   )       ;
43: LINK    OUTH                    =       FWTVN   (       ,SRC42  )       ;
44: LINK    RDAT0                   =       FOUTR0  (IRED0          )       ;
45: LINK    SRC52                   =       FQUE1   (SRC50, SRC91   )       ;
46: LINK    COP30                   =       FWTCT   (       ,SRC51  )       ;
47: LINK    IRED1,  SRC70,  SRC71   =       FGEN4   (SRC90, SRC60   )       ;
48: LINK    SRC31,  SRC80           =       FNOP1   (SRC61, SRCA0   )       ;
49: LINK    OUTH                    =       FWTVN   (       ,SRC62  )       ;
50: LINK    RDAT1                   =       FOUTR1  (IRED1          )       ;
51: LINK    SRC90,  SRC91           =       FNOP2   (SRC70, SRC80   )       ;
52: LINK    COP30                   =       FWTCT   (       ,SRC71  )       ;
53: LINK    ARV1                    =       FOR     (IMOR0, IMOR1   )       ;
54: LINK    COP10,  COP20           =       FARV1   (ARV1           )       ;
55: LINK    RED80,  TRN2            =       FCOP1   (COP10          )       ;
56: LINK    TRN1                    =       FRED1   (RED80          )       ;
57: LINK    RED12                   =       FTRN1   (TRN1           )       ;
58: LINK    ADD10                   =       FRED12  (RED12          )       ;
59: LINK    RED08                   =       FTRN2   (TRN2           )       ;
60: LINK    ADD11                   =       FRED08  (RED08          )       ;
```

```
  61: LINK     ADD30                    =       FADD1   (ADD10, ADD11   )       ;
  62: LINK     RED81,  TRN4             =       FCOP2   (COP20          )       ;
  63: LINK     TRN3                     =       FRED3   (RED81          )       ;
  64: LINK     RED04                    =       FTRN3   (TRN3           )       ;
  65: LINK     ADD20                    =       FRED04  (RED04          )       ;
  66: LINK     ADD21                    =       FTRN4   (TRN4           )       ;
  67: LINK     ADD31                    =       FADD2   (ADD20, ADD21   )       ;
  68: LINK     ARV2                     =       FADD3   (ADD30, ADD31   )       ;
  69: LINK     OUTW0,  SRC80            =       FARV2   (ARV2           )       ;
  70: LINK     WDAT,   WADR             =       FOUTW   (OUTW0, DST20   )       ;
  71: LINK     DST30,  DST31            =       FADD0   (DST21          )       ;
  72: LINK     OUTH                     =       FWTVN   (       ,DST22  )       ;
  73: LINK     DST11                    =       FQUE2   (DST30, DST40   )       ;
  74: LINK     COP30                    =       FWTCT   (       ,DST31  )       ;
  75: LINK     SRCA0,  DST40            =       FCOP3   (COP30          )       ;
  76: LINK     PEND                     =       FOUTE   (OUTH           )       ;
  77: ;
  78: ;**************************************
  79: ;
  80: ;       FUNCTION TABLE
  81: ;
  82: ;--------------------------------------
  83: ;
  84: FUNCTION     FOUTR0  =   OUT1    (READ,  0)                                  ;
  85: FUNCTION     FOUTR1  =   OUT1    (READ,  1)                                  ;
  86: FUNCTION     FOUTW   =   OUT2    (WRTIE, 20H, 0),QUEUE   (QUEW,  16)         ;
  87: FUNCTION     FOUTE   =   OUT1    (HOST,  0)                                  ;
  88: FUNCTION     FGEN1   =   COPYBK  (1,   2*L),     CNTGE   (M          )       ;
  89: FUNCTION     FGEN2   =   COPYBK  (H ,    1),     CNTGE   (2          )       ;
  90: FUNCTION     FGEN3   =   COPYBK  (1,   2*L),     CNTGE   (M          )       ;
  91: FUNCTION     FGEN4   =   COPYBK  (H,     1),     CNTGE   (2          )       ;
  92: FUNCTION     FGEN5   =   COPYBK  (H,     1),     CNTGE   (M          )       ;
  93: FUNCTION     FCOP0   =   COPYM   (2,     L)                                  ;
  94: FUNCTION     FCOP1   =   COPYM   (2,     0)                                  ;
  95: FUNCTION     FCOP2   =   COPYM   (2,     0)                                  ;
  96: FUNCTION     FCOP3   =   COPYM   (2,     0)                                  ;
  97: FUNCTION     FARV1   =   PICKUP  (2      )                                   ;
  98: FUNCTION     FARV2   =   COUNT   (8      )                                   ;
  99: FUNCTION     FRED1   =   SHR     (X      ),      RDCYCS  (DAT08, 1)          ;
 100: FUNCTION     FRED3   =   SHR     (X      ),      RDCYCS  (DAT08, 1)          ;
 101: FUNCTION     FRED12  =   SHL     (X      ),      RDCYCS  (DAT12, 1)          ;
 102: FUNCTION     FRED08  =   SHL     (X      ),      RDCYCS  (DAT08, 1)          ;
 103: FUNCTION     FRED04  =   SHL     (X      ),      RDCYCS  (DAT04, 1)          ;
 104: FUNCTION     FTRN1   =   RDIDX   (TRNDAT )                                   ;
 105: FUNCTION     FTRN2   =   RDIDX   (TRNDAT )                                   ;
 106: FUNCTION     FTRN3   =   RDIDX   (TRNDAT )                                   ;
 107: FUNCTION     FTRN4   =   RDIDX   (TRNDAT )                                   ;
 108: FUNCTION     FADD0   =   ADD     (XY     ),      RDCYCS  (DTL,   1)          ;
 109: FUNCTION     FADD1   =   ADD     (X      ),      QUEUE   (QUE6,  8)          ;
 110: FUNCTION     FADD2   =   ADD     (X      ),      QUEUE   (QUE7,  8)          ;
 111: FUNCTION     FADD3   =   ADD     (X      ),      QUEUE   (QUE8,  3)          ;
 112: FUNCTION     FNOP0   =   NOP     (XY     ),      QUEUE   (QUE3,  1)          ;
 113: FUNCTION     FNOP1   =   NOP     (XY     ),      QUEUE   (QUE4,  1)          ;
 114: FUNCTION     FNOP2   =   NOP     (XY     ),      QUEUE   (QUE5,  1)          ;
 115: FUNCTION     FQUE0   =   QUEUE   (QUE0,  1)                                  ;
 116: FUNCTION     FQUE1   =   QUEUE   (QUE1,  1)                                  ;
 117: FUNCTION     FQUE2   =   QUEUE   (QUE2,  1)                                  ;
 118: FUNCTION     FWTCT   =   WRCYCS  (WRTCT, 3)                                  ;
 119: FUNCTION     FWTVN   =   WRCYCS  (WRTVN, 3)                                  ;
 120: FUNCTION     FOR     =   OR      (X      ),      QUEUE   (QUE9, 16)          ;
```

```
121: ;
122: ;*****************************************
123: ;
124: ;       DATA MEMORY
125: ;
126: ;-----------------------------------------
127: ;
128: MEMORY  QUE0    =       AREA    (1      )       ;
129: MEMORY  QUE1    =       AREA    (1      )       ;
130: MEMORY  QUE2    =       AREA    (1      )       ;
131: MEMORY  QUE3    =       AREA    (1      )       ;
132: MEMORY  QUE4    =       AREA    (1      )       ;
133: MEMORY  QUE5    =       AREA    (1      )       ;
134: MEMORY  QUE6    =       AREA    (8      )       ;
135: MEMORY  QUE7    =       AREA    (8      )       ;
136: MEMORY  QUE8    =       AREA    (8      )       ;
137: MEMORY  QUE9    =       AREA    (16     )       ;
138: MEMORY  QUEW    =       AREA    (16     )       ;
139: MEMORY  DAT04   =       4                       ;
140: MEMORY  DAT08   =       8                       ;
141: MEMORY  DAT12   =       12                      ;
142: MEMORY  DTL     =       L-H                     ;
143: MEMORY  WRTCT   =       AREA    (3      )       ;
144: MEMORY  WRTVN   =       AREA    (3      )       ;
145: MEMORY  TRNDAT  =        0, 1, 1, 1, 2, 3, 3, 3, 2, 3, 3, 3, 2, 3, 3, 3,
146:                          4, 5, 5, 5, 6, 7, 7, 7, 6, 7, 7, 7, 6, 7, 7, 7,
147:                          4, 5, 5, 5, 6, 7, 7, 7, 6, 7, 7, 7, 6, 7, 7, 7,
148:                          4, 5, 5, 5, 6, 7, 7, 7, 6, 7, 7, 7, 6, 7, 7, 7,
149:                          8, 9, 9, 9,10,11,11,11,10,11,11,11,10,11,11,11,
150:                         12,13,13,13,14,15,15,15,14,15,15,15,14,15,15,15,
151:                         12,13,13,13,14,15,15,15,14,15,15,15,14,15,15,15,
152:                         12,13,13,13,14,15,15,15,14,15,15,15,14,15,15,15,
153:                          8, 9, 9, 9,10,11,11,11,10,11,11,11,10,11,11,11,
154:                         12,13,13,13,14,15,15,15,14,15,15,15,14,15,15,15,
155:                         12,13,13,13,14,15,15,15,14,15,15,15,14,15,15,15,
156:                         12,13,13,13,14,15,15,15,14,15,15,15,14,15,15,15,
157:                          8, 9, 9, 9,10,11,11,11,10,11,11,11,10,11,11,11,
158:                         12,13,13,13,14,15,15,15,14,15,15,15,14,15,15,15,
159:                         12,13,13,13,14,15,15,15,14,15,15,15,14,15,15,15,
160:                         12,13,13,13,14,15,15,15,14,15,15,15,14,15,15,15 ;
161: ;
162: ;*****************************************
163: ;
164: ;       START
165: ;
166: ;-----------------------------------------
167: START                                               ;
168: ;
169: DATA    EXEC    (IPP,   LSA0,   STARTS  )       ;
170: DATA    EXEC    (IPP,   LDA0,   STARTD  )       ;
171: ;
172: END                                               ;
```

4.3 Neighboring 16-Point Addition One-Quarter Shrinking

4.3.1 Processing Explained

A one-quarter shrinking involves shrinking 4 horizontal and 4 vertical bits, for a total of 16 bits, into one bit. In this neighboring 16-point addition reduction, the number of "1"s in the 16-bit data are counted and shrunk data is set to "1" if the sum is greater than a given value and "0" otherwise.

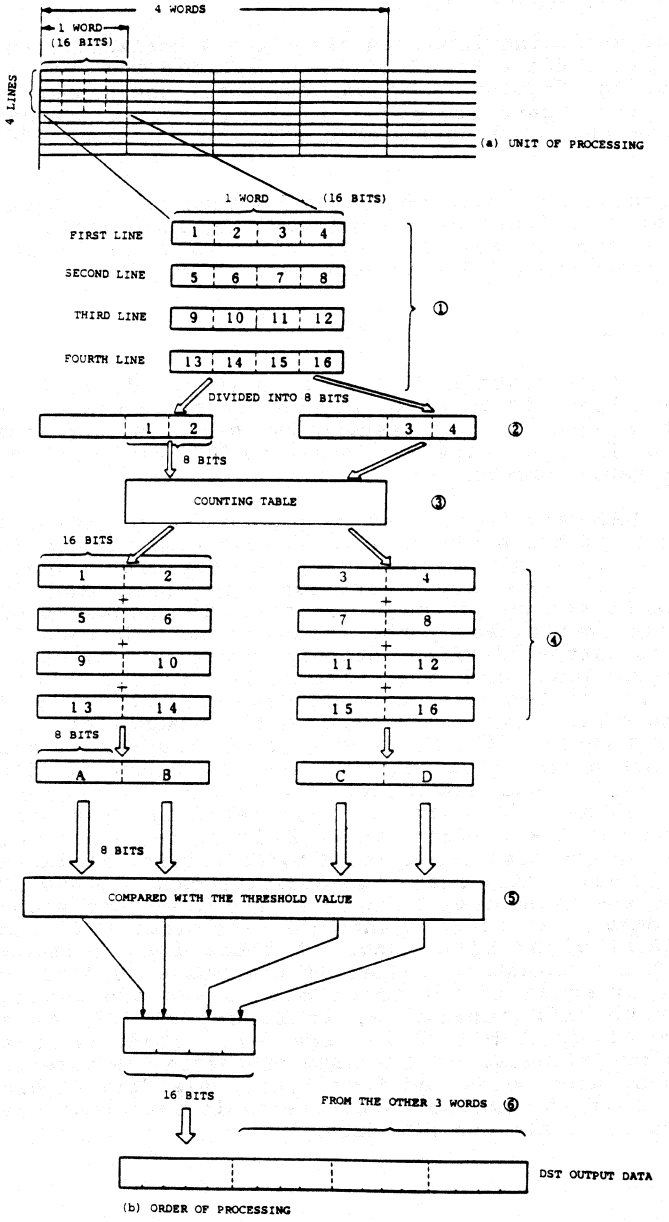
The basic size of source image area (SRC) that can be shrunk by this program is four horizontal words by four vertical words, as shown in Figure 4-7. This is because the destination image area (DST) is addressed in word units.

4.3.2 Algorithm

The number of "1"s contained in a given 16-bit data, four bits by four lines, is counted. If the number is number is greater than or equal to a threshold value, the shrunk data bit is set to "1"; otherwise, it is set to "0". The counting is done by a table lookup.

To write the DST data into image memory in 1-word (16-bit) units, (1) each of the 4 SRC data words read from the SRC is divided into (2) high-order 8-bit and low-order 8-bit segments to reduce the size of the table required, and (3) the table is looked up for each segment. The contents of this table are determined by the following method: each 8-bit segment is further divided into high-order and low-order 4-bit subsegments, and the number of "1"s in these subsegments is written in a word addressed by the value of each 8-bit segment. (The table used in the example program is 256-words x 16-bit data divided into the high-order and low-order 8-bit parts whose contents are the number of "1"s in the high-order and low-order 4-bits of the segment respectively.) Then, (4) data for four vertical lines are read from the table and added, thus tallying the number of "1"s included in the area encompassed by four vertical lines and four horizontal bits. Then two additive data, AB and CD (created from the high-order eight bits and low-order eight bits of SRC data), are divided into (5) high-order (A,C) and low-order (B,D) eight bits. Each of these four data is compared with the threshold value. If the number of "1"s is greater than or equal to the threshold value, the shrunk data is set to "1"; otherwise, it is set to "0". As a result, the four 1-bit data are extracted as the constituents of DST data. (6) The same processing is done on the continuous three words and four lines, yielding 12-bit shrunk data. This is combined with the 4-bit data from the previous step to create the DST data.

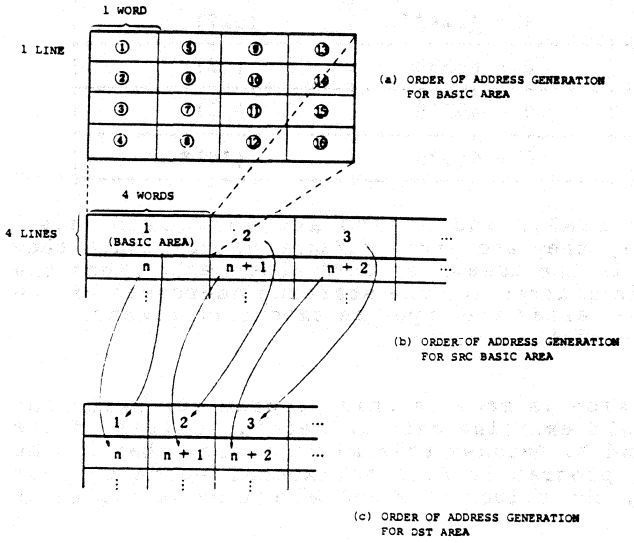
Figure 4-7
Algorithm for the Neighboring 16-Point Addition
One-Quarter Shrinking Method



The generation of SRC area addresses is based on the generation of addresses for an area defined by four horizontal words and four vertical lines. This constitutes the minimum unit of a shrinking processing. This basic area is shifted horizontally (Figure 4-8 (b)). The addresses in the basic area are generated four vertical lines at a time, as indicated in Figure 4-8 (a).

For the generation of addresses for a DST area, one address is created horizontally for each 16 addresses of the SRC basic area, as shown in Figure 4-8 (c).

Figure 4-8
Order of Address Generation



4.3.3 Parameters and Their Applicable Range

<Assembler-coded parameters>

L ... Number of image memory words in horizontal direction
H ... Number of source image area (SRC) words in horizontal direction
M ... Number of source image area (SRC) lines in vertical direction

<Start-up token-defined parameters>

THRDAT ... Threshold value
STARTS ... Source image area (SRC) starting address
STARTD ... Destination image area (DST) starting address

The allowable values of these parameters are indicated in the table below.

Parameter	Applicable range	(Value set in the example program)
L	0 - 16383	(64)
H	4 - 1024**	(32)
M	4 - 1024**	(512)
THRDAT	0 - 65535	(8)*
STARTS	0 - 65535	(0)*
STARTD	0 - 65535	(48)*

* : Although STARTS and STARTD are variables (i.e., addresses), they are given default values since they are used in the assembler DATA statement. When the uPD7281 is started up, the starting addresses of the SRC and DST areas are input as execution tokens.
** : A multiple of 4

Since no provision is made in this program for switching banks, you should exercise care in setting values of the parameters H and M. Because this minimum area that can be shrunk by this program is four horizontal words by four vertical lines, the values of H and M must be multiples of four.

4.3.4 Flow Graph Explained

Since, in this program the image memory is accessed in units of 16 bits, data are written to DST in word units. This also simplifies processing. Therefore, the SRC area is treated by using a 4-word horizontal by 4-line vertical area as the minimum unit.

The following explains the flow graph in Figure 4-9.

FGEC1 creates the addresses for every other four vertical lines of the SRC area, starting from the SRC starting

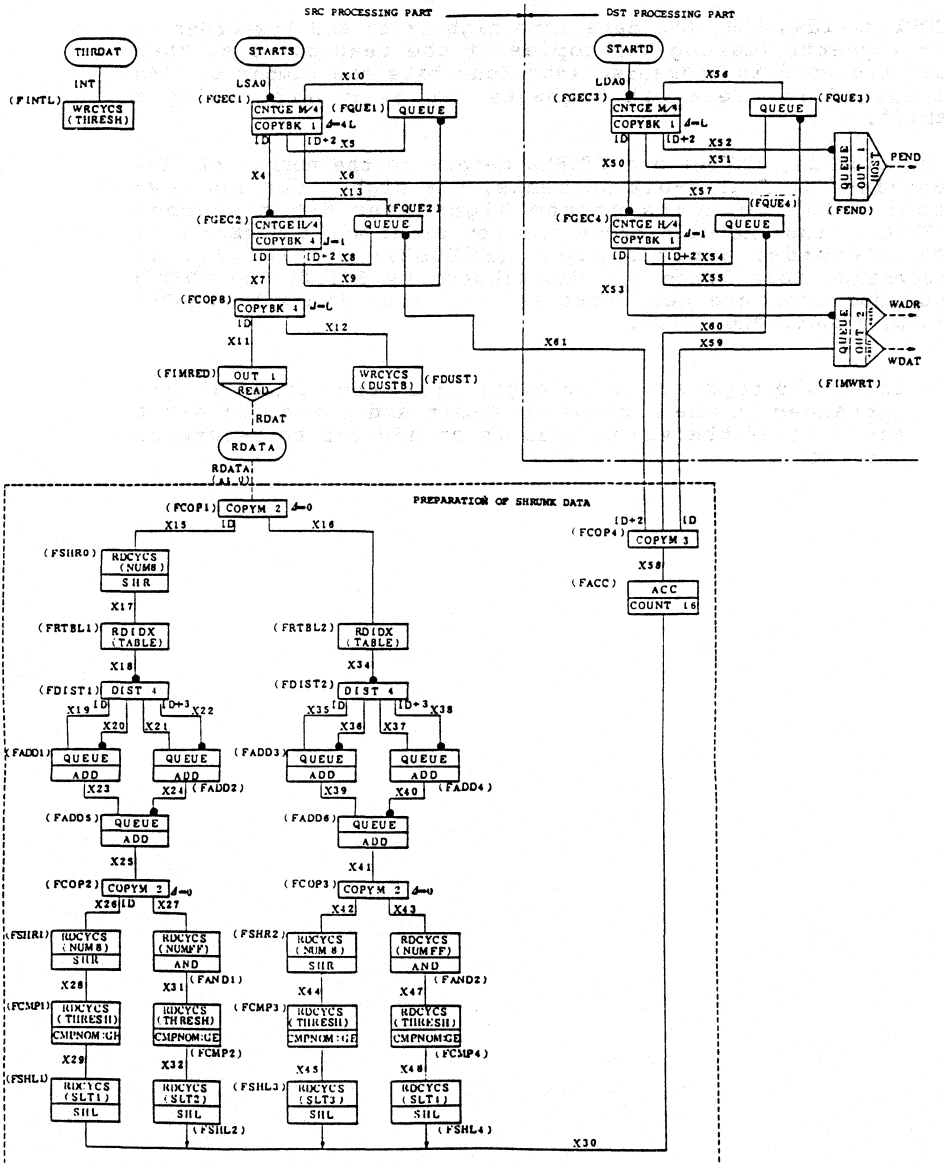
address (STARTS) given by the host computer. Then, FGEC2 creates four horizontal addresses from the given addresses. Upon receipt of these addresses, FCOPB creates addresses for four lines, making them into SRC read addresses.

FCOP1 divides the SRC data into high-order and low-order 8-bit segments (making two copies of the read data) so that when the word is segmented into four bits the number of "1"s contained in the 4-bit segments can be counted using a table*.

Specifically, FRTBL1 and FTTBL2 determine the number of "1"s contained in four 4-bit segments, the high-order and low-order 4-bit segments of the high-order 8-bit segment (FRTBL1), and the high-order and low-order 4-bit segments of the low-order 8-bit segment (FTTBL2); the results of processing four lines are distributed by FDIST1 and FDST2; and additions are performed for the four lines by FADD5, FADD3, FADD4, and FADD6.

* : The table represents, in eight bits, the number of "1"s contained in the high-order 4-bit and low-order 4-bit segments of the value that is an address to reference the table.

Figure 4-9
A Flow Graph of the Neighboring 16-Point Addition
One-Quarter Shrinking Method



The high-order 8-bit and low-order 8-bit segments of the data resulting from the addition operation indicate the number of "1"s in the four bit by four line area and the number of "1"s in the next area, respectively; high-order eight bits and low-order eight bits are divided into two words. If the value of this word is less than the value specified in the program, 1-bit data "0" is created and shifted by FSHL1 through FSHL4 so as to create the constituent of DST data. Otherwise, 1-bit data "1" is created and shifted in the same manner.

One DST address is created in the horizontal direction for each 16 addresses (4 horizontal words by 4 vertical lines) of the SRC area, using the DST starting address (STARTD) sent from the host computer as the starting point.

When all 16 bits of DST data have been generated, they are written to the given DST address.

4.3.5 Assembler Source Listing

```
;****************************************
;
;       SHRINK 1/4
;
;----------------------------------------
;
MODULE  IPP     =       8               ;
;
EQUATE  L       =       64              ;
EQUATE  H       =       32              ;
EQUATE  M       =       512             ;
;
EQUATE  THRDAT  =       8               ;
EQUATE  STARTS  =       0               ;
EQUATE  STARTD  =       48              ;
;
EQUATE  HOST    =       0               ;
EQUATE  READ    =       4               ;
EQUATE  WRITE   =       5               ;
;
;****************************************
;
;       INPUT-OUTPUT
;
;----------------------------------------
;
INPUT   INT.    LSA0.   LDA0.   RDATA AT 0      ;
;
OUTPUT  RDAT.   WDAT.   WADR.   PEND            ;
;
;****************************************
;
;       LINK TABLE
;
;----------------------------------------
;
LINK                            =       FINTL   (INT            )       ;
LINK    X4.     X5.     X6      =       FGEC1   (X10.   LSA0    )       ;
LINK    X7.     X8.     X9      =       FGEC2   (X13.   X4      )       ;
LINK    X10                     =       FQUE1   (X5.    X9      )       ;
LINK    PEND                    =       FEND    (X6.    X52     )       ;
LINK    X11.    X12             =       FCOPB   (X7             )       ;
LINK    X13                     =       FQUE2   (X8.    X61     )       ;
LINK    RDAT                    =       FINRED  (X11            )       ;
LINK                            =       FDUST   (X12            )       ;
LINK    X15.    X16             =       FCOP1   (RDATA          )       ;
LINK    X17                     =       FSHR0   (X15            )       ;
LINK    X18                     =       FRTBL1  (X17            )       ;
LINK    X19.    X20.    X21. X22=       FDIST1  (       .X18    )       ;
LINK    X23                     =       FADD1   (X19.   X20     )       ;
LINK    X24                     =       FADD2   (X21.   X22     )       ;
LINK    X25                     =       FADD5   (X23.   X24     )       ;
LINK    X26.    X27             =       FCOP2   (X25            )       ;
LINK    X28                     =       FSHR1   (X26            )       ;
```

```
 55: LINK     X29                          =       FCMP1   (X28            )     ;
 56: LINK     X30                          =       FSHL1   (X29            )     ;
 57: LINK     X31                          =       FAND1   (X27            )     ;
 58: LINK     X32                          =       FCMP2   (X31            )     ;
 59: LINK     X33                          =       FSHL2   (X32            )     ;
 60: LINK     X34                          =       FRTBL2  (X16            )     ;
 61: LINK     X35,  X36,  X37,  X38        =       FDIST2  (       ,X34    )     ;
 62: LINK     X39                          =       FADD3   (X35,   X36     )     ;
 63: LINK     X40                          =       FADD4   (X37,   X38     )     ;
 64: LINK     X41                          =       FADD6   (X39,   X40     )     ;
 65: LINK     X42,    X43                  =       FCOP3   (X41            )     ;
 66: LINK     X44                          =       FSHR2   (X42            )     ;
 67: LINK     X45                          =       FCMP3   (X44            )     ;
 68: LINK     X30                          =       FSHL3   (X45            )     ;
 69: LINK     X47                          =       FAND2   (X43            )     ;
 70: LINK     X48                          =       FCMP4   (X47            )     ;
 71: LINK     X30                          =       FSHL4   (X48            )     ;
 72: LINK     X58                          =       FACC    (X30            )     ;
 73: LINK     X59,    X60,    X61          =       FCOP4   (X58            )     ;
 74: LINK     X50,    X51,    X52          =       FGEC3   (X56,   LDA0    )     ;
 75: LINK     X53,    X54,    X55          =       FGEC4   (X57,   X50     )     ;
 76: LINK     X56                          =       FQUE3   (X51,   X55     )     ;
 77: LINK     WDAT,   WADR                 =       FIMWRT  (X59,   X53     )     ;
 78: LINK     X57                          =       FQUE4   (X54,   X60     )     ;
 79: ;
 80: ;************************************
 81: ;
 82: ;       FUNCTION TABLE
 83: ;
 84: ;----------------------------------------
 85: ;
 86: FUNCTION   FINTL   =   WRCYCS  (THRESH,1)                                    ;
 87: FUNCTION   FIMRED  =   OUT1    (READ,   0)                                   ;
 88: FUNCTION   FIMWRT  =   OUT2    (WRTIE, 20H, 0),QUEUE   (QUEW, 1)             ;
 89: FUNCTION   FEND    =   OUT1    (HOST,   0),    QUEUE   (QUEE, 1)             ;
 90: FUNCTION   FGEC1   =   COPY3K  (1,   4*L),     CNTGE   (M/4    )             ;
 91: FUNCTION   FGEC2   =   COPY3K  (4,     1),     CNTGE   (M/4    )             ;
 92: FUNCTION   FGEC3   =   COPY3K  (1,     L),     CNTGE   (M/4    )             ;
 93: FUNCTION   FGEC4   =   COPY3K  (1,     1),     CNTGE   (M/4    )             ;
 94: FUNCTION   FQUE1   =   QUEUE   (QUE1,  1)                                    ;
 95: FUNCTION   FQUE2   =   QUEUE   (QUE2,  1)                                    ;
 96: FUNCTION   FQUE3   =   QUEUE   (QUE3,  1)                                    ;
 97: FUNCTION   FQUE4   =   QUEUE   (QUE4,  1)                                    ;
 98: FUNCTION   FCOP8   =   COPY3K  (4,     L)                                    ;
 99: FUNCTION   FCOP1   =   COPYM   (2,     0)                                    ;
100: FUNCTION   FCOP2   =   COPYM   (2,     0)                                    ;
101: FUNCTION   FCOP3   =   COPYM   (2,     0)                                    ;
102: FUNCTION   FCOP4   =   COPYM   (3,     0)                                    ;
103: FUNCTION   FSHR0   =   SHR     (X       ),     RDCYCS  (NUM8,  1)            ;
104: FUNCTION   FSHR1   =   SHR     (X       ),     RDCYCS  (NUM8,  1)            ;
105: FUNCTION   FSHR2   =   SHR     (X       ),     RDCYCS  (NUM8,  1)            ;
106: FUNCTION   FSHL1   =   SHL     (X       ),     RDCYCS  (SLT1,  4)            ;
107: FUNCTION   FSHL2   =   SHL     (X       ),     RDCYCS  (SLT2,  4)            ;
108: FUNCTION   FSHL3   =   SHL     (X       ),     RDCYCS  (SLT3,  4)            ;
109: FUNCTION   FSHL4   =   SHL     (X       ),     RDCYCS  (SLT4,  4)            ;
110: FUNCTION   FAND1   =   AND     (X       ),     RDCYCS  (NUMFF, 1)            ;
111: FUNCTION   FAND2   =   AND     (X       ),     RDCYCS  (NUMFF, 1)            ;
112: FUNCTION   FCMP1   =   CMPNOM  (X,    GE),     RDCYCS  (THRESH,1)            ;
113: FUNCTION   FCMP2   =   CMPNOM  (X,    GE),     RDCYCS  (THRESH,1)            ;
114: FUNCTION   FCMP3   =   CMPNOM  (X,    GE),     RDCYCS  (THRESH,1)            ;
115: FUNCTION   FCMP4   =   CMPNOM  (X,    GE),     RDCYCS  (THRESH,1)            ;
116: FUNCTION   FRTBL1  =   RDIDX   (TABLE   )                                    ;
117: FUNCTION   FRTBL2  =   RDIDX   (TABLE   )                                    ;
118: FUNCTION   FADD1   =   ADD     (X       ),     QUEUE   (QUEAD1,1)            ;
119: FUNCTION   FADD2   =   ADD     (X       ),     QUEUE   (QUEAD2,1)            ;
120: FUNCTION   FADD3   =   ADD     (X       ),     QUEUE   (QUEAD3,1)            ;
121: FUNCTION   FADD4   =   ADD     (X       ),     QUEUE   (QUEAD4,1)            ;
122: FUNCTION   FADD5   =   ADD     (X       ),     QUEUE   (QUEAD5,1)            ;
123: FUNCTION   FADD6   =   ADD     (X       ),     QUEUE   (QUEAD6,1)            ;
124: FUNCTION   FDIST1  =   DIST    (4       )                                    ;
125: FUNCTION   FDIST2  =   DIST    (4       )                                    ;
126: FUNCTION   FACC    =   ACC     (X       ),     COUNT   (16     )             ;
127: FUNCTION   FDUST   =   WRCYCS  (DUSTB, 1)                                    ;
128: ;
```

```
129: ;****************************************
130: ;
131: ;        DATA MEMORY
132: ;
133: ;----------------------------------------
134: ;
135: MEMORY  NUM8    =       8                                       ;
136: MEMORY  NUMFF    =       0FFH                                    ;
137: MEMORY  SLT1    =       15.     11.     7.      3               ;
138: MEMORY  SLT2    =       14.     10.     6.      2               ;
139: MEMORY  SLT3    =       13.     9.      5.      1               ;
140: MEMORY  SLT4    =       12.     8.      4.      0               ;
141: MEMORY  THRESH  =       AREA    (1      )                       ;
142: MEMORY  DUSTB   =       AREA    (1      )                       ;
143: MEMORY  QUEW    =       AREA    (1      )                       ;
144: MEMORY  QUEE    =       AREA    (1      )                       ;
145: MEMORY  QUE1    =       AREA    (1      )                       ;
146: MEMORY  QUE2    =       AREA    (1      )                       ;
147: MEMORY  QUE3    =       AREA    (1      )                       ;
148: MEMORY  QUE4    =       AREA    (1      )                       ;
149: MEMORY  QUEAD1  =       AREA    (1      )                       ;
150: MEMORY  QUEAD2  =       AREA    (1      )                       ;
151: MEMORY  QUEAD3  =       AREA    (1      )                       ;
152: MEMORY  QUEAD4  =       AREA    (1      )                       ;
153: MEMORY  QUEAD5  =       AREA    (1      )                       ;
154: MEMORY  QUEAD6  =       AREA    (1      )                       ;
155: MEMORY  TABLE   =       0000H.  0001H.  0001H.  0002H.
156:                         0001H.  0002H.  0002H.  0003H.
157:                         0001H.  0002H.  0002H.  0003H.
158:                         0002H.  0003H.  0003H.  0004H.
159:                         0100H.  0101H.  0101H.  0102H.
160:                         0101H.  0102H.  0102H.  0103H.
161:                         0101H.  0102H.  0102H.  0103H.
162:                         0102H.  0103H.  0103H.  0104H.
163:                         0100H.  0101H.  0101H.  0102H.
164:                         0101H.  0102H.  0102H.  0103H.
165:                         0101H.  0102H.  0102H.  0103H.
166:                         0102H.  0103H.  0103H.  0104H.
167:                         0200H.  0201H.  0201H.  0202H.
168:                         0201H.  0202H.  0202H.  0203H.
169:                         0201H.  0202H.  0202H.  0203H.
170:                         0202H.  0203H.  0203H.  0204H.
171:                         0100H.  0101H.  0101H.  0102H.
172:                         0101H.  0102H.  0102H.  0103H.
173:                         0101H.  0102H.  0102H.  0103H.
174:                         0102H.  0103H.  0103H.  0104H.
175:                         0200H.  0201H.  0201H.  0202H.
176:                         0201H.  0202H.  0202H.  0203H.
177:                         0201H.  0202H.  0202H.  0203H.
178:                         0202H.  0203H.  0203H.  0204H.
179:                         0200H.  0201H.  0201H.  0202H.
180:                         0201H.  0202H.  0202H.  0203H.
```

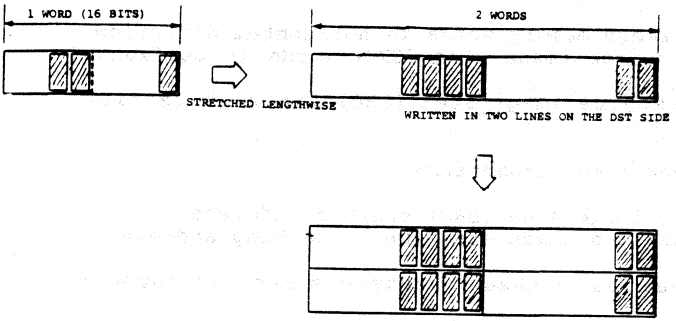
```
181:                                0201H.  0202H.  0202H.  0203H.
182:                                0202H.  0203H.  0203H.  0204H.
183:                                0300H.  0301H.  0301H.  0302H.
184:                                0301H.  0302H.  0302H.  0303H.
185:                                0301H.  0302H.  0302H.  0303H.
186:                                0302H.  0303H.  0303H.  0304H.
187:                                0100H.  0101H.  0101H.  0102H.
188:                                0101H.  0102H.  0102H.  0103H.
189:                                0101H.  0102H.  0102H.  0103H.
190:                                0102H.  0103H.  0103H.  0104H.
191:                                0200H.  0201H.  0201H.  0202H.
192:                                0201H.  0202H.  0202H.  0203H.
193:                                0201H.  0202H.  0202H.  0203H.
194:                                0202H.  0203H.  0203H.  0204H.
195:                                0200H.  0201H.  0201H.  0202H.
196:                                0201H.  0202H.  0202H.  0203H.
197:                                0201H.  0202H.  0202H.  0203H.
198:                                0202H.  0203H.  0203H.  0204H.
199:                                0300H.  0301H.  0301H.  0302H.
200:                                0301H.  0302H.  0302H.  0303H.
201:                                0301H.  0302H.  0302H.  0303H.
202:                                0302H.  0303H.  0303H.  0304H.
203:                                0200H.  0201H.  0201H.  0202H.
204:                                0201H.  0202H.  0202H.  0203H.
205:                                0201H.  0202H.  0202H.  0203H.
206:                                0202H.  0203H.  0203H.  0204H.
207:                                0300H.  0301H.  0301H.  0302H.
208:                                0301H.  0302H.  0302H.  0303H.
209:                                0301H.  0302H.  0302H.  0303H.
210:                                0302H.  0303H.  0303H.  0304H.
211:                                0300H.  0301H.  0301H.  0302H.
212:                                0301H.  0302H.  0302H.  0303H.
213:                                0301H.  0302H.  0302H.  0303H.
214:                                0302H.  0303H.  0303H.  0304H.
215:                                0400H.  0401H.  0401H.  0402H.
216:                                0401H.  0402H.  0402H.  0403H.
217:                                0401H.  0402H.  0402H.  0403H.
218:                                0402H.  0403H.  0403H.  0404H   ;
219: ;
220: ;****************************************
221: ;
222: ;       START
223: ;
224: ;----------------------------------------
225: START                                                       ;
226: ;
227: DATA    EXEC    (IPP,   INT,    THRDAT  )                  ;
228: DATA    EXEC    (IPP,   LSA0,   STARTS  )                  ;
229: DATA    EXEC    (IPP,   LDA0,   STARTD  )                  ;
230: ;
231: END                                                         ;
```

4.4 Simple Double Enlargement

4.4.1 Processing Explained

In simple double enlargement, data is read from the the source image area (SRC) and each bit is duplicated, as shown in Figure 4-10, to double the data in horizontal direction. The data are also written in two consecutive lines of the destination image area (DST) to double them vertically.

Figure 4-10
Simple Double Enlargement

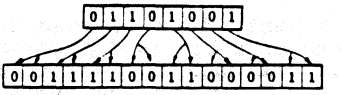


4.4.2 Algorithm

As shown in Figure 4-10, (1) one word is read from the SRC, then (2) the word is separated into high-order 8-bit and low-order 8-bit segments*. The SRC data divided into two 8-bit segments (3) are used in looking up the double enlargement table, in terms of the high-order and low-order segment values, and become two words of 16-bit data (see Figure 4-11). Each of two words of the 16-bit data (4) is written to the contiguous addresses and also in the contiguous lines of the DST. (5) This processing is performed on all SRC data.

* : This procedure is intended to double the data by means of a table lookup. However, if one word (16 bits) were used directly as an address for table lookup, 65,536 locations would be required, exceeding the number of the uPD7281 DM words (512 locations). Therefore, it is necessary to segment the data read from the SRC into 8-bit data. In this case, only 256 locations are required for a lookup table.

Figure 4-11
An Example of Double Horizontal Enlargement



4.4.3 Parameters and Their Applicable Ranges

<Assembler-coded parameters>

L ... Number of image memory words in horizontal direction
H ... Number of source image area (SRC) words in horizontal direction
M ... Number of source image area (SRC) lines in vertical direction

<Start-up token-defined parameters>

STARTS ... Source image area (SRC) starting address
STARTD ... Destination image area (DST) starting address

The allowable values of these parameters are indicated in the table below.

Parameter	Applicable range	(Value set in the example program)
L	0 - 65535	(64)
H	1 - 256	(8)
M	1 - 256	(128)
STARTS	0 - 65535	(0)*
STARTD	0 - 65535	(32)*

* : Although STARTS and STARTD are variables (i.e., addresses), they are given default values since they are used in the assembler DATA statement. When the uPD7281 is started up, the starting addresses of the SRC and DST areas are input as execution tokens.

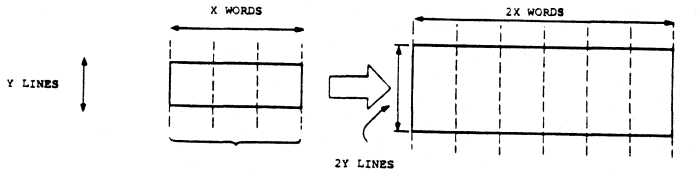
Since no provision is made in this program for switching banks, you should exercise care in setting the values of parameters H and M.

The maximum size of source image area allowed is 256 horizontal words and 256 vertical lines.

<Initial values>

To set initial values, determine the size of the SRC to be expanded horizontally in word units (word boundary), and the size in vertical line units.

Since this is a double enlargement, the size of the DST after enlargement is determined automatically once the size of the SRC is determined.



THE PROCESSING DONE BY WORD BOUNDARIES

4.4.4 Flow Graph Explained

A flow graph for this program is shown in Figure 4-12.

Start-up tokens indicating the starting addresses, STARTS and STARTD, of the SRC and DST areas required in performing the enlargement processing are received from the host computer and sent to the SRC and DST processing parts, respectively.

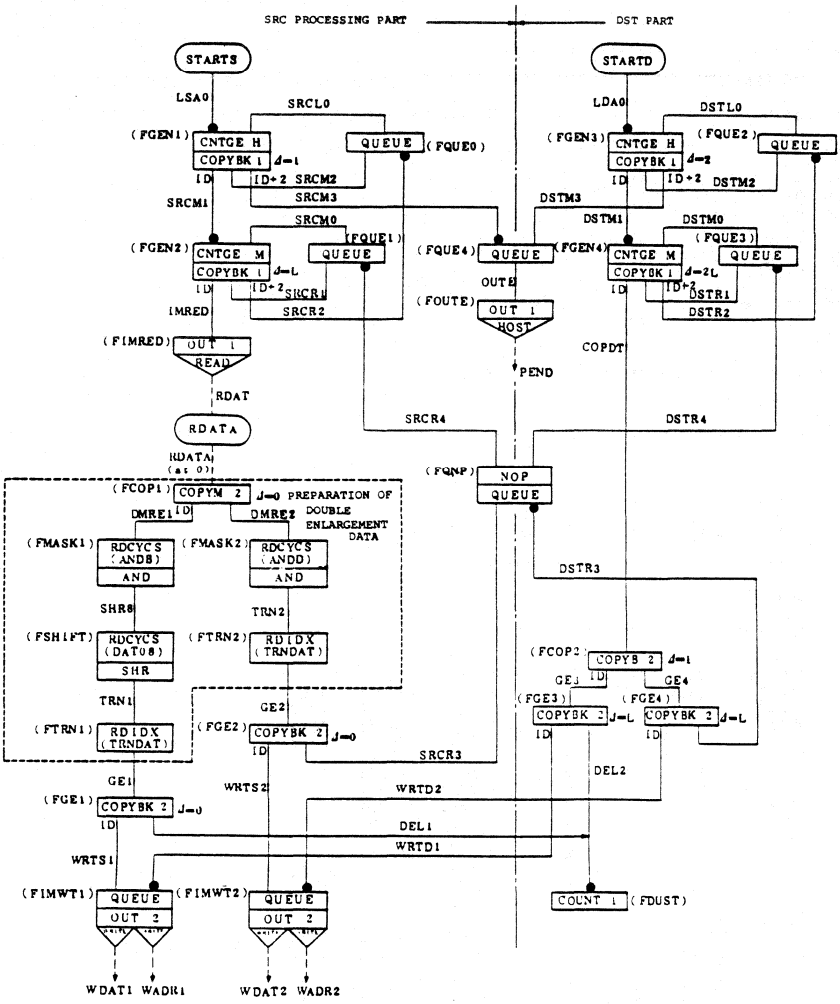
The SRC processing part creates SRC addresses on the basis of the SRC address (STARTS) and reads the contents of these addresses. The SRC addresses are created in FGEN1 and FGEN2 contiguously in the vertical direction starting from STARTS. FIMRED reads the contents of the addresses thus created. Two copies of this SRC data are made for reasons explained above, and the values of high-order eight bits and low-order eight bits are used in looking up the double enlargement table, with the result being DST data. Then, FGE1 and FGE2 make two copies of the DST data to create output data for two lines.

The DST processing part creates DST addresses on the basis of the DST starting address (STARTD).

Four DST addresses, consisting of two contiguous words by two lines, must be created to write one SRC data. First, FGEN3 and FGEN4 create addresses for every other word horizontally and every other line vertically, starting from STARTD. Upon receipt of these addresses, FCOP2 creates addresses for two contiguous words. Further, FGE3 and FGE4 create two lines of addresses for two contiguous words as the addresses for writing the DST data generated in the SRC processing part.

The writing of the DST data is done in FIMWT1 and FIMWT2.

Figure 4.12
A flow Chart of a Simple Double Enlargement



FIMWT1 writes the DST (enlargement) data representing the high-order eight bits of the data read from SRC, and FIMWT2 writes the low-order eight bits .

4.4.5 Assembler Source Listing

```
1: ;****************************************
2: ;
3: ;       ENLARGE X2
4: ;
5: ;----------------------------------------
6: ;
7: MODULE  IPP     =       8       ;
8: ;
9: EQUATE  L       =       64      ;
10: EQUATE  H       =       8       ;
11: EQUATE  M       =       128     ;
12: ;
13: EQUATE  HOST    =       0       ;
14: EQUATE  READ    =       4       ;
15: EQUATE  WRITE   =       5       ;
16: ;
17: EQUATE  STARTS  =       0       ;
18: EQUATE  STARTD  =       32      ;
19: ;
20: ;
21: ;****************************************
22: ;
23: ;       INPUT-OUTPUT
24: ;
25: ;----------------------------------------
26: ;
27: ;
28: ;
29: INPUT   LSA0,   LDA0,   RDATA AT 0                      ;
30: ;
31: OUTPUT  RDAT,   WDAT1,  WADR1,  WDAT2,  WADR2,  PEND    ;
32: ;
33: ;****************************************
34: ;
35: ;       LINK TABLE
36: ;
37: ;----------------------------------------
38: ;
39: ;
40: LINK    SRCM1,  SRCM2,  SRCM3   =       FGEN1   (SRCL0, LSA0    )       ;
41: LINK    IMRED,  SRCR1,  SRCR2   =       FGEN2   (SRCM0, SRCM1   )       ;
42: LINK    SRCL0                   =       FQUE0   (SRCM2, SRCR2   )       ;
43: LINK    OUTE                    =       FQUE4   (DSTM3, SRCM3   )       ;
44: LINK    SRCM0                   =       FQUE1   (SRCR1, SRCR4   )       ;
45: LINK    RDAT                    =       FIMRED  (IMRED          )       ;
46: LINK    DMRE1,  DMRE2           =       FCOP1   (RDATA          )       ;
47: LINK    SHR8                    =       FMASK1  (DMRE1          )       ;
48: LINK    TRN2                    =       FMASK2  (DMRE2          )       ;
49: LINK    TRN1                    =       FSHIFT  (SHR8           )       ;
50: LINK    GE1                     =       FTRN1   (TRN1           )       ;
51: LINK    WRTS1,  DEL1            =       FGE1    (GE1            )       ;
52: LINK    WDAT1,  WADR1           =       FIMWT1  (WRTS1, WRTD1   )       ;
53: LINK                            =       FDUST   (       ,DEL1   )       ;
54: LINK    GE2                     =       FTRN2   (TRN2           )       ;
55: LINK    WRTS2,  SRCR3           =       FGE2    (GE2            )       ;
56: LINK    WDAT2,  WADR2           =       FIMWT2  (WRTS2, WRTD2   )       ;
57: LINK    SRCR4,  DSTR4           =       FQNP    (SRCR3, DSTR3   )       ;
58: ;
59: LINK    DSTM1,  DSTM2,  DSTM3   =       FGEN3   (DSTL0, LDA0    )       ;
60: LINK    COPDT,  DSTR1,  DSTR2   =       FGEN4   (DSTM0, DSTM1   )       ;
```

```
 61: LINK    DSTL0                       =       FQUE2   (DSTM2, DSTR2   )
 62: LINK    DSTM0                       =       FQUE3   (DSTR1, DSTR4   )
 63: LINK    GE3,    GE4                 =       FCOP2   (COPDT          )
 64: LINK    WRTD1,  DEL2                =       FGE3    (GE3            )
 65: LINK    WRTD2,  DSTR3               =       FGE4    (GE4            )
 66: LINK                                =       FDUST   (       .DEL2   )
 67: LINK    PEND                        =       FOUTE   (OUTE           )
 68: ;
 69: ;
 70: ;*************************************
 71: ;
 72: ;       FUNCTION TABLE
 73: ;
 74: ;-------------------------------------
 75: ;
 76: FUNCTION    FIMRED  =   OUT1    (READ,  0)
 77: FUNCTION    FIMWT1  =   OUT2    (WRITE, 20H, 0),QUEUE   (QUEW1, 2)
 78: FUNCTION    FIMWT2  =   OUT2    (WRTIE, 20H, 0),QUEUE   (QUEW2, 2)
 79: FUNCTION    FOUTE   =   OUT1    (HOST,  0)
 80: FUNCTION    FGEN1   =   COPYBK  (1,     1),     CNTGE   (H      )
 81: FUNCTION    FGEN2   =   COPYBK  (1,     L),     CNTGE   (M      )
 82: FUNCTION    FGEN3   =   COPYBK  (1,     2),     CNTGE   (H      )
 83: FUNCTION    FGEN4   =   COPYBK  (1,   2*L),     CNTGE   (M      )
 84: FUNCTION    FMASK1  =   AND     (X      ),      RDCYCS  (ANDB,  1)
 85: FUNCTION    FMASK2  =   AND     (X      ),      RDCYCS  (ANDD,  1)
 86: FUNCTION    FSHIFT  =   SHR     (X      ),      RDCYCS  (DAT08, 1)
 87: FUNCTION    FTRN1   =   RDIDX   (TRNDAT )
 88: FUNCTION    FTRN2   =   RDIDX   (TRNDAT )
 89: FUNCTION    FCOP1   =   COPYM   (2,     0)
 90: FUNCTION    FCOP2   =   COPYM   (2,     1)
 91: FUNCTION    FGE1    =   COPYBK  (2,     0)
 92: FUNCTION    FGE2    =   COPYBK  (2,     0)
 93: FUNCTION    FGE3    =   COPYBK  (2,     L)
 94: FUNCTION    FGE4    =   COPYBK  (2,     L)
 95: FUNCTION    FQNP    =   NOP     (XX     ),      QUEUE   (QUEN,  1)
 96: FUNCTION    FQUE0   =   QUEUE   (QUE0,  1)
 97: FUNCTION    FQUE1   =   QUEUE   (QUE1,  1)
 98: FUNCTION    FQUE2   =   QUEUE   (QUE2,  1)
 99: FUNCTION    FQUE3   =   QUEUE   (QUE3,  1)
100: FUNCTION    FQUE4   =   QUEUE   (QUE4,  1)
101: FUNCTION    FDUST   =   COUNT   (1      )
102: ;
103: ;
104: ;*************************************
105: ;
106: ;       DATA MEMORY
107: ;
108: ;-------------------------------------
109: ;
110: MEMORY  QUE0    =       AREA    (1      )               ;
111: MEMORY  QUE1    =       AREA    (1      )               ;
112: MEMORY  QUE2    =       AREA    (1      )               ;
113: MEMORY  QUE3    =       AREA    (1      )               ;
114: MEMORY  QUE4    =       AREA    (1      )               ;
115: MEMORY  QUEN    =       AREA    (1      )               ;
116: MEMORY  QUEW1   =       AREA    (2      )               ;
117: MEMORY  QUEW2   =       AREA    (2      )               ;
118: MEMORY  DAT08   =       8                               ;
119: MEMORY  ANDB    =       0FF00H                          ;
120: MEMORY  ANDD    =       000FFH                          ;
```

```
121: MEMORY  TRNDAT  =        00000H,00003H,0000CH,0000FH,
122:                          00030H,00033H,0003CH,0003FH,
123:                          000C0H,000C3H,000CCH,000CFH,
124:                          000F0H,000F3H,000FCH,000FFH,
125:                          00300H,00303H,0030CH,0030FH,
126:                          00330H,00333H,0033CH,0033FH,
127:                          003C0H,003C3H,003CCH,003CFH,
128:                          003F0H,003F3H,003FCH,003FFH,
129:                          00C00H,00C03H,00C0CH,00C0FH,
130:                          00C30H,00C33H,00C3CH,00C3FH,
131:                          00CC0H,00CC3H,00CCCH,00CCFH,
132:                          00CF0H,00CF3H,00CFCH,00CFFH,
133:                          00F00H,00F03H,00F0CH,00F0FH,
134:                          00F30H,00F33H,00F3CH,00F3FH,
135:                          00FC0H,00FC3H,00FCCH,00FCFH,
136:                          00FF0H,00FF3H,00FFCH,00FFFH,
137:                          03000H,03003H,0300CH,0300FH,
138:                          03030H,03033H,0303CH,0303FH,
139:                          030C0H,030C3H,030CCH,030CFH,
140:                          030F0H,030F3H,030FCH,030FFH,
141:                          03300H,03303H,0330CH,0330FH,
142:                          03330H,03333H,0333CH,0333FH,
143:                          033C0H,033C3H,033CCH,033CFH,
144:                          033F0H,033F3H,033FCH,033FFH,
145:                          03C00H,03C03H,03C0CH,03C0FH,
146:                          03C30H,03C33H,03C3CH,03C3FH,
147:                          03CC0H,03CC3H,03CCCH,03CCFH,
148:                          03CF0H,03CF3H,03CFCH,03CFFH,
149:                          03F00H,03F03H,03F0CH,03F0FH,
150:                          03F30H,03F33H,03F3CH,03F3FH,
151:                          03FC0H,03FC3H,03FCCH,03FCFH,
152:                          03FF0H,03FF3H,03FFCH,03FFFH,
153:                          0C000H,0C003H,0C00CH,0C00FH,
154:                          0C030H,0C033H,0C03CH,0C03FH,
155:                          0C0C0H,0C0C3H,0C0CCH,0C0CFH,
156:                          0C0F0H,0C0F3H,0C0FCH,0C0FFH,
157:                          0C300H,0C303H,0C30CH,0C30FH,
158:                          0C330H,0C333H,0C33CH,0C33FH,
159:                          0C3C0H,0C3C3H,0C3CCH,0C3CFH,
160:                          0C3F0H,0C3F3H,0C3FCH,0C3FFH,
161:                          0CC00H,0CC03H,0CC0CH,0CC0FH,
162:                          0CC30H,0CC33H,0CC3CH,0CC3FH,
163:                          0CCC0H,0CCC3H,0CCCCH,0CCCFH,
164:                          0CCF0H,0CCF3H,0CCFCH,0CCFFH,
165:                          0CF00H,0CF03H,0CF0CH,0CF0FH,
166:                          0CF30H,0CF33H,0CF3CH,0CF3FH,
167:                          0CFC0H,0CFC3H,0CFCCH,0CFCFH,
168:                          0CFF0H,0CFF3H,0CFFCH,0CFFFH,
169:                          0F000H,0F003H,0F00CH,0F00FH,
170:                          0F030H,0F033H,0F03CH,0F03FH,
171:                          0F0C0H,0F0C3H,0F0CCH,0F0CFH,
172:                          0F0F0H,0F0F3H,0F0FCH,0F0FFH,
173:                          0F300H,0F303H,0F30CH,0F30FH,
174:                          0F330H,0F333H,0F33CH,0F33FH,
175:                          0F3C0H,0F3C3H,0F3CCH,0F3CFH,
176:                          0F3F0H,0F3F3H,0F3FCH,0F3FFH,
177:                          0FC00H,0FC03H,0FC0CH,0FC0FH,
178:                          0FC30H,0FC33H,0FC3CH,0FC3FH,
179:                          0FCC0H,0FCC3H,0FCCCH,0FCCFH,
180:                          0FCF0H,0FCF3H,0FCFCH,0FCFFH,
```

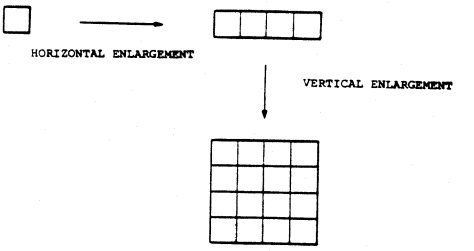
```
181:                            0FF00H,0FF03H,0FF0CH,0FF0FH,
182:                            0FF30H,0FF33H,0FF3CH,0FF3FH,
183:                            0FFC0H,0FFC3H,0FFCCH,0FFCFH,
184:                            0FFF0H,0FFF3H,0FFFCH,0FFFFH     ;
185: ;
186: ;
187: ;****************************************
188: ;
189: ;       START
190: ;
191: ;----------------------------------------
192: ;
193: START                                                     ;
194: ;
195: DATA    EXEC    (IPP,   LSA0,   STARTS  )               ;
196: DATA    EXEC    (IPP,   LDA0,   STARTD  )               ;
197: ;
198: END
```

4.5 Simple Quadruple Enlargement

4.5.1 Processing Explained

The simple quadruple enlargement method, shown in Figure 4-13, simply expands source image area (SRC) data fourfold horizontally and vertically. Similar to the case of the simple double enlargement method discussed in Section 4.3, in this method four output bits are assigned to each input bit, a fourfold enlargement is performed horizontally, followed by the writing of the data in four contiguous lines.

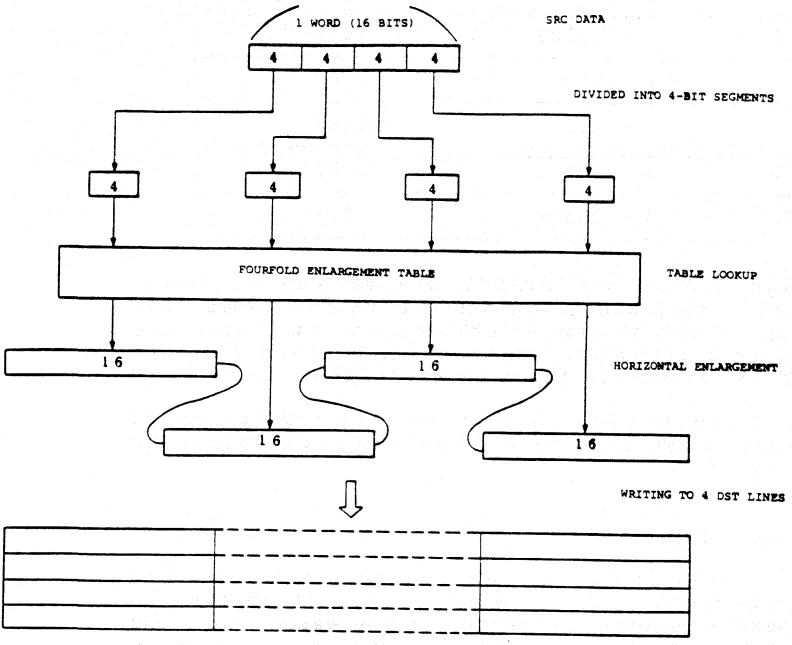
Figure 4-13
Simple Quadruple Enlargement



4.5.2 Algorithm

The algorithm employed in this method is similar to that used in the simple double enlargement method, as shown in Figure 4-14. It differs in the fact that, to achieve the fourfold enlargement, each SRC word (16 bits) is segmented into 4 bits, and a fourfold enlargement table is used for each of these 4-bit segments to obtain 4 DST words. First, for a given SRC data word four DST data words representing a horizontal enlargement, are obtained. These four words are then copied to four lines in the vertical direction. This completes processing of an SRC data word.

Figure 4-14
Algorithm for Simple Quadruple Enlargement



4.5.3 Parameters and Their Applicable Ranges

<Assembler-coded parameters>

L ... Number of image memory words in horizontal direction
H ... Number of source image area (SRC) words in horizontal direction
V ... Number of source image area (SRC) lines in vertical direction

<Start-up token-defined parameters>

STARTS ... Source image area (SRC) starting address
STARTD ... Destination image area (DST) starting address

The allowable values of these parameters are indicated in the table below.

Parameter	Applicable range	(Value set in the example program)
L	0 - 65535	(64)
H	1 - 256	(8)
V	1 - 256	(128)
STARTS	0 - 65535	(0)*
STARTD	0 - 65535	(32)*

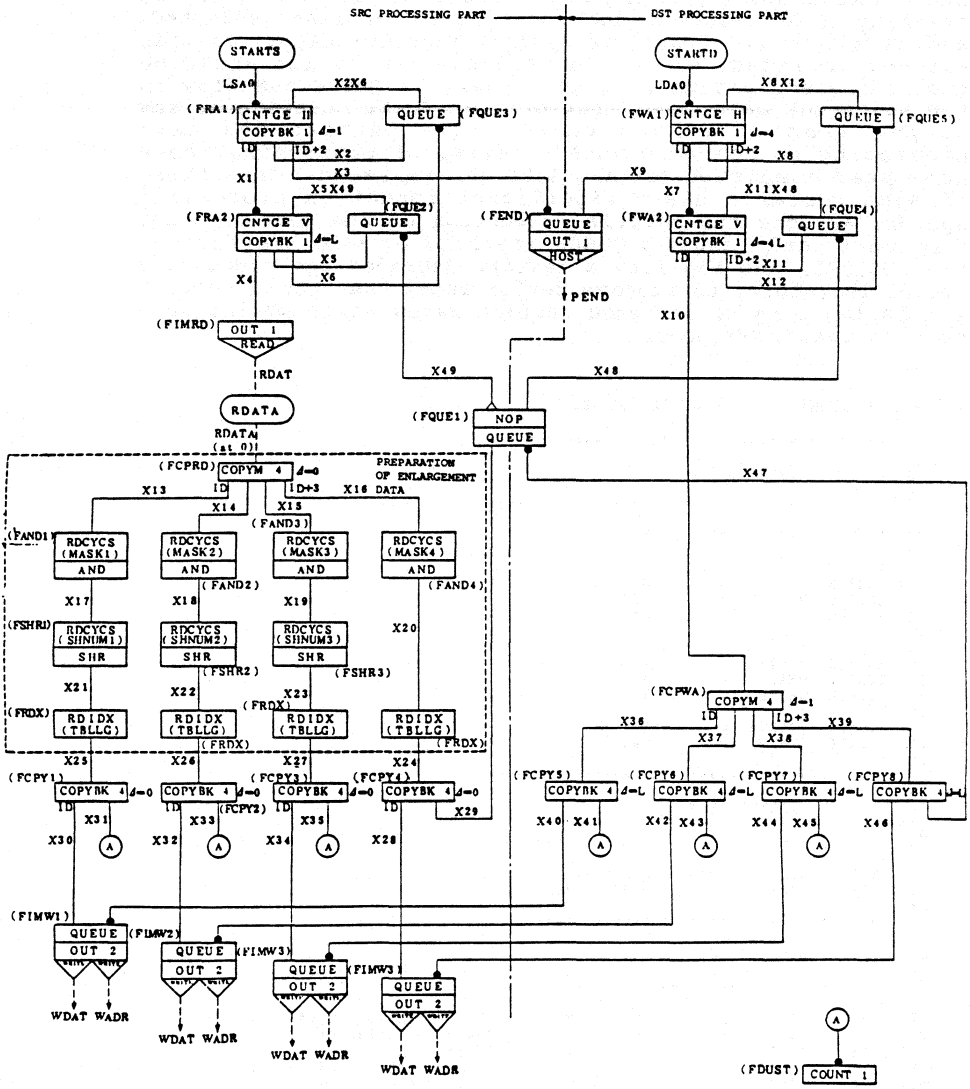
* : Although STARTS and STARTD are variables (i.e., addresses), they are given default values since they are used in the assembler DATA statement. When the uPD7281 is started up, the starting addresses of the SRC and DST areas are input as execution tokens.

Since no provision is made in this program for switching banks, you should exercise care in setting the values of parameters H and M.

4.5.4 Flow Graph Explained

Shown in Figure 4-15 is a flow graph for this program. This flow graph is basically the same as that provided for the double enlargement program. However, to the extent that the division of words into 4-bit segments is required to produce fourfold enlargement, this flow graph is more complicated.

Figure 4.15
A Flow Graph of the Simple Quadruple Enlargement Method



4.5.5 Tips on Preparing Flow Graphs

This program uses an FQUE1 node (in the center of Figure 4.15) to synchronize the generation of SRC addresses and DST basic addresses (i.e., the addresses that form the basis of creating 16 addresses in DST for each SRC address generated) and to adjust the amount of uPD7281 internal data. If this was not provided, SRC and DST address generation would be uncoordinated, giving rise to a possible QUEUE overflow in the write-out part. The input arc for FQUE1 constitutes the output from FCPY4 and FCPY8; this permits the next processing to commence when a series of processing tasks have been completed (or about to be completed). The timing of such restarts has a significant impact on processing speeds, as does the parallel execution of the job. Too fast a restart could cause a QUEUE overflow or a GQ overflow on the uPD7281, and too slow a restart degrades the processing speed. Therefore, to improve performance, one should conduct simulation runs to increase restart rates while making sure that no overflows occur.

4.5.6 Assembler Source Listing

```
;*************************************
;
;       ENLARGE  X4
;
;-------------------------------------
;
MODULE  IPP     =       8       ;
;
;
EQUATE  H       =       8       ;
EQUATE  V       =       128     ;
EQUATE  L       =       64      ;
;
;
EQUATE  HOST    =       0       ;
EQUATE  READ    =       4       ;
EQUATE  WRITE   =       5       ;
;
EQUATE  STARTS  =       0       ;
EQUATE  STARTD  =       20H     ;
;
;*************************************
;
;       INPUT-OUTPUT
;
;-------------------------------------
;
INPUT   LSA0.   LDA0.   RDATA AT 0              ;
OUTPUT  RDAT.   WDAT.   WADR.   PEND            ;
;
;*************************************
;
;       LINK TABLE
;
;-------------------------------------
;
LINK    X1.     X2.     X3      =       FRA1    (X2X6.  LSA0    )       ;
LINK    X4.     X5.     X6      =       FRA2    (X5X49. X1      )       ;
LINK    X2X6                    =       FQUE3   (X2.    X6      )       ;
LINK    PEND                    =       FEND    (X9.    X3      )       ;
LINK    RDAT                    =       FIMRD   (X4             )       ;
LINK    X5X49                   =       FQUE2   (X5.    X49     )       ;
;
```

```
 44: LINK     X13,  X14,  X15,  X16   =       FCPRD   (RDATA          )       ;
 45: LINK     X17                     =       FAND1   (X13            )       ;
 46: LINK     X18                     =       FAND2   (X14            )       ;
 47: LINK     X19                     =       FAND3   (X15            )       ;
 48: LINK     X20                     =       FAND4   (X16            )       ;
 49: LINK     X21                     =       FSHR1   (X17            )       ;
 50: LINK     X22                     =       FSHR2   (X18            )       ;
 51: LINK     X23                     =       FSHR3   (X19            )       ;
 52: LINK     X24                     =       FRDX    (X20            )       ;
 53: LINK     X25                     =       FRDX    (X21            )       ;
 54: LINK     X26                     =       FRDX    (X22            )       ;
 55: LINK     X27                     =       FRDX    (X23            )       ;
 56: LINK     X28,     X29            =       FCPY4   (X24            )       ;
 57: LINK     X30,     X31            =       FCPY1   (X25            )       ;
 58: LINK     X32,     X33            =       FCPY2   (X26            )       ;
 59: LINK     X34,     X35            =       FCPY3   (X27            )       ;
 60: LINK     WDAT,    WADR           =       FIMW4   (X28,    X46    )       ;
 61: LINK     WDAT,    WADR           =       FIMW1   (X30,    X40    )       ;
 62: LINK     WDAT,    WADR           =       FIMW2   (X32,    X42    )       ;
 63: LINK     WDAT,    WADR           =       FIMW3   (X34,    X44    )       ;
 64: LINK                             =       FDUST   (        ,X31   )       ;
 65: LINK                             =       FDUST   (        ,X33   )       ;
 66: LINK                             =       FDUST   (        ,X35   )       ;
 67: ;
 68: LINK     X7,      X8,      X9    =       FWA1    (X8X12, LDA0    )       ;
 69: LINK     X10,     X11,     X12   =       FWA2    (X11X48, X7     )       ;
 70: LINK     X8X12                  =       FQUE5   (X8,     X12    )       ;
 71: LINK     X11X48                 =       FQUE4   (X11,    X48    )       ;
 72: LINK     X36,  X37,  X38,  X39  =       FCPWA   (X10            )       ;
 73: LINK     X40,     X41           =       FCPY5   (X36            )       ;
 74: LINK     X42,     X43           =       FCPY6   (X37            )       ;
 75: LINK     X44,     X45           =       FCPY7   (X38            )       ;
 76: LINK     X46,     X47           =       FCPY8   (X39            )       ;
 77: LINK     X48,     X49           =       FQUE1   (X29     ,X47   )       ;
 78: LINK                             =       FDUST   (        ,X41   )       ;
 79: LINK                             =       FDUST   (        ,X43   )       ;
 80: LINK                             =       FDUST   (        ,X45   )       ;
 81: ;
 82: ;*************************************
 83: ;
 84: ;        FUNCTION TABLE
 85: ;
 86: ;-------------------------------------
 87: ;
 88: FUNCTION     FRA1    =    COPYBK (1,      1),     CNTGE   (H          )       ;
 89: FUNCTION     FQUE3   =    QUEUE  (DQUE3, 1)                                   ;
 90: FUNCTION     FRA2    =    COPYBK (1,      L),     CNTGE   (V          )       ;
 91: FUNCTION     FQUE2   =    QUEUE  (DQUE2, 1)                                   ;
 92: FUNCTION     FIMRD   =    OUT1   (READ,  0)                                   ;
 93: FUNCTION     FEND    =    OUT1   (HOST,  0),     QUEUE   (ENDQUE, 1)          ;
 94: ;
 95: FUNCTION     FCPRD   =    COPYM  (4,     0)                                   ;
 96: FUNCTION     FAND1   =    AND,                   RDCYCS (MASK1,  1)           ;
 97: FUNCTION     FAND2   =    AND,                   RDCYCS (MASK2,  1)           ;
 98: FUNCTION     FAND3   =    AND,                   RDCYCS (MASK3,  1)           ;
 99: FUNCTION     FAND4   =    AND,                   RDCYCS (MASK4,  1)           ;
100: FUNCTION     FSHR1   =    SHR,                   RDCYCS (SHNUM1, 1)           ;
101: FUNCTION     FSHR2   =    SHR,                   RDCYCS (SHNUM2, 1)           ;
102: FUNCTION     FSHR3   =    SHR,                   RDCYCS (SHNUM3, 1)           ;
103: FUNCTION     FRDX    =    RDIDX  (TBLLG   )                                   ;
104: FUNCTION     FCPY1   =    COPYBK (4,     0)                                   ;
105: FUNCTION     FCPY2   =    COPYBK (4,     0)                                   ;
106: FUNCTION     FCPY3   =    COPYBK (4,     0)                                   ;
107: FUNCTION     FCPY4   =    COPYBK (4,     0)                                   ;
108: FUNCTION     FIMW1   =    OUT2 (WRITE, 20H, 0),  QUEUE   (QUEWR1, 4)          ;
109: FUNCTION     FIMW2   =    OUT2 (WRITE, 20H, 0),  QUEUE   (QUEWR2, 4)          ;
110: FUNCTION     FIMW3   =    OUT2 (WRITE, 20H, 0),  QUEUE   (QUEWR3, 4)          ;
```

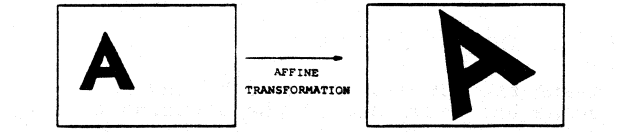
```
111: FUNCTION    FIMW4   =    OUT2 (WRITE, 20H, 0),  QUEUE  (QUEWR4, 4)      ;
112: FUNCTION    FDUST   =    COUNT  (1         )                            ;
113: ;
114: FUNCTION    FWA1    =    COPYBK (1,     4),    CNTGE  (H         )     ;
115: FUNCTION    FWA2    =    COPYBK (1,   4*L),    CNTGE  (V         )     ;
116: FUNCTION    FQUE5   =    QUEUE  (DQUE5, 1)                            ;
117: FUNCTION    FQUE4   =    QUEUE  (DQUE4, 1)                            ;
118: FUNCTION    FQUE1   =    NOP    (XY      ),    QUEUE  (DQUE1,  1)     ;
119: ;
120: FUNCTION    FCPWA   =    COPYM  (4,     L)                            ;
121: FUNCTION    FCPY5   =    COPYBK (4,     L)                            ;
122: FUNCTION    FCPY6   =    COPYBK (4,     L)                            ;
123: FUNCTION    FCPY7   =    COPYBK (4,     L)                            ;
124: FUNCTION    FCPY8   =    COPYBK (4,     L)                            ;
125: ;
126: ;*************************************
127: ;
128: ;       DATA MEMORY
129: ;
130: ;-------------------------------------
131: ;
132: MEMORY  DQUE1   =       AREA    (1       )      ;
133: MEMORY  DQUE2   =       AREA    (1       )      ;
134: MEMORY  DQUE3   =       AREA    (1       )      ;
135: MEMORY  DQUE4   =       AREA    (1       )      ;
136: MEMORY  DQUE5   =       AREA    (1       )      ;
137: MEMORY  ENDQUE  =       AREA    (1       )      ;
138: MEMORY  MASK1   =       0F000H                  ;
139: MEMORY  MASK2   =       0F00H                   ;
140: MEMORY  MASK3   =       00F0H                   ;
141: MEMORY  MASK4   =       000FH                   ;
142: MEMORY  SHNUM1  =       12                      ;
143: MEMORY  SHNUM2  =       8                       ;
144: MEMORY  SHNUM3  =       4                       ;
145: MEMORY  TBLLG.  =       00000H,0000FH,000F0H,000FFH,00F00H,00F0FH,
146:                         00FF0H,00FFFH,0F000H,0F00FH,0F0F0H,0F0FFH,
147:                         0FF00H,0FF0FH,0FFF0H,0FFFFH      ;
148: MEMORY  QUEWR1  =       AREA    (4       )      ;
149: MEMORY  QUEWR2  =       AREA    (4       )      ;
150: MEMORY  QUEWR3  =       AREA    (4       )      ;
151: MEMORY  QUEWR4  =       AREA    (4       )      ;
152: ;
153: ;*************************************
154: ;
155: ;       START
156: ;
157: ;-------------------------------------
158: ;
159: START                                           ;
160: ;
161: ;
162: DATA    EXEC    (IPP,   LSA0,   STARTS  )      ;
163: DATA    EXEC    (IPP,   LDA0,   STARTD  )      ;
164: ;
165: END                                             ;
```

Chapter 5

Affine Transformation

5.1 Processing Explained

The affine transformation provides a versatile means of performing image transformations. It can be used for screen enlargement, reduction, rotation, and displacement simultaneously.

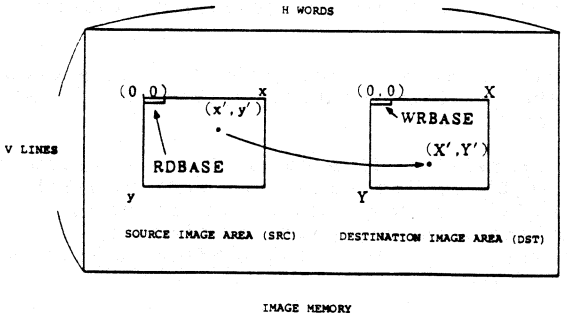


However, this technique takes longer to execute than a program utilizing double enlargement, quadruple enlargement, 90 degree rotation, or some other specific algorithms. Therefore, the affine transformation should be used only when required by the application.

5.2 Algorithm

<Logical coordinates and physical addresses>

Figure 5-1
Logical Coordinates and Physical Addresses



An image memory may be thought of as being comprised of H words horizontally and V lines vertically (i.e., 16 x H horizontal dots and V vertical dots). In this image memory are two image areas (source image area "SRC" and destination image area "DST). These graphic image areas have logical coordinates (x,y) and (X,Y), respectively, as shown in Figure 5-1. Further, the origin (0,0) of the logical coordinates (x,y) corresponds with the MSB of physical address RD BASE, and the origin (0,0) of the logical coordinates (X,Y) corresponds with the MSB of physical address WR BASE*. In this case the coordinates (x',y') of an arbitrary point on SRC are physically:

$$\begin{aligned} \text{Physical address} &\quad \text{RDBASE} + y' \cdot H + [\, x' / 16 \,] \\ \text{Bit position} &\quad 15 - R(\, x' / 16 \,) \end{aligned} \qquad (1)$$

where [a/b] means the division of a by b, and
R(a/b) means the remainder of the division of a by b.

* : The MSB correspondence differs for the NEC graphic display controllers uPD7220 and uPD7220A.

Similarly, the coordinates (X',Y') correspond with:

$$\begin{aligned} \text{Physical address} &\quad \text{WRBASE} + Y' \cdot H + [\, X' / 16 \,\} \\ \text{Bit position} &\quad 15 - R(\, X' / 16 \,) \end{aligned} \qquad (2)$$

In the affine transformation used in this program, the DST logical coordinates (X',Y') and the SRC logical coordinates (x',y') are related to each other by the following formulas:

$$\begin{aligned} x' &= aX' + bY' + c \\ y' &= dX' + eY' + f \end{aligned} \qquad (3)$$

where a, b, c, d, e, and f are parameters that determine whether a given transformation is an enlargement, reduction, rotation, or displacement.

Note: The affine transformation formulas used in this program specify the coordinates before transformation in terms of coordinates after transformation.

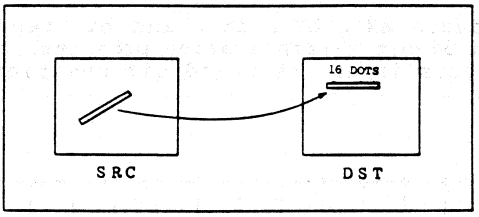
In this program the logical coordinates (X',Y') are allowed to vary from the point (0,0) to the point (max.(X),max.(Y)). The logical coordinates (x',y') corresponding to each of the coordinates are calculated, and the bits for these coordinates are read from SRC and written to DST. Information is written to DST in word units (16 bits).

<Process Flow>

This program assumes that a DST is a 256 x 256 dots word boundary area.

First, (X',Y') is set to (0,0). Then the coordinates (x',y') corresponding to the variation of (X',Y') from the origin to (256,0) are calculated. However, because all 256 values cannot be calculated at once, 256 is divided into 16 segments and 16 dots are calculated at a time. Since Y does not vary during this process, the amount of computation required is cut down by calculating bY' + c and eY' + f in Equation (3) only once for the processing of the first line and by adding these values to aX' and dX'.

Actual physical addresses and bit positions are calculated for each 16 sets of (x',y') values obtained by calculations on one word (16 bits) of DST. SRC data are read using these physical addresses, and the desired bits are extracted using the values for bit positions.

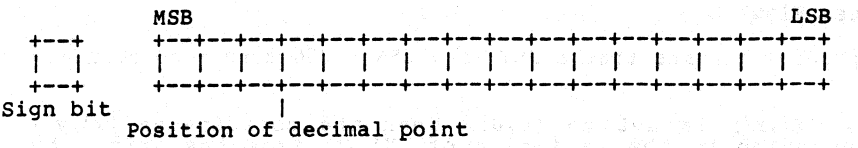


The 16-bit data obtained in this manner are packed into a word and written in DST. Since one DST line has 256 dots, writing 16 words completes the processing of one line. This process is repeated line by line for as many lines as there are lines in DST to complete the processing of one screen image.

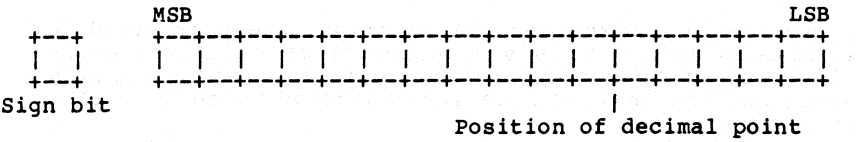
<Precision of Computation>

In multiplying 16 bits by 16 bits in the uPD7281, it is possible to select the high-order 16 bits and/or low-order 16 bits from the results. This program selects the high-order 16 bits only. For this reason the following scheme is employed in calculating the values of parameters a, b, c, d, e, and f, and coordinates (X',Y') and (x',y'):

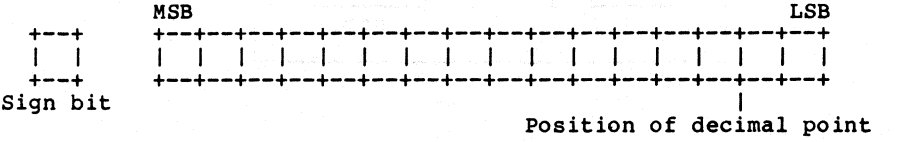
(a) The parameters a, b, d, and e have a 16-bit precision (17 bits including the sign bit) with the decimal point occurring in the 14th bit position.



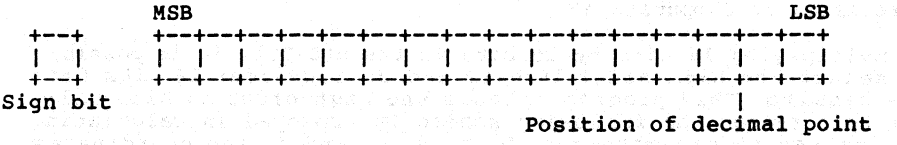
(b) Logically, the coordinates (X',Y') are allowed to increment by one at a time over DST, but the placement of the decimal point for X' and Y' at the sixth bit position from the LSB actually makes them increment by 32 at a time.



(c) Since the products aX', bY', dX', and eY' represent the high-order 16 bits of 32-bit multiplication products, the decimal point in these products is at the third bit position from the LSB.



(d) Consequently, to be consistent with the precision of their preceding terms, the parameters c and f assume values with the decimal point set at the third bit position from the LSB.



(e) As a result, the values of (x',y') have the decimal point set at the third bit position from the LSB. Therefore, a given value is reduced to an integer by performing a 2-bit right shift (note).

Note: (x',y') values have a precision of two bits below the decimal point. Therefore, if a given value was 0.75 or 1.75, retaining a value of 1 or 2 instead of rounding off the bits below the decimal point through bit shifting would result in a more accurate screen image. For this reason, 0.5 is added to the values of x' and y' in this program. After this, a 2-bit right shift is performed. The addition of 0.5 to these parameters is done by adding 0.5 to parameters c and f.

5.3 Parameters and Their Applicable Ranges

<Assembler-coded parameters>

L ... Number of image memory words in horizontal direction
H ... Number of destination image area (DST) words in horizontal direction
V ... Number of destination image area (DST) lines in vertical direction

<Start-up token-defined parameters>

SETA ..... Affine transformation parameter a
SETB ..... Affine transformation parameter b
SETC ..... Affine transformation parameter c
SETD ..... Affine transformation parameter d
SETE ..... Affine transformation parameter e
SETF ..... Affine transformation parameter f
RDBASE ... Source image area (SRC) starting address
WRBASE ... Destination image area (DST) starting address

The allowable values of these parameters are indicated in the table below.

Parameter	Applicable range	(Value set in the example program)
L	0 - 65535	(64)
H	1 - 256	(16)
V	1 - 256	(256)
SETA	-7.999 - +7.999	(0)*
SETB	-7.999 - +7.999	(0)*
SETC	-2047.9 - +2047.9	(0)*
SETD	-7.999 - +7.999	(0)*
SETE	-7.999 - +7.999	(0)*
SETF	-2047.9 - +2047.9	(0)*
RDBASE	0 - 65535	(0)*
WRBASE	0 - 65535	(32)*

* : Although STARTS and STARTD are variables (i.e., addresses), they are given default values since they are used in the assembler DATA statement. When the uPD7281 is started up, the starting addresses of the SRC and DST areas are input as execution tokens.

Since no provision is made in this program for switching banks, you should exercise care in setting the values for parameters H and V.

<Input Tokens>

This program provides ten input tokens. The first and last of these tokens require special explanations.

The first token is a command reset. It clears the ACC register of the PU in the uPD7281 to 0. Bear in mind that, when this token is input into the uPD7281, all programs in execution at the time are erased. Make sure, when inputting this token, that no program is running within the uPD7281.

The last token is a program start-up token. When this token is input, the uPD7281 starts operation. Since the data value for this token is also used as the line counter for the number of DST lines it should always be input as 0.

5.4 Flow Graph Explained

<Explanation of Main Nodes>

FGENE : Controls the counting of the number of DST lines. If the CNTGE instruction parameter is set to 256, 256 lines of images are processed.

FBR, FCR, FW1 : Calculates bY' + c and writes the results to W1BUF. This value needs to be calculated only once for each value of Y', and it can be read from W1BUF at any time when an addition to aX' is performed.

FBRD, FCRD, FW2 : Calculates eY' + f and writes the results to W2BUF.

FGEN2 : Increments X' from 0 to max.(X) and outputs the results as a token. Since not all values for 0 through max.(X) can be output at once, 16 values are output at a time.

FTT, PNYD, PSH : Calculates dX' + eY' + f and performs a right shift to round off values below the decimal point.

FTTD, PNY, PSHD : Calculates aX' + by' + c and performs a right shift to round off values below the decimal point.

PIMX, FSA, FADDX, FADDZ : From a computed value (x',y'), calculates the physical address of the given coordinates.

FREAD : Reads SRC data.

PMY, FSUB : Calculates bit positions from x' values.

PBG1, PSHL3, PACC : Extracts bits from the bit positions calculated by PMY and FSUB from data that were read, and sets these bits in their proper positions in a word of DST data.

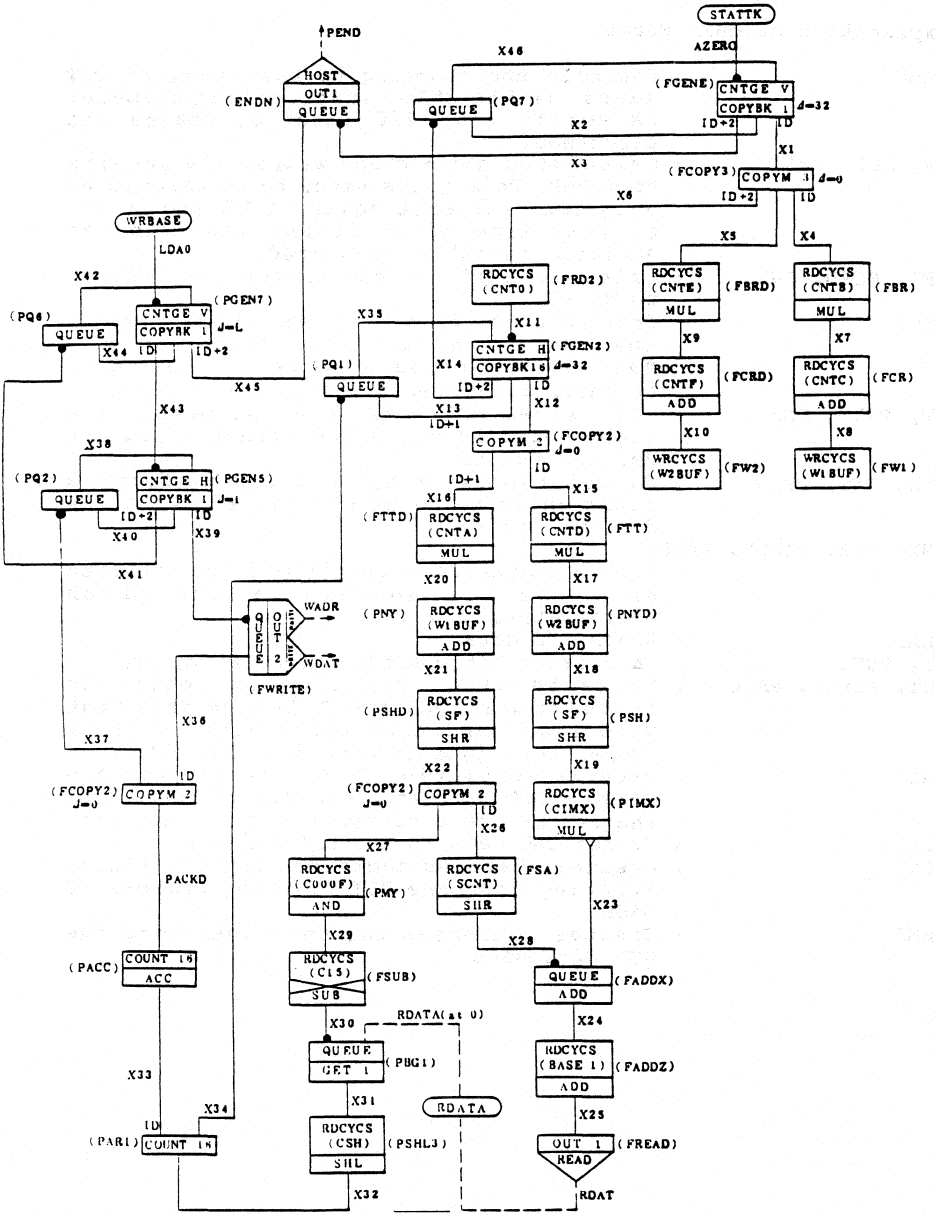
PAR1 : Detects completion of the setting of one word of DST data (the arrival of 16 tokens) and starts the processing of the next word.

FWRITE : Writes DST data to the DST address.

PGEN7 : Creates starting addresses for DST lines, starting from the DST starting address WR BASE.

PGEN5 : Creates addresses for one line among the DST addresses.

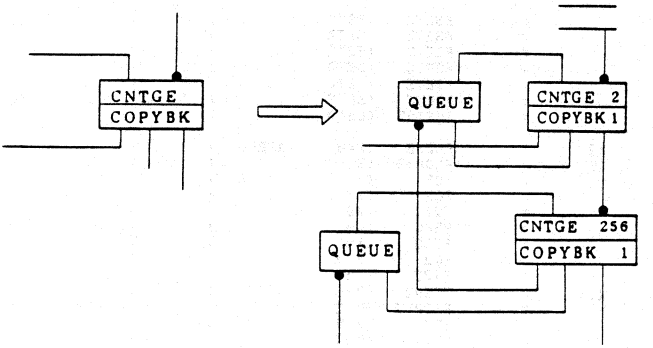
Figure 5.2
An Affine Transformation Flow Graph



5.5 Tips on Writing Flow Graphs

This program assumes that a DST is 256 dots by 256 dots. If a 512-dot by 512-dot DST is desired, you must use loop-forming nodes (as shown in Figure 5-3) in addition to the nodes FGENE and PGEN7 and change the GNTGE instruction parameter in PGEN5 to 512/16=32.

Figure 5-2
A Method for Enlarging a Processing Area



The CNTGE instruction parameter can not be allowed to exceed 256. To accommodate more than 256 values the node addition described above is required.

5.6 Assembler Source Listing

```
 1: ;*****************************************
 2: ;
 3: ;       AFFINE TRANSFORMATION
 4: ;
 5: ;-----------------------------------------
 6: ;
 7: MODULE  IPP     =       8               ;
 8: ;
 9: EQUATE  H       =       16              ;
10: EQUATE  L       =       64              ;
11: EQUATE  V       =       256             ;
12: ;
13: EQUATE  HOST    =       0               ;
14: EQUATE  READ    =       4               ;
15: EQUATE  WRITE   =       5               ;
16: ;
17: EQUATE  RDBASE  =       0               ;
18: EQUATE  WRBASE  =       32              ;
19: ;
```

```
20: ;****************************************
21: ;
22: ;       INPUT-OUTPUT
23: ;
24: ;----------------------------------------
25: ;
26: INPUT   LSA0.   LDA0.   RDATA AT 0                                    ;
27: ;
28: INPUT   SETA.   SETB.   SETC.   SETD.   SETE.   SETF.   AZERO        ;
29: ;
30: OUTPUT  RDAT.   WDAT.   WADR.   PEND                                  ;
31: ;
32: ;****************************************
33: ;
34: ;       LINK TABLE
35: ;
36: ;----------------------------------------
37: ;
38: LINK                            =       FSETA   (SETA           )   ;
39: LINK                            =       FSETB   (SETB           )   ;
40: LINK                            =       FSETC   (SETC           )   ;
41: LINK                            =       FSETD   (SETD           )   ;
42: LINK                            =       FSETE   (SETE           )   ;
43: LINK                            =       FSETF   (SETF           )   ;
44: LINK                            =       FSETRB  (LSA0           )   ;
45: ;
46: LINK    X1.     X2.     X3      =       FGENE   (X46    ,AZERO  )   ;
47: LINK    X4.     X5.     X6      =       FCOPY3  (X1             )   ;
48: LINK    X7                      =       FBR     (X4             )   ;
49: LINK    X8                      =       FCR     (X7             )   ;
50: LINK                            =       FW1     (X8             )   ;
51: LINK    X9                      =       FBRD    (X5             )   ;
52: LINK    X10                     =       FCRD    (X9             )   ;
53: LINK                            =       FW2     (X10            )   ;
54: LINK    X11                     =       FRD2    (X6             )   ;
55: LINK    X12.    X13.    X14     =       FGEN2   (X35    ,X11    )   ;
56: LINK    X15.    X16             =       FCOPY2  (X12            )   ;
57: LINK    X17                     =       FTT     (X15            )   ;
58: LINK    X18                     =       PNYD    (X17            )   ;
59: LINK    X19                     =       PSH     (X18            )   ;
60: LINK    X20                     =       FTTD    (X16            )   ;
61: LINK    X21                     =       PNY     (X20            )   ;
62: LINK    X22                     =       PSHD    (X21            )   ;
63: LINK    X23                     =       PINX    (X19            )   ;
64: LINK    X24                     =       FADDX   (X23    ,X28    )   ;
65: LINK    X25                     =       FADDZ   (X24            )   ;
66: LINK    RDAT                    =       FREAD   (X25            )   ;
67: LINK    X26.    X27             =       FCOPY2  (X22            )   ;
68: LINK    X28                     =       FSA     (X26            )   ;
69: LINK    X29                     =       PMY     (X27            )   ;
70: LINK    X30                     =       FSUB    (X29            )   ;
71: LINK    X31                     =       PSGI    (RDATA  ,X30    )   ;
72: LINK    X32                     =       PSHL3   (X31            )   ;
73: LINK    X33.    X34             =       PARI    (X32            )   ;
74: LINK    X35                     =       PQ1     (X13    ,X34    )   ;
75: LINK    PACKD                   =       PACC    (X33            )   ;
76: LINK    X36.    X37             =       FCOPY2  (PACKD          )   ;
77: LINK    WDAT.   WADR            =       FWRITE  (X36    ,X39    )   ;
78: LINK    X38                     =       PQ2     (X40    ,X37    )   ;
79: LINK    X39.    X40.    X41     =       PGEN5   (X38    ,X43    )   ;
80: LINK    X42                     =       PQ6     (X44    ,X41    )   ;
81: LINK    X43.    X44.    X45     =       PGEN7   (X42    ,LDA0   )   ;
82: LINK    PEND                    =       ENDN    (X45    ,X3     )   ;
83: LINK    X46                     =       PQ7     (X2     ,X14    )   ;
84: ;
85: ;****************************************
86: ;
87: ;       FUNCTION TABLE
88: ;
89: ;----------------------------------------
90: ;
```

```
 91: FUNCTION    FGENE   =   COPYBK  (1,    32),     CNTGE   (V       )     ;
 92: FUNCTION    FCOPY3  =   COPYM   (3,     0)                             ;
 93: FUNCTION    FBR     =   MUL.                     RDCYCS  (CNTB,   1)    ;
 94: FUNCTION    FCR     =   ADD.                     RDCYCS  (CNTC,   1)    ;
 95: FUNCTION    FW2     =   WRCYCS  (W2BUF, 1)                              ;
 96: FUNCTION    FBRD    =   MUL.                     RDCYCS  (CNTE,   1)    ;
 97: FUNCTION    FRD2    =   RDCYCS  (CNT0 , 1)                              ;
 98: FUNCTION    FCRD    =   ADD.                     RDCYCS  (CNTF,   1)    ;
 99: FUNCTION    FW1     =   WRCYCS  (W1BUF, 1)                              ;
100: FUNCTION    FCOPY2  =   COPYM   (2,     0)                             ;
101: FUNCTION    FGEN2   =   COPYBK  (16,    32),     CNTGE   (H       )     ;
102: FUNCTION    FTT     =   MUL.                     RDCYCS  (CNTD,   1)    ;
103: FUNCTION    PNY     =   ADD.                     RDCYCS  (W1BUF,  1)    ;
104: FUNCTION    PSH     =   SHR.                     RDCYCS  (SF,     1)    ;
105: FUNCTION    FTTD    =   MUL.                     RDCYCS  (CNTA,   1)    ;
106: FUNCTION    PNYD    =   ADD.                     RDCYCS  (W2BUF,  1)    ;
107: FUNCTION    PSHD    =   SHR.                     RDCYCS  (SF,     1)    ;
108: FUNCTION    PIMX    =   MUL     (Y       ),      RDCYCS  (CIMX,   1)    ;
109: FUNCTION    FADDX   =   ADD.                     QUEUE   (QUE1,  16)    ;
110: FUNCTION    FADDZ   =   ADD.                     RDCYCS  (BASE1,  1)    ;
111: FUNCTION    FREAD   =   OUT1    (READ,  0)                              ;
112: FUNCTION    FSA     =   SHR.                     RDCYCS  (SCNT,   1)    ;
113: FUNCTION    PMY     =   AND.                     RDCYCS  (C000F,  1)    ;
114: FUNCTION    FSUB    =   SUB     (XCH     ),      RDCYCS  (C15,    1)    ;
115: FUNCTION    PBG1    =   GET1.                    QUEUE   (BSF,   16)    ;
116: FUNCTION    PSHL3   =   SHL.                     RDCYCS  (CSH,   16)    ;
117: FUNCTION    PARI    =   COUNT   (16      )                              ;
118: FUNCTION    PQ1     =   QUEUE   (QUE3,  1)                              ;
119: FUNCTION    PACC    =   ACC.                     COUNT   (16      )     ;
120: FUNCTION    FWRITE  =   OUT2    (WRITE, 20H, 0), QUEUE   (QUE4,  16)    ;
121: FUNCTION    PQ2     =   QUEUE   (QUE5,  1)                              ;
122: FUNCTION    PGEN5   =   COPYBK  (1,     1),      CNTGE   (H       )     ;
123: FUNCTION    PQ6     =   QUEUE   (QUE8,  1)                              ;
124: FUNCTION    PGEN7   =   COPYBK  (1,     L),      CNTGE   (V       )     ;
125: FUNCTION    ENDN    =   OUT1    (HOST,  0),      QUEUE   (QUE9,   1)    ;
126: FUNCTION    PQ7     =   QUEUE   (QUE11, 1)                              ;
127: FUNCTION    FSETA   =   WRCYCS  (CNTA,  1)                              ;
128: FUNCTION    FSETB   =   WRCYCS  (CNTB,  1)                              ;
129: FUNCTION    FSETC   =   WRCYCS  (CNTC,  1)                              ;
130: FUNCTION    FSETD   =   WRCYCS  (CNTD,  1)                              ;
131: FUNCTION    FSETE   =   WRCYCS  (CNTE,  1)                              ;
132: FUNCTION    FSETF   =   WRCYCS  (CNTF,  1)                              ;
133: FUNCTION    FSETRB  =   WRCYCS  (BASE1, 1)                              ;
134: ;
135: ;*****************************************
136: ;
137: ;       DATA MEMORY
138: ;
139: ;-----------------------------------------
140: ;
141: MEMORY  CNTA    =       0                                               ;
142: MEMORY  CNTB    =       2000H                                           ;
143: MEMORY  CNTC    =       0                                               ;
144: MEMORY  CNTD    =       2000H                                           ;
145: MEMORY  CNTE    =       0                                               ;
146: MEMORY  CNTF    =       0                                               ;
147: MEMORY  W1BUF   =       AREA    (1       )                              ;
148: MEMORY  W2BUF   =       AREA    (1       )                              ;
149: MEMORY  CNT0    =       0                                               ;
150: MEMORY  SF      =       2                                               ;
151: MEMORY  CIMX    =       1                                               ;
152: MEMORY  BASE1   =       0                                               ;
153: MEMORY  SCNT    =       4                                               ;
154: MEMORY  C000F   =       000FH                                           ;
155: MEMORY  C15     =       15                                              ;
156: MEMORY  BSF     =       AREA    (16      )                              ;
157: MEMORY  CSH     =       0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15           ;
158: MEMORY  QUE1    =       AREA    (16      )                              ;
159: MEMORY  QUE3    =       1                                               ;
160: MEMORY  QUE4    =       AREA    (16      )                              ;
161: MEMORY  QUE5    =       1                                               ;
162: MEMORY  QUE8    =       AREA    (1       )                              ;
163: MEMORY  QUE9    =       AREA    (1       )                              ;
164: MEMORY  QUE11   =       AREA    (1       )                              ;
165: ;
```

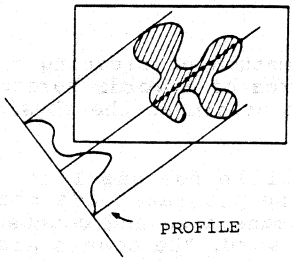
```
166: ;*****************************************
167: ;
168: ;       START
169: ;
170: ;-----------------------------------------
171: ;
172: START                                                    ;
173: ;
174: DATA    CRESET  (IPP                          )         ;
175: DATA    EXEC    (IPP,   SETA,   2000H        )         ;
176: DATA    EXEC    (IPP,   SETB,   0000H        )         ;
177: DATA    EXEC    (IPP,   SETC,   0000H        )         ;
178: DATA    EXEC    (IPP,   SETD,   0000H        )         ;
179: DATA    EXEC    (IPP,   SETE,   2000H        )         ;
180: DATA    EXEC    (IPP,   SETF,   0000H        )         ;
181: DATA    EXEC    (IPP,   LSA0,   RDBASE       )         ;
182: DATA    EXEC    (IPP,   LDA0,   WRBASE       )         ;
183: DATA    EXEC    (IPP,   AZERO,  0            )         ;
184: ;
185: END                                                      ;
```

Chapter 6

Profiling

Profiling is the process of projecting a binary image in a certain direction, counting the number of pixel having the value of "1", and forming a histogram of the results. Profiling is illustrated in Figure 6-1.

Figure 6-1
Profiling

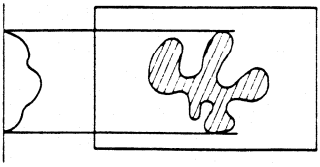


6.1 Horizontal Profiling

6.1.1 Processing Explained

A horizontal profile is a histogram that is obtained by projecting a binary image in the horizontal direction, as shown in Figure 6-2.

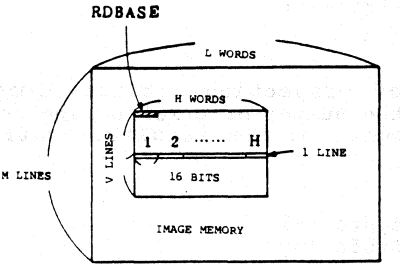
Figure 6-2
Horizontal Profiling



6.1.2 Algorithm

Suppose that an image memory is comprised of L words horizontally and M lines vertically, as shown in Figure 6-3.

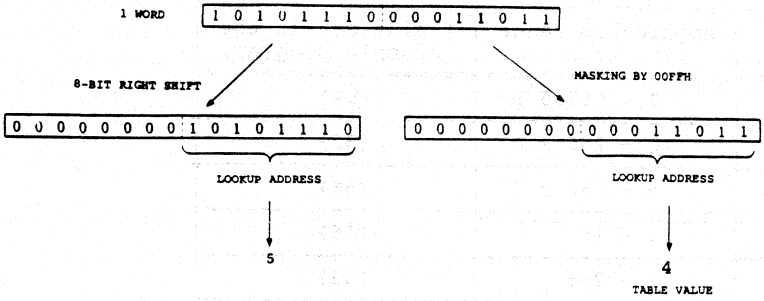
Figure 6-3
Definition of a Source Image Area



The following explains the method of creating a horizontal profile for a source image area of H words horizontally and V lines vertically, with the origin at the physical address RD BASE.

Consider the horizontal profile for one line. Since the line consists of H words, the program reads these H words one by one from the source image area and counts the number of "1" bits existing in each word. The counts are added for all H words. Let the resulting value be the histogram value (the horizontal profile) of the line.

The "1" bits in each word are counted by means of a table lookup within the DM of the uPD7281. The uPD7281 RDIDX instruction in the AG&FC is a table lookup instruction. This instruction is capable of specifying an index address in eight bits. Consequently, the word (16 bits) read from the image memory is divided into 2 parts of 8 bits each, and each of these values is used as an index to the lookup table. If the numbers of "1" bits in the corresponding addresses are stored in the table beforehand, the number of "1" bits in a word can be counted in an efficient manner by cumulative addition of the values obtained from the table lookup process.



To accumulate the looked-up values, the ACC instruction in the PU of the uPD7281 is used. This requires that the ACC register be set to 0 by the reset command before start of the program. Also, because each word is divided into two segments, the number of additions performed for n words will be 2n.

Since reading information from the image memory one word at a time would be an inefficient use of the uPD7281's internal pipelined ring, the program reads Q words of data continuously. Therefore, "H words" used in this program must a multiple of Q. Further, the cumulative values obtained through the addition operations are written in the image memory contiguously from a specified address.

6.1.3 Parameters and Their Applicable Ranges

<Assembler-coded parameters>

L ... Number of image memory words in horizontal direction
H ... Number of source image area (SRC) words in horizontal direction
V ... Number of source image area (SRC) lines in vertical direction
Q ... Number of continuous reads from the image memory (horizontal direction)
R ... Number of segments read from the image memory (vertical direction)

<Start-up token-defined parameters>

STARTS ... Source image area (SRC) starting address
STARTD ... Starting address of a profile storage

The allowable values of these parameters are indicated in the table below.

Parameter	Applicable range	(Value set in the example program)
L	0 - 65535/R	(64)
H	Q - 256 x Q	(32)
V	R - 256 x R	(256)
Q	1 - 16	(16)
R	1 - 256	(32)
STARTS	0 - 65535	(0)*
STARTD	0 - 65535	(1FFH)*

* : Although STARTS and STARTD are variables (i.e., addresses), they are given default values since they are used in the assembler DATA statement. When the uPD7281 is started up, the starting addresses of the SRC and DST areas are input as execution tokens.

Since no provision is made in this program for switching banks, you should exercise care in setting values of parameters H and V.

6.1.4 Flow Graph Explained

<Explanation of main nodes>

FGEN1, FGEN2 : Generates the starting addresses of lines on which profiling computations are performed based on the SRC starting address STARTS. Since only up to 256 can be counted by one CNTGE instruction, CNTGE instruction are used in a double loop in this program to permit a count exceeding 256.

FGEN5 : Generates data read addresses for a line in the horizontal direction. For improved internal processing of the uPD7281, this node generates addresses for Q contiguous words.

FOUTR : Reads SRC data.

FCOP1, FAND1, FSHR1: Makes two copies of the SRC data read and extracts the high-order (FSHR1) and low-order (FAND1) eight bits.

FTAB1 : Reads the uPD7281's DM by using the 8-bit data extracted in FAND1 and FSHR1 as index addresses. In other words,

this node obtains the number of "1"s in the 8-bit data by using a table lookup.

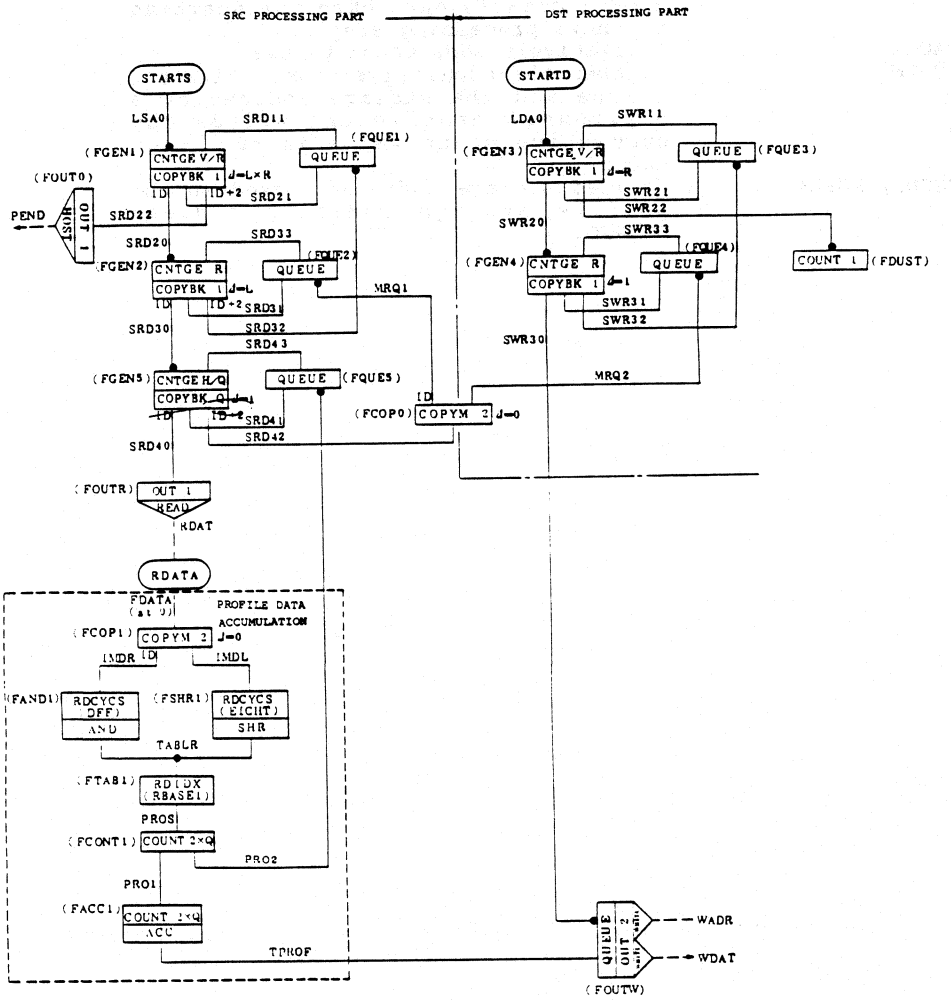
FCONT1 : Detects completion of the operation of counting Q words and starts up FGEN5. When 2 x Q tokens arrive in this node, FCONT1 creates one token for starting the next processing step.

FACC1 : Cumulatively adds count values.

FOUTW : Matches the cumulative count value for a line and the storage address of a horizontal profile and writes the cumulative count value in the image memory.

FGEN3, FGEN4 : Creates storage addresses based on the profile storage starting address STARTD.

Figure 6.4
A Flow Graph of a Horizontal Profile



6.1.5 Assembler Source Listing

```
;****************************************
;
;       HORIZONTAL PROFILE
;
;----------------------------------------
;
MODULE  IPP     =       8               ;
;
EQUATE  L       =       64              ;
EQUATE  H       =       32              ;
EQUATE  Q       =       16              ;
EQUATE  R       =       32              ;
EQUATE  V       =       256             ;
;
EQUATE  HOST    =       0               ;
EQUATE  READ    =       4               ;
EQUATE  WRITE   =       5               ;
;
EQUATE  STARTS  =       0               ;
EQUATE  STARTD  =       1FFH            ;
;
;****************************************
;
;       INPUT-OUTPUT
;
;----------------------------------------
INPUT   LSA0,   LDA0,   RDATA AT 0      ;
;
OUTPUT  PEND,   WDAT,   WADR,   RDAT    ;
;
;****************************************
;
;       LINK TABLE
;
;----------------------------------------
;
LINK    SRD20,  SRD21,  SRD22   =       FGEN1   (SRD11, LSA0    )       ;
LINK    SRD30,  SRD31,  SRD32   =       FGEN2   (SRD33, SRD20   )       ;
LINK    SRD40,  SRD41,  SRD42   =       FGEN5   (SRD43, SRD30   )       ;
LINK    SRD11                   =       FQUE1   (SRD21, SRD32   )       ;
LINK    SRD33                   =       FQUE2   (SRD31, MRQ1    )       ;
LINK    SRD43                   =       FQUE5   (SRD41, PRO2    )       ;
LINK    PEND                    =       FOUT0   (SRD22          )       ;
LINK    RDAT                    =       FOUTR   (SRD40          )       ;
LINK    MRQ1,   MRQ2            =       FCOP0   (SRD42          )       ;
;
LINK    SWR20,  SWR21,  SWR22   =       FGEN3   (SWR11, LDA0    )       ;
LINK    SWR30,  SWR31,  SWR32   =       FGEN4   (SWR33, SWR20   )       ;
LINK    SWR11                   =       FQUE3   (SWR21, SWR32   )       ;
LINK    SWR33                   =       FQUE4   (SWR31, MRQ2    )       ;
LINK                            =       FDUST   (       ,SWR22  )       ;
;
LINK    IMDR,   IMDL            =       FCOP1   (RDATA          )       ;
LINK    TABLR                   =       FAND1   (IMDR           )       ;
LINK    TABLR                   =       FSHR1   (IMDL           )       ;
LINK    PROS                    =       FTAB1   (TABLR          )       ;
LINK    PRO1,   PRO2            =       FCONT1  (PROS           )       ;
LINK    TPROF                   =       FACC1   (PRO1           )       ;
LINK    WDAT,   WADR            =       FOUTW   (TPROF, SWR30   )       ;
;
```

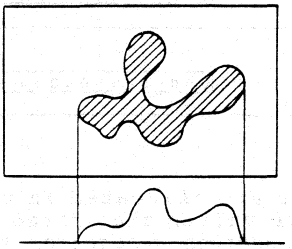
```
;****************************************
;
;       FUNCTION TABLE
;
;----------------------------------------
;
FUNCTION    FGEN1   =   COPYBK  (1.   L*R).     CNTGE   (V/R     )       ;
FUNCTION    FGEN2   =   COPYBK  (1.     L).     CNTGE   (R       )       ;
FUNCTION    FGEN5   =   COPYBK  (Q.     1).     CNTGE   (H/Q     )       ;
FUNCTION    FQUE1   =   QUEUE   (QBASE1.1)                               ;
FUNCTION    FQUE2   =   QUEUE   (QBASE2.1)                               ;
FUNCTION    FQUE5   =   QUEUE   (QBASE5.1)                               ;
FUNCTION    FOUT0   =   OUT1    (HOST.  0)                               ;
FUNCTION    FOUTR   =   OUT1    (READ.  0)                               ;
FUNCTION    FCOP0   =   COPYM   (2.     0)                               ;
FUNCTION    FGEN3   =   COPYBK  (1.     R).     CNTGE   (V/R     )       ;
FUNCTION    FGEN4   =   COPYBK  (1.     1).     CNTGE   (R       )       ;
FUNCTION    FQUE3   =   QUEUE   (QBASE3.1)                               ;
FUNCTION    FQUE4   =   QUEUE   (QBASE4.1)                               ;
FUNCTION    FDUST   =   COUNT   (1      )                                ;
FUNCTION    FCOP1   =   COPYM   (2.     0)                               ;
FUNCTION    FAND1   =   AND     (X      ).     RDCYC3  (OFF.   1)       ;
FUNCTION    FSHR1   =   SHR     (X      ).     RDCYC5  (EIGHT. 1)       ;
FUNCTION    FTAB1   =   RDIDX   (RBASE1 )                                ;
FUNCTION    FCONT1  =   COUNT   (2*Q    )                                ;
FUNCTION    FACC1   =   ACC     (X      ).     COUNT   (2*Q     )       ;
FUNCTION    FOUTW   =   OUT2    (WRITE. 20H. 0).QUEUE   (QBASE6.1)       ;
;
;****************************************
;
;       DATA MEMORY
;
;----------------------------------------
;
MEMORY  QBASE1  =       AREA    (1      )       ;
MEMORY  QBASE2  =       AREA    (1      )       ;
MEMORY  QBASE3  =       AREA    (1      )       ;
MEMORY  QBASE4  =       AREA    (1      )       ;
MEMORY  QBASE5  =       AREA    (1      )       ;
MEMORY  QBASE6  =       AREA    (1      )       ;
MEMORY  OFF     =       00FFH                   ;
MEMORY  EIGHT   =       8                       ;
MEMORY  RBASE1  =       0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4,
                        1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,
                        1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,
                        2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
                        1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,
                        2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
                        2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
                        3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
                        1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,
                        2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
                        2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
                        3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
                        2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
                        3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
                        3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
                        4,5,5,6,5,6,6,7,5,6,6,7,6,7,7,8 ;
;
;****************************************
;
;       START
;
;----------------------------------------
;
START                                                   ;
;
DATA    CRESET  (IPP                        )           ;
DATA    EXEC    (IPP.   LSA0.   STARTS      )           ;
DATA    EXEC    (IPP.   LDA0.   STARTD      )           ;
;
END                                                     ;
```

6.2 Vertical Profiling

6.2.1 Processing Explained

A vertical profile is a histogram obtained by projecting a binary image in the vertical direction, as shown in Figure 6-5.

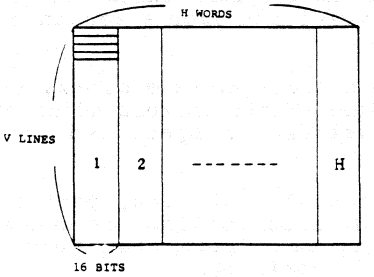
Figure 6-5
A Vertical Profile



6.2.2 Algorithm

An object screen is comprised of H words horizontally and V lines vertically.

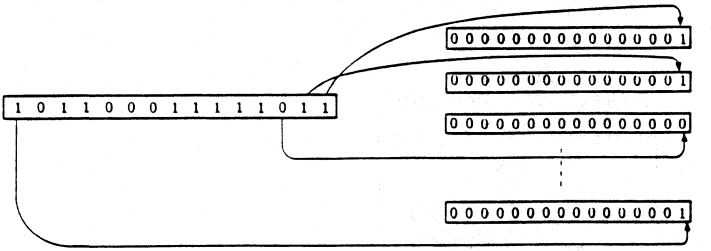
Figure 6-6
Definition of the Source Image Area



In a vertical profiling, the screen is divided horizontally into H blocks of one word each. In this program a vertical block is sought for each block. Since there are 16 bits in a word, processing a single block produces 16 cumulative values.

The processing of a block starts with one word which comprises the block copying it to 16 parts. The PU

instruction GET1 is used to shift all bits of the data to the LSBs of its 16 copies. This processing can be carried on efficiently by the folowing procedure: have the data 0-15 stored in the DM, and execute the GET1 instruction while reading this data cyclically by the use of the AG & FC instructsion RDCYCS.

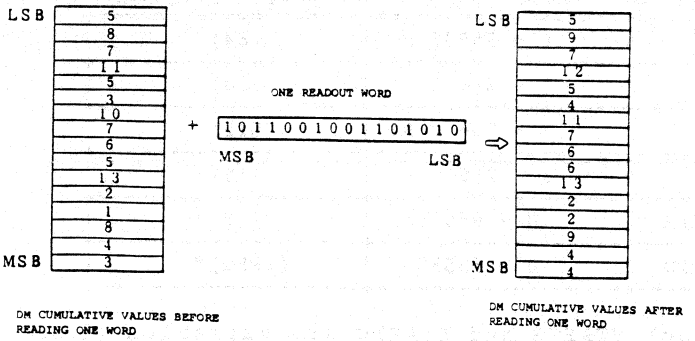


Sixteen contiguous areas are allocated in the DM, and the corresponding cumulative values are stored in these areas (Figure 6-7). The cyclic address generation function of the RDCYCS instruction in the AG&FC is used in reading from and writing to these 16 DM areas. These 16 DM areas are cleared to 0 before the processing of a block begins. In this program these processing steps and the clearing operation are performed in parallel in order to minimize the loss of efficiency through the generation of read addresses and access to external memories. (Although the clearing operation can be performed much more quickly in this program than the writing of cumulative values, the sequence of execution of PU and GE instructions cannot be predicted in general. You should investigate such a sequence using the uPD7281 simulator).

Upon completion of the processing of a block, these areas hold the cumulative values of the corresponding bits. These values are read and written in specified image memory addresses.

As in the case of horizontal profiling, control of the H blocks is with the CNTGE instruction. Be aware that the order in which data are read from the image memory is different with horizontal profiling.

Figure 6-7
Updating the DM



6.2.3 Parameters and Their Applicable Ranges

<Assembler-coded parameters>

L ... Number of image memory words in horizontal direction
H ... Number of source image area (SRC) words in horizontal direction (number of blocks)
V ... Number of source image area (SRC) lines in vertical direction
R ... Number of segments read from the image memory (vertical direction)

<Start-up token-defined parameters>

STARTS ... Source image area (SRC) starting address
STARTD ... Starting address of a profile storage

The allowable values of these parameters are indicated in the table below.

Parameter	Applicable range	(Value set in the example program)
L	0 - 65535/R	(64)
H	1 - 256	(16)
V	R - 256 x R	(512)
R	1 - 256	(16)
STARTS	0 - 65535	(0)*
STARTD	0 - 65535	(1FFH)*

* : Although STARTS and STARTD are variables (i.e., addresses), they are given default values since they are used in the assembler DATA statement. When the uPD7281 is started up, the starting addresses of the SRC and DST areas are input as execution tokens.

Since no provision is made in this program for switching banks, you should exercise care in setting values of parameters H and V. It is assumed that source areas are given in word boundaries.

6.2.4 Flow Graph Explained

<Explanation of main nodes>

FBLOK : Creates the starting addresses of blocks based on the SRC starting address STARTS.

FRDZ, FGEZ, FWRZ : Clears the save area (BASE1) to 0 for 16 count values.

FRDAD1, FRDAD2 : Creates the read addresses within a block using the GNTGE instruction in double loops.

FOUTR : Reads SRC data.

FGEDA : Makes 16 copies of one-word SRC data that was read.

FGET : Outputs the value of a given bit position in the read data in the LSB bit. Information specifying the bit positions is contained in the DM, written contiguously. FGET reads this information and passes it as a GET1 instruction token.

DM data
15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0

For example, the first token arriving

in this node reads 15 from the DM and executes the GET1 instruction. Therefore, the fifteenth bit of the arriving data is shifted to the LSB. The next token arriving in the node reads 14 from the DM, causing the fourteenth bit to be shifted to the LSB. By using the DM's cyclic read function and the GET1 instruction of the PU in this manner it is possible to output any desired bit with one node.

FADD, FWRP : Reads the contents of the DM count value save area (BASE1) successively, adds to them the bits that were output at node FGET, and writes the results back to the same area. The cyclic read and cyclic write are also used in this case.

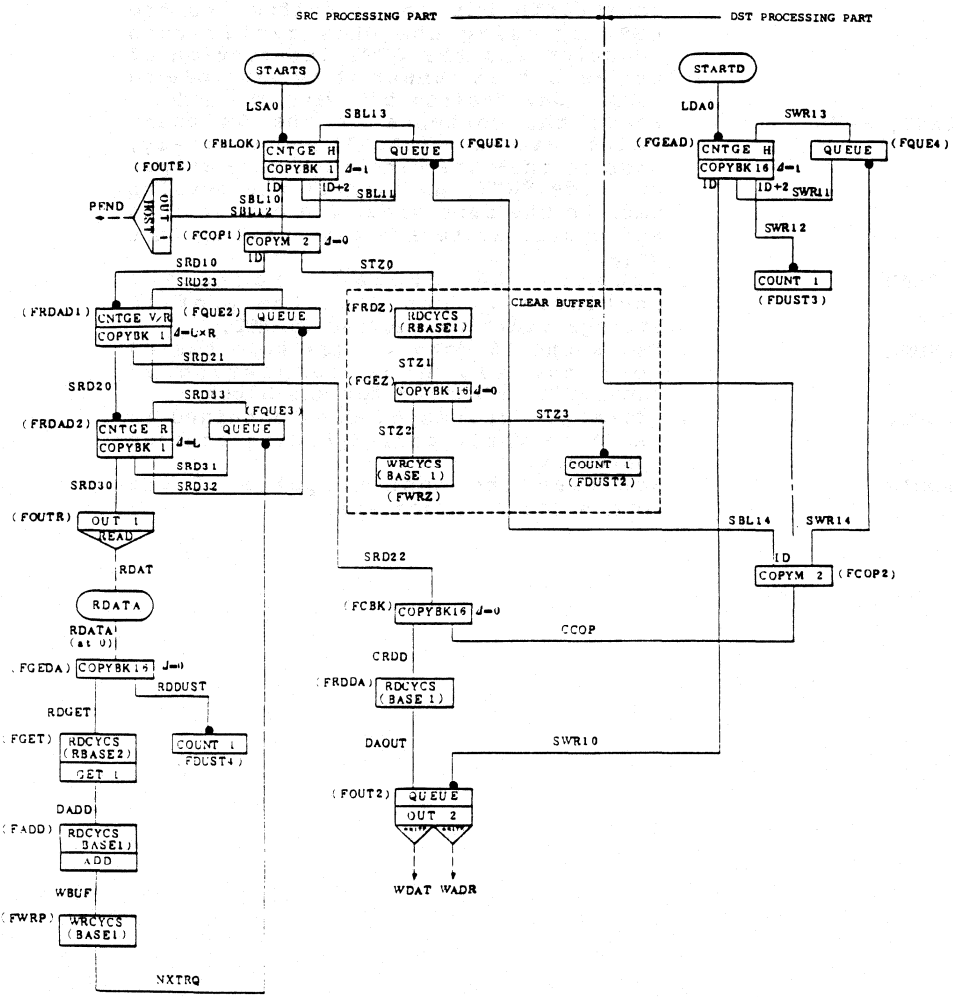
FGEAD : Creates 16 storage addresses continuously based on the profile storage starting address STARTD.

FRDDA : Reads the 16 data values from the DM continuously upon completion of cumulative addition for one block.

FOUT2 : Writes 16 values of data obtained in FRDDA to their corresponding storage addresses.

FOUTE : Notifies the host computer of the completion of processing.

Figure 6.8
A Flow Graph of Vertical Profiling



6.2.5 Assembler Source Listing

```
 1: ;****************************************
 2: ;
 3: ;       VERTICAL PROFILE
 4: ;
 5: ;----------------------------------------
 6: ;
 7: MODULE  IPP     =       8               ;
 8: ;
 9: EQUATE  H       =       16              ;
10: EQUATE  V       =       512             ;
11: EQUATE  R       =       16              ;
12: EQUATE  L       =       64              ;
13: ;
14: EQUATE  HOST    =       0               ;
15: EQUATE  READ    =       4               ;
16: EQUATE  WRITE   =       5               ;
17: ;
18: EQUATE  STARTS  =       0               ;
19: EQUATE  STARTD  =       1FFH            ;
20: ;
21: ;****************************************
22: ;
23: ;       INPUT-OUTPUT
24: ;
25: ;----------------------------------------
26: ;
27: INPUT   LSA0.   LDA0.   RDATA AT 0       ;
28: ;
29: OUTPUT  PEND.   RDAT.   WDAT.   WADR     ;
30: ;
31: ;****************************************
32: ;
33: ;       LINK TABLE
34: ;
35: ;----------------------------------------
36: ;
37: LINK    SBL10.  SBL11.  SBL12   =       FBLOK   (SBL13. LSA0    )
38: LINK    SBL13                   =       FQUE1   (SBL11. SBL14   )
39: LINK    SRD10.  STZ0            =       FCOP1   (SBL10          )
40: LINK    PEND                    =       FOUTE   (SBL12          )
41: LINK    SBL14.  SWR14           =       FCOP2   (CCOP           )
42: LINK    STZ1                    =       FRDZ    (STZ0           )
43: LINK    STZ2.   STZ3            =       FGEZ    (STZ1           )
44: LINK                            =       FWRZ    (STZ2           )
45: LINK                            =       FDUST2  (       .STZ3   )
46: ;
47: LINK    SRD20.  SRD21.  SRD22   =       FRDAD1  (SRD23  .SRD10  )
48: LINK    SRD23                   =       FQUE2   (SRD21  .SRD32  )
49: LINK    SRD30.  SRD31.  SRD32   =       FRDAD2  (SRD33  .SRD20  )
50: LINK    SRD33                   =       FQUE3   (SRD31  .NXTRQ  )
51: LINK    RDAT                    =       FOUTR   (SRD30          )
52: ;
53: LINK    RDGET.  RDDUST          =       FGEDA   (RDATA          )
54: LINK                            =       FDUST4  (       .RDDUST )
55: LINK    DADD                    =       FGET    (RDGET          )
56: LINK    WBUF                    =       FADD    (DADD           )
57: LINK    NXTRQ                   =       FWRP    (       .WBUF   )
58: ;
59: LINK    SWR10.  SWR11.  SWR12   =       FGEAD   (SWR13  .LDA0   )
60: LINK    SWR13                   =       FQUE4   (SWR11  .SWR14  )
61: LINK                            =       FDUST3  (       .SWR12  )       ;
62: ;
63: LINK    CRDD.   CCOP            =       FCBK    (SRD22          )       ;
64: LINK    DAOUT                   =       FRDDA   (CRDD           )       ;
65: LINK    WDAT.   WADR            =       FOUT2   (DAOUT  .SWR10  )       ;
66: ;
```

```
 67: ;****************************************
 68: ;
 69: ;       FUNCTION TABLE
 70: ;
 71: ;----------------------------------------
 72: ;
 73: FUNCTION    FBLOK   =   COPYBK  (1,     1),     CNTGE   (H      )
 74: FUNCTION    FQUE1   =   QUEUE   (QBASE1,1)
 75: FUNCTION    FCOP1   =   COPYM   (2,     0)
 76: FUNCTION    FOUTE   =   OUT1    (HOST,  0)
 77: FUNCTION    FCOP2   =   COPYM   (2,     0)
 78: FUNCTION    FRDZ    =   RDCYCS  (RBASE1,1)
 79: FUNCTION    FGEZ    =   COPYBK  (16,    0)
 80: FUNCTION    FWRZ    =   WRCYCS  (BASE1,16)
 81: FUNCTION    FDUST2  =   COUNT   (1      )
 82: ;
 83: FUNCTION    FRDAD1  =   COPYBK  (1,   L*R),     CNTGE   (V/R    )
 84: FUNCTION    FQUE2   =   QUEUE   (QBASE2,1)
 85: FUNCTION    FRDAD2  =   COPYBK  (1,     L),     CNTGE   (R      )
 86: FUNCTION    FQUE3   =   QUEUE   (QBASE3,1)
 87: FUNCTION    FOUR    =   OUT1    (READ,  0)
 88: ;
 89: FUNCTION    FGEDA   =   COPYBK  (16,    0)
 90: FUNCTION    FDUST4  =   COUNT   (1      )
 91: FUNCTION    FGET    =   GET1    (X      ),      RDCYCS  (RBASE2,16)
 92: FUNCTION    FADD    =   ADD     (X      ),      RDCYCS  (BASE1, 16)
 93: FUNCTION    FWRP    =   WRCYCS  (BASE1,16)
 94: ;
 95: FUNCTION    FGEAD   =   COPYBK  (16,    1),     CNTGE   (H      )
 96: FUNCTION    FQUE4   =   QUEUE   (QBASE4,16)
 97: FUNCTION    FDUST3  =   COUNT   (       1)
 98: ;
 99: FUNCTION    FCBK    =   COPYBK  (16,    0)
100: FUNCTION    FRDDA   =   RDCYCS  (BASE1,16)
101: FUNCTION    FOUT2   =   OUT2    (WRITE, 20H, 0), QUEUE   (QBASE5,16)
102: ;
103: ;****************************************
104: ;
105: ;       DATA MEMORY
106: ;
107: ;----------------------------------------
108: ;
109: MEMORY  QBASE1  =       AREA    (1      )                       ;
110: MEMORY  QBASE2  =       AREA    (1      )                       ;
111: MEMORY  QBASE3  =       AREA    (1      )                       ;
112: MEMORY  QBASE4  =       AREA    (16     )                       ;
113: MEMORY  QBASE5  =       AREA    (16     )                       ;
114: MEMORY  RBASE1  =       0                                       ;
115: MEMORY  BASE1   =       AREA    (16     )                       ;
116: MEMORY  RBASE2  =       15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0   ;
117: ;
118: ;****************************************
119: ;
120: ;       START
121: ;
122: ;----------------------------------------
123: ;
124: START                                                   ;
125: ;
126: DATA    EXEC    (IPP,   LSA0,   STARTS  )       ;
127: DATA    EXEC    (IPP,   LDA0,   STARTD  )       ;
128: ;
129: END                                                   ;
```

Chapter 7

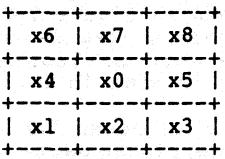
Mask Processing

The mask processing discussed in this chapter performs the following conversion of nine points, including the object point x0:

$$x0' = F(x0, x1, x2, x3, x4, x5, x6, x7, x8)$$

The value x0' of the object point x0 is determined by the eight adjoining points and x0.

Figure 7-1
Mask Processing



Mask processing can be roughly divided structurally into two processing parts: the image memory address generation part and the computation part.

This Application Library contains three examples of mask processing: smoothing, thinning, and edge detection. Image memory address generation, which is common to all these examples, is explained as a separate item.

7.1 Common Processing (Image Memory Address Generation)

7.1.1 Processing Explained

As the name implies, image memory address generation concerns the generation of addresses for image memory read/write. These addresses are required in the computational processing discussed later in this chapter. Image memory address generation consists of the following three parts:
(1) a read address generation part which generates read addresses and reads the data;
(2) a write address generation part which generates write addresses and writes the data processed in the computation part; and
(3) a final processing part which writes final data and initializes some of the FTT fields for the AG & FC

instruction.

The address generation schemes employed in the read address generation part and the write address generation part share the same flow graph organization.

The final processing part reinitializes the counters in the FTT that underwent changes in value as a result of the actions of the read address generation part, write address generation part, and the computation part (discussed later). This ensures that these counters will be ready to operate correctly when the next startup token is input without downloading the same program to the uPD7281 again.

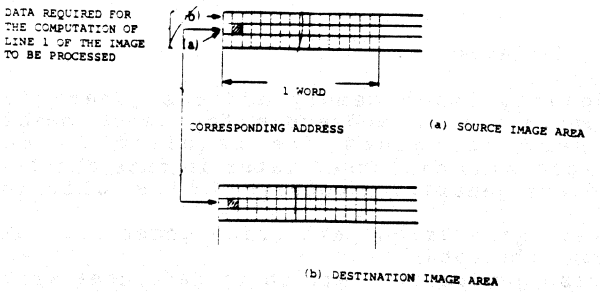
7.1.2 Algorithm

All mask processing programs described in this chapter require data occupying an area three dots horizontally by three dots vertically, as shown in Figure 7-1. Consequently, the reading of image memory source area is done in units of a total of three words: one word horizontally and three lines vertically. The writing is done in 1-word units.

Further, since an object point x0 needs the adjoining eight points, the read address generation part of this program generates read addresses for the line containing the point x0 and the lines above and below simultaneously. Therefore, the processing of H words horizontally involves the generation of 3 x H readout addresses.

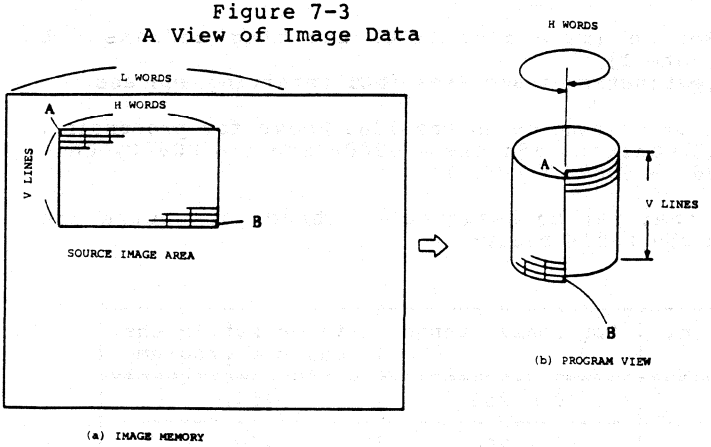
As a result, the address for the source image area in this program is specified as (b), rather than (a) in Figure 7-2.

Figure 7-2
Address Correspondence



Further, when one word of write data is generated from three words of read data, the LSB of the output data is indeterminate, as will be discussed later (see Section 7.2, "Computational Processing"). To get around these problems, data obtained in this program is first stored in the Data Memory (DM) of the uPD7281, and the writing of data from the next computational pass is done by reading it from the DM and adding one bit to it. This scheme, however, requires that each source area have preceding data and each data at the right margin of screen have successive data. In response to this requirement image data is treated as a loop in this program, as shown in Figure 7-3(b), on the assumption that, in general, data at the right or left edge of a screen is not significant. This means that only the points A (starting point) and B (end point) in the figure need to be considered. The data read from the DM at the starting point A is invalidated (through processing done in the computation part), and the data value from bit 2 at the end point (next to the LSB) is written to the LSB, which was indeterminate (final processing part).

Figure 7-3
A View of Image Data



In a binary image, an image in memory can be a 0-background image or a 1-background image, depending on the type of background data utilized. Mask processing produces different results depending on the image's background. For instance, if a 0-background thinning program was used on a 1-background graphic image, the line would get wider instead of narrower. However, the same program can be used for both 0-background and 1-background graphic image processing tasks if the computation parts of a program are written for the 0 background and data bits are reversed from 0 to 1 or from 1 to 0, as appropriate.

Therefore, the C bit of the STARTD token (which indicates the starting address of a destination image area) is used in this program to determine whether the destination image has a 0-background or a 1-background. If it has a 1-background the data is reversed between the input and output parts of the computation part. This requires that the C bit of image data is always 0 in this program.

7.1.3 Parameters and Their Applicable Ranges

<Assembler-Coded Parameters>

L ... Number of image memory words in horizontal direction
H ... Number of source image area (SRC) words in horizontal direction
V ... Number of source image area (SRC) lines in vertical direction
R ... Number of segments read from the image memory (vertical direction)

<Startup Token-Defined Parameters>

STARTS ... Source image area (SRC) starting address - L (Note 1)
STARTD ... Destination image area (DST) starting address

Note 1: Mask processing needs the line above for processing. (i.e., if SRC starting address = 0000H and L = 0080H, then STARTS = 0000H - 0080H = FF80H)

The values that can be assigned to these parameters are indicated in the table below:

Parameter	Applicable range	(Value set in the example program)
L	0 - 255	(64)
H	1 - 256	(32)
V	R - 256 x R	(512)
R	1 - 256	(2)
STARTS	0 - 65535	(FF80H)*
STARTD	0 - 65535	(20H)*

* : Although STARTS and STARTD are variables (i.e., addresses), they are given default values since they

are used in the assembler DATA statement. When the uPD7281 is started up, the starting addresses of the SRC and DST areas are input as execution tokens.

Since no provision is made in this program for switching banks, you should exercise care in setting values for parameters H and V.

7.1.4 Flow Graph Explained

A flow graph of the common processing part (image memory address generation) is shown in Figure 7-4. A combination of this flow graph and one of the computational part described below makes up a flow graph of a mask processing program.

<Explanation of Main Nodes>

FT1 and FT2 extract the information that determine whether the image has a 0-background or a 1-background from the C bit of the STARTD token, and store the information in the Data Memory (DM). This information is used in the computation part.

FT3, FT4, and FT5 generate destination image write addresses based on STARTD. Two steps, FT3 and FT4, are required because there can be more than 257 vertical lines.

FT6, FT7, and FT8 serve to synchronize the generation of addresses. FT8 synchronizes the activity of the read address generation part as well. FT10 through FT12 and FT15 through FT17 generate source image read addresses upon receipt of the STARTS token (the source image area starting address minus L) In this process FT17 works in synchronization with the computation part.

FT13 is a node for creating addresses for three contiguous vertical words.

FT27 and FT28 are nodes for creating read tokens and write tokens, respectively.

In this process the arc X46 is a write data token generated in the computation part.

The word at the end is processed in the computation part. The results of the computation are stored in the Data Memory (DM) of the uPD7281, and the token returns to FT17. This causes FT12, FT16, FT11, FT15, and FT10 to be driven, in this order, and the token is transferred to the final processing part.

The final processing part takes the results of processing the end word in FT18, FT59, FT73, FT21, FT58, and FT23 and

sets the LSB of this word equal to the value of the second bit. Then the destination image data is complete.

In addition to this data generation, FT59 updates the read counter located in the FTT part so that the values in this read counter will indicate that all data have been read from the storage area. As a result, FT59 keeps itself in synchronization with the computation part.

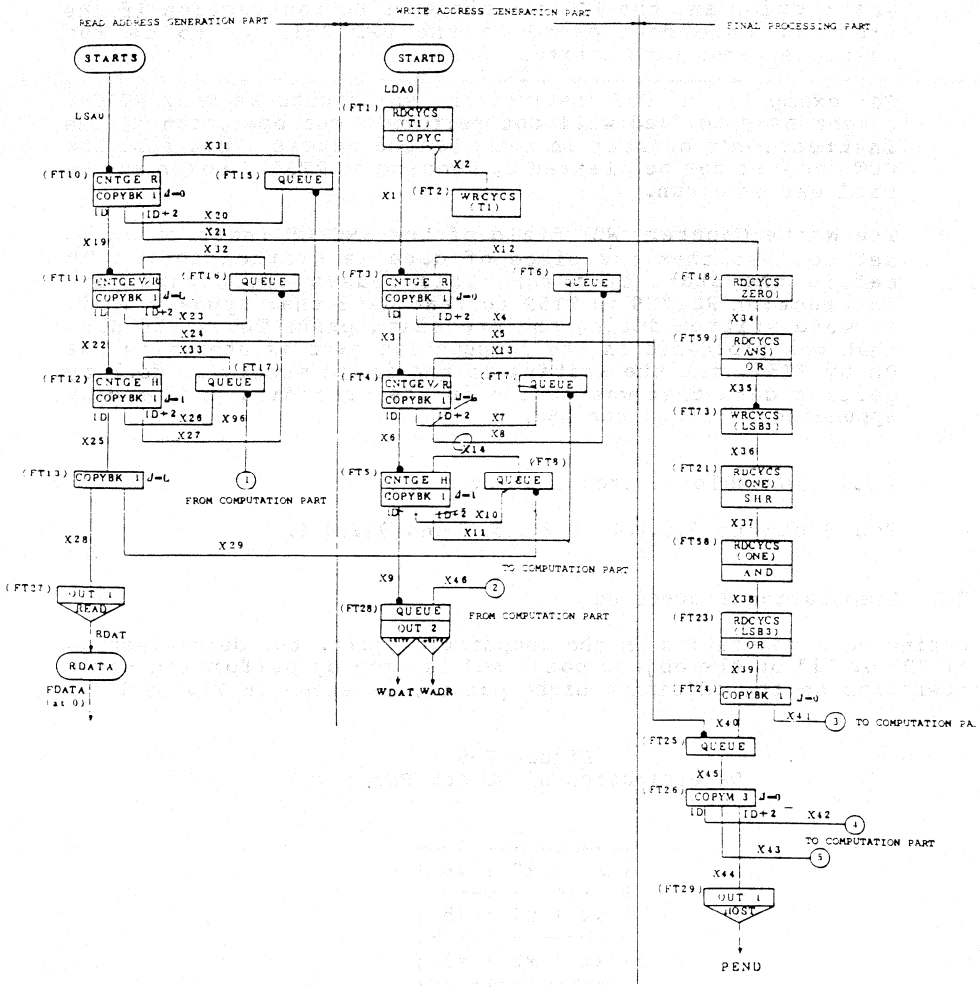
The process of performing an OR operation in FT59 requires another operand. Data for this operand is generated in FT18.

FT73 performs copying by storing one copy in the Data Memory (DM). Setting FTRC = 1 keeps the input data at the node from being erased.

FT21 and FT58 extract bit 2 from this data and set it to the value of the LSB. FT73 generates processing result data; this is stored in the Data Memory (DM).

The processing result data copied in FT24 is sent to the computation part and written to the destination image area. The other copy data waits for completion of the generation of destination area write addresses (FT25). It then initializes the uPD7281 counters (arc X42, X43) and issues a completion notification token (FT29).

Figure 7-4
A Flow Graph of a Mask Processing Image Memory
Address Generation Part



7.1.5 Tips on Writing Flow Graphs

To ensure the proper execution of this program, the counters in the FTT field must be initialized. (This is due to the way the computational part works, as will be discussed later.) You must be careful when initializing the ACC instruction and the FTT for the AG & FC instruction if the program, once downloaded to the uPD7281, is to be run multiple times with startup tokens.

For example, the CUT instruction, which cuts as many FTRC=0 tokens as specified will not perform a cut operation if the instruction's counter is not cleared before a new run. The CUT counter can be cleared by sending an FTRC=1 token at the tail end of a run.

The Write Counter (WC) field of the WRCYCS instruction is set to 1 so that one piece of data is written to the DM before the start of a run. This causes the counterpart instruction RDCYCS in FT59 to read data that appears as if it were written during the preceding pass. Thus, the data that was processed in the computation part is stored in the DM by FT57, and the writing to the image memory is done by getting data that was stored in the preceding pass and by appending 1 bit to the LSB.

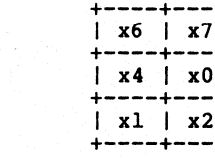
7.1.6 Assembler Source Listing

See Sections 7.2.1.4, 7.2.2.5., and 7.2.3.4.

7.2 Computation Processing

During data generation in the computation part, the determination of "0" or "1" on the object point x0' is made by performing the F operation on the adjoining eight points, as shown in Figure 7-5.

Figure 7-5
Determination of Object Point x0'



$$x0' = F(x0, x1, x2, \ldots, x8)$$

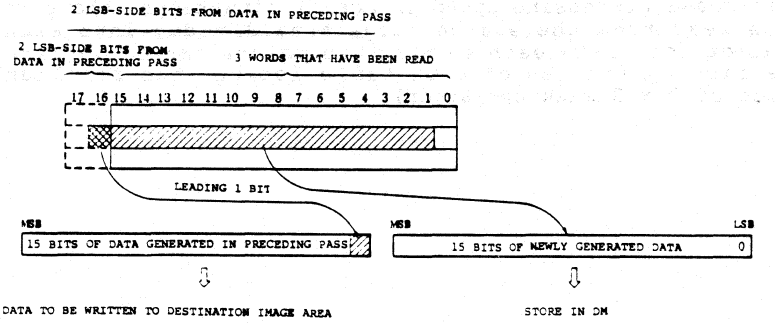
where x0' represents x0 after transformation.

For this purpose, one word of destination image data is generated by reading a total of three words from the source image area lines vertically (as explained in Section 7.1). To be more specific, the one word of written data is derived from a total of 18 x 3 bits of data: the 3 words of data read from the source image area and bits 1 and 0 of the 3-word data that was read in the preceding pass (see Figure 7-6).

The low-order 15 bits of the 16-bit data thus generated are stored in the DM of the uPD7281, along with an appended LSB of 0. The leading bit is combined with 15 bits that were stored in the DM in the preceding pass to form one word of destination image data which is written to the target image area. This is because not all eight bits in the positions adjoining the LSB of the word are available during a pass, preventing the execution of the F operation. The determination of value of the LSB, therefore, has to be deferred until the next pass.

To facilitate subsequent computations, the LSB is set to 0.

Figure 7-6
Creation of 1-Word Data to be Written



7.2.1 Smoothing

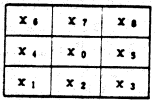
7.2.1.1 Processing Explained

Smoothing is used to remove minor noise data from images and to smooth out their edges. The processing involves masking using object point x0 and picture element data in the adjoining points (8-point neighborhood).

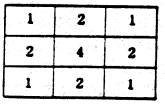
7.2.1.2 Algorithm

Point x0' is determined by the use of a three by three mask shown in Figure 7-7. The point x0' is a mapping of x0.

Figure 7-7
Smoothing Algorithm



X0 AND ADJOINING POINTS



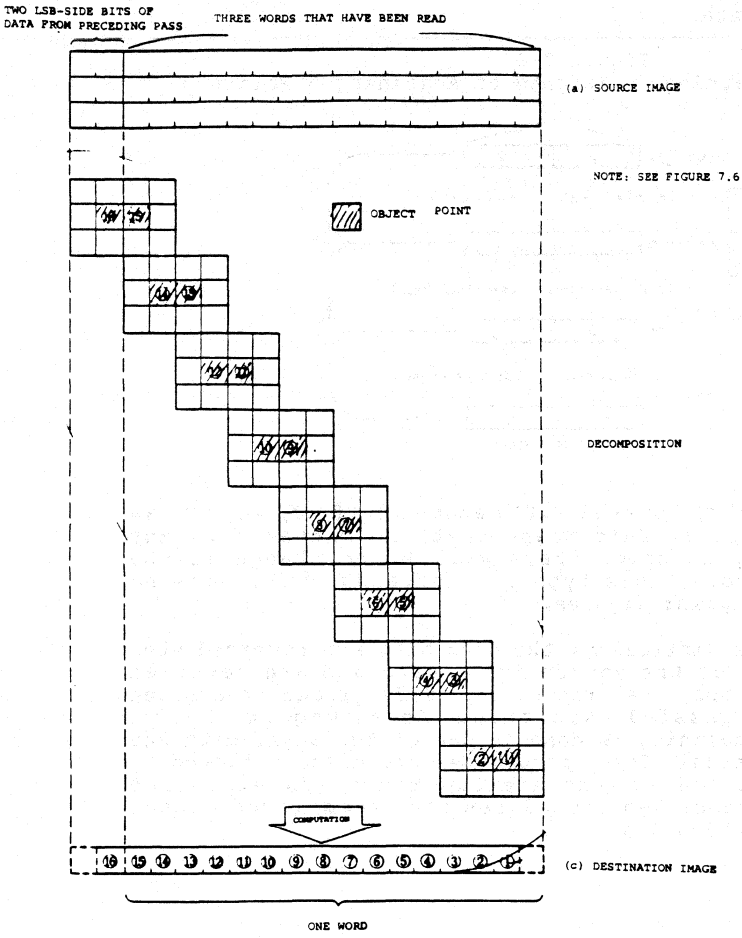
3 X 3 MASK

$$X = x_1 + 2x_2 + x_3 + 2x_4 + 4x_0 + 2x_5 + x_6 + 2x_7 + x_8$$

$$x_0' = \begin{cases} 1 \cdots\cdots X \geqq 6 \\ 0 \cdots\cdots X < 6 \end{cases}$$

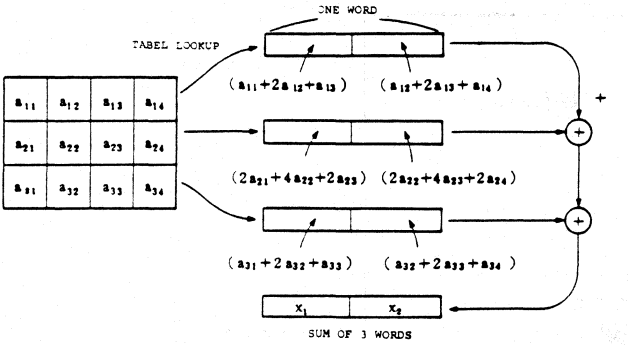
To increase processing speed in actual situations, the three words read from the source image area divided into eight areas of four bits each so that two points can be obtained at a time by the use of a DM table lookup (the resultant values of 3 x 3 mask operation).

Figure 7-8
Data Decomposition and Assembly



Although this processing mainly involves computations on x0 and the adjoining points, a table lookup is performed in this program on the basis of 4-bit decomposition data to obtain the sum of the weights of the lines in terms of an high-order byte (x1) and a low-order byte (x2). In other words, calculations on nine adjoining points centered on each of a22 and a23 are carried on simultaneously. This processing is performed in parallel on the eight 4-bit decomposition data.

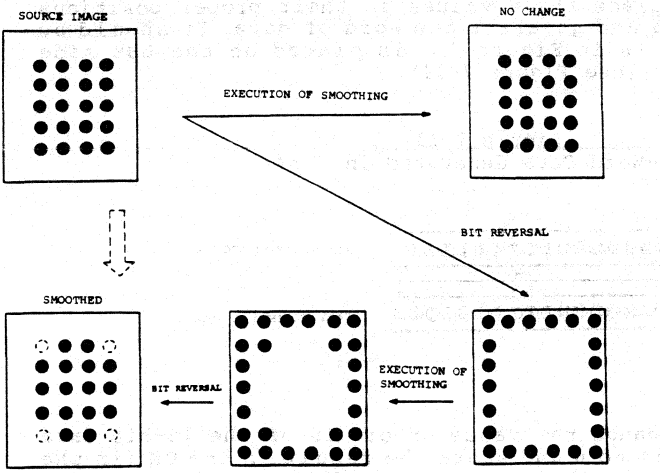
Figure 7-9
A Method of Obtaining Weights of Adjoining 9 Points



By this algorithm, however, x0' becomes 1 if x0, x1, x2, x4 are all equal to 1. This means that, no matter how many times smoothing is done, there will be no change in the results. Therefore, this type of smoothing would have no effect on rectangular figures.

To overcome this difficulty, the data bits are reversed when they are read from the source image area and are reversed back again when they are written out. This procedure ensures x0'=0 and successful smoothing (see Figure 7-10). Consequently, smoothing is done in one of two ways: with and without bit reversal. These procedures are differentiated by means of the C bit in the startup token (as explained previously). These two procedures can be employed either singly or in combination.

Figure 7-10
Smoothing of a Rectangular Figure



7.2.1.3 Flow Graph Explained

Figure 7-12 shows a flow graph (computation part) for a smoothing operation. The smoothing program is made up of this flow graph and a flow graph which is discussed in Section 7.1.

<Explanation of Main Nodes>

FT30 is the node for reversing image data bits in cases where the background is 1. Whether a given background is 0 or 1 is determined on the basis of bit C of the startup token STARTD (Figure 7-12).

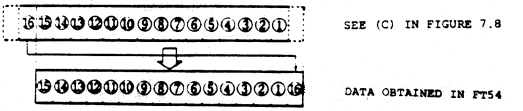
FT31, FT32, and FT35 through FT48 produce a computed value X from source image data.

FT31 distributes the read data according to lines in the vertical direction. FT32 and FT35 through FT43 create 4-bit data. FT44 and FT45 determine line values in order to obtain a computed value X by table lookup. FT47 and FT48 obtain the computed value X by adding together the line values determined in FT44 and FT45.

In addition to creating 4-bit data, FT32 copies data so that the lowest two bits of the data can be stored in the DM (LSB1, LSB2, and LSB3 areas). This is necessary to perform computations on the next data.

FT49 through FT54 evaluate the computed value X. FT49 decomposes the binary values within a word obtained in FT48, and FT50 and FT51 determine whether these values are 0 or 1. FT52 and FT53 place these values in their proper positions within the word and generate one word of data. It should be noted that bit 16 in Figure 7-8 is placed on the LSB side within the word (see Figure 7-11).

Figure 7-11
One-Word Data Generated in FT54



FT56 and FT57 change the value of bit 16 of the 16-bit data obtained in FT54 to 0 and store the result in the DM (in the ANS area) for use in the computation of the subsequent step.

FT58, FT59, and FT60 add bit 16 of the data obtained in FT54 to the 15-bit data that was stored in the preceding computation pass. They pass the result to the address generation part.

In the above process the leading edge word in the source image area has invalid data in the ANS area of the DM. Node FT60 erases this invalid data.

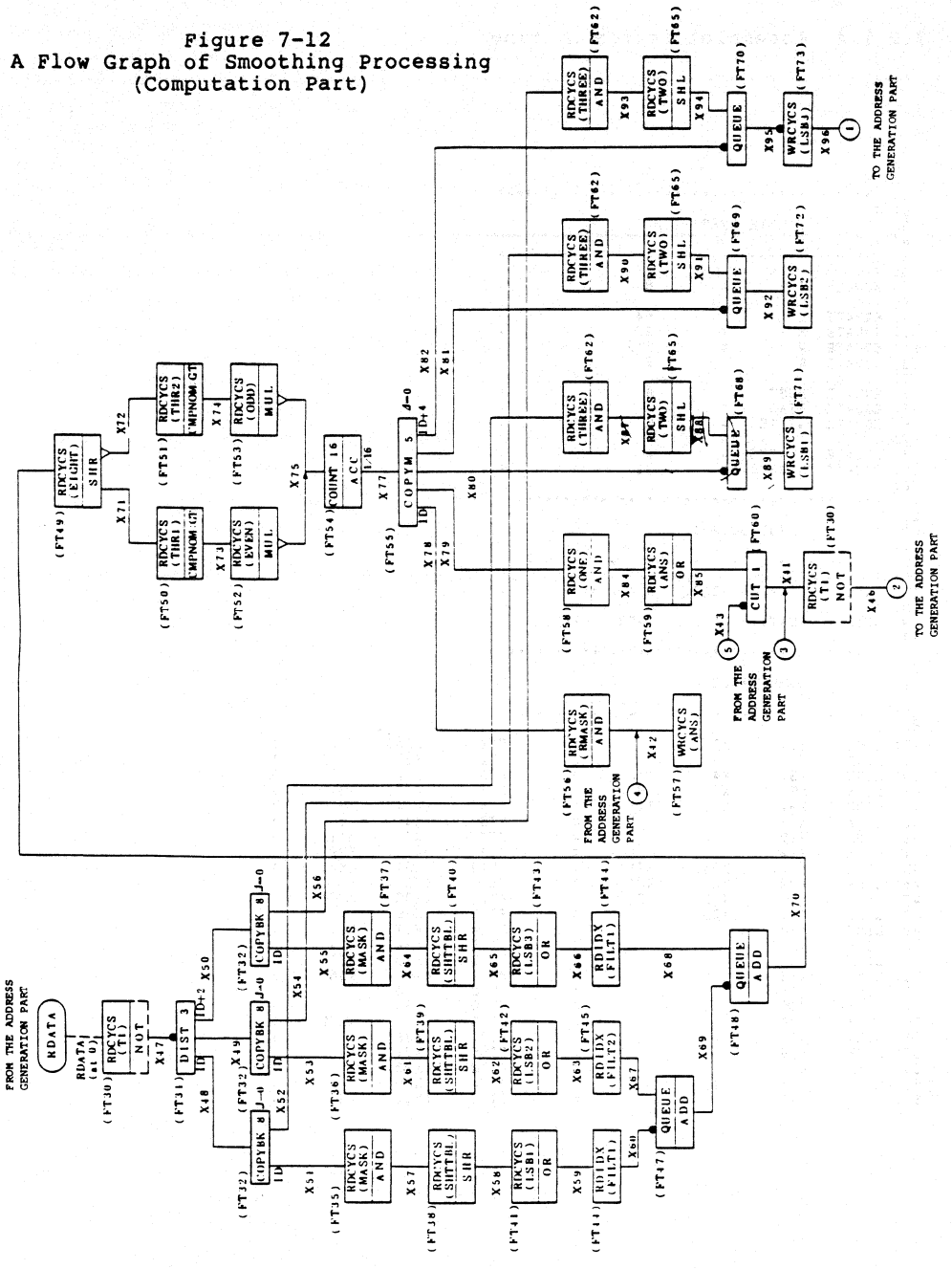
The tail end word left in the ANS area of the DM with its LSB still undetermined is sent to the final processing part of the address generation part, where it undergoes final processing.

The LSB data of this write data is undetermined (and is set to 0); in the program the value of the second lowest bit is used as an LSB value. This is done on the assumption that such a substitution made on the right hand edge of the area to be processed will have no appreciable impact.

FT62, FT65, and FT68-FT73 store the low-order two bits in the LSB1 - LSB3 area of the DM for use in subsequent computation. Nodes FT68 - FT70 ensure synchronization with the computation operations.

FT73 signals completion of the processing of a line to the address generation part. FTRC is set to 1 so that the data will not be erased.

Figure 7-12
A Flow Graph of Smoothing Processing
(Computation Part)



7.2.1.4 Assembler Source Listing

```
  1: ;*************************************
  2: ;
  3: ;       SMOOTH OPERATION
  4: ;
  5: ;-------------------------------------
  6: ;
  7: MODULE  IPP     =       8               ;
  8: ;
  9: EQUATE  L       =       64              ;
 10: EQUATE  H       =       32              ;
 11: EQUATE  V       =       512             ;
 12: EQUATE  R       =       2               ;
 13: ;
 14: EQUATE  HOST    =       0               ;
 15: EQUATE  READ    =       4               ;
 16: EQUATE  WRITE   =       5               ;
 17: ;
 18: EQUATE  STARTS  =       0               ;
 19: EQUATE  STARTD  =       32              ;
 20: ;
 21: ;*************************************
 22: ;
 23: ;       INPUT-OUTPUT
 24: ;
 25: ;-------------------------------------
 26: ;
 27: INPUT   LSA0,   LDA0,   RDATA AT 0      ;
 28: ;
 29: OUTPUT  RDAT,   WDAT,   WADR,   PEND    ;
 30: ;
 31: ;*************************************
 32: ;
 33: ;       LINK TABLE
 34: ;
 35: ;-------------------------------------
 36: ;
 37: LINK    X1,     X2              =       FT1     (LDA0           )       ;
 38: LINK                            =       FT2     (X2             )       ;
 39: LINK    X3,     X4,     X5      =       FT3     (X12,   X1      )       ;
 40: LINK    X6,     X7,     X8      =       FT4     (X13,   X3      )       ;
 41: LINK    X9,     X10,    X11     =       FT5     (X14,   X6      )       ;
 42: LINK    X12                     =       FT6     (X4,    X8      )       ;
 43: LINK    X13                     =       FT7     (X7,    X11     )       ;
 44: LINK    X14                     =       FT8     (X10,   X29     )       ;
 45: LINK    X19,    X20,    X21     =       FT10    (X31,   LSA0    )       ;
 46: LINK    X22,    X23,    X24     =       FT11    (X32,   X19     )       ;
 47: LINK    X25,    X26,    X27     =       FT12    (X33,   X22     )       ;
 48: LINK    X28,    X29             =       FT13    (X25            )       ;
 49: LINK    X30                     =       FT14    (X17,   X21     )       ;
 50: LINK    X31                     =       FT15    (X20,   X24     )       ;
 51: LINK    X32                     =       FT16    (X23,   X27     )       ;
 52: LINK    X33                     =       FT17    (X26,   X96     )       ;
 53: LINK    X34                     =       FT18    (X18            )       ;
 54: LINK    X35                     =       FT59    (X34            )       ;
 55: LINK    X36                     =       FT73    (       ,X35    )       ;
 56: LINK    X37                     =       FT21    (X36            )       ;
 57: LINK    X38                     =       FT58    (X37            )       ;
 58: LINK    X39                     =       FT23    (X38            )       ;
 59: LINK    X40,    X41             =       FT24    (X39            )       ;
 60: LINK    X45                     =       FT25    (X40,   X5      )       ;
```

```
 61: LINK    X42.    X43.    X44      =      FT26   (X45          )    ;
 62: LINK    RDAT                     =      FT27   (X28          )    ;
 63: LINK    WDAT.   WADR             =      FT28   (X46.   X9    )    ;
 64: LINK    PEND                     =      FT29   (X44          )    ;
 65: ;
 66: LINK    X47                      =      FT30   (RDATA        )    ;
 67: LINK    X48.    X49.    X50      =      FT31   (       .X47  )    ;
 68: LINK    X51.    X52              =      FT32   (X48          )    ;
 69: LINK    X53.    X54              =      FT32   (X49          )    ;
 70: LINK    X55.    X56              =      FT32   (X50          )    ;
 71: LINK    X57                      =      FT35   (X51          )    ;
 72: LINK    X58                      =      FT38   (X57          )    ;
 73: LINK    X59                      =      FT41   (X58          )    ;
 74: LINK    X60                      =      FT44   (X59          )    ;
 75: LINK    X61                      =      FT36   (X53          )    ;
 76: LINK    X62                      =      FT39   (X61          )    ;
 77: LINK    X63                      =      FT42   (X62          )    ;
 78: LINK    X67                      =      FT45   (X63          )    ;
 79: LINK    X69                      =      FT47   (X67.   X60   )    ;
 80: LINK    X64                      =      FT37   (X55          )    ;
 81: LINK    X65                      =      FT40   (X64          )    ;
 82: LINK    X66                      =      FT43   (X65          )    ;
 83: LINK    X68                      =      FT44   (X66          )    ;
 84: LINK    X70                      =      FT48   (X68.   X69   )    ;
 85: LINK    X71.    X72              =      FT49   (X70          )    ;
 86: LINK    X73                      =      FT50   (X71          )    ;
 87: LINK    X74                      =      FT51   (X72          )    ;
 88: LINK    X75                      =      FT52   (X73          )    ;
 89: LINK    X75                      =      FT53   (X74          )    ;
 90: LINK    X77                      =      FT54   (X75          )    ;
 91: LINK    X78. X79. X80. X81. X82  =      FT55   (X77          )    ;
 92: LINK    X42                      =      FT56   (X78          )    ;
 93: LINK                             =      FT57   (X42          )    ;
 94: LINK    X84                      =      FT58   (X79          )    ;
 95: LINK    X85                      =      FT59   (X84          )    ;
 96: LINK    X41                      =      FT60   (X85.   X43   )    ;
 97: LINK    X46                      =      FT30   (X41          )    ;
 98: LINK    X87                      =      FT62   (X52          )    ;
 99: LINK    X88                      =      FT65   (X87          )    ;
100: LINK    X89                      =      FT68   (X88.   X80   )    ;
101: LINK    X90                      =      FT62   (X54          )    ;
102: LINK    X91                      =      FT65   (X90          )    ;
103: LINK    X92                      =      FT69   (X91.   X81   )    ;
104: LINK    X93                      =      FT62   (X56          )    ;
105: LINK    X94                      =      FT65   (X93          )    ;
106: LINK    X95                      =      FT70   (X94.   X82   )    ;
107: LINK                             =      FT71   (X89          )    ;
108: LINK                             =      FT72   (X92          )    ;
109: LINK    X96                      =      FT73   (       .X95  )    ;
110: ;
```

```
111: ;****************************************
112: ;
113: ;        FUNCTION TABLE
114: ;
115: ;----------------------------------------
116: ;
117: FUNCTION    FT1    =   COPYC    (XY       ),     RDCYCS  (T1,     1)
118: FUNCTION    FT2    =   WRCYCS   (T1,      1)                              ;
119: FUNCTION    FT3    =   COPYBK   (1,       0),    CNTGE   (R       )      ;
120: FUNCTION    FT4    =   COPYBK   (1,       L),    CNTGE   (V/R     )      ;
121: FUNCTION    FT5    =   COPYBK   (1,       1),    CNTGE   (H       )      ;
122: FUNCTION    FT6    =   QUEUE    (QUE1,    1)                              ;
123: FUNCTION    FT7    =   QUEUE    (QUE2,    1)                              ;
124: FUNCTION    FT8    =   QUEUE    (QUE3,    1)                              ;
125: FUNCTION    FT10   =   COPYBK   (1,       0),    CNTGE   (R       )      ;
126: FUNCTION    FT11   =   COPYBK   (1,       L),    CNTGE   (V/R     )      ;
127: FUNCTION    FT12   =   COPYBK   (1,       1),    CNTGE   (H       )      ;
128: FUNCTION    FT13   =   COPYBK   (3,       L)                              ;
129: FUNCTION    FT14   =   QUEUE    (QUE4,    1)                              ;
130: FUNCTION    FT15   =   QUEUE    (QUE5,    1)                              ;
131: FUNCTION    FT16   =   QUEUE    (QUE6,    1)                              ;
132: FUNCTION    FT17   =   QUEUE    (QUE7,    1)                              ;
133: FUNCTION    FT18   =   RDCYCS   (ZERO,    1)                              ;
134: FUNCTION    FT21   =   SHR,                        RDCYCS  (ONE,    1)     ;
135: FUNCTION    FT23   =   OR,                         RDCYCS  (LSB3,   1)     ;
136: FUNCTION    FT24   =   COPYBK   (1,       0)                              ;
137: FUNCTION    FT25   =   QUEUE    (QUE14,   1)                              ;
138: FUNCTION    FT26   =   COPYM    (3,       0)                              ;
139: FUNCTION    FT27   =   OUT1     (READ,    0)                              ;
140: FUNCTION    FT28   =   OUT2     (WRITE, 20H, 0), QUEUE   (QUE8, 16)       ;
141: FUNCTION    FT29   =   OUT1     (HOST,    0)                              ;
142: ;
143: FUNCTION    FT30   =   NOT      (CNOP     ),   RDCYCS  (T1,     1)       ;
144: FUNCTION    FT31   =   DIST     (3        )                               ;
145: FUNCTION    FT32   =   COPYBK   (8,       0)                              ;
146: FUNCTION    FT35   =   AND,                        RDCYCS  (MASK,   8)     ;
147: FUNCTION    FT36   =   AND,                        RDCYCS  (MASK,   8)     ;
148: FUNCTION    FT37   =   AND,                        RDCYCS  (MASK,   8)     ;
149: FUNCTION    FT38   =   SHR,                        RDCYCS  (SHTTBL,8)      ;
150: FUNCTION    FT39   =   SHR,                        RDCYCS  (SHTTBL,8)      ;
151: FUNCTION    FT40   =   SHR,                        RDCYCS  (SHTTBL,8)      ;
152: FUNCTION    FT41   =   OR,                         RDCYCS  (LSB1,   8)     ;
153: FUNCTION    FT42   =   OR,                         RDCYCS  (LSB2,   8)     ;
154: FUNCTION    FT43   =   OR,                         RDCYCS  (LSB3,   8)     ;
155: FUNCTION    FT44   =   RDIDX    (FILT1    )                               ;
156: FUNCTION    FT45   =   RDIDX    (FILT2    )                               ;
157: FUNCTION    FT47   =   ADD,                        QUEUE   (QUE9,   8)     ;
158: FUNCTION    FT48   =   ADD,                        QUEUE   (QUE10,  8)     ;
159: FUNCTION    FT49   =   SHR      (XY       ),   RDCYCS  (EIGHT,  1)       ;
160: FUNCTION    FT50   =   CMPNOM   (GT       ),   RDCYCS  (THR1,   1)       ;
161: FUNCTION    FT51   =   CMPNOM   (GT       ),   RDCYCS  (THR2,   1)       ;
162: FUNCTION    FT52   =   MUL      (Y        ),   RDCYCS  (EVEN,   8)       ;
163: FUNCTION    FT53   =   MUL      (Y        ),   RDCYCS  (ODD,    8)       ;
164: FUNCTION    FT54   =   ACC,                        COUNT   (16      )      ;
165: FUNCTION    FT55   =   COPYM    (5,       0)                              ;
166: FUNCTION    FT56   =   AND,                        RDCYCS  (RMASK,  1)     ;
167: FUNCTION    FT57   =   WRCYCS   (ANS,     2),      1                      ;
168: FUNCTION    FT58   =   AND,                        RDCYCS  (ONE,    1)     ;
169: FUNCTION    FT59   =   OR,                         RDCYCS  (ANS,    2)     ;
170: FUNCTION    FT60   =   CUT      (1        )                               ;
171: FUNCTION    FT62   =   AND,                        RDCYCS  (THREE,  1)     ;
172: FUNCTION    FT65   =   SHL,                        RDCYCS  (TWO,    1)     ;
173: FUNCTION    FT68   =   QUEUE    (QUE11,   1)                              ;
174: FUNCTION    FT69   =   QUEUE    (QUE12,   1)                              ;
175: FUNCTION    FT70   =   QUEUE    (QUE13,   1)                              ;
176: FUNCTION    FT71   =   WRCYCS   (LSB1,    1)                              ;
177: FUNCTION    FT72   =   WRCYCS   (LSB2,    1)                              ;
178: FUNCTION    FT73   =   WRCYCS   (LSB3,    1)                              ;
179: ;
```

```
180: ;*****************************************
181: ;
182: ;        DATA MEMORY
183: ;
184: ;-----------------------------------------
185: ;
186: MEMORY  T1      =       0                                       ;
187: MEMORY  MASK    =       0C000H, 0F000H, 3C00H,  0F00H,
188:                         03C0H,  00F0H,  003CH,  000FH   ;
189: MEMORY  SHTTBL  =       14, 12, 10, 8,  6,  4,  2,  0   ;
190: MEMORY  LSB1    =       0,  0,  0,  0,  0,  0,  0,  0   ;
191: MEMORY  LSB2    =       0,  0,  0,  0,  0,  0,  0,  0   ;
192: MEMORY  LSB3    =       0,  0,  0,  0,  0,  0,  0,  0   ;
193: MEMORY  FILT1   =       0H,     1H,     102H,   103H,
194:                         201H,   202H,   303H,   304H,
195:                         100H,   101H,   202H,   203H,
196:                         301H,   302H,   403H,   404H    ;
197: MEMORY  FILT2   =       0H,     2H,     204H,   206H,
198:                         402H,   404H,   606H,   608H,
199:                         200H,   202H,   404H,   406H,
200:                         602H,   604H,   806H,   808H    ;
201: MEMORY  ZERO    =       0                                       ;
202: MEMORY  ONE     =       1                                       ;
203: MEMORY  TWO     =       2                                       ;
204: MEMORY  THREE   =       3                                       ;
205: MEMORY  EIGHT   =       8                                       ;
206: MEMORY  THR1    =       5H                                      ;
207: MEMORY  THR2    =       500H                                    ;
208: MEMORY  ODD     =       0001H,  4000H,  1000H,  0400H,
209:                         0100H,  0040H,  0010H,  0004H   ;
210: MEMORY  EVEN    =       8000H,  2000H,  0800H,  0200H,
211:                         0080H,  0020H,  0008H,  0002H   ;
212: MEMORY  RMASK   =       0FFFEH                                  ;
213: MEMORY  ANS     =       AREA    (2      )                       ;
214: MEMORY  QUE1    =       AREA    (1      )                       ;
215: MEMORY  QUE2    =       AREA    (1      )                       ;
216: MEMORY  QUE3    =       AREA    (1      )                       ;
217: MEMORY  QUE4    =       AREA    (1      )                       ;
218: MEMORY  QUE5    =       AREA    (1      )                       ;
219: MEMORY  QUE6    =       AREA    (1      )                       ;
220: MEMORY  QUE7    =       AREA    (1      )                       ;
221: MEMORY  QUE8    =       AREA    (16     )                       ;
222: MEMORY  QUE9    =       AREA    (8      )                       ;
223: MEMORY  QUE10   =       AREA    (8      )                       ;
224: MEMORY  QUE11   =       AREA    (1      )                       ;
225: MEMORY  QUE12   =       AREA    (1      )                       ;
226: MEMORY  QUE13   =       AREA    (1      )                       ;
227: MEMORY  QUE14   =       AREA    (1      )                       ;
228: ;
229: ;*****************************************
230: ;
231: ;        START
232: ;
233: ;-----------------------------------------
234: ;
235: START                                                   ;
236: ;
237: DATA    EXEC    (IPP,   LDA0,   STARTD  )               ;
238: DATA    EXEC    (IPP,   LSA0,   STARTS  )               ;
239: ;
240: END                                                     ;
```

7.2.2 Thinning

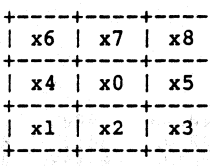
7.2.2.1 Processing Explained

The purpose of this procedure is to erase the contours of a binary image line by line, using the adjoining point data. By repeatedly executing this program it is possible to determine the center of a linear graph.

7.2.2.2 Algorithm

The point x0' after transformation of point x0 is determined by performing the operation F on the adjoining eight points, as shown in Figure 7-13.

Figure 7-13
Determination of Object Point x0'

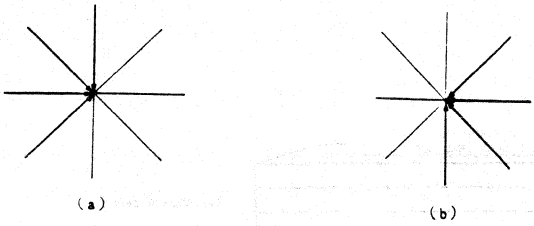


$$x0' = F(x0, x1, x2, x3, x4, x5, x6, x7, x8)$$
$$= x0 \cdot G(x1, x2, x3, x4, x5, x6, x7, x8)$$

where G(.) is a function yielding 1 or 0, depending on the particular pattern of the adjoining eight points. Hence, to make object point x0' = 0, you need only plug in 0 for G. To make x0' = x0, plug in 1 for G.

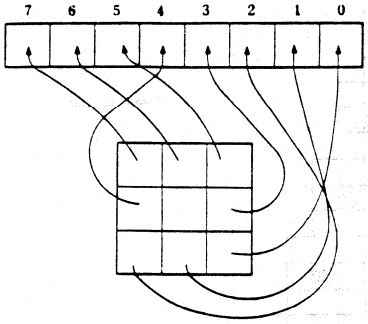
A thinning program working with a 0 background assigns 0 or 1 to each of the 256 patterns generated by 8 points (points x1 through x8) and executes the processing by means of a table lookup. However, instead of erasing points in all eight directions, the program performs the erasure from four directions in two passes. This is because, in parallel processing as opposed to serial processing, it would be difficult to perform a thinning, such as reducing a 2-dot line into a 1-dot line, if the points were erased from eight directions.

Figure 7-14
Direction of Erasure



For table lookup, the adjoining eight points are ordered as shown in Figure 7-15.

Figure 7-15
Ordering of Adjoining Eight Points

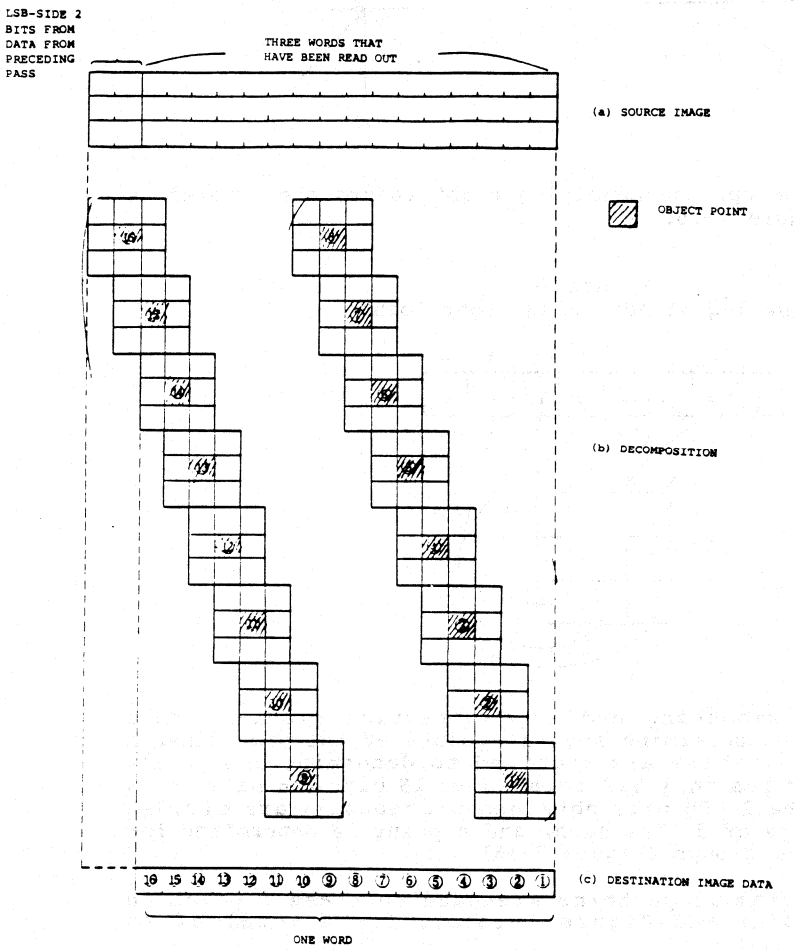


Unlike the smoothing and edge detection program, this program cannot determine several points at the same time. A total of 256 tables are required to determine one point. Since the DM has only 512 locations, 18 bits (16 bits for a word plus the 2 LSB bits obtained previously) are divided into 16 groups of 3 bits each, and a point is determined for each of these groups (Figure 7-16).

In this program 15 patterns are used to erase a point in Figure 7-14(a) and Figure 7-14(b), for a total of 30 patterns.

The 30 patterns used in this program are not necessarily the best possible patterns for producing good quality images. In some cases it would be necessary to generate and try various patterns to arrive at an appropriate pattern for the particular application.

Figure 7-16
Data Decomposition and Assembly



Values produced by table lookups can consist of values for Figure 7-14(a) and those for Figure 7-14(b). The appropriate values should be used.

This program uses a CHGDIR token, which has a value of 0 or 8, to control the number of right shifts.

7.2.2.3 Flow Graph Explained

Shown in Figure 7-17 is a thinning flow graph (computation part) that, together with the image memory address generation part flow graph discussed in Section 7.1, makes up the thinning program.

FT30 complements image data bits from 1 to 0 and 0 to 1 in cases where the background is 1.

FT21, FT31, FT32 and FT35 through FT51 determine transformed object point data from source image data.

FT31 distributes source image data. FT32 and FT35 through FT43 create data (Figure 7-16(b)) to serve as units of computation.

This thinning requires the generation of data for the table lookup as shown in Figure 7-15. For this purpose, data are directed to bit positions 7, 6, and 5 in FT38 (Figure 7-15) and to the lowest three bit positions in FT39 and FT40.

FT41 through FT43 treat data as units of computation by adding to each datum either a 1 bit or the 2 LSB-side bits in the data from the preceding pass (Note 1).

FT44, FT45, FT46, and FT48 through FT50 are used to determine the values of G, shown in Figure 7-13. The bits 7-5 and bits 2-0 in Figure 7-15 are generated in FT41 and FT43. FT45 makes two copies of data (Note 2) and generates bits 4 and 3 by performing a table lookup (MTBl in the DM).

After the generation of lookup table data, the values of G are determined through a table lookup in FT49.

However, since the values so obtained include both values for Figure 7-14(a) and Figure 7-14(b), FT50 chooses between these sets of values.

FT47 and FT21 extract data on point x0 from the other copy of data made in FT45, and FT51 determines the computed value of F from the extracted data.

FT52 and FT53 reconstitute the object point x0' after the transformation of a word which is organized as indicated in Figure 7-16(c).

FT54 copies the destination image data sought and furnishes this data for processing in the subsequent steps.

FT55 and FT56 add 0 to the LSBs of the low-order 15 bits (bits 15-1 in Figure 7-16(c)) of the destination image data obtained in FT53, writes the results to the DM (the ANS area), and uses them in the subsequent computation.

FT57, FT59, and FT60 extract bit 16 (the MSB) from the destination image data, make it into LSB data, add to it the high-order 15 bits of data which was stored in the DM in the previous pass, and pass the results onto the address generation part.

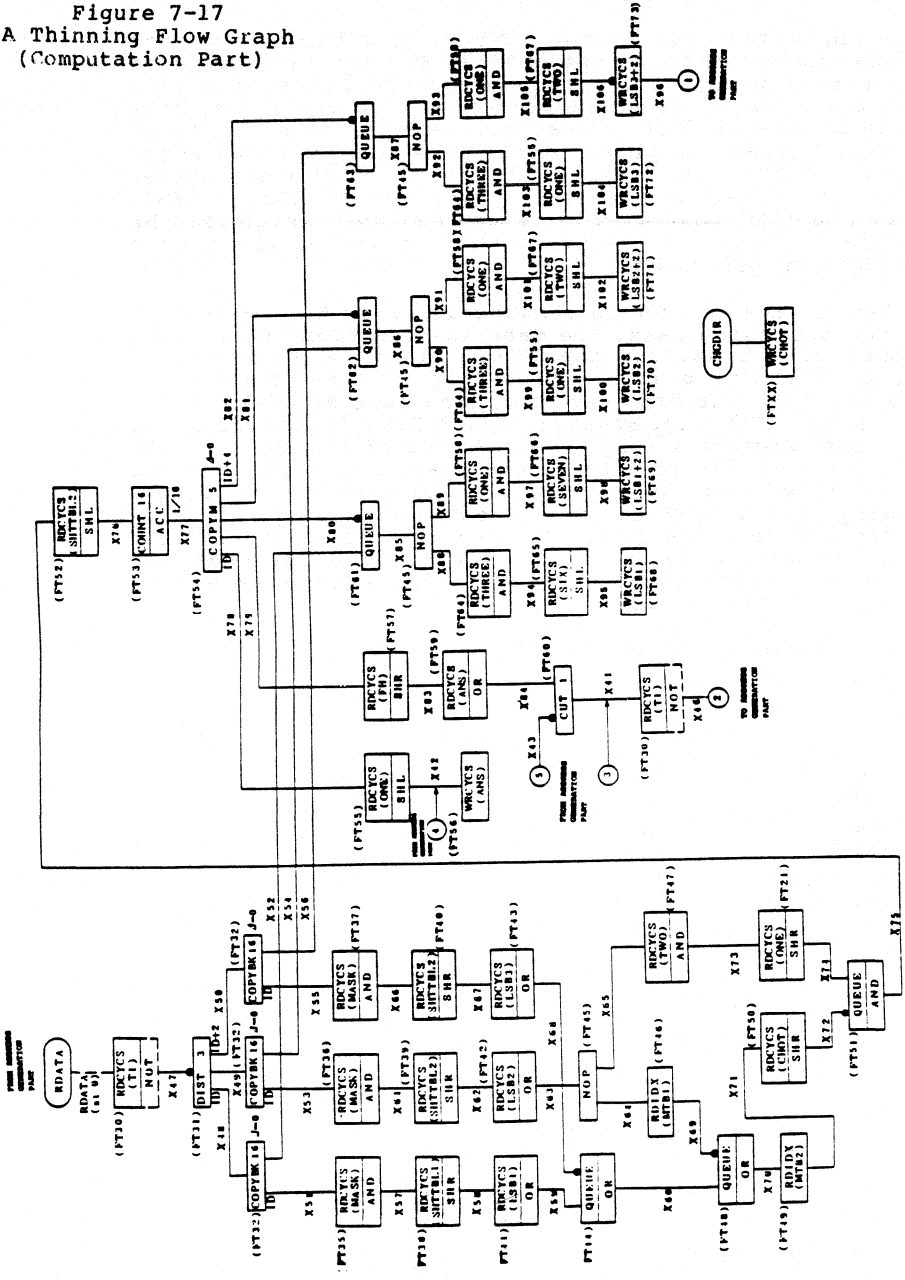
The procedure employed in this program for handling the leading and ending edges is that same as that used in the smoothing program. Refer to Section 7.2.1 for further details.

FT61 - FT73 are nodes used for storing the lowest two bits and the LSB bit of the read data copied in FT32 in the LSB1 - LSB3 of the DM, for use in subsequent computations (Note 1). FT61 - FT63 are concerned with keeping these operations in synchronization with the computational steps.

FT45 perform copying operations by using NOP (Note 2).

Note 1: See Section 7.2.2.4.
Note 2: Generally COPYBK is used to copy data. This program uses the XX output of the NOP instruction to preclude overflows on the Generate Queue of the uPD7281.

Figure 7-17
A Thinning Flow Graph
(Computation Part)



7.2.2.4 Tips on Writing Flow Graphs

In the smoothing program two LSB bits of source image data were stored in the DM of the uPD7281 for use in subsequent processing. In this thinning program the lowest two bits from a preceding pass can be used in two ways: use two bits to generate 3-bit data, or use only one bit to generate 3-bit data. Therefore, this program provides separate 2-bit and 1-bit storage locations in the DM. Although these spaces can be considered as one area, comprised of LSB1 - LSB3, they can be differentiated by means of a table which will be discussed later. This requires that copies of the source image data be generated.

When 3-bit data is made in conjunction with the 1-bit data from the preceding pass, the data is read from LSB1, LSB2, and LSB3 of the DM, and the OR operation is performed on them. It would be a simple matter if it were possible to place the LSB1 bit from the preceding pass at LSB1 + 1 (or LSB2 + 1, or LSB3 + 1). However, the WRCYCS instruction can assign base addresses only in even-numbered addresses of the DM. To provide for this fact, the space for storing 1-bit data is moved back by one position to reverse the order. Accordingly, the MASK table is used in creating 3-bit data from the source image data. The tables SHTTBL1 and SHTTBL2 used in resequencing the data are ordered differently to reflect this fact (see Line 193 in the Assembler Source listing: ...LSB1 + 2 ...).

7.2.2.5 Assembler Source Listing

```
  1: ;*************************************
  2: ;
  3: ;       THIN OPERATION
  4: ;
  5: ;-------------------------------------
  6: ;
  7: MODULE  IPP     =       8       ;
  8: ;
  9: EQUATE  L       =       64      ;
 10: EQUATE  H       =       32      ;
 11: EQUATE  V       =       512     ;
 12: EQUATE  R       =       2       ;
 13: ;
 14: EQUATE  HOST    =       0       ;
 15: EQUATE  READ    =       4       ;
 16: EQUATE  WRITE   =       5       ;
 17: ;
 18: EQUATE  STARTS  =       0FF30H  ;
 19: EQUATE  STARTD  =       20H     ;
 20: ;
 21: ;*************************************
 22: ;
 23: ;       INPUT-OUTPUT
 24: ;
 25: ;-------------------------------------
 26: ;
 27: INPUT   LSA0.   LDA0.   CHGDIR. RDATA AT 0      ;
 28: ;
 29: OUTPUT  RDAT.   WDAT.   WADR.   PEND            ;
 30: ;
 31: ;*************************************
 32: ;
 33: ;       LINK TABLE
 34: ;
 35: ;-------------------------------------
 36: ;
 37: LINK    X1.     X2              =       FT1     (LDA0           )       ;
 38: LINK                            =       FT2     (X2             )       ;
 39: LINK    X3.     X4.     X5      =       FT3     (X12.   X1      )       ;
 40: LINK    X6.     X7.     X8      =       FT4     (X13.   X3      )       ;
 41: LINK    X9.     X10.    X11     =       FT5     (X14.   X6      )       ;
```

```
  42: LINK    X12                          =    FT6     (X4,    X8      )
  43: LINK    X13                          =    FT7     (X7,    X11     )  ;
  44: LINK    X14                          =    FT8     (X10,   X29     )  ;
  45: LINK    X19,    X20,    X21          =    FT10    (X31,   LSA0    )  ;
  46: LINK    X22,    X23,    X24          =    FT11    (X32,   X19     )  ;
  47: LINK    X25,    X26,    X27          =    FT12    (X33,   X22     )  ;
  48: LINK    X28,    X29                  =    FT13    (X25            )  ;
  49: LINK    X31                          =    FT15    (X20,   X24     )  ;
  50: LINK    X32                          =    FT16    (X23,   X27     )  ;
  51: LINK    X33                          =    FT17    (X26,   X96     )  ;
  52: LINK    X34                          =    FT18    (X21            )  ;
  53: LINK    X35                          =    FT59    (X34            )  ;
  54: LINK    X36                          =    FT73    (       .X35    )  ;
  55: LINK    X37                          =    FT21    (X36            )  ;
  56: LINK    X38                          =    FT58    (X37            )  ;
  57: LINK    X39                          =    FT23    (X38            )  ;
  58: LINK    X40,    X41                  =    FT24    (X39            )  ;
  59: LINK    X45                          =    FT25    (X40,   X5      )  ;
  60: LINK    X42,    X43,    X44          =    FT26    (X45            )  ;
  61: LINK    RDAT                         =    FT27    (X28            )  ;
  62: LINK    WDAT,   WADR                 =    FT28    (X46,   X9      )  ;
  63: LINK    PEND                         =    FT29    (X44            )  ;
  64: ;
  65: LINK    X47                          =    FT30    (RDATA          )  ;
  66: LINK    X48,    X49,    X50          =    FT31    (       .X47    )  ;
  67: LINK    X51,    X52                  =    FT32    (X48            )  ;
  68: LINK    X53,    X54                  =    FT32    (X49            )  ;
  69: LINK    X55,    X56                  =    FT32    (X50            )  ;
  70: LINK    X57                          =    FT35    (X51            )  ;
  71: LINK    X58                          =    FT38    (X57            )  ;
  72: LINK    X59                          =    FT41    (X58            )  ;
  73: LINK    X60                          =    FT44    (X59,   X68     )  ;
  74: LINK    X61                          =    FT36    (X53            )  ;
  75: LINK    X62                          =    FT39    (X61            )  ;
  76: LINK    X63                          =    FT42    (X62            )  ;
  77: LINK    X64,    X65                  =    FT45    (X63            )  ;
  78: LINK    X66                          =    FT37    (X55            )  ;
  79: LINK    X67                          =    FT40    (X66            )  ;
  80: LINK    X68                          =    FT43    (X67            )  ;
  81: LINK    X69                          =    FT46    (X64            )  ;
  82: LINK    X70                          =    FT48    (X60,   X69     )  ;
  83: LINK    X71                          =    FT49    (X70            )  ;
  84: LINK    X72                          =    FT50    (X71            )  ;
  85: LINK    X73                          =    FT47    (X65            )  ;
  86: LINK    X74                          =    FT21    (X73            )  ;
  87: LINK    X75                          =    FT51    (X74,   X72     )  ;
  88: LINK    X76                          =    FT52    (X75            )  ;
  89: LINK    X77                          =    FT53    (X76            )  ;
  90: LINK    X78, X79, X80, X81, X82      =    FT54    (X77            )  ;
  91: LINK    X42                          =    FT55    (X78            )  ;
  92: LINK    X83                          =    FT57    (X79            )  ;
  93: LINK    X84                          =    FT59    (X83            )  ;
  94: LINK    X85                          =    FT61    (X52,   X80     )  ;
  95: LINK    X86                          =    FT62    (X54,   X81     )  ;
  96: LINK    X87                          =    FT63    (X56,   X82     )  ;
  97: LINK    X88,    X89                  =    FT45    (X85            )  ;
  98: LINK    X90,    X91                  =    FT45    (X86            )  ;
  99: LINK    X92,    X93                  =    FT45    (X87            )  ;
 100: LINK    X94                          =    FT64    (X88            )  ;
 101: LINK    X95                          =    FT65    (X94            )  ;
 102: LINK    X96                          =    FT73    (       .X106   )  ;
 103: LINK    X97                          =    FT58    (X89            )  ;
 104: LINK    X98                          =    FT66    (X97            )  ;
 105: LINK    X99                          =    FT64    (X90            )  ;
 106: LINK    X100                         =    FT55    (X99            )  ;
 107: LINK    X101                         =    FT58    (X91            )  ;
 108: LINK    X102                         =    FT67    (X101           )  ;
 109: LINK    X103                         =    FT64    (X92            )  ;
 110: LINK    X104                         =    FT55    (X103           )  ;
 111: LINK    X105                         =    FT58    (X93            )  ;
 112: LINK    X106                         =    FT67    (X105           )  ;
 113: LINK                                 =    FT68    (X95            )  ;
 114: LINK                                 =    FT69    (X98            )  ;
 115: LINK                                 =    FT70    (X100           )  ;
 116: LINK                                 =    FT71    (X102           )  ;
 117: LINK                                 =    FT72    (X104           )  ;
 118: LINK    X42                          =    FT55    (X78            )  ;
 119: LINK                                 =    FT56    (X42            )  ;
 120: LINK    X41                          =    FT60    (X84,   X43     )  ;
```

```
121: LINK    X46                    =       FT30    (X41            )       ;
122: ;
123: LINK                           =       FTXX    (CHGDIR         )       ;
124: ;
125: ;**************************************
126: ;
127: ;       FUNCTION TABLE
128: ;
129: ;--------------------------------------
130: ;
131: FUNCTION    FT1     =   COPYC   (XY      ),      RDCYCS  (T1.    1)      ;
132: FUNCTION    FT2     =   WRCYCS  (T1.    1)      ;
133: FUNCTION    FT3     =   COPYBK  (1.     0),     CNTGE   (R      )       ;
134: FUNCTION    FT4     =   COPYBK  (1.     L),     CNTGE   (W/R    )       ;
135: FUNCTION    FT5     =   COPYBK  (1.     1),     CNTGE   (H      )       ;
136: FUNCTION    FT6     =   QUEUE   (QUE1.  1)                              ;
137: FUNCTION    FT7     =   QUEUE   (QUE2.  1)                              ;
138: FUNCTION    FT8     =   QUEUE   (QUE3.  1)                              ;
139: FUNCTION    FT10    =   COPYBK  (1.     0),     CNTGE   (R      )       ;
140: FUNCTION    FT11    =   COPYBK  (1.     L),     CNTGE   (W/R    )       ;
141: FUNCTION    FT12    =   COPYBK  (1.     1),     CNTGE   (H      )       ;
142: FUNCTION    FT13    =   COPYBK  (3.     L)                              ;
143: FUNCTION    FT15    =   QUEUE   (QUE5.  1)                              ;
144: FUNCTION    FT16    =   QUEUE   (QUE6.  1)                              ;
145: FUNCTION    FT17    =   QUEUE   (QUE7.  1)                              ;
146: FUNCTION    FT18    =   RDCYCS  (ZERO.  1)                              ;
147: FUNCTION    FT21    =   SHR.                        RDCYCS  (ONE.   1)  ;
148: FUNCTION    FT23    =   OR.                         RDCYCS  (LSB3.  1)  ;
149: FUNCTION    FT24    =   COPYBK  (1.     0)                              ;
150: FUNCTION    FT25    =   QUEUE   (QUE4.  1)                              ;
151: FUNCTION    FT26    =   COPYM   (3.     0)                              ;
152: FUNCTION    FT27    =   OUT1    (READ.  0)                              ;
153: FUNCTION    FT28    =   OUT2    (WRITE. 200. 0). QUEUE  (QUE8. 16)      ;
154: FUNCTION    FT29    =   OUT1    (HOST.  0)                              ;
155: ;
156: FUNCTION    FT30    =   NOT     (CNOP    ),     RDCYCS  (T1.    1)      ;
157: FUNCTION    FT31    =   DIST    (3      )                               ;
158: FUNCTION    FT32    =   COPYBK  (16.    0)                              ;
159: FUNCTION    FT35    =   AND.                        RDCYCS  (MASK. 16)  ;
160: FUNCTION    FT36    =   AND.                        RDCYCS  (MASK. 16)  ;
161: FUNCTION    FT37    =   AND.                        RDCYCS  (MASK. 16)  ;
162: FUNCTION    FT38    =   SHR.                        RDCYCS  (SHTTBL1. 16)   ;
163: FUNCTION    FT39    =   SHR.                        RDCYCS  (SHTTBL2. 16)   ;
164: FUNCTION    FT40    =   SHR.                        RDCYCS  (SHTTBL2. 16)   ;
165: FUNCTION    FT41    =   OR.                         RDCYCS  (LSB1. 16)  ;
166: FUNCTION    FT42    =   OR.                         RDCYCS  (LSB2. 16)  ;
167: FUNCTION    FT43    =   OR.                         RDCYCS  (LSB3. 16)  ;
168: FUNCTION    FT44    =   OR.                         QUEUE   (QUE9. 16)  ;
169: FUNCTION    FT45    =   NOP     (XX      )                              ;
170: FUNCTION    FT46    =   RDIDX   (MTB1    )                              ;
171: FUNCTION    FT47    =   AND.                        RDCYCS  (TWO.   1)  ;
172: FUNCTION    FT48    =   OR.                         QUEUE   (QUE10.16)  ;
173: FUNCTION    FT49    =   RDIDX   (MTB2    )                              ;
174: FUNCTION    FT50    =   SHR.                        RDCYCS  (CHOT.  1)  ;
175: FUNCTION    FT51    =   AND.                        QUEUE   (QUE11.16)  ;
176: FUNCTION    FT52    =   SHL.                        RDCYCS  (SHTTBL2. 16)   ;
177: FUNCTION    FT53    =   ACC.                        COUNT   (16     )   ;
178: FUNCTION    FT54    =   COPYM   (5.     0)                              ;
179: FUNCTION    FT55    =   SHL.                        RDCYCS  (ONE.   1)  ;
180: FUNCTION    FT56    =   WRCYCS  (ANS.   2).         1                   ;
181: FUNCTION    FT57    =   SHR.                        RDCYCS  (FH.    1)  ;
182: FUNCTION    FT58    =   AND.                        RDCYCS  (ONE.   1)  ;
183: FUNCTION    FT59    =   OR.                         RDCYCS  (ANS.   2)  ;
184: FUNCTION    FT60    =   CUT     (1       )                              ;
185: FUNCTION    FT61    =   QUEUE   (QUE12. 1)                              ;
186: FUNCTION    FT62    =   QUEUE   (QUE13. 1)                              ;
187: FUNCTION    FT63    =   QUEUE   (QUE14. 1)                              ;
188: FUNCTION    FT64    =   AND.                        RDCYCS  (THREE. 1)  ;
189: FUNCTION    FT65    =   SHL.                        RDCYCS  (SIX.   1)  ;
190: FUNCTION    FT66    =   SHL.                        RDCYCS  (SEVEN. 1)  ;
191: FUNCTION    FT67    =   SHL.                        RDCYCS  (TWO.   1)  ;
192: FUNCTION    FT68    =   WRCYCS  (LSB1.  1)                              ;
193: FUNCTION    FT69    =   WRCYCS  (LSB1+2.1)                              ;
194: FUNCTION    FT70    =   WRCYCS  (LSB2.  1)                              ;
195: FUNCTION    FT71    =   WRCYCS  (LSB2+2.1)                              ;
196: FUNCTION    FT72    =   WRCYCS  (LSB3.  1)                              ;
197: FUNCTION    FT73    =   WRCYCS  (LSB3+2.1)                              ;
198: ;
199: FUNCTION    FTXX    =   WRCYCS  (CHOT.  1)                              ;
200: ;
```

```
;*****************************************
;
;       DATA MEMORY
;
;-----------------------------------------
;
MEMORY  T1      =       0                                       ;
MEMORY  CHOT    =       0                                       ;
MEMORY  MASK    =       08000H, 0E000H, 0C000H, 07000H,
                        03800H, 01C00H, 00E00H, 00700H,
                        00380H, 001C0H, 000E0H, 00070H,
                        00038H, 0001CH, 0000EH, 00007H  ;
MEMORY  SHTTBL1 =       10, 8, 9, 7, 6, 5, 4, 3,
                         2, 1, 0,-1,-2,-3,-4,-5         ;
MEMORY  SHTTBL2 =       15,13,14,12,11,10, 9, 8,
                         7, 6, 5, 4, 3, 2, 1, 0         ;
MEMORY  LSB1    =       0, 0, 0, 0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0, 0, 0, 0          ;
MEMORY  LSB2    =       0, 0, 0, 0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0, 0, 0, 0          ;
MEMORY  LSB3    =       0, 0, 0, 0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0, 0, 0, 0          ;
MEMORY  ZERO    =       0                                       ;
MEMORY  ONE     =       1                                       ;
MEMORY  TWO     =       2                                       ;
MEMORY  THREE   =       3                                       ;
MEMORY  SIX     =       6                                       ;
MEMORY  SEVEN   =       7                                       ;
MEMORY  FH      =       15                                      ;
MEMORY  MTB1    =       00H, 00H, 00H, 00H, 10H, 18H, 10H, 18H  ;
MEMORY  MTB2    =       101H,101H,101H,101H,101H,101H,101H,101H,
                        101H,101H,101H,100H,101H,101H,101H,100H,
                        101H,101H,101H,101H,101H,101H,001H,001H,
                        101H,101H,101H,101H,101H,101H,101H,100H,
                        101H,101H,101H,101H,101H,101H,101H,101H,
                        101H,101H,101H,100H,101H,101H,101H,100H,
                        101H,101H,101H,101H,101H,101H,101H,101H,
                        101H,101H,101H,101H,101H,101H,101H,100H,
                        101H,101H,101H,101H,101H,101H,101H,101H,
                        101H,101H,101H,101H,101H,101H,101H,101H,
                        101H,101H,101H,101H,101H,101H,101H,101H,
                        101H,101H,101H,101H,101H,101H,101H,101H,
                        100H,100H,101H,100H,101H,101H,101H,100H,
                        101H,101H,101H,101H,101H,101H,101H,101H,
                        101H,101H,101H,101H,101H,101H,101H,101H,
                        101H,101H,101H,101H,101H,101H,101H,101H,
                        101H,101H,101H,101H,101H,101H,001H,001H,
                        101H,101H,101H,101H,101H,101H,101H,100H,
                        101H,101H,101H,101H,101H,101H,101H,101H,
                        101H,101H,101H,101H,101H,101H,101H,101H,
                        101H,101H,101H,101H,101H,101H,101H,100H,
                        101H,101H,101H,101H,101H,101H,101H,101H,
                        101H,101H,101H,101H,101H,101H,101H,101H,
                        001H,101H,101H,101H,001H,101H,001H,001H,
                        001H,101H,101H,101H,001H,001H,101H,101H;
MEMORY  ANS     =       AREA    (2      )       ;
MEMORY  QUE1    =       AREA    (1      )       ;
MEMORY  QUE2    =       AREA    (1      )       ;
MEMORY  QUE3    =       AREA    (1      )       ;
MEMORY  QUE4    =       AREA    (1      )       ;
MEMORY  QUE5    =       AREA    (1      )       ;
MEMORY  QUE6    =       AREA    (1      )       ;
MEMORY  QUE7    =       AREA    (1      )       ;
MEMORY  QUE8    =       AREA    (16     )       ;
MEMORY  QUE9    =       AREA    (16     )       ;
MEMORY  QUE10   =       AREA    (16     )       ;
MEMORY  QUE11   =       AREA    (16     )       ;
MEMORY  QUE12   =       AREA    (1      )       ;
MEMORY  QUE13   =       AREA    (1      )       ;
MEMORY  QUE14   =       AREA    (1      )       ;
;
;*****************************************
```

```
280: ;
281: ;       START
282: ;
283: ;----------------------------------------
284: ;
285: START                                                    ;
286: ;
287: DATA    EXEC    (IPP.   CHGDIR. 0       )               ;
288: DATA    EXEC    (IPP.   LDA0.   STARTD  )               ;
289: DATA    EXEC    (IPP.   LSA0.   STARTS  )               ;
290: ;
291: END     ;
292:
```

7.2.3 Edge Detection

7.2.3.1 Processing Explained

The purpose of this procedure is to extract contour lines in which binary image data changes from 0 to 1 and from 1 to 0. Contour lines produced by this program form an 8-way connected figure* (a line figure consisting of points connected in eight directions).

* : An 8-way connected figure is one in which any point of those comprising the figure has an adjoining point that is comprised of line segments in 8 directions, vertical, horizontal, and diagonal. In contrast, a figure whose points are comprised of vertical and horizontal lines only is called a 4-way connected figure.

7.2.3.2 Algorithm

Figure 7-18

x6	x7	x8
x4	x0	x5
x1	x2	x3

In this program transformed object point x0' is determined by:

$$x0' = F(x0, x1, x2, x3, x4, x5, x6, x7, x8)$$

$$= x0 \cdot \overline{x2 \cdot x5 \cdot x7 \cdot x4}$$

In actual processing the program processes four object points at a time to optimize processing speed. This requires a three by six area, shown in Figure 7-19. (1) Specifically, 4-bit data are obtained for each line from 6-bit data of each line (Figure 7-20). Then, (2) an OR is performed on the three 4-bit data thus obtained to yield a mask pattern. (3) The object point data after the transformation are obtained by applying the AND operation between the four object points and the mask pattern.

Patterns for the lines are obtained in the program principally by means of table lookups. Lines 1 and 3 contain points which correspond to x2 and x7 of Figure 7-18 and line 2 contains points which correspond with x4 and x5. The value of these points enter into operation F. If any of these points are 0, the transformed object point x0' should be the same as x0 of the source image.

Figure 7-19
Data Decomposition and Assembly

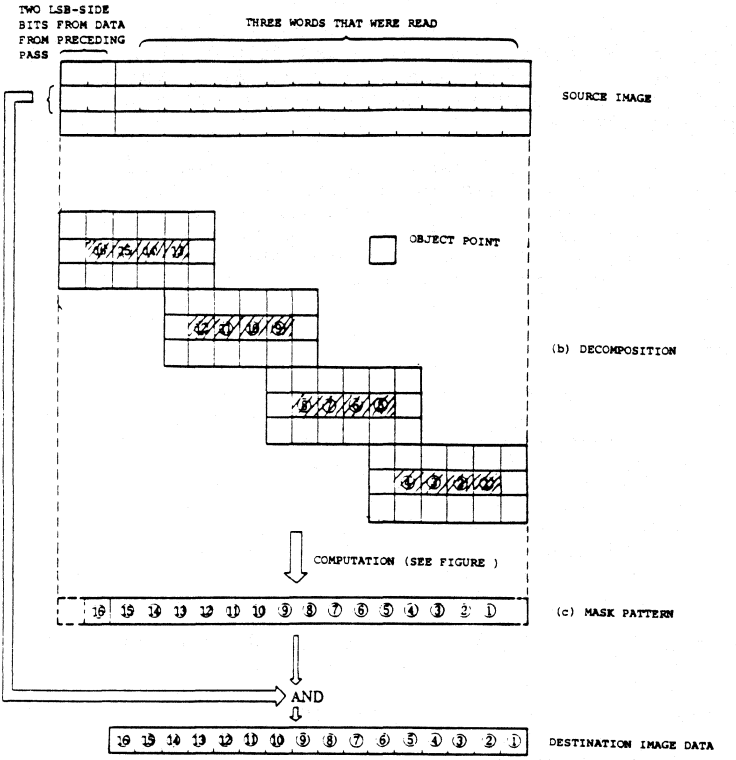
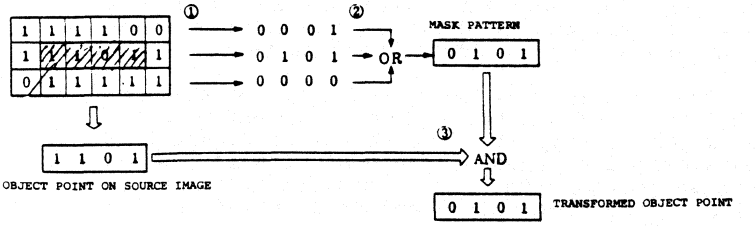


Figure 7-20



7.2.3.3 Flow Graph Explained

Shown in Figure 7-21 is an edge detection flow graph (computation part) which, together with the image memory address generation part flow graph discussed in Section 7.1, makes up the edge detection program.

<Explanation of Main Nodes>

FT30 reverses image data bits for cases where the background is 1 (see Section 7.1.2).

FT31, FT32, FT35 through FT48, FT21, and FT24 determine transformed object points from the source image data. FT31 distributes the data that were read to vertical lines. FT32 and FT35 through FT43 create 6-bit data. FT44 and FT46 perform the table lookup on the basis of the 6-bit data to obtain 4-bit data that form the basis for mask patterns.

Since the creation of transformed object data requires source image object points, FT24 copies the data that are 1-bit right shifted in FT21 for position alignment.

FT45 and FT47 are concerned with mask pattern creation. The patterns created in these nodes are AND-operated with the source image object points generated in FT21, resulting in transformed object points.

In addition to creating 6-bit data, FT32 copies the data so that its lowest two bits can be stored in the DM (the LSB1-LSB3 area) for use in subsequent computations.

FT49 and FT50 reconstitute the transformation object point data obtained in FT48 into one word. This word is organized as shown in Figure 7-19(d).

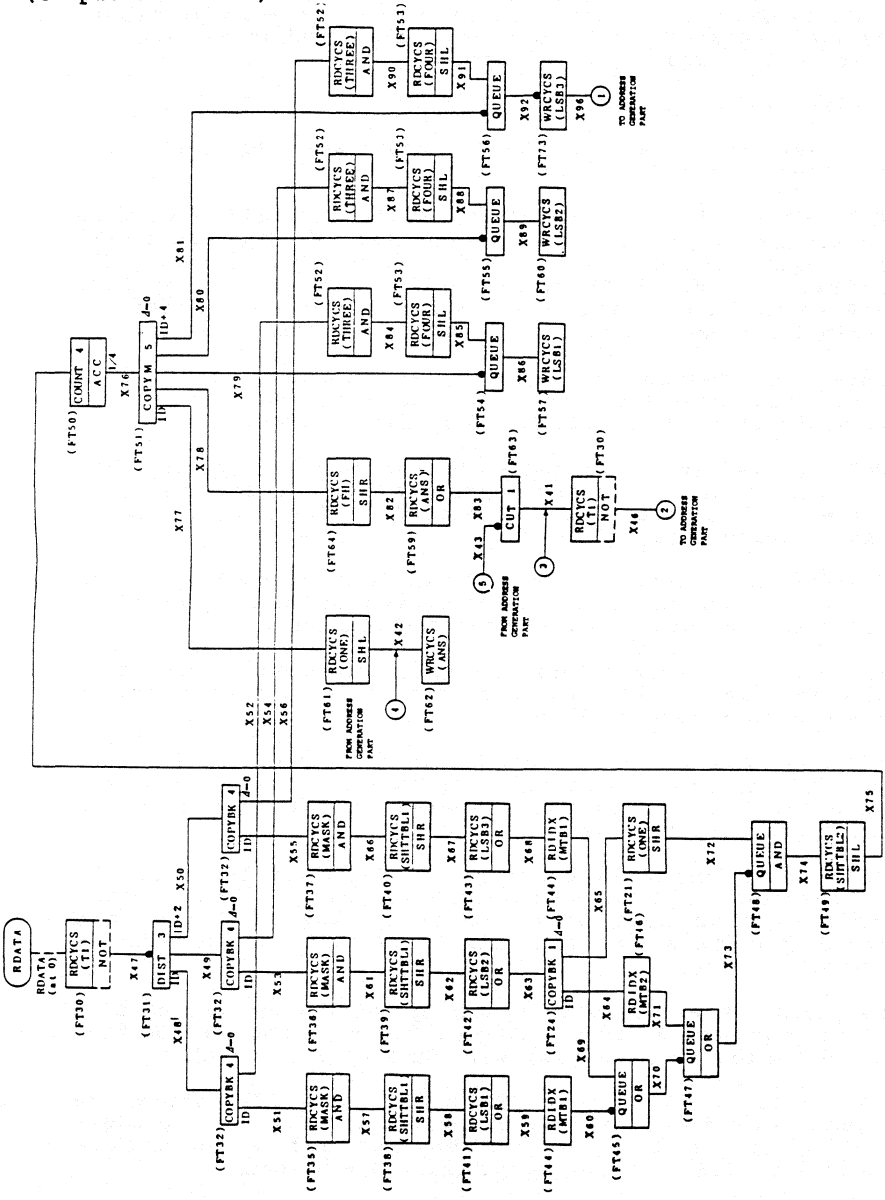
FT61 and FT62 add 0 to the LSBs of the low-order 15 bits, (15)-(1), of the destination image data obtained in FT50 and write the results in the ANS area of the DM for use in subsequent computations.

FT64, FT59, and FT63 extract the MSB (bit 16) from the destination image data to use as LSB data, add the high-order 15 bits that were stored in the DM during the preceding computation, and pass the results to the address generation part.

The procedure employed in handling leading and ending edge data is the same as that used in other programs (see Section 7.2.1.3, "Flow Graph Explained," Smoothing Program).

FT52, FT53, FT54 through FT57, FT60, and FT73 store the lowest two bits of read data, copied in FT32, in the LSB1-LSB3 area of the DM. FT54 through FT56 ensure synchronization with the computation operations.

Figure 7-21
An Edge Detection Flow Graph
(Computation Part)



7.2.3.4 Assembler Source Listing

```
;****************************************
;
;       EDGE DETECTION
;
;----------------------------------------
;
MODULE  IPP     =       8               ;
;
EQUATE  L       =       64              ;
EQUATE  H       =       32              ;
EQUATE  V       =       512             ;
EQUATE  R       =       2               ;
;
EQUATE  HOST    =       0               ;
EQUATE  READ    =       4               ;
EQUATE  WRITE   =       5               ;
;
EQUATE  STARTS  =       0FF80H          ;
EQUATE  STARTD  =       32              ;
;
;****************************************
;
;       INPUT-OUTPUT
;
;----------------------------------------
;
INPUT   LDA0,   LSA0,   RDATA AT 0      ;
;
OUTPUT  RDAT,   WDAT,   WADR,   PEND    ;
;
;****************************************
;
;       LINK TABLE
;
;----------------------------------------
;
LINK    X1,     X2              =       FT1     (LDA0           )       ;
LINK                            =       FT2     (X2             )       ;
LINK    X3,     X4,     X5      =       FT3     (X12,   X1      )       ;
LINK    X6,     X7,     X8      =       FT4     (X13,   X3      )       ;
LINK    X9,     X10,    X11     =       FT5     (X14,   X6      )       ;
LINK    X12                     =       FT6     (X4,    X8      )       ;
LINK    X13                     =       FT7     (X7,    X11     )       ;
LINK    X14                     =       FT8     (X10,   X29     )       ;
LINK    X19,    X20,    X21     =       FT10    (X31,   LSA0    )       ;
LINK    X22,    X23,    X24     =       FT11    (X32,   X19     )       ;
LINK    X25,    X26,    X27     =       FT12    (X33,   X22     )       ;
LINK    X28,    X29             =       FT13    (X25            )       ;
LINK    X31                     =       FT15    (X20,   X24     )       ;
LINK    X32                     =       FT16    (X23,   X27     )       ;
LINK    X33                     =       FT17    (X26,   X96     )       ;
LINK    X34                     =       FT18    (X21            )       ;
LINK    X35                     =       FT59    (X34            )       ;
LINK    X36                     =       FT73    (       ,X35    )       ;
LINK    X37                     =       FT21    (X36            )       ;
LINK    X38                     =       FT58    (X37            )       ;
LINK    X39                     =       FT23    (X38            )       ;
LINK    X40,    X41             =       FT24    (X39            )       ;
LINK    X45                     =       FT25    (X40,   X5      )       ;
LINK    X42,    X43,    X44     =       FT26    (X45            )       ;
```

```
  61: LINK    RDAT                          =     FT27   (X28          )   ;
  62: LINK    WDAT,   WADR                  =     FT28   (X46,   X9    )   ;
  63: LINK    PEND                          =     FT29   (X44          )   ;
  64: ;
  65: LINK    X47                           =     FT30   (RDATA        )   ;
  66: LINK    X48,    X49,    X50           =     FT31   (       ,X47  )   ;
  67: LINK    X51,    X52                   =     FT32   (X48          )   ;
  68: LINK    X53,    X54                   =     FT32   (X49          )   ;
  69: LINK    X55,    X56                   =     FT32   (X50          )   ;
  70: LINK    X57                           =     FT35   (X51          )   ;
  71: LINK    X58                           =     FT38   (X57          )   ;
  72: LINK    X59                           =     FT41   (X58          )   ;
  73: LINK    X60                           =     FT44   (X59          )   ;
  74: LINK    X61                           =     FT36   (X53          )   ;
  75: LINK    X62                           =     FT39   (X61          )   ;
  76: LINK    X63                           =     FT42   (X62          )   ;
  77: LINK    X64,    X65                   =     FT24   (X63          )   ;
  78: LINK    X66                           =     FT37   (X55          )   ;
  79: LINK    X67                           =     FT40   (X66          )   ;
  80: LINK    X68                           =     FT43   (X67          )   ;
  81: LINK    X69                           =     FT44   (X68          )   ;
  82: LINK    X70                           =     FT45   (X69,   X60   )   ;
  83: LINK    X71                           =     FT46   (X64          )   ;
  84: LINK    X72                           =     FT21   (X65          )   ;
  85: LINK    X73                           =     FT47   (X71,   X70   )   ;
  86: LINK    X74                           =     FT48   (X72,   X73   )   ;
  87: LINK    X75                           =     FT49   (X74          )   ;
  88: LINK    X76                           =     FT50   (X75          )   ;
  89: LINK    X77, X78, X79, X80, X81       =     FT51   (X76          )   ;
  90: LINK    X82                           =     FT64   (X78          )   ;
  91: LINK    X83                           =     FT59   (X82          )   ;
  92: LINK    X84                           =     FT52   (X52          )   ;
  93: LINK    X85                           =     FT53   (X84          )   ;
  94: LINK    X86                           =     FT54   (X85,   X79   )   ;
  95: LINK    X87                           =     FT52   (X54          )   ;
  96: LINK    X88                           =     FT53   (X87          )   ;
  97: LINK    X89                           =     FT55   (X88,   X80   )   ;
  98: LINK    X90                           =     FT52   (X56          )   ;
  99: LINK    X91                           =     FT53   (X90          )   ;
 100: LINK    X92                           =     FT56   (X91,   X81   )   ;
 101: LINK    X96                           =     FT73   (       ,X92  )   ;
 102: LINK    X41                           =     FT53   (X83,   X43   )   ;
 103: LINK    X46                           =     FT30   (X41          )   ;
 104: LINK                                  =     FT61   (X77          )   ;
 105: LINK                                  =     FT62   (X42          )   ;
 106: LINK                                  =     FT57   (X86          )   ;
 107: LINK                                  =     FT60   (X89          )   ;
 108: ;
```

```
;****************************************
;
;       FUNCTION TABLE
;
;----------------------------------------
;
FUNCTION    FT1     =   COPYC   (XY       ),      RDCYCS  (T1,    1)
FUNCTION    FT2     =   WRCYCS  (T1,      1)
FUNCTION    FT3     =   COPYBK  (1,       0),     CNTGE   (R      )
FUNCTION    FT4     =   COPYBK  (1,       L),     CNTGE   (V/R    )
FUNCTION    FT5     =   COPYBK  (1,       1),     CNTGE   (H      )
FUNCTION    FT6     =   QUEUE   (QUE1,    1)
FUNCTION    FT7     =   QUEUE   (QUE2,    1)
FUNCTION    FT8     =   QUEUE   (QUE3,    1)
FUNCTION    FT10    =   COPYBK  (1,       0),     CNTGE   (R      )
FUNCTION    FT11    =   COPYBK  (1,       L),     CNTGE   (V/R    )
FUNCTION    FT12    =   COPYBK  (1,       1),     CNTGE   (H      )
FUNCTION    FT13    =   COPYBK  (3,       L)
FUNCTION    FT15    =   QUEUE   (QUE5,    1)
FUNCTION    FT16    =   QUEUE   (QUE6,    1)
FUNCTION    FT17    =   QUEUE   (QUE7,    1)
FUNCTION    FT18    =   RDCYCS  (ZERO,    1)
FUNCTION    FT21    =   SHR,                        RDCYCS  (ONE,   1)
FUNCTION    FT23    =   OR,                         RDCYCS  (LSB3,  1)
FUNCTION    FT24    =   COPYBK  (1,       0)
FUNCTION    FT25    =   QUEUE   (QUE4,    1)
FUNCTION    FT26    =   COPYM   (3,       0)
FUNCTION    FT27    =   OUT1    (READ,    0)
FUNCTION    FT28    =   OUT2    (WRITE,  20H, 0),   QUEUE   (QUE8, 16)
FUNCTION    FT29    =   OUT1    (HOST,    0)
;
FUNCTION    FT30    =   NOT     (CNOP     ),      RDCYCS  (T1,    1)
FUNCTION    FT31    =   DIST    (3        )
FUNCTION    FT32    =   COPYBK  (4,       0)
FUNCTION    FT35    =   AND,                        RDCYCS  (MASK,  4)
FUNCTION    FT36    =   AND,                        RDCYCS  (MASK,  4)
FUNCTION    FT37    =   AND,                        RDCYCS  (MASK,  4)
FUNCTION    FT38    =   SHR,                        RDCYCS  (SHTTBL1,4)
FUNCTION    FT39    =   SHR,                        RDCYCS  (SHTTBL1,4)
FUNCTION    FT40    =   SHR,                        RDCYCS  (SHTTBL1,4)
FUNCTION    FT41    =   OR,                         RDCYCS  (LSB1,  4)
FUNCTION    FT42    =   OR,                         RDCYCS  (LSB2,  4)
FUNCTION    FT43    =   OR,                         RDCYCS  (LSB3,  4)
FUNCTION    FT44    =   RDIDX   (MTB1     )
FUNCTION    FT45    =   OR,                         QUEUE   (QUE9,  4)
FUNCTION    FT46    =   RDIDX   (MTB2     )
FUNCTION    FT47    =   OR,                         QUEUE   (QUE10, 4)
FUNCTION    FT48    =   AND,                        QUEUE   (QUE11, 4)
FUNCTION    FT49    =   SHL,                        RDCYCS  (SHTTBL2,4)
FUNCTION    FT50    =   ACC,                        COUNT   (4      )
FUNCTION    FT51    =   COPYM   (5,       0)
FUNCTION    FT52    =   AND,                        RDCYCS  (THREE, 1)
FUNCTION    FT53    =   SHL,                        RDCYCS  (FOUR,  1)
FUNCTION    FT54    =   QUEUE   (QUE12,   1)
FUNCTION    FT55    =   QUEUE   (QUE13,   1)
FUNCTION    FT56    =   QUEUE   (QUE14,   1)
FUNCTION    FT57    =   WRCYCS  (LSB1,    1)
FUNCTION    FT60    =   WRCYCS  (LSB2,    1)
FUNCTION    FT73    =   WRCYCS  (LSB3,    1)
FUNCTION    FT61    =   SHL,                        RDCYCS  (ONE,   1)
FUNCTION    FT62    =   WRCYCS  (ANS,     2),       1
FUNCTION    FT54    =   SHR,                        RDCYCS  (FH,    1)
FUNCTION    FT58    =   AND,                        RDCYCS  (ONE,   1)
FUNCTION    FT59    =   OR,                         RDCYCS  (ANS,   2)
FUNCTION    FT63    =   CUT     (1        )
;
```

```
175: ;****************************************
176: ;
177: ;       DATA MEMORY
178: ;
179: ;----------------------------------------
180: ;
181: MEMORY  T1      =       0                                       ;
182: MEMORY  MASK    =       0F000H, 03F00H, 003F0H, 0003FH          ;
183: MEMORY  SHTTBL1 =       12.     8.      4.      0               ;
184: MEMORY  SHTTBL2 =       12.     8.      4.      0               ;
185: MEMORY  LSB1    =       0.      0.      0.      0               ;
186: MEMORY  LSB2    =       0.      0.      0.      0               ;
187: MEMORY  LSB3    =       0.      0.      0.      0               ;
188: MEMORY  ZERO    =       0                                       ;
189: MEMORY  ONE     =       1                                       ;
190: MEMORY  TWO     =       2                                       ;
191: MEMORY  THREE   =       3                                       ;
192: MEMORY  FOUR    =       4                                       ;
193: MEMORY  FH      =       15                                      ;
194: MEMORY  MTB1    =       0FH, 0FH, 0EH, 0EH, 0DH, 0DH, 0CH, 0CH,
195:                         0BH, 0BH, 0AH, 0AH, 09H, 09H, 08H, 08H,
196:                         07H, 07H, 06H, 06H, 05H, 05H, 04H, 04H,
197:                         03H, 03H, 02H, 02H, 01H, 01H, 00H, 00H,
198:                         0FH, 0FH, 0EH, 0EH, 0DH, 0DH, 0CH, 0CH,
199:                         0BH, 0BH, 0AH, 0AH, 09H, 09H, 08H, 08H,
200:                         07H, 07H, 06H, 06H, 05H, 05H, 04H, 04H,
201:                         03H, 03H, 02H, 02H, 01H, 01H, 00H, 00H;
202: MEMORY  MTB2    =       0FH, 0FH, 0FH, 0FH, 0FH, 0EH, 0FH, 0EH,
203:                         0FH, 0FH, 0DH, 0DH, 0FH, 0EH, 0DH, 0CH,
204:                         0FH, 0FH, 0FH, 0FH, 0BH, 0AH, 0BH, 0AH,
205:                         0FH, 0FH, 0DH, 0DH, 0BH, 0AH, 09H, 08H,
206:                         0FH, 0FH, 0FH, 0FH, 0FH, 0EH, 0FH, 0EH,
207:                         07H, 07H, 05H, 05H, 07H, 06H, 05H, 04H,
208:                         0FH, 0FH, 0FH, 0FH, 0BH, 0AH, 0BH, 0AH,
209:                         07H, 07H, 05H, 05H, 03H, 03H, 01H, 00H  ;
210: ;
211: MEMORY  ANS     =       AREA    (2      )       ;
212: MEMORY  QUE1    =       AREA    (1      )       ;
213: MEMORY  QUE2    =       AREA    (1      )       ;
214: MEMORY  QUE3    =       AREA    (1      )       ;
215: MEMORY  QUE4    =       AREA    (1      )       ;
216: MEMORY  QUE5    =       AREA    (1      )       ;
217: MEMORY  QUE6    =       AREA    (1      )       ;
218: MEMORY  QUE7    =       AREA    (1      )       ;
219: MEMORY  QUE8    =       AREA    (16     )       ;
220: MEMORY  QUE9    =       AREA    (4      )       ;
221: MEMORY  QUE10   =       AREA    (4      )       ;
222: MEMORY  QUE11   =       AREA    (4      )       ;
223: MEMORY  QUE12   =       AREA    (1      )       ;
224: MEMORY  QUE13   =       AREA    (1      )       ;
225: MEMORY  QUE14   =       AREA    (1      )       ;
226: MEMORY  QUE15   =       AREA    (1      )       ;
227: ;
228: ;****************************************
229: ;
230: ;       START
231: ;
232: ;----------------------------------------
233: ;
234: START                                               ;
235: ;
236: DATA    EXEC    (IPP,   LDA0,   STARTD  )       ;
237: DATA    EXEC    (IPP,   LSA0,   STARTS  )       ;
238: ;
239: END                                                 ;
```

Appendix A

Image Memory Read/Write

The system addressed in this document performs image memory access through the uPD9305. The contents of access tokens used are listed in Table A-1. For further details consult the "uPD9305 Users' Manual".

Table A-1
MN Values and Token Types

Data Category	MN	ID	Function	Abbr
(1)	0 0 0 0	* * * * * * *	uPD7281 output data to the host computer	CPU
(2)	0 0 0 1	MN' ID'	Image memory read #1 (RHAR1-selected)	IMR
		1 1 1 * * * *	RHAR1 assignment (Note 2)	
	0 0 1 0	MN' ID'	Image memory read #2 (RHAR2-selected)	
		1 1 1 * * * *	RHAR2 assignment (Note 2)	
	0 0 1 1	MN' ID'	Image memory read #3 (RHAR3-selected)	
		1 1 1 * * * *	RHAR3 assignment	
	0 1 0 0	MN' ID'	Image memory read #4 (RHAR4-selected)	
		1 1 1 * * * *	RHAR4 assignment (Note 2)	
	0 1 0 1	0 0 0 0 0 DIR	Image memory write	IMW
		0 0 1 * * DIR	High address assignment for write (selection register: DIR=1)	IMWHA
		0 1 0 * * DIR	Data assignment for write (selection register: DIR=1)	IMWD
		0 1 1 * * DIR	High address assignment for read (selection register: DIR=1)	IMRHA
		1 0 0 Mask DIR	Read-modify-write 1	RMW1
		1 0 1 * * DIR	Read-modify-write 2 (Mask is selected with CS bit of image memory write data)	RMW2
(3)		1 1 0 * * * *	DMA1 (Host -> uPD7281)	DMA1
		1 1 1 * * * *	DMA2 (uPD7281 -> host)	DMA2
(2)	0 1 1 0	0 0 * * * DIR	Self Object Load 1	SOL1
		0 1 * * * DIR	Self Object Load 2 (MN-interchanged)	SOL2
		1 * * * * * *	MN assignment for Self Object Load	SOLN
(4)	0 1 1 1		Module number for uPD7281 (Valid when RHASEL=1)	PASS
	1 0 0 0 1 0 0 1 1 0 1 0 1 0 1 1 1 1 0 0 1 1 0 1 1 1 1 0		Module numbers for uPD7281	
(5)	1 1 1 1		Deletion	VANISH

Note 1: MN' indicates returned MN (MN' ≠ 111).
ID' indicates returned ID.

Note 2: Becomes an image memory read token when RHASEL of the Mode Register is 1.

VOLUME II

GRAY-SCALE IMAGE PROCESSING

This volume addresses gray-scale image processing, one of the application areas of the uPD7281. The following applications are included:

* Binarization
* Continuous-tone transformation
* Dither transformation
* Mask processing
* Affine transformation

These applications are explained in terms of the following subtopics:

* Explanation of processing
* Algorithm
* Allowable ranges of parameters and how to set them
* Flow graph explanation
* Tips on preparing flow graphs
* Assembler source listing

Section 1 System Configuration

The programs in this application library are assumed to run on a system that uses the uPD9305 memory access and general bus interface chip (MAGIC), a uPD7281 peripheral support chip. See figure 1-1. The tokens used to access image memory (IM) are defined by the uPD9305. Consequently, the programs in this manual are not directly usable on a system without a uPD9305.

Although the programs use only one uPD7281, the uPD9305 allows the use of multiple uPD7281s. It is possible to increase the processing speed by partitioning memory and using uPD7281s with the same program in each of the partitions. (Such a program is not directly usable for mask processing due to a problem of boundary areas, which occurs if memory is partitioned.) When employing this technique, the token from a particular memory area is directed to the uPD7281 operating on that area by setting the module number (MN) of the token to that of the uPD7281. Alternatively, the algorithm can be partitioned among several uPD7281s to increase the processing speed.

Figure 1-1. System Configuration

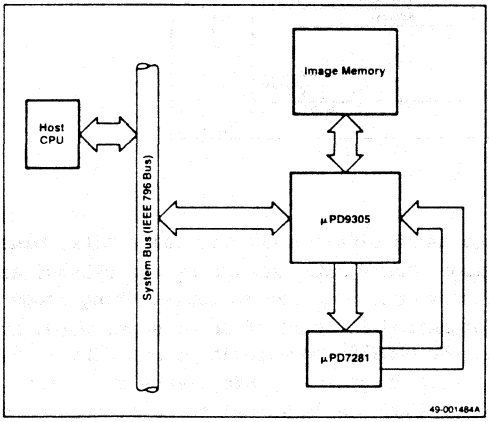
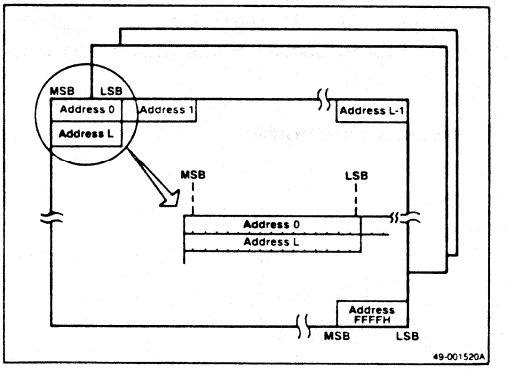


IMAGE MEMORY ORGANIZATION

Figure 1-2 shows the image memory organization of the system this document addresses. If memory is pictured as a display screen, the upper left corner of the screen will correspond to the most significant bit (MSB), address 0000H. The lower right corner will correspond to the least significant bit, address FFFFH.

In this system, addresses 0000H to FFFFH comprise a bank. There are eight banks, and it is possible to switch among them by setting the upper eight bits of the uPD9305's 24-bit image memory address. These eight bits are the bank-setting read high address (RHA) or write high address (WHA).

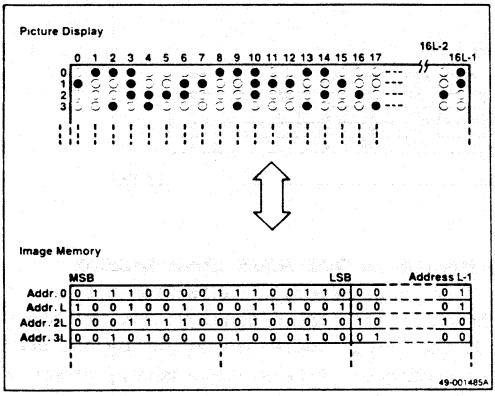
Figure 1-2. Image Memory Organization



In the uPD7281, a word defined as 18 bits: 16 numeric value bits, plus the C and S bits. Although the image memory data handled by the uPD9305 are also 18 bits, the C and S bits are not required in representing image data. Consequently, the memory is organized in terms of 16 bits per word, as shown in figure 1-2. Thus, if the uPD9305 mode register bit CSSEL = 0, the C and S bits of the data read from image memory are always zero. (See the uPD7281 User's Manual and the uPD9305 User's Manual for a discussion of the C, S, and CSSEL bits.)

In binary images, the memory and display are related as shown in figure 1-3. Specifically, a pixel corresponds to a bit in memory, with one word (16 bits) required to represent 16 pixels. Therefore, the amount of image memory available in a bank is sufficient to represent 1,048,576H pixels. If 1,024 pixels are present in a horizonal line, a bank can represent 1,024 x 1,024 pixels.

Figure 1-3. Relationship Between Image Memory and a Binary Picture with 16 x L Horizontal Pixels



The applications presented in this manual assume the display and memory to be related as shown in figure 1-4. In contrast to the binary image representation, in these applications a word is used to represent a pixel, a pixel being eight bits, with 256 gradations or gray scale values. The upper eight bits of a word (16 bits) are ignored. Therefore, a bank of image memory is required to represent 65,536H pixels of images. If the number of horizontal pixels is 256, a bank can represent an image 256 x 256 pixels in size.

Figure 1-4. Image Memory and a Gray Scale Image with L Horizontal Pixels

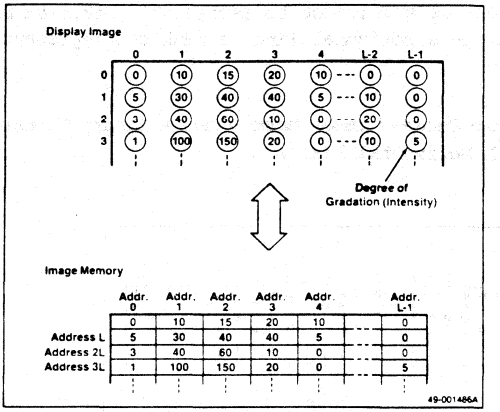


IMAGE MEMORY ORGANIZATION FOR USE IN GRAY SCALE IMAGE DISPLAY

Figure 1-5 is an example of an image memory organization for gray scale images. As shown, the eight banks are placed contiguously, with a word comprised of eight bits. Of the 24-bit addresses output by the uPD9305, the upper five bits specify the contiguous 8-bank image memory group. Within the eight banks, each bank is specified with the lower three bits. The remaining 16 bits are used for local addresses within a bank.

Suppose, for example, that data representing 10003H are output from the uPD9305 to the address bus. Then on the image memory, the memory location corresponding to address 2000H on bank 3 will be accessed. Thus, the bank boundaries viewed by the uPD9305 are different from the actual bank boundaries, but this is transparent to the user. (This type of memory organization allows for greater convenience of display.)

In a binary image display, a single read of the image memory fetched 16 pixels, corresponding to a memory access rate that is 1/16 the display cycle of a pixel. This increased the memory access cycle time for display, and reduced the amount of time the uPD9305 had to wait during memory access because of a display operation, thereby speeding up the uPD9305 memory access.

In gray scale image display, a memory read fetches one pixel. Therefore, memory reads must be performed at the same cycle as the display cycle, requiring a large amount of fast memory. Even if this condition is met, it would be difficult for the uPD9305 to access memory at the same time as a display cycle. The memory organization shown in figure 1-5 is designed to overcome this difficulty. Figure 1-6 is a summary of the display portion.

For the memory read on the display side, all eight banks are specified, and using 16-bit local addresses, eight words of data are set in eight 8-bit shift registers in a single operation. A data byte from bank 0 is placed as follows:

Bit 0 goes to shift register 0 bit 0.
Bit 1 goes to shift register 1 bit 0.
Bit 2 goes to shift register 2 bit 0.
.
.
Bit 7 goes to shift register 7 bit 0.

Thus, if the shift registers are viewed as shown in figure 1-6, stacked one upon the other, each of the bytes from the banks would be vertical; a byte from bank 0 is placed in the bit 0 positions of the eight shift registers (so that the byte will be shifted out all at once). Likewise, a byte from bank 1 is placed in the bit 1 positions of the eight shift registers, and so on, up to a byte from bank 7 being placed into the bit 7 positions of the shift registers.

Looking at the shift registers (horizontal view), shift register 0 will contain data from the bit 0 positions of the bytes from the banks, shift register 1 will contain data from the bit 1 positions of bytes from the banks, and so on, to shift register 7 containing data from the bit 7 positions of bytes from the banks. For example, figure 1-7 shows how data are set in shift register 3.

Each shift register shifts in synchronization with the display cycle. Thus, data output from the shift registers are equivalent to the 8-bit data that are read from the banks. Further, the data output from the eight shift registers are converted through a D/A converter to analog values, so they can be input to the display unit.

Reading eight words of data at once and displaying eight pixels, as described above, reduces the memory access for display to 1/8 the display cycle for a pixel, resulting in a lower cost system configuration.

Figure 1-5. Gray Scale Image Memory Organization Example

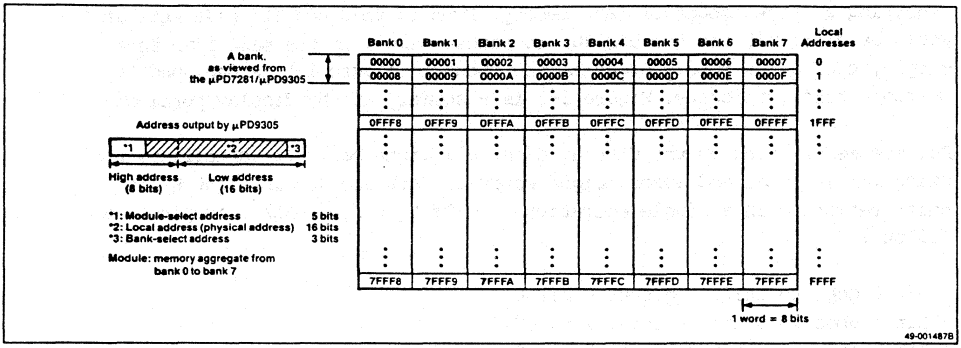


Figure 1-6. Image Memory System Block Diagram

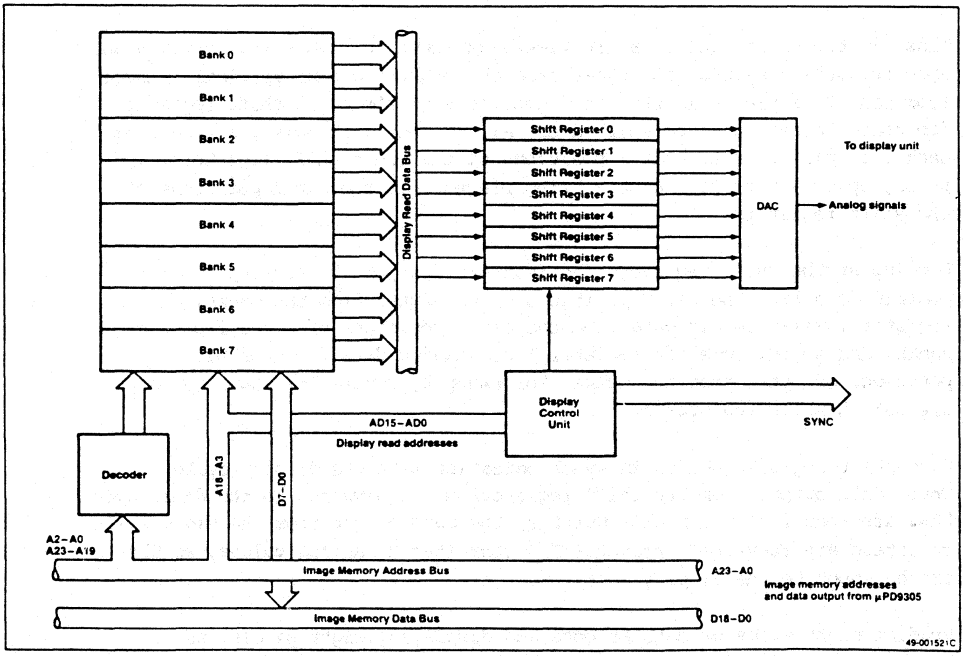
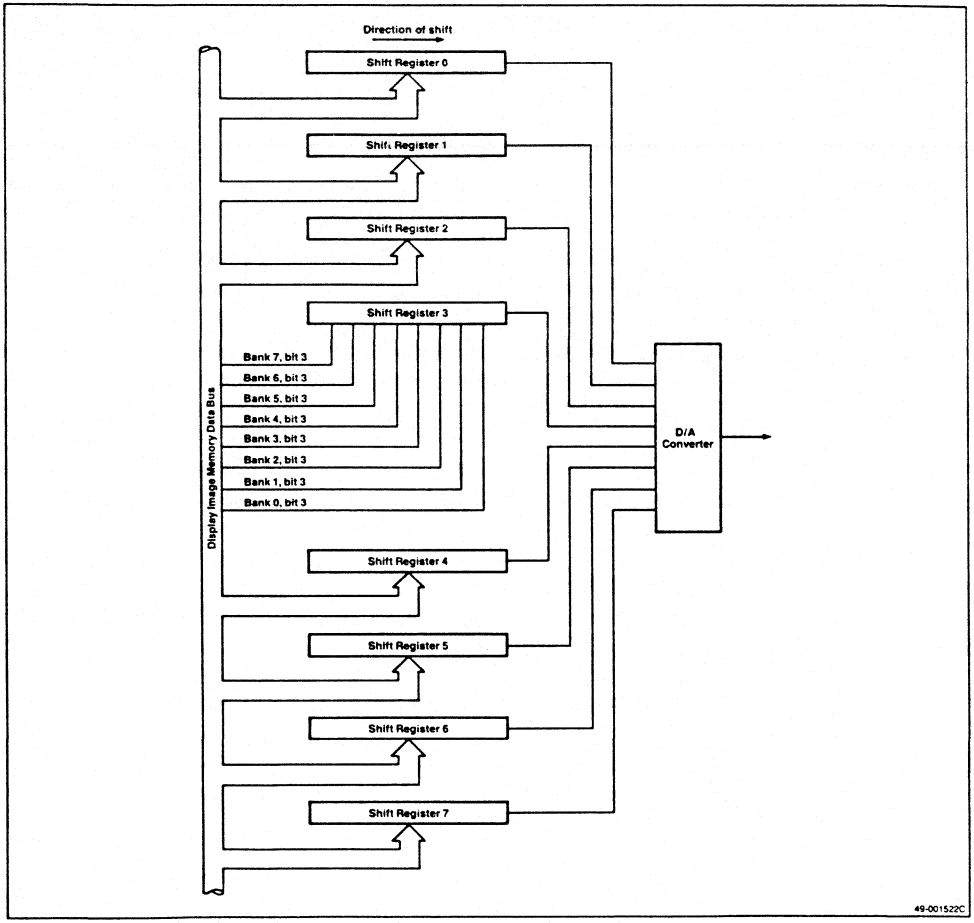
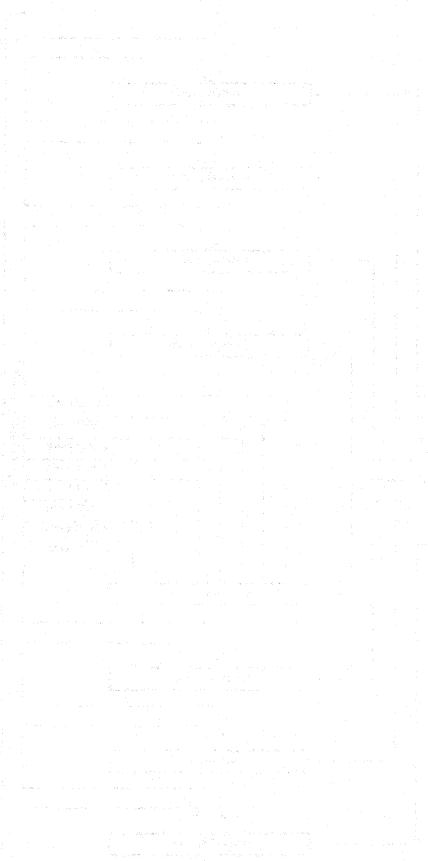


Figure 1-7. Shift Registers and Data Bus





Section 2 Binarization

Binarization is the conversion of gray scale image data, represented by eight bits per pixel (gradations of gray), into binary data where each pixel is represented by either 0 or 1 (black or white).

ALGORITHM

This program assumes that the image area to be processed consists of a maximum of 256 x 256 pixels. It takes into account the use of read/write bank switching with respect to image memory. Image memory is assumed to be organized as shown in figure 2-1.

It is also assumed that each pixel in the gray scale data is set in the lower byte (8 bits) of an image memory word, and that the binary image data resulting from the conversion are made up of units of words (16 pixels per word). Therefore, to create a unit of binary image data, 16 horizontally contiguous words of gray scale data are required.

This program is comprised of the following functions: source (SRC) image area address generation; SRC data read along with destination (DST) image data generation; DST address generation; and DST data write.

SRC area addresses are generated 16 addresses at a time in the horizontal direction, and are used to read the SRC data. For each 16 SRC addresses generated, one DST area address is generated in the horizontal direction. In the bank switching for reading or writing image memory data, the read- or write-high addresses are rewritten whenever a low-address generated exceeds 0FFFFH.

Each unit of 16-pixel SRC data (gray scale image data) read from the SRC area is compared with a corresponding threshold value. If it is higher than the threshold value, it is replaced with a 1; otherwise it is replaced with 0. The DST data are generated by packing these 16 bits of data into one word. See figure 2-2.

Figure 2-1. Image Memory Organization

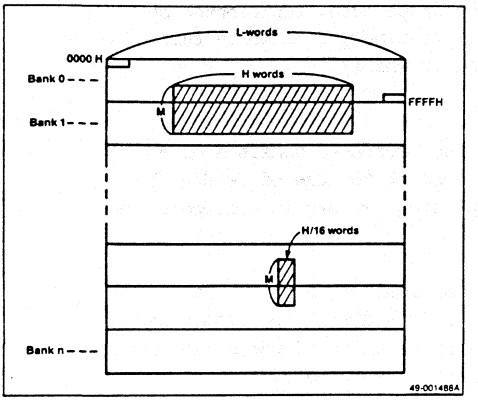
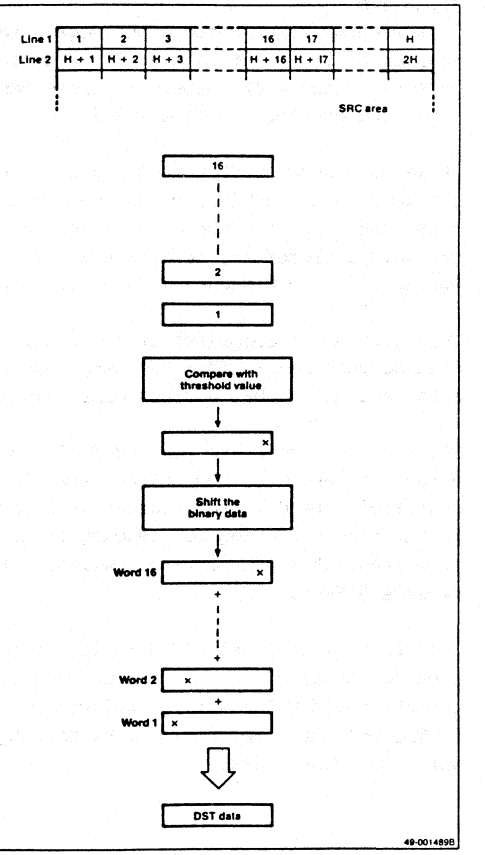


Figure 2-2. DST Data Generation



PARAMETERS

Table 2-1 lists allowable values for the parameters, which are defined as follows.

Assembler-Coded Parameters

L - Number of image memory horizontal words.
H - Number of SRC image area horizontal words.
M - Number of SRC image area vertical lines.

Start-Up Token Defined Parameters

THRVAL - Threshold value.
RHADR - Read-high address of SRC image area starting address.
WHADR - Write-high address of DST image area starting address.
SRCADR - SRC image area starting address.
DSTADR - DST image area starting address.

Table 2-1. Parameter Values

Parameter	Range Min	Range Max	Value in Program Example
L	0	65,535	1024
H	A multiple of 16, in the range 16 - 4096		16
M	1	256	256
THRVAL	0	255	80H
RHADR	0	65,535	0
WHADR	0	65,535	10H
SRCADR	0	65,535	0
DSTADR	0	65,535	0

FLOW GRAPH EXPLANATION

Figure 2-3 is a flow graph depicting binarization processing. Binarization involves the conversion of 16-bit data into 1-bit units, with the DST data comprised of 16 1-bit units. Therefore, the horizontal scale, H, for the SRC area is in multiples of 16, and that for the DST area is H/16. (Both areas have the same vertical scale.)

In terms of address generation, both areas generate the starting address for a vertical line each time a horizontal line is generated. Then, the SRC generates 16 addresses for each horizontal line based on the starting address for that line. The DST generates one address for every 16 units of SRC data read.

For the image memory bank switching, node FINCL checks for overflow, and updates the low addresses as well as the write-high addresses.

Once read, the 16 units of SRC data are masked on their upper byte and compared with the threshold value. If the SRC data are less than the thresholdvalue, they are replaced with a 1; otherwise they are replaced with 0. Each of the 16 bits that comprise DST data is placed in its position in a word through shift and addition operations.

Table 2-2 explains the principal nodes.

Table 2-2. Principal Nodes

Node	Function
FTHRE	Sets threshold values.
FREDHA	Sets the initial values of image memory read-high addresses.
FWRTHA	Sets the initial values of image memory write-high addresses.
FINCL	Updates read-low addresses and checks for overflows.
FINCRH, FRHA, FSRHAI	Update read-high addresses.
FGEN1	Generates the starting addresses of lines in the SRC area.
FGEN2	Generates addresses for SRC area horizontal lines. 16 addresses per line.
FREAD	Reads SRC data.
FAND, FCMP	Mask, then compare data for a pixel with a threshold value, and convert them to 0's and 1's.
FSHL, FPACK	Place 16-bit data in their respective positions in a word to generate the DST data.
FGEN3	Generates starting addresses for lines in the DST area.
FGEN4	Each time SRC data are read, this node generates 16 DST area horizontal line addresses.

FINCWH Update read-high addresses.
FWHA,
FSWHAI

FWRITE Writes DST data to the DST area.

Figure 2-3. Binarization Flow Graph (Top)

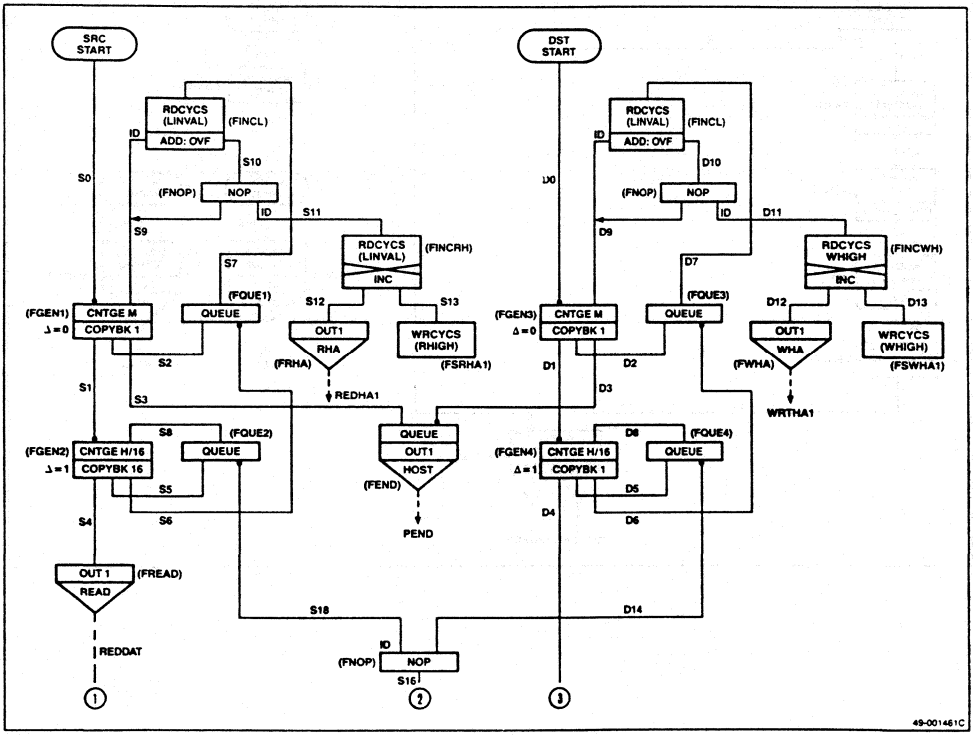
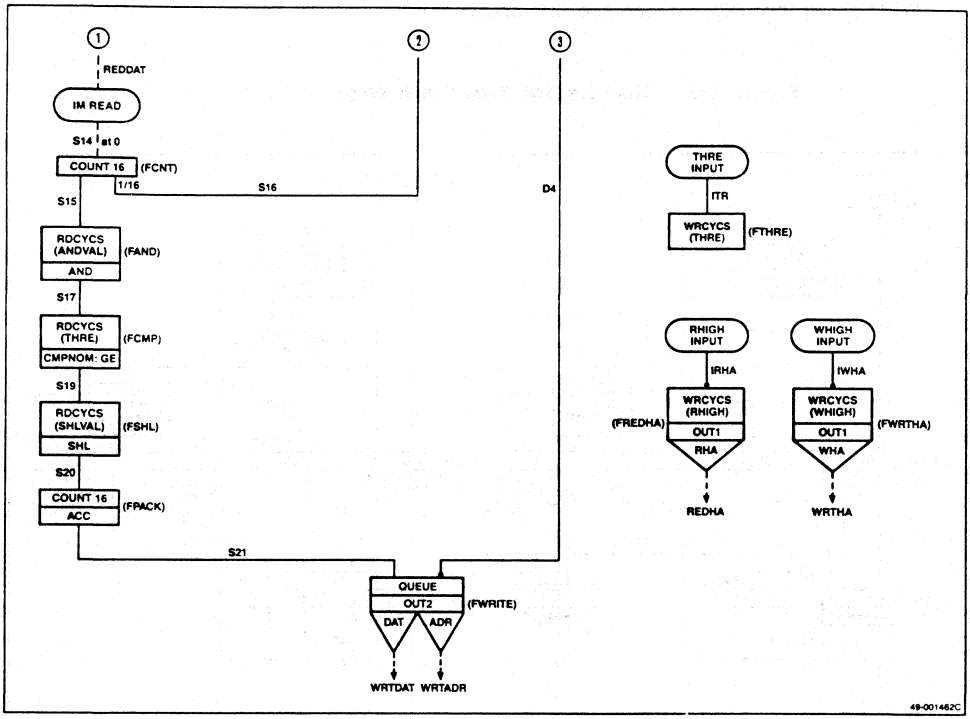


Figure 2-3. Binarization Flow Graph (Bottom)



TIPS ON PREPARING FLOW GRAPHS

When an arc is to be shared, attention should be paid to other output arc IDs of different nodes having common arcs (same IDs).

Figure 2-4 shows an example where the node FNOP1 shown in figure 2-3 has its first and second IDs reversed, so that first output IDs are sharing arc S9 instead of the second output ID.

The ID following S9 in figure 2-4, when viewed from FINCL, is S10, while the ID following S9 as viewed from FNOP is S11. Thus, there are two IDs following S9. When viewed in the flow graph, these two arcs appear to be different IDs, but at the program object level they must be the same. Further, any given ID may have two or more IDs before and after it. Therefore, a flow graph of the type shown in figure 2-4 is invalid.

In the flow graph shown in figure 2-3, the ID preceding S9 is S11, and the ID following it is S10; there is no possibility of different output IDs existing in duplicate. The same is true of D10 and D11.

Figure 2-4. An Arc Shared between Different First Output IDs

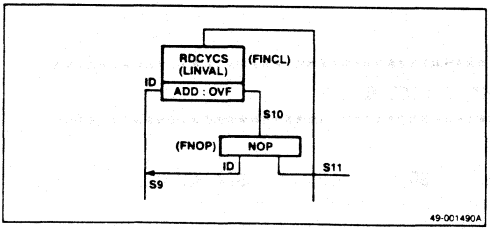


Figure 2-5. Binarization Source Program

```
;*****************************************************************
;
;                        BINARIZATION
;
;*****************************************************************
;
MODULE  IMPP    =       8                                       ;
;
EQUATE  L       =       1024                                    ;
EQUATE  H       =       256                                     ;
EQUATE  M       =       256                                     ;
;
EQUATE  THRVAL  =       80H                                     ;
EQUATE  RHADR   =       0                                       ;
EQUATE  WHADR   =       0                                       ;
EQUATE  SRCADR  =       0                                       ;
EQUATE  DSTADR  =       256                                     ;
;
EQUATE  FINE    =       0                                       ;
EQUATE  READ    =       4                                       ;
EQUATE  WRITE   =       5                                       ;
;
;*****************************************************************
;                        INPUT & OUTPUT
;*****************************************************************
;
INPUT   IRHA,   IWHA,   ITR,    S0,     D0,     S14 AT 0        ;
;
OUTPUT  REDHA,  WRTHA,  REDHA1, WRTHA1, PEND,   WRTDAT          ;
OUTPUT  WRTADR, REDDAT                                          ;
;
```

Figure 2-5. Binarization Source Program (cont)

```
;*****************************************************************
;                          LINK
;*****************************************************************
;
LINK    S1,     S2,     S3      = FGEN1  (S9,    S0   )      ;
LINK    S4,     S5,     S6      = FGEN2  (S8,    S1   )      ;
LINK    S7                      = FQUE1  (S2,    S6   )      ;
LINK    PEND                    = FEND   (S3,    D3   )      ;
LINK    REDDAT                  = FREAD  (S4          )      ;
LINK    S8                      = FQUE2  (S5,    S18  )      ;
LINK    S9,     S10             = FINCL  (S7          )      ;
LINK    S11,    S9              = FNOP   (S10         )      ;
LINK    S12,    S13             = FINCRH (S11         )      ;
LINK    REDHA1                  = FRHA   (S12         )      ;
LINK                            = FSRHA1 (S13         )      ;
LINK    S15,    S16             = FCNT   (S14         )      ;
LINK    S17                     = FAND   (S15         )      ;
LINK    S18,    D14             = FNOP   (S16         )      ;
LINK    S19                     = FCMP   (S17         )      ;
LINK    S20                     = FSHL   (S19         )      ;
LINK    S21                     = FPACK  (S20         )      ;
LINK    D1,     D2,     D3      = FGEN3  (D9,    D0   )      ;
LINK    D4,     D5,     D6      = FGEN4  (D8,    D1   )      ;
LINK    D7                      = FQUE3  (D2,    D6   )      ;
LINK    D8                      = FQUE4  (D5,    D14  )      ;
LINK    D9,     D10             = FINCL  (D7          )      ;
LINK    D11,    D9              = FNOP   (D10         )      ;
LINK    D12,    D13             = FINCWH (D11         )      ;
LINK    WRTHA1                  = FWHA   (D12         )      ;
LINK                            = FSWHA1 (D13         )      ;
LINK    WRTDAT, WRTADR          = FWRITE (S21,   D4   )      ;
LINK    SETRHA                  = FSRHA  (    ,  IRHA )      ;
LINK    REDHA                   = FREDHA (SETRHA      )      ;
LINK    SETWHA                  = FSWHA  (    ,  IWHA )      ;
LINK    WRTHA                   = FWRTHA (SETWHA      )      ;
LINK                            = FTHRE  (ITR         )      ;
;
```

Figure 2-5. Binarization Source Program (cont)

```
;*****************************************************************
;                        FUNCTION
;*****************************************************************
;
FUNCTION        FEND    = OUT1   (FINE,    0), QUEUE  (QUEE, 1) ;
FUNCTION        FREDHA  = OUT1   (READ,  70H)                    ;
FUNCTION        FRHA    = OUT1   (READ,  70H)                    ;
FUNCTION        FREAD   = OUT1   (READ,    0)                    ;
FUNCTION        FWRTHA  = OUT1   (WRITE, 10H)                    ;
FUNCTION        FWHA    = OUT1   (WRITE, 10H)                    ;
FUNCTION        FWRITE  = OUT2  (WRITE,20H,0), QUEUE  (QUEW,16) ;
FUNCTION        FGEN1   = COPYBK ( 1,      0), CNTGE  (M      ) ;
FUNCTION        FGEN2   = COPYBK (16,      1), CNTGE  (H/16   ) ;
FUNCTION        FGEN3   = COPYBK ( 1,      0), CNTGE  (M      ) ;
FUNCTION        FGEN4   = COPYBK ( 1,      1), CNTGE  (H/16   ) ;
FUNCTION        FQUE1   = QUEUE  (QUE1,    1)                    ;
FUNCTION        FQUE2   = QUEUE  (QUE2,    1)                    ;
FUNCTION        FQUE3   = QUEUE  (QUE3,    1)                    ;
FUNCTION        FQUE4   = QUEUE  (QUE4,    1)                    ;
FUNCTION        FNOP    = NOP    (XX       )                    ;
FUNCTION        FINCL   = ADD   (X, OVF, BRC), RDCYCS (LINVAL,1);
FUNCTION        FINCRH  = INC    (XX,    XCH), RDCYCS (RHIGH, 1);
FUNCTION        FINCWH  = INC    (XX,    XCH), RDCYCS (WHIGH, 1);
FUNCTION        FSRHA1  = WRCYCS (RHIGH,   1)                    ;
FUNCTION        FSWHA1  = WRCYCS (WHIGH,   1)                    ;
FUNCTION        FTHRE   = WRCYCS (THRE,    1)                    ;
FUNCTION        FCNT    = COUNT  (16       )                    ;
FUNCTION        FAND    = AND    (X        ), RDCYCS (ANDVAL,1);
FUNCTION        FCMP    = CMPNOM (X,      GE), RDCYCS (THRE,  1);
FUNCTION        FSHL    = SHL    (X        ), RDCYCS(SHLVAL,16);
FUNCTION        FPACK   = ACC    (X        ), COUNT  (16      );
;
```

Figure 2-5. Binarization Source Program (cont)

```
;*****************************************************************
;                          MEMORY
;*****************************************************************
;
MEMORY  QUE1    = AREA ( 1)                                        ;
MEMORY  QUE2    = AREA ( 1)                                        ;
MEMORY  QUE3    = AREA ( 1)                                        ;
MEMORY  QUE4    = AREA ( 1)                                        ;
MEMORY  QUEE    = AREA ( 1)                                        ;
MEMORY  QUEW    = AREA (16)                                        ;
MEMORY  RHIGH   = AREA ( 1)                                        ;
MEMORY  WHIGH   = AREA ( 1)                                        ;
MEMORY  THRE    = AREA ( 1)                                        ;
MEMORY  LINVAL  = L                                                ;
MEMORY  ANDVAL  = 0FFH                                             ;
MEMORY  SHLVAL  = 15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0            ;
;
;*****************************************************************
;                          START
;*****************************************************************
;
START                                                              ;
DATA    EXEC    (IMPP,  ITR,    THRVAL)                          ;
DATA    EXEC    (IMPP,  IRHA,   RHADR )                          ;
DATA    EXEC    (IMPP,  IWHA,   WHADR )                          ;
DATA    EXEC    (IMPP,  D0,     DSTADR)                          ;
DATA    EXEC    (IMPP,  S0,     SRCADR)                          ;
;
END;
```

Section 3. Continuous-Tone Transformation

Continuous-tone transformation is the conversion of gray scale data, in which the tone of a pixel is expressed in eight bits, into arbitrary gray-scale data displayable in eight bits. Such data conversion permits the data to be processed for simulated color, uniform-tone displays, and contrast enhancement.

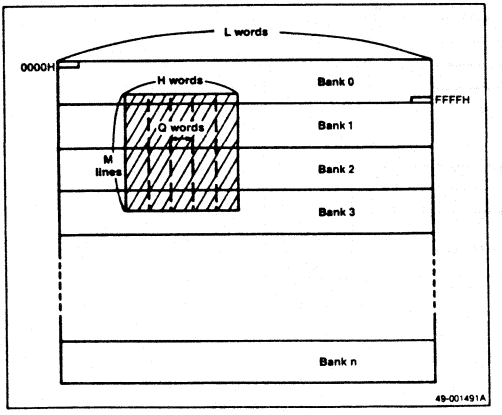
ALGORITHM

This program assumes that data for a pixel are located in the lower byte of an image memory word. The program converts data from an image area consisting of 256 x 256 pixels. Like the binarization program, it provides for image memory bank switching. See figure 3-1.

For the generation of image area addresses, 16 addresses per horizontal line are generated for each of the SRC and DST areas. They are to be used for reading SRC data and writing DST data.

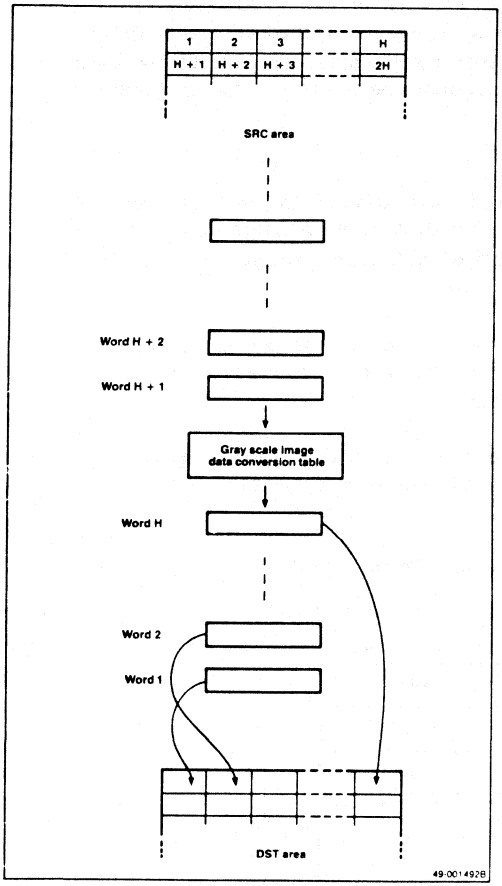
In updating low addresses for the image memory bank switching, a read or write address is updated whenever a low address exceeds 0FFFFH.

Figure 3-1. Image Memory Organization



To generate DST data, a gray scale data conversion table is referenced using the lower byte value of the SRC data that have been read, and the converted data are read from the table. A gray scale data conversion is one in which the lower byte of each word in a 256-word area is used to define image tone.

Figure 3-2. Procedures for DST Data Generation



PARAMETERS

Table 3-1 lists allowable values for the parameters, which are defined as follows.

Assembler-Coded Parameters

L - Number of image memory horizontal words.
H - Number of SRC image area horizontal words.
Q - Number of words read contiguously across from the SRC image area.
M - Number of SRC image area vertical lines.

Start-Up Token Defined Parameters

RHADR - Read-high address for SRC image area starting address.
WHADR - Write-high address for DST image area starting address.
SRCADR - Read-low address for SRC image area starting address.
DSTADR - Write-low address for DST image area starting address.

Table 3-1. Parameter Values

Parameter	Range Min	Range Max	Value in Program Example
L	0	65,535	1024
H	Q	256 x Q	256
Q	1	16	16
M	1	256	256
RHADR	0	65,535	0
WHADR	0	65,535	0
SRCADR	0	65,535	0
DSTADR	0	65,535	256

FLOW GRAPH EXPLANATION

A flow graph for continuous-tone transformation is shown in figure 3-3.

The size of the SRC and DST areas can be represented by H x M. The addresses for these areas are generated as follows. First, the starting address of a vertical line is created each time the addresses for a horizontal line are generated. Using that starting address, the addresses for the horizontal line are divided using created Q addresses and the formula H/Q. By setting an appropriate value for Q, any rectangular area from 1 x 1 through 256 x 256 can be processed.

In this program, a separate node is provided for updating the vertical addresses of each area; the nodes are also used to detect overflows. Image memory bank switching is performed by updating the read and write high addresses whenever an overflow is detected.

By masking the higher byte, data corresponding to one pixel are extracted from the lower byte of each word of the SRC data read. The mask data are converted to DST data through the use of the gray scale data conversion table. Table 3-2 explains the principal nodes.

Table 3-2. Principal Nodes

Node	Function
FREDHA	Sets the initial values of image memory read-high addresses.
FWRTHA	Sets the initial values of image memory write-high addresses.
FGEN1	Generates the starting addresses of lines in the SRC area.
FGEN2	Generates addresses for SRC area horizontal lines, Q addresses at a time.
FINCL	Updates read-low and write-low addresses and checks for overflows.
FINCRH, FRHA, FSRHA	Update SRC area read-high addresses.
FREAD	Reads SRC data.
FAND, FRIDX	Extract the lower byte from SRC data, and read the converted data using the gray-scale data conversion table.
FGEN3	Generates starting addresses for lines in the DST area.
FGEN4	Generates addresses for DST area horizontal lines, Q addresses/line.
FINCWH FWHA, FSWHA	Update DST area write high addresses.
FWRITE	Writes DST data to the DST area.

Figure 3-3. Continuous-Tone Transformation Flow Graph (Top)

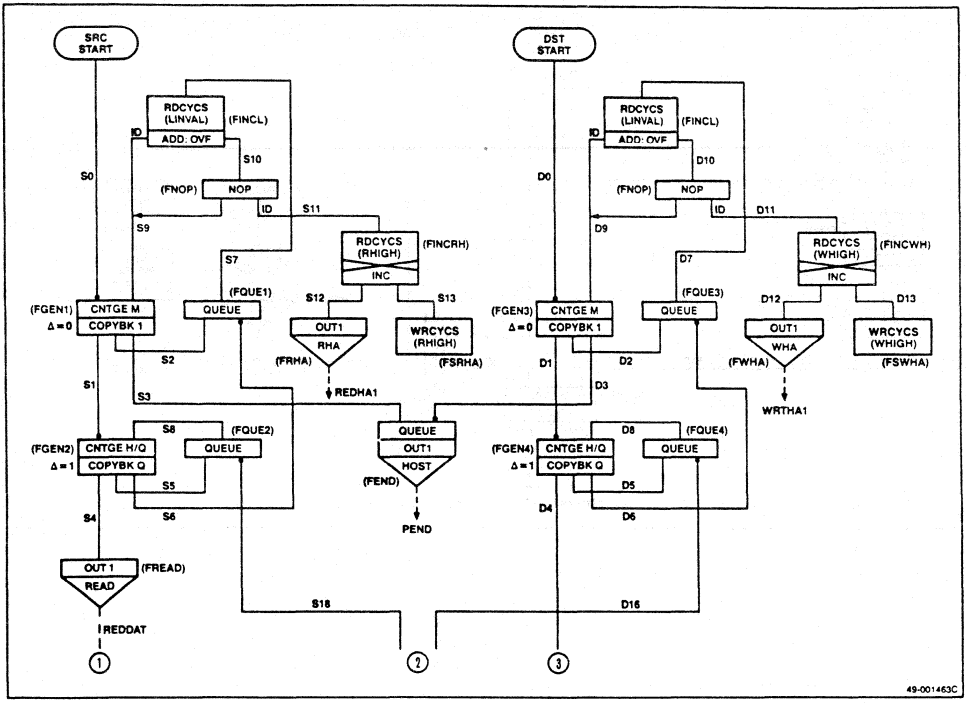


Figure 3-3. Continuous-Tone Transformation Flow Graph (Bottom)

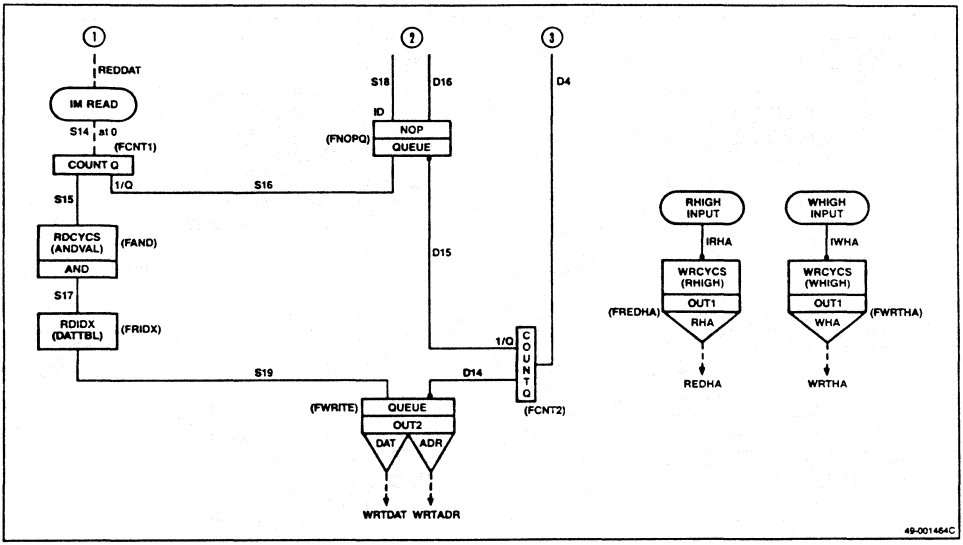


Figure 3-4. Continuous-Tone Transformation Source Program

```
;*****************************************************************
;
;                    GRAY SCALE CONVERSION
;
;*****************************************************************
;
MODULE  IMPP    =       8                                       ;
;
EQUATE  L       =       1024                                    ;
EQUATE  H       =       256                                     ;
EQUATE  Q       =       16                                      ;
EQUATE  M       =       256                                     ;
;
EQUATE  RHADR   =       0                                       ;
EQUATE  WHADR   =       0                                       ;
EQUATE  SRCADR  =       0                                       ;
EQUATE  DSTADR  =       256                                     ;
;
EQUATE  HOST    =       0                                       ;
EQUATE  READ    =       1                                       ;
EQUATE  WRITE   =       5                                       ;
;
;*****************************************************************
;                       INPUT - OUTPUT
;*****************************************************************
;
INPUT   IRHA,   IWHA,   S0,     D0,     S14 AT 0                ;
;
OUTPUT  REDHA,  WRTHA,  REDHA1, WRTHA1, PEND,   WRTDAT          ;
OUTPUT  WRTADR, REDDAT                                          ;
;
```

Figure 3-4. Continuous-Tone Transformation Source Program (cont)

```
;*****************************************************************
;                        LINK TABLE
;*****************************************************************
;
LINK    S1,     S2,     S3      = FGEN1  (S9,    S0   )      ;
LINK    S4,     S5,     S6      = FGEN2  (S8,    S1   )      ;
LINK    S7                      = FQUE1  (S2,    S6   )      ;
LINK    PEND                    = FEND   (S3,    D3   )      ;
LINK    REDDAT                  = FREAD  (S4          )      ;
LINK    S8                      = FQUE2  (S5,    S18  )      ;
LINK    S9,     S10             = FINCL1 (S7          )      ;
LINK    S11,    S9              = FNOP1  (S10         )      ;
LINK    S12,    S13             = FINCRH (S11         )      ;
LINK    REDHA1                  = FRHA   (S12         )      ;
LINK                            = FSRHA  (S13         )      ;
LINK    S15,    S16             = FCNT1  (S14         )      ;
LINK    S17                     = FAND   (S15         )      ;
LINK    S18,    D16             = FNOPQ  (S16,   D15  )      ;
LINK    S19                     = FRIDX  (S17         )      ;
LINK    D1,     D2,     D3      = FGEN3  (D9,    D0   )      ;
LINK    D4,     D5,     D6      = FGEN4  (D8,    D1   )      ;
LINK    D7                      = FQUE3  (D2,    D6   )      ;
LINK    D8                      = FQUE4  (D5,    D16  )      ;
LINK    D9,     D10             = FINCL  (D7          )      ;
LINK    D11,    D9              = FNOP   (D10         )      ;
LINK    D12,    D13             = FINCWH (D11         )      ;
LINK    D14,    D15             = FCNT2  (D4          )      ;
LINK    WRTHA1                  = FWHA   (D12         )      ;
LINK                            = FSWHA  (D13         )      ;
LINK    WRTDAT, WRTADR          = FWRITE (S19,   D14  )      ;
LINK    REDHA                   = FREDHA (      ,IRHA )      ;
LINK    WRTHA                   = FWRTHA (      ,IWHA )      ;
;
```

Figure 3-4. Continuous-Tone Transformation Source Program (cont)

```
;*****************************************************************
;                        FUNCTION
;*****************************************************************
;
FUNCTION        FEND    = OUT1   (HOST,    0), QUEUE  (QUEE, 1) ;
FUNCTION        FREDHA  = OUT1   (READ,  70H), WRCYCS (RHIGH, 1) ;
FUNCTION        FRHA    = OUT1   (READ,  70H)                    ;
FUNCTION        FREAD   = OUT1   (READ,    0)                    ;
FUNCTION        FWRTHA  = OUT1   (WRITE, 10H), WRCYCS (WHIGH, 1) ;
FUNCTION        FWHA    = OUT1   (WRITE, 10H)                    ;
FUNCTION        FWRITE  = OUT2   (WRITE,20H,0), QUEUE  (QUEW,16) ;
FUNCTION        FGEN1   = COPYBK ( 1,      0), CNTGE  (M      ) ;
FUNCTION        FGEN2   = COPYBK ( Q,      1), CNTGE  (E/Q    ) ;
FUNCTION        FGEN3   = COPYBK ( 1,      0), CNTGE  (M      ) ;
FUNCTION        FGEN4   = COPYBK ( Q,      0), CNTGE  (E/Q    ) ;
FUNCTION        FQUE1   = QUEUE  (QUE1,    1)                    ;
FUNCTION        FQUE2   = QUEUE  (QUE2,    1)                    ;
FUNCTION        FQUE3   = QUEUE  (QUE3,    1)                    ;
FUNCTION        FQUE4   = QUEUE  (QUE4,    1)                    ;
FUNCTION        FNOPQ   = NOP    (XX      ), QUEUE  (QUEN, 1) ;
FUNCTION        FNOP    = NOP    (XX      )                    ;
FUNCTION        FINCL1  = ADD    (OVF, BRC), RDCYCS (LINVAL,1) ;
FUNCTION        FINCL2  = ADD   (X, OVF, BRC), RDCYCS (LINVAL,1);
FUNCTION        FINCRH  = INC    (XX,    XCH), RDCYCS (RHIGH, 1);
FUNCTION        FINCWH  = INC    (XX,    XCH), RDCYCS (WHIGH, 1);
FUNCTION        FSRHA   = WRCYCS (RHIGH,   1)                    ;
FUNCTION        FSWHA   = WRCYCS (WHIGH,   1)                    ;
FUNCTION        FCNT1   = COUNT  (Q        )                     ;
FUNCTION        FCNT2   = COUNT  (Q        )                     ;
FUNCTION        FAND    = AND,                 RDCYCS (ANDVAL,1);
FUNCTION        FRIDX   = RDIDX  (DATTBL   )                     ;
;
```

Figure 3-4. Continuous-Tone Transformation Source Program (cont)

```
;*****************************************************************
;                          DATA MEMORY
;*****************************************************************
;
MEMORY  QUE1    = AREA ( 1)                                        ;
MEMORY  QUE2    = AREA ( 1)                                        ;
MEMORY  QUE3    = AREA ( 1)                                        ;
MEMORY  QUE4    = AREA ( 1)                                        ;
MEMORY  QUEN    = AREA ( 1)                                        ;
MEMORY  QUEE    = AREA ( 1)                                        ;
MEMORY  QUEW    = AREA (16)                                        ;
MEMORY  RHIGH   = AREA ( 1)                                        ;
MEMORY  WHIGH   = AREA ( 1)                                        ;
MEMORY  LINVAL  = L                                                ;
MEMORY  ANDVAL  = 0FFE                                             ;
MEMORY  DATTBL  = 000H,000H,000H,000H,000H,000H,000H,000H,
                  000H,000H,000H,000H,000H,000H,000H,000H,
                  011H,011H,011H,011H,011H,011H,011H,011H,
                  011H,011H,011H,011H,011H,011H,011H,011H,
                  022H,022H,022H,022H,022H,022H,022H,022H,
                  022H,022H,022H,022H,022H,022H,022H,022H,
                  033H,033H,033H,033H,033H,033H,033H,033H,
                  033H,033H,033H,033H,033H,033H,033H,033H,
                  044H,044H,044H,044H,044H,044H,044H,044H,
                  044H,044H,044H,044H,044H,044H,044H,044H,
                  055H,055H,055H,055H,055H,055H,055H,055H,
                  055H,055H,055H,055H,055H,055H,055H,055H,
                  066H,066H,066H,066H,066H,066H,066H,066H,
                  066H,066H,066H,066H,066H,066H,066H,066H,
                  077H,077H,077H,077H,077H,077H,077H,077H,
                  077H,077H,077H,077H,077H,077H,077H,077H,
                  088H,088H,088H,088H,088H,088H,088H,088H,
                  088H,088H,088H,088H,088H,088H,088H,088H,
                  099H,099H,099H,099H,099H,099H,099H,099H,
                  099H,099H,099H,099H,099H,099H,099H,099H,
                  0AAH,0AAH,0AAH,0AAH,0AAH,0AAH,0AAH,0AAH,
                  0AAH,0AAH,0AAH,0AAH,0AAH,0AAH,0AAH,0AAH,
                  0BBH,0BBH,0BBH,0BBH,0BBH,0BBH,0BBH,0BBH,
                  0BBH,0BBH,0BBH,0BBH,0BBH,0BBH,0BBH,0BBH,
                  0CCH,0CCH,0CCH,0CCH,0CCH,0CCH,0CCH,0CCH,
                  0CCH,0CCH,0CCH,0CCH,0CCH,0CCH,0CCH,0CCH,
                  0DDH,0DDH,0DDH,0DDH,0DDH,0DDH,0DDH,0DDH,
```

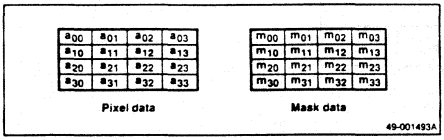
Figure 3-4. Continuous-Tone Transformation Source Program (cont)

```
                0DDH,0DDH,0DDH,0DDH,0DDH,0DDH,0DDH,0DDH,
                0EEH,0EEH,0EEH,0EEH,0EEH,0EEH,0EEH,0EEH,
                0EEH,0EEH,0EEH,0EEH,0EEH,0EEH,0EEH,0EEH,
                0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,
                0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH      ;
;
;*****************************************************************
;                       START
;*****************************************************************
;
START                                                           ;
DATA    EXEC    (IMPP,  IRHA,   RHADR )                         ;
DATA    EXEC    (IMPP,  IWHA,   WHADR )                         ;
DATA    EXEC    (IMPP,  S0,     SRCADR)                         ;
DATA    EXEC    (IMPP,  D0,     DSTADR)                         ;
;
END;
```

Section 4. Dither Transformation

Dither transformation is the conversion of gray scale data to binary data through a comparison with mask data. For example, referring to figure 4-1, the 4 x 4 pixel gray scale data is compared with the corresponding mask data, and converted to 4 x 4-pixel binary data. The purpose of dither transformation is to have a display that consists of binary data scattered in such a way that it appears like the gradations of a gray scale.

Figure 4-1. Pixel and Mask Data



ALGORITHM

In the program listed at the end of this section, the gray scale data are stored in image memory, and each of the pixels comprising the bit-map data is represented in the lower byte of a word. The SRC area consists of 256 x 256 pixels and the DST area consists of 256 x 256 pixels. Data transfer is done on a word-by-word basis. See figure 4-2.

As the algorithm in figure 4-3 indicates, the program reads pixel data one line at a time, compares them with the mask data that are split for each line, and generates bit data one line at a time.

The generated 1-bit data are packed into 16 bits in a manner similar to the binarization process described in Section 2, and made into DST data.

Figure 4-2. Image Memory

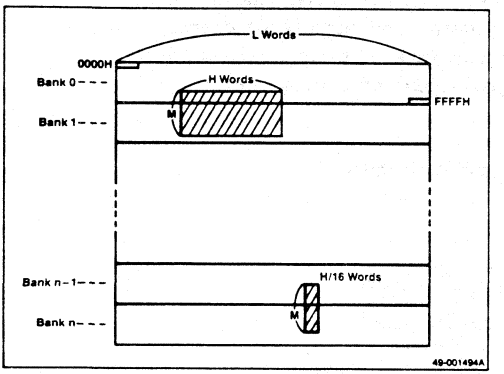
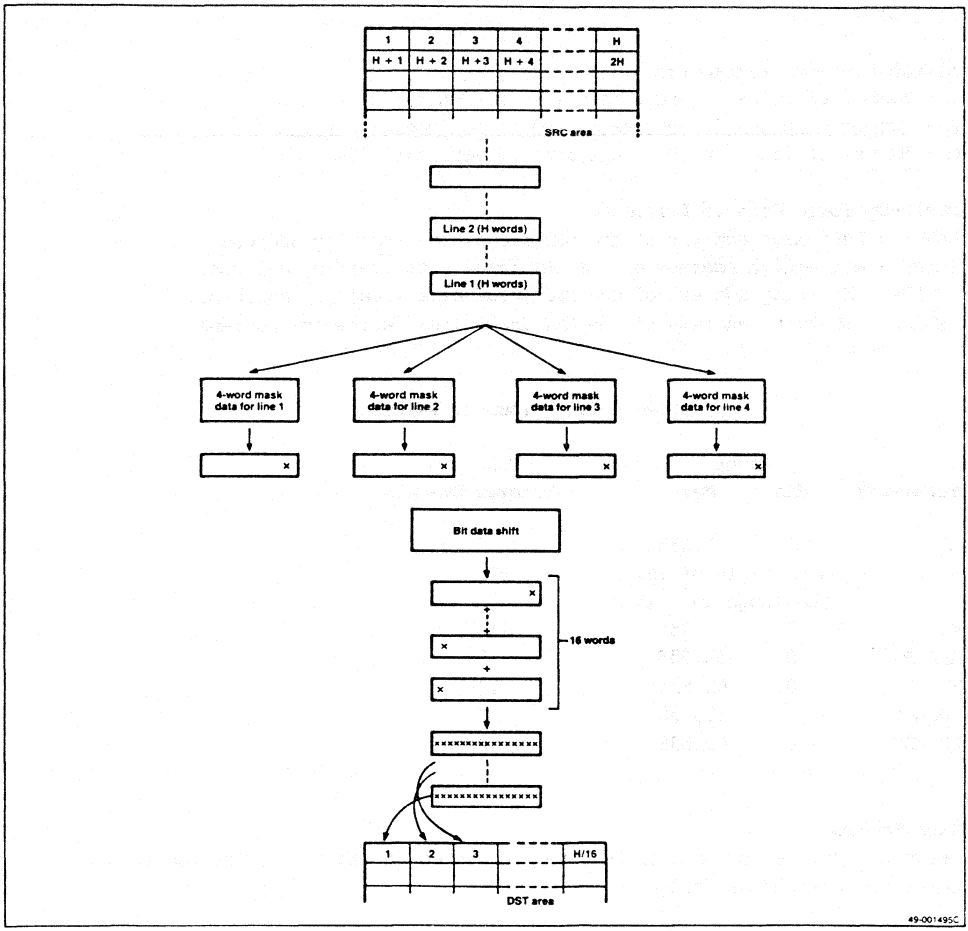


Figure 4-3. Dither Transformation Algorithm



PARAMETERS

Table 4-1 lists allowable values for the parameters, which are defined as follows.

Assembler-Coded Parameters

L - Number of words in image memory in horizontal direction.
H - Number of words in SRC image area in horizontal direction.
M - Number of lines in SRC image area in vertical direction.

Start-Up Token Defined Parameters

RHADR - Read-high address of the SRC image area starting address.
WHADR - Write-high address of the DST image area starting address.
SRCADR - Starting address of the SRC image area (read-low address).
DSTADR - Starting address of the DST image area (write-low address).

Table 4-1. Parameter Values

Parameter	Range Min	Range Max	Value in Program Example
L	0	65,535	1024
H	A multiple of 16, in the range 16 - 256.		256
M	1	256	256
RHADR	0	65,535	0
WHADR	0	65,535	10H
SRCADR	0	65,535	0
DSTADR	0	65,535	0

Mask Tables

Parameter values for mask table data are listed in table 4-2. The table names are defined as follows:

LINE1 - Line 1 mask data
LINE2 - Line 2 mask data
LINE3 - Line 3 mask data
LINE4 - Line 4 mask data

Table 4-2. Parameter Values for Mask Table Data

Table Name		Allowable Range	Value in Program Example
LINE1	m00	-FFFFH to FFFFH	0H
	m01	"	80H
	m02	"	20H
	m03	"	A0H
LINE2	m10	"	C0H
	m11	"	40H
	m12	"	E0H
	m13	"	60H
LINE3	m20	"	30H
	m21	"	B0H
	m22	"	10H
	m23	"	90H
LINE4	m30	"	F0H
	m31	"	70H
	m32	"	D0H
	m33	"	50H

FLOW GRAPH EXPLANATION

As described previously, the program reads pixel data line by line, instead of 4 x 4, and processes each line by switching the four mask tables (four words). Figure 4-4 shows a dither transformation flow graph.

Sixteen read addresses are generated in the horizontal direction at a time. For each 16 read addresses, a write address is generated.

The switching of mask tables is done through the use of the SAVE instruction as follows. First, using the AT statement, addresses are assigned to the input IDs of the nodes used to compare the mask tables with the pixel data. The nodes are FCMPL1 (line 1), FCMPL2 (line 2), FCMPL3 (line 3), and FCMPL4 (line 4). The addresses are saved in the dither matrix (DM) as data tables (assembler-coded). Then, the starting address of each line is generated, the addresses are fetched from the data tables, and are set by node FSAVE (with input ID having FTRC=0). Output ID of node FSAVE for FTRC=1 is thus determined.

Table 4-3 explains the principal nodes.

Table 4-3. Principal Nodes

Node	Function
FRHA	Saves and sets image memory read-high addresses.
FWHA	Saves and sets image memory write-high addresses.
FINCL	Updates read/write-low addresses and checks for overflow conditions.
FINCRH, FRHA	Update, set, and save read-high addresses.
FINCWH, FNHA	Update, set, and save write-low addresses.
FGEN1	Generates the starting address for each of the SRC area lines.
FGEN2	Generates the addresses for the SRC area horizontal lines. Sixteen addresses are generated per line.
FREAD	Reads SRC data.
FRDID, FSAVE	Set the output IDs when the input IDs in node FSAVE are set to FTRC=1.
FSAVE, FCMPL1, FCMPL2, FCMPL3, FCMPL4	Alter the output ID of FSAVE on each line, compare the SRC data that have been read in with the corresponding mask data, and generate 1-bit data.
FSHL, FPACK	Using shift and addition operations, these nodes group the horizontally generated 1-bit data into 16-bit data to generate DST data.
FGEN3	Generates the starting address for each of the DST area lines.
FGEN4	For each 16 lines of SRC data that have been read in, this node generates a DST area horizonal line address.
FWRITE	Writes DST data to the DST area.

Figure 4-4. Dither Transformation Flow Graph (Top)

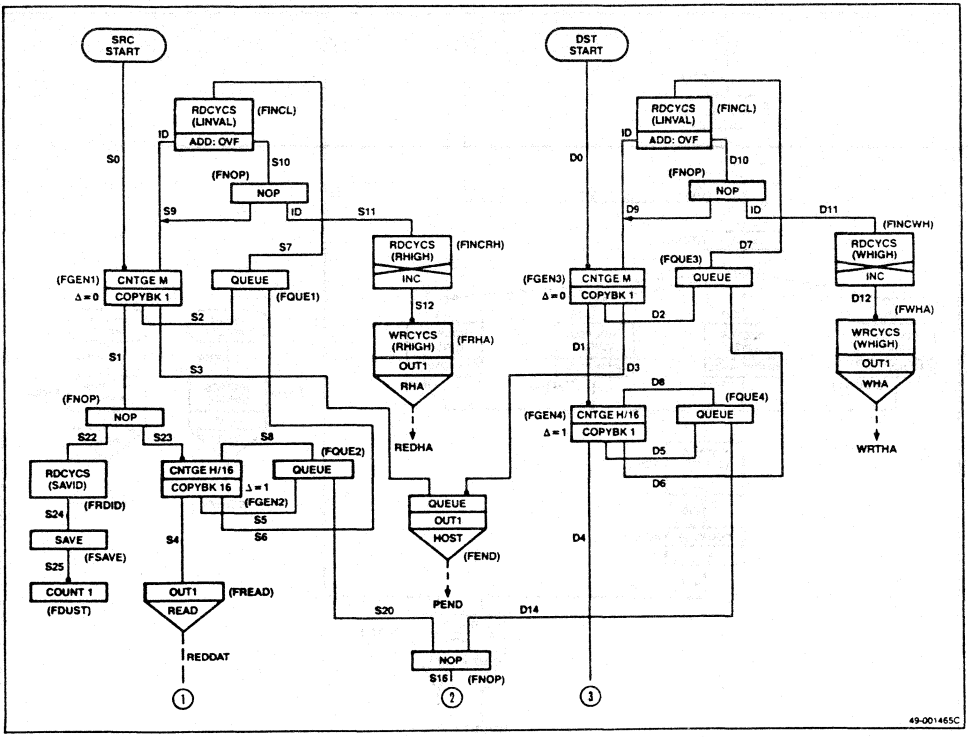
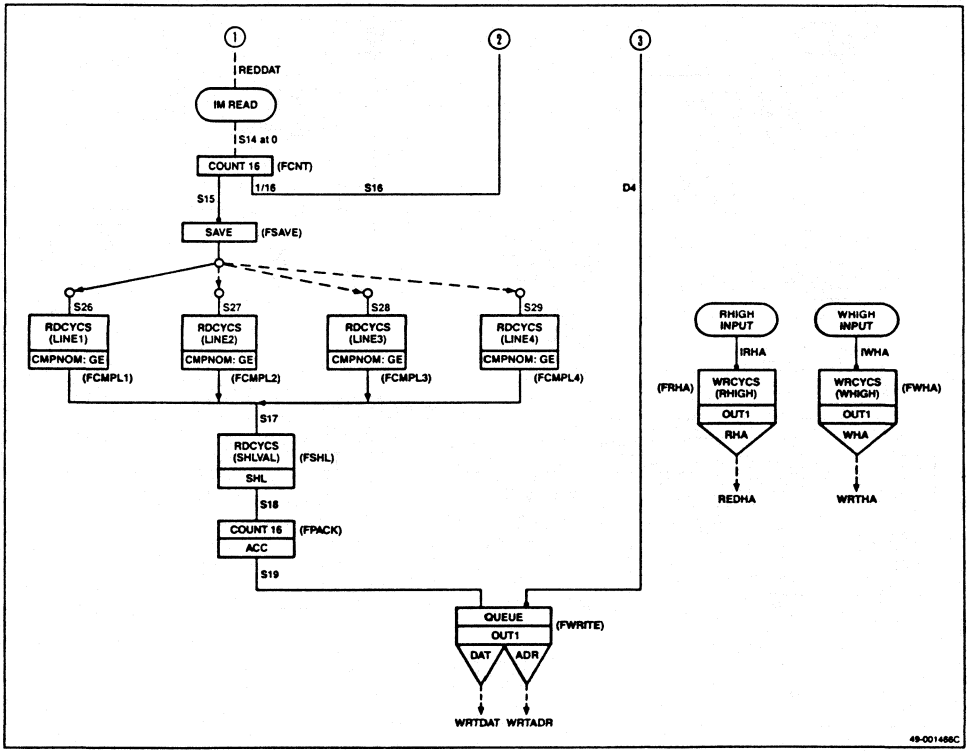


Figure 4-4. Dither Transformation Flow Graph (Bottom)



TYPICAL DITHER MATRICES

Figure 4-5 shows three typical dither matrices. It should be noted that the threshold values in these matrices are intended for their relative rank only. If the SRC image to be processed were a 256-graduated image, for example, the matrix in figure 4-5a could be used with the threshold values shown in figure 4-6.

By setting the mask data values of figure 4-6 in the DM area of a source program, a Bayer type dither image will result. The source program listing (figure 4-7) uses mask data as shown in figure 4-6.

Figure 4-5. Dither Matrices

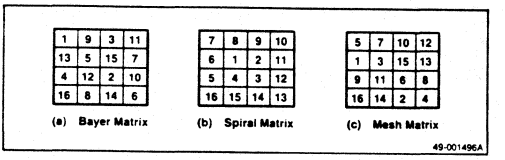


Figure 4-6. Example of Actual Mask Data

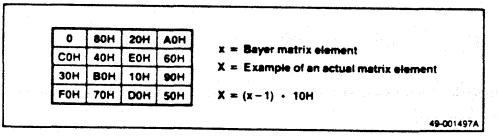


Figure 4-7. Dither Source Program Listing

```
;*****************************************************************
;
;                       DITHER
;
;*****************************************************************
;
MODULE  DITHER  =       8                                       ;
;
EQUATE  L       =       1024                                    ;
EQUATE  H       =       256                                     ;
EQUATE  M       =       256                                     ;
;
EQUATE  RHADR   =       3                                       ;
EQUATE  WHADR   =       3                                       ;
EQUATE  SRCADR  =       0                                       ;
EQUATE  DSTADR  =       256                                     ;
;
EQUATE  FINE    =       0                                       ;
EQUATE  READ    =       4                                       ;
EQUATE  WRITE   =       5                                       ;
;
;*****************************************************************
;                       INPUT
;*****************************************************************
;
INPUT   IRHA,   IWHA,   ITR,    S0,     D0,     S14 AT 0        ;
;
;*****************************************************************
;                       OUTPUT
;*****************************************************************
;
OUTPUT  REDHA,  WRTHA,  PEND,   WRTDAT, WRTADR, REDDAT          ;
;
;*****************************************************************
;                       DEFINE
;*****************************************************************
;
DEFINE  S26     AT      1                                       ;
DEFINE  S27     AT      2                                       ;
DEFINE  S28     AT      3                                       ;
DEFINE  S29     AT      4                                       ;
;
```

Figure 4-7. Dither Source Program Listing (cont)

```
;*****************************************************************
;                        LINK
;*****************************************************************
;
LINK    S1,     S2,     S3      = FGEN1   (S9,  S0 )          ;
LINK    S22,    S23             = FNOP    (S1     )           ;
LINK    S24                     = FRDID   (S22    )           ;
LINK    S25                     = FSAVE   (S24    )           ;
LINK                            = FDUST   (   , S25)          ;
LINK    S4,     S5,     S6      = FGEN2   (S8,  S23)          ;
LINK    S7                      = FQUE1   (S2,  S6 )          ;
LINK    PEND                    = FEND    (S3,  D3 )          ;
LINK    REDDAT                  = FREAD   (S4     )           ;
LINK    S8                      = FQUE2   (S5,  S20)          ;
LINK    S9,     S10             = FINCL   (S7     )           ;
LINK    S11,    S9              = FNOP    (S10    )           ;
LINK    S12                     = FINCRH  (S11    )           ;
LINK    REDHA                   = FRHA    (   , S12)          ;
LINK    S15,    S16             = FCNT    (S14    )           ;
LINK                            = FSAVE   (   , S15)          ;
LINK    S17                     = FCMPL1  (S26    )           ;
LINK    S17                     = FCMPL2  (S27    )           ;
LINK    S17                     = FCMPL3  (S28    )           ;
LINK    S17                     = FCMPL4  (S29    )           ;
LINK    S18                     = FSHL    (S17    )           ;
LINK    S19                     = FPACK   (S18    )           ;
LINK    S20,    D14             = FNOP    (S16    )           ;
LINK    D1,     D2,     D3      = FGEN3   (D9,  D0 )          ;
LINK    D4,     D5,     D6      = FGEN4   (D8,  D1 )          ;
LINK    D7                      = FQUE3   (D2,  D6 )          ;
LINK    D8                      = FQUE4   (D5,  D14)          ;
LINK    D9,     D10             = FINCL   (D7     )           ;
LINK    D11,    D9              = FNOP    (D10    )           ;
LINK    D12                     = FINCWH  (D11    )           ;
LINK    WRTHA                   = FWHA    (   , D12)          ;
LINK    WRTDAT, WRTADR          = FWRITE  (S19, D4 )          ;
LINK    REDHA                   = FRHA    (   , IRHA)         ;
LINK    WRTHA                   = FWHA    (   , IWHA)         ;
;
```

Figure 4-7. Dither Source Program Listing (cont)

```
;*****************************************************************
;                        FUNCTION
;*****************************************************************
;
FUNCTION        FEND    = OUT1   (FINE,   0), QUEUE  (QUEE, 1)  ;
FUNCTION        FRHA    = OUT1   (READ, 70H), WRCYCS (RHIGH,1)  ;
FUNCTION        FREAD   = OUT1   (READ,   0)                    ;
FUNCTION        FWHA    = OUT1   (WRITE,10H), WRCYCS (WHIGH,1)  ;
FUNCTION        FWRITE  = OUT2 (WRITE,20H,0), QUEUE  (QUEW,16)  ;
FUNCTION        FGEN1   = COPYBK (1,      0), CNTGE  (M      )  ;
FUNCTION        FGEN2   = COPYBK (16,     1), CNTGE  (H/16   )  ;
FUNCTION        FGEN3   = COPYBK (1,      0), CNTGE  (M      )  ;
FUNCTION        FGEN4   = COPYBK (1,      1), CNTGE  (H/16   )  ;
FUNCTION        FQUE1   = QUEUE  (QUE1,   1)                    ;
FUNCTION        FQUE2   = QUEUE  (QUE2,   1)                    ;
FUNCTION        FQUE3   = QUEUE  (QUE3,   1)                    ;
FUNCTION        FQUE4   = QUEUE  (QUE4,   1)                    ;
FUNCTION        FNOP    = NOP    (XX      )                     ;
FUNCTION        FINCL   = ADD  (X, OVF, BRC), RDCYCS (LINVAL,1) ;
FUNCTION        FINCRH  = INC    (XCH     ), RDCYCS (RHIGH, 1) ;
FUNCTION        FINCWH  = INC    (XCH     ), RDCYCS (WHIGH, 1) ;
FUNCTION        FCNT    = COUNT  (16      )                     ;
FUNCTION        FSHL    = SHL    (X       ), RDCYCS (SHLVAL,16);
FUNCTION        FPACK   = ACC    (X       ), COUNT  (16     )  ;
FUNCTION        FRDID   = RDCYCS (SAVID,  4)                    ;
FUNCTION        FSAVE   = SAVE                                  ;
FUNCTION        FDUST   = COUNT  (1       )                     ;
FUNCTION        FCMPL1  = CMPNOM (X,   GE ), RDCYCS (LINE1, 4)  ;
FUNCTION        FCMPL2  = CMPNOM (X,   GE ), RDCYCS (LINE2, 4)  ;
FUNCTION        FCMPL3  = CMPNOM (X,   GE ), RDCYCS (LINE3, 4)  ;
FUNCTION        FCMPL4  = CMPNOM (X,   GE ), RDCYCS (LINE4, 4)  ;
;
```

Figure 4-7. Dither Source Program Listing (cont)

```
;*****************************************************************
;                          MEMORY
;*****************************************************************
;
MEMORY  QUE1    = AREA ( 1)                                        ;
MEMORY  QUE2    = AREA ( 1)                                        ;
MEMORY  QUE3    = AREA ( 1)                                        ;
MEMORY  QUE4    = AREA ( 1)                                        ;
MEMORY  QUEE    = AREA ( 1)                                        ;
MEMORY  QUEW    = AREA (16)                                        ;
MEMORY  RHIGH   = AREA ( 1)                                        ;
MEMORY  WHIGH   = AREA ( 1)                                        ;
MEMORY  LINVAL  = L                                                ;
MEMORY  SHLVAL  = 15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0            ;
MEMORY  SAVID   = 1,2,3,4                                          ;
MEMORY  LINE1   = 000H, 080H,   020H,   0A0H                       ;
MEMORY  LINE2   = 0C0H, 040H,   0E0H,   060H                       ;
MEMORY  LINE3   = 030H, 0B0H,   010H,   090H                       ;
MEMORY  LINE4   = 0F0H, 070H,   0D0H,   050H                       ;
;
;*****************************************************************
;                          START
;*****************************************************************
;
START;
DATA    EXEC    (DITHER,        IRHA,   RHADR)                     ;
DATA    EXEC    (DITHER,        IWHA,   WHADR)                     ;
DATA    EXEC    (DITHER,        D0,     DSTADR)                    ;
DATA    EXEC    (DITHER,        S0,     SRCADR)                    ;
;
END;
```

Section 5. Mask Processing

Given a pixel, mask processing refers to the technique of obtaining a mapping on the basis of nine pixels, the pixel of interest and the adjoining eight pixels. This technique permits processing tasks such as smoothing, edge emphasis, and edge detection through alterations of the 3 x 3 mask elements.

ALGORITHM

Mapping X_0' is obtained from the point of interest X_0 and the adjoining eight pixels, shown in figure 5-1a, and via the computation with 3 x 3 mask data. X_0' is obtained from the following formulae:

$$X_0' = X_0 * M_0 + X_1 * M_1 + \ldots + X_8 * M_8$$

$$= \sum_{i=0}^{8} X_i * M_i$$

Where:
X_i is 8-bit image data containing one pixel.
M_i is 9-bit mask data including a sign bit.
X_0' is 17-bit data including a sign bit.

The DST data consist of eight contiguous bits out of the 16 bits in X_0', exclusive of the sign bit.

Figure 5-2 shows the algorithm involved. First, three lines of contiguous image data in the SRC area are read three pixels at a time. Three copies are made of the pixel data that have been read in, and each of these copies is multiplied with the 9-word mask data. The multiplication is carried out as follows: the pixel data in the first line are multiplied with the 3-word mask data in the upper row; the pixel data in the second line with the three words in the middle row; and the pixel data in the third line with the three words in the bottom row. Then, the products X, Y, and Z of the results of multiplications with the rows in the mask data are obtained. By extracting X_n, Y_{n+1}, and Z_{n+2} from the multiplication results, the sum (X_0') of the products between the 3 x 3 pixels and mask data can be obtained.

Figure 5-1. Pixel Information and a 3 x 3 Mask

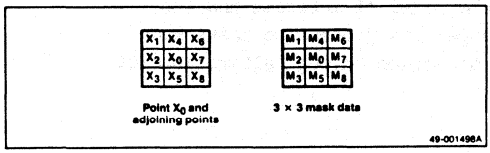
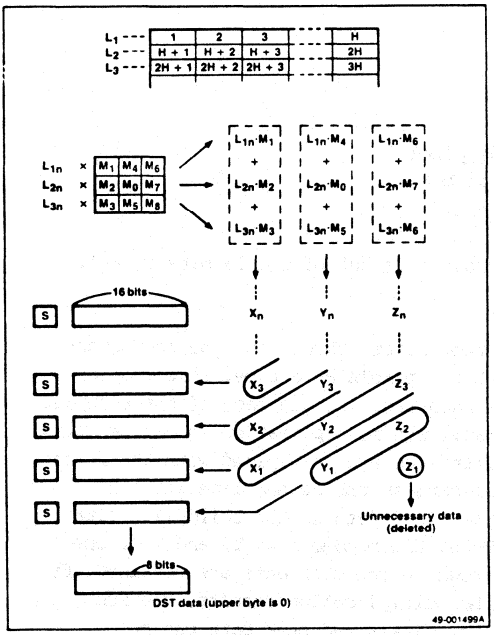
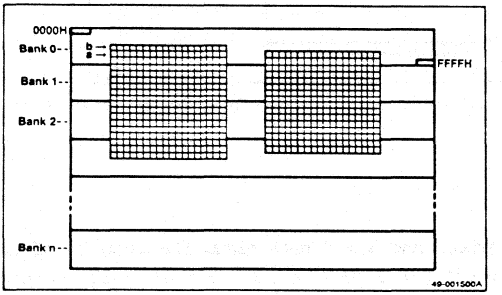


Figure 5-2. Algorithm



To process a pixel, an area three pixels across and three pixels down is required, as shown in figure 5-1. An SRC area address generation involves the generation of the addresses for three lines: the line containing the point of interest X_0, one line above and one line below, in H-words across. This means the SRC must contain an area containing the source image area, a line above it, and a line below it. Further, the SRC start-up token issued by the host system sets (b), rather than (a), as shown in figure 5-3.

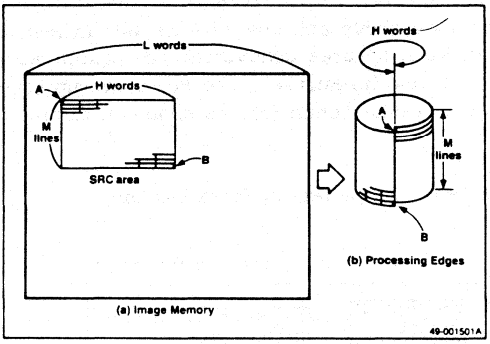
Figure 5-3. Image Memory Organization



Address generation for the DST side is done by generating a horizontal H-word for every three pixel addresses generated on the SRC side.

Since the edges of an image are considered unimportant, processing on the right and left edges of the image is ignored in this program. Image data are treated as being continuous, as shown in figure 5-4. Consequently, no normal data can be obtained for the two pixels X_0' on the right and left borders.

Figure 5-4. Concept of Image Data



Computation Precision

Multiplication between pixel and 3 x 3 mask data. The results of the multiplication between 8-bit pixel data and 3 x 3 mask data, comprised of a sign bit and eight data bits, are expressed in 17 bits. The 17 bits include a sign bit.

Addition of multiplication of nine pixels. 17-bit multiplications are performed through the addition of 17 bits, including a sign bit. The results are expressed in 17 bits still including the sign bit. In the event of an overflow or underflow, the computation is carried out without adjustments in the values.

DST data. Through parameter specifications, DST data are obtained by extracting contiguous groups of eight bits from the 16 data bits, exclusive of the sign bit, of the result of a multiplication (with the higher byte set to 0).

PARAMETERS

Table 5-1 lists allowable values for the parameters, which are defined as follows.

Assembler-Coded Parameters

L - Number of horizontal words in the image memory.
H - Number of horizontal words in the SRC image area.
M - Number of vertical lines in the SRC image area.
SHRX - Number of times X_0' data are shifted.
MSKL1 - Top row mask data.
MSKL2 - Middle row mask data.
MSKL3 - Bottom row mask data.

Start-Up Token Defined Parameters

RHADR - High address of the SRC image starting address.
WHADR - High address of the DST image area starting address.
SRCADR - Low address on a line of the SRC image area starting address.
DSTADR - Low address of the DST image area starting address.

Table 5-1. Parameter Values

Parameter	Range Min	Range Max	Value in Program Example
L	0	65,535	1024
H	A multiple of 4, in the range 4 - 1024.		255
M	1	256	255
SHRX	0	15	4
MSKL1 M_6	-FFH	FFH	1
M_4	"	"	2
M_1	"	"	1
MSKL2 M_7	"	"	2
M_0	"	"	4
M_2	"	"	2
MSKL3 M_8	"	"	1
M_5	"	"	2
M_3	"	"	1
RHADR	0	65,535	0
WHADR	0	65,535	0
SRCADR	0	65,535	0
DSTADR	0	65,535	256

FLOW GRAPH EXPLANATION

In 3 x 3 mask processing, three lines of SRC data are required for each line of DST data generated. If three lines of pixel data are to be read from the SRC area, they may not be in the same bank. Therefore, three register files of the uPD9305 are used for each IM-read of three lines, and a read-high address is set for each of these lines in the register files.

Four read and four write addresses are generated horizontally in different process flows. The banks are switched for each line of read/write by adding the values of L. The L values indicate the amount of the IM in the horizontal direction when the starting address is generated for each line. If an overflow is detected on a line, the high address for that line is updated.

As shown in figure 5-5, mask data are divided into top, middle, and bottom rows, each containing three words, and are used to generate the data table sequentially from the right. The purpose of this step is to determine the sums Z, Y, and X, which are obtained from the row-by-row multiplications of the mask data that have been split and the vertical three pixels.

The sums X and Y of the multiplications thus obtained are stored in a 5-word buffers. The Z sum, Z_1, which is obtained first, is deleted, and Z_2 and all subsequent values are added to the data stored in the X and Y buffers to determine the value of X_0'.

Table 5-2 defines the principal nodes.

Figure 5-5. Mask Data

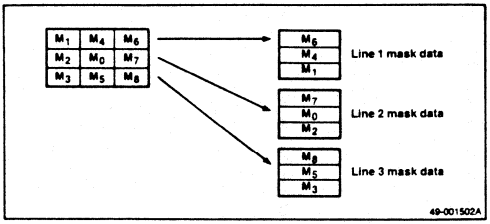


Figure 5-6. Addition of Sums of Row-by-Row Multiplications

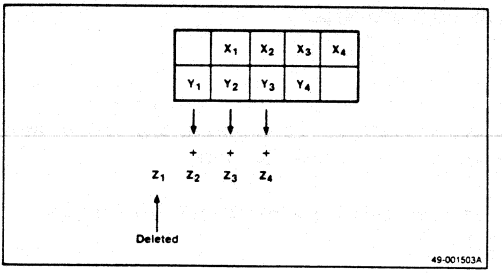


Table 5-2. Principal Nodes

Node	Function
FRHAL1	Saves the high address of the SRC area starting address (line 1), and sets the read high address of register file 0, which is used for the line 1 IM-read.
FWHA	Saves the high address of the DST area starting address, and sets a write high address.
FR1LAD, FIRHAC, FRHAL2	Calculate the line 2 SRC area high address, save it in the DM, and set the read high address for register file 1, which is used for the line 2 IM-read.
FR2LAD, FIRHAC2, FRHAL3	Calculate the line 3 SRC area high address, save it in the DM, and set the read high address for Register File 2 which is used in the line 3 IM-read.
FGEN1	Each time a line is processed, this node generates the starting address for vertical M lines, with the origin at the SRC area starting address.
FGEN2	Based on the starting address for each line generated in FGEN1, this node generates 4 horizontal addresses per line.
FREDL1	Reads the SRC data for line 1.
FINCL	Generates and updates starting addresses, and checks for bank switching (overflow).

Table 5-2. Principal Nodes (cont)

Node	Function
FIRHL1, FRHAL1	Update line 1 read high addresses.
FGEN3	Each time a line is processed, this node generates the starting address for vertical M lines, with the origin at the starting address of line 2 in the SRC area.
FGEN4	Creates 4 copies of the starting address for each line generated in FGEN3.
FREDL2	Reads the SRC data for line 2.
FIRHL2, FRHAL2	Update line 2 read high addresses.
FGEN5	Each time a line is processed, this node generates the starting address for vertical M lines, with the origin at the starting address of line 3 in the SRC area.
FGEN6	Based on the starting address for each line generated in FGEN5, this node generates 4 horizontal addresses for a line.
FREDL3	This node reads the SRC data for line 3.
FIRHL3, FRHAL3	Update line 3 read-high addresses.
FGEN7	Each time a line is processed, this node generates the starting address for a line, with the origin at the starting address of the DST area.
FGEN8	Based on the starting address for a line generated in FGEN7, this node generates 4 horizontal addresses for the line.
FIWHA, FWHA	Update write high addresses.
FCPYB2, FMUL1	Make 3 copies of the line 1 input data, and multiply the data with the top 3 data points of the mask data.

Table 5-2. Principal Nodes (cont)

Node	Function
FCPYB2, FMUL2	Make 3 copies of the line 2 input data, and multiply the data with the middle 3 data points of the mask data.
FCPYB2, FMUL3	Make 3 copies of the line 3 input data, and multiply the data with the bottom 3 data points of the mask data.
FADD1, FADD2	Hold the sum of the results of multiplications between the vertical 3 pixels and the right 3 words of mask data.
FADD3, FADD4, FWRRW2	Hold the sum of the results of multiplications between the vertical 3 pixels and the middle 3 words of mask data, saving the sums in the DM.
FADD5, FADD6, FWRRW1	Hold the sum of the results of multiplications between the vertical 3 pixels and the left 3 words of mask data, saving the sums in the DM.
FCUT	Deletes the first extraneous DST data.
FRDRW1, FRDRW2	Read the sums X_n and Y_{n+1} of row-by-row multiplications, and add them to Z_{n+2} to obtain X_0'.
FRDSHR, FRDAND	Shift X_0' to the right by a specified number of bits, mask the upper eight bits, and generate DST data.
FWRITE	Writes DST data to the DST area.
FCNT4, FCPYM4	Count the number of sums Z of the multiplications between vertical three pixels and the middle three words of mask data, and restart the generation of read/write addresses on each generation of four words.
FRDRW3, FRDRW1, FRDRW2, FRDSHR, FRDAND	Generate DST data corresponding to the previously deleted word.
FEND	Notifies the host system that all processing has ended.

Figure 5-7. 3 x 3 Mask Flow Graph (right)

(a) Setting Read/Write Addresses and Generating Read Addresses

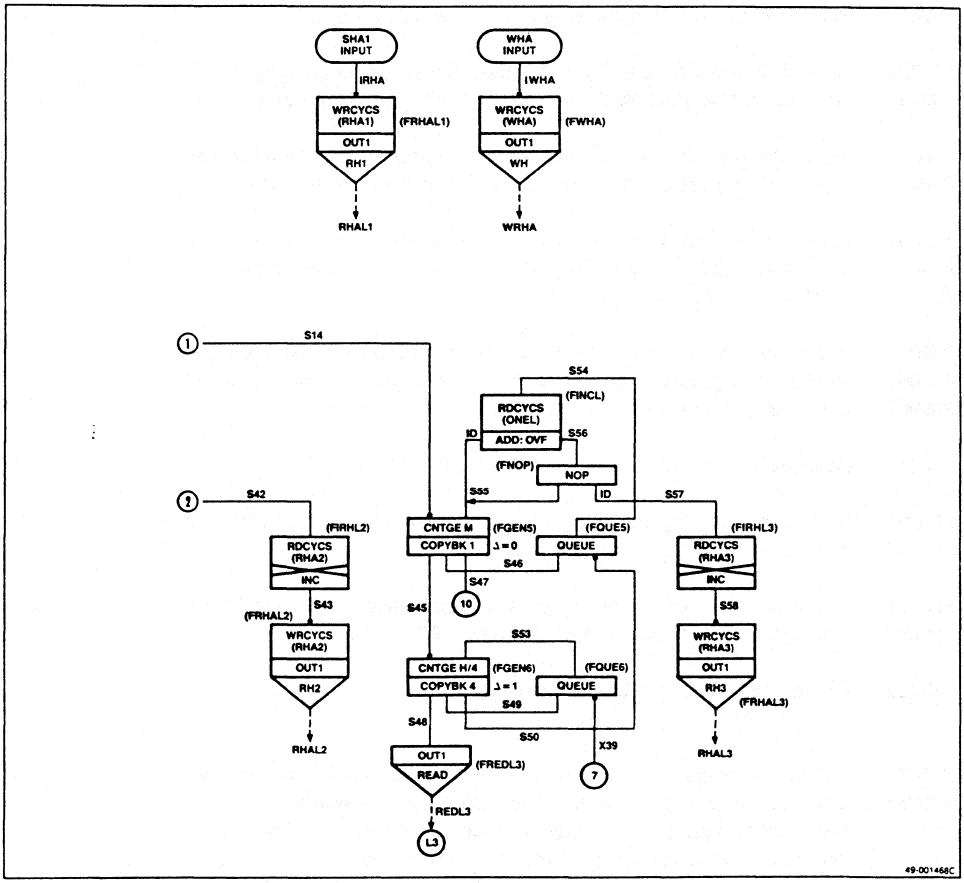


Figure 5-7. 3 x 3 Mask Flow Graph (left)

(a) Setting Read/Write Addresses and Generating Read Addresses

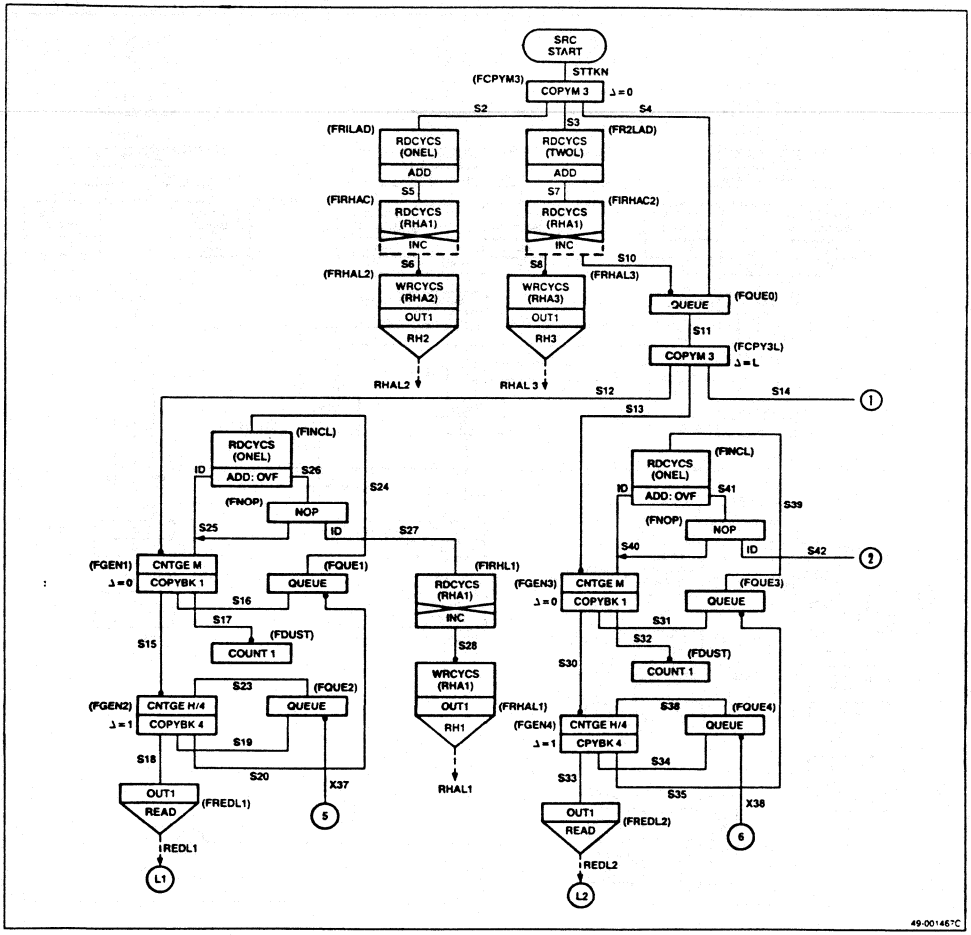


Figure 5-7. 3 x 3 Mask Flow Graph

(b) Computation Part

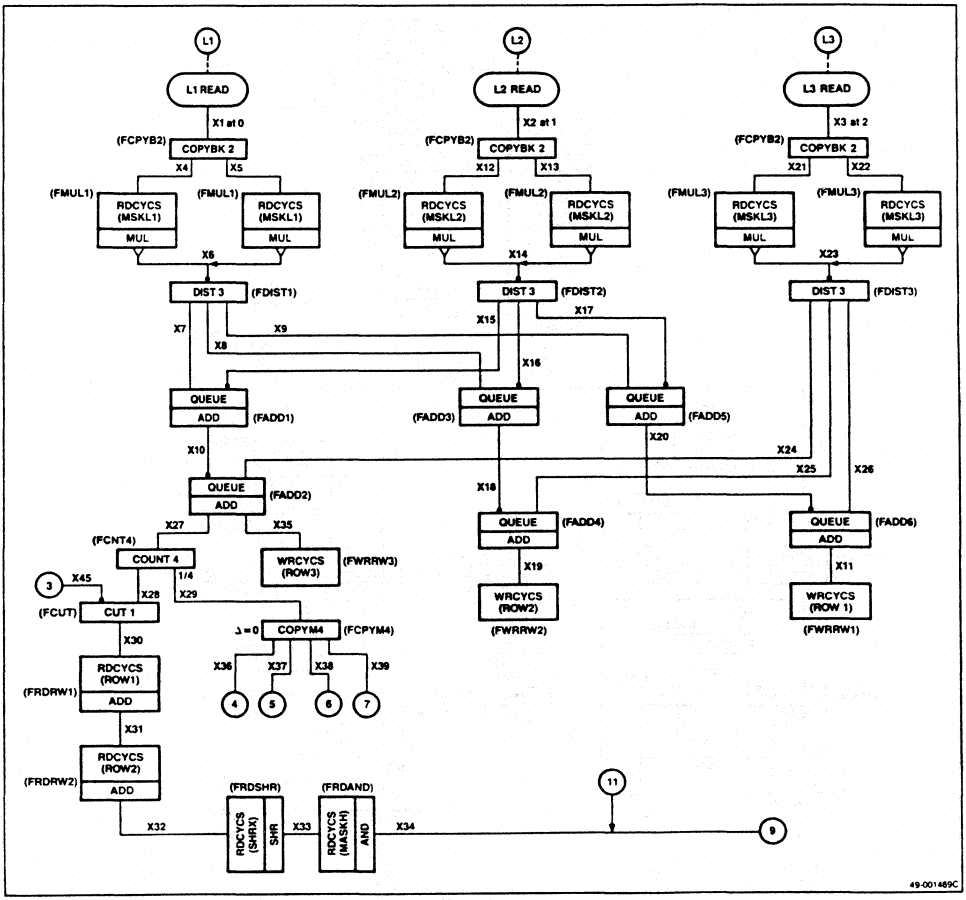
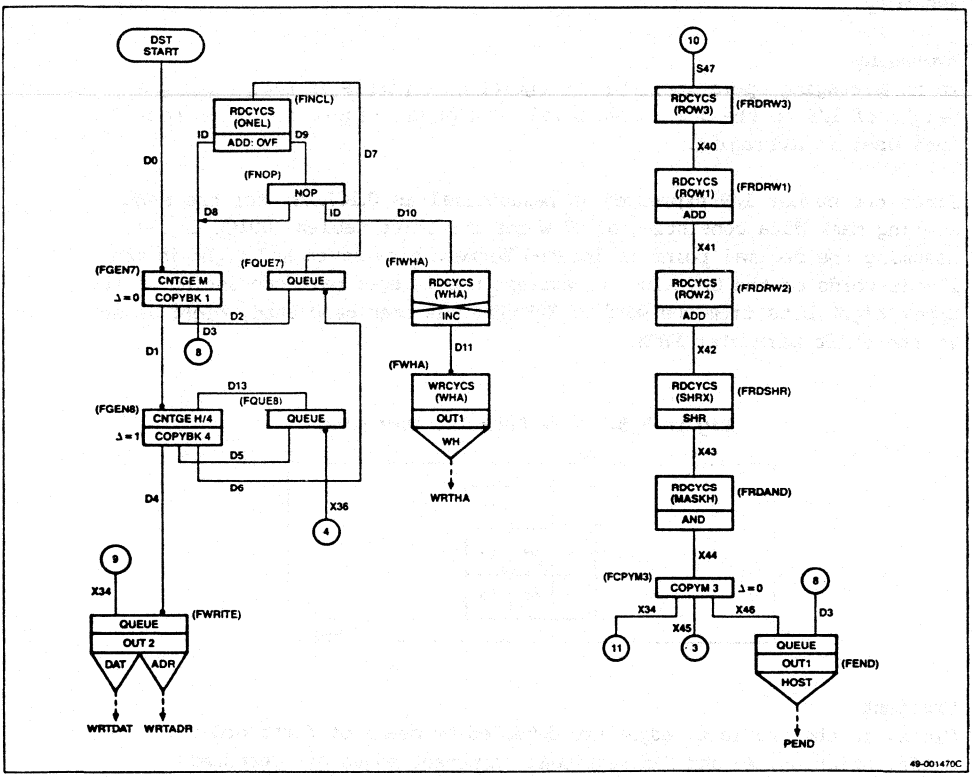


Figure 5-7. 3 x 3 Mask Flow Graph

(c) Write Address Generation

(d) Processing of the Last Point



USING A MASK TABLE

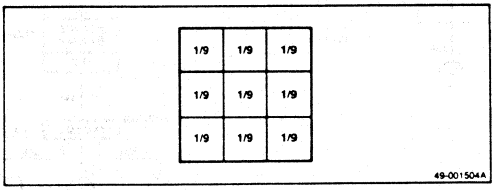
This section provides several examples of operation of frequently used 3 x 3 masks. Before using these masks, the mask data to be used should be coded in the MSKTBL area of the assembler source listing, followed by a program assembly.

Averaging

In an averaging operation, the average of nine pixels is taken, using a weight of 1/9 on the 3 x 3 area pixel data group. Figure 5-8 shows mask data used in averaging.

Since the number 1/9 expressed in hexadecimal is 0.1C7...H, for the MASK storing mask data consisting of 27 words for three tables, 001CH is set, assuming the decimal point is located between bits seven and eight in the 17-bit words of the DM. Also, to extract the integer part consisting of the upper eight bits from one word of X_0' that has been generated, eight is set in the shift parameter SHRX.

Figure 5-8. Mask Data for Averaging



Gradient

Since, in the gradient, edges are detected by means of first-order differentiation, X- and Y-directional component edges are detected separately. Figures 5-9 and 5-10 show mask data in the X and Y directions, respectively. First-order differentiation has better edge detection capability than the second-order differentiation, which is explained below.

Figure 5-9. Mask Data in the X-Direction (Horizontal)

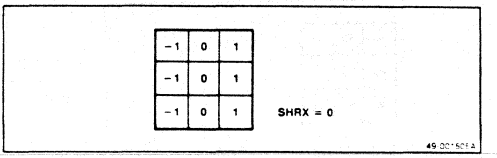
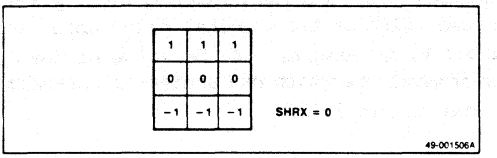


Figure 5-10. Mask Data in the Y-Direction (Vertical)



Laplacian

The Laplacian is used to perform edge detection on the basis of second-order differentiation. Figure 5-11 is an example.

For figure 5-12, assume x to be the weight for pixel data a_0, and y to be the weight for the pixel data for the adjoining eight points, with the ratio, R, of the weights being R = x/8y. Then in Laplacian mask data, the closer the value of R to 1, the more pronounced the high-region components. When using these mask data, SHRX of the shift parameter is set to 0.

Figure 5-11. Laplacian Mask Data

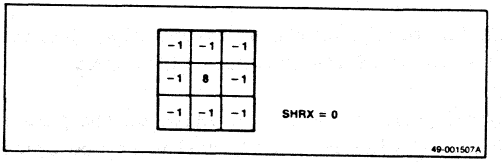
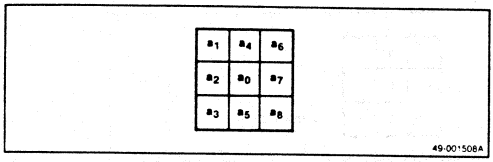


Figure 5-12. Pixel Data



Noise Removal

In noise removal, if the central point and the adjoining eight pixel data are equal, X_0' assumes the same value as the original pixel data, and if the adjoining eight points are 0, X_0' assumes 1/4 the value of the original pixel data. Thus, the noise removal operation can achieve a one-half reduction in image noise. (See figure 5-13.)

Figure 5-13. Mask Data for Noise Removal

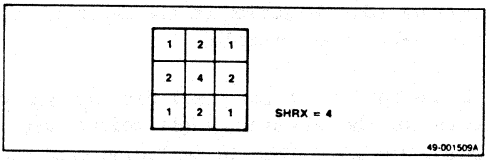


Image Emphasis

In image emphasis:

- If the central point pixel is 1/2 the value of the pixels at the adjoining eight points, X_0' will be 0000H.
- If the central point and the pixels at the adjoining eight points are equal, X_0' will be 1/4 the value of the original pixel data.
- If the pixel at the central point is twice the value of the pixels at the adjoining eight points, X_0' will be 3/4 the value of the original pixel data.

In this way, the operation enhances the pixel data intensity and produces emphasized images. (See figure 5-14.)

Figure 5-14. Mask Data for Image Emphasis

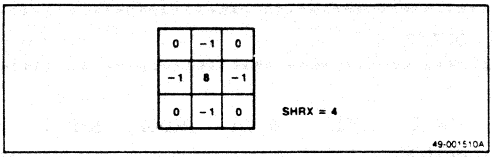


Figure 5-15. 3 x 3 Mask Source Program

```
;*****************************************************************
;
;                       3 X 3  MASK
;
;*****************************************************************
;
MODULE  MASK    =       8                                       ;
;
EQUATE  L       =       1024                                    ;
EQUATE  H       =       256                                     ;
EQUATE  M       =       256                                     ;
;
EQUATE  RHADRS  =       3                                       ;
EQUATE  WHADRS  =       3                                       ;
EQUATE  RLADRS  =       0                                       ;
EQUATE  WLADRS  =       256                                     ;
;
EQUATE  FINE    =       0                                       ;
EQUATE  READ    =       4                                       ;
EQUATE  WRITE   =       5                                       ;
;
EQUATE  BIT     =       4                                       ;
;
;*****************************************************************
;                       INPUT
;*****************************************************************
;
INPUT   IR,     IW,     D0,     STTKN                           ;
INPUT   X1 AT 0,        X2 AT 1,        X3 AT 2                 ;
;
```

Figure 5-15. 3 x 3 Mask Source Program (cont)

```
;*****************************************************************
;                        OUTPUT
;*****************************************************************
;
OUTPUT  WRTHI,  RHAL1,  RHAL2,  RHAL3,  REDL1,  REDL2,  REDL3   ;
OUTPUT  PEND,   WRTDAT, WRTLOW                                  ;
;
;*****************************************************************
;                        LINK
;*****************************************************************
;
LINK    RHAL1                   = FRHAL1  (    , IR )           ;
LINK    WRTHI                   = FWHA    (    , IW )           ;
;
LINK    S2,     S3,     S4      = FCPYM3  (STTKN    )           ;
LINK    S5                      = FR1LAD  (S2       )           ;
LINK    S6                      = FIRHAC  (S5       )           ;
LINK    RHAL2                   = FRHAL2  (    , S6 )           ;
LINK    S7                      = FR2LAD  (S3       )           ;
LINK    S8,     S10             = FIRHAC2 (S7       )           ;
LINK    RHAL3                   = FRHAL3  (    , S8 )           ;
LINK    S11                     = FQUE0   (S4,  S10)            ;
LINK    S12,    S13,    S14     = FCPY3L  (S11      )           ;
;
LINK    S15,    S16,    S17     = FGEN1   (S25, S12)            ;
LINK    S18,    S19,    S20     = FGEN2   (S23, S15)            ;
LINK    REDL1                   = FREDL1  (S18      )           ;
LINK    S23                     = FQUE2   (S19, X37)            ;
LINK    S24                     = FQUE1   (S16, S20)            ;
LINK    S25,    S26             = FINCL   (S24      )           ;
LINK    S27,    S25             = FNOP    (S26      )           ;
LINK    S28                     = FIRHL1  (S27      )           ;
LINK    RHAL1                   = FRHAL1  (    , S28)           ;
LINK                            = FDUST   (    , S17)           ;
;
LINK    S30,    S31,    S32     = FGEN3   (S40, S13)            ;
LINK    S33,    S34,    S35     = FGEN4   (S38, S30)            ;
LINK    REDL2                   = FREDL2  (S33      )           ;
LINK    S38                     = FQUE4   (S34, X38)            ;
LINK    S39                     = FQUE3   (S31, S35)            ;
LINK    S40,    S41             = FINCL   (S39      )           ;
LINK    S42,    S40             = FNOP    (S41      )           ;
```

Figure 5-15. 3 x 3 Mask Source Program (cont)

```
LINK    S43                     = FIRHL2  (S42      )          ;
LINK    RHAL2                   = FRHAL2  (   , S43)           ;
LINK                            = FDUST   (   , S32)           ;
;
LINK    S45,    S46,    S47     = FGEN5   (S55, S14)           ;
LINK    S48,    S49,    S50     = FGEN6   (S53, S45)           ;
LINK    REDL3                   = FREDL3  (S48      )          ;
LINK    S53                     = FQUE6   (S49, X39)           ;
LINK    S54                     = FQUE5   (S46, S50)           ;
LINK    S55,    S56             = FINCL   (S54      )          ;
LINK    S57,    S55             = FNOP    (S56      )          ;
LINK    S58                     = FIRHL3  (S57      )          ;
LINK    RHAL3                   = FRHAL3  (   , S58)           ;
;
LINK    X4,     X5              = FCPYB2  (X1       )          ;
LINK    X6                      = FMUL1   (X4       )          ;
LINK    X6                      = FMUL1   (X5       )          ;
LINK    X7,     X8,     X9      = FDIST2  (   , X6  )          ;
LINK    X10                     = FADD1   (X7,  X15)           ;
LINK    X27,    X35             = FADD2   (X24, X10)           ;
LINK    X28,    X29             = FCNT4   (X27      )          ;
LINK    X30                     = FCUT    (X28, X45)           ;
LINK    X31                     = FRDRW1  (X30      )          ;
LINK    X32                     = FRDRW2  (X31      )          ;
LINK    X33                     = FRDSHR  (X32      )          ;
LINK    X34                     = FRDAND  (X33      )          ;
LINK                            = FWRRW3  (X35      )          ;
LINK    X36,  X37,  X38,  X39   = FCPYM4  (X29      )          ;
LINK    X12,    X13             = FCPYB2  (X2       )          ;
LINK    X14                     = FMUL2   (X12      )          ;
LINK    X14                     = FMUL2   (X13      )          ;
LINK    X15,    X16,    X17     = FDIST3  (   , X14)           ;
LINK    X18                     = FADD3   (X8,  X16)           ;
LINK    X19                     = FADD4   (X26, X20)           ;
LINK                            = FWRRW2  (X19      )          ;
LINK    X20                     = FADD5   (X9,  X17)           ;
LINK    X21,    X22             = FCPYB2  (X3       )          ;
LINK    X23                     = FMUL3   (X21      )          ;
LINK    X23                     = FMUL3   (X22      )          ;
LINK    X24,    X25,    X26     = FDIST4  (   , X23)           ;
LINK    X11                     = FADD6   (X25, X18)           ;
LINK                            = FWRRW1  (X11      )          ;
```

Figure 5-15. 3 x 3 Mask Source Program (cont)

```
LINK    X40                     = FRDRW3  (S47     )          ;
LINK    X41                     = FRDRW1  (X40     )          ;
LINK    X42                     = FRDRW2  (X41     )          ;
LINK    X43                     = FRDSHR  (X42     )          ;
LINK    X44                     = FRDAND  (X43     )          ;
LINK    X34,    X45,    X46     = FCPYM3  (X44     )          ;
;
LINK    D1,     D2,     D3      = FGEN7   (D8,  D0 )          ;
LINK    D4,     D5,     D6      = FGEN8   (D13, D1 )          ;
LINK    D7                      = FQUE7   (D2,  D6 )          ;
LINK    D8,     D9              = FINCL   (D7      )          ;
LINK    D10,    D8              = FNOP    (D9      )          ;
LINK    D11                     = FIWHA   (D10     )          ;
LINK    WRITHI                  = FWHA    (    , D11)         ;
LINK    D13                     = FQUE8   (D5,  X36)          ;
LINK    PEND                    = FEND    (D3,  X46)          ;
LINK    WRIDAT, WRILOW          = FWRITE  (X34, D4 )          ;
;
```

Figure 5-15. 3 x 3 Mask Source Program (cont)

```
;*****************************************************************
;                        FUNCTION
;*****************************************************************
;
FUNCTION        FADD1   = ADD,           ,  QUEUE  (QUEA1, 4)   ;
FUNCTION        FADD2   = ADD   (XX      ),  QUEUE  (QUEA2, 4)   ;
FUNCTION        FADD3   = ADD,              QUEUE  (QUEA3, 4)   ;
FUNCTION        FADD4   = ADD,              QUEUE  (QUEA4, 4)   ;
FUNCTION        FADD5   = ADD,              QUEUE  (QUEA5, 4)   ;
FUNCTION        FADD6   = ADD,              QUEUE  (QUEA6, 4)   ;
FUNCTION        FCNT4   = COUNT  (4      )                      ;
FUNCTION        FCPYB2  = COPYBK2 (2,   0)                      ;
FUNCTION        FCPY3L  = COPYM  (3,    L)                      ;
FUNCTION        FCPYB2  = COPYBK (2,    0)                      ;
FUNCTION        FCPYM3  = COPYM  (3,    0)                      ;
FUNCTION        FCPYM4  = COPYM  (4,    0)                      ;
FUNCTION        FCUT    = CUT    (1      )                      ;
FUNCTION        FDIST2  = DIST   (3      )                      ;
FUNCTION        FDIST3  = DIST   (3      )                      ;
FUNCTION        FDIST4  = DIST   (3      )                      ;
FUNCTION        FDUST   = COUNT  (1      )                      ;
FUNCTION        FEND    = OUT1   (0,    0), QUEUE  (QUEE, 1)    ;
FUNCTION        FGEN1   = COPYBK (1,    0), CNTGE  (M      )    ;
FUNCTION        FGEN2   = COPYBK (4,    1), CNTGE  (H/4    )    ;
FUNCTION        FGEN3   = COPYBK (1,    0), CNTGE  (M      )    ;
FUNCTION        FGEN4   = COPYBK (4,    1), CNTGE  (H/4    )    ;
FUNCTION        FGEN5   = COPYBK (1,    0), CNTGE  (M      )    ;
FUNCTION        FGEN6   = COPYBK (4,    1), CNTGE  (H/4    )    ;
FUNCTION        FGEN7   = COPYBK (1,    0), CNTGE  (M      )    ;
FUNCTION        FGEN8   = COPYBK (4,    1), CNTGE  (H/4    )    ;
FUNCTION        FINCL   = ADD   (BRC, OVF), RDCYCS (ONEL, 1)    ;
FUNCTION        FIRHAC  = INC  (CNOP, XCH), RDCYCS (RHA1, 1)    ;
FUNCTION        FIRHAC2 = INC (XX,CNOP,XCH),RDCYCS (RHA1, 1)    ;
FUNCTION        FIRHL1  = INC    (XCH    ), RDCYCS (RHA1, 1)    ;
FUNCTION        FIRHL2  = INC    (XCH    ), RDCYCS (RHA2, 1)    ;
FUNCTION        FIRHL3  = INC    (XCH    ), RDCYCS (RHA3, 1)    ;
FUNCTION        FIWHA   = INC    (XCH    ), RDCYCS (WHA,  1)    ;
FUNCTION        FMUL1   = MUL    (Y      ), RDCYCS (MSKL1,3)    ;
FUNCTION        FMUL2   = MUL    (Y      ), RDCYCS (MSKL2,3)    ;
FUNCTION        FMUL3   = MUL    (Y      ), RDCYCS (MSKL3,3)    ;
FUNCTION        FNOP    = NOP    (XX     )                      ;
FUNCTION        FQUE0   = QUEUE  (QUE0, 1)                      ;
```

Figure 5-15. 3 x 3 Mask Source Program (cont)

```
FUNCTION        FQUE1   = QUEUE  (QUE1, 1)                          ;
FUNCTION        FQUE2   = QUEUE  (QUE2, 1)                          ;
FUNCTION        FQUE3   = QUEUE  (QUE3, 1)                          ;
FUNCTION        FQUE4   = QUEUE  (QUE4, 1)                          ;
FUNCTION        FQUE5   = QUEUE  (QUE5, 1)                          ;
FUNCTION        FQUE6   = QUEUE  (QUE6, 1)                          ;
FUNCTION        FQUE7   = QUEUE  (QUE7, 1)                          ;
FUNCTION        FQUE8   = QUEUE  (QUE8, 1)                          ;
FUNCTION        FR1LAD  = ADD,             RDCYCS (ONEL, 1)         ;
FUNCTION        FR2LAD  = ADD,             RDCYCS (TWOL, 1)         ;
FUNCTION        FRDAND  = AND,             RDCYCS (MASKH,1)         ;
FUNCTION        FRDRW1  = ADD,             RDCYCS (ROW1, 5)         ;
FUNCTION        FRDRW2  = ADD,             RDCYCS (ROW2, 5)         ;
FUNCTION        FRDRW3  = RDCYCS (ROW2, 1)                          ;
FUNCTION        FRDSHR  = SHR,             RDCYCS (SHRX, 1)         ;
FUNCTION        FREDL1  = OUT1   (1,    0)                          ;
FUNCTION        FREDL2  = OUT1   (2,    1)                          ;
FUNCTION        FREDL3  = OUT1   (3,    2)                          ;
FUNCTION        FRHAL1  = OUT1   (1,  70H), WRCYCS (RHA1, 1)         ;
FUNCTION        FRHAL2  = OUT1   (2,  70H), WRCYCS (RHA2, 1)         ;
FUNCTION        FRHAL3  = OUT1   (3,  70H), WRCYCS (RHA3, 1)         ;
FUNCTION        FWRITE  = OUT2 (5, 20H, 0), QUEUE  (QUEW,16)         ;
FUNCTION        FWRRW1  = WRCYCS (ROW1, 5), 1                       ;
FUNCTION        FWRRW2  = WRCYCS (ROW2, 5)                          ;
FUNCTION        FWRRW3  = WRCYCS (ROW3, 1)                          ;
FUNCTION        FWHA    = OUT1   (5,  10H), WRCYCS (WHA,  1)        ;
;
```

Figure 5-15. 3 x 3 Mask Source Program (cont)

```
;*****************************************************************
;                          MEMORY
;*****************************************************************
;
MEMORY  MASKH   = 0FFH                                          ;
MEMORY  MSKL1   = 1,    2,      1                               ;
MEMORY  MSKL2   = 2,    4,      2                               ;
MEMORY  MSKL3   = 1,    2,      1                               ;
MEMORY  ONEL    = L                                             ;
MEMORY  QUE0    = AREA ( 1)                                     ;
MEMORY  QUE1    = AREA ( 1)                                     ;
MEMORY  QUE2    = AREA ( 1)                                     ;
MEMORY  QUE3    = AREA ( 1)                                     ;
MEMORY  QUE4    = AREA ( 1)                                     ;
MEMORY  QUE5    = AREA ( 1)                                     ;
MEMORY  QUE6    = AREA ( 1)                                     ;
MEMORY  QUE7    = AREA ( 1)                                     ;
MEMORY  QUE8    = AREA ( 1)                                     ;
MEMORY  QUEA1   = AREA ( 4)                                     ;
MEMORY  QUEA2   = AREA ( 4)                                     ;
MEMORY  QUEA3   = AREA ( 4)                                     ;
MEMORY  QUEA4   = AREA ( 4)                                     ;
MEMORY  QUEA5   = AREA ( 4)                                     ;
MEMORY  QUEA6   = AREA ( 4)                                     ;
MEMORY  QUEE    = AREA ( 1)                                     ;
MEMORY  QUEW    = AREA (16)                                     ;
MEMORY  RHA1    = AREA ( 1)                                     ;
MEMORY  RHA2    = AREA ( 1)                                     ;
MEMORY  RHA3    = AREA ( 1)                                     ;
MEMORY  ROW1    = AREA ( 5)                                     ;
MEMORY  ROW2    = AREA ( 5)                                     ;
MEMORY  ROW3    = AREA ( 1)                                     ;
MEMORY  SHRX    = BIT                                           ;
MEMORY  TWOL    = 2*L                                           ;
MEMORY  WHA     = AREA ( 1)                                     ;
;
```

Figure 5-15. 3 x 3 Mask Source Program (cont)

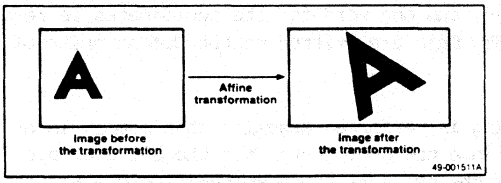
```
;*****************************************************************
;                        START
;*****************************************************************
;
START                                                              ;
DATA    EXEC    (MASK,  IR,     RHADRS )                         ;
DATA    EXEC    (MASK,  IW,     WHADRS )                         ;
DATA    EXEC    (MASK,  DO,     WLADRS )                         ;
DATA    EXEC    (MASK,  STTKN,  RLADRS )                         ;
;
END;
```

Section 6 Affine Transformation

Affine transformation permits such processing tasks as image enlargement, rotation, and translation on an integral basis. Figure 6-1 is a simple example.

Affine transformation takes more processing time than programs incorporating specific algorithms, such as reverse double enlargement, quadruple enlargement, and 90o rotation, so its use should be carefully chosen based on the application.

Figure 6-1. Affine Transformation Example



ALGORITHM

Logical Coordinates and Physical Addresses

An image memory can be pictured as having L horizontal words and V vertical lines (that is, L horizontal pixels and V vertical pixels). Image memory has two image areas: the SRC and DST areas, which have logical coordinates (x,y) and (X,Y), respectively. Their logical origins (0,0) correspond to physical addresses RDBASE and WRBASE, respectively. The physical address for the arbitrary point (x',y') on the SRC image area can be determined using the following formula:

RDBASE + y'L + x' (Equation 1)

Similarly, the physical address of (X',Y') can be obtained from:

WRBASE + Y'L + X' (Equation 2)

In the affine transformation program listed at the end of this section, the logical coordinates (X',Y') for the DST area are related to logical coordinates (x',y') for SRC, as follows:

$$x' = aX' + bY' + c \qquad \text{(Equation 3)}$$
$$y' = dX' + eY' + f$$

Where a, b, c, d, e, and f are coefficients whose values determine whether the transformation is for an enlargement, reduction, rotation, or translation. The two parts of equation 3 specify coordinates before the transformation when coordinates after the transformation are given.

The logical coordinates (X',Y') are allowed to vary from (0,0) to (max(X), max(Y)). Logical coordinates (x',y') corresponding to each set of coordinates are calculated, and one word for the coordinates is read from the SRC and written to DST. Data are written to the DST in units of words (16 bits).

Image memory bank switching is performed assuming that memory is organized as shown in figure 6-2. Since the coordinates for the SRC area are determined from those for the DST area, when an overflow due to a calculation of the next line address occurs during the generation of a write low address, the write high addresses for the DST image area are updated. Read high addresses for the SRC area are obtained by the following formula:

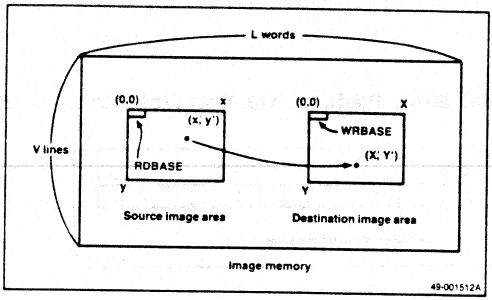
$$(y' + y'')/(65536/L) + RDBASEH$$

Where

$y'' = RDBASE/L$
RDBASEH is the high address for RDBASE

Unlike the affine transformation for binary image processing, the affine transformation for gray scale image processing represents a pixel with a word. This means that it is not necessary to calculate bit positions in order to extract bits from each word in the SRC data. However, the large memory requirements make IM bank switching a necessity.

Figure 6-2. Logical Coordinates and Physical Addresses



Processing Flow

In this program, pixel data is stored in the lower byte of a word, and the DST area consists of 256 x 256 pixels.

First, (X',Y') is set to (0,0). While allowing this to vary up to (255,0), the corresponding values of (x',y') are calculated. Since 256 pixels cannot be calculated at once, 256/16, or 16 pixels, are calculated at a time. Since Y remains unchanged during this process, in the first processing of a line, bY' + c and eY' + f in equation 3 are calculated only once. They are then added to the values of aX' and dX'.

A DST address is generated, and from this, a point (x',y') in the SRC area is obtained. Then, the actual physical address of (x',y') is determined. Using this physical address, the SRC data are read and written to the DST area.

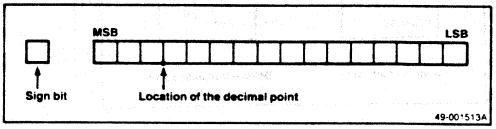
Since a line in the DST area has 256 pixels, writing 256 words completes the processing of the line. The processing is repeated line-by-line for the 256 lines of the DST area, until the entire image is processed.

Computation Precision

On the uPD7281, multiplication involves the computation of 16 bits x 16 bits; the results can be extracted as upper 16 bits and lower 16 bits. Since in this program the results are extracted as the 16 bits only, the following provisions are made in regard to the values of coefficients a, b, c, d, e, and f, and coordinates (X',Y') and (x',y'):

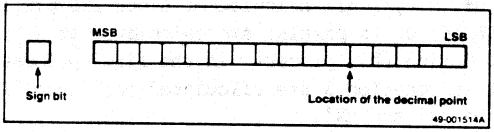
- The coefficients a, b, d, and e are treated in 16-bit precision (17 bits including the sign bit), with the decimal point between bits 12 and 13. See figure 6-3.

Figure 6-3. Decimal Point Position for Coefficients a, b, d, and e



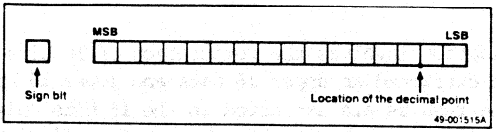
- Although logically the DST coordinates (X',Y') vary in increments of 1, since the decimal point for X' and Y' is between bits 4 and 5, they actually vary in increments of 32. See figure 6-4.

Figure 6-4. Decimal Point Position for X' and Y'



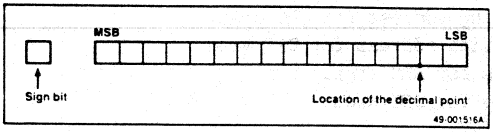
- Since the values of aX', bY', dX' and eY' are obtained by extracting the upper 16 bits from 32-bit computation results, the decimal point is located between bits 1 and 2. See figure 6-5.

Figure 6-5. Decimal Point Position of aX', bY', dX', and eY'



- To ensure compatibility with the above precision, the values of the coefficients c and f also have the decimal point located between bits 1 and 2. See figure 6-6.

Figure 6-6. Decimal Point Position for Coefficients C and F



- Accordingly, the values of (x',y') have their decimal point located between bits 1 and 2. Therefore, a value obtained is shifted to the right by two bits to reduce it into an integer value. Note that the values of (x',y') have a precision of two bits below the decimal point. Suppose the values of x' and y' were 0.75 and 1.75. Then a more correct image might result if the values were rounded up to 1 and 2, instead of being truncated below the decimal point through bit shifts. Therefore, in this program 0.5 is added to the values of x' and y' obtained, followed by a bit shift two bits to the right. In actual implementation, 0.5 is added to coefficients c and f before they enter into the calculation.

PARAMETERS

Table 6-1 lists allowable values for the parameters, which are defined as follows.

Assembler-Coded Parameters

L - Number of horizontal image memory words.
H - Number of horizontal DST words.
M - Number of vertical DST words.

Start-Up Token Defined Parameters

AVAL - Affine transformation coefficient a
BVAL - " b
CVAL - " c
DVAL - " d
EVAL - " e
FVAL - " f
RHADR - Read-high starting address for SRC area.
WHADR - Write-high starting address for DST area.
RDBASE - Starting address for SRC area.
WRBASE - Starting address for DST area.
ZERO - Initial value of Y.

Table 6-1. Parameter Values

Parameter	Range Min	Range Max	Value in Program Example
L	0	65,535	1024
H	1	256	256
M	1	256	256
AVAL	-7.999	7.999	1
BVAL	-7.999	7.999	0
CVAL	-2,047.9	2,047.9	0
DVAL	-7.999	7.999	0
EVAL	-7.999	7.999	1
FVAL	-2,047.9	2,047.9	0
RHADRS	0	65,535	0
WHADRS	0	65,535	0
RDBASE	0	65,535	0
WRBASE	0	65,535	256
ZERO	0		0

FLOW GRAPH EXPLANATION

Table 6-2 explains the principal nodes.

Table 6-2. Principal Nodes

Node	Function
FGEN1	Counts the DST lines. If the parameter for the CNTGE instruction is 256, then 256 lines of images are processed.
FREDB, FREDC, FSAVW1	Calculate bY' + c and write the results in W1BUF. This value needs to be calculated only once for a given value of Y; whenever it is to be added to aX', it is read from W1BUF.
FREDE, FREDF, FSAVW2	Calculate eY' + f and write the results to W2BUF.
FGEN2	Increments X' from 0 to max(X), outputting the results as a token. Since the values 0 - max(X) cannot be output in a single operation, they are output in groups of 16.
FREDD, FREDW2, FREDS1	Calculate dX' + eY' + f, and perform a right shift on the results to truncate the value below the decimal point.
FREDA, FREDW1, FREDS2	Calculate aX' + bY' + c, and perform a right shift on the results to truncate the value below the decimal point.
FREDHN, FADDXY, FREDRL	Given computed values of (x',y'), these nodes calculate the physical address for the coordinates.
FREAD	Reads SRC data.
FCNT	Detects when the setting of 16 words of DST data has completed (16 tokens have arrived), and starts processing the next 16 words.
FWRITE	Writes DST data to DST addresses.
FGEN3	Generates the starting addresses for DST lines based on WRBASE, the DST starting address.

Table 6-2. Principal Nodes (cont)

Node	Function
FGEN4	Generates addresses for one line of DST addresses.
FREDBY, FREDDV, FREDRH	Calculate read high addresses in the SRC.
FREDWH, FWRTHI, FSAVWHl	Calculate write high addresses in the DST.
FREDL	Updates addresses for the generation of starting addresses for DST lines, and checks for overflows.

Figure 6-7. Affine Transformation Flow Graph 1

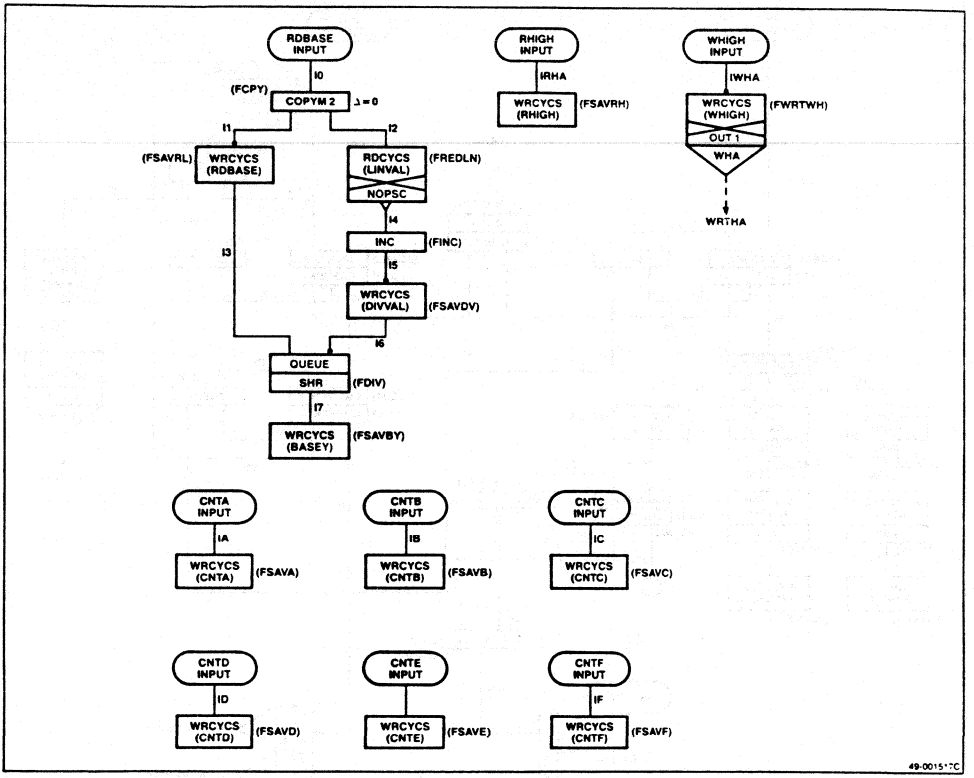


Figure 6-8. Affine Transformation Flow Graph 2

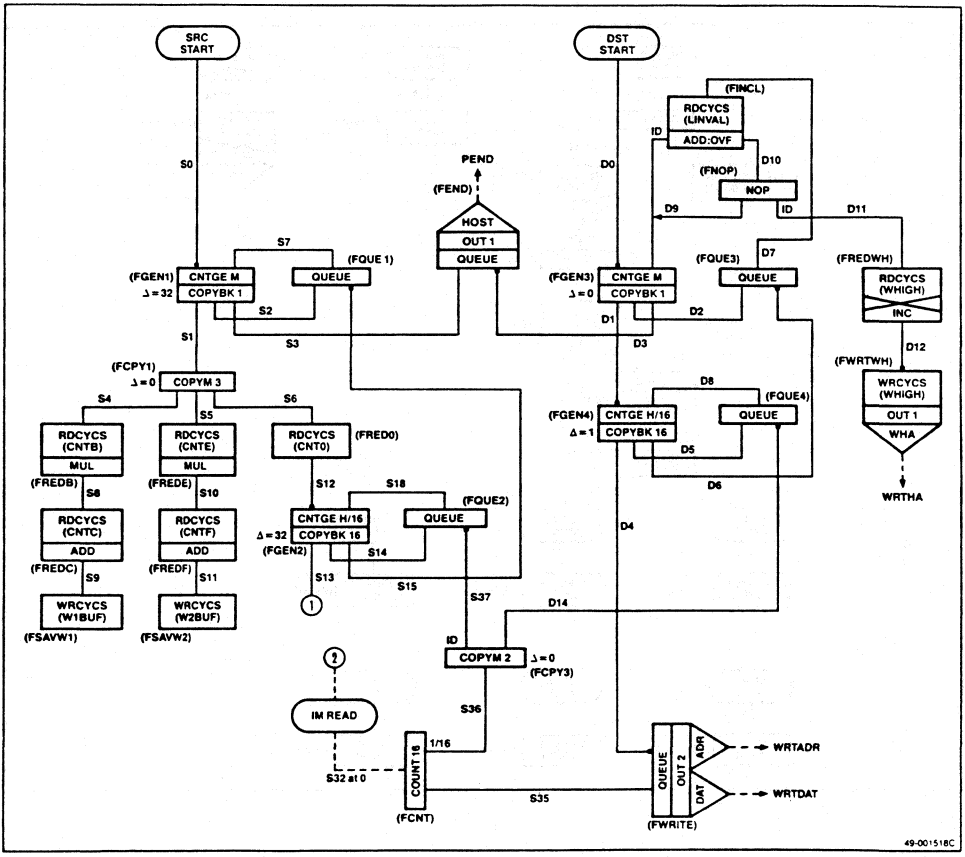


Figure 6-8. Affine Transformation Flow Graph 2 (cont)

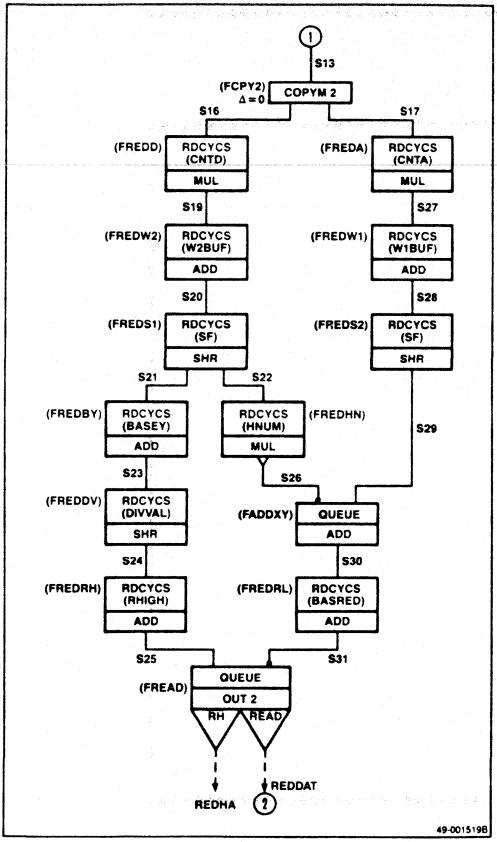


Figure 6-10. Affine Transformation Source Program

```
;*****************************************************************
;
;                       AFFINE
;
;*****************************************************************
;
MODULE  IMPP    =       8                                       ;
;
EQUATE  L       =       1024                                    ;
EQUATE  H       =       256                                     ;
EQUATE  M       =       256                                     ;
;
EQUATE  AVAL    =       2000H                                   ;
EQUATE  BVAL    =       0                                       ;
EQUATE  CVAL    =       0                                       ;
EQUATE  DVAL    =       0                                       ;
EQUATE  EVAL    =       2000H                                   ;
EQUATE  FVAL    =       0                                       ;
EQUATE  RHADDR  =       0                                       ;
EQUATE  WHADDR  =       0                                       ;
EQUATE  RDBASE  =       0                                       ;
EQUATE  WRBASE  =       256H                                    ;
EQUATE  ZERO    =       0                                       ;
;
EQUATE  FINE    =       0                                       ;
EQUATE  READ    =       4                                       ;
EQUATE  WRITE   =       5                                       ;
;
;*****************************************************************
;                       INPUT
;*****************************************************************
;
INPUT   I0,     IRHA,   IWHA,   IA,     IB,     IC,     ID      ;
INPUT   IE,     IF,     S0,     D0,     S32 AT 0                ;
;
;*****************************************************************
;                       OUTPUT
;*****************************************************************
;
OUTPUT  WRIWHA, SETRHA, REDDAT, SETWHA, WRIDAT, WRITADR, PEND   ;
;
```

Figure 6-10. Affine Transformation Source Program (cont)

```
;*****************************************************************
;                          LINK
;*****************************************************************
;
LINK                          = FSAVA   (IA      )          ;
LINK                          = FSAVB   (IB      )          ;
LINK                          = FSAVC   (IC      )          ;
LINK                          = FSAVD   (ID      )          ;
LINK                          = FSAVE   (IE      )          ;
LINK                          = FSAVF   (IF      )          ;
LINK                          = FSAVRH  (IRHA    )          ;
LINK    ISWHA                 = FSAVWH  (  ,IWHA)           ;
LINK    WRTWHA                = FWRTWH  (ISWHA   )          ;
LINK    I1,     I2            = FCPY    (I0      )          ;
LINK    I3                    = FSAVRL  (  , I1  )          ;
LINK    I4                    = FREDLN  (I2      )          ;
LINK    I5                    = FINC    (I4      )          ;
LINK    I6                    = FSAVDV  (  , I5  )          ;
LINK    I7                    = FDIV    (I3, I6  )          ;
LINK                          = FSAVBY  (I7      )          ;
;
LINK    S1,     S2,     S3    = FGEN1   (S7, S0  )          ;
LINK    S4,     S5,     S6    = FCPY1   (S1      )          ;
LINK    S7                    = FQUE1   (S2, S15)           ;
LINK    PEND                  = FEND    (S3, D3  )          ;
LINK    S8                    = FREDB   (S4      )          ;
LINK    S9                    = FREDC   (S8      )          ;
LINK                          = FSAVW1  (S9      )          ;
LINK    S10                   = FREDE   (S5      )          ;
LINK    S11                   = FREDF   (S10     )          ;
LINK                          = FSAVW2  (S11     )          ;
LINK    S12                   = FRED0   (S6      )          ;
LINK    S13,    S14,    S15   = FGEN2   (S18,S12)           ;
LINK    S16,    S17           = FCPY2   (S13     )          ;
LINK    S18                   = FQUE2   (S14,S37)           ;
LINK    S19                   = FREDD   (S16     )          ;
LINK    S20                   = FREDW2  (S19     )          ;
LINK    S21,    S22           = FREDS1  (S20     )          ;
LINK    S23                   = FREDBY  (S21     )          ;
LINK    S24                   = FREDDV  (S23     )          ;
LINK    S25                   = FREDRH  (S24     )          ;
LINK    S26                   = FREDHN  (S22     )          ;
```

Figure 6-10. Affine Transformation Source Program (cont)

```
LINK    S27                     = FREDA   (S17    )          ;
LINK    S28                     = FREDW1  (S27    )          ;
LINK    S29                     = FREDS2  (S28    )          ;
LINK    S30                     = FADDXY  (S29,S26)          ;
LINK    S31                     = FREDRL  (S30    )          ;
LINK    SETRHA, REDDAT          = FREAD   (S25,S31)          ;
LINK    S35,    S36             = FCNT    (S32    )          ;
LINK    WRTDAT, WRTADR          = FWRITE  (S35, D4)          ;
LINK    S37,    D14             = FCPY3   (S36    )          ;
LINK    D1,     D2,     D3      = FGEN3   (D9,  D0)          ;
LINK    D4,     D5,     D6      = FGEN4   (D8,  D1)          ;
LINK    D7                      = FQUE3   (D2,  D6)          ;
LINK    D8                      = FQUE4   (D5, D14)          ;
LINK    D9,     D10             = FREDL   (D7     )          ;
LINK    D11,    D9              = FNOP    (D10    )          ;
LINK    D12,    D13             = FREDWH  (D11    )          ;
LINK    SETWHA                  = FWRTHI  (D12    )          ;
LINK                            = FSAVWH1 (D13    )          ;
;
;*****************************************************************
;                       FUNCTION
;*****************************************************************
;
FUNCTION        FEND    = OUT1   (FINE,  0), QUEUE  (QUEE, 1)  ;
FUNCTION        FREAD   = OUT2 (READ,70H,0), QUEUE  (QUER,16)  ;
FUNCTION        FWRTWH  = OUT1  (WRITE,10H)                    ;
FUNCTION        FWRTHI  = OUT1  (WRITE,10H)                    ;
FUNCTION        FWRITE  = OUT2 (WRITE,20H,0),QUEUE  (QUEW,16)  ;
FUNCTION        FGEN1   = COPYBK (1,    32), CNTGE  (M      )  ;
FUNCTION        FGEN2   = COPYBK (16,   32), CNTGE  (H/16   )  ;
FUNCTION        FGEN3   = COPYBK (1,     0), CNTGE  (M      )  ;
FUNCTION        FGEN4   = COPYBK (16,    1), CNTGE  (H/16   )  ;
FUNCTION        FQUE1   = QUEUE  (QUE1,  1)                    ;
FUNCTION        FQUE2   = QUEUE  (QUE2,  1)                    ;
FUNCTION        FQUE3   = QUEUE  (QUE3,  1)                    ;
FUNCTION        FQUE4   = QUEUE  (QUE4,  1)                    ;
FUNCTION        FCPY    = COPYM  (2,     0)                    ;
FUNCTION        FCPY1   = COPYM  (3,     0)                    ;
FUNCTION        FCPY2   = NOP    (XX      )                    ;
FUNCTION        FCPY3   = COPYM  (2,     0)                    ;
FUNCTION        FSAVRL  = WRCYCS (BASRED,1)                    ;
FUNCTION        FSAVDV  = WRCYCS (DIVVAL,1)                    ;
```

Figure 6-10. Affine Transformation Source Program (cont)

```
FUNCTION        FSAVBY  = WRCYCS (BASEY, 1)                              ;
FUNCTION        FSAVRH  = WRCYCS (RHA,   1)                              ;
FUNCTION        FSAVWH  = WRCYCS (WHA,   1)                              ;
FUNCTION        FSAVA   = WRCYCS (CNTA,  1)                              ;
FUNCTION        FSAVB   = WRCYCS (CNTB,  1)                              ;
FUNCTION        FSAVC   = WRCYCS (CNTC,  1)                              ;
FUNCTION        FSAVD   = WRCYCS (CNTD,  1)                              ;
FUNCTION        FSAVE   = WRCYCS (CNTE,  1)                              ;
FUNCTION        FSAVF   = WRCYCS (CNTF,  1)                              ;
FUNCTION        FSAVW1  = WRCYCS (W1BUF, 1)                              ;
FUNCTION        FSAVW2  = WRCYCS (W2BUF, 1)                              ;
FUNCTION        FSAVWH1 = WRCYCS (WHA,   1)                              ;
FUNCTION        FRED0   = RDCYCS (CNT0,  1)                              ;
FUNCTION        FREDLN  = NOPSC  (Y,    XCH), RDCYCS (LNUM, 1)           ;
FUNCTION        FREDA   = MUL,                RDCYCS (CNTA, 1)           ;
FUNCTION        FREDB   = MUL,                RDCYCS (CNTB, 1)           ;
FUNCTION        FREDC   = ADD,                RDCYCS (CNTC, 1)           ;
FUNCTION        FREDD   = MUL,                RDCYCS (CNTD, 1)           ;
FUNCTION        FREDE   = MUL,                RDCYCS (CNTE, 1)           ;
FUNCTION        FREDF   = ADD,                RDCYCS (CNTF, 1)           ;
FUNCTION        FREDW1  = ADD,                RDCYCS (W1BUF,1)           ;
FUNCTION        FREDW2  = ADD,                RDCYCS (W2BUF,1)           ;
FUNCTION        FREDS1  = SHR    (XX      ), RDCYCS (SF,   1)           ;
FUNCTION        FREDS2  = SHR,                RDCYCS (SF,   1)           ;
FUNCTION        FREDBY  = ADD,                RDCYCS (BASEY,1)           ;
FUNCTION        FREDDV  = SHR,                RDCYCS (DIVVAL,1)          ;
FUNCTION        FREDRH  = ADD,                RDCYCS (RHA,  1)           ;
FUNCTION        FREDHN  = MUL    (Y       ), RDCYCS (HNUM, 1)           ;
FUNCTION        FREDRL  = ADD,                RDCYCS (BASRED,1)          ;
FUNCTION        FREDL   = ADD   (X,BRC,OVF), RDCYCS (LNUM, 1)           ;
FUNCTION        FREDWH  = INC    (XX,   XCH), RDCYCS (WHA,  1)           ;
FUNCTION        FDIV    = SHR,                QUEUE  (QUES, 1)           ;
FUNCTION        FADDXY  = ADD,                QUEUE  (QUEA,16)           ;
FUNCTION        FCNT    = COUNT  (16      )                              ;
FUNCTION        FNOP    = NOP    (XX      )                              ;
FUNCTION        FINC    = INC                                           ;
;
```

Figure 6-10. Affine Transformation Source Program (cont)

```
;*****************************************************************
;                          MEMORY
;*****************************************************************
;
MEMORY  BASEY   = AREA ( 1)                                      ;
MEMORY  BASRED  = AREA ( 1)                                      ;
MEMORY  CNT0    = 0                                              ;
MEMORY  CNTA    = AREA ( 1)                                      ;
MEMORY  CNTB    = AREA ( 1)                                      ;
MEMORY  CNTC    = AREA ( 1)                                      ;
MEMORY  CNTD    = AREA ( 1)                                      ;
MEMORY  CNTE    = AREA ( 1)                                      ;
MEMORY  CNTF    = AREA ( 1)                                      ;
MEMORY  DIVVAL  = AREA ( 1)                                      ;
MEMORY  HNUM    = H                                              ;
MEMORY  LNUM    = L                                              ;
MEMORY  QUE1    = AREA ( 1)                                      ;
MEMORY  QUE2    = AREA ( 1)                                      ;
MEMORY  QUE3    = AREA ( 1)                                      ;
MEMORY  QUE4    = AREA ( 1)                                      ;
MEMORY  QUEA    = AREA (16)                                      ;
MEMORY  QUEE    = AREA ( 1)                                      ;
MEMORY  QUER    = AREA (16)                                      ;
MEMORY  QUES    = AREA ( 1)                                      ;
MEMORY  QUEW    = AREA (16)                                      ;
MEMORY  RHA     = AREA ( 1)                                      ;
MEMORY  SF      = 2                                              ;
MEMORY  WHA     = AREA ( 1)                                      ;
;
```

Figure 6-10. Affine Transformation Source Program (cont)

```
;*****************************************************************
;                        START
;*****************************************************************
;
START                                                              ;
;
DATA    EXEC    (IMPP,  I0,     RDBASE)                          ;
DATA    EXEC    (IMPP,  IRHA,   RHADDR)                          ;
DATA    EXEC    (IMPP,  IWHA,   WHADDR)                          ;
DATA    EXEC    (IMPP,  IA,     AVAL  )                          ;
DATA    EXEC    (IMPP,  IB,     BVAL  )                          ;
DATA    EXEC    (IMPP,  IC,     CVAL  )                          ;
DATA    EXEC    (IMPP,  ID,     DVAL  )                          ;
DATA    EXEC    (IMPP,  IE,     EVAL  )                          ;
DATA    EXEC    (IMPP,  IF,     FVAL  )                          ;
DATA    EXEC    (IMPP,  D0,     WRBASE)                          ;
DATA    EXEC    (IMPP,  S0,     ZERO  )                          ;
;
END;
```

VOLUME III

NUMERICAL CALCULATIONS

Introduction

This Application Library (Volume III) addresses "Numerical Calculations", which is one of the application areas of the uPD7281. The applications presented in the following pages include:

* Product of a Matrix and a Vector
* Multiplication
* Floating Point Computations
* Distance Squared Calculations
* Fast Fourier Transform
* Evaluation of Polynominals

Chapter 1

System Configuration

1.1 System Configuration

The programs described in Chapters 2 through 6 are assumed to run on a system as shown in Figure 1-1, while Chapter 7 contains programs intended for a system as shown in Figure 1-2. Since the system shown in Figure 1-2 uses the uPD9305, a peripheral support chip for the uPD7281, the tokens used to access the image memory (IM) are defined by the uPD9305. Therefore, the programs described in Chapter 7 are not directly operable on systems that do not use the uPD9305. Please consult the uPD9305 (MAGIC) User's Manual for an explanation of these tokens.

Figure 1-1
System Configuration 1

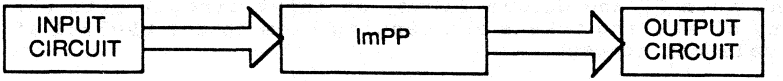
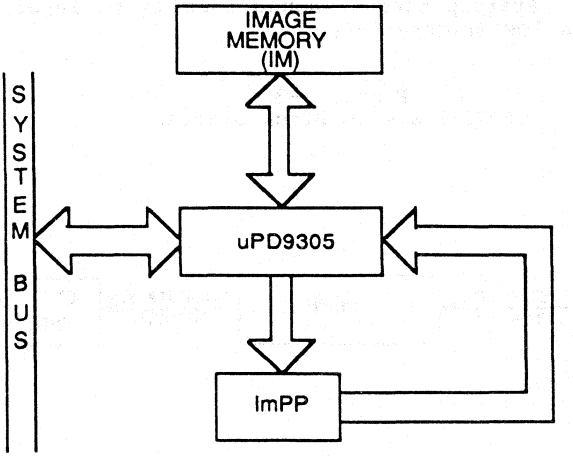


Figure 1-2
System Configuration 2



1.2 Data Input Methods

The programs contained in Chapters 2 through 6, are based on the assumption that the fixed values of input data are predefined in the uPD7281 Data Memory (DM) while the variables are treated as startup tokens. Some of these programs are capable of handling continuous input, others are not.

With the programs capable of handling continuous input, if a large volume of tokens are input too rapidly, a Generator Queue (GQ) overflow could occur. This happens when several Generate (GE) instructions strung together transmit tokens too rapidly, such that the number of tokens waiting to enter the Processing Unit (PU) in order to be copied exceeds the available GQ space. This is also true of the Data Queue (DQ) with tokens waiting for access to the PU. The DQ overflow problem could be avoided through the input of a Set Operation Mode (SETMD) token to limit the rate of input tokens; however, a DQ overflow could still result if tokens are not able to be output from the uPD7281 fast enough. If the Output Queue (OQ) is full, output tokens will accumulate in the DQ until the DQ overflows, causing the uPD7281 to enter the Break Mode.

To prevent such DQ and GQ overflows, an input circuit with the capability of storing tokens and sending them to the uPD7281 at set intervals can be used.

1.2.1 Setting up the Input Circuit

The operation of the Link Table (LT) is determined through simulation studies. An input circuit (Figure 1-3) is set up to prevent ImPP overflows, and a timer is provided so that the startup tokens will be input to the ImPP at equal intervals. Startup tokens should ideally be input when the LT usage is low (Figure 1-4).

Figure 1-3
uPD7281 System Block Diagram

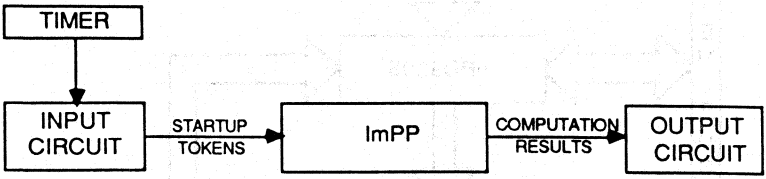
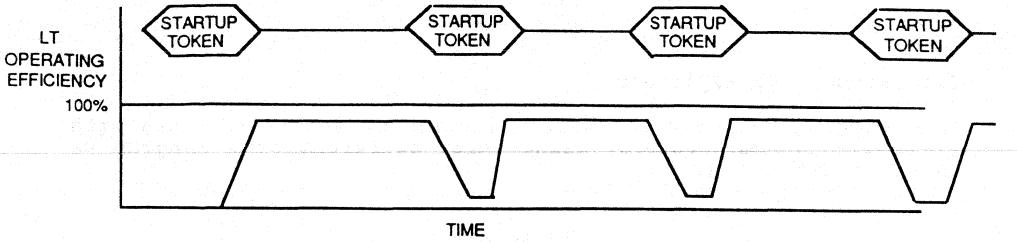


Figure 1-4
Token Input and Link Table Utilization

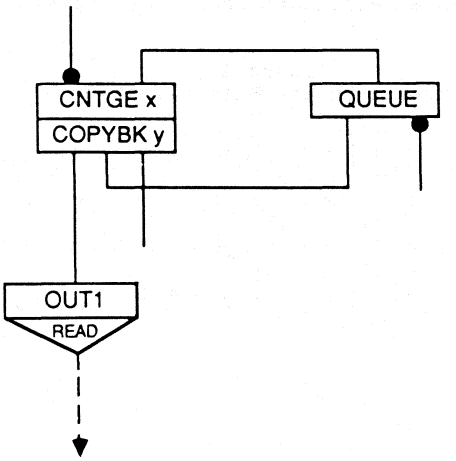


1.2.2 Generating Memory Addresses

The data to be processed can be stored in the IM. The procedures for generating the IM addresses, reading the data, and performing the necessary computations are built into the program, using the same idea as IM address generation and image data read employed in uPD7281 image processing programs.

Generating address for reading the input data from the IM is controlled by setting appropriate x and y values for the CNTGE and COPYBK instruction parameters in Figure 1-5 and by defining a feedback token which is transmitted as an FTRC=1 token to the QUEUE instruction.

Figure 1-5
Image Memory Address Generation



Chapter 2

Matrix-Vector Product

2.1 Processing Explained

This program performs the multiplication of 4-by-4 matrices with 4-element column vectors, using three different data lengths as shown in Table 2-1.

Table 2-1
Data Lengths Used in Matrix-Vector Product

MATRIX ELEMENT	INPUT VECTOR COMPONENT	OUTPUT VECTOR COMPONENT
17 BITS	17 BITS	17 BITS
17 BITS	17 BITS	33 BITS
33 BITS	33 BITS	33 BITS

(INCLUDES SIGN BIT)

The matrix-vector multiplication supported by this program can be used in affine and perspective transformations.

2.2 Algorithm (Common to the Three Data Lengths)

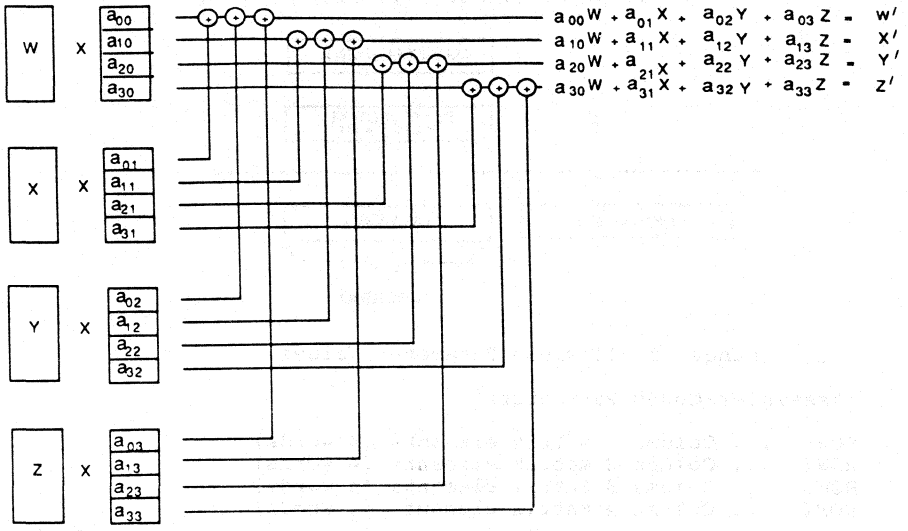
The product between a matrix and vector is expressed as follows:

Figure 2-1
Matrix-Vector Multiplication

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} W \\ X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} a_{00}W + a_{01}X + a_{02}Y + a_{03}Z \\ a_{10}W + a_{11}X + a_{12}Y + a_{13}Z \\ a_{20}W + a_{21}X + a_{22}Y + a_{23}Z \\ a_{30}W + a_{31}X + a_{32}Y + a_{33}Z \end{bmatrix} = \begin{bmatrix} W' \\ X' \\ Y' \\ Z' \end{bmatrix}$$

As shown in Figure 2-1, the product between a matrix and a column vector is obtained by multiplying the first component of the input vector with the elements of the first column of the matrix, the second component of the input vector with the elements of the second column of the matrix, and so forth, and adding each row of the matrix-vector component products column by column. The results constitute the components of the output vector. Thus, the element X in the input vector is multiplied with elements a_{01}, a_{11}, a_{21}, and a_{31} in column 2 of the matrix. The output vector component X' is the sum $a_{10} \times W$, $a_{11} \times X$, $a_{12} \times Y$, and $a_{13} \times Z$.

Figure 2-2
Matrix-Vector Product



2.3 17 Bit x 17 Bit = 17 Bit

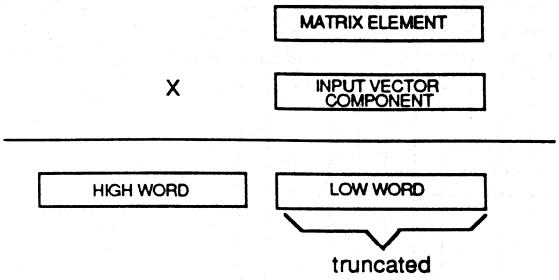
2.3.1 Algorithm

The data for matrix elements and input vector components are expressed in 17 bits, 16 bits for the numeric value and a sign bit. Although these data may have the decimal point anywhere, since the addition of the results of the multiplication are performed by truncating the low word and adding the high words column by column, the products of the multiplication must have the decimal point at the same bit position.

If the addition of the high words results in either an overflow or an underflow, the value in excess of 17 bits will be discarded. Consequently, the components of the output vector sought are expressed in 17 bits, with 16 numeric value bits and a sign bit.

In setting up matrix element and vector component data, it is advisable to place the values near the MSB of each word and ensure that all data values have the decimal point at the same bit position.

Figure 2-3
Multiplication Between a Matrix Element and a Vector Component
(17 bit x 17 bit = 17 bit)



2.3.2 Range of Allowable Parameter Values

<Assembler-Coded Parameters>

ROW1 .... Column 1 matrix elements (4 words)
ROW2 .... Column 2 matrix elements (4 words)
ROW3 .... Column 3 matrix elements (4 words)
ROW4 .... Column 4 matrix elements (4 words)

<Startup Token-Defined Parameters>

W0 .... Input vector row 1 component
X0 .... Input vector row 2 component
Y0 .... Input vector row 3 component
Z0 .... Input vector row 4 component

Table 2-2 indicates the allowable parameter values.

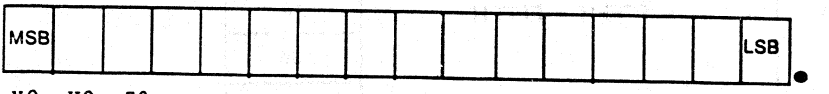
Table 2-2
Range of Allowable Parameter Values
(17 bit x 17 bit = 17 bit)

PARAMETER	RANGE OF ALLOWABLE VALUES	VALUE IN PROGRAM EXAMPLE
ROW1	−FFFFH ~ FFFFH	10
		50
		90
		130
ROW2	−FFFFH ~ FFFFH	20
		60
		100
		140
ROW3	−FFFFH ~ FFFFH	30
		70
		110
		150
ROW4	−FFFFH ~ FFFFH	40
		80
		120
		160
W0	−FFFFH ~ FFFFH	199AH
X0	−FFFFH ~ FFFFH	3334H
Y0	−FFFFH ~ FFFFH	4CCDH
Z0	−FFFFH ~ FFFFH	6667H

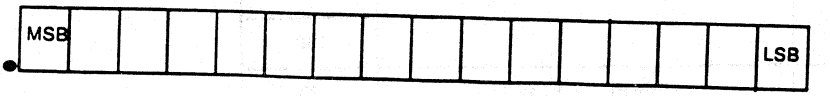
The program examples referred to in Table 2-2 use the following decimal point positions for the parameters:

Figure 2-4
Decimal Position for Matrix Multiplication Example

ROW1, ROW2, ROW3, ROW4



W0, X0, Y0, Z0



W0 = 199AH = 0.1
X0 = 3334H = 0.2
Y0 = 4CCDH = 0.3
Z0 = 6667H = 0.4

2.3.3 The Flow Graph Explained

This program stores the elements of a matrix in the DM (in ROW1 through ROW4 through the use of MEMORY statements in the source program) column by column. The four input vector components are used as startup tokens.

The four startup tokens (corresponding to the four input vector components), after being input, are copied four at a time. They are used to read the matrix elements stored in the DM on a column by column basis, and to perform the multiplications.

The multiplication yields 33 bits of data, including a sign bit; from these, the high 16 bits and the sign bit are extracted to form a 17-bit data value. The results of the multiplication are added column-by-column to produce the output vector components.

The output vector components are produced in the order of W', X', Y', and Z', each consisting of 16 addition data bits and a sign bit for a total of 17 bits. Any bits from the addition operation in excess of 17 bits are truncated. Therefore, care should be taken in the placement of the decimal point to insure that the column-by-column additions do not produce overflows.

<Explanation of the Nodes>

FCOPY4

This node makes four copies of each of the four input vector components.

FMULW

This node multiplies component W of the input vector with the elements in Column 1 of the matrix, giving 17-bit data (including the sign bit).

FMULX

This node multiplies component X of the input vector with the elements in Column 2 of the matrix, giving 17-bit data (including the sign bit).

FMULY

This node multiplies component Y of the input vector with the elements in Column 3 of the matrix, giving 17-bit data (including the sign bit).

FMULZ

This node multiplies component Z of the input vector with the elements in Column 4 of the matrix, giving 17-bit data (including the sign bit).

FADDWX, FADDYZ, FAWXYZ

These nodes add the results of the multiplication column-by-column, giving output vector components.

FANS

This node sends the four components of the output vector to the host system in the order of W', X', Y', and Z'.

Figure 2-5
A Matrix-Vector Product Flow Graph
(17 bit x 17 bit = 17 bit)

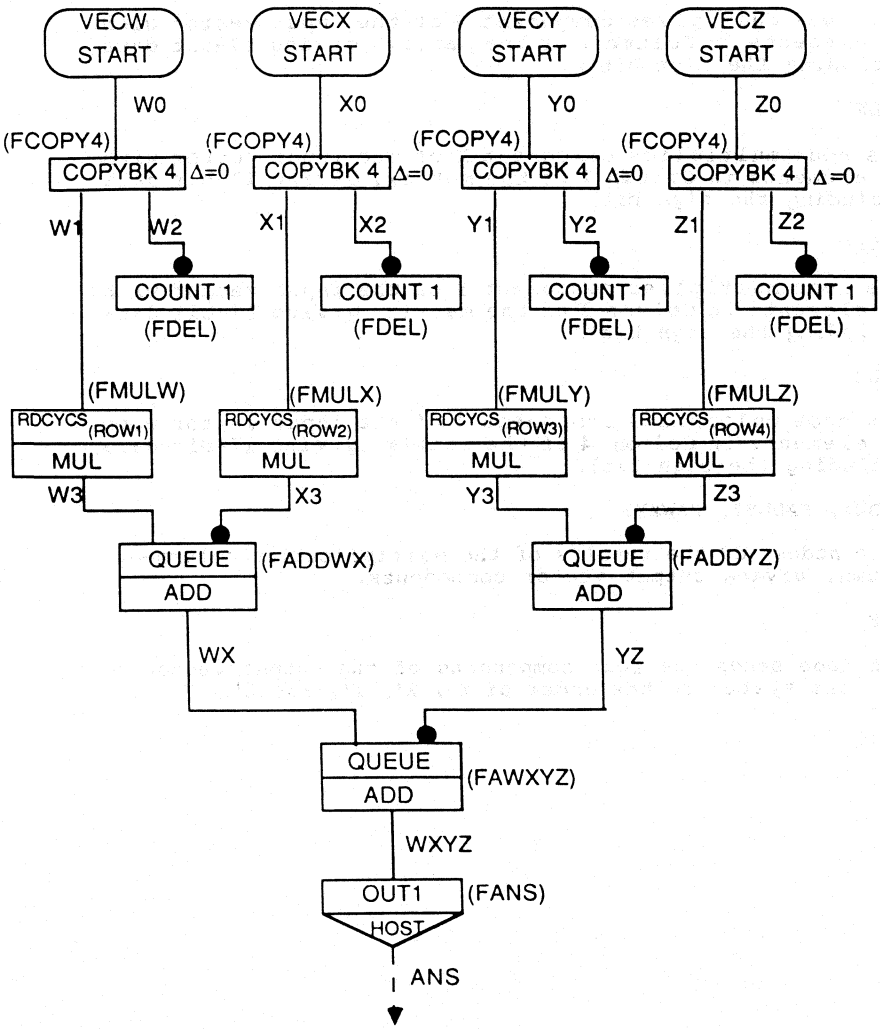


Figure 2-6
Matrix-Vector Product Source Program
(17 bit x 17 bit = 17 bit)

```
;*****************************************************************
;
;               MATRIX x VECTOR  (17 bits x 17 bits = 17 bits)
;
;*****************************************************************
;
MODULE  MAT1    =       8                                       ;
;
EQUATE  VECW    =       199AH                                   ;
EQUATE  VECX    =       3334H                                   ;
EQUATE  VECY    =       4CCDH                                   ;
EQUATE  VECZ    =       6667H                                   ;
;
EQUATE  HOST    =       0                                       ;
;
;*****************************************************************
;                       INPUT
;*****************************************************************
;
INPUT   W0,     X0,     Y0,     Z0                              ;
;
;*****************************************************************
;                       OUTPUT
;*****************************************************************
;
OUTPUT  ANS                                                     ;
;
;*****************************************************************
;                       LINK
;*****************************************************************
;
LINK    W1,     W2              = FCOPY4 (W0    )               ;
LINK    W3                      = FMULW  (W1    )               ;
LINK                            = FDEL   (  , W2)               ;
LINK    X1,     X2              = FCOPY4 (X0    )               ;
LINK    X3                      = FMULX  (X1    )               ;
LINK                            = FDEL   (  , X2)               ;
LINK    WX                      = FADDWX (W3, X3)               ;
LINK    Y1,     Y2              = FCOPY4 (Y0    )               ;
LINK    Y3                      = FMULY  (Y1    )               ;
LINK                            = FDEL   (  , Y2)               ;
LINK    Z1,     Z2              = FCOPY4 (Z0    )               ;
LINK    Z3                      = FMULZ  (Z1    )               ;
LINK                            = FDEL   (  , Z2)               ;
LINK    YZ                      = FADDYZ (Y3, Z3)               ;
LINK    WXYZ                    = FAWXYZ (WX, YZ)               ;
LINK    ANS                     = FANS   (WXYZ)                 ;
;
```

Figure 2-6 (cont.)
Matrix-Vector Product Source Program
(17 bit x 17 bit = 17 bit)

```
;*****************************************************************
;                        FUNCTION
;*****************************************************************
;
FUNCTION        FCOPY4  = COPYBK (4, 0)                          ;
FUNCTION        FMULW   = MUL    (X   ), RDCYCS (ROW1, 4)        ;
FUNCTION        FMULX   = MUL    (X   ), RDCYCS (ROW2, 4)        ;
FUNCTION        FMULY   = MUL    (X   ), RDCYCS (ROW3, 4)        ;
FUNCTION        FMULZ   = MUL    (X   ), RDCYCS (ROW4, 4)        ;
FUNCTION        FDEL    = COUNT  (1   )                          ;
FUNCTION        FADDWX  = ADD,              QUEUE (QUE1, 16)     ;
FUNCTION        FADDYZ  = ADD,              QUEUE (QUE2, 16)     ;
FUNCTION        FAWXYZ  = ADD,              QUEUE (QUE3, 16)     ;
FUNCTION        FANS    = OUT1 (HOST, 0)                         ;
;
;*****************************************************************
;                        MEMORY
;*****************************************************************
;
MEMORY  QUE1    =       AREA (16)                                ;
MEMORY  QUE2    =       AREA (16)                                ;
MEMORY  QUE3    =       AREA (16)                                ;
MEMORY  ROW1    =       10,     50,     90,     130              ;
MEMORY  ROW2    =       20,     60,     100,    140              ;
MEMORY  ROW3    =       30,     70,     110,    150              ;
MEMORY  ROW4    =       40,     80,     120,    160              ;
;
;*****************************************************************
;                        START
;*****************************************************************
;
START                                                            ;
DATA    EXEC    (MAT1,  W0,     VECW    )                       ;
DATA    EXEC    (MAT1,  X0,     VECX    )                       ;
DATA    EXEC    (MAT1,  Y0,     VECY    )                       ;
DATA    EXEC    (MAT1,  Z0,     VECZ    )                       ;
;
END                                                              ;
```

2.4 17 Bit x 17 Bit = 33 Bit

2.4.1 Algorithm

The data for matrix elements and input vector components are expressed in 17 bits, 16 bits for the numeric value and a sign bit. Although these data may have the decimal point anywhere, since the addition of the results of the multiplication is performed by adding the high words and low words column by column, the products of the multiplication must have the decimal point at the same bit position.

The output vector components are expressed in 32 numerical value bits plus a sign bit, with the decimal point set at the decimal point position of the product of the matrix element and input vector components.

The results of multiplication between a matrix element and an input vector component are expressed in 33 bits. Since the data field of a token is 17 bits (including sign bit), the results are divided into high and low words.

To add the multiplication products, a procedure for permitting the carries from the addition of low words to the high words is required. Any carries from the addition of the high words are truncated.

2.4.2 Range of Allowable Parameter Values

<Assembler-Coded Parameters>

ROW1 .... Column 1 matrix elements (4 words)
ROW2 .... Column 2 matrix elements (4 words)
ROW3 .... Column 3 matrix elements (4 words)
ROW4 .... Column 4 matrix elements (4 words)

<Startup Token-Defined Parameters>

W0 .... Input vector row 1 component
X0 .... Input vector row 2 component
Y0 .... Input vector row 3 component
Z0 .... Input vector row 4 component

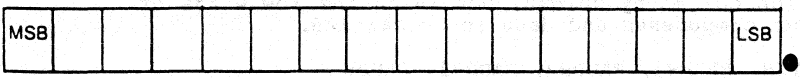
Table 2-3 indicates the allowable values of the parameters.

Table 2-3
Range of Allowable Parameter Values
(17 bit x 17 bit = 33 bit)

PARAMETER	RANGE OF ALLOWABLE VALUES	VALUE IN PROGRAM EXAMPLE
ROW1	−FFFFH ~ FFFFH	1
		5
		9
		13
ROW2	−FFFFH ~ FFFFH	2
		6
		10
		14
ROW3	−FFFFH ~ FFFFH	3
		7
		11
		15
ROW4	−FFFFH ~ FFFFH	4
		8
		12
		16
W0	−FFFFH ~ FFFFH	1
X0	−FFFFH ~ FFFFH	2
Y0	−FFFFH ~ FFFFH	3
Z0	−FFFFH ~ FFFFH	4

Note: Since a parameter can have its decimal point at any bit position, its allowable value is indicated in hexadecimal.

In the program examples referred to in Table 2-3, the decimal point is located to the right of the LSB.



2.4.3 Explanation of the Flow Graph

This program stores the matrix elements in the DM (by coding them in MEMORY statements in the source program) column by column and uses startup tokens as the source of the input vector components.

The output vector components are obtained in the W', X', Y', Z' sequence. Before being sent to the host system, these double precision data are adjusted so that their high and low word sign bits will agree.

<Explanation of the Nodes>

FCOPY4

This node makes four copies each of the four input vector components.

FMULW

This node multiplies the W component of the input vector with the Column 1 matrix elements, obtaining the results as separate high and low words.

FMULX

This node multiplies the X component of the input vector with the Column 2 matrix elements, obtaining the results as separate high and low words.

FMULY

This node multiplies the Y component of the input vector with the Column 3 matrix elements, obtaining the results as separate high and low words.

FMULZ

This node multiplies the Z component of the input vector with the Column 4 matrix elements, obtaining the results as separate high and low words.

FADWXL, FADYZL, FADDL

These nodes add the low words of the results of the multiplication column by column, obtaining the low words of output vector components and data to be carried.

FADWXH, FADYZH, FADDH1, FADDH2, FADDH3, FADDH

These nodes add the high words of the results of the multiplication and the carry data of the low words, obtaining the high words of output vector components.

FADJL

This node adjusts the high and low words of the output vector components so that the words will have the same sign.

Figure 2-7
A Matrix-Vector Multiplication Flow Graph
(17 bit x 17 bit = 33 bit)

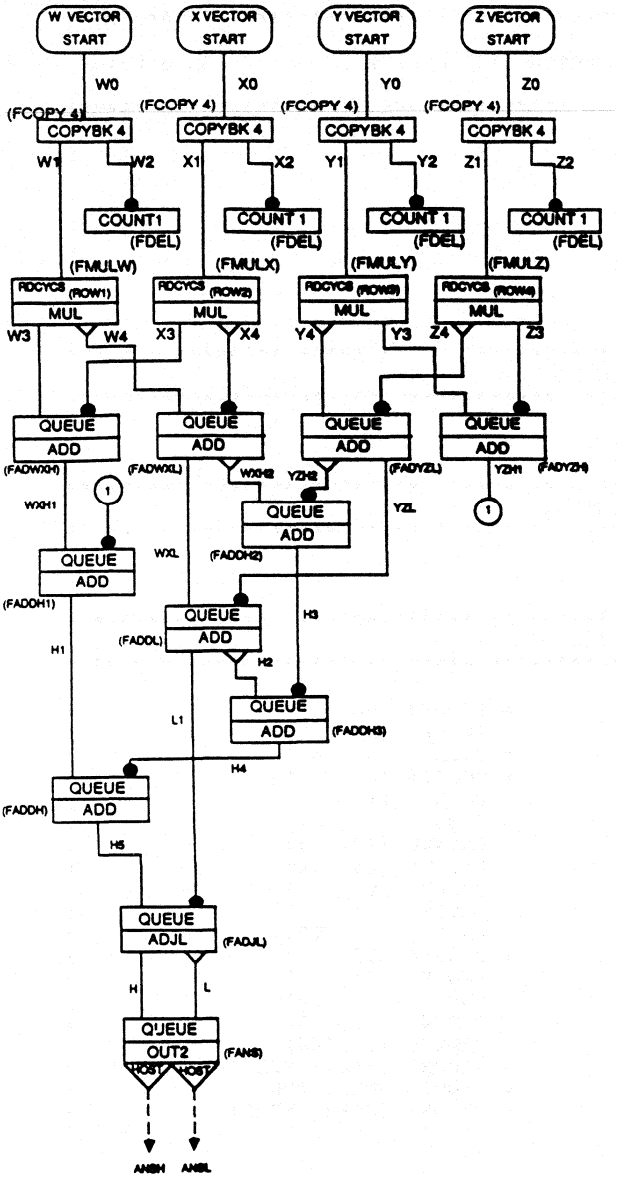


Figure 2-8
Matrix-Vector Multiplication Source Program
(17 bit x 17 bit = 33 bit)

```
;*****************************************************************
;
;              MATRIX x VECTOR  (17 bits x 17 bits = 33 bits)
;
;*****************************************************************
;
MODULE  MAT2    =       8                                       ;
;
EQUATE  VECW    =       1                                       ;
EQUATE  VECX    =       2                                       ;
EQUATE  VECY    =       3                                       ;
EQUATE  VECZ    =       4                                       ;
;
EQUATE  HOST    =       0                                       ;
;
;*****************************************************************
;                       INPUT
;*****************************************************************
;
INPUT   W0,     X0,     Y0,     Z0                              ;
;
;*****************************************************************
;                       OUTPUT
;*****************************************************************
;
OUTPUT  ANSH,   ANSL                                            ;
;
;*****************************************************************
;                       LINK
;*****************************************************************
;
LINK    W1,     W2              = FCOPY4 (W0    )               ;
LINK    W3,     W4              = FMULW  (W1    )               ;
LINK                            = FDEL   (  , W2)               ;
LINK    X1,     X2              = FCOPY4 (X0    )               ;
LINK    X3,     X4              = FMULX  (X1    )               ;
LINK                            = FDEL   (  , X2)               ;
LINK    WXH1                    = FADWXH (W3, X3)               ;
LINK    WXL,    WXH2            = FADWXL (W4, X4)               ;
LINK    Y1,     Y2              = FCOPY4 (Y0    )               ;
LINK    Y3,     Y4              = FMULY  (Y1    )               ;
LINK                            = FDEL   (  , Y2)               ;
LINK    Z1,     Z2              = FCOPY4 (Z0    )               ;
LINK    Z3,     Z4              = FMULZ  (Z1    )               ;
LINK                            = FDEL   (  , Z2)               ;
LINK    YZH1                    = FADYZH (Y3, Z3)               ;
LINK    YZL,    YZH2            = FADYZL (Y4, Z4)               ;
LINK    H1                      = FADDH1 (WXH1, YZH1)           ;
LINK    L1,     H2              = FADDL  (WXL, YZL)             ;
LINK    H3                      = FADDH2 (WXH2, YZH2)           ;
```

Figure 2-8 (cont.)
Matrix-Vector Multiplication Source Program
(17 bit x 17 bit = 33 bit)

```
LINK    H4                      = FADDH3 (H2, H3)                ;
LINK    H5                      = FADDH  (H1, H4)                ;
LINK    H,      L               = FADJL  (H5, L1)                ;
LINK    ANSH,   ANSL            = FANS   (H,  L )                ;
;
;*****************************************************************
;                       FUNCTION
;*****************************************************************
;
FUNCTION        FCOPY4  = COPYBK (4, 0)                          ;
FUNCTION        FMULW   = MUL    (XY  ), RDCYCS (ROW1, 4)        ;
FUNCTION        FMULX   = MUL    (XY  ), RDCYCS (ROW2, 4)        ;
FUNCTION        FMULY   = MUL    (XY  ), RDCYCS (ROW3, 4)        ;
FUNCTION        FMULZ   = MUL    (XY  ), RDCYCS (ROW4, 4)        ;
FUNCTION        FDEL    = COUNT  (1   )                          ;
FUNCTION        FADWXH  = ADD,             QUEUE (QUE1, 16)      ;
FUNCTION        FADWXL  = ADD  (XY      ), QUEUE (QUE2, 16)      ;
FUNCTION        FADYZH  = ADD,             QUEUE (QUE3, 16)      ;
FUNCTION        FADYZL  = ADD  (XY      ), QUEUE (QUE4, 16)      ;
FUNCTION        FADDH1  = ADD,             QUEUE (QUE5, 16)      ;
FUNCTION        FADDH2  = ADD,             QUEUE (QUE6, 16)      ;
FUNCTION        FADDH3  = ADD,             QUEUE (QUE7, 16)      ;
FUNCTION        FADDH   = ADD,             QUEUE (QUE8, 16)      ;
FUNCTION        FADDL   = ADD  (XY      ), QUEUE (QUE9, 16)      ;
FUNCTION        FADJL   = ADJL (XY      ), QUEUE (QUE10,16)      ;
FUNCTION        FANS    = OUT2 (HOST,0,0), QUEUE (QUEA, 16)      ;
;
;*****************************************************************
;                       MEMORY
;*****************************************************************
;
MEMORY  QUE1    =       AREA (16)                                ;
MEMORY  QUE2    =       AREA (16)                                ;
MEMORY  QUE3    =       AREA (16)                                ;
MEMORY  QUE4    =       AREA (16)                                ;
MEMORY  QUE5    =       AREA (16)                                ;
MEMORY  QUE6    =       AREA (16)                                ;
MEMORY  QUE7    =       AREA (16)                                ;
MEMORY  QUE8    =       AREA (16)                                ;
MEMORY  QUE9    =       AREA (16)                                ;
MEMORY  QUE10   =       AREA (16)                                ;
MEMORY  QUEA    =       AREA (16)                                ;
MEMORY  ROW1    =       1,      5,      9,      13               ;
MEMORY  ROW2    =       2,      6,      10,     14               ;
MEMORY  ROW3    =       3,      7,      11,     15               ;
MEMORY  ROW4    =       4,      8,      12,     16               ;
;
```

Figure 2-8 (cont.)
Matrix-Vector Multiplication Source Program
(17 bit x 17 bit = 33 bit)

```
;*****************************************************************
;                          START
;*****************************************************************
;
START                                                              ;
DATA    EXEC    (MAT2,  W0,     VECW    )                        ;
DATA    EXEC    (MAT2,  X0,     VECX    )                        ;
DATA    EXEC    (MAT2,  Y0,     VECY    )                        ;
DATA    EXEC    (MAT2,  Z0,     VECZ    )                        ;
;
END                                                                ;
```

2.5 33 Bit x 33 Bit = 33 Bit

2.5.1 Algorithm

In this computation, the matrix elements and the input vector components data are treated as 33 bits, consisting of 32 numeric value bits and a sign bit, with the decimal point occurring at an arbitrary position. The output vector components resulting from these data have the same organization.

When a numerical value involved is 17 bits or more in length, the data must be grouped into 16-bit units starting from the LSB. Thus, the matrix elements and input vector components illustrated in Figure 2-9 are divided into high and low words before a multiplication is performed.

The results of matrix element and vector component multiplications are obtained as the higher 32 bits out of 64-bit numerical values, to be divided into high and low words and added together column by column.

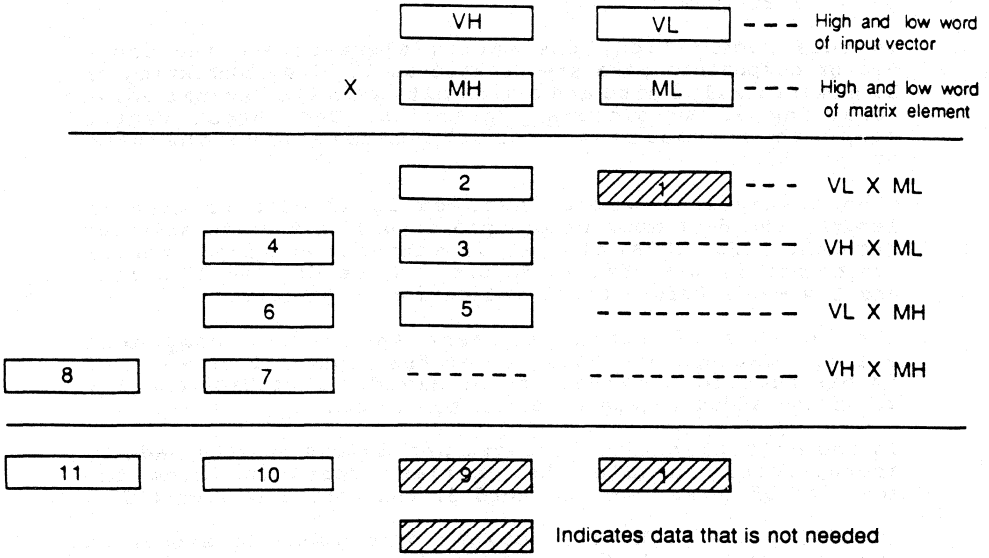
In the multiplication process, provisions must be made to appropriately handle the carries resulting from the addition of terms Word 2, Word 3, and Word 5 in Figure 2-9.

In adding the high words column by column, if either an overflow or an underflow occurs the carries are truncated. If the same type of problem occurs in the addition of low words, the carries are added to the high word.

The output vector components thus obtained consist of 2 words each, column-wise additions of high and low words. Each word of data is expressed in 17 bits, 16 numeric value bits and a sign bit.

In setting up matrix element and vector component data, it is advisable to place the values near the MSB of the word, and ensure that all data values have the decimal point at the same bit position.

Figure 2-9
Multiplication of Matrix Elements and an Input Vector
(33 bit x 33 bit = 33 bit)



2.5.2 Range of Allowable Parameter Values

<Assembler-Coded Parameters>

ROWTH .... Sequentially stored matrix high word elements
ROWTL .... Sequentially stored matrix low word elements

<Startup Token-Defined Parameters>

H0 .... Input vector high word components
L0 .... Input vector low word components

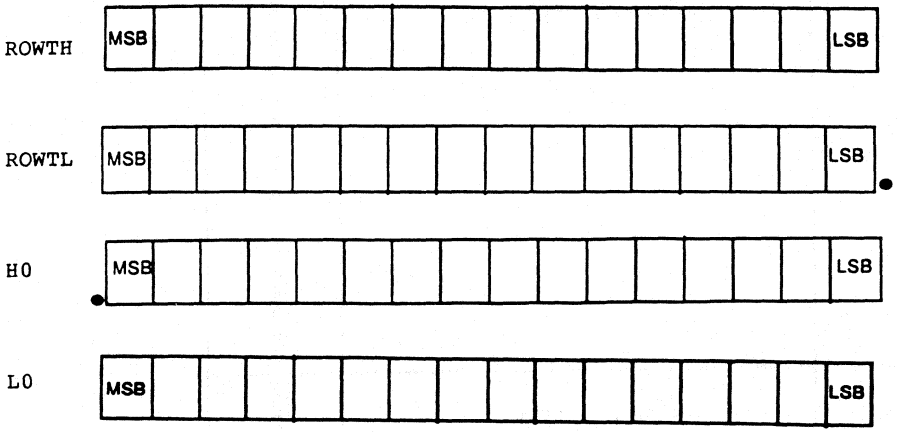
Table 2-4 shows the allowable values of these parameters.

Table 2-4
Range of Allowable Parameter Values
(33 bit x 33 bit = 33 bit)

PARAMETER	RANGE OF ALLOWABLE VALUES	VALUE IN PROGRAM EXAMPLE
ROWTH	−FFFFH ~ FFFFH	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ROWTL	−FFFFH ~ FFFFH	10 50 90 130 20 60 100 140 30 70 110 150 40 80 120 160
H0	−FFFFH ~ FFFFH	1999H 3333H 4CCCH 6666H
L0	−FFFFH ~ FFFFH	999AH 3334H CCCDH 6667H

The parameters ROWTH, ROWTL, H0, and L0 shown in Table 2-4 are related to each other in high word/low word relations. In the program examples referred to, these parameters have the decimal point at the following positions:

Figure 2-10
Matrix-Vector Multiplication (33 bit x 33 bit = 33 bit)
Example Program Decimal Point Positions



Input Vector Values in Example Program

	H0	L0
W = 0.1 =	1999	999AH
X = 0.2 =	3333	3334H
Y = 0.3 =	4CCC	CCCDH
Z = 0.4 =	6666	6667H

2.5.3 The Flow Graph Explained

Since the matrix elements and input vector components handled in this program are 33 bits in length, they are divided into high and low words. The matrix elements are stored in the DM sequentially as high and low words. The input vectors are treated as startup tokens, each representing either a high or a low word.

Since the input arcs of the startup tokens are used for inputting all the high and low words of the vector components, the program requires that they be entered in the order of W, X, Y, and Z. Additional nodes would be needed if the input vector components were allowed to be input in

an arbitary order.

The output vector components obtained are adjusted so that the sign of the high and low words are the same. The output vector decimal point will be at the same bit position as the vector-matrix product, and will be sent to the host system in the order of high and low words.

Since the allocated DM storage areas LINLB and LINHB are initialized in a MEMORY statement, the program can only be run once without reloading the object code. If it is desired to run the program multiple times, then additional nodes can be added to set the eight locations in the two storage areas to '0', at the start or finish of each program execution.

<Explanations of the Nodes>

FCPYM2, FCPYB4

These nodes made four copies each of the high and low words of the 4-component input vector.

FMULH1

This node multiplies the matrix element high words, previously stored sequentially, with the high words of input vector components, each of which has been copied into four duplicates. The results of the multiplications are obtained in terms of high and low words (Word 8 and Word 7 in Figure 2-9).

FMULH2

This node multiplies the matrix element high words, previously stored sequentially, with the low words of input vector components, each of which has been copied into four duplicates. The results of the multiplications are obtained in terms of high and low words (Word 6 and Word 5 in Figure 2-9).

FMULL1

This node multiplies the matrix element low words, previously stored sequentially, with the high words of input vector components, each of which has been copied into 4 duplicates. The results of the multiplications are obtained in terms of high and low words (Word 4 and Word 3 in Figure 2-9).

FMULL2

This node multiplies the matrix element's low words, previously stored sequentially, with the low words of input vector components, each of which has been copied into four

duplicates. The results of the multiplications are obtained in terms of high words (Word 2 in Figure 2-9).

FADDL1, FADDL2, FADDL3

These nodes add the multiplication data (Word 2, Word 3, and Word 5 in Figure 2-9), corresponding to the second word (Word 9) of the resultant value, and determine the carries.

FADDH1, FADDH3, FADDL3, FADDL4

These nodes add the multiplication data (Word 4, Word 6, and Word 7 in Figure 2-9) corresponding to the third word of the resultant value, and determine the value of the third word (Word 10 in Figure 2-9) and the carries.

FADDH2, FADDH4, FADDH5

These nodes determine the fourth word (Word 11 in Figure 2-9) of the resultant value.

FADLBL, FWRLBL

These nodes add the fourth words and the carries data from the results of the multiplications, column by column, and save the results in the DM.

FCPYB4, FRDLBH

These nodes read the third word of each of the summation data (the low words of the output vectors) which have been saved in the DM.

FCPYB4, FRDLBL

These nodes read the fourth word of each of the summation data (the high words of the output vectors) which have been saved in the DM.

FADJL

This node adjusts the precision of the high and low words of each of the output vector components, so that the words will have the same sign.

FANS

This node sends each of the output vector components in the high word-low word sequence to the host system.

Figure 2-11
Matrix-Vector Multiplication Flow Graph
(33 bit x 33 bit = 33 bit)

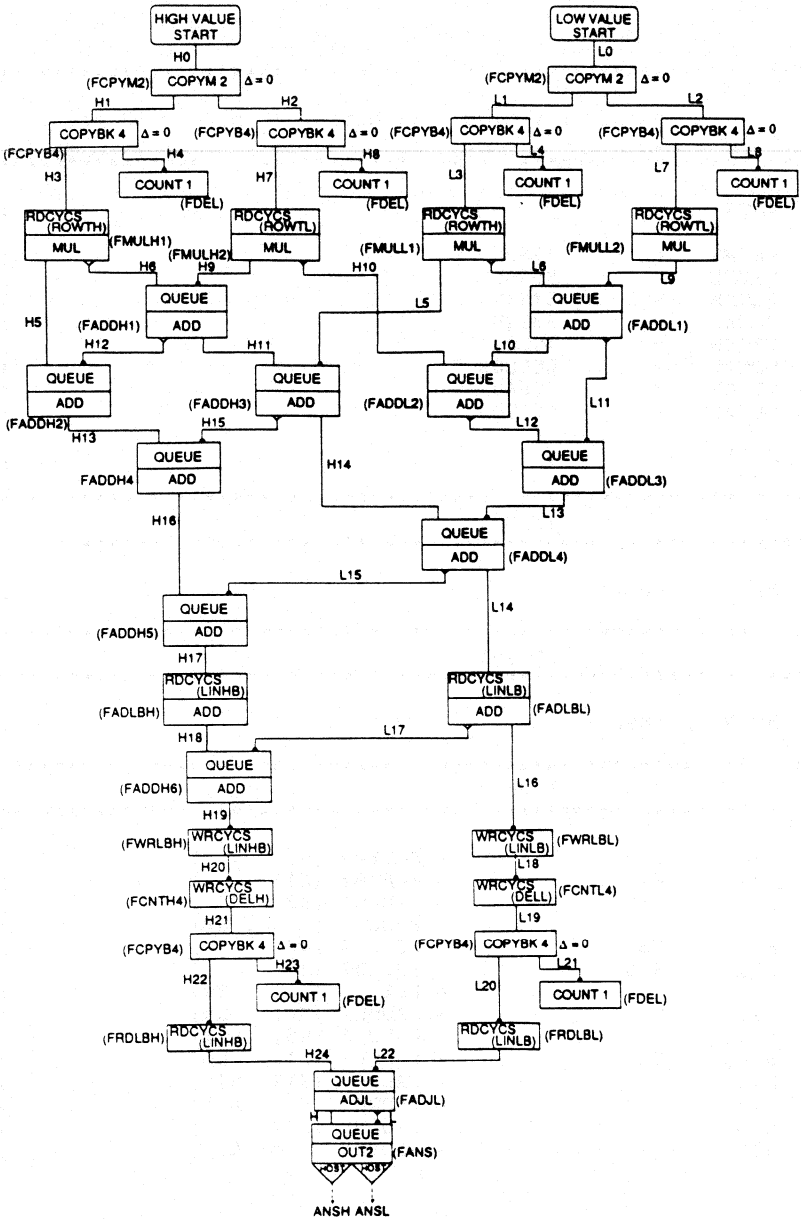


Figure 2-12
Matrix-Vector Multiplication Source Program
(33 bit x 33 bit = 33 bit)

```
;*****************************************************************
;
;               MATRIX x VECTOR  (33 bits x 33 bits = 33 bits)
;
;*****************************************************************
;
MODULE  MAT3    =       8                                       ;
;
EQUATE  VECWH   =       1999H                                   ;
EQUATE  VECWL   =       999AH                                   ;
EQUATE  VECXH   =       3333H                                   ;
EQUATE  VECXL   =       3334H                                   ;
EQUATE  VECYH   =       4CCCH                                   ;
EQUATE  VECYL   =       0CCCDH                                  ;
EQUATE  VECZH   =       6666H                                   ;
EQUATE  VECZL   =       6667H                                   ;
;
EQUATE  HOST    =       0                                       ;
;
;*****************************************************************
;                       INPUT
;*****************************************************************
;
INPUT   H0,     L0                                              ;
;
;*****************************************************************
;                       OUTPUT
;*****************************************************************
;
OUTPUT  ANSH,   ANSL                                            ;
;
;*****************************************************************
;                       LINK
;*****************************************************************
;
LINK    H1,     H2              = FCPYM2 (H0      )             ;
LINK    H3,     H4              = FCPYB4 (H1      )             ;
LINK    H5,     H6              = FMULH1 (H3      )             ;
LINK                            = FDEL   (    , H4)             ;
LINK    H7,     H8              = FCPYB4 (H2      )             ;
LINK    H9,     H10             = FMULH2 (H7      )             ;
LINK                            = FDEL   (    , H8)             ;
LINK    H11,    H12             = FADDH1 (H6,   H9)             ;
LINK    H13                     = FADDH2 (H5,  H12)             ;
LINK    H14,    H15             = FADDH3 (H11,  L5)             ;
LINK    H16                     = FADDH4 (H13, H15)             ;
LINK    H17                     = FADDH5 (H16, L15)             ;
LINK    H18                     = FADLBH (H17     )             ;
LINK    H19                     = FADDH6 (H18, L17)             ;
```

Figure 2-12 (cont.)
Matrix-Vector Multiplication Source Program
(33 bit x 33 bit = 33 bit)

```
LINK    H20                     = FWRLBH (    , H19)          ;
LINK    H21                     = FCNTH4 (    , H20)          ;
LINK    H22,    H23             = FCPYB4 (H21      )          ;
LINK    H24                     = FRDLBH (H22      )          ;
LINK                            = FDEL   (    , H23)          ;
LINK    L1,     L2              = FCPYM2 (L0       )          ;
LINK    L3,     L4              = FCPYB4 (L1       )          ;
LINK    L5,     L6              = FMULL1 (L3       )          ;
LINK                            = FDEL   (    , L4 )          ;
LINK    L7,     L8              = FCPYB4 (L2       )          ;
LINK    L9                      = FMULL2 (L7       )          ;
LINK                            = FDEL   (    , L8 )          ;
LINK    L10,    L11             = FADDL1 (L6 , L9 )           ;
LINK    L12                     = FADDL2 (H10, L10)           ;
LINK    L13                     = FADDL3 (L12, L11)           ;
LINK    L14,    L15             = FADDL4 (H14, L13)           ;
LINK    L16,    L17             = FADLBL (L14      )          ;
LINK    L18                     = FWRLBL (    , L16)          ;
LINK    L19                     = FCNTL4 (    , L18)          ;
LINK    L20,    L21             = FCPYB4 (L19      )          ;
LINK    L22                     = FRDLBL (L20      )          ;
LINK                            = FDEL   (    , L21)          ;
LINK    H,      L               = FADJL  (H24, L22)           ;
LINK    ANSH,   ANSL            = FANS   (H,   L  )           ;
;
;*****************************************************************
;                       FUNCTION
;*****************************************************************
;
FUNCTION        FCPYM2  = COPYM  (2,     0)                     ;
FUNCTION        FCPYB4  = COPYBK (4,     0)                     ;
FUNCTION        FMULH1  = MUL    (XY      ), RDCYCS (ROWTH, 16) ;
FUNCTION        FMULH2  = MUL    (XY      ), RDCYCS (ROWTL, 16) ;
FUNCTION        FMULL1  = MUL    (XY      ), RDCYCS (ROWTH, 16) ;
FUNCTION        FMULL2  = MUL,               RDCYCS (ROWTL, 16) ;
FUNCTION        FDEL    = COUNT  ( 1      )                     ;
FUNCTION        FADDH1  = ADD    (XY      ), QUEUE  (QUE1,  16) ;
FUNCTION        FADDH2  = ADD,               QUEUE  (QUE2,  16) ;
FUNCTION        FADDH3  = ADD    (XY      ), QUEUE  (QUE3,  16) ;
FUNCTION        FADDH4  = ADD,               QUEUE  (QUE4,  16) ;
FUNCTION        FADDH5  = ADD,               QUEUE  (QUE5,  16) ;
FUNCTION        FADDH6  = ADD,               QUEUE  (QUE6,  16) ;
FUNCTION        FADDL1  = ADD    (XY      ), QUEUE  (QUE7,  16) ;
FUNCTION        FADDL2  = ADD    ( Y      ), QUEUE  (QUE8,  16) ;
FUNCTION        FADDL3  = ADD,               QUEUE  (QUE9,  16) ;
FUNCTION        FADDL4  = ADD    (XY      ), QUEUE  (QUE10, 16) ;
FUNCTION        FADLBH  = ADD,               RDCYCS (LINHB,  4) ;
FUNCTION        FADLBL  = ADD    (XY      ), RDCYCS (LINLB,  4) ;
FUNCTION        FRDLBH  = RDCYCS (LINHB, 4)                     ;
FUNCTION        FRDLBL  = RDCYCS (LINLB, 4)                     ;
```

Figure 2-12 (cont.)
Matrix-Vector Multiplication Source Program
(33 bit x 33 bit = 33 bit)

```
FUNCTION        FWRLBH  = WRCYCS (LINHB, 4)                              ;
FUNCTION        FWRLBL  = WRCYCS (LINLB, 4)                              ;
FUNCTION        FCNTH4  = WRCYCS (DELH,  4)                              ;
FUNCTION        FCNTL4  = WRCYCS (DELL,  4)                              ;
FUNCTION        FADJL   = ADJL   (XY      ), QUEUE   (QUEA, 16)          ;
FUNCTION        FANS    = OUT2 (HOST, 0, 0), QUEUE   (QUEE, 16)        ;
;
;*****************************************************************
;                       MEMORY
;*****************************************************************
;
MEMORY  DELH    =       AREA ( 4)                                        ;
MEMORY  DELL    =       AREA ( 4)                                        ;
MEMORY  LINHB   =       0,      0,      0,      0                        ;
MEMORY  LINLB   =       0,      0,      0,      0                        ;
MEMORY  QUE1    =       AREA (16)                                        ;
MEMORY  QUE2    =       AREA (16)                                        ;
MEMORY  QUE3    =       AREA (16)                                        ;
MEMORY  QUE4    =       AREA (16)                                        ;
MEMORY  QUE5    =       AREA (16)                                        ;
MEMORY  QUE6    =       AREA (16)                                        ;
MEMORY  QUE7    =       AREA (16)                                        ;
MEMORY  QUE8    =       AREA (16)                                        ;
MEMORY  QUE9    =       AREA (16)                                        ;
MEMORY  QUE10   =       AREA (16)                                        ;
MEMORY  QUEA    =       AREA (16)                                        ;
MEMORY  QUEE    =       AREA (16)                                        ;
MEMORY  ROWTH   =       0,      0,      0,      0,
                        0,      0,      0,      0,
                        0,      0,      0,      0,
                        0,      0,      0,      0                        ;
MEMORY  ROWTL   =       10,     50,     90,     130,
                        20,     60,     100,    140,
                        30,     70,     110,    150,
                        40,     80,     120,    160                      ;
;
;*****************************************************************
;                       START
;*****************************************************************
;
START                                                                    ;
DATA    EXEC    (MAT3,  H0,     VECWH   )                               ;
DATA    EXEC    (MAT3,  L0,     VECWL   )                               ;
DATA    EXEC    (MAT3,  H0,     VECXH   )                               ;
DATA    EXEC    (MAT3,  L0,     VECXL   )                               ;
DATA    EXEC    (MAT3,  H0,     VECYH   )                               ;
DATA    EXEC    (MAT3,  L0,     VECYL   )                               ;
DATA    EXEC    (MAT3,  H0,     VECZH   )                               ;
DATA    EXEC    (MAT3,  L0,     VECZL   )                               ;
END                                                                      ;
```

Chapter 3

Evaluation of Polynomials

3.1 Processing Explained

The polynomials treated in this program are expressed by Equation 3-1:

$$\sum_{i=0}^{n} a_i x^i \qquad \text{(Eq. 3-1)}$$

where the variable x has an absolute value less than 1.0. The decimal point is at the right of the MSB. The data lengths represented in these programs are shown in Table 3-1.

The polynomial expression described above can be used in the computation of sine, cosine and other numerical values.

Table 3-1
Polynomial Programs Data Lengths

CONSTANT a_i	VARIABLE x	COMPUTATION RESULT
17 BITS	17 BITS	17 BITS
33 BITS	33 BITS	33 BITS

(INCLUDES SIGN BIT)

3.2 Algorithm

Equation 3-1 can be expanded as follows:

$$\begin{aligned}\sum_{i=0}^{n} a_i x^i &= a_0 x^0 + a_1 x^1 + a_2 x^2 + \cdots\cdots + a_n x^n \\ &= a_0 + a_1 x^1 + a_2 x^2 + \cdots\cdots + a_n x^n \\ &= a_0 + x(a_1 + a_2 x^1 + \cdots\cdots + a_n x^{n-1}) \\ &= a_0 + x(a_1 + x(a_2 + \cdots\cdots x(a_{n-1} + x a_n) \cdots\cdots))\end{aligned}$$

According to Equation 3-2, a polynomial can be evaluated by multiplying the variable x with the constant a_n, followed by an addition of the resulting product to the constant a_{n-1}. The result of the addition is multiplied by the variable x and added to the constant a_{n-2}, and the process is repeated.

3.3 17 Bit x 17 Bit = 17 Bit

3.3.1 Algorithm

In this computational procedure, 17-bit results are obtained through iterations of the multiplication of the 17-bit data variable x with the constants a_i and the addition of the products.

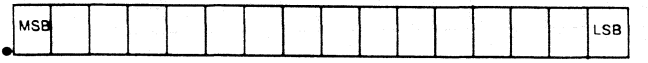
The variable x and the last-term constant a_n are multiplied to yield a 33-bit value, from which the higher 17 bits are added to the constant a_{n-1}. The result of the addition is represented in 17 bits.

Using the result of the addition, the multiplication of the sum by variable x, and the addition of the product to the constant a_i are repeated as many times as necessary.

To perform these multiplications and additions iteratively, the decimal point must be at a fixed position. Also, to permit addition to the constant a_i, the product of multiplication with the variable x must have the same number of bits in its integer part and the same decimal point position as does the constant a_i.

Therefore, in processing the input data the decimal point of the variable x is fixed at the MSB and that of the constant a_i is set at an arbitrary position, as long as it is the same for all data values.

The decimal point of variable x is located as follows:



3.3.2 Range of Allowable Parameter Values

<Assembler-Coded Parameters>

DATAA .... Constant a_n for the last term of the polynomial
ATBL .... Constant a_i for all terms except for the last term ($i = 0, 1, \ldots, n-1$)
I .... The number of ATBL data values

<Startup Token-Defined Parameters>

M0 .... Variable x

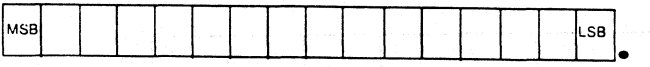
Table 3-2 shows the allowable parameter values.

Table 3-2
Allowable Parameter Values
(17 bit x 17 bit = 17 bit)

PARAMETER	RANGE OF ALLOWABLE VALUES	VALUE IN PROGRAM EXAMPLE
DATAA	−FFFFH ~ FFFFH	1000H
ATBL	−FFFFH ~ FFFFH	10H 20H 40H 80H 100H 200H
I	1 ~ 16	6
M0	−FFFFH ~ FFFFH (−0.999985 ~ 0.999985)	8000H (0.5)

Note: The allowable ranges for DATAA and ATBL, in which the decimal point can be at an arbitrary position, are shown in hexadecimal.

In the program examples referred to in Table 3-2, DATAA and ATBL have their decimal points located at the right of the LSB.



3.3.3 Explanations of the Flow Graph

In this program, the constants a_i are stored in the DM in two groups, one consisting of a_0 - a_{n-1} and the other a_n. The constants a_0 - a_{n-1} are stored in the a_{n-1}, a_{n-2}, ..., a_1, a_0 sequence. The variable x is treated as a startup token.

The startup token (variable x) which has been input from the

host system is saved in DM (DATAX) so that it is available for subsequent multiplications.

After the variable x has been saved, the last-term constant a_n of the polynomial is read and multiplied with the variable x. The high word of the result of the multiplication is added to a_{n-1}, which is read from the data table (ATBL) containing the constants a_i.
The variable x, which has been saved, is read and multiplied with the result of the addition. The high word of the product of the multiplication is added to the next constant a_i which is read, and the process is repeated as many times as necessary.

<Explanations of the Nodes>

FWRTX

This node saves the variable x in the DM.

FMULA

This node reads the last-term constant a_n of the polynomial, multiplies it with the variable x, and extracts the high word from the product.

FADDA

This node reads the constants a_{n-1} through a_0, stored in the DM, in order and adds them to the high word of the result of the multiplication in FMULA or FMULX.

FMULX

This node reads the variable x which has been saved, multiplies it with the summation data, and extracts the high word from the product.

FPICK

This node terminates the loop involving the multiplication of the variable x and addition of the results to the constants a_i.

Figure 3-1
Flow Graph for Polynomial Evaluation
(17 bit x 17 bit = 17bit)

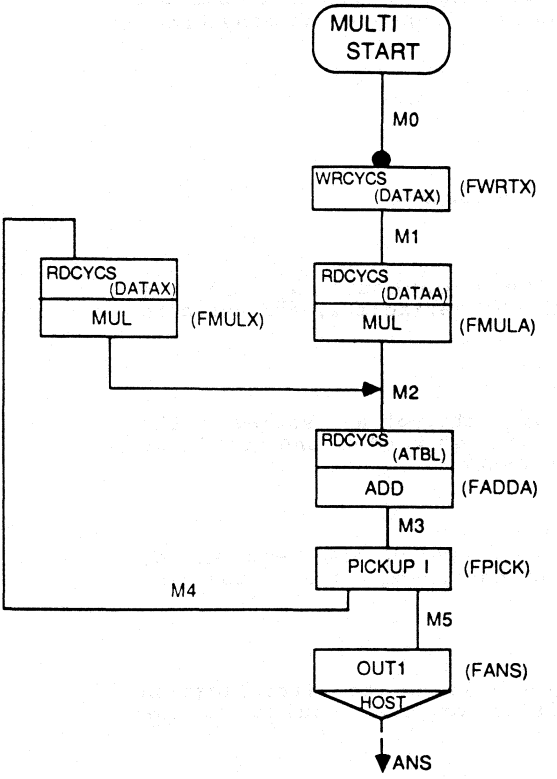


Figure 3-2
Polynomial Evaluation Source Program
(17 bit x 17 bit = 17 bit)

```
;*****************************************************************
;
;               MULTI-TERM EXPRESSION (17 bit X 17 bit = 17 bit)
;
;*****************************************************************
;
MODULE  MUL1    =       8                                       ;
;
EQUATE  VALX    =       8000H                                   ;
EQUATE  I       =       6                                       ;
;
EQUATE  HOST    =       0                                       ;
;
;*****************************************************************
;               INPUT & OUTPUT
;*****************************************************************
;
INPUT   M0                                                      ;
;
OUTPUT  ANS                                                     ;
;
;*****************************************************************
;               LINK
;*****************************************************************
;
LINK    M1                      = FWRTX  (     ,  M0)           ;
LINK    M2                      = FMULA  (M1      )             ;
LINK    M3                      = FADDA  (M2      )             ;
LINK    M4,     M5              = FPICK  (M3      )             ;
LINK    M2                      = FMULX  (M4      )             ;
LINK    ANS                     = FANS   (M5      )             ;
;
;*****************************************************************
;               FUNCTION
;*****************************************************************
;
FUNCTION        FWRTX   = WRCYCS (DATAX, 1)                     ;
FUNCTION        FMULA   = MUL,              RDCYCS (DATAA, 1)   ;
FUNCTION        FMULX   = MUL,              RDCYCS (DATAX, 1)   ;
FUNCTION        FADDA   = ADD,              RDCYCS (ATBL,  I)   ;
FUNCTION        FPICK   = PICKUP (I      )                      ;
FUNCTION        FANS    = OUT1   (HOST,  0)                     ;
;
;*****************************************************************
;               MEMORY
;*****************************************************************
;
MEMORY  ATBL    =       10H, 20H, 40H, 80H, 100H, 200H          ;
MEMORY  DATAA   =       1000H                                   ;
MEMORY  DATAX   =       AREA (1)                                ;
```

Figure 3-2 (cont.)
Polynomial Evaluation Source Program
(17 bit x 17 bit = 17 bit)

```
;*****************************************************************
;                START
;*****************************************************************
;
START                                                            ;
DATA    EXEC    (MUL1,  M0,     VALX)                          ;
;
END                                                              ;
```

3.4 33 Bit x 33 Bit = 33 Bit

3.4.1 Algorithm

This algorithm is identical to the algorithm described in section 3.3 except that the data handled are in 33 bits rather than 17 bits. The following explanations concern the part of the algorithm that deals with 33 bit data.

Since the results of the multiplication with the variable x are added to the constants a_i, those results as well as the constants a_i must have their decimal point at the same bit position.

In what follows, the decimal point of the variable x is fixed at the left of the MSB, while that of the constants a_i can be arbitrary as long as all constant a_i data have the decimal point at the same position.

Variable x and constants a_i are divided into high and low words in the input data stream.

As shown in Figure 3-3, the multiplication with the variable x is performed on a word by word basis and the results are expressed in high and low words. The results of the multiplication are the higher two out of four words, and the higher words are added to the constant a_i, with the lower words truncated.

For the addition operation, the addition of carries from one word to the next higher word, except from the highest word, must be provided.

The decimal point position of the variable X is as shown:

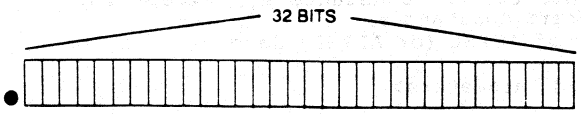
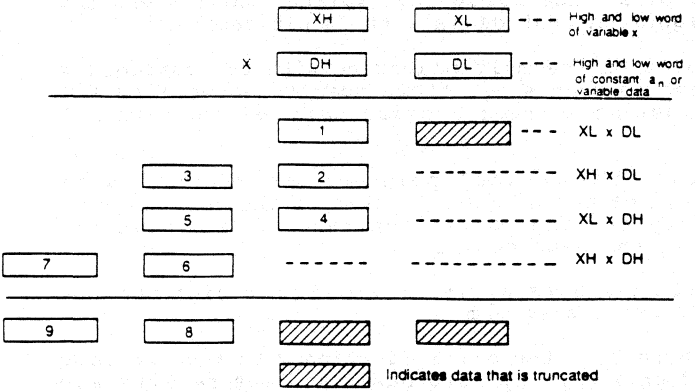


Figure 3-3
Multiplication of Variable x with
Last-Term Constant a_n or Summation Data



<Assembler-Coded Parameters>

DATAH .... High-word data for last-term constant a_n
DATAL .... Low-word data for last-term constant a_n
AHTBL .... High-word data for the constants a_i, except for the last-term constant
ALTBL .... Low-word data for the constants a_i, except for the last-term constant a_n
I .... The number of AHTBL (or ALTBL) data

<Startup Token-Defined Parameters>

H0 .... High-word data for the variable x
L0 .... Low-word data for the variable x

Table 3-3 shows allowable values of the parameters.

Table 3-3
Allowable Range of Parameter Values
(33 bit x 33 bit = 33 bit)

PARAMETER	RANGE OF ALLOWABLE VALUES	VALUE IN PROGRAM EXAMPLE
DATAH	−FFFFH ~ FFFFH	0000H
DATAL	−FFFFH ~ FFFFH	0004H
AHTBL	−FFFFH ~ FFFFH	0000H 0000H −002DH 0555H −4000H 8000H
ALTBL	−FFFFH ~ FFFFH	024FH D00DH 82D8H 5558H 0000H 0000H
I	1 − 16	6
H0	−FFFFH ~ FFFFH (−0.999985 ~ 0.999985)	4000H
L0	−FFFFH ~ FFFFH (−0.999985 ~ 0.999985)	0000H

Note: Since DATAH, DATAL, AHTBL, and ALTBL can have the decimal point at any bit position, their allowable parameter value ranges are shown in hexadecimal. See section 3.5 for values used in the program examples referred to in Table 3-3.

3.4.3 Explanation of the Flow Graph

In this program each of the constants a_i and variable x are divided into high 17 bits and low 17 bits. The high and low words of the variable x are used in startup tokens. Each of the constants a_i are divided four ways and stored in the DM: high and low words, each of which is divided into a_0 - a_{n-1} and a_n groups.

The results of the multiplication are sent to the host system in the high word-low word sequence.

<Explanations of the Nodes>

FWRTXH

This node saves the high word of the variable x in the DM (DATXH), for use later in the multiplication with the sum of the high words of the constants a_i.

FWRTXL

This node saves the low word of the variable x in the DM (DATXL), for use later in the multiplication with the sum of the low words of the constants a_i.

FCPYM2

This node makes two copies each of the high and low words of the variable x and the high and low words of the sum of the constants a_i.

FMULAH

This node reads the high word of the last-term constant a_n, multiplies it with the high and low words of the variable x, producing 2-word multiplication data from each of the operations (Word 4, Word 5, Word 6, and Word 7 in Figure 3-3).

FMLHAL

This node reads the low word of the last-term constant a_n and multiplies it with the high word of the variable x, giving 2-word multiplication data (Word 2 and Word 3 in Figure 3-3).

FMLLAL

This node reads the low word of the last-term constant a_n, multiplies it with the low word of the variable x, giving high-word multiplication data (Word 1 in Figure 3-3).

FADDL1, FADDL2, FADDL3

These nodes add the multiplication data (Word 1, Word 2, and Word 4 in Figure 3-3) corresponding to the second word of the resultant value, and determine the data to be carried to a higher word.

FADDH1, FADDH2, FADDL4

These nodes add the data (Word 3, Word 5, Word 6, and carry data in Figure 3-3) corresponding to the third word (Word 8) of the resultant value, and determine the data to be carried to the third word high word (Word 9 in Figure 3-3).

FADDH3, FADDH4, FADDH5

These nodes determine the fourth word of the resultant value (Word 9 in Figure 3-3).

FADDAL

This node adds the low words of the constants a_i, except for the last-term constant a_n, to the third word of the multiplication results.

FADDAH

This node adds the high words of the constants a_i, except for the last-term constant a_n, to the fourth word of the multiplication results.

FADDH6

This node adds to the high word data the carries from the addition of the low words of the constants a_i.

FMULXH

This node reads the high word of the previously saved variable x from the DM, multiplies it with the high and low words of the summation data, giving 2-word multiplication data (Word 7, Word 6, Word 5 and Word 4 in Figure 3-3).

FMLHXL

This node reads the low word of the previously saved variable x from the DM, multiplies it with the high and low words of the summation data, giving 2-word multiplication data (Word 3 and Word 2 in Figure 3-3).

FMLLXL

This node reads the low word of the previously saved variable x from the DM, and multiplies it with the low word of the summation data, giving high-word multiplication data (Word 1 in Figure 3-3).

FADJL

This node adjusts the precision of the high and low words in the final results so that the words will have the same sign.

FANS

This node transmits the results of the computations to the host system in the high word-low word sequence.

3.5 Computation of Cosine by Taylor Series Expansion

The Taylor series expansion of cosine may be expressed as follows:

$$\cos\theta = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i)!}\theta^{2i} = 1 - \frac{1}{2!}\theta^2 + \frac{1}{4!}\theta^4 - \frac{1}{6!}\theta^6 + \cdots\cdots$$

where θ is the radian angle (rads)

(Eq. 3-3)

where θ is the radian angle (rad).

Expressing this formula in terms of Equation 3-1, a_i and x become:

$$a_i = \frac{(-1)^i}{(2i)!}, \quad x = \theta^2 \qquad \text{(Eq. 3-4)}$$

In the evaluation of Equation 3-3 using this program, the range of values that can be assumed by θ will be $0 \leq \theta < 1$. The following illustrates the computation using the procedure in section 3.4, at 33-bit precision.

Since none of the constants a_i assumes a value greater than 1, for these variables the decimal point is placed between bit 32 and bit 31. From this, it follows that the resultant value will consist of 7 terms, i = 0 through 6; for the 8th and higher terms, the constants a_i cannot be expressed within the limitation of 32 bits.

Let θ = 0.5 [rad]. Then the parameter values will be as follows:

$x = \theta^2 = 0.5^2 = 0.25 \longrightarrow$ 4000 0000H

$a_0 = 1 \longrightarrow$ 8000 0000H

$a_1 = -\frac{1}{2!} = -0.5 \longrightarrow$ 4000 0000H

$a_2 = \frac{1}{4!} = 0.0416667 \longrightarrow$ 0555 5558H

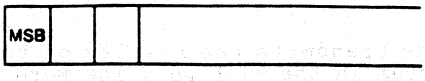
$a_3 = -\frac{1}{6!} = -0.00138889 \longrightarrow$ 002D 82D8H

$a_4 = \frac{1}{8!} = 0.000024802 \longrightarrow$ 0000 D00DH

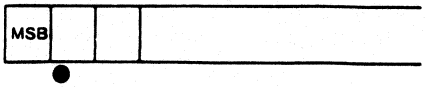
$a_5 = -\frac{1}{10!} = -0.000000276 \longrightarrow$ 0000 024FH

$a_6 = \frac{1}{12!} = 0.000000002 \longrightarrow$ 0000 0004H

Decimal point location for x:



Decimal point location for a_i:



The result of a computation using these parameters will be:

7054 A019H (0.877582561)

with the decimal point located between bit 32 and bit 31, the same as in the constants a_i.

The number enclosed in the parentheses is a decimal conversion of the hexadecimal value.

Figure 3-4
Polynomial Evaluation Flow Graph
(33 bit x 33 bit = 33 bit)

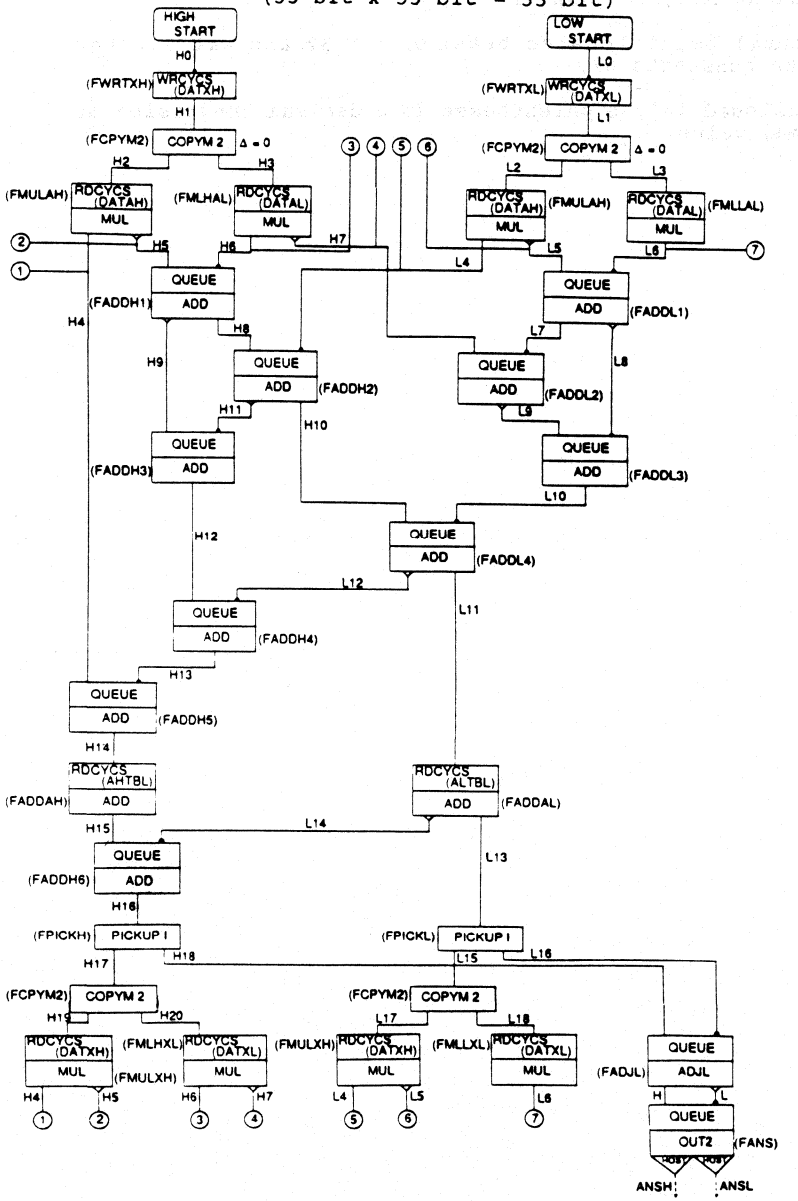


Figure 3-5
Polynomial Evaluation Source Program
(33 bit x 33 bit = 33 bit)

```
;*****************************************************************
;
;               MULTI-TERM EXPRESSION (33 bit x 33 bit = 33 bit)
;
;*****************************************************************
;
MODULE  MUL3    =       8                                       ;
;
EQUATE  I       =       6                                       ;
;
EQUATE  XH      =       4000H                                   ;
EQUATE  XL      =       0000H                                   ;
;
EQUATE  HOST    =       0                                       ;
;
;*****************************************************************
;                       INPUT
;*****************************************************************
;
INPUT   H0,     L0                                              ;
;
;*****************************************************************
;                       OUTPUT
;*****************************************************************
;
OUTPUT  ANSH,   ANSL                                            ;
;
;*****************************************************************
;               LINK
;*****************************************************************
;
LINK    H1                      = FWRTXH (   , H0 )             ;
LINK    H2,     H3              = FCPYM2 (H1      )             ;
LINK    H4,     H5              = FMULAH (H2      )             ;
LINK    H6,     H7              = FMLHAL (H3      )             ;
LINK    H8,     H9              = FADDH1 (H5 , H6 )             ;
LINK    H10,    H11             = FADDH2 (H8 , L4 )             ;
LINK    H12                     = FADDH3 (H9 , H11)             ;
LINK    H13                     = FADDH4 (H12, L12)             ;
LINK    H14                     = FADDH5 (H4 , H13)             ;
LINK    H15                     = FADDAH (H14     )             ;
LINK    H16                     = FADDH6 (H15, L14)             ;
LINK    H17,    H18             = FPICKH (H16     )             ;
LINK    H19,    H20             = FCPYM2 (H17     )             ;
LINK    H4,     H5              = FMULXH (H19     )             ;
LINK    H6,     H7              = FMLHXL (H20     )             ;
LINK    L1                      = FWRTXL (   , L0 )             ;
LINK    L2,     L3              = FCPYM2 (L1      )             ;
LINK    L4,     L5              = FMULAH (L2      )             ;
LINK    L6                      = FMLLAL (L3      )             ;
```

Figure 3-5 (cont.)
Polynomial Evaluation Source Program
(33 bit x 33 bit = 33 bit)

```
LINK    L7,     L8              = FADDL1 (L5 , L6 )             ;
LINK    L9                      = FADDL2 (H7 , L7 )             ;
LINK    L10                     = FADDL3 (L9 , L8 )             ;
LINK    L11,    L12             = FADDL4 (H10, L10)             ;
LINK    L13,    L14             = FADDAL (L11     )             ;
LINK    L15,    L16             = FPICKL (L13     )             ;
LINK    L17,    L18             = FCPYM2 (L15     )             ;
LINK    L4,     L5              = FMULXH (L17     )             ;
LINK    L6                      = FMLLXL (L18     )             ;
LINK    H,      L               = FADJL  (H18, L16)             ;
LINK    ANSH,   ANSL            = FANS   (H   , L  )            ;
;
;*****************************************************************
;               FUNCTION
;*****************************************************************
;
FUNCTION        FWRTXH  = WRCYCS (DATXH, 1)                     ;
FUNCTION        FWRTXL  = WRCYCS (DATXL, 1)                     ;
FUNCTION        FCPYM2  = COPYM  (2,     0)                     ;
FUNCTION        FMULAH  = MUL    (XY      ), RDCYCS (DATAH, 1)  ;
FUNCTION        FMLHAL  = MUL    (XY      ), RDCYCS (DATAL, 1)  ;
FUNCTION        FMLLAL  = MUL,               RDCYCS (DATAL, 1)  ;
FUNCTION        FMULXH  = MUL    (XY      ), RDCYCS (DATXH, 1)  ;
FUNCTION        FMLHXL  = MUL    (XY      ), RDCYCS (DATXL, 1)  ;
FUNCTION        FMLLXL  = MUL,               RDCYCS (DATXL, 1)  ;
FUNCTION        FADDH1  = ADD    (XY      ), QUEUE  (QUEH1, 1)  ;
FUNCTION        FADDH2  = ADD    (XY      ), QUEUE  (QUEH2, 1)  ;
FUNCTION        FADDH3  = ADD,               QUEUE  (QUEH3, 1)  ;
FUNCTION        FADDH4  = ADD,               QUEUE  (QUEH4, 1)  ;
FUNCTION        FADDH5  = ADD,               QUEUE  (QUEH5, 1)  ;
FUNCTION        FADDH6  = ADD,               QUEUE  (QUEH6, 1)  ;
FUNCTION        FADDL1  = ADD    (XY      ), QUEUE  (QUEL1, 1)  ;
FUNCTION        FADDL2  = ADD    (Y       ), QUEUE  (QUEL2, 1)  ;
FUNCTION        FADDL3  = ADD,               QUEUE  (QUEL3, 1)  ;
FUNCTION        FADDL4  = ADD    (XY      ), QUEUE  (QUEL4, 1)  ;
FUNCTION        FADDAH  = ADD,               RDCYCS (AHTBL, I)  ;
FUNCTION        FADDAL  = ADD    (XY      ), RDCYCS (ALTBL, I)  ;
FUNCTION        FPICKH  = PICKUP (I       )                     ;
FUNCTION        FPICKL  = PICKUP (I       )                     ;
FUNCTION        FADJL   = ADJL   (XY      ), QUEUE  (QUE1,  1)  ;
FUNCTION        FANS    = OUT2 (HOST, 0, 0), QUEUE  (QUE2,  1)  ;
;
;*****************************************************************
;               MEMORY
;*****************************************************************
;
MEMORY  AHTBL   =  0000H,  0000H, -002DH,  0555H, -4000H,  8000H;
MEMORY  ALTBL   = -024FH, 0D00DH, -82D8H,  5558H,  0000H,  0000H;
MEMORY  DATXH   =       AREA (1)                                ;
MEMORY  DATXL   =       AREA (1)                                ;
```

Figure 3-5 (cont.)
Polynomial Evaluation Source Program
(33 bit x 33 bit = 33 bit)

```
MEMORY  DATAH   =       0000H                                   ;
MEMORY  DATAL   =       0004H                                   ;
MEMORY  QUE1    =       AREA (1)                                ;
MEMORY  QUE2    =       AREA (1)                                ;
MEMORY  QUEH1   =       AREA (1)                                ;
MEMORY  QUEH2   =       AREA (1)                                ;
MEMORY  QUEH3   =       AREA (1)                                ;
MEMORY  QUEH4   =       AREA (1)                                ;
MEMORY  QUEH5   =       AREA (1)                                ;
MEMORY  QUEH6   =       AREA (1)                                ;
MEMORY  QUEL1   =       AREA (1)                                ;
MEMORY  QUEL2   =       AREA (1)                                ;
MEMORY  QUEL3   =       AREA (1)                                ;
MEMORY  QUEL4   =       AREA (1)                                ;
;
;*****************************************************************
;               START
;*****************************************************************
;
START                                                           ;
DATA    EXEC    (MUL3,  H0,     XH)                             ;
DATA    EXEC    (MUL3,  L0,     XL)                             ;
;
END                                                             ;
```

Chapter 4

Multiplication

4.1 Processing Explained

In this computation two terms are multiplied using the data lengths shown in Table 4-1 and represented by Equation 4-1.

$$z = x \cdot y \qquad \text{(Eq. 4-1)}$$

Table 4-1
Multiplication Data Lengths

VARIABLE X	VARIABLE Y	RESULT OF MULTIPLICATION
33 BITS	33 BITS	65 BITS
49 BITS	49 BITS	97 BITS

(INCLUDING SIGN BIT)

4.2 33 Bit x 33 Bit = 65 Bit

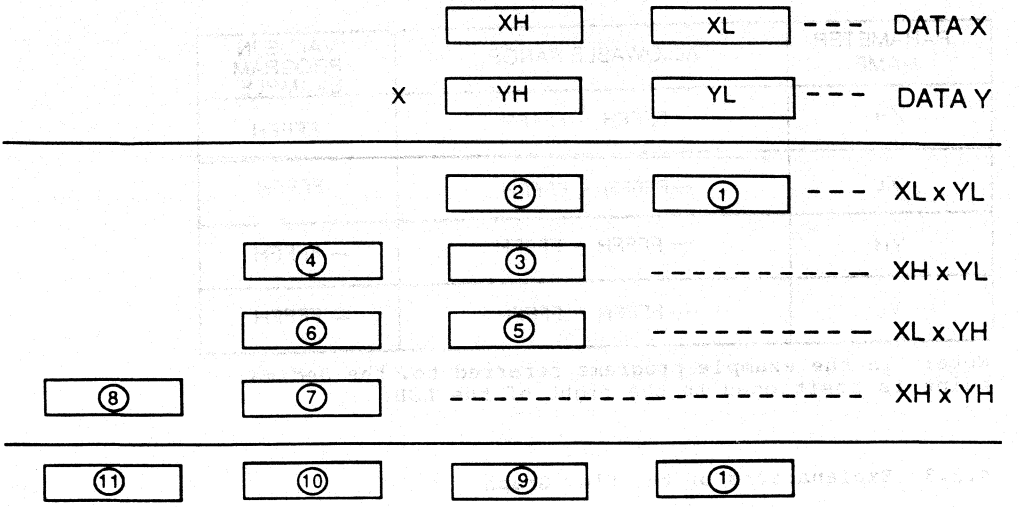
4.2.1 Algorithm

The data stored in variables x and y are expressed in 32 numerical value bits and a sign bit, for a total of 33 bits. The decimal point may be placed anywhere within these bits. The results of multiplication are represented in 65 bits, consisting of 64 bits for the numerical value and a sign bit. The decimal point placement will be at a position which is the sum of the decimal point positions of variables x and y.

In multiplying x and y, the data are divided into high and low words of 17 bits each, as shown in Figure 4-1.

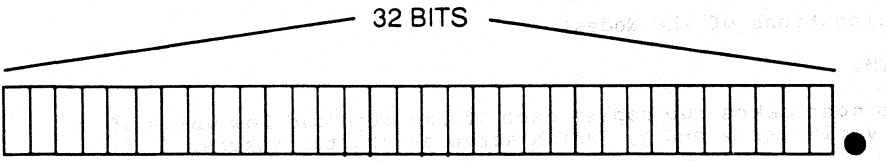
The multiplication data obtained by the partitioning of x and y are divided into high words and low words, each of which is added to the words having corresponding bit positions. Further, to add the products of multiplication, there must be a provision for adding the carries from the lower words to the higher words.

Figure 4-1
The Multiplication of 33-Bit Data Variables x and y



When the data contained in variables x and y are integers, the decimal point is positioned as shown in Figure 4-2.

Figure 4-2
Decimal Point Placement for Integer Multiplication



4.2.2 The Allowable Range of Parameter Values

<Startup Token-Defined Parameters>

XH .... High-word data for variable x
XL .... Low-word data for variable x
YH .... High-word data for variable y
YL .... Low-word data for variable y

Table 4-2 lists the range of values allowed for the parameters.

Table 4-2
Allowable Range of Parameter Values
(33 bit x 33 bit = 65 bit)

PARAMETER NAME	ALLOWABLE RANGE	VALUE IN PROGRAM EXAMPLE
XH	—FFFFH ~ FFFFH	FFFFH
XL	—FFFFH ~ FFFFH	FFFFH
YH	— FFFFH ~ FFFFH	— FFFFH
YL	— FFFFH ~ FFFFH	— FFFFH

Note: In the example programs referred to, the decimal point is positioned to the right of the LSB.

4.2.3 Explanations of the Flow Graph

In this program the high and low words of each of the variables x and y, for a total of four words of data, are contained in startup tokens.

The results of 65-bit multiplications are expressed in four data words. Starting from the most significant (highest) word, the results are identified by 4, 2, 1, and 0, respectively, in the ID fields of the output tokens.

<Explanations of the Nodes>

FCPYM2

This node makes two copies each of the high and low words of the variables x and y, which serve as startup tokens.

FMULHH

This node multiplies the high words of variables x and y.

FMULHL

This node multiplies the high word of variables x with the low word of variable y.

FMULLH

This node multiplies the low word of variable x with the high word of variable y.

FMULLL

This node multiplies the low words of variables x and y.

FADLH1, FADLH2

These nodes add the products of the multiplication, corresponding to the second word of the resultant value.

FADHL1, FADHL2, FADHL3, FADHL4

These nodes add the products of the multiplication, corresponding to the third word of the resultant value.

FADHH1, FADHH2, FADHH3

These nodes add the products of the multiplication, corresponding to the fourth word of the resultant value.

FANSH

This node sends the data in the fourth and third words of the results obtained to the host system. The numbers in the ID fields of the output tokens are 4 and 2, respectively.

FANSL

This node sends the data in the second and first words of the results obtained to the host system. The numbers in the ID fields of the output tokens are 1 and 0, respectively.

Figure 4-3
Multiplication Flow Graph
(33 bit x 33 bit = 65 bit)

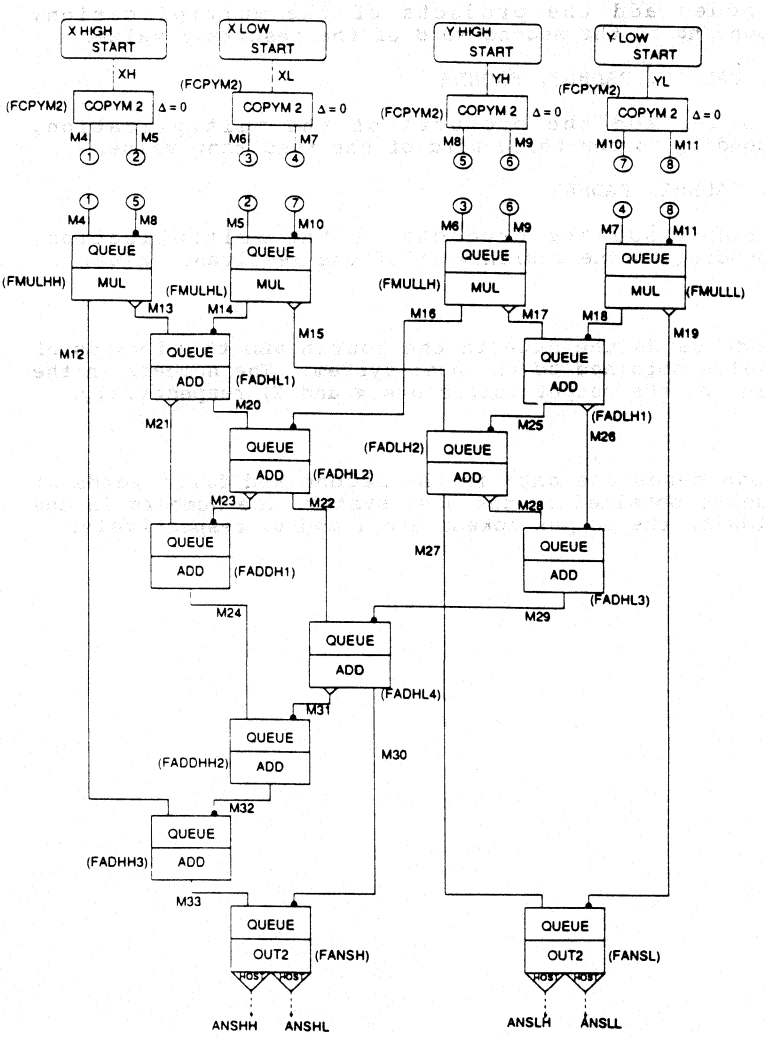


Figure 4-4
Multiplication Source Program
(33 bit x 33 bit = 65 bit)

```
;*****************************************************************
;
;               MULTIPLICATION    (33 bit X 33 bit = 65 bit)
;
;*****************************************************************
;
MODULE  MULT65  =       8                                       ;
;
EQUATE  HOST    =       0                                       ;
;
EQUATE  XHVAL   =       0FFFFH                                  ;
EQUATE  XLVAL   =       0FFFFH                                  ;
EQUATE  YHVAL   =       -0FFFFH                                 ;
EQUATE  YLVAL   =       -0FFFFH                                 ;
;
;*****************************************************************
;                       INPUT
;*****************************************************************
;
INPUT   XH,     XL,     YH,     YL                              ;
;
;*****************************************************************
;                       OUTPUT
;*****************************************************************
;
OUTPUT  ANSHH,  ANSHL,  ANSLH,  ANSLL                           ;
;
;*****************************************************************
;                       LINK
;*****************************************************************
;
LINK    M4,     M5              = FCPYM2 (XH      )             ;
LINK    M6,     M7              = FCPYM2 (XL      )             ;
LINK    M8,     M9              = FCPYM2 (YH      )             ;
LINK    M10,    M11             = FCPYM2 (YL      )             ;
LINK    M12,    M13             = FMULHH (M4,  M8 )             ;
LINK    M14,    M15             = FMULHL (M5,  M10)             ;
LINK    M16,    M17             = FMULLH (H6,  M9 )             ;
LINK    M18,    M19             = FMULLL (M7,  M11)             ;
LINK    M20,    M21             = FADHL1 (M13, M14)             ;
LINK    M22,    M23             = FADHL2 (M20, M16)             ;
LINK    M24                     = FADHH1 (M21, M23)             ;
LINK    M25,    M26             = FADLH1 (M17, M18)             ;
LINK    M27,    M28             = FADLH2 (M15, M25)             ;
LINK    M29                     = FADHL3 (M28, M26)             ;
LINK    M30,    M31             = FADHL4 (M22, M29)             ;
LINK    M32                     = FADHH2 (M24, M31)             ;
LINK    M33                     = FADHH3 (M12, M32)             ;
LINK    ANSHH,  ANSHL           = FANSH  (M33, M30)             ;
LINK    ANSLH,  ANSLL           = FANSL  (M27, M19)             ;
```

Figure 4-4 (cont.)
Multiplication Source Program
(33 bit x 33 bit = 65 bit)

```
;*****************************************************************
;                          FUNCTION
;*****************************************************************
;
FUNCTION        FCPYM2  = COPYM  (2,   0)                          ;
FUNCTION        FMULHH  = MUL    (XY    ), QUEUE  (QUEHH, 16)      ;
FUNCTION        FMULHL  = MUL    (XY    ), QUEUE  (QUEHL, 16)      ;
FUNCTION        FMULLH  = MUL    (XY    ), QUEUE  (QUELH, 16)      ;
FUNCTION        FMULLL  = MUL    (XY    ), QUEUE  (QUELL, 16)      ;
FUNCTION        FADHH1  = ADD,             QUEUE  (QUEA1, 16)      ;
FUNCTION        FADHH2  = ADD,             QUEUE  (QUEA2, 16)      ;
FUNCTION        FADHH3  = ADD,             QUEUE  (QUEA3, 16)      ;
FUNCTION        FADHL1  = ADD    (XY    ), QUEUE  (QUEA4, 16)      ;
FUNCTION        FADHL2  = ADD    (XY    ), QUEUE  (QUEA5, 16)      ;
FUNCTION        FADHL3  = ADD,             QUEUE  (QUEA6, 16)      ;
FUNCTION        FADHL4  = ADD    (XY    ), QUEUE  (QUEA7, 16)      ;
FUNCTION        FADLH1  = ADD    (XY    ), QUEUE  (QUEA8, 16)      ;
FUNCTION        FADLH2  = ADD    (XY    ), QUEUE  (QUEA9, 16)      ;
FUNCTION        FANSH   = OUT2 (HOST, 4, 2),QUEUE (QUEH,  16)      ;
FUNCTION        FANSL   = OUT2 (HOST, 1, 0),QUEUE (QUEL,  16)      ;
;
;*****************************************************************
;                          MEMORY
;*****************************************************************
;
MEMORY   QUEA1   =       AREA (16)                                 ;
MEMORY   QUEA2   =       AREA (16)                                 ;
MEMORY   QUEA3   =       AREA (16)                                 ;
MEMORY   QUEA4   =       AREA (16)                                 ;
MEMORY   QUEA5   =       AREA (16)                                 ;
MEMORY   QUEA6   =       AREA (16)                                 ;
MEMORY   QUEA7   =       AREA (16)                                 ;
MEMORY   QUEA8   =       AREA (16)                                 ;
MEMORY   QUEA9   =       AREA (16)                                 ;
MEMORY   QUEH    =       AREA (16)                                 ;
MEMORY   QUEL    =       AREA (16)                                 ;
MEMORY   QUEHH   =       AREA (16)                                 ;
MEMORY   QUEHL   =       AREA (16)                                 ;
MEMORY   QUELH   =       AREA (16)                                 ;
MEMORY   QUELL   =       AREA (16)                                 ;
;
;*****************************************************************
;                          START
;*****************************************************************
START                                                              ;
DATA     EXEC    (MULT65,        XH,     XHVAL)                    ;
DATA     EXEC    (MULT65,        XL,     XLVAL)                    ;
DATA     EXEC    (MULT65,        YH,     YHVAL)                    ;
DATA     EXEC    (MULT65,        YL,     YLVAL)                    ;
END                                                                ;
```

4.3 49 Bit x 49 Bit = 97 Bit

4.3.1 Algorithm

The data in variables x and y are represented in 49 bits, 48 numerical value bits and a sign bit, with the decimal point positioned anywhere in those bits.

The results of the multiplication are expressed in 97 bits, 96 numerical value bits and a sign bit, with the decimal point at a position which is the sum of the decimal point positions of x and y. Since the 97-bit results of a multiplication cannot be expressed in a single word, they are represented in 6 separate words, each word consisting of 17 bits, including a sign bit.

As described in section 4.2, the multiplication of variables x and y is represented by dividing the data. In this example, the data are divided into high, middle, and low words (Figure 4-5).

The multiplication results are divided into a signed high word and low word, each of which is added to the product data having corresponding bit positions. For the addition of multiplication data, a provision must be made for adding the carries.

Figure 4-5
The Multiplication of 49-Bit Data Variables x and y

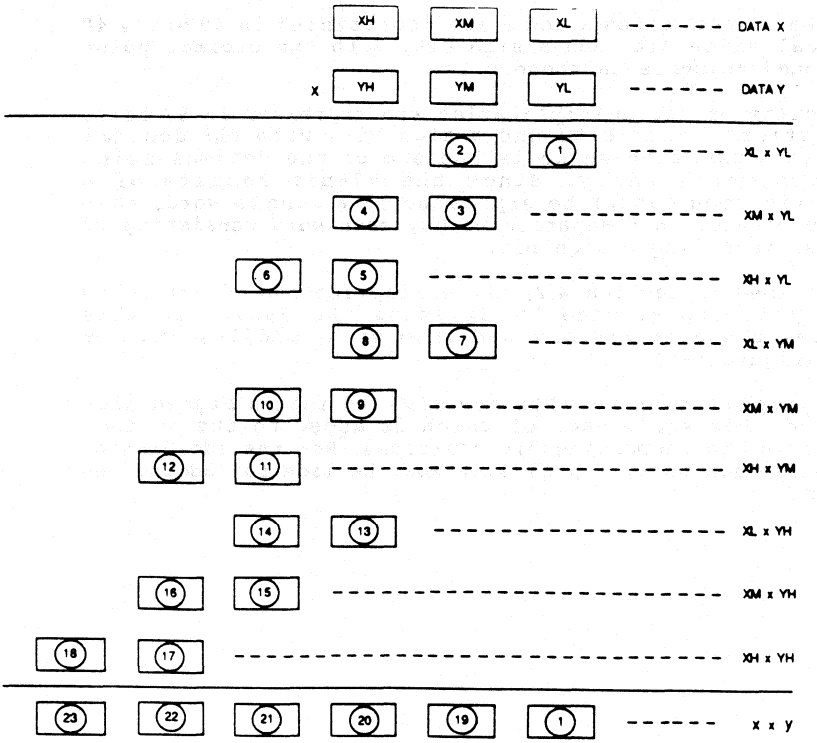
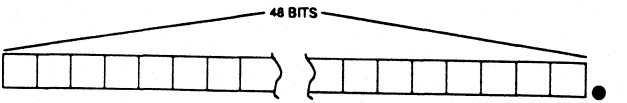


Figure 4-6 shows the placement of the decimal point when the data contained in variables x and y are integers.

Figure 4-6
Decimal Point Position for 49-Bit Integer



4.3.2 Allowable Range of Parameter Values

<Startup Token-Defined Parameters>

XH .... High-word data for variable x
XM .... Middle-word data for variable x
XL .... Low-word data for variable x
YH .... High-word data for variable y
YM .... Middle-word data for variable y
YL .... Low-word data for variable y

Table 4-3 shows the allowable values of the parameters.

Table 4-3
Allowable Range of Parameter Values
(49 bit x 49 bit = 97 bit)

PARAMETER NAME	ALLOWABLE RANGE	VALUE IN PROGRAM EXAMPLE
XH	—FFFFH ~ FFFFH	—FFFFH
XM	— FFFFH ~ FFFFH	— FFFFH
XL	— FFFFH ~ FFFFH	— FFFFH
YH	— FFFFH ~ FFFFH	FFFFH
YM	— FFFFH ~ FFFFH	FFFFH
YL	—FFFFH ~ FFFFH	FFFFH

Note: In the example programs referred to, the decimal point is located to the right of the LSB.

4.3.3 Explanations of the Flow Graph

In this program, a total of 6 words of data, comprised of the high, middle, and low words of variables x and y, are used in startup tokens.

The 97-bit multiplication result is expressed in 6 words of data. Starting with the most significant (highest) word, the data are identified by 10H, 8, 4, 2, 1, and 0, respectively, in the ID field of the output tokens.

<Explanations of the Nodes>

FCPYM3

This node makes three copies each of the high, middle, and low words of variables x and y.

FMULHH

This node multiplies the high word of variable x with the high word of variable y.

FMULMH

This node multiplies the middle word of variable x with the high word of variable y.

FMULLH

This node multiplies the low word of variable x with the high word of variable y.

FMULHM

This node multiplies the high word of variable x with the middle word of variable y.

FMULMM

This node multiplies the middle word of variable x with the middle word of variable y.

FMULLM

This node multiplies the low word of variable x with the middle word of variable y.

FMULHL

This node multiplies the high word of variable x with the low word of variable y.

FMULML

This node multiplies the middle word of variable x with the low word of variable y.

FMULLL

This node multiplies the low word of variable x with the low word of variable y.

FADD21, FADD22

These nodes add the multiplication product data corresponding to the second word of the multiplication resultant value.

FADD31 - FADD36

These nodes add the multiplication product and carries data corresponding to the third word of the resultant value.

FADD41 - FADD49

These nodes add the multiplication product and carries data corresponding to the fourth word of the resultant value.

FADD51 - FADD57

These nodes add the multiplication product and carries data corresponding to the fifth word of the resultant value.

FADD61 - FADD63

These nodes add the multiplication product and carries data corresponding to the sixth word of the resultant value.

FANSH

This node sends the data in the sixth and fifth words of the resultant value to the host system.

FANSM

This node sends the data in the fourth and third words of the resultant value to the host system.

FANSL

This node sends the data in the second and first words of the resultant value to the host system.

Figure 4-7
Multiplication Flow Graph
(49 bit x 49 bit = 97 bit)

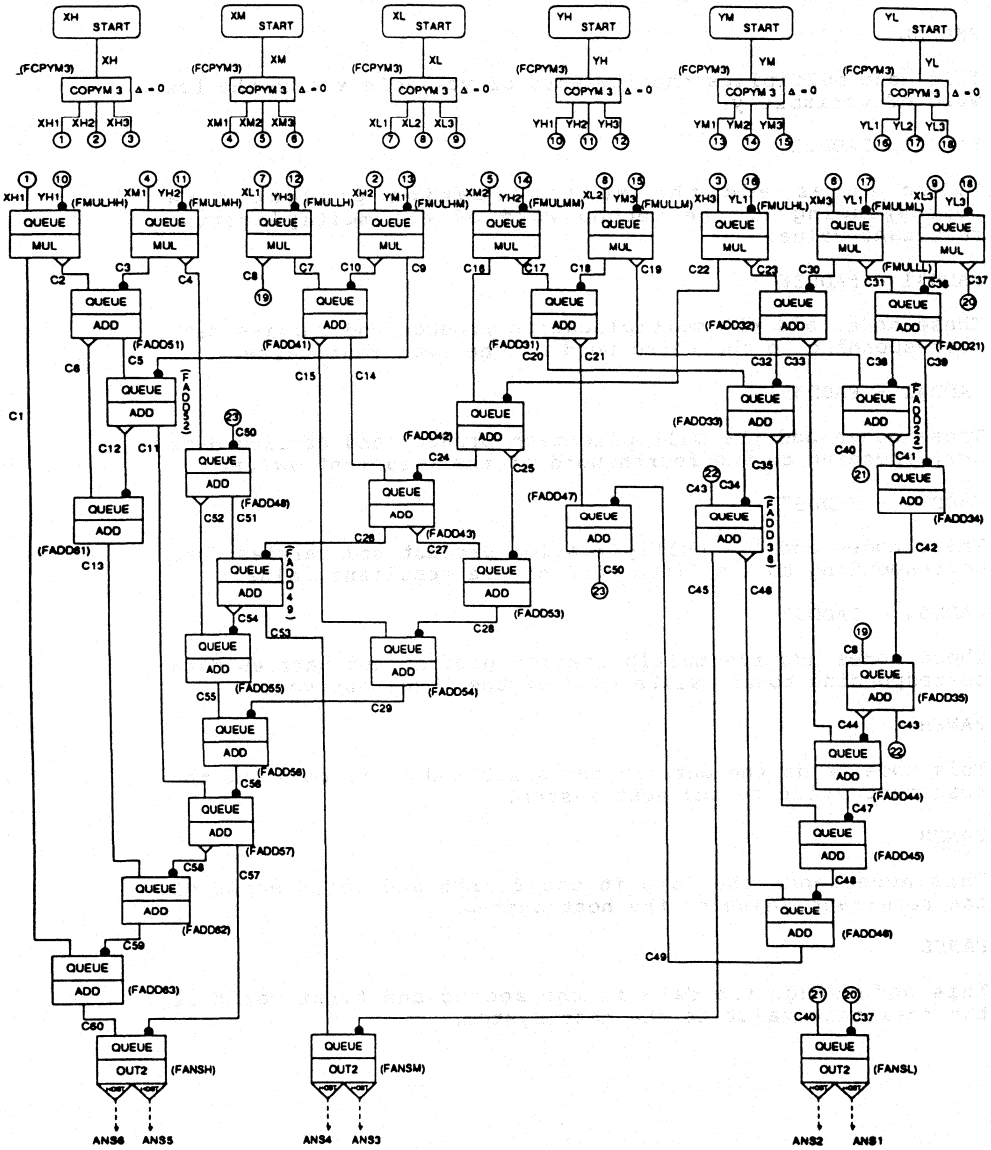


Figure 4-8
Multiplication Source Program
(49 bit x 49 bit = 97 bit)

```
;*****************************************************************
;
;              MULTIPLICATION (49 bit X 49 bit = 97 bit)
;
;*****************************************************************
;
MODULE  MULT97  =       8                                       ;
;
EQUATE  XHVAL   =       -0FFFFH                                 ;
EQUATE  XMVAL   =       -0FFFFH                                 ;
EQUATE  XLVAL   =       -0FFFFH                                 ;
EQUATE  YHVAL   =       0FFFFH                                  ;
EQUATE  YMVAL   =       0FFFFH                                  ;
EQUATE  YLVAL   =       0FFFFH                                  ;
;
EQUATE  HOST    =       0                                       ;
;
;*****************************************************************
;                       INPUT
;*****************************************************************
;
INPUT   XH,     XM,     XL,     YH,     YM,     YL              ;
;
;*****************************************************************
;                       OUTPUT
;*****************************************************************
;
OUTPUT  ANS6,   ANS5,   ANS4,   ANS3,   ANS2,   ANS1            ;
;
;*****************************************************************
;                       LINK
;*****************************************************************
;
LINK    XH1,    XH2,    XH3     = FCPYM3 (XH      )             ;
LINK    XM1,    XM2,    XM3     = FCPYM3 (XM      )             ;
LINK    XL1,    XL2,    XL3     = FCPYM3 (XL      )             ;
LINK    YH1,    YH2,    YH3     = FCPYM3 (YH      )             ;
LINK    YM1,    YM2,    YM3     = FCPYM3 (YM      )             ;
LINK    YL1,    YL2,    YL3     = FCPYM3 (YL      )             ;
LINK    C1,     C2              = FMULHH (XH1, YH1)             ;
LINK    C3,     C4              = FMULMH (XM1, YH2)             ;
LINK    C5,     C6              = FADD51 (C2,  C3 )             ;
LINK    C7,     C8              = FMULLH (XL1, YH3)             ;
LINK    C9,     C10             = FMULHM (XH2, YM1)             ;
LINK    C11,    C12             = FADD52 (C5,  C9 )             ;
LINK    C13                     = FADD61 (C6,  C12)             ;
LINK    C14,    C15             = FADD41 (C7,  C10)             ;
LINK    C16,    C17             = FMULMM (XM2, YM2)             ;
LINK    C18,    C19             = FMULLM (XL2, YM3)             ;
LINK    C20,    C21             = FADD31 (C17, C18)             ;
```

Figure 4-8 (cont.)
Multiplication Source Program
(49 bit x 49 bit = 97 bit)

```
LINK     C22,    C23             = FMULHL (XH3, YL1)             ;
LINK     C24,    C25             = FADD42 (C16, C22)             ;
LINK     C26,    C27             = FADD43 (C14, C24)             ;
LINK     C28                     = FADD53 (C27, C25)             ;
LINK     C29                     = FADD54 (C15, C28)             ;
LINK     C30,    C31             = FMULML (XM3, YL2)             ;
LINK     C32,    C33             = FADD32 (C23, C30)             ;
LINK     C34,    C35             = FADD33 (C20, C32)             ;
LINK     C36,    C37             = FMULLL (XL3, YL3)             ;
LINK     C38,    C39             = FADD21 (C31, C36)             ;
LINK     C40,    C41             = FADD22 (C19, C38)             ;
LINK     C42                     = FADD34 (C41, C39)             ;
LINK     C43,    C44             = FADD35 (C8,  C42)             ;
LINK     C45,    C46             = FADD36 (C43, C34)             ;
LINK     C47                     = FADD44 (C33, C44)             ;
LINK     C48                     = FADD45 (C35, C47)             ;
LINK     C49                     = FADD46 (C46, C48)             ;
LINK     C50                     = FADD47 (C21, C49)             ;
LINK     C51,    C52             = FADD48 (C4,  C50)             ;
LINK     C53,    C54             = FADD49 (C51, C26)             ;
LINK     C55                     = FADD55 (C52, C54)             ;
LINK     C56                     = FADD56 (C55, C29)             ;
LINK     C57,    C58             = FADD57 (C11, C56)             ;
LINK     C59                     = FADD62 (C13, C58)             ;
LINK     C60                     = FADD63 (C1,  C59)             ;
LINK     ANS6,   ANS5            = FANSH  (C60, C57)             ;
LINK     ANS4,   ANS3            = FANSM  (C53, C45)             ;
LINK     ANS2,   ANS1            = FANSL  (C40, C37)             ;
;
;*****************************************************************
;                          FUNCTION
;*****************************************************************
;
FUNCTION         FCPYM3  = COPYM  (3,   0)                       ;
FUNCTION         FMULHH  = MUL    (XY    ), QUEUE  (QUEM1, 11)   ;
FUNCTION         FMULMH  = MUL    (XY    ), QUEUE  (QUEM2, 11)   ;
FUNCTION         FMULLH  = MUL    (XY    ), QUEUE  (QUEM3, 11)   ;
FUNCTION         FMULHM  = MUL    (XY    ), QUEUE  (QUEM4, 11)   ;
FUNCTION         FMULMM  = MUL    (XY    ), QUEUE  (QUEM5, 11)   ;
FUNCTION         FMULLM  = MUL    (XY    ), QUEUE  (QUEM6, 11)   ;
FUNCTION         FMULHL  = MUL    (XY    ), QUEUE  (QUEM7, 11)   ;
FUNCTION         FMULML  = MUL    (XY    ), QUEUE  (QUEM8, 11)   ;
FUNCTION         FMULLL  = MUL    (XY    ), QUEUE  (QUEM9, 11)   ;
FUNCTION         FADD21  = ADD    (XY    ), QUEUE  (QUEA1, 11)   ;
FUNCTION         FADD22  = ADD    (XY    ), QUEUE  (QUEA2, 11)   ;
FUNCTION         FADD31  = ADD    (XY    ), QUEUE  (QUEA3, 11)   ;
FUNCTION         FADD32  = ADD    (XY    ), QUEUE  (QUEA4, 11)   ;
FUNCTION         FADD33  = ADD    (XY    ), QUEUE  (QUEA5, 11)   ;
FUNCTION         FADD34  = ADD,           QUEUE  (QUEA6, 11)   ;
FUNCTION         FADD35  = ADD    (XY    ), QUEUE  (QUEA7, 11)   ;
```

Figure 4-8 (cont.)
Multiplication Source Program
(49 bit x 49 bit = 97 bit)

```
FUNCTION        FADD36  = ADD     (XY     ), QUEUE  (QUEA8, 11)   ;
FUNCTION        FADD41  = ADD     (XY     ), QUEUE  (QUEA9, 11)   ;
FUNCTION        FADD42  = ADD     (XY     ), QUEUE  (QUEA10, 11)  ;
FUNCTION        FADD43  = ADD     (XY     ), QUEUE  (QUEA11, 11)  ;
FUNCTION        FADD44  = ADD,               QUEUE  (QUEA12, 11)  ;
FUNCTION        FADD45  = ADD,               QUEUE  (QUEA13, 11)  ;
FUNCTION        FADD46  = ADD,               QUEUE  (QUEA14, 11)  ;
FUNCTION        FADD47  = ADD,               QUEUE  (QUEA15, 11)  ;
FUNCTION        FADD48  = ADD     (XY     ), QUEUE  (QUEA16, 11)  ;
FUNCTION        FADD49  = ADD     (XY     ), QUEUE  (QUEA17, 11)  ;
FUNCTION        FADD51  = ADD     (XY     ), QUEUE  (QUEA18, 11)  ;
FUNCTION        FADD52  = ADD     (XY     ), QUEUE  (QUEA19, 11)  ;
FUNCTION        FADD53  = ADD,               QUEUE  (QUEA20, 11)  ;
FUNCTION        FADD54  = ADD,               QUEUE  (QUEA21, 11)  ;
FUNCTION        FADD55  = ADD,               QUEUE  (QUEA22, 11)  ;
FUNCTION        FADD56  = ADD,               QUEUE  (QUEA23, 11)  ;
FUNCTION        FADD57  = ADD     (XY     ), QUEUE  (QUEA24, 11)  ;
FUNCTION        FADD61  = ADD,               QUEUE  (QUEA25, 11)  ;
FUNCTION        FADD62  = ADD,               QUEUE  (QUEA26, 11)  ;
FUNCTION        FADD63  = ADD,               QUEUE  (QUEA27, 11)  ;
FUNCTION        FANSH   = OUT2  (HOST,10H,8),QUEUE  (QUEAH,  11)  ;
FUNCTION        FANSM   = OUT2  (HOST, 4, 2),QUEUE  (QUEAM,  11)  ;
FUNCTION        FANSL   = OUT2  (HOST, 1, 0),QUEUE  (QUEAL,  11)  ;
;
;*****************************************************************
;                       MEMORY
;*****************************************************************
;
MEMORY  QUEA1   =       AREA (11)                                  ;
MEMORY  QUEA2   =       AREA (11)                                  ;
MEMORY  QUEA3   =       AREA (11)                                  ;
MEMORY  QUEA4   =       AREA (11)                                  ;
MEMORY  QUEA5   =       AREA (11)                                  ;
MEMORY  QUEA6   =       AREA (11)                                  ;
MEMORY  QUEA7   =       AREA (11)                                  ;
MEMORY  QUEA8   =       AREA (11)                                  ;
MEMORY  QUEA9   =       AREA (11)                                  ;
MEMORY  QUEA10  =       AREA (11)                                  ;
MEMORY  QUEA11  =       AREA (11)                                  ;
MEMORY  QUEA12  =       AREA (11)                                  ;
MEMORY  QUEA13  =       AREA (11)                                  ;
MEMORY  QUEA14  =       AREA (11)                                  ;
MEMORY  QUEA15  =       AREA (11)                                  ;
MEMORY  QUEA16  =       AREA (11)                                  ;
MEMORY  QUEA17  =       AREA (11)                                  ;
MEMORY  QUEA18  =       AREA (11)                                  ;
MEMORY  QUEA19  =       AREA (11)                                  ;
MEMORY  QUEA20  =       AREA (11)                                  ;
MEMORY  QUEA21  =       AREA (11)                                  ;
MEMORY  QUEA22  =       AREA (11)                                  ;
```

Figure 4-8 (cont.)
Multiplication Source Program
(49 bit x 49 bit = 97 bit)

```
MEMORY   QUEA23   =        AREA (11)                                    ;
MEMORY   QUEA24   =        AREA (11)                                    ;
MEMORY   QUEA25   =        AREA (11)                                    ;
MEMORY   QUEA26   =        AREA (11)                                    ;
MEMORY   QUEA27   =        AREA (11)                                    ;
MEMORY   QUEAH    =        AREA (11)                                    ;
MEMORY   QUEAM    =        AREA (11)                                    ;
MEMORY   QUEAL    =        AREA (11)                                    ;
MEMORY   QUEM1    =        AREA (11)                                    ;
MEMORY   QUEM2    =        AREA (11)                                    ;
MEMORY   QUEM3    =        AREA (11)                                    ;
MEMORY   QUEM4    =        AREA (11)                                    ;
MEMORY   QUEM5    =        AREA (11)                                    ;
MEMORY   QUEM6    =        AREA (11)                                    ;
;
;*****************************************************************
;                          START
;*****************************************************************
;
START
DATA     EXEC     (MULT97,        XH,     XHVAL)                      ;
DATA     EXEC     (MULT97,        XM,     XMVAL)                      ;
DATA     EXEC     (MULT97,        XL,     XLVAL)                      ;
DATA     EXEC     (MULT97,        YH,     YHVAL)                      ;
DATA     EXEC     (MULT97,        YM,     YMVAL)                      ;
DATA     EXEC     (MULT97,        YL,     YLVAL)                      ;
;
END                                                                    ;
```

Chapter 5

Floating-Point Computations

5.1 Processing Explained

The floating-point operations illustrated in this chapter are multiplication and addition. The operations are defined by the equations:

Multiplication

$$(a \times 2^x) \times (b \times 2^y) = c \times 2^z \qquad \text{(Eq. 5-1)}$$

Addition

$$(a \times 2^x) + (b \times 2^y) = c \times 2^z \qquad \text{(Eq. 5-2)}$$

The data lengths of the calculations are shown in Table 5-1.

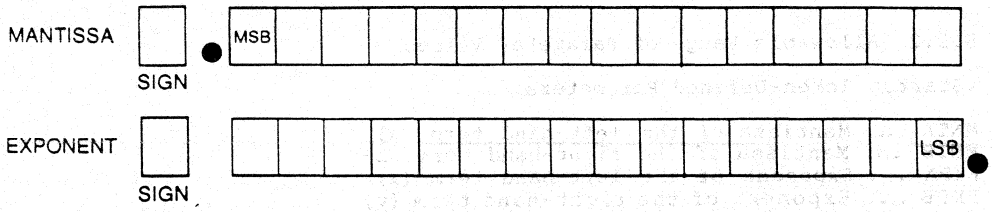
Table 5-1
Data Lengths Used in Floating-Point Examples
Addition and Multiplication

MANTISSA A	EXPONENT X	MANTISSA B	EXPONENT Y	RESULTANT MANTISSA C	RESULTANT EXPONENT Z
17 BITS	17 BITS	17 BITS	17 BITS	17 BITS	17 BITS

(INCLUDES SIGN BIT)

The mantissas of the floating-point numbers are represented by a, b, and c, with the decimal point of each mantissa fixed at the left of the MSB. The data is normalized so that the absolute value of the mantissa, m, is $0.5 \leq |m| < 1.0$. This insures that the value of the MSB will always be '1'. The exponents x, y, and z are signed integers, with the decimal point positioned to the right of the LSB.

Figure 5-1
Floating-Point Number Representation



5.2 Floating-Point Multiplication

5.2.1 Algorithm

Multiplication of exponential functions is performed by adding the exponents (assuming the functions have the same exponential base) as shown in Equation 5-3.

$$\chi^m \times \chi^n = \chi^{(m+n)} \qquad \text{(Eq. 5-3)}$$

Multiplication of floating-point numbers is performed by multiplying the mantissas and adding the exponents. Therefore, Equation 5-1 can be written as shown in Equation 5-4.

$$(a \times 2^{\chi}) \times (b \times 2^{y}) = a \times b \times 2^{(\chi + y)} \qquad \text{(Eq. 5-4)}$$

where mantissas a and b are fractional real numbers and exponents x and y are integers. These data are expressed in 17 bits, 16 numerical value bits and a sign bit.

Multiplication of mantissas a and b yields 32 numerical bits. This resultant value is then left-shifted (up to 16 times) until its MSB is '1'. The high word resulting from the shift operation will be the resultant mantissa. The low word resulting from the shift operation is truncated.

Addition of exponents x and y gives 16-numerical value bits, from which the number of left shifts performed during the mantissa computation is subtracted. The result is the desired exponent.

If the mantissa obtained by the above procedure is 0, the resultant exponent is set to its minimum possible value, -FFFFH.

When either an overflow or underflow occurs during the computation of an exponent, the exponent value is set to +FFFFH or -FFFFH, respectively.

5.2.2 Allowable Range of Parameter Values

<Startup Token-Defined Parameters>

MNTA .... Mantissa of the left-hand term (a)
MNTB .... Mantissa of the right-hand term (b)
EXPA .... Exponent of the left-hand term (x)
EXPB .... Exponent of the right-hand term (y)

Table 5-2 gives the allowable range of values for these parameters.

Table 5-2
Allowable Range of Parameter Values
(Floating-Point Multiplication)

PARAMETER NAME	ALLOWABLE RANGE	VALUE IN PROGRAM EXAMPLE
MNTA	—FFFFH ~ FFFFH	CF07H
MNTB	—FFFFH ~ FFFFH	BE84H
EXPA	— 65535 ~ 65535	— 16
EXPB	— 65535 ~ 65535	— 17

The left and right terms used in the example programs referred to in the Table 5-2 have the following decimal values:

Left term (a): 0.00001234
Right term (b): 0.000005678

5.2.3 Explanations of the Flow Graph

This program starts processing using four data points in startup tokens: mantissas a and b, and exponents x and y.

<Explanations of the Nodes>

FMUL

This node multiplies the mantissas, giving a 33-bit product including the sign, divided into high and low words.

FNOPSC

The node determines the number of bits the product of the mantissas is to be left-shifted by counting the number of leading zeros in the high word of the product.

FSHL1, FSHL2, FADDM

These nodes perform left-shifts on the multiplication product of the mantissas (so that the MSB will be '1') and extract the resulting high word, which will be the resultant mantissa. If this value is 0, its C Bit of the token is set to 1.

FADDE

This node adds the exponents.

FSUB

This node subtracts from the exponent value the number of shift operations performed on the mantissa.

FCOPYC, FNOP, FRDMIN

When the value of the mantissa is 0 (C Bit is 1), these nodes replace the resultant exponent value with -FFFFH.

FANS

This node sends the mantissa and exponent values obtained to the host system.

FORCBN

When either an overflow or an underflow occurs during the computation of an exponent value, this node sets the value of the exponent to +FFFFH or -FFFFH, respectively.

Figure 5-2
Floating-Point Multiplication Flow Graph

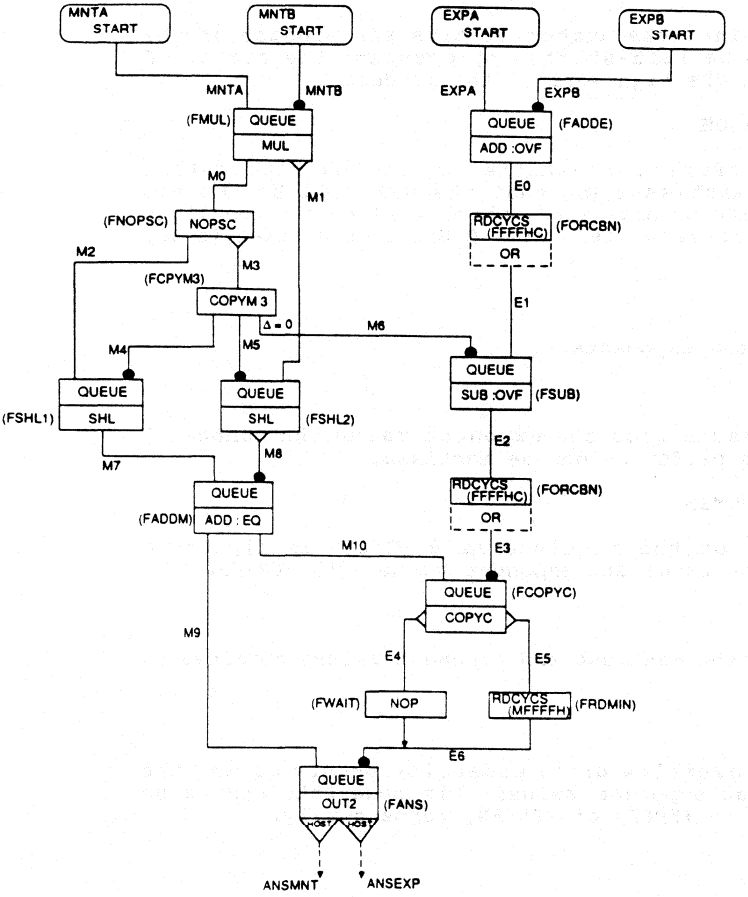


Figure 5-3
Floating-Point Multiplication Source Program

```
;*****************************************************************
;
;               FLOATING-POINT MULTIPLICATION
;
;               (MANTISSA : 17 bit x 17 bit = 17 bit)
;               (EXPONENT : 17 bit x 17 bit = 17 bit)
;
;*****************************************************************
;
MODULE  EXPMUL  =       8                                       ;
;
EQUATE  HOST    =       0                                       ;
;
EQUATE  MAVAL   =       0CF07H                                  ;
EQUATE  MBVAL   =       0BE84H                                  ;
EQUATE  EAVAL   =       -16                                     ;
EQUATE  EBVAL   =       -17                                     ;
;
;*****************************************************************
;                       INPUT
;*****************************************************************
;
INPUT   MNTA,   MNTB,   EXPA,   EXPB                            ;
;
;*****************************************************************
;                       OUTPUT
;*****************************************************************
;
OUTPUT  ANSMNT, ANSEXP                                          ;
;
;*****************************************************************
;                       LINK
;*****************************************************************
;
LINK    M0,     M1              = FMUL   (MNTA, MNTB)           ;
LINK    M2,     M3              = FNOPSC (M0       )            ;
LINK    M4,     M5,     M6      = FCPYM3 (M3       )            ;
LINK    M7                      = FSHL1  (M2,   M4 )            ;
LINK    M8                      = FSHL2  (M1,   M5 )            ;
LINK    M9,     M10             = FADDM  (M7,   M8 )            ;
LINK    E0                      = FADDE  (EXPA, EXPB)           ;
LINK    E1                      = FORCBN (E0       )            ;
LINK    E2                      = FSUB   (E1,   M6 )            ;
LINK    E3                      = FORCBN (E2       )            ;
LINK    E4,     E5              = FCOPYC (M10,  E3 )            ;
LINK    E6                      = FRDMIN (E5       )            ;
LINK    E6                      = FWAIT  (E4       )            ;
LINK    ANSMNT, ANSEXP          = FANS   (M9,   E6 )            ;
;
```

Figure 5-3 (cont.)
Floating-Point Multiplication Source Program

```
;*****************************************************************
;                        FUNCTION
;*****************************************************************
;
FUNCTION        FADDE   = ADD     (OVF     ), QUEUE   (QUEAE, 16)  ;
FUNCTION        FADDM   = ADD     (XX,  EQ), QUEUE   (QUEAM, 16)  ;
FUNCTION        FANS    = OUT2    (0, 0, 0), QUEUE   (QUEE,  16)  ;
FUNCTION        FCPYM3  = COPYM   (3,    0)                       ;
FUNCTION        FCOPYC  = COPYC   (Y,  BRC), QUEUE   (QUEC,  16)  ;
FUNCTION        FMUL    = MUL     (XY      ), QUEUE   (QUEM,  16)  ;
FUNCTION        FNOPSC  = NOPSC   (XY      )                       ;
FUNCTION        FORCBN  = OR      (CNOP    ), RDCYCS (FFFFHC, 1)   ;
FUNCTION        FRDMIN  = RDCYCS  (MFFFFH, 1)                     ;
FUNCTION        FSHL1   = SHL,                QUEUE   (QUES1, 16)  ;
FUNCTION        FSHL2   = SHL     (Y       ), QUEUE   (QUES2, 16)  ;
FUNCTION        FSUB    = SUB     (OVF     ), QUEUE   (QUES,  16)  ;
FUNCTION        FWAIT   = NOP                                       ;
;
;*****************************************************************
;                        MEMORY
;*****************************************************************
;
MEMORY  FFFFHC  =       0FFFFH.C                                    ;
MEMORY  MFFFFH  =       -0FFFFH                                     ;
MEMORY  QUEAE   =       AREA (16)                                   ;
MEMORY  QUEAM   =       AREA (16)                                   ;
MEMORY  QUEC    =       AREA (16)                                   ;
MEMORY  QUEE    =       AREA (16)                                   ;
MEMORY  QUEM    =       AREA (16)                                   ;
MEMORY  QUES    =       AREA (16)                                   ;
MEMORY  QUES1   =       AREA (16)                                   ;
MEMORY  QUES2   =       AREA (16)                                   ;
;
;*****************************************************************
;                        START
;*****************************************************************
;
START                                                               ;
DATA    EXEC    (EXPMUL,        MNTA,   MAVAL)                      ;
DATA    EXEC    (EXPMUL,        MNTB,   MBVAL)                      ;
DATA    EXEC    (EXPMUL,        EXPA,   EAVAL)                      ;
DATA    EXEC    (EXPMUL,        EXPB,   EBVAL)                      ;
;
END                                                                 ;
```

5.3 Floating-Point Addition

5.3.1 Algorithm

In the addition of exponential functions, the coefficients of different terms are added as shown in Equation 5-5, provided the terms have the same base and exponent.

$$(\chi \times 2^n) + (y \times 2^n) = (\chi + y) \times 2^n \quad \text{(Eq. 5-5)}$$

Therefore, to perform floating-point addition as shown in Equation 5-2, the expression is modified by multiplying the mantissa of the term having the smaller of the exponents with a scaling factor, so that both terms contain exponents of the same power as shown in Equation 5-6.

$$(\chi \times 2^3) + (y \times 2^2) = (\chi + y \times 2^{-1}) \times 2^3 \quad \text{(Eq. 5-6)}$$

From Equation 5-2 it follows that mantissas a and b are fractional real numbers and exponents x and y are integers. These values are expressed in 17 bits, 16 numerical value bits and a sign bit.

First, exponents x and y of the terms are compared. The higher of these values is chosen to be the provisional exponent of the resultant data. Secondly, the difference between the exponents of the terms is obtained. This value is used as the number of shift operations to be performed on the mantissa of the term having the lower of the two exponents.

The mantissa of the lower exponent term is shifted and added to the other mantissa, while making sure that they have the same order of magnitude. The result of the addition is left-shifted until its MSB is '1'. This operation gives the resultant mantissa.

When either an overflow or an underflow occurs during the addition of mantissas, the result is logically OR'ed with 8000H so that the MSB is '1'. This will be the resultant mantissa.

If, after the addition of mantissas, the data has to be shifted to the left, the value is taken to be positive; if the shift is to the right, the value is taken to be negative. This value is subtracted from the provisional exponent data to give the resultant exponent value.

If the exponent subtraction results in either an overflow or an underflow, the result of the subtraction is replaced with +FFFFH or -FFFFH, respectively.

If the resulting mantissa value is 0, the final exponent value is replaced with -FFFFH, its lowest possible value.

5.3.2 Allowable Range of Parameter Values

<Startup Token-Defined Parameters>

MNTA .... Mantissa of the left-hand term (a)
MNTB .... Mantissa of the right-hand term (b)
EXPA .... Exponent of the left-hand term (x)
EXPB .... Exponent of the right-hand term (y)

Table 5-3 gives the allowable range of values for these parameters.

Table 5-3
Allowable Range of Parameter Values
(Floating-Point Addition)

PARAMETER NAME	ALLOWABLE RANGE	VALUE IN PROGRAM EXAMPLE
MNTA	— FFFFH ~ FFFFH	A59FH
MNTB	—FFFFH ~ FFFFH	9C13H
EXPA	— 65535 ~ 65535	— 19
EXPB	— 65535 ~ 65535	— 30

The decimal values of the left and right terms used in the example programs referred to in Table 5-3 are as follows:

Left term (a): 0.000001234
Right term (b): 0.0000000005678

5.3.3 Explanations of the Flow Graph

The execution of this program starts with the receipt of four startup tokens: mantissas a and b, and exponents x and y.

<Explanations of the Nodes>

FCPYC0

This node sets the C Bit of mantissa a to 0.

FCPYC1

This node sets the C Bit of mantissa b to 1.

FQCMP, FCPYC0

These nodes compare the exponent values x and y, select the larger of the two values, and set its C Bit to 0.

FSUB1

This node subtracts exponent x from exponent y, and determines the number of shift operations to be performed. As a result of the subtraction, it sets the C Bit to 1 if the number is negative, 0 otherwise. The value of the C Bit set in this step will determine the manner in which either coefficient value a or b is to be shifted.

FRCMP1

This node compares the positive number of shift operations determined in the FSUB1 node with 10H (16). The lesser of these values is used as the number of shift operations to be performed on mantissa a.

FRCMP2

This node compares the negative number of shift operations determined in the FSUB1 node with -10H.C (-10H with C Bit=1). If the number is greater than -10H, it is used as the number of shift operations to be performed. If it is less than -10H, -10H.C is used as the number of shift operations to be performed on mantissa b.

FSHR

This node performs shift operations when the C Bit of the token transmitted via link X9 is 0. (Exponent value x < y).

FSHL1

This node performs shift operations when the C Bit of the token transmitted via X10 is 1. (Exponent value x > y).

FADDSC

This node adds the adjusted mantissas and determines the number of shift operations to be performed on the result.

FSHL2, FRNOP, FOR

These nodes shift the addition data. When an overflow or an underflow occurs, the result is shifted 1 bit to the right, and the MSB is set to '1' through a logical OR operation. Otherwise, the shifting is done to the left so that the

value of "1" will be brought over to the MSB.

FSUB2

This node subtracts from the provisional exponent value the number of shift operations performed on the mantissa addition value.

FRCMP3

If the value of the mantissa is 0, this node sets the C Bit of the token to 1.

FQCPYC, FWAIT, FRDMIN

These nodes replace the resultant exponent value with -FFFFH when the mantissa is 0 (when the C Bit is 1).

FANS

This node sends the resultant mantissa and exponent values to the host system.

FORCBN

When either an overflow or an underflow occurs during the computation of an exponent value, this node replaces the value with +FFFFH if the condition is an overflow, and with -FFFFH if the condition is an underflow.

Figure 5-4
Floating-Point Addition Flow Graph

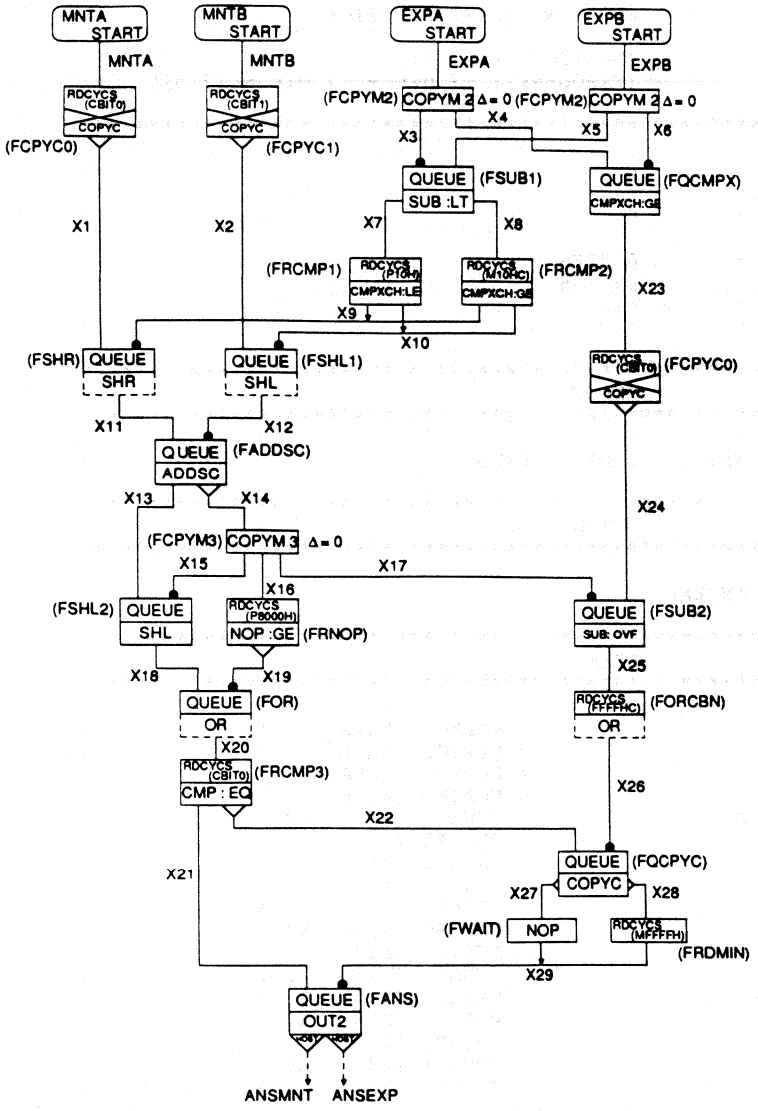


Figure 5-5
Floating-Point Addition Source Program

```
;*****************************************************************
;
;                       FLOATING-POINT ADDITION
;
;                       (MANTISSA : 17 bit )
;                       (EXPONENT : 17 bit + 17 bit = 17 bit)
;
;*****************************************************************
;
MODULE  EXPADD  =       8                                           ;
;
EQUATE  HOST    =       0                                           ;
;
EQUATE  MAVAL   =       0A59FH                                      ;
EQUATE  MBVAL   =       09C13H                                      ;
EQUATE  EAVAL   =       -19                                         ;
EQUATE  EBVAL   =       -30                                         ;
;
;*****************************************************************
;                       INPUT
;*****************************************************************
;
INPUT   MNTA,   MNTB,   EXPA,   EXPB                                ;
;
;*****************************************************************
;                       OUTPUT
;*****************************************************************
;
OUTPUT  ANSMNT, ANSEXP                                              ;
;
;*****************************************************************
;                       LINK
;*****************************************************************
;
LINK    X1                      = FCPYC0 (MNTA     )                ;
LINK    X2                      = FCPYC1 (MNTB     )                ;
LINK    X3,     X4              = FCPYM2 (EXPA     )                ;
LINK    X5,     X6              = FCPYM2 (EXPB     )                ;
LINK    X7,     X8              = FSUB1  (X5,   X3 )                ;
LINK    X9,     X10             = FRCMP1 (X7       )                ;
LINK    X9,     X10             = FRCMP2 (X8       )                ;
LINK    X11                     = FSHR   (X1,   X9 )                ;
LINK    X12                     = FSHL1  (X2,   X10)                ;
LINK    X13,    X14             = FADDSC (X11,  X12)                ;
LINK    X15,    X16,    X17     = FCPYM3 (X14      )                ;
LINK    X18                     = FSHL2  (X13,  X15)                ;
LINK    X19                     = FRNOP  (X16      )                ;
LINK    X20                     = FOR    (X18,  X19)                ;
LINK    X21,    X22             = FRCMP3 (X20      )                ;
LINK    X23                     = FQCMPX (X4,   X6 )                ;
LINK    X24                     = FCPYC0 (X23      )                ;
```

Figure 5-5 (cont.)
Floating-Point Addition Source Program

```
LINK    X25                     = FSUB2  (X24,  X17)          ;
LINK    X26                     = FORCBN (X25      )          ;
LINK    X27,    X28             = FQCPYC (X22,  X26)          ;
LINK    X29                     = FWAIT  (X27      )          ;
LINK    X29                     = FRDMIN (X28      )          ;
LINK    ANSMNT, ANSEXP          = FANS   (X21,  X29)          ;
;
;*****************************************************************
;                       FUNCTION
;*****************************************************************
;
FUNCTION        FADDSC  = ADDSC  (XY    ), QUEUE  (QUEA,  16)   ;
FUNCTION        FANS    = OUT2 (HOST,0,0), QUEUE  (QUEE,  16)   ;
FUNCTION        FCPYC0  = COPYC  (Y, XCH), RDCYCS (CBIT0,  1)   ;
FUNCTION        FCPYC1  = COPYC  (Y, XCH), RDCYCS (CBIT1,  1)   ;
FUNCTION        FCPYM2  = COPYM  (2,   0)                       ;
FUNCTION        FCPYM3  = COPYM  (3,   0)                       ;
FUNCTION        FOR     = OR     (CNOP  ), QUEUE  (QUEOR, 16)   ;
FUNCTION        FORCBN  = OR     (CNOP  ), RDCYCS (FFFFHC, 1)   ;
FUNCTION        FRDMIN  = RDCYCS (MFFFFH, 1)                    ;
FUNCTION        FQCMPX  = CMPXCH (GE    ), QUEUE  (QUECP, 16)   ;
FUNCTION        FQCPYC  = COPYC  (Y, BRC), QUEUE  (QUEC,  16)   ;
FUNCTION        FRCMP1  = CMPXCH (XX, LE), RDCYCS (P10H,   1)   ;
FUNCTION        FRCMP2  = CMPXCH (XX, GE), RDCYCS (M10HC,  1)   ;
FUNCTION        FRCMP3  = CMP    (XY, EQ), RDCYCS (CBIT0,  1)   ;
FUNCTION        FRNOP   = NOP    (Y,  GE), RDCYCS (P8000H, 1)   ;
FUNCTION        FSHL1   = SHL    (CNOP  ), QUEUE  (QUES1, 16)   ;
FUNCTION        FSHL2   = SHL,             QUEUE  (QUES2, 16)   ;
FUNCTION        FSHR    = SHR    (CNOP  ), QUEUE  (QUES3, 16)   ;
FUNCTION        FSUB1   = SUB (X, LT, BRC),QUEUE  (QUEB1, 16)   ;
FUNCTION        FSUB2   = SUB    (OVF   ), QUEUE  (QUEB2, 16)   ;
FUNCTION        FWAIT   = NOP                                   ;
;
;*****************************************************************
;                       MEMORY
;*****************************************************************
;
MEMORY  CBIT0   =       0                                       ;
MEMORY  CBIT1   =       0.C                                     ;
MEMORY  FFFFHC  =       0FFFFH.C                                ;
MEMORY  M10HC   =       -0010H.C                                ;
MEMORY  MFFFFH  =       -0FFFFH                                 ;
MEMORY  P10H    =       0010H                                   ;
MEMORY  P8000H  =       8000H                                   ;
MEMORY  QUEA    =       AREA (16)                               ;
MEMORY  QUEB1   =       AREA (16)                               ;
MEMORY  QUEB2   =       AREA (16)                               ;
MEMORY  QUEC    =       AREA (16)                               ;
MEMORY  QUECP   =       AREA (16)                               ;
MEMORY  QUEE    =       AREA (16)                               ;
MEMORY  QUEOR   =       AREA (16)                               ;
```

Figure 5-5 (cont.)
Floating-Point Addition Source Program

```
MEMORY   QUES1    =        AREA (16)                                    ;
MEMORY   QUES2    =        AREA (16)                                    ;
MEMORY   QUES3    =        AREA (16)                                    ;
;
;*****************************************************************
;                          START
;*****************************************************************
;
START                                                                   ;
DATA     EXEC     (EXPADD,          MNTA,    MAVAL)                   ;
DATA     EXEC     (EXPADD,          MNTB,    MBVAL)                   ;
DATA     EXEC     (EXPADD,          EXPA,    EAVAL)                   ;
DATA     EXEC     (EXPADD,          EXPB,    EBVAL)                   ;
;
END                                                                     ;
```

Chapter 6

Distance Squared Calculation

6.1 Processing Explained

In this program, the distance squared calculations are performed using Equation 6-1 and data lengths as indicated in Table 6-1.

$$\sum_{i=0}^{n} (x_i - y_i)^2 \qquad \text{(Eq. 6-1)}$$

Table 6-1
Data Lengths Used in Distance Calculations

VARIABLE X	VARIABLE Y	COMPUTATION RESULT
17 BITS	17 BITS	33 BITS

(INCLUDES SIGN BIT)

6.2 Algorithm

Expansion of Equation 6-1 yields a sum of terms 0 through n, each term being the square of the difference of x_i and y_i.

$$\sum_{i=0}^{n} (x_i - y_i)^2 = (x_0 - y_0)^2 + (x_1 - y_1)^2 + \cdots + (x_n - y_n)^2 \qquad \text{(Eq. 6-2)}$$

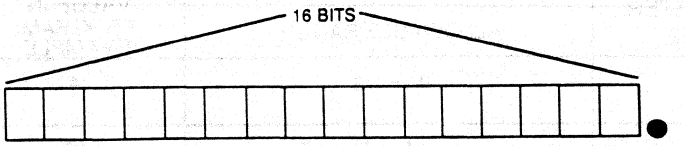
The results of the computations are expressed in 33 bits, 32 numerical value bits and a sign bit.

The difference between variables x_i and y_i is represented in 17 bits, including a sign bit; the square of the difference is represented in 33 bits. The square of each difference is divided into high and low words, to be added to other squared values. In performing these additions, a carry from a low word must be added to the corresponding high word. Any carry from the addition of high words, however, is truncated.

The data in variables x_i and y_i are expressed in 16 numerical value bits and a sign bit. The decimal point may be at an arbitrary position, provided all the data values have the same decimal point location in order to accommodate the addition operations performed in this program.

Figure 6-1 shows the location of the decimal point when variables x and y are integers.

Figure 6-1
Location of Decimal Point for Integers
in Distance Calculation



6.3 Allowable Range of Parameter Values

<Assembler-Coded Parameters>

I .... Number of input data values
CV1 .... Number of COVNO command counts on the high word side
CV2 .... Number of COVNO command counts on the low word side

<Startup Token-Defined Parameters>

XD .... Input data for variable x
YD .... Input data for variable y

Table 6-2 shows the allowable values of these parameters.

Table 6-2
Allowable Range of Parameter Values

PARAMETER NAME	ALLOWABLE RANGE	VALUE IN PROGRAM EXAMPLE
I	1 ~ 16	4
CV1	4 x I — 3	13
CV2	2 x I — 1	7
XD	—FFFFH ~ FFFFH	000FH 00FFH 0FFFH FFFFH
YD	—FFFFH ~ FFFFH	0005H 0055H 0555H 5555H

In the example programs referred to in Table 6-2, the decimal point for variables XD and YD is located to the right the LSB.

6.4 Explanations of the Flow Graph

In this program the data values for variables x_i and y_i are used in startup tokens. The maximum allowable value of i is 16, due to the fact that the maximum amount of DM area that can be allocated by the QUEUE command is 16 words.

Therefore, up to 16 x_i and 16 y_i data values can be processed continuously.

<Explanations of the Nodes>

FSUB, FMUL

These nodes determine the square of the difference between variable x_i and y_i. The squared data values are expressed in 33 bits, divided into high and low words.

FCONV2, FADD2

These nodes add the low words of the squared data values. They send the high word (carry data) of the result of the addition to the high-word processing nodes.

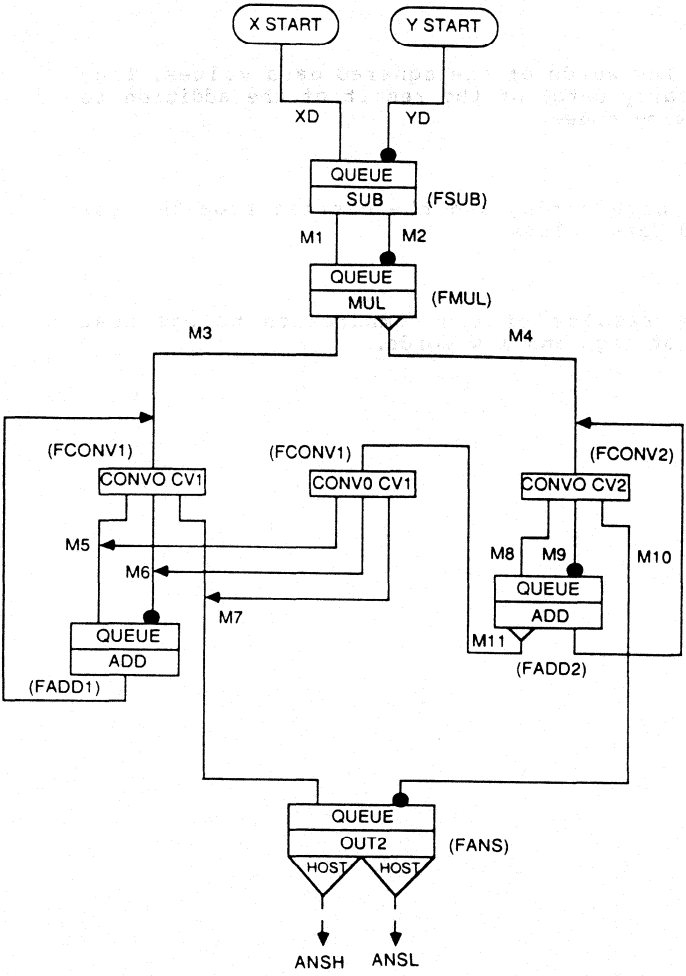
FCONV1, FADD1.

These nodes add the high words, and the carries from the low words, to the squared data values.

FANS

This node sends the results of the computation to the host system, in the order of high and low words.

Figure 6-2
Distance Squared Calculation Flow Graph



6.5 Tips on Preparing Flow Graphs

The node FCONV1 shown in Figure 6-2 has two distinct input arcs with FTRC = 0, sharing the same output ID of the node. Ideally it should share the output ID's of the nodes FMUL and FADD2, as shown in Figure 6-3. However, since the Y output of FADD2 cannot be a first ID (Y output must be ID+1), the arc name is changed so that the output ID and the node for FCONV1 can be shared.

This arrangement permits the Function Table Temporary (FTT) field of the FCONV1 instructions to be shared, and for normal distribution of the output tokens.

Figure 6-3
Unworkable Distance Squared Calculation Flow Graph

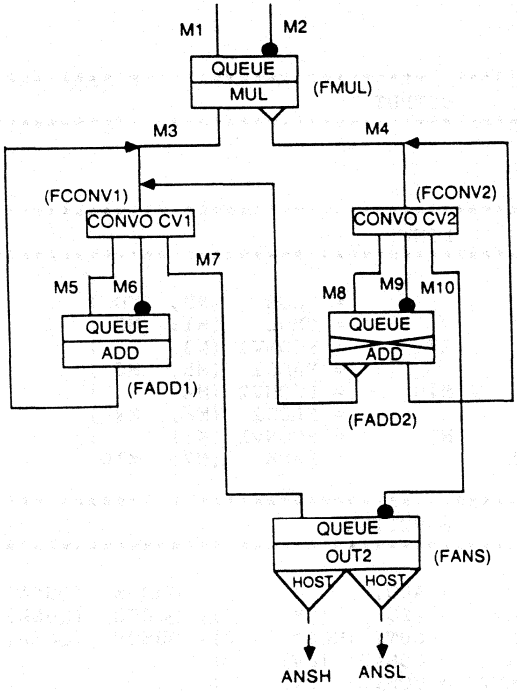


Figure 6-4
Distance Squared Calculation Source Program

```
;*****************************************************************
;
;                        DISTANCE CALCULATION
;
;                        (17 bit x 17 bit = 33 bit)
;
;*****************************************************************
;
MODULE   LENGTH  =       8                                          ;
;
EQUATE   HOST    =       0                                          ;
EQUATE   I       =       4                                          ;
EQUATE   CV1     =       4*I-3                                      ;
EQUATE   CV2     =       2*I-1                                      ;
;
;*****************************************************************
;                        INPUT
;*****************************************************************
;
INPUT    XD,     YD                                                 ;
;
;*****************************************************************
;                        OUTPUT
;*****************************************************************
;
OUTPUT   ANSH,   ANSL                                               ;
;
;*****************************************************************
;                        LINK
;*****************************************************************
;
LINK     M1,     M2              = FSUB   (XD,  YD )                ;
LINK     M3,     M4              = FMUL   (M1,  M2 )                ;
LINK     M5,     M6,     M7      = FCONV1 (M3      )                ;
LINK     M3                      = FADD1  (M5,  M6 )                ;
LINK     M8,     M9,     M10     = FCONV2 (M4      )                ;
LINK     M4,     M11             = FADD2  (M8,  M9 )                ;
LINK     M5,     M6,     M7      = FCONV1 (M11     )                ;
LINK     ANSH,   ANSL            = FANS   (M7,  M10)                ;
;
;*****************************************************************
;                        FUNCTION
;*****************************************************************
;
FUNCTION         FADD1   = ADD,                  QUEUE  (QUEA1, 16)  ;
FUNCTION         FADD2   = ADD    (XY     ), QUEUE  (QUEA2, 16)  ;
FUNCTION         FANS    = OUT2 (HOST, 0,0), QUEUE  (QUEA,  16)  ;
FUNCTION         FCONV1  = CONVO  (CV1    )                      ;
FUNCTION         FCONV2  = CONVO  (CV2    )                      ;
FUNCTION         FMUL    = MUL    (XY     ), QUEUE  (QUEM,  16)  ;
FUNCTION         FSUB    = SUB    (XX     ), QUEUE  (QUES,  16)  ;
```

Figure 6-4 (Con't)
Distance Calculation Source Program

```
;
;*****************************************************************
;                          MEMORY
;*****************************************************************
;
MEMORY   QUEA     =        AREA (16)                                ;
MEMORY   QUEA1    =        AREA (16)                                ;
MEMORY   QUEA2    =        AREA (16)                                ;
MEMORY   QUEM     =        AREA (16)                                ;
MEMORY   QUES     =        AREA (16)                                ;
;
;*****************************************************************
;                          START
;*****************************************************************
;
START                                                               ;
DATA     EXEC     (LENGTH,          XD,      0000FH)               ;
DATA     EXEC     (LENGTH,          YD,      00005H)               ;
DATA     EXEC     (LENGTH,          XD,      000FFH)               ;
DATA     EXEC     (LENGTH,          YD,      00055H)               ;
DATA     EXEC     (LENGTH,          XD,      00FFFH)               ;
DATA     EXEC     (LENGTH,          YD,      00555H)               ;
DATA     EXEC     (LENGTH,          XD,      0FFFFH)               ;
DATA     EXEC     (LENGTH,          YD,      05555H)               ;
;
END                                                                 ;
```

Chapter 7

Fast Fourier Transform (FFT)

7.1 Processing Explained

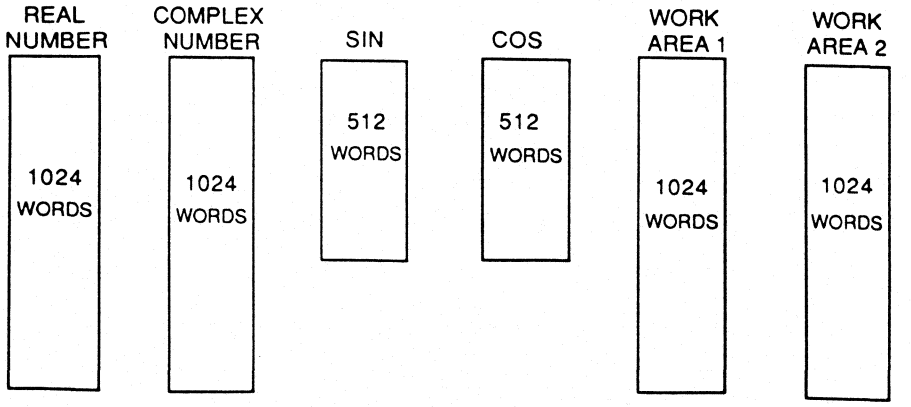
This program determines the frequency components of 1,024 sampled data points consisting of real and complex parts, using a Fast Fourier Transform (FFT) based on radix-2 decimation-in-frequency techniques.

Fast Fourier Transforms, as well as inverse FFTs, find applications in the extraction of frequency information, filtering such as noise removal, and voice synthesis and reconstruction. Also, in image processing involving planar and stereo-configuration data, two and higher dimensional FFTs are used.

7.2 Algorithm

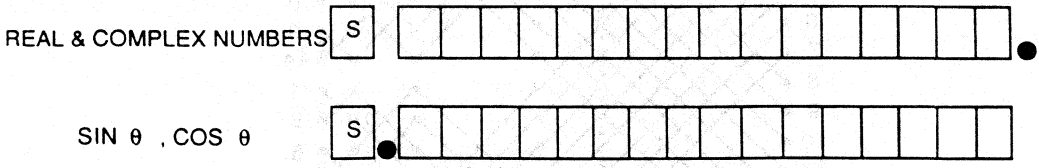
This program assumes that 1,024 sampled data points, each of which consists of a real part word and a complex part word, are to be Fast Fourier Transformed using radix-2. These data are to be stored in the IM prior to the start of processing. The IM is initialized with sine and cosine lookup tables (512 words each) and a 1,024 word work area (Figure 7-1).

Figure 7-1
Data and Work Area Layout of the IM



Each word of a data point is represented in 17 bits, 16 numerical value bits and a sign bit. Of these data points, the real and complex data values are treated as integers having the decimal point at the right of the LSB. The sine and cosine data values are treated as fractional numbers with absolute value less than 1 and with the decimal point at the left of the MSB.

Figure 7-2
Number Representation in FFT Program



The sine and cosine lookup tables contain, respectively, sin θ and cos θ values for θ in the range $0 \geq \theta > -\pi$ divided into 512 equal intervals. For sin $\theta = -1$ ($\theta = -\pi/2$) the value stored in the table is -FFFFH, and for cos $\theta = 1$ ($\theta = 0$) it is set to FFFFH.

The 2,048-word work area is used alternately with the real/complex number areas for the purpose of reading and writing the intermediate results of computations and for substituting the data with final computation results.

Two types of Fast Fourier Transform algorithms are a decimation-in-time algorithm, in which the sampled data are divided into odd and even parts, and a decimation-in-frequency algorithm, in which the sampled data are divided into first and second halves. This program uses the latter type of algorithm.

Figure 7-3
16-Point FFT Decimation-in-Frequency Data Flow Graph

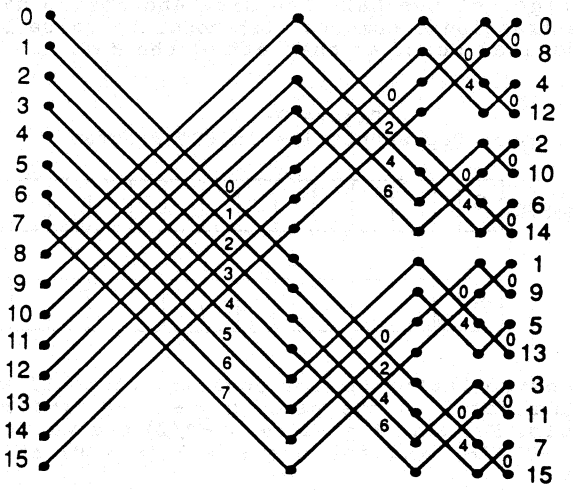


Figure 7-4
Butterfly Operation

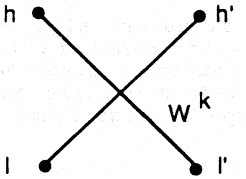


Figure 7-3 illustrates the decimation-in-frequency algorithm, using 16 sampled data points. Figure 7-4 shows the "butterfly" computation and represents the fundamental operation used in this algorithm. The butterfly operation is described by Equation 7-1.

$$h' = h + l$$

$$l' = (h - l)\ W^k \qquad \text{(Eq. 7-1)}$$

h : First half of input data
l : Second half of input data
h' : First half of output data
l' : Second half of output data
W^k : Multiplication (twiddle) factor

In a butterfly computation, part of the sampled data, divided into first (h) and second (l) halves, are extracted and operated upon according to Equation 7-1.

The number of butterfly operations performed in each stage is equal to one-half the number of the sampled data points. The total number of stages needed for all of the sampled data is $\log_2 N$, where N is the total number of sampled data points.

The procedure can be illustrated as follows, using Figure 7-3 as an example:

Since there are 16 sampled data points (N = 16), the total number of stages is 4 ($\log_2 16$), with eight butterfly operations performed per stage. Input data h and l for the butterfly operation in stage 1 will be, respectively, the first eight and second eight sampled data points. In the second stage, the results of the computation in stage 1 are divided into first and second halves, each consisting of eight data points. These are further subdivided into first and second halves, each consisting of four data points. The first four data points are used as h, the second four data points as l. Similarly, in stages 3, 4, and all subsequent stages, the number of subdivisions into first and second halves is doubled for each stage. In each case the first half of data is used as h, the second half as l. The destination of the output of the results of computation h and l will be located at the same position as the input data (in-place computations). Therefore, input data h and l and output data h' and l' for stage 1 will be 0 & 8; 1 & 9; 2 & 10; ... and 7 & 15. Those for stage 2 will be 0 & 4; 1 & 5; 2 & 6;... and 12 & 15. Those for stage 3 will be 0 & 2; 1 & 3; 4 & 6; ...; 13 & 15. And those for stage 4 will be 0 & 1; 2 & 3; 3 & 4; ... and 14 & 15.

Since the real and complex parts are calculated separately in this FFT, Equation 7-1 can be replaced by Equation 7-2:

$$Rh' = Rh + Rl$$

$$Ih' = Ih + Il$$

$$Rl' = (Rh - Rl)\cos\theta + (Ih - Il)\sin\theta \qquad \text{(Eq. 7-2)}$$

$$Il' = (Ih - Il')\cos\theta - (Rh - Rl)\sin\theta$$

Rh ... real part, first half of input data
Rl ... real part, second half of input data
Rh' ... real part, first half of output data
Rl' ... real part, second half of output data
Ih ... complex part, first half of input data
Il ... complex part, second half of input data
Ih' ... complex part, first half of output data
Il' ... complex part, second half of output data

In the FFT, the results of computation for a stage obtained by means of Equation 7-2 are used as input data for the next stage, to be repeated for a total of $\log_2 N$ stages. For the program given in this chapter, 10 stages ($\log_2 1024$) are required. To repeat such processing, the output data must have the decimal point at the same position as as the initial input data.

Therefore, the decimal point of Rh', Ih', Rl', and Il' obtained at the different stages must be to the right of the LSB, identical to the decimal point position of Rh, Ih, Rl, and Il. To ensure this outcome, in additions and subtractions the lower 16 bits are taken to be the results of the computation, while in multiplications the higher 16 bits are used as results.

In generating the read and write addresses for these sampled data points, the algorithm depicted in Figure 7-3 is not used directly as a process flow because the order in which the read/write addresses are to be generated will differ for each stage, making the process flow unduly complicated.

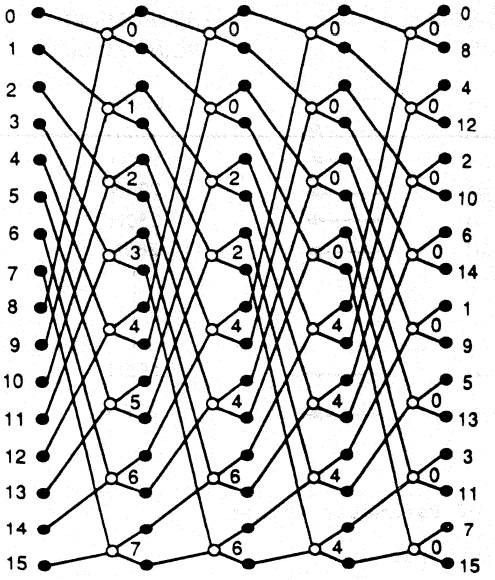
To avoid this problem, the destination of output of the computational results can be altered as shown in Figure 7-5, so that the same order of address generation can be used for all the stages. For this purpose, the exponent 'k' of the twiddle factor W^k can be determined by assigning values 0 through (N/2)-1 to the N/2 computations per stage, starting from the top, and by masking as many lower bits as the number of stages involved in performing computations on these values, minus one.

Real and complex number read addresses are generated by dividing each area into a first and second half, and by taking the addresses of the divided areas in sequence, starting from the top.

Sine and cosine read addresses are produced by generating 512 (0 to 511) values for each step, masking as many lower number bits as the stage number minus one (for example, for stage 1 no bits are masked, for stage 2 the LSB is masked, for stage 3 the two lowest bits are masked, etc.) and then adding the starting addresses of the tables. The SIN/COS tables are provided with sin θ and cos θ absolute values less than 1 for radian θ ($= \pi k/512$) corresponding to the 'k' in twiddle factor W^k.

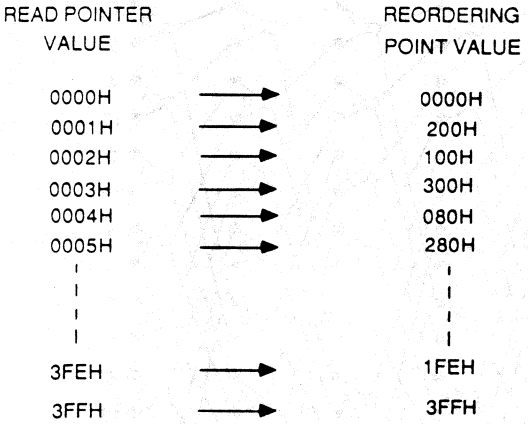
The write addresses for the real and complex parts of the results of computation are generated by using every other word of the first and second half data addresses, starting from the starting address of each write area.

Figure 7-5
A Radix-2 16-Point Decimation-in-Frequency FFT Algorithm
Having the Same Geometry for Each Stage



Final results of the computation obtained through decimation-in-frequency and input alignment are not in frequency sequence as shown in Figure 7-3. To put the data in frequency sequence, the pointer values of the data in which the final results are stored must be replaced with bit-reversed (pointing) values. This procedure, however, will work only when the number of sampled data values is a power of 2. Thus, since the number of sampled data used in this program is 1,024, or 2^{10}, the data reordering can be accomplished in 10-bit reversals.

Figure 7-6
Bit Reversals for 10-Bit Number



7.3 Allowable Range of Parameter Values

<Assembler-Coded Parameters>

READADR .... Real/complex read starting address for each stage
WRTRAD .... Real write starting address for each stage
WRTIAD .... Complex write starting address for each stage
TBLADR .... Sin/cos table starting address

<Startup Token-Defined Parameters>

STTKN .... Program startup token (arbitrary value)

The allowable range of values of these parameters is given in Table 7-1.

Table 7-1
Allowable Range of Parameter Values

PARAMETER NAME	ALLOWABLE RANGE	VALUE IN PROGRAM EXAMPLE
REDADR	0 ~ 65535	0 * 2048 **
WRTRAD	0 ~ 65535	2048 * 0 **
WRTIAD	0 ~ 65535	1024 * 3072 **
TBLADR	0 ~ 65535	4096
STTKN	−65535 ~ 65535	0

* = even number of stages
** = odd number of stages

7.4 Explanations of the Flow Graph

This program consists of the following 6 process flows:
- Generation of addresses and reading of real/complex data
- Generation of addresses and reading of SIN/COS data
- Generation of addresses and writing of real number data
- Generation of addresses and writing of complex number data
- Butterfly computations
- Rearranging final computation result data

The real/complex number, SIN/COS, and work 1 and work 2 areas are allocated in contiguous addresses, as shown in Figure 7-7. The flow graphs and source program examples use these program-specific values for setting parameter values that depend on area addresses.

The read and write areas for the real and complex numbers are assigned by alternately using the initial real/complex data storage areas and work 1 and work 2 areas.

The program is started when a startup token is input to the STTKN arc. The startup token triggers the generation of real/complex and SIN/COS read addresses, and real/complex write addresses.

The real/complex read addresses are generated by dividing each data area into a first and second half, and obtaining four addresses from each of the four divided areas.

The SIN/COS read addresses are generated by generating four addresses representing the increment by one starting from the table starting address for each stage, by masking as many bits as the number of computational steps minus one on the values so obtained, and by making copies, using the difference between the SIN and COS addresses (512) as the increment. In such a case, however, the lowest nine bits of the starting address of a table must be all 0.

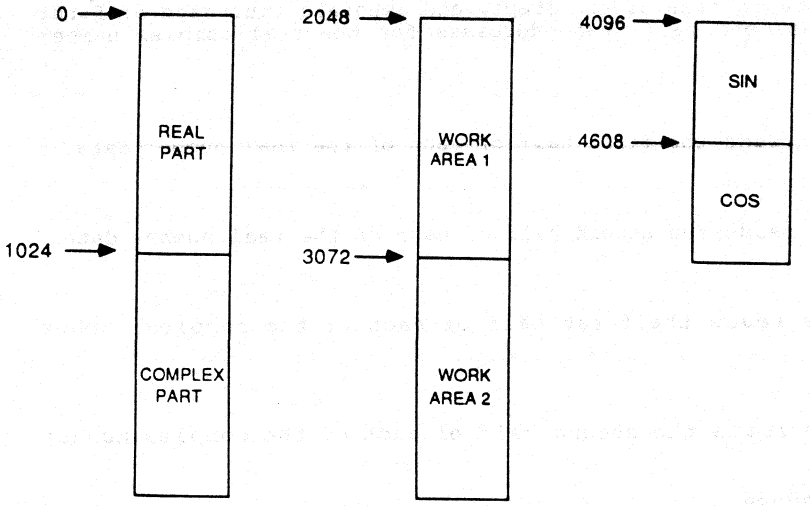
Eight each of real/complex write addresses are generated from the starting address of each area. Each of the write addresses generated is used as either a first half or a second half address for storing the computation results. A generated read address is used immediately to read the data. The data that has been read is input into the butterfly computation process to commence the computation operations.

Address generation operations are restarted by counting the number of COS data, the slowest input process, and by starting up the address generation processes after every fourth count.

For restarting stages 1 through 9 after the completion of computations, the second halves of 256 complex number data, obtained through butterfly computations, are counted twice in the FPICKA and FWBUF2 nodes, and the STTKN arc is started, thus repeating the address generation and butterfly computation processes. When the final computation in stage 10 is finished, the sorting of final computation result data is performed by starting the BRV arc.

When all processing is completed, a process termination token is sent to the host system.

Figure 7-7
Data Layout in the IM



The final computational results are stored in real and complex data areas. Sorting of these computational results is performed simultaneously on the real and complex parts of the results data. The results are stored in work areas 1 and 2.

Since each area is 1,024 words long, each data pointer is indicated in 10 bits. Sorting is performed by relocating the data pointed to by a pointer to a position indicated by a bit reversal of the pointer value. Write addresses are obtained by adding a write starting address to the bit-reversed values of the read pointers. However, since the data points are processed 16 bits at a time, to obtain the bit reversal of a 10-bit pointer the pointer value must be shifted 6 bits to the left prior to the bit reversing. Thus, to generate write addresses, 1,024 data points having an initial value of 0 and an increment of 64 are generated, and to each data point, after a bit reversal, the starting addresses of work areas 1 and 2 are added. The read addresses are used to generate addresses, using the starting addresses of the real and complex parts to read data.

<Explanations of the Nodes>

FCPYM4

This node activates the generation of read/write addresses.

FRREDA, FGEN1, FCPY4

These nodes read the starting read addresses of the read/complex number data in each of the steps, and generate four each of first half and second half read addresses for the real/complex number data.

FREDR1

This node reads the first half of each of the real number data.

FREDR2

This node reads the second half of each of the real number data.

FREDI1

This node reads the first half of each of the complex number data.

FREDI2

This node reads the second half of each of the complex number data.

FRMASK, FWMASK

These nodes set up the mask data for SIN/COS read addresses.

FRTBLA, FGEN2, FRAND, FCPY2

These nodes read the starting addresses of the SIN/COS data tables and generate four each of SIN/COS table read addresses for twiddle factor W^K.

FREDS

This node reads SIN data.

FREDC

This node reads COS data.

FRWRRA, FGEN3

These nodes read the write starting addresses of the real number data in each step, generating eight each of real number data write addresses.

FWRTR1

This node writes the first half (Rh') of the real number data created by the computations.

FWRTR2

This node writes the second half (Rl') of the real number data created by the computations.

FRWRIA, FGEN4

These nodes read the write starting addresses of complex number data in each step, generating 8 each of complex number data write addresses.

FWRTIl

This node writes the first half (Ih') of the complex number data created by the computations.

FWRTI2

This node writes the second half (Il') of the complex number data created by the computations.

FDIST2

This node separates real/complex number write addresses into first and second halves.

FCNTFF, FWBUF2

These nodes count the number of complex number second half write data to confirm completion of stage l computations.

FPICKA

This node restarts stage 2 and up. Upon completion of the stage l0 computations, it activates the data sorting processing.

FADDl

This node determines the real number first halves (Rh') resulting from butterfly computations.

FADD2

This node determines the complex number first halves (Ih') resulting from butterfly computations.

FSUBl, FSUB2, FMULl, FMUL3, FADD3

These nodes determine the real number second halves (Rl') resulting from butterfly computations.

(FSUBl, FSUB2,) FMUL2, FMUL4, FSUB3

These nodes determine the complex number second halves (Il'), resulting from butterfly computations.

FCNT4, FCPYM4

These nodes restart the read/write address generation process at the input of every four COS read data.

FRZERO, FGEN7, FCPY1

These nodes generate the real/complex number data read addresses of final computation results, eight addresses at a time, starting from the top.

FREDR

This node reads the real number data of final computation results.

FREDI

This node reads the complex number data of final computation results.

(FRZERO,) FBRV, FRADD1, FRADD2

These nodes determine the bit reversals of read pointers and generate eight each of sorted write addresses based on the work 1 and work 2 areas.

FWRTR

This node writes real number data to the work 1 area.

FWRTI

This node writes complex number data to the work 2 area.

FCNT8, FNOPXX

These nodes restart the generation of the next set of read/write addresses after the input of every eight complex data.

Figure 7-8a
FFT Flow Graph
Read Address Generation

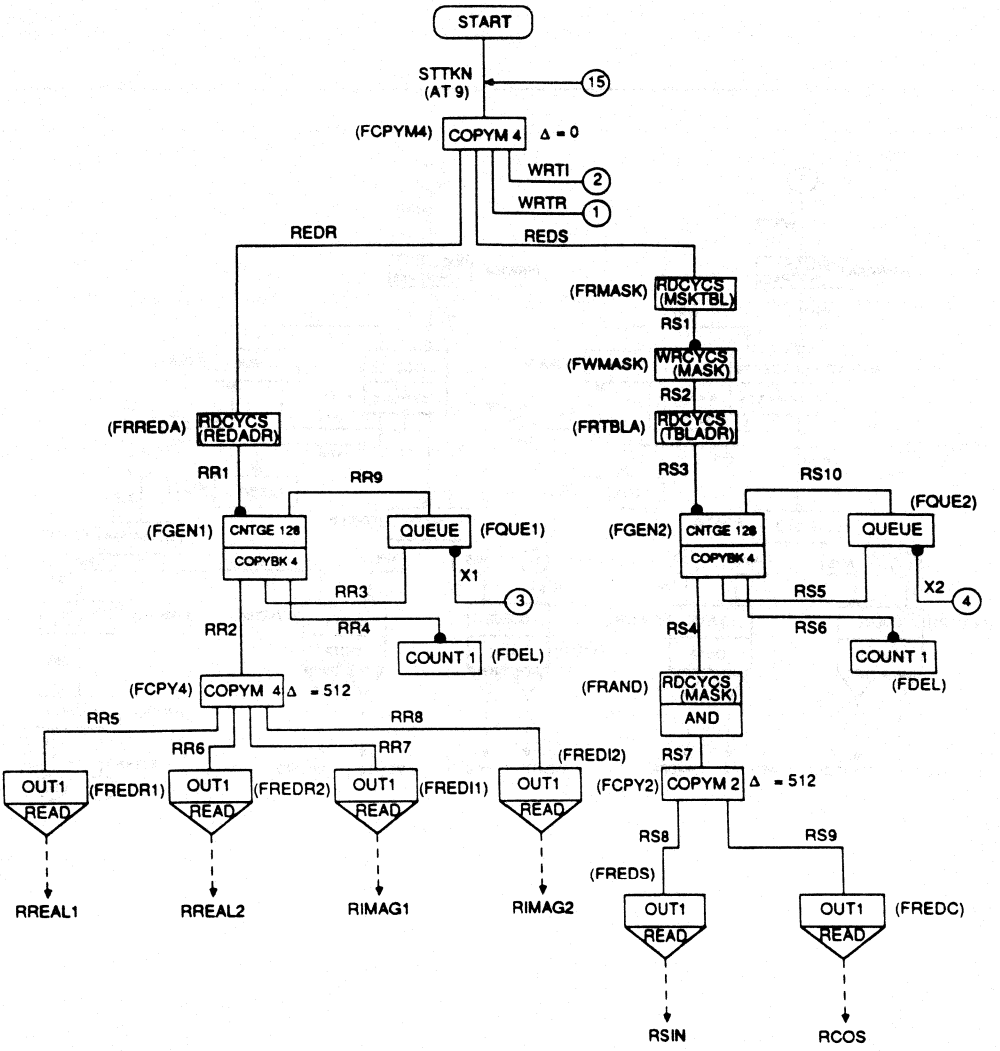


Figure 7-8b
FFT Flow Graph
Write Address Generation

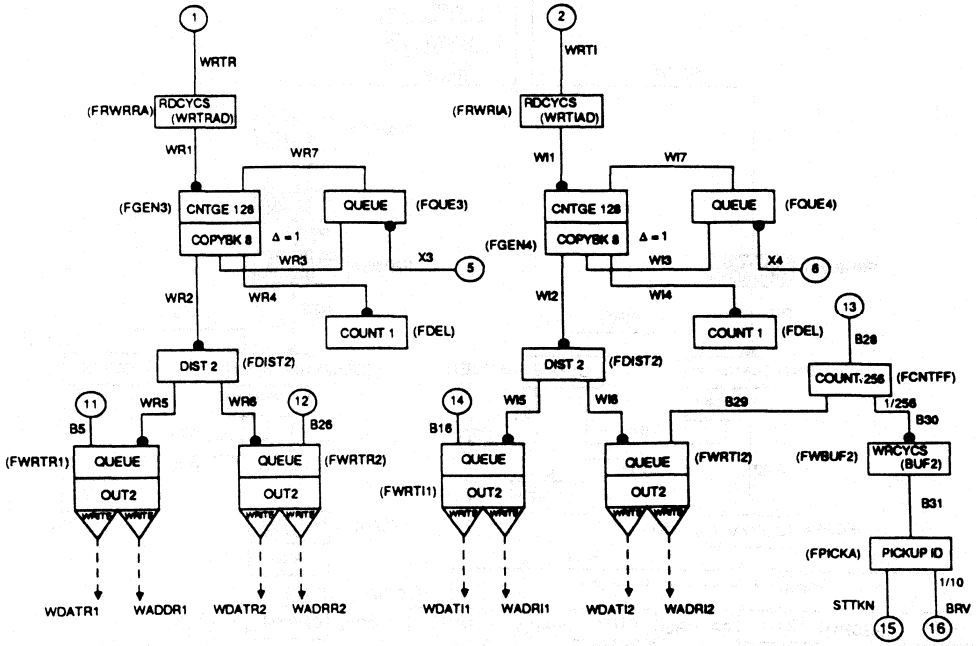


Figure 7-8c
FFT Flow Graph
Butterfly Computation

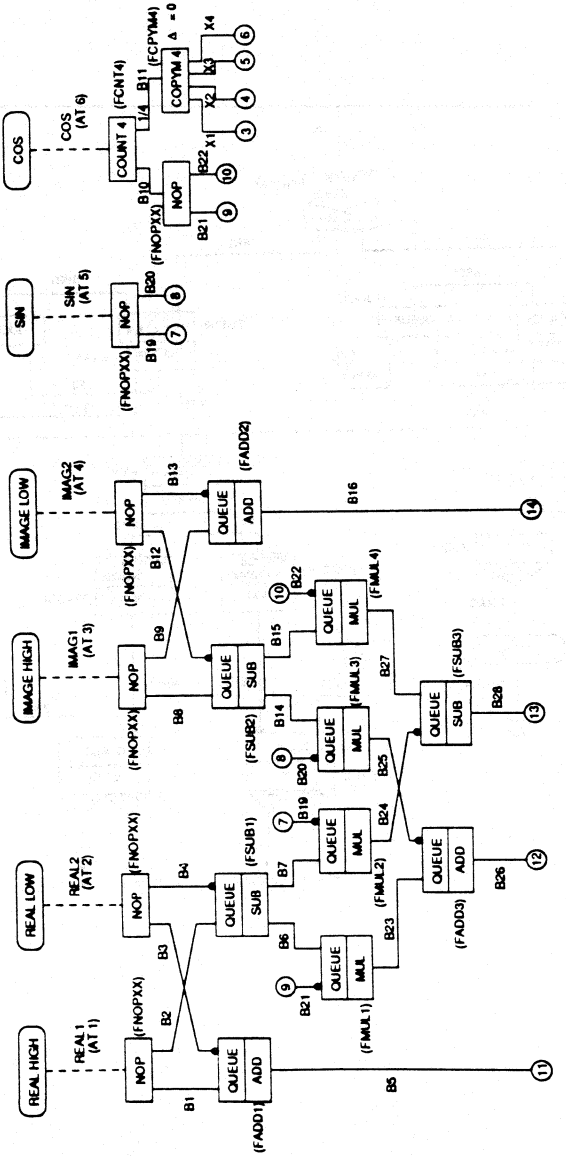


Figure 7-8d
FFT Flow Graph
Data Sorting

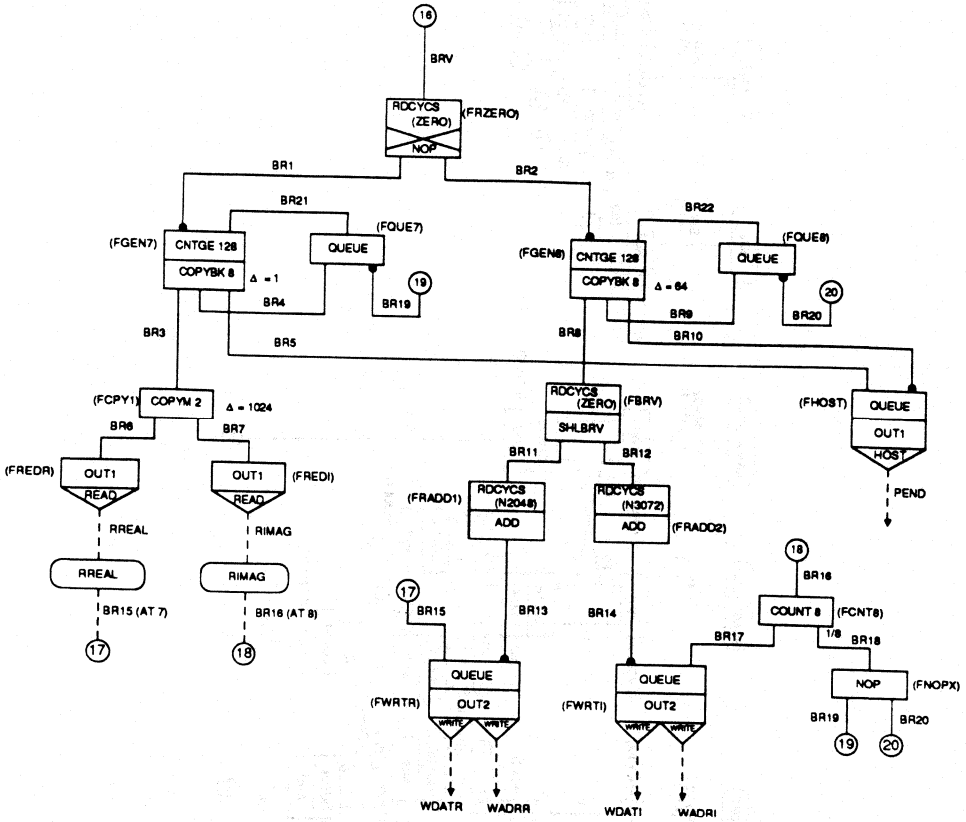


Figure 7-9 is a graph of the operating efficiency of LT and IM for the first 700 steps of stage 1. The arrows indicate the positions at which address generation is restarted.

The areas with low LT operating efficiency are where the system has to wait for the input of IM data. Items (1), (3), and (5) correspond to the boundaries between address generation and butterfly computation, at which the system has to wait for the input of data for butterfly computation. Items (2), (4), and (6) represent a state where the system waits for the input of the data to be multiplied with SIN/COS data. The wait state is caused by a delay in SIN/COS read address generation, as indicated by the IM operating efficiency graph.

Simulation studies indicate that the program requires 61.3 msec. processing time between the input of startup tokens and the writing of computational results from 10 stages, and 3.9 msec. for the sorting of final results, for a total of 65.2 msec.

Figure 7-9
LT and IM Operating Efficiencies under FFT

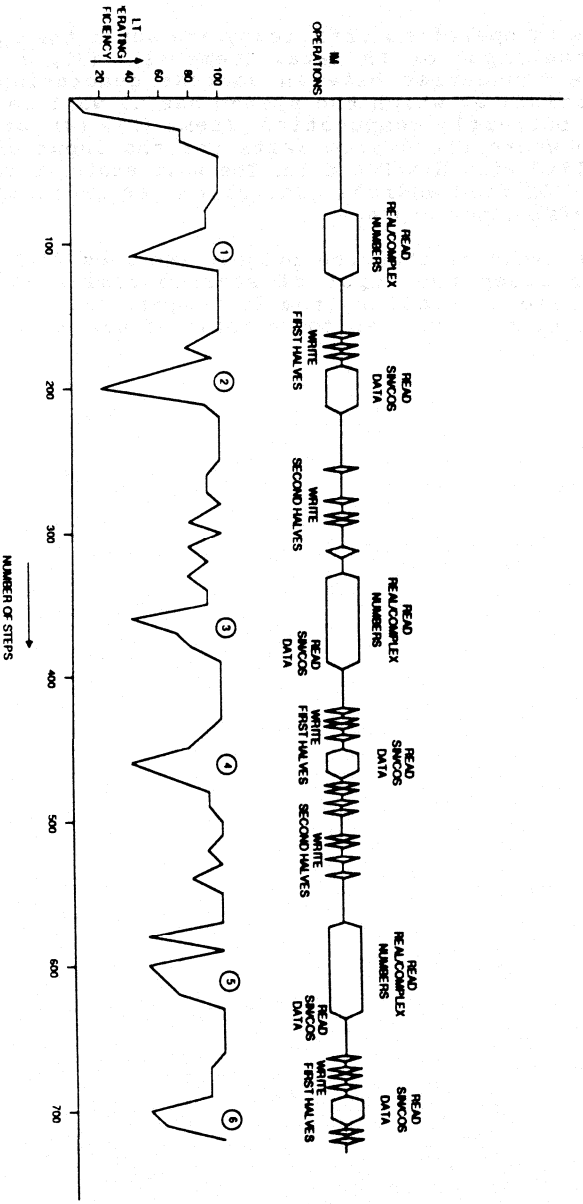


Figure 7-10
FFT Source Program

```
;*****************************************************************
;
;               FFT (FAST FOURIER TRANSFORM)
;
;*****************************************************************
;
MODULE  FFT1U   =       8                                       ;
;
EQUATE  HOST    =       0                                       ;
EQUATE  READ    =       4                                       ;
EQUATE  WRITE   =       5                                       ;
;
;*****************************************************************
;                       INPUT
;*****************************************************************
;
INPUT   STTKN   AT      9                                       ;
INPUT   REAL1   AT      1                                       ;
INPUT   REAL2   AT      2                                       ;
INPUT   IMAG1   AT      3                                       ;
INPUT   IMAG2   AT      4                                       ;
INPUT   SIN     AT      5                                       ;
INPUT   COS     AT      6                                       ;
INPUT   BR15    AT      7                                       ;
INPUT   BR16    AT      8                                       ;
;
;*****************************************************************
;                       OUTPUT
;*****************************************************************
;
OUTPUT  WDATR1, WADRR1, WDATR2, WADRR2, WDATI1, WADRI1, WDATI2  ;
OUTPUT  WADRI2                                                  ;
OUTPUT  RREAL1, RREAL2, RIMAG1, RIMAG2, RSIN,   RCOS            ;
OUTPUT  RREAL,  RIMAG,  WDATR,  WADRR,  WDATI,  WADRI           ;
OUTPUT  SOL,    PEND                                            ;
;
;*****************************************************************
;                       LINK
;*****************************************************************
;
LINK    REDR, REDS, WRTR, WRTI  = FCPYM4 (STTKN     )           ;
LINK    RR1                     = FRREDA (REDR      )           ;
LINK    RR2,    RR3,    RR4     = FGEN1  (RR9,  RR1 )           ;
LINK    RR5,  RR6,  RR7,  RR8   = FCPY4  (RR2       )           ;
LINK    RREAL1                  = FREDR1 (RR5       )           ;
LINK    RREAL2                  = FREDR2 (RR6       )           ;
LINK    RIMAG1                  = FREDI1 (RR7       )           ;
LINK    RIMAG2                  = FREDI2 (RR8       )           ;
LINK    RR9                     = FQUE1  (RR3,  X1  )           ;
LINK                            = FDEL   (    ,  RR4 )          ;
;
```

Figure 7-10 (cont.)
FFT Source Program

```
LINK    RS1                         = FRMASK (REDS       )          ;
LINK    RS2                         = FWMASK (    , RS1  )          ;
LINK    RS3                         = FRTBLA (RS2        )          ;
LINK    RS4,    RS5,    RS6         = FGEN2  (RS10, RS3  )          ;
LINK    RS7                         = FRAND  (RS4        )          ;
LINK    RS8,    RS9                 = FCPY2  (RS7        )          ;
LINK    RSIN                        = FREDS  (RS8        )          ;
LINK    RCOS                        = FREDC  (RS9        )          ;
LINK    RS10                        = FQUE2  (RS5,  X2   )          ;
LINK                                = FDEL   (   ,  RS6  )          ;
;
LINK    WR1                         = FRWRRA (WRTR       )          ;
LINK    WR2,    WR3,    WR4         = FGEN3  (WR7,  WR1  )          ;
LINK    WR5,    WR6                 = FDIST2 (   ,  WR2  )          ;
LINK    WDATR1, WADRR1              = FWRTR1 (B5,   WR5  )          ;
LINK    WDATR2, WADRR2              = FWRTR2 (B26,  WR6  )          ;
LINK    WR7                         = FQUE3  (WR3,  X3   )          ;
LINK                                = FDEL   (   ,  WR4  )          ;
;
LINK    WI1                         = FRWRIA (WRTI       )          ;
LINK    WI2,    WI3,    WI4         = FGEN4  (WI7,  WI1  )          ;
LINK    WI5,    WI6                 = FDIST2 (   ,  WI2  )          ;
LINK    WDATI1, WADRI1              = FWRTI1 (B16,  WI5  )          ;
LINK    WDATI2, WADRI2              = FWRTI2 (B29,  WI6  )          ;
LINK    WI7                         = FQUE4  (WI3,  X4   )          ;
LINK                                = FDEL   (   ,  WI4  )          ;
;
LINK    B1,     B2                  = FNOPXX (REAL1      )          ;
LINK    B3,     B4                  = FNOPXX (REAL2      )          ;
LINK    B5                          = FADD1  (B1,   B3   )          ;
LINK    B6,     B7                  = FSUB1  (B2,   B4   )          ;
LINK    B8,     B9                  = FNOPXX (IMAG1      )          ;
LINK    B12,    B13                 = FNOPXX (IMAG2      )          ;
LINK    B14,    B15                 = FSUB2  (B8,   B12  )          ;
LINK    B16                         = FADD2  (B9,   B13  )          ;
LINK    B19,    B20                 = FNOPXX (SIN        )          ;
LINK    B10,    B11                 = FCNT4  (COS        )          ;
LINK    B21,    B22                 = FNOPXX (B10        )          ;
LINK    X1,  X2,   X3,   X4         = FCPYM4 (B11        )          ;
LINK    B23                         = FMUL1  (B6,   B21  )          ;
LINK    B24                         = FMUL2  (B7,   B19  )          ;
LINK    B25                         = FMUL3  (B14,  B20  )          ;
LINK    B26                         = FADD3  (B23,  B25  )          ;
LINK    B27                         = FMUL4  (B15,  B22  )          ;
LINK    B28                         = FSUB3  (B27,  B24  )          ;
LINK    B29,    B30                 = FCNTFF (B28        )          ;
LINK    B31                         = FWBUF2 (   ,  B30  )          ;
LINK    STTKN,  BRV                 = FPICKA (B31        )          ;
;
LINK    BR1,    BR2                 = FRZERO (BRV        )          ;
LINK    BR3,    BR4,    BR5         = FGEN7  (BR21, BR1  )          ;
```

Figure 7-10 (cont.)
FFT Source Program

```
LINK    BR6,    BR7             = FCPY1  (BR3       )          ;
LINK    RREAL                   = FREDR  (BR6       )          ;
LINK    RIMAG                   = FREDI  (BR7       )          ;
LINK    BR8,    BR9,    BR10    = FGEN8  (BR22, BR2 )          ;
LINK    BR11,   BR12            = FBRV   (BR8       )          ;
LINK    BR13                    = FRADD1 (BR11      )          ;
LINK    BR14                    = FRADD2 (BR12      )          ;
LINK    WDATR,  WADRR           = FWRTR  (BR15, BR13)          ;
LINK    BR17,   BR18            = FCNT8  (BR16      )          ;
LINK    WDATI,  WADRI           = FWRTI  (BR17, BR14)          ;
LINK    BR19,   BR20            = FNOPXX (BR18      )          ;
LINK    BR21                    = FQUE7  (BR4,  BR19)          ;
LINK    BR22                    = FQUE8  (BR9,  BR20)          ;
LINK    PEND                    = FHOST  (BR5,  BR10)          ;
;
;*****************************************************************
;                       FUNCTION
;*****************************************************************
;
FUNCTION        FADD1   = ADD,                  QUEUE  (QUEA1, 16) ;
FUNCTION        FADD2   = ADD,                  QUEUE  (QUEA2, 16) ;
FUNCTION        FADD3   = ADD,                  QUEUE  (QUEA3, 16) ;
FUNCTION        FBRV    = SHLBRV (XX      ), RDCYCS (ZERO,   1) ;
FUNCTION        FCNT4   = COUNT  (4       )                       ;
FUNCTION        FCNT8   = COUNT  (8       )                       ;
FUNCTION        FCNTFF  = COUNT  (256     )                       ;
FUNCTION        FCPY1   = COPYM  (2,  1024)                       ;
FUNCTION        FCPY2   = COPYM  (2,   512)                       ;
FUNCTION        FCPY4   = COPYM  (4,   512)                       ;
FUNCTION        FCPYM4  = COPYM  (4,     0)                       ;
FUNCTION        FDEL    = COUNT  (1       )                       ;
FUNCTION        FDIST2  = DIST   (2       )                       ;
FUNCTION        FGEN1   = COPYBK (4,     1), CNTGE  (128      ) ;
FUNCTION        FGEN2   = COPYBK (4,     1), CNTGE  (128      ) ;
FUNCTION        FGEN3   = COPYBK (8,     1), CNTGE  (128      ) ;
FUNCTION        FGEN4   = COPYBK (8,     1), CNTGE  (128      ) ;
FUNCTION        FGEN7   = COPYBK (8,     1), CNTGE  (128      ) ;
FUNCTION        FGEN8   = COPYBK (8,    64), CNTGE  (128      ) ;
FUNCTION        FHOST   = OUT1   (0,     0), QUEUE  (QUEH,   1) ;
FUNCTION        FNOPXX  = NOP    (XX      )                       ;
FUNCTION        FMUL1   = MUL,                  QUEUE  (QUEM1, 16) ;
FUNCTION        FMUL2   = MUL,                  QUEUE  (QUEM2, 16) ;
FUNCTION        FMUL3   = MUL,                  QUEUE  (QUEM3, 16) ;
FUNCTION        FMUL4   = MUL,                  QUEUE  (QUEM4, 16) ;
FUNCTION        FPICKA  = PICKUP (10      )                       ;
FUNCTION        FQUE1   = QUEUE  (QUE1,  1)                       ;
FUNCTION        FQUE2   = QUEUE  (QUE2,  1)                       ;
FUNCTION        FQUE3   = QUEUE  (QUE3,  1)                       ;
FUNCTION        FQUE4   = QUEUE  (QUE4,  1)                       ;
FUNCTION        FQUE7   = QUEUE  (QUE7,  1)                       ;
FUNCTION        FQUE8   = QUEUE  (QUE8,  1)                       ;
```

Figure 7-10 (cont.)
FFT Source Program

```
FUNCTION        FRADD1  = ADD,                      RDCYCS (N2048,  1) ;
FUNCTION        FRADD2  = ADD,                      RDCYCS (N3072,  1) ;
FUNCTION        FRAND   = AND,                      RDCYCS (MASK,   1) ;
FUNCTION        FREDC   = OUT1   (READ,  6)                            ;
FUNCTION        FREDI   = OUT1   (READ,  7)                            ;
FUNCTION        FREDI1  = OUT1   (READ,  3)                            ;
FUNCTION        FREDI2  = OUT1   (READ,  4)                            ;
FUNCTION        FREDR   = OUT1   (READ,  8)                            ;
FUNCTION        FREDR1  = OUT1   (READ,  1)                            ;
FUNCTION        FREDR2  = OUT1   (READ,  2)                            ;
FUNCTION        FREDS   = OUT1   (READ,  5)                            ;
FUNCTION        FRMASK  = RDCYCS (MSKTBL, 10)                          ;
FUNCTION        FRREDA  = RDCYCS (REDADR,2)                            ;
FUNCTION        FRTBLA  = RDCYCS (TBLADR,1)                            ;
FUNCTION        FRWRIA  = RDCYCS (WRTIAD,2)                            ;
FUNCTION        FRWRRA  = RDCYCS (WRTRAD,2)                            ;
FUNCTION        FRZERO  = NOP    (XX      ), RDCYCS (ZERO,   1)        ;
FUNCTION        FSUB1   = SUB    (XX      ), QUEUE  (QUES1, 16)        ;
FUNCTION        FSUB2   = SUB    (XX      ), QUEUE  (QUES2, 16)        ;
FUNCTION        FSUB3   = SUB,               QUEUE  (QUES3, 16)        ;
FUNCTION        FWBUF2  = WRCYCS (BUF2,  2)                            ;
FUNCTION        FWMASK  = WRCYCS (MASK,  1)                            ;
FUNCTION        FWRTI   = OUT2 (WRITE, 20H, 0),QUEUE(QUEW6, 16)        ;
FUNCTION        FWRTI1  = OUT2 (WRITE, 20H, 0),QUEUE(QUEW3, 16)        ;
FUNCTION        FWRTI2  = OUT2 (WRITE, 20H, 0),QUEUE(QUEW4, 16)        ;
FUNCTION        FWRTR   = OUT2 (WRITE, 20H, 0),QUEUE(QUEW5, 16)        ;
FUNCTION        FWRTR1  = OUT2 (WRITE, 20H, 0),QUEUE(QUEW1, 16)        ;
FUNCTION        FWRTR2  = OUT2 (WRITE, 20H, 0),QUEUE(QUEW2, 16)        ;
;
;*****************************************************************
;                       DATA MEMORY
;*****************************************************************
;
MEMORY  BUF2    =       AREA ( 2)                                      ;
MEMORY  MASK    =       AREA ( 1)                                      ;
MEMORY  MSKTBL  =       0FFFFH, 0FFFEH, 0FFFCH, 0FFF8H, 0FFF0H,
                        0FFE0H, 0FFC0H, 0FF80H, 0FF00H, 00000H         ;
MEMORY  N2048   =       2048                                           ;
MEMORY  N3072   =       3072                                           ;
MEMORY  QUE1    =       AREA ( 1)                                      ;
MEMORY  QUE2    =       AREA ( 1)                                      ;
MEMORY  QUE3    =       AREA ( 1)                                      ;
MEMORY  QUE4    =       AREA ( 1)                                      ;
MEMORY  QUE7    =       AREA ( 1)                                      ;
MEMORY  QUE8    =       AREA ( 1)                                      ;
MEMORY  QUEA1   =       AREA (16)                                      ;
MEMORY  QUEA2   =       AREA (16)                                      ;
MEMORY  QUEA3   =       AREA (16)                                      ;
MEMORY  QUEH    =       AREA ( 1)                                      ;
MEMORY  QUEM1   =       AREA (16)                                      ;
MEMORY  QUEM2   =       AREA (16)                                      ;
```

Figure 7-10 (cont.)
FFT Source Program

```
MEMORY   QUEM3    =        AREA (16)                                    ;
MEMORY   QUEM4    =        AREA (16)                                    ;
MEMORY   QUES1    =        AREA (16)                                    ;
MEMORY   QUES2    =        AREA (16)                                    ;
MEMORY   QUES3    =        AREA (16)                                    ;
MEMORY   QUEW1    =        AREA (16)                                    ;
MEMORY   QUEW2    =        AREA (16)                                    ;
MEMORY   QUEW3    =        AREA (16)                                    ;
MEMORY   QUEW4    =        AREA (16)                                    ;
MEMORY   QUEW5    =        AREA (16)                                    ;
MEMORY   QUEW6    =        AREA (16)                                    ;
MEMORY   REDADR   =        0,      2048                                 ;
MEMORY   TBLADR   =        4096                                         ;
MEMORY   WRTIAD   =        3072,   1024                                 ;
MEMORY   WRTRAD   =        2048,   0                                    ;
MEMORY   ZERO     =        0                                            ;
;
;*********************************************************************
;                          START
;*********************************************************************
;
START                                                                   ;
DATA     EXEC     (FFT1U, STTKN,  0)                                   ;
;
END                                                                     ;
```

Figure 7-11
IM Dump Listing of SIN Data Table

```
1000:  0000 -0192 -0324 -04B6 -0648 -07DA -096C -0AFE
1008: -0C90 -0E21 -0FB3 -1144 -12D5 -1466 -15F7 -1787
1010: -1918 -1AA8 -1C38 -1DC7 -1F56 -20E5 -2274 -2402
1018: -2590 -271E -28AB -2A38 -2BC4 -2D50 -2EDC -3067
1020: -31F1 -337C -3505 -368E -3817 -399F -3B27 -3CAE
1028: -3E34 -3FBA -413F -42C3 -4447 -45CB -474D -48CF
1030: -4A50 -4BD1 -4D50 -4ECF -504D -51CB -5348 -54C3
1038: -563E -57B9 -5932 -5AAA -5C22 -5D99 -5F0F -6084
1040: -61F8 -636B -64DD -664E -67BE -692D -6A9B -6C08
1048: -6D74 -6EDF -7049 -71B2 -731A -7480 -75E6 -774A
1050: -78AD -7A10 -7B70 -7CD0 -7E2F -7F8C -80E8 -8243
1058: -839C -84F5 -864C -87A1 -88F6 -8A49 -8B9A -8CEB
1060: -8E3A -8F88 -90D4 -921F -9368 -94B0 -95F7 -973C
1068: -9880 -99C2 -9B03 -9C42 -9D80 -9EBC -9FF7 -A130
1070: -A268 -A39E -A4D2 -A605 -A736 -A866 -A994 -AAC1
1078: -ABEB -AD14 -AE3C -AF62 -B086 -B1A8 -B2C9 -B3E8
1080: -B505 -B620 -B73A -B852 -B968 -BA7D -BB8F -BCA0
1088: -BDAF -BEBC -BFC7 -C0D1 -C1D8 -C2DE -C3E2 -C4E4
1090: -C5E4 -C6E2 -C7DE -C8D9 -C9D1 -CAC7 -CBBC -CCAE
1098: -CD9F -CE8E -CF7A -D065 -D14D -D234 -D318 -D3FB
10A0: -D4DB -D5BA -D696 -D770 -D848 -D91E -D9F2 -DAC4
10A8: -DB94 -DC62 -DD2D -DDF7 -DEBE -DF83 -E046 -E107
10B0: -E1C6 -E282 -E33C -E3F4 -E4AA -E55E -E610 -E6BF
10B8: -E76C -E817 -E8BF -E966 -EA0A -EAAB -EB4B -EBE8
10C0: -EC83 -ED1C -EDB3 -EE47 -EED9 -EF68 -EFF5 -F080
10C8: -F109 -F18F -F213 -F295 -F314 -F391 -F40C -F484
10D0: -F4FA -F56E -F5DF -F64E -F6BA -F724 -F78C -F7F1
10D8: -F854 -F8B4 -F913 -F96E -F9C8 -FA1F -FA73 -FAC5
10E0: -FB15 -FB62 -FBAD -FBF5 -FC3B -FC7F -FCC0 -FCFE
10E8: -FD3B -FD74 -FDAC -FDE1 -FE13 -FE43 -FE71 -FE9C
10F0: -FEC4 -FEEB -FF0E -FF30 -FF4E -FF6B -FF85 -FF9C
10F8: -FFB1 -FFC4 -FFD4 -FFE1 -FFEC -FFF5 -FFFB -FFFF
1100: -FFFF -FFFE -FFFB -FFF5 -FFEC -FFE1 -FFD4 -FFC4
1108: -FFB1 -FF9C -FF85 -FF6B -FF4E -FF30 -FF0E -FEEB
1110: -FEC4 -FE9C -FE71 -FE43 -FE13 -FDE1 -FDAC -FD74
1118: -FD3B -FCFE -FCC0 -FC7F -FC3B -FBF5 -FBAD -FB62
1120: -FB15 -FAC5 -FA73 -FA1F -F9C8 -F96E -F913 -F8B4
1128: -F854 -F7F1 -F78C -F724 -F6BA -F64E -F5DF -F56E
1130: -F4FA -F484 -F40C -F391 -F314 -F295 -F213 -F18F
1138: -F109 -F080 -EFF5 -EF68 -EED9 -EE47 -EDB3 -ED1C
1140: -EC83 -EBE8 -EB4B -EAAB -EA0A -E966 -E8BF -E817
1148: -E76C -E6BF -E610 -E55E -E4AA -E3F4 -E33C -E282
1150: -E1C6 -E107 -E046 -DF83 -DEBE -DDF7 -DD2D -DC62
1158: -DB94 -DAC4 -D9F2 -D91E -D848 -D770 -D696 -D5BA
1160: -D4DB -D3FB -D318 -D234 -D14D -D065 -CF7A -CE8E
1168: -CD9F -CCAE -CBBC -CAC7 -C9D1 -C8D9 -C7DE -C6E2
1170: -C5E4 -C4E4 -C3E2 -C2DE -C1D8 -C0D1 -BFC7 -BEBC
1178: -BDAF -BCA0 -BB8F -BA7D -B968 -B852 -B73A -B620
1180: -B505 -B3E8 -B2C9 -B1A8 -B086 -AF62 -AE3C -AD14
1188: -ABEB -AAC1 -A994 -A866 -A736 -A605 -A4D2 -A39E
1190: -A268 -A130 -9FF7 -9EBC -9D80 -9C42 -9B03 -99C2
1198: -9880 -973C -95F7 -94B0 -9368 -921F -90D4 -8F88
11A0: -8E3A -8CEB -8B9A -8A49 -88F6 -87A1 -864B -84F5
11A8: -839C -8243 -80E8 -7F8C -7E2F -7CD0 -7B70 -7A10
11B0: -78AD -774A -75E6 -7480 -731A -71B2 -7049 -6EDF
11B8: -6D74 -6C08 -6A9B -692D -67BE -664E -64DD -636B
11C0: -61F8 -6084 -5F0F -5D99 -5C22 -5AAA -5932 -57B9
11C8: -563E -54C3 -5348 -51CB -504D -4ECF -4D50 -4BD1
11D0: -4A50 -48CF -474D -45CB -4447 -42C3 -413F -3FBA
11D8: -3E34 -3CAE -3B27 -399F -3817 -368E -3505 -337C
11E0: -31F1 -3067 -2EDC -2D50 -2BC4 -2A38 -28AB -271E
11E8: -2590 -2402 -2274 -20E5 -1F56 -1DC7 -1C38 -1AA8
11F0: -1918 -1787 -15F7 -1466 -12D5 -1144 -0FB3 -0E21
11F8: -0C90 -0AFE -096C -07DA -0648 -04B6 -0324 -0192
```

Figure 7-12
IM Dump Listing of COS Data Table

```
1200:  FFFF  FFFF  FFFB  FFF5  FFEC  FFE1  FFD4  FFC4
1208:  FFB1  FF9C  FF85  FF6B  FF4E  FF30  FF0E  FEEB
1210:  FEC4  FE9C  FE71  FE43  FE13  FDE1  FDAC  FD74
1218:  FD3B  FCFE  FCC0  FC7F  FC3B  FBF5  FBAD  FB62
1220:  FB15  FAC5  FA73  FA1F  F9C8  F96E  F913  F8B4
1228:  F854  F7F1  F78C  F724  F6BA  F64E  F5DF  F56E
1230:  F4FA  F484  F40C  F391  F314  F295  F213  F18F
1238:  F109  F080  EFF5  EF68  EED9  EE47  EDB3  ED1C
1240:  EC83  EBE8  EB4B  EAAB  EA0A  E966  E8BF  E817
1248:  E76C  E6BF  E610  E55E  E4AA  E3F4  E33C  E282
1250:  E1C6  E107  E046  DF83  DEBE  DDF7  DD2D  DC62
1258:  DB94  DAC4  D9F2  D91E  D848  D770  D696  D5BA
1260:  D4DB  D3FB  D318  D234  D14D  D065  CF7A  CE8E
1268:  CD9F  CCAE  CBBC  CAC7  C9D1  C8D9  C7DE  C6E2
1270:  C5E4  C4E4  C3E2  C2DE  C1D8  C0D1  BFC7  BEBC
1278:  BDAF  BCA0  BB8F  BA7D  B968  B852  B73A  B620
1280:  B505  B3E8  B2C9  B1A8  B086  AF62  AE3C  AD14
1288:  ABEB  AAC1  A994  A866  A736  A605  A4D2  A39E
1290:  A268  A130  9FF7  9EBC  9D80  9C42  9B03  99C2
1298:  9880  973C  95F7  94B0  9368  921F  90D4  8F88
12A0:  8E3A  8CEB  8B9A  8A49  88F6  87A1  864C  84F5
12A8:  839C  8243  80E8  7F8C  7E2F  7CD0  7B70  7A10
12B0:  78AD  774A  75E6  7480  731A  71B2  7049  6EDF
12B8:  6D74  6C08  6A9B  692D  67BE  664E  64DD  636B
12C0:  61F8  6084  5F0F  5D99  5C22  5AAA  5932  57B9
12C8:  563E  54C3  5348  51CB  504D  4ECF  4D50  4BD1
12D0:  4A50  48CF  474D  45CB  4447  42C3  413F  3FBA
12D8:  3E34  3CAE  3B27  399F  3817  368E  3505  337C
12E0:  31F1  3067  2EDC  2D50  2BC4  2A38  28AB  271E
12E8:  2590  2402  2274  20E5  1F56  1DC7  1C38  1AA8
12F0:  1918  1787  15F7  1466  12D5  1144  0FB3  0E21
12F8:  0C90  0AFE  096C  07DA  0648  04B6  0324  0192
1300:  0000 -0192 -0324 -04B6 -0648 -07DA -096C -0AFE
1308: -0C90 -0E21 -0FB3 -1144 -12D5 -1466 -15F7 -1787
1310: -1918 -1AA8 -1C38 -1DC7 -1F56 -20E5 -2274 -2402
1318: -2590 -271E -28AB -2A38 -2BC4 -2D50 -2EDC -3067
1320: -31F1 -337C -3505 -368E -3817 -399F -3B27 -3CAE
1328: -3E34 -3FBA -413F -42C3 -4447 -45CB -474D -48CF
1330: -4A50 -4BD1 -4D50 -4ECF -504D -51CB -5348 -54C3
1338: -563E -57B9 -5932 -5AAA -5C22 -5D99 -5F0F -6084
1340: -61F8 -636B -64DD -664E -67BE -692D -6A9B -6C08
1348: -6D74 -6EDF -7049 -71B2 -731A -7480 -75E6 -774A
1350: -78AD -7A10 -7B70 -7CD0 -7E2F -7F8C -80E8 -8243
1358: -839C -84F5 -864C -87A1 -88F6 -8A49 -8B9A -8CEB
1360: -8E3A -8F88 -90D4 -921F -9368 -94B0 -95F7 -973C
1368: -9880 -99C2 -9B03 -9C42 -9D80 -9EBC -9FF7 -A130
1370: -A268 -A39E -A4D2 -A605 -A736 -A866 -A994 -AAC1
1378: -ABEB -AD14 -AE3C -AF62 -B086 -B1A8 -B2C9 -B3E8
1380: -B505 -B620 -B73A -B852 -B968 -BA7D -BB8F -BCA0
1388: -BDAF -BEBC -BFC7 -C0D1 -C1D8 -C2DE -C3E2 -C4E4
1390: -C5E4 -C6E2 -C7DE -C8D9 -C9D1 -CAC7 -CBBC -CCAE
1398: -CD9F -CE8E -CF7A -D065 -D14D -D234 -D318 -D3FB
13A0: -D4DB -D5BA -D696 -D770 -D848 -D91E -D9F2 -DAC4
13A8: -DB94 -DC62 -DD2D -DDF7 -DEBE -DF83 -E046 -E107
13B0: -E1C6 -E282 -E33C -E3F4 -E4AA -E55E -E610 -E6BF
13B8: -E76C -E817 -E8BF -E966 -EA0A -EAAB -EB4B -EBE8
13C0: -EC83 -ED1C -EDB3 -EE47 -EED9 -EF68 -EFF5 -F080
13C8: -F109 -F18F -F213 -F295 -F314 -F391 -F40C -F484
13D0: -F4FA -F56E -F5DF -F64E -F6BA -F724 -F78C -F7F1
13D8: -F854 -F8B4 -F913 -F96E -F9C8 -FA1F -FA73 -FAC5
13E0: -FB15 -FB62 -FBAD -FBF5 -FC3B -FC7F -FCC0 -FCFE
13E8: -FD3B -FD74 -FDAC -FDE1 -FE13 -FE43 -FE71 -FE9C
13F0: -FEC4 -FEEB -FF0E -FF30 -FF4E -FF6B -FF85 -FF9C
13F8: -FFB1 -FFC4 -FFD4 -FFE1 -FFEC -FFF5 -FFFB -FFFF
```

D

EBIBM-7281GS

EVALUATION BOARD

USER'S MANUAL

CHAPTER 1 PRODUCT OVERVIEW

1.1 Introduction

A new dataflow processor uPD7281 is the base of an image processing board for standard PCs. All parts needed are implemented. An input multiplexer selects one of 16 CCIR sources (camera or VTR) with either internal or external synchronization. This signal is converted to digital with an 8 bit flash-converter and can be manipulated by an input look-up-table. Four image memory planes are available with each 512*512*8bit. Two of them can be connected to a 512 x 512 x 16 bit plane advantageous for special applications. Three analog composite video outputs with output look-up-tables allow pseudo color representation of the selected memory plane.

Four data-flow processors (uPD7281) and a chip for memory and host interface (uPD9305) provide up to 20 MIPS. An internal bus transfers the data between memory and dataflow processors to avoid any performance degradation.

Pipelining and parallel processing together with the efficient instructions of the dataflow processors result in high speed image processing. An on-board 16 bit x 64 k RAM or ROM contains programs for the dataflow processors. By simple commands part of the information can be loaded into the four processors. Initialisation and downloading is done by the PC or other logic in stand-alone applications typically for industrial use.

A software package supports the board. Via menus the system parameters (input channel select, memory plane configuration etc.) and program modules for image processing (different types of filter, edge detection ...) can be selected from the host PC. Another software package contains an assembler and simulator to generate user software for the dataflow processors. So the board is also a good tool for familiarisation and design of dataflow based hardware and software development.

The board was developed in close co-operation with Electronique Lyonnaise who proposed the nick name 'PC Oeil' what means 'PC Eye'.

1.2 GENERAL PRESENTATION

PC Oeil Software is destined to run the image library developed for the PC Oeil board in an IBM-PC environment.

This library proposes around 60 functions that enable one to deal with image acquisition, processing, analysis or manipulation.

However this software remains open and the number of the functions is not limited.

The Software conversional part (processing choice, parameter capture, etc...) is developed in Turbo-Pascal. In the other hand image processing primitives are developed in 7281 assembler, since they use the "PC Oeil" board processor : µPD7281 data-stream processors.

According to the operation to be executed, they use 1, 2, 3 or 4 of them.

CHAPTER 2. THE SOFTWARE ORGANIZATION

2.1 THE SOFTWARE ORGANIZATION

To suppress for the operator the difficulties due to datamation, the image library primitives are open in evoluting menus. This software is conversional. For advancing in the menus, the operator can use either a mouse or certain keys on the keyboard. The primitives are organized around three important classes which appear in the main menu like this :

- Image Management Utilities
- Operations on grey level images
- Operations on binary images.

The figure 1 describes the general organization and the menus chaining.

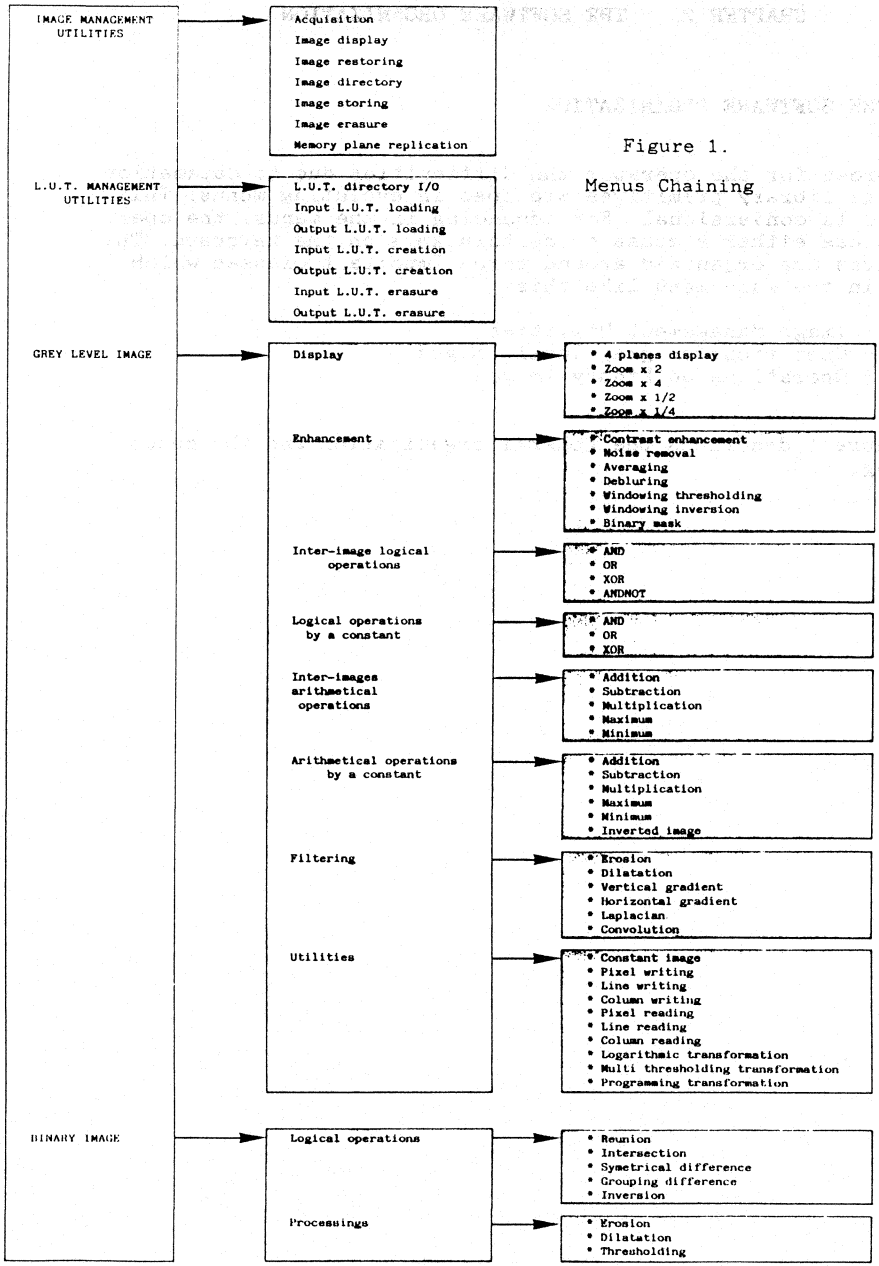


Figure 1.

Menus Chaining

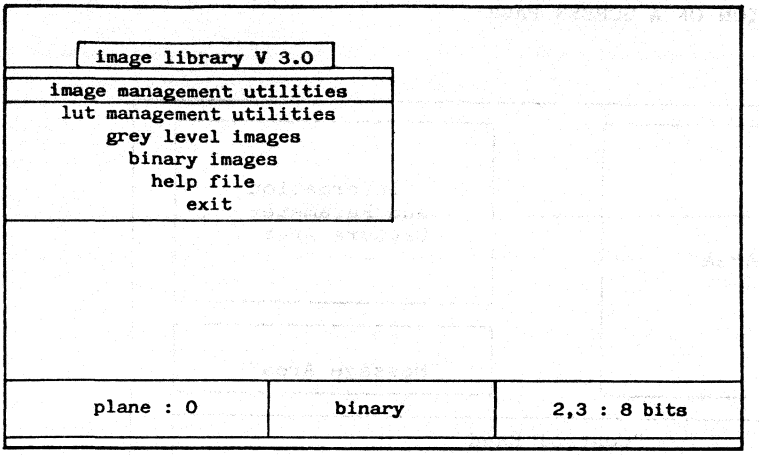


Figure 2. Main Menu

2.2 ORGANIZATION OF A SCREEN PAGE

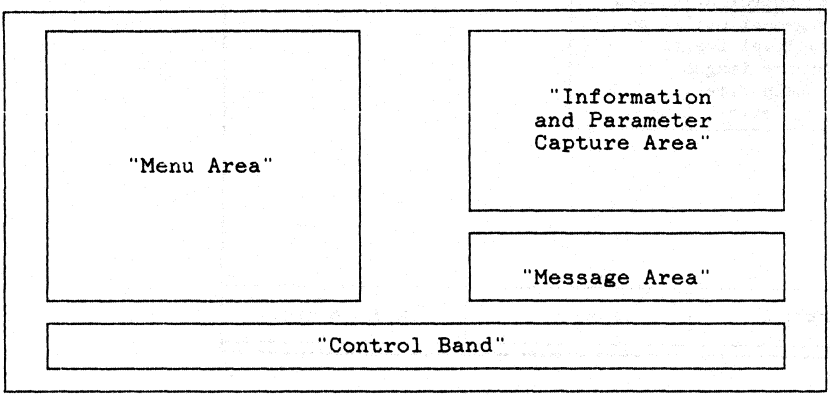


Figure 3. Screen page

Menu area

This area is kept for menus display. The operator successive choices remain on the screen in proportion as the menus are connected. Thus in our example, the operator selected binary images at first and then the logic operations.

In this area, the operator finds four kinds of data that are distinguished between one another by their colors :

- the menu titles : binary images and logic operations
- the submenu titles
- menus management utilities.

These Utilities enable the operator at any level of the library to edit the Help File, to turn back to the main menu or to directly go out of the image library.

Control Band

This area reflects a part of the board configuration state.

- The displayed plane number :
  This number is linked to the type of the processed image.
  Grey level images : 4 planes
  Binary images : 16 planes

- The plane type : a binary or a grey level one.

- Planes 2 and 3 are associated in order to work in 16 bits, or they are separated.

Note : In the case of Binary Image, Planes 2 and 3 are automatically associated.

At any level of the menus, these three control band elements can be modified without using primitives in order to simplify the manipulations.

Then, the operator selects the element he wants to modify and works on it by using the vetting keys of the mouse or the keyboard. (See the example). Successive vettings lead to a procession of facilities.

Example : - procession of possible plane numbers
- alternation of grey level image with Binary images.

In the special case of the "Image Acquisition" primitive, this control band is completed by the following data :

- the number of the video channel used : 0 to 15.
- the input LUT number : 0 to 7.
- the Synchronization type : internal or external.

Information and parameter capture area

In this area appears :

- either a primitive description if the operator used its documentation.
- or the Help File content if the operator selected it in the menu area.
- or the set of parameters essential for a primitive execution. They must be of a conversional type. Values by flow are proposed to the operator. They correspond to the board configuration. The operator can keep (validation key) or modify (a new value + validation key) them.

Message area

Two types of message appear in this area :

- either error messages the list of which is given in Appendix A.
- or the execution time (expressed in seconds) of the last primitive requested by the operator.

2.3 HUMAN INTERFACE

The operator may use either a double touch mouse or certain keys of the keyboard.

2.3.1 Function keys

- < ↑ > and < ↓ >
  These keys are used for vertical movement in the menus. A roller mode is set on the elements of the menu selected by the operator and on the control band.

- < → > and < ← >
  These cursor keys are used for movements in the control band. The elements are covered in roller mode.

- <RET> <SPACE>
  These two keys play the same role. They enable to validate either an element of the menus, or an element of the control band. (See paragraph 2.2), or a parameter of the information area.

- <D>
  After a primitive selection, the operator can ask its description by using this key. Then, the documentation appears in the information area.

- <BS>
  Key for character erasing. It is used during parameter seizings.

- <ESC>
  This key enables at any level of the software to return to the previous level. There are two possibilities :

  - its most common use is made in the menu area. This key result is then a return to the previous level.

  - another less common but at least necessary use is met during the execution of a image processing primitive.

  After seizing a primitive, the operator must be able to escape from it without making a processing. That result is produced by the <ESC> key that then leads the operator to the menu area back.

2.3.2 The mouse

This optional element of the material configuration brings at the same time comfort and a real communication tool between the computer and the operator. The mouse is easy to use and enables to select a function in half a thick without using the cursor keys.

An only two key mouse is sufficient.

The key on the left is equivalent to the keyboard validation keys <RET> and <SPACE>.

The key on the right is equivalent to the return key to the previous menu or to the output of an image processing primitive <ESC>.

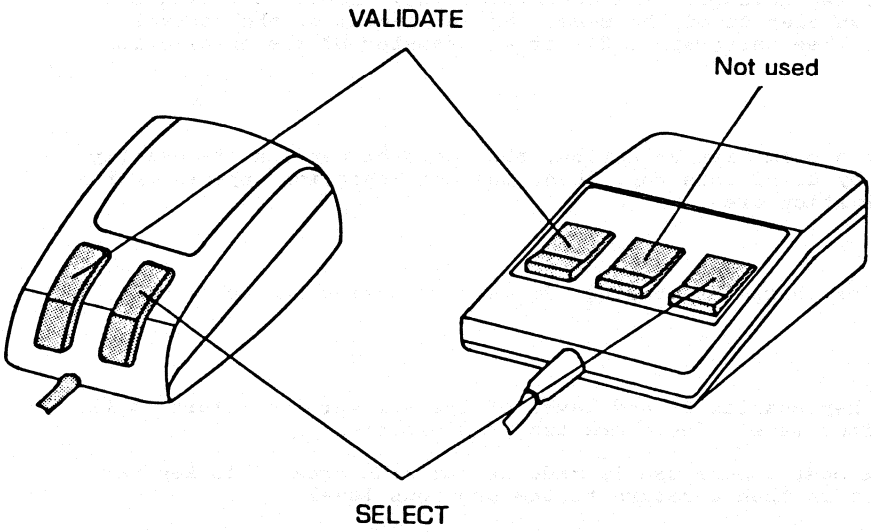


Figure 4. Use of the mouse

2.4 HELP PROCEDURE

At any level of the software the operator can accede to an help procedure that reminds him of the meaning and the use of the function keys specific to the software (see paragraph 2.3.1) and of the use of the mouse (see paragraph 2.3.2).

The access to this procedure is made from any menu by selecting HELP in the menu area.

CHAPTER 3. IMAGE LIBRARY

3.1 THE LIBRARY OBJECTS

When using the library PC Oeil Software, the operator must manipulate the following objets :

. Images :

There are two kinds :
- grey level images : 1 pixel = 1 octet
- binary images : 1 pixel = 1 bit

They are stored according to two directories :
- GREY for the Grey level images, and
- BINARY for the Binary images.

When manipulating the images, the operator must not precise his extension. So, he can only use eight characters for naming the image.

. Memory planes :

The PC Oeil board has four memory planes composed by 256 K (512 x 512 x 8 bits). They have two modes of utilization according to the kinds of images manipulated by the operator : the grey level images or the binary ones.

- Grey level images :

  In that case, one image fills a memory plane of the board. So, the operator can use four memory planes on images planes and their logic numbers vary from zero to three.

- Binary images :

  Here, the images are handled according to the 16 bits mode. Planes 2 and 3 are imploded.

1 Binary image	=	512 x 512 x 1 bit	=	32 K
Plane 2 + Plane 3	=	256 K + 256 K	=	512 K

  So that, the operator can use 16 Binary Planes. Their logic numbers vary from zero to 15.

Note In the whole Software, the words Displayed Plane or Viewing Plane are used for a grey level image plane.

. Camera Inputs :

PC Oeil board can manage up to 16 Camera Inputs (4 in its basic configuration, and 16 with the camera extended module). For every camera input, a channel logic number is associated. So that, according to the configuration, the operator can use the numbers of the channel varying from 0 to 3 or 0 to 15.

. LUT files

There are two kinds :
- input LUT files. These are 256 byte wide files.
- output LUT files. These are 768 byte wide files (256 byte for one colour beam).

They are stored with two possible extensions :
- SOR for output LUT file
- ENT for input LUT file.

. Input LUTs

The PC Oeil has 8 Input LUTs. They are refered to by their logic numbers (from 0 to 7). The manipulator can associate any camera input with any LUT as he likes.

. Output LUT

The PC Oeil board has only one output RGB LUT.

3.2 IMAGE MANAGEMENT UTILITIES

. Image Acquisition :

This function enables the operator to set in a memory plane an image issuing from a camera. He must then select :

- the input channel number
- the acquisition mode (binary or grey level)
- the memory plane number. This number varies between 0 and 3 for the grey level images and between 0 and 15 for the binary ones.
- the synchronization type of the camera (internal or external)
- the input LUT number.

. Image display :

The operator can choose the type (binary or grey level) and the plane he wants to visualize on the monitor.

. Image restoring :

This function allows to load a grey level or a binary image which has been stored on a fixed disk in a memory plane selected by the operator. This memory plane is visualized by default.

. Image directory :

This function enables to get the name list of the grey level or the binary images saved on the fixed disk.

Extension : .IMA : grey level images
.BIN : binary images

. Image storing :

This function enables to store an image according to a name, especially chosen by the operator on a fixed disk. By default this image squares with the displayed plane. However, the operator may choose another plane. The saved image can belong to the binary or the grey level type.

. Image Erasure :

This function enables to erase a grey level or a binary image stored on a fixed disk. The operator must give the image name. Before erasing, the system requires confirmation.

. Memory Plane Replication :

This function enables to recopy a memory plane on another. The displayed plane squares with the origin plane.

3.3 LUT MANAGEMENT UTILITIES

. Input LUT loading :

This function enables thes operator to load a 256 byte wide file in one of the 8 input LUT.

. Output LUT loading :

As like the preceding function, the operator can load a 768 byte wide file in the output LUT. This LUT remains active until a new file is loaded. By default a monochrome LUT is defined during software initialization.

. LUT directory :

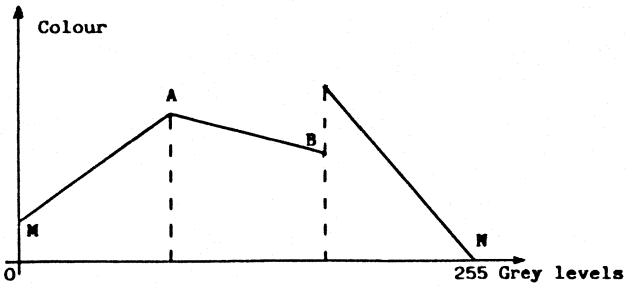
By selecting this function, the operator can access the name of all the input and output LUT files.

Extension : .ENT : input LUT file
.SOR : output LUT file

. Output LUT creation :

This function enables the operator to create a colour LUT file. For each channel (Red, Green and Blue), he is asked for defining the "segments" of the colour function.

Example :



MA, AB and BN are segments.

The operator must enter the number of segments and their upper limits.

Then, for each so defined segment, he has to determine the colours of the lower abscisse point and the higher one. The colour function is then computed for each point by a linear approximation algorithm.

. Input LUT creation :

This function is exactly the same as the preceding function, excepting that only one channel must be defined (the input LUTs are monochrome ones).

. Output LUT erasure :

By selecting this function, the operator may erase any output LUT file stored on the fixed disk.

. Input LUT erasure :

As the preceding function, the operator may erase any input LUT file.

3.4 GREY LEVEL IMAGES

3.4.1 Image Display

. Four plane Display :

The user can view all the memory planes on only one. Then the selected plane for the viewing is modified.

Displayed Plane

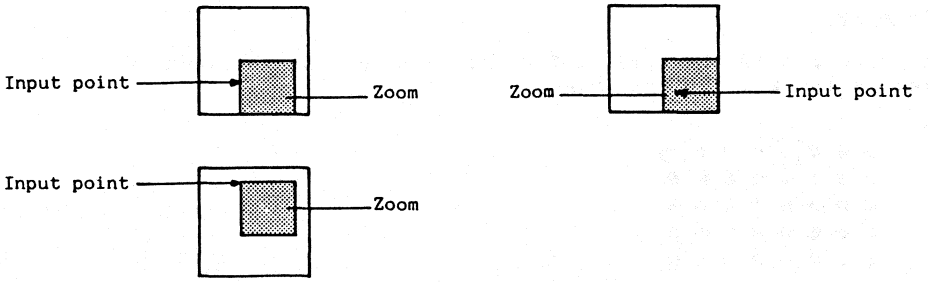
Selected Plane	S P + 1
S P + 2	S P + 3

S P = Selected Plane

. Zoom by 2 :

This function produces a ZOOM by 2 of a part of an image chosen by the operator. This part limited by the first upper left point is 256 x 256 pixels high. Every point is doubled for x and y. The result image is classified in the memory plane selected by the operator. This becomes a viewing plane.

Example :



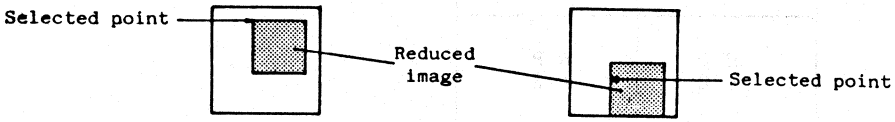
. Zoom by 4 :

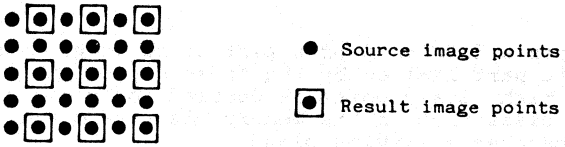
The principle is the same as for the zoom by 2. The size of the part of the image plane on which the zoom is produced is 128 x 128 pixels. Every point is replaced by a 4 x 4 matrix according to which all the points get the same value as the origin point.

. Zoom by one half :

This function enables to reduce by half an image plane selected by the operator in another also selected image plane. This is the following principle. The reduced image is put in the result image plane in a 256 x 256 pixels area determined by its first upper left point.

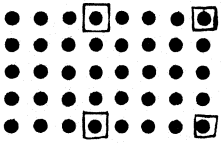
Result Plane Examples :





. Zoom by 1/4 :

The principle is the same as for the zoom by 1/2. The reduced image is 128 x 128 pixels high.



3.4.2 Image Enhancement

. Contrast Accentuation :

The following mask is used :

$$\begin{matrix} 0 & -1 & 0 \\ -1 & 8 & -1 \\ 0 & -1 & 0 \end{matrix}$$

It consists in adding the grey function of the initial image to a Laplace mask.

A special attention is given to horizontal and vertical directions.

The source memory plane must be different from the result memory plane. The result plane becomes the displayed plane.

. Noise cleaning :

The following mask is used :

$$\frac{1}{16} \begin{matrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{matrix}$$

This function enables to suppress high frequencies from the image. The source memory plane must be different from the result memory plane. The latter is viewed on the monitor.

. Averaging :

The following mask is used :

$$\frac{1}{9} \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$$

This transformation enables to produce a noise cleaning and a border smoothing. The source memory plane must be different from the result memory plane. The latter is viewed on the monitor.

. Debluring :

The following mask is used :

$$\begin{matrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{matrix}$$

Such a transformation consists in adding the initial image to its Laplacian mask. The source memory plane must be different from the result memory plane. The latter is viewed on the monitor.

. Window Inversion :

Grey levels of an image may be inverted only in a window. The operator has to determine the upper left corner and lower right corner coordinates.

Source plane must be the number 0 one. But destination may be one of the four image planes.

. Window Thresholding :

Grey levels of an image may be thresholded in a window. The operator has to determine the upper left corner and lower right corner coordinates.

The resulting image is a grey level one, with only two levels (0 and 255).

. Binary mask :

This function realizes the multiplication of a grey level image with a binary image. The result (grey level image) is viewed.

3.4.3 Logical Operations between an Image and a Constant

. And, OX, XOR

From a source image plane and a constant (included between 0 and 255) given by the operator, this function calculates the resultant image in making a logic operation selected bit by bit for each pixel.

3.4.4 Logical Operations between two Images

. And, OR, XOR, ANDNOT

From two source image planes selected by the operator, this function calculates a logic operation chosen pixel by pixel, bit by bit.

3.4.5 Arithmetical Operations between an Image and a Constant

The constant value is an integer value included between 0 and 255.

. Addition

This function adds to each pixel value the constant value. Values superior to 255 are replaced by 255. The operator selects the source image plane and the result image plane. The latter is viewed.

. Subtraction

This function subtracts from each pixel value the constant value. Values inferior to 0 are replaced by 0. The operator selects the source image plane and the result image plane. The latter is viewed.

. Multiplication

This function multiplies each pixel value of the source image plane by the constant value selected by the operator. Values superior to 255 are replaced by 255. The result image plane is viewed.

. Maximum

This function compares each pixel value of the source image plane with the constant value selected by the operator. The higher value is retained in the result image plane. The latter is viewed.

. Minimum

This function compares each pixel value of the source image plane with the constant value selected by the operator. The lower value is retained in the result image plane. The latter is viewed.

. Image Inversion

This function enables to get the inverted image plane (255-pixel value) from the source image plane selected by the operator. The result image plane is viewed.

3.4.6 Arithmetical Operations between two Images

. Addition

The operator selects two source image planes P_1 and P_2. This function adds each P_1 pixel value to its corresponding P_2 pixel value. The result is set in an image plane selected by the operator. Values superior to 255 are replaced by 255. The result plane is viewed.

. Subtraction

The operator selects two source image planes P_1 and P_2. This function subtracts the P_2 pixel value from its corresponding P_1 pixel value. The result is set in an image plane selected by the operator. Values inferior to 0 are replaced by 0. The result plane is viewed.

. Multiplication

The operator selects two source image planes P_1 and P_2. This function multiplies each P_1 pixel value by its corresponding P_2 pixel value. The result is set in an image plane selected by the operator. Values superior to 255 are replaced by 255. The result plane is viewed.

. Maximum

The operator selects two source image planes P_1 and P_2. This function compares each P_1 pixel value with its corresponding P_2 pixel value. The higher value is retained in the result image plane. The latter is viewed.

. Minimum

The operator selects two source image planes P_1 and P_2. This function compares each P_1 pixel value with its corresponding P_2 pixel value. The lower value is retained in the result image plane. The latter is viewed.

3.3.7 Filtering

. Erosion

Every pixel of the source image plane selected by the operator is replaced by the inferior limit of the grey function in the considered pixel environment. The operator may choose to make this transformation several times. That is the reason why he must specify the number of the successive erosions just as the source image plane and a working image plane. The viewed image plane is the result image plane.

. Dilatation

Every pixel of the source image plane selected by the operator is replaced by the superior limit of the grey function in the considered pixel environment. The operator may choose to make this transformation several times. That is the reason why he must specify the number of the successive dilatations just as the source image plane and a working image plane. The viewed image plane is the result image plane.

. Vertical Gradient

The following mask is used :

$$\begin{matrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{matrix}$$

Vertical outlines of the image plane selected by the operator can then be detected. The source image plane must be different from the result image plane. And the latter is viewed on the monitor.

. Horizontal Gradient

The following mask is used :

```
1  0 -1
1  0 -1
1  0 -1
```

The horizontal outlines of the source image plane selected by the operator can then be detected. The source image plane must be different from the result image plane. The latter is viewed on the monitor.

. Laplacian

The following mask is used :

```
-1 -1 -1
-1  8 -1
-1 -1 -1
```

That transformation gives an approximative value for the grey function second derived of the image plane selected by the operator. The source image plane must be different from the result image plane. The latter is viewed on the monitor.

. Convolution

The operator can determine the negative or positive integer coefficients of the 3 x 3 mask which must be realized on the image plane he selected. The source image plane must be different from the result image plane. The latter is viewed on the monitor.

3.4.8 Utilities

. Constant Image Creation

This function enables to fill an image plane with constant pixel values selected by the operator.

. Pixel writing

With this function the operator can change the pixel value of an image plane.

. Line writing

With this function the operator can change the value of a line pixels in an image plane. Then, every pixel of that line has the same value.

. Column writing

With this function, the operator can change the value of a column pixels in an image plane. Then, every pixel of that column has the same value.

. Pixel reading

This function enables to know an image plane pixel value.

. Line reading

This function enables to know a line pixel values in a image plane.

. Column reading

This function enables to know a column pixel values in an image plane.

. Logarithmic transformation

This function transforms a grey level image in another one. At any source image pixel is associated a destination pixel as follows :

$$D_{ij} = 46 \times \ln (1 + S_{ij})$$

S_{ij} and D_{ij} are radiometries of pixel (i, j) from source image and destination image.

. Multithresholding

This function transforms a grey level image in another one which has only a few discrete grey levels.

The operator must enter thresholds (up to 20) if S_i and S_{i+1} are two consecutive thresholds, and if the condition $S_i < S_{ij} < S_{i+1}$ is true, then $D_{ij} = S_i$ (S_{ij} and D_{ij} have the same definition as for the preceding function).

. LUT transformation

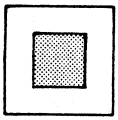
As for LUT file creation (cf paragraph 3.3), the operator may define a LUT. The source image is then computed with this table and stored in destination plane.

3.5 BINARY IMAGES

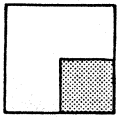
3.5.1 Logical Operations

. Reunion

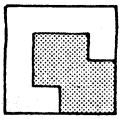
This processing is executed between two binary image planes selected by the operator. The selected result image plane contains the group of points of the two source image planes. The displayed plane is the result image plane.



Plane 1



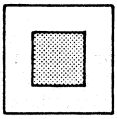
Plane 2



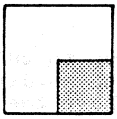
Result plane

. Intersection

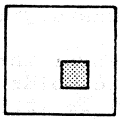
This processing is executed between two binary image planes selected by the operator. The result image plane contains the group of points common to the two source image planes. The displayed plane is the result image plane.



Plane 1



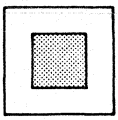
Plane 2

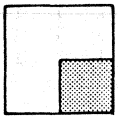


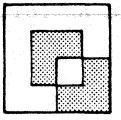
Result plane

. Symmetric difference

This processing is executed between two binary image planes selected by the operator. The result image plane contains the group of points of the two source image planes except for the points common to the two planes. The displayed plane is the result plane.

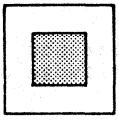

Plane 1

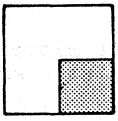

Plane 2

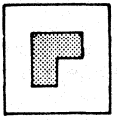

Result plane

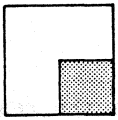
. Set difference

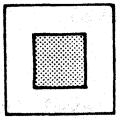
This processing is executed between two binary image planes selected by the operator. The result image plane contains the group of points of the first image plane except for those common to the second image plane. The displayed plane is the result image plane.

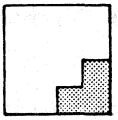

Plane 1


Plane 2


Result plane

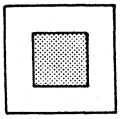

Plane 1

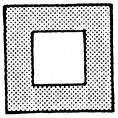

Plane 2


Result plane

. Inversion

This function enables to get on the result image plane the complementary image plane of the source image plane.


Source plane


Result plane

3.5.2 Processings

. Erosion

Given a binary mask, eroding a binary image consists in determining all the intersections of the translated image in the directions defined by the mask.

The mask is defined as follows :

1 1 1
1 1 1
1 1 1

. Dilatation

Given a binary mask, dilating a binary image consists in determining all the unions of the translated image in the directions defined by the mask.

The mask is defined as follows :

1 1 1
1 1 1
1 1 1

. Thresholding

Given any threshold S (fixed by the operator, and contained in a [O....255] scale, this function computes a grey level image as follows :

$f(P_{ij}) = 0$ if $f(P_{ij}) < S$
$f(P_{ij}) = 1$ if $f(P_{ij}) \geq S$

where $f(P_{ij})$ is the P_{ij} pixel grey level.

Note This function computes a grey level image, but the resulting one has a binary type.

APPENDIX A

LIST OF ERROR MESSAGES

0	Graphics initialization error
1	Graphics board driver does not exist
2	Configuration file does not exist
3	PC type not supported
4	'REPERTOI.MEN' file does not exist
5	Incorrect 'REPERTOI.MEN' file
6	Documentation file does not exist
7	Menu file does not exist
20	Out of scale value
22	Same source and destination planes
30	Executive program does not exist
31	Transfer to MAGIC can't be achieved
39	Completed function
40	Given up function
98	File does not exist
99	Incorrect 'CONFIG.EYE' file

E

FAXXX-XXXX-7281
ASSEMBLER AND SIMULATOR SOFTWARE

USER'S MANUAL

3

ASSEMBLER AND SIMULATOR SOFTWARE

IMAGE PIPELINED PROCESSOR

FAM86-08SS-7281

FAC86-15DD-7281

FAMSD-15DD-7281

FAUNX-0T16-7281

FUNCTIONAL ASSEMBLER

SIMULATOR

OBJECT CONVERTER

USER'S MANUAL

PREFACE

Utilizing a novel computer architecture called "DATA FLOW", NEC has developed a high-speed digital signal processor specifically designed for image processing applications. The official NEC part number for the processor is uPD7281, known also as Image Pipelined Processor(ImPP). The employed data flow architecture for the uPD7281 inherently provides a true parallel processing capability, thus meeting the requirements for much needed processing power for any typical digital image processing application.

In order to help develop user application software for the uPD7281, NEC is providing a software package containing three basic software tools: uPD7281 assembler, software simulator and object-code converter.

This manual contains detailed explanations on how to use the three software tools mentioned above. It also contains information on software development procedure for the uPD7281. The manual is divided into four basic parts, and a brief description of each part is as follows:

PART I OUTLINE OF THE SOFTWARE PACKAGE
Contains information about system requirements in order to run this software package. Also contains features and usage of the package.

PART II USAGE OF THE ASSEMBLER
Explains functions, features and usage of the Assembler, structure of source programs, etc.

PART III USAGE OF THE SIMULATOR
Contains an outline of the software simulator and explains its functions and its usage.

PART IV USAGE OF THE OBJECT-CODE CONVERSION PROGRAM
Contains an outline of the Object-code Conversion Program and explains its functions, its usage and the object-code formats.

PART V VAX/UNIX BASED SOFTWARE
Describes the differences between PC based software and that which is available on the VAX (UNIX-BASED).

PART I

OUTLINE OF THE SOFTWARE PACKAGE

INTRODUCTION

This part contains information about the system requirements needed to execute the uPD7281 software. In addition, it also contains information about the notations used through out this manual, how to invoke the assembler, the software simulator and the object-code converter, and program development procedure using the software package.

This section is organized as follows:

USING THE SOFTWARE PACKAGE

Contains information about the system requirements in order to run the uPD7281 software, contents of the software package, and the notations used in the manual.

CHAPTER 1 - OUTLINE

Contains brief description of each different software tool contained in the package.

CHAPTER 2 - PROGRAM DEVELOPMENT PROCEDURES

Explains how to develop user specific application programs for the uPD7281 using the software package.

USING THE SOFTWARE PACKAGE

The uPD7281 Software Package is to be run under CP/M-86 or MP/M-86* Operating Systems using the MD-086 microcomputer development support system, an NEC development system. Use of this particular development system is not critical. However, the system does provide the compatibility with hardware evaulation boards for uPD7281 called "EXECUTION SYSTEM". Both the Execution System and the NEC MD-086 development system have Multibus** interface.

Even though the uPD7281 software package can be used under both CP/M-86 and MP/M-86 operating systems, all of the references made to the operating system and command examples used through out this manual are for CP/M-86 operating system only.

The FAMSD-software works on computers with MS-DOS *** operating system.

The following lists the computer system structure, the contents of software package, and the notations used through out this manual.

1. Computer System Structure

   (1) Host computer
   NEC microcomputer development support system, MD-086
   or PC with 5 1/4 Floppy disk drive

   (2) Operating Systems
   MP/M-86 (Version 2.0 Rev. 3 or higher),
   CP/M-86 (Version 1.1 or higher)
   or MS-DOS (Version 2.0 or higher)
   ****

   (3) Memory Requirements
   At least 128K bytes of user memory.

   (4) Console
   CRT, keyboard.

2. Contents of the Software Package

   The uPD7281 Software Package is composed of the following three software programs:

   1) uPD7281 Assembler
   2) uPD7281 Simulator
   3) uPD7281 Object-code Conversion Program

   (1) File names
   1) uPD7281 Assembler
   AS7281.CMD
   AS7281.OMn ..... Overlay file
   (n is a sequence number.)

---

* MP/M-86 and CP/M-86 are trade marks of Digital Research, Inc.
** Multibus is a trademark of Intel Corporation.
*** MS-DOS is a trademark of Microsoft Inc.
**** for VAX/UNIX based Software see Part V

2) uPD7281 Simulator
   SM7281.CMD
   SM7281.OMn ..... Overlay file
                    (n is a sequence number.)

3) uPD7281 Object-code Conversion Program
   OH7281.CMD

(2) Floppy disk format
The software package is supplied on either single-side/single-density floppy disks or a double-side/double-density floppy disk for the NEC MD-086 development system.

FAMSD software is supplied on a 5 1/4' double sided double density floppy disk.

3. Notations

This user's manual uses a number of symbols for explaining commands and operations of the software package. These notations are explained below.

CR ; stands for either carriage return (0DH) or combination of carriage return and line feed.

LF ; stands for line feed (0AH).

[ ] ; means that the character string inside the brackets may be omitted. When this term is omitted, a pre-defined default value(s) will be automatically substituted. ([] is not to be coded.)

{ } ; means that one of the character strings inside the braces must be selected.

Example:

DEFINE { SYMBOL / SYM } represents

either 'DEFINE SYMBOL' or 'DEFINE SYM'.

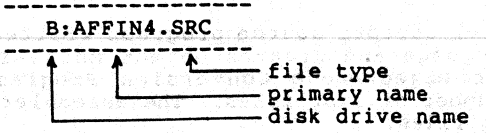
,... ; means that the item preceding comma (,) may be entered repeatedly.

file name ; 'file name', when present in general input formats throughout this manual, may represent either a file on a disk or a file as a peripheral unit. These are shown below.

1) File on disk
The file name has the following format.

```
-----------------------------------------
  [drive name:] primary name [.file type]
-----------------------------------------
```

Example;



The primary name may be up to 8 characters long and file type may be up to 3 characters long. In some cases, the notation d: (d=A - P) is used for clear indication of the disk drive name.

2) File name corresponding to peripheral

LST:	printer
CON:	console
PUN:	paper tape punch

NOTE: Peripheral files such as PUN:, CON: and LST: are supported only by symbol-table module files and object module files (described in Part IV), and they are supported by neither the Assembler (described in Part II) nor the Simulator (described in Part III). For details, refer to proper sections of this manual.

CHAPTER 1 OUTLINE

This chapter provides a brief description of each software tool (Assembler, Simulator and Object-code Conversion Program) in the uPD7281 Software Package. In addition, it also outlines uPD7281 software development using this package.

1.1 uPD7281 Assembler

This assembler accepts source programs written in uPD7281 assembly language and creates object modules for either Simulator or Object-code Conversion Program. It also produces a number of list files. The assembler consists of the following files.

```
--------------
  AS7281.CMD
  AS7281.OM0
  AS7281.OM1
  AS7281.OM2
--------------
```

The above-mentioned AS7281.OM0, AS7281.OM1 and AS7281.OM2 are overlay files which are automatically loaded and executed by the Assembler when necessary.

1.2 uPD7281 Simulator

The Simulator accepts object modules produced by the Assembler and simulates the program under specified system parameters.

The Simulator consists of the following files.

```
-------------
 SM7281.CMD
 SM7281.OM0
 SM7281.OM1
 SM7281.OM2
-------------
```

The above-mentioned SM7281.OM0, SM7281.OM1 and SM7281.OM2 are overlay files which are automatically loaded and executed by the Simulator when necessary.

1.3 uPD7281 Object-code Conversion Program

The Object-code Conversion Program accepts object modules produced by the assembler and converts them to either HEX-format or ASCII-data-format object modules, so that they can be loaded into uPD7281 processor.

The file name of the Object-code Conversion Program is shown below.

```
------------
 OH7281.CMD
------------
```

1.4 Development of programs using the Software Package

The uPD7281 Assembler enables a user to write the specific application programs for the uPD7281 in full symbolic manner, thereby increasing the programming efficiency. The uPD7281 Simulator is capable of simulating the uPD7281s in many different image processing system environments. These different system environments may or may not include the companion chip of uPD7281, called uPD9305 MAGIC (Memory Access and General bus Interface Chip). Therefore, utilizing the Simulator, one may be able to simulate a user-specific image processing system using uPD7281s and a specific application program without actually building the hardware. Moreover, the built-in symbolic debugging capability of the Simulator permits use of the same symbols as in the Assembler.

Fig.1-1 shows the flow of program development procedure using the uPD7281 Software Package.

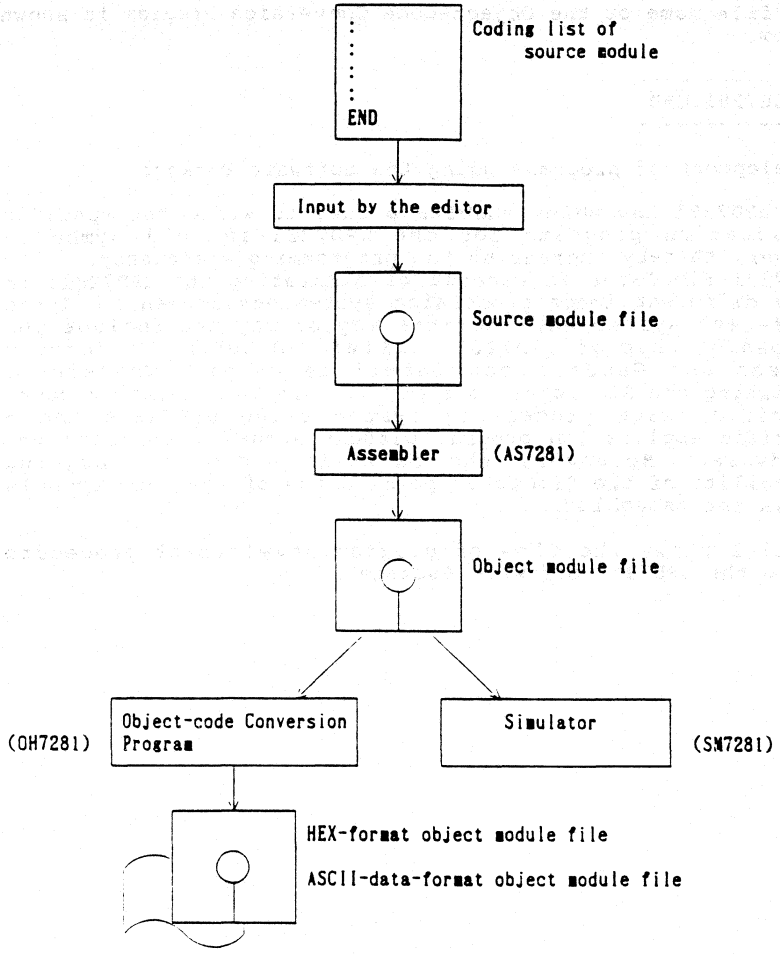


Fig. 1-1 Program development flowchart when uPD7281 Software Package is used

CHAPTER 2 PROGRAM DEVELOPMENT PROCEDURES

This chapter briefly describes each stage of the program development cycle using the uPD7281 Assembler, Simulator and the Object-code Conversion Program.

2.1 Creating source module file

A source module may be created using any text editor. After creating the source file, it is input to the uPD7281 Assembler. Of course, the source file must follow certain rules of syntax for the assembler, and it is explained in Part II of this manual.

2.2 Execution of Assembler (creation of object module file)

After a program is created in the source module file, it is assembled by the assembler. The assembler creates an object module file and a number of list files. The list files include assembly listing file, error listing file, and cross-reference symbol listing file.

The object module file contains the object code produced by the assembler. The generated object code is input to the Simulator and/or the Object-code Conversion Program.

After an assembly, the number of errors and warnings that are related to grammatical mistakes within the source module are displayed on the console. When the assembler detects an error in the source file, it places an error message line in the error listing file immediately after the line containing the error.

The procedures to generate an error free object file are shown in Fig. 2-1.

2.3 Simulation

Once an error free object file is generated by the assembler, it is input to the Simulator for software simulation. The uPD7281 Simulator has a capability to fully simulate the user program under various system configurations. One should note that the system configuration is also software programmable.

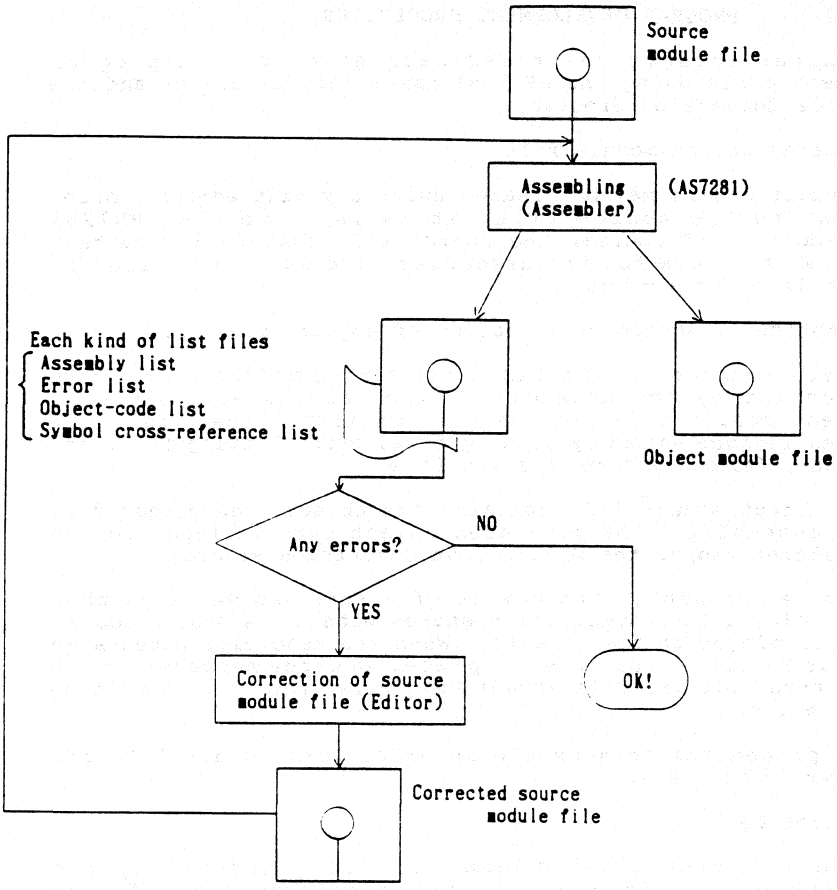


Fig. 2-1 Creation of object module file

In order to understand the uPD7281 Simulator operations, the user must be familiar with the internal architecture of the uPD7281. Otherwise, the user may be unable to fully utilize the debugging capability of the Simulator. During a simulation, the user can either step through or trace through a user program with several break points. Utilizing the STEP, TRACE and many other powerful commands plus macro commands, a user program can be efficiently debugged.

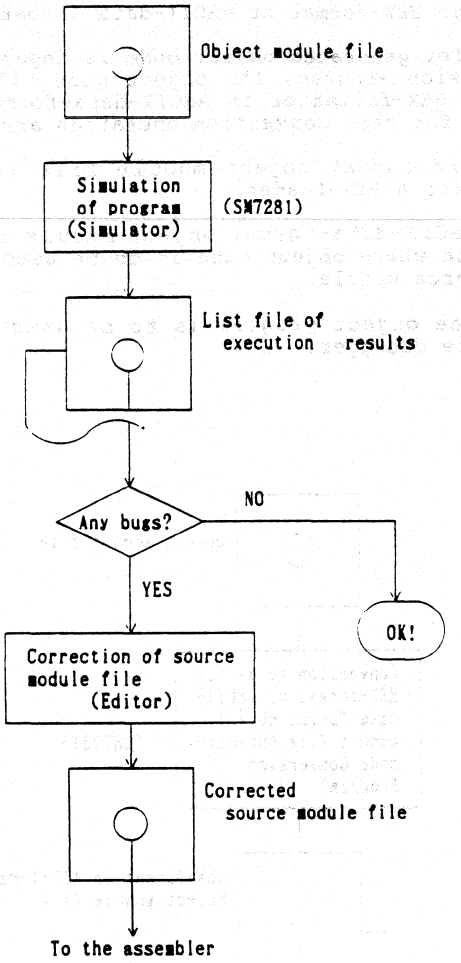


Fig. 2-2 Simulation Procedures

2.4 Conversion to HEX-format or ASCII-data format object module

When assembler generated object code is input to the Object-Code Conversion Program, the object code file is converted to either a HEX-format or an ASCII-data-format object file. The reasons for this conversion operation are as follow:

1) When HEX-format object module file is necessary as input for a HEX-Loader.

2) When ASCII-data-format object module is necessary in the case where object code is to be used as data within the source module.

3) When the object program is to be used as input for a hardware debugger.

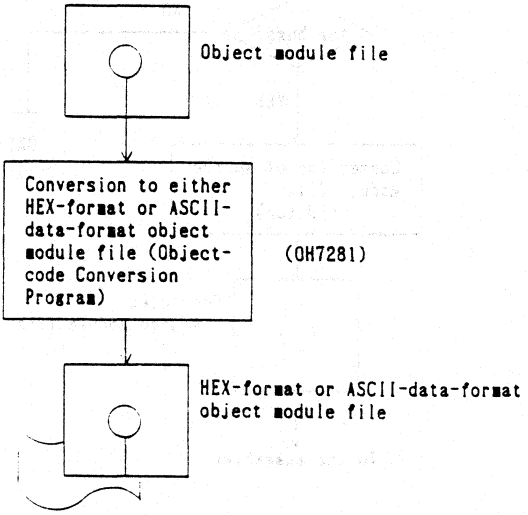


Fig. 2-3 Creation of HEX-format or ASCII-data-format Object module file

PART II

USAGE OF THE ASSEMBLER

CHAPTER 1 OUTLINE OF SOFTWARE

1.1 Outline of the assembler's functions

The assembler accepts source module files coded in uPD7281 assembly language, transforms (assembles) them into uPD7281 machine code, and outputs an object module file as a result. The assembler can also output the following five files as list files.

1) Assembly list
2) Error list
3) Object code list (mnemonic format)
4) Object code list (HEX format)
5) Symbol cross-reference list

In order to output lists 2), 4), and 5), the optional control parameters must be specified when invoking the assembler. For details, see Section 6.2 of this manual.

The object module files which are output by the Assembler are used as input files for the Simulator and the Object-code Conversion Program. For more information on the Simulator and the Object-code Conversion Program, please see Parts III and IV.

The input/output relationships between the Assembler, the Simulator and the Object-code Conversion Program are shown in Fig. 1-1.

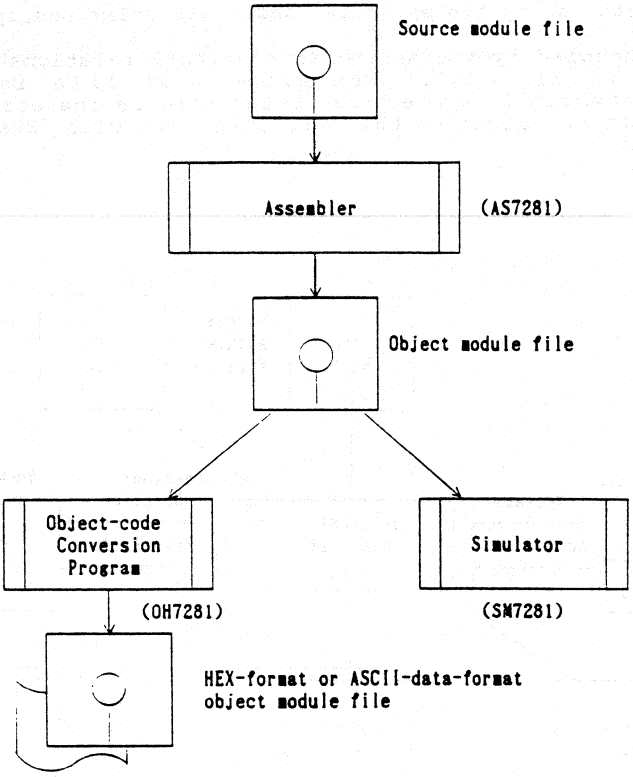


Fig. 1-1 Input/Output Relation Diagram of uPD7281 Software Package

1.2 Files handled by the assembler and their relationships

Files handled by the assembler and their relationships are shown in Fig. 1-2. The error list file is shown independently from the 'list file' because the error list file can be output to the file specified with 'ERRORLIST' control.

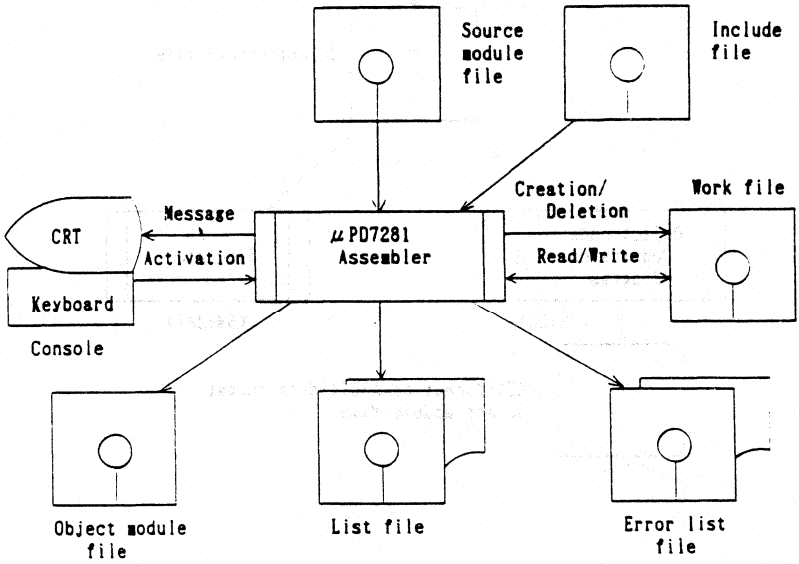


Fig. 1-2 Organization of files

The Include file and the Work file are explained below.

(1) Include file

This is a file other than the current source module file, and it can be included in the source module file by specifying the 'INCLUDE' control. (The organization of this file is the same as a source module file.)

(2) Work file

The assembler creates work files on the disk specified by 'WORKFILES' control during the time of assembly, and deletes them after the assembly is completed. The work files have the following file name format.

AS7281.$$n (n is a sequence number starting from 0.)

Since the assembler deletes any file named AS7281.$$n, a user should not name files with the format of the assembler workfile names.

NOTE: For more information about the optional control parameters, see section 6.2.

1.3 Assembler's features

Since the uPD7281 employs a token-based data flow architecture, the execution of a program within a uPD7281 is significantly different from an ordinary von Neumann type processor. In order for a uPD7281 to be able to execute a program, first object code needs to be downloaded into the uPD7281, and then the data are downloaded into the uPD7281 to be processed. Since a uPD7281 can communicate only through the use of a "token", somehow both the object code and the data need to be transformed into tokens with a certain format which can be understood by the uPD7281 processor. The main task of the assembler is this transformation process from source statements to pre-defined uPD7281 token formats.

When the assembler is invoked along with a source file, it indeed generates an object file consisting of many different types of tokens. Among these tokens, the majority of them can be classified into two basic types of tokens: program related tokens and executable data tokens. The program type tokens are used to download programs to the uPD7281, and the data tokens are used to download the data to be processed to the uPD7281.

Before a uPD7281 can execute a program, assembler generated program tokens are downloaded into the uPD7281's program storage areas such as the Link Table(LT) and the Function Table(FT). These program tokens also include any numerical constants that need to be stored in Data Memory(DM). During the assembly, the assembler automatically generates required object code to set a certain portion of Data Memory with numerical constants used in a program. Therefore, the user need not worry about setting DM with certain constants during the source program design.

After a program download, the execution of the program is initiated by sending data tokens to the uPD7281 to be processed.

CHAPTER 2 BASIC USAGE

2.1 Flow graph

Just as a flow chart is used in programming an ordinary von Neumann type computer, a data flow architectured computer requires a similar programming aid, called a "flow graph". The main difference between a flow chart and a flow graph is as follows. A flow chart for an ordinary von Neumann processor represents how the processor controls the program flow, whereas a flow graph for a data flow processor represents how the processor controls the flow of data.

Specifying programs using a flow graph for a data flow processor has an important advantage over a von Neumann flow chart, because a data flow graph can show the parallelism built in to a program. In other words, a flow graph can show which computations may be executed concurrently using parallel processors. In contrast, a flow chart specified program for a von Neumann processor does not take advantage of concurrency built into a program, since the program is executed sequentially, one computation after another.

Specifying a computation using a flow graph is also a fairly easy task as shown below.

(1) Flow graph of 'Y=A*B+C*D'

The expression 'Y=A*B+C*D' is represented in the flow graph in Fig. 2-1. In the figure, the blocks representing multiplication and addition operations ('X' and '+') are called 'nodes', while the directed arrows which show the flow of data are called 'arcs'.

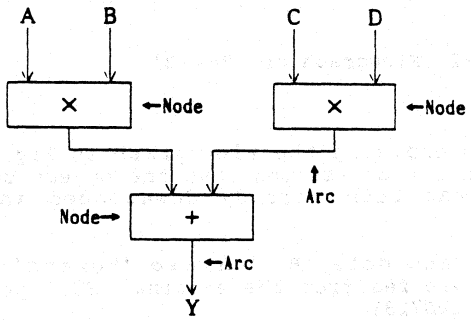


Fig. 2-1 Flow graph of 'Y=A*B+C*D'

(2) Flow graph of 'Y=A+B'

The flow graph of the expression 'Y=A+B" is shown in Fig. 2-2. This flow graph is a specific example of data flow graph in that it actually uses the uPD7281 instruction set and also conforms to the uPD7281 data flow graph specifications.

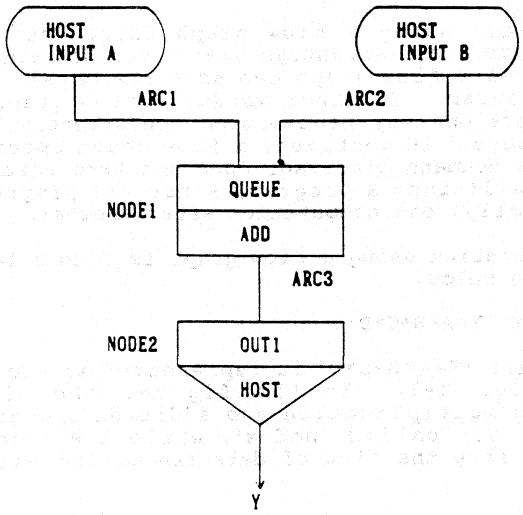


Fig. 2-2 Flowgraph of 'Y=A+B'

The following explains the flow graph in Fig. 2-2 in detail. It should be assumed that the object code for the program has been already downloaded into the uPD7281.

1) Data 'A' and data 'B' (called 'tokens' in the uPD7281) are fed from the external HOST processor into the uPD7281.

2) The tokens A and B flow through the arcs 'ARC1' and 'ARC2' toward the node 'NODE1'.

3) At 'NODE1', the token A and the token B are paired together by "QUEUE" instruction and added together by "ADD" instruction. Both instructions "QUEUE" and "ADD" together are said to form a functional node, named "NODE 1". Each functional node in a flow graph corresponds to a FUNTION statement (explained later) in a source program file.

4) The information (token) resulted from the operation flows toward 'NODE2' through 'ARC3'.

5) Due to the instruction at 'NODE2', the result of 'A+B' is output to HOST as 'Y'.

2.2 General organization of source modules

A source program module of the uPD7281 is composed of a group of statements. This group of statements is logically divided into three sections, named DECLARATION SECTION, EXECUTION SECTION and DATA SECTION. The organization of a source module is shown in Fig. 2-3.

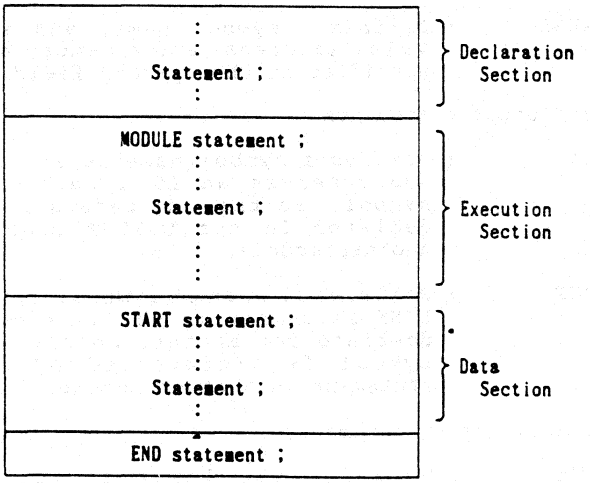


Fig. 2-3 General structure of a source module

Each statement must be terminated by a semicolon, as shown in Fig. 2-3.

2.3 List of statements

Statements of the assembler are functionally divided into seven groups. Each functional group of statements is listed and explained below.

(1) Program structuring statements

MODULE ; Defines a module,and assigns a module name and a module number.

START ; Defines the beginning of the Data Section.

(2) Name definition statements

NAME ; Defines the module name of the output object module.

LITERAL ; Defines a name, and assigns a string of characters to the name.

EQUATE ; Defines a name, and assigns a value and an attribute of an expression specified in the operand field.

ADDRESS ; Defines a symbol name, and assigns a value (address) and a memory attribute specified in the operand field.

(3) Input/Output statements

INPUT ; Defines a symbol name as an LT symbol, and reserves an LT location for the symbol. Further, it refers to a symbol declared in the OUTPUT statement in another module.

OUTPUT ; Declares that an LT symbol, defined by a LINK statement in the same module, is destined for another module. This LT symbol is referenced by an INPUT statement in another module.

(4) Area assignment statements

LOCATE ; Declares the limit of each memory (LT,FT,DM) area which is to be used within a module.

MEMORY ; Defines a symbol name as a DM symbol, and reserves a DM area.

```
DMSETLT     ; Specifies an address of an LT location
              to be used in a DM data setting program.

DMSETFT     ; Specifies an address of a FT location to
              be used in a DM data setting program.
```

(5) Execution statements

```
LINK        ; Reserves an LT memory location(s), and
              creates a template (object code) for
              that location according to the
              subsequent parameters on the same line.

FUNCTION    ; Reserves a FT memory location, and
              creates a template (object code) for
              that location according to the
              subsequent parameters on the same line.

DEFINE      ; Defines a specified symbol as an LT
              symbol, reserves an area, and declares a
              symbol to be referenced by other LINK
              statements.
```

(6) Data definition statement

```
DATA        ; Generates a data token (32 bits) to be
              input to the uPD7281 during the
              execution of a program.
```

(7) Assemble end statement

```
END         ; Ends assembly.
```

2.4 Starting the assembler

Syntax:

```
-------------------------------------------------
  AS7281 source_file_name [Control List] (CR)
-------------------------------------------------
```

To start the uPD7281 Assembler, enter its name after the CP/M-86 prompt. The command AS7281 must be followed by a source file name, followed by the optional control list parameters. For more details on the control parameters, please refer to Section 6.2.

Examples;

1) A> B:AS7281 C:EXAMPL.SRC (CR)

2) A> AS7281 B:AFIN1.SRC PRINT(LST:) (CR)

CHAPTER 3 SOURCE MODULE FORMAT

3.1 Organization of the source module

The source module of the uPD7281 is a group of statements, and it consists of three logically separated blocks called Declaration Section, Execution Section and Data Section. An example of a source module is shown below.

Example;

```
I  EQUATE HOST=0;                              I } Declaration
I                                              I } Section
I  MODULE  EXONE=8;/*** 7281 MN=8 ***/         I
I  INPUT   ARC1,ARC2,ARC3;                     I
I  OUTPUT  ARC7;                               I
I  LINK    ARC4=NODE1(ARC1,ARC2);              I
I  LINK    ARC5=NODE2(ARC3);                   I
I  LINK    ARC6=NODE3(ARC4,ARC5);              I
I  LINK    ARC7=NODE4(ARC6);                   I } Execution
I  FUNCTION NODE1=ADD,QUEUE(QUE1,1);           I } Section
I  FUNCTION NODE2=RDCYCS(FIVE,1);              I
I  FUNCTION NODE3=MUL,QUEUE(QUE2,1);           I
I  FUNCTION NODE4=OUT1(HOST,8);                I
I  MEMORY  QUE1=AREA(1);                       I
I  MEMORY  QUE2=AREA(1);                       I
I  MEMORY  FIVE=5;                             I
I                                              I
I  START;                                      I
I  DATA    EXEC (EXONE,ARC1,2);                I
I  DATA    EXEC (EXONE,ARC2,3);                I } Data Section
I  DATA    EXEC (EXONE,ARC3,0);                I
I                                              I
I  END;                                        I
```

3.1.1 Locations where statements can be used

Statements are classified into four different groups, according to the location where they can be coded. (For detailed explanation of each statement, see Chapter 4, 'Statements'.)

(1) Statements which can be used in Declaration Section and Execution Section

LITERAL, EQUATE, ADDRESS

(2) Statements which can be used in Execution Section only

MODULE, NAME, INPUT, OUTPUT, LOCATE, MEMORY, LINK, FUNCTION, DMSETLT, DMSETFT, DEFINE

(3) Statements which can be used in Data Section only

START, DATA

(4) Statements which must be used to terminate the source module

END

Aside from the above statements, other general control statements can also be used in the source module. For more information, refer to Section 6.2.

3.1.2 Organization of the Declaration Section

In a source program file, the Declaration Section is located at the top most position within a source file. In this section, the only allowed statements are LITERAL, EQUATE, and ADDRESS statements.

The names defined in the Declaration Section can be referenced by any statements in the source module. Furthermore, if the Declaration Section is not necessary, it can be omitted.

Fig. 3-1 shows the organization of the Declaration Section.

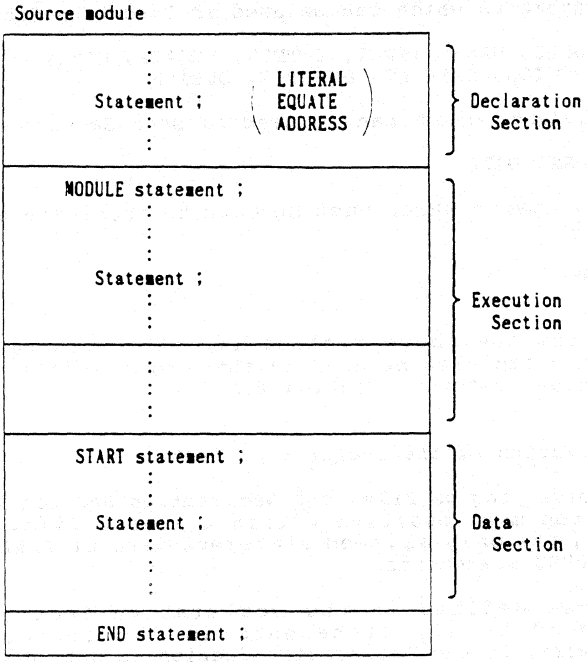


Fig. 3-1 Structure of the Declaration Section

3.1.3 Organization of the Execution Section

The Execution Section of the code immediately follows the Declaration Section. The only statements allowed within the Execution Section are MODULE, NAME, INPUT, OUTPUT, LOCATE, MEMORY, LINK, FUNCTION, DMSETLT, DMSETFT and DEFINE statements. The Execution Section statements are the actual instructions to be executed by the uPD7281 processor.

Since the uPD7281 utilizes data flow architecture, many uPD7281s can be connected in cascade to form a parallel processing machine. In this case, the Execution Section in a source file must include all of the instructions for all of the uPD7281s used in the system. Thus, if a program for more than one uPD7281 is to be coded in the Execution Section, each module must be separated by a MODULE statement.

Up to fourteen modules can be coded in one source module. A name or a label defined in the Execution Section is valid only in the module where they are defined. However, a label declared by the INPUT and OUTPUT statements can also be referenced from other modules.

There may be some instances where only the Declaration Section and the Data Section are required for a source file. In such cases, the Execution Section may be omitted. The organization of the Execution Section is shown in Fig. 3-2.

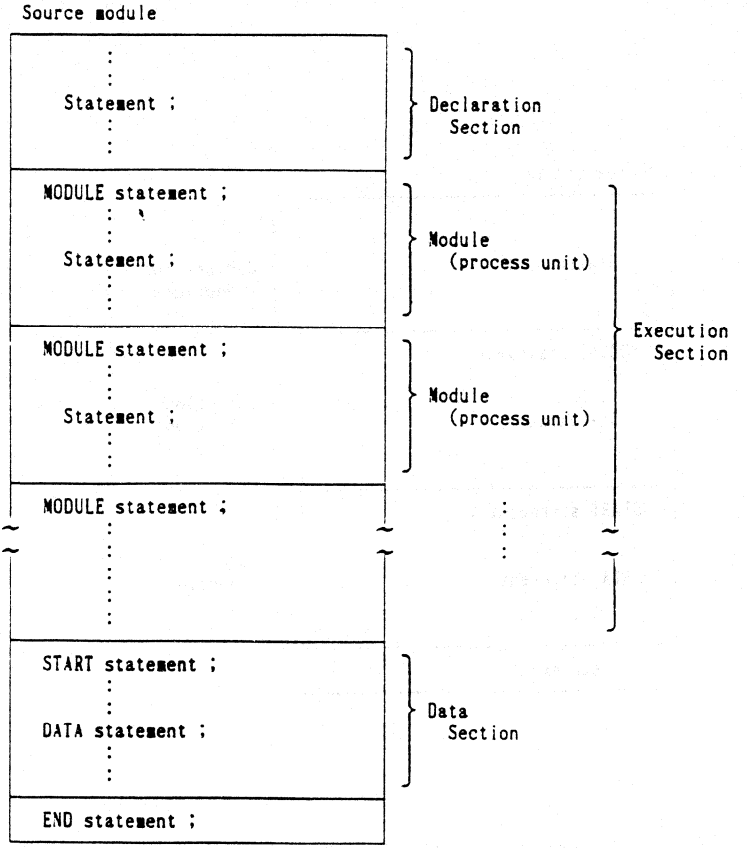


Fig. 3-2 Structure of the Execution Section

3.1.4 Organization of the Data Section

The Data Section of a source file is placed after the Execution Section. The only statements allowed in the Data Section are START and DATA statements. If the Data Section is not necessary for a source file, it may be omitted. The organization of the Data Section is shown in Fig. 3-3.

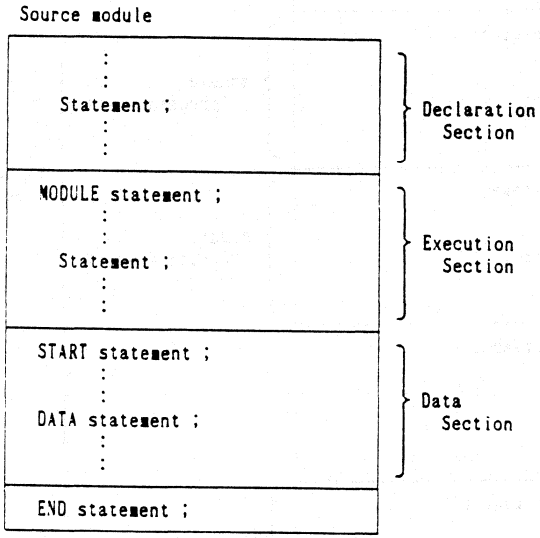


Fig. 3-3 Structure of the Data Section

3.2 Statements

A statement in the uPD7281 assembly language tells AS7281 what action to perform. There are two different types of statements: instructions and directives. Instructions are translated by the assembler into uPD7281 machine codes. Directives are not translated into machine codes but instead direct the assembler to perform certain functions.

Each assembly language statement must terminate with a semicolon (;). All carriage returns (CR) and line feeds (LF) within a source file are regarded as spaces by the assembler. Therefore, a statement may take several lines as long as the statement is valid.

A statement is composed of three fields, namely, a statement field, a symbol field and an operand field. The configuration of the statement is shown below.

Statement field	Symbol field	Operand field
MODULE	EXONE	= 8 ;

At least one space is required to separate the Statement field and the Symbol field.

When both the symbol field and the operand field are necessary, they are separated by an equal sign (=). A blank (or blanks) may be inserted between the symbol and equal sign and/or equal sign and the operand.

The statements are coded in a free format; as long as a statement is coded in the order indicated above (i.e., statement field, symbol field and operand field) it can be started in any arbitrary column. However, as mentioned above, each field must be separated by a space (or spaces) or an 'equal' sign.

Up to 116 characters can be coded in a single line, but it is not possible to code more than one statement in the same line. Each statement field is explained below.

3.2.1 Statement field

Only those statements listed in Section 2.3 may be used in the statement field. A detailed explanation for each statement can be found in Chapter 4.

Examples:

1) NAME EXAMPL; ..... The program is named 'EXAMPL'. (There is both a statement field and a symbol field in this example.)

2) START; ..... Defines the beginning of the Data Section. (There is only the statement field in this example.)

3.2.2 Symbol field

A symbol may be defined in the symbol field using a statement and an operand (operands). Any symbol names reserved for the assembler may not be used in the symbol field. In Section 3.4, more details on the symbol field usage are given.

3.2.3 Operand field

The operand field may contain an instruction mnemonic and other operands to the instruction. In the case where more than one operand are required, they must be separated by a comma(s) (,) or a space(s). For more detail on the operand field, refer to Section 3.5.

3.2.4 Comments

A comment field is enclosed by a '/*' at the beginning and a '*/' at the end. Since the assembler treats the comment field as one blank space, the comment field enclosed by '/*' and '*/' may appear anywhere in a source file. Additionally, a string of characters followed by a semicolon up to a CR is also regarded as a comment. Comments improve the readability of program. Of course, the comment field is optional.

Examples:

```
/* THIS IS A COMMENT LINE */
MODULE EXONE = 8 ; /** 7281 MN=8 **/
MODULE /** 7281 MN=8 **/ EXONE = 8 ;
MODULE EXONE = 8 ;  *7281 MN=8*
```

3.2.5 Tab function

For a clear arrangement of a source statement, a tab function may be used. A tab function positions the cursor on multiples of 8 character locations so that the statement fields can be aligned in columns.

Examples:

```
Statement field  Symbol field   Operand field
---------------  ------------  -------------
  MEMORY          QUE1        =    AREA(16)    ;
  LINK            D           =    FUNC1(A,B)  ;
                  ↑                ↑
                  └────────────────┘
                                   column (8n+1)
                                     n=1, 2, 3, ...
```

When using the assembler's tab function during a source file creation, press the 'TAB' key before entering the first character in each statement field. The 'TAB' key corresponds to 'HT' (code=09H) in the ASCII code table.

When the source module created in this way is assembled and listed, each field on the list is aligned in columns of multiples of 8.

3.3 Character set

The character set available for writing source statements is shown below.

(1) Alphabetic characters (upper-case and lower-case letters)

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

(2) Numeric characters

```
0 1 2 3 4 5 6 7 8 9
```

(3) Special characters

```
blank ? @ _ , . + - * / ( ) $
= ; ' !
```

(4) Control characters

```
HT LF CR FF DEL
```

NOTE: Characters available for coding a symbol are limited. For detail, refer to Section 3.4.1, 'Rules on coding symbols'. Lowercase letters used in a symbol or in a reserved word are interpreted as uppercase letters.

3.3.1 Alphanumeric characters

Alphabetic characters and numeric characters together are called alphanumeric characters.

3.3.2 Numeric digits

Binary digits ; The two characters, 0 and 1, are called binary digits.

Octal digits ; The 8 characters - the binary digits, 2, 3, 4, 5, 6 and 7 - are called octal digits

Decimal digits ; The 10 characters - the octal digits, 8, and 9 - are called decimal digits.

Hexadecimal digits; The 16 characters - the decimal digits, A, B, C, D, E, and F - are called hexadecimal digits.

3.3.3 Use of special characters

Character	Name	Meaning
	space	Delimiter for separating each field.
?	question mark	Alphabet-equivalent character.
@	unit-price mark	Alphabet-equivalent character.
_	underscore	Alphabet-equivalent character.
,	comma	Delimiter for operands.
.	period	Delimiter for identifying a control-bit.
+	plus	Positive sign or add operator.
-	minus	Negative sign or subtract operator.
*	asterisk	Multiply operator or a part of delimiter for comment.
/	slash	Divide operator or a part of delimiter for comment.

(.....)	left/right parenthesis	For change in precedence of operations.
$	dollar mark	This character is ignored by the assembler.
=	equal sign	Separator between symbols and operands.
;	semicolon	End of statement mark.
'	apostrophe	Beginning/end mark of a character string.
!	exclamation mark	Representation of illegal characters on the assembly list.

3.3.4 Use of control characters

Character	Name	Meaning
TAB	horizontal tab	Character equivalent to spaces.
LF	line feed	End mark of a line.
CR	carriage return	
FF	form feed	Has no meaning.
DEL	delete code	

3.4 Symbols

Writing a symbol in the symbol field of a statement is called defining a symbol. The symbols handled by the assembler are those that become names or labels.

(1) Symbols to be names

A symbol defined by either the NAME, MODULE, LITERAL or EQUATE statement becomes a name. In the case where a number is defined to a symbol by the EQUATE statement, that symbol becomes NUMBER-attributed.

(2) Symbols to be labels

A symbol defined in either an execution statement (LINK, FUNCTION, DEFINE), an input/output statement (INPUT, OUTPUT), an ADDRESS statement, or a MEMORY statement becomes a label. A symbol that becomes a label has an address and an attribute. The three different types of attributes for labels are the following.

1) LT (Link Table) attribute

2) FT (Function Table) attribute

3) DM (Data Memory) attribute

A symbol may be assigned different attributes depending upon the type of the statement used to define the symbol. Table 3-1 shows attribute assignments depending on the types of statement used.

Table 3-1 Attributes of symbols and types of statements

Symbol's attribute	Types of statements
LT	LINK, INPUT, OUTPUT, DEFINE, EQUATE*, ADDRESS (when LT is specified).
FT	FUNCTION, EQUATE*, ADDRESS (when FT is specified).
DM	MEMORY, EQUATE*, ADDRESS (when DM is specified).
NUMBER	EQUATE*, LITERAL (the value itself).

(*) This differs according to the attribute of the expression coded in the operand field.

3.4.1 Rules on using symbols

A symbol must satisfy all of the following 5 rules.

(1) A symbol must be coded using alphanumeric and/or alphabet equivalent characters (@,?,_). However, decimal digits (0 ~ 9) cannot be used as the first character of a symbol.

(2) A symbol must be from 1 to 8 characters long. If a symbol is coded with more than 8 characters, only the first 8 characters are valid, and the characters after the 8th are ignored.

(3) Reserved words cannot be used as symbols. (Refer to Appendix 1, 'List of reserved words'.)

(4) The same symbol cannot be defined more than once.

(5) A symbol must be terminated with a '='.

Examples:

1) Examples of legal symbols.

```
MODULE SAMPLE = 1;
LINK   D      = FUNC1(A,B);
FUNCTION FUNC1 = ADD;
```

2) Examples of illegal symbols.

```
EQUATE 3ABC  = 3;  .... A decimal digit cannot be used
                        as the first character of a
                        symbol.
LINK E FUNC2(C,D); ..... The symbol (E) must be
                        followed with a '='.
MEMORY DATA  = 2;  ..... Reserved word (DATA) cannot be
                        used.
```

3) Example of a long symbol.

```
EQUATE ABCDEFGHK = 3; ...Characters after the 8th are
                        ignored, and ABCDEFGH is
                        defined as a symbol.
```

3.4.2 Remarks on coding symbols

There are three types of statements in regard to whether or not the statement requires a symbol in the symbol field. The three different types of statements are listed in groups as follows:

(1) Statements that require a symbol

MODULE, NAME, LITERAL, EQUATE, ADDRESS, MEMORY, FUNCTION, DEFINE

(2) Statements in which a symbol is optional

LINK *

(*) Even when no symbol is coded in the LINK statement, '=' must be coded.

(3) Statements that must not be used with a symbol

START, END, INPUT, OUTPUT, LOCATE, DMSETLT, DMSETFT, DATA

Examples:

1) Examples of legal symbols.

```
LITERAL FUNC = 'FUNCTION';
FUNC    FMUL = MUL;
```

2) Examples of illegal symbols.

```
EQUATE  3;          ..... Symbol must be coded.
DMSETLT SIZ = 128;  ..... Symbol cannot be coded.
```

3.5 Coding format for the operand field

In an operand field, five different types of operands may be used. These five different types of operands are:

1) Reserved words
2) Symbols
3) Numeric constants
4) Expressions
5) Character strings

3.5.1 Reserved words

(1) Instructions (operands for FUNCTION statement)

1) AG&FC instructions

QUEUE	RDCYCS	RDCYCL	WRCYCS
WRCYCL	RDWR	RDIDX	PICKUP
COUNT	CUT	DIVCYC	DIV
DIST	CONVO	SAVE	CNTGE

2) PU instructions

OR	AND	XOR	ANDNOT
NOT	ADD	ADDSC	SUB
SUBSC	MUL	MULSC	NOP
NOPSC	INC	DEC	SHL
SHLBRV	SHR	SHRBRV	CMPNOM
CMP	CMPXCH	ACC	COPYC
GET1	SET1	CLR1	ANDMSK
ORMSK	CVT2AB	CVTAB2	ADJL

3) GE instructions

COPYBK	COPYM	SETCTL

4) OUT instructions

OUT1 OUT2

(2) Input tokens (operands for DATA statement)

SETLT	SETFTR	SETFTL	SETFTT
RDLT	RDFTR	RDFTL	RDFTT
CRESET	SETMD	SETBRK	DUMP
CBRK	VAN	PASS	EXEC
DMS(DM Set)		DMR(DM Read)	

(3) Parameters of PU instructions

XCH	BRC	EQ	LT
LE	GT	GE	NE
OVF	CNOP	X	Y
XX	XY		

(4) Operators

OR	AND	NOT	XOR
SHL	SHR		

(5) Others

LT	FT	DM	AREA
C			

3.5.2 Symbols

A symbol(s) defined in the symbol field can be used as an operand(s) for a statement.

3.5.3 Numeric constants

Numeric constants are classified into binary, octal, decimal and hexadecimal constants. As it is shown in the examples below, a negative constant can be expressed by putting the '-' (minus) sign before the numeric constant.

(1) Binary constants

A binary constant is a number represented by binary digits and followed by a letter 'B' at the end.

Examples:

011B ..... Represents the decimal number '3'.
-101B ..... Represents the decimal number '-5'.

(2) Octal constants

An octal constant is a number represented by octal digits and followed by a letter 'O' at the end.

Examples:

037O ..... Represents the decimal number '31'.
-17O ..... Represents the decimal number '-15'.

(3) Decimal constants

A decimal constant is a number represented by decimal digits, followed or not by a letter 'D' at the end. If no radix suffix is attached to a constant, it is assumed to be a decimal number.

Examples:

123D, 0123 ..... Both represent the decimal number '123'.
-456D ..... Represents the decimal number '-456'.

(4) Hexadecimal constants

A hexadecimal constant is a number represented by hexadecimal digits and followed by a letter 'H' at the end. However, if a hexadecimal constant begins with one within A and F, it must be preceded by a '0'.

Examples:

13H ..... Represents the decimal number '19'.
0FFH ..... Represents the decimal number '255'.
-1AH ..... Represents the decimal number '-26'.

In the case where C-bit (control bit) is added to the numeric constant, '.C' is added after the numeric constant. When '.C' is omitted in a number, the C-bit is assumed to be 0.

Examples:

```
0123H.C  .......    Represents C-bit='1'.
0123H    .......    Represents C-bit='0'.
```

3.5.4 Expressions

An expression may be a constant, a symbol, or a combination of constants and symbols linked by operators. For more information about the operators, please Section 3.6.

```
Examples:   6/2+1
            SYM1+3      (SYM1 is a symbol.)
```

3.5.5 Character strings

A character string is enclosed with apostrophes. When the apostrophe itself is to be used as a character, two consecutive apostrophes must be entered.

```
Examples:     'FUNCTION'
              'A'''
```

3.6 Operators

The AS7281 operators fall into the following categories: arithmetic, logical, shift, and others. As mentioned in Section 3.5.4, a combination of constants and symbols linked by operators is called an expression.

(1)Arithmetic operators

+, -, *, /

2) Logical operators

OR, AND, NOT, XOR

3) Shift operators

SHR, SHL

4) Other operators

(,)

When there are two operands in a statement, the operand on the left-hand side of the operator will be designated as the first operand, and that on the right-hand side will be called the second operand. (Where there is only one operand, it is called the operand.) The following will show the general format for unary and binary operators.

(1) General format of unary operators

```
-----------------------
  operator  operand
-----------------------
```

Examples:

```
+5     ......  Adds the unary operator '+' before the
               decimal number '5'.
-3AH   ......  Adds the unary operator '-' before the
               hexadecimal number '3AH'.
-SYM1  .....   Adds the unary operator '-' before the
               symbol 'SYM1'.
```

(2) General format of binary operators

```
-------------------------------------------
  first_operand  operator  second_operand
-------------------------------------------
```

Examples:

```
SYM1 AND 02H ....  The binary operator 'AND' connects
                   the first operand 'SYM1' and the
                   second operand '02H'.
123H - SYM1  ....  The binary operator '-' connects
                   the first operand '123H' and the
                   second operand 'SYM1'.
```

NOTE: Operators such as logical operators (OR, AND, NOT, XOR) and shift operators (SHR, SHL) must be preceded and followed by blanks (or a blank).

3.6.1 Order of precedence of operators

The order of precedence among operators is predetermined. An expression containing more than one operator is evaluated according to their order of precedence. In the case where operators with the same order of precedence exist, the evaluation proceeds from the left to the right.

Table 3-2 shows the order of precedence of operators. (The highest precedence is set to 1.)

Table 3-2 Order of precedence of operators

Precedence	Operator
1	(,) (See NOTE)
2	NOT, + (unary operator), - (unary operator)
3	*, /, SHL, SHR
4	+ (binary operator), - (binary operator)
5	AND
6	OR, XOR

NOTE: The order of evaluation in an expression can be changed by using the operator '( )'. For details, refer to Section 3.6.5.

3.6.2 Arithmetic operators

Using the arithmetic operators as binary operators, addition, subtraction, multiplication and division operations can be performed.

All arithmetic operations are performed on 17-bit data consisting of a sign bit and a 16-bit absolute value. The control bit (C-bit) of the operation's result is the same as that of the first operand.

(1) +

This operator performs an addition operation between the first and the second operands.

Examples: 2+3 (The result is 5.)

(2) -

This operator performs subtraction operation between the first and the second operands.

Examples: 3-2 (The result is 1.)

(3) *

This operator multiplies the first operand by the second one.

Example: 3*2 (The result is 6.)

(4) /

This operator divides the first operand by the second operand and returns the integer part of the result.

Example: 8/3 (The result is 2.)

3.6.3 Logical operators

A logical operator performs bitwise logical operations using the first and the second operands. Both the control bit (C-bit) and the sign bit (S-bit) of the result become the same as those of the first operand.

(1) OR

This operator performs bitwise logical "OR" operations using the two operands and returns the resulting value.

Example: 1234H OR 5678H (The result is 567CH.)

(2) AND

This operator performs bitwise logical "AND" operations using the two operands and returns the resulting value.

Example: 1234H AND 5678H (The result is 1230H.)

(3) NOT

This operator returns the logical negation of the operand.

Example: NOT 1234H (The result is 0EDCBH.)

(4) XOR

This operator performs bitwise logical "XOR" operations using the two operands and returns the resulting value.

Example: 1234H XOR 5678H (The result is 444CH.)

3.6.4 Shift operators

A shift operator performs a shift operation on the bit pattern stored in the first operand. The number of bits to be shifted is contained in the second operand. Both the control bit (C-bit) and the sign bit (S-bit) of the result remain the same as those of the first operand.

(1) SHR (SHIFT RIGHT)

This operator performs shift right operation on the

first operand using the number of shifts specified by the second operand. Bit positions at the MSB side corresponding to the number of shifts are occupied by zeros.

Example: 1234H SHR 4 (The result is 0123H.)

(2) SHL (SHIFT LEFT)

This operator performs shift right operation on the first operand using the number of shift specified by the second operand. Bit positions at the LSB side corresponding to the number of shifts are occupied by zeros.

Example: 1234H SHL 4 (The result is 2340H.)

3.6.5 Other operators

(1) (,)

This operator has the highest precedence. It causes the expression between '(' and ')' to be evaluated first. When '()'s are nested, evaluation proceeds from the expression within the innermost '()' to the expression within the outermost '()'.

Example: '1+2+3*1+2' equals '8', but '(1+2+3)*(1+2) equals '18'.

(((1+2)*3)+5)/7 (The result is 2.)

3.6.6 Restrictions on the operators and their operands

There are certain restrictions on combinations of operators and their operands. A symbol in a statement is assigned an attribute among the following 4 types of attributes.

1) LT attribute ....A memory address of the Link Table.

2) FT attribute ....A memory address of the Function Table.

3) DM attribute ....A memory address of the Data Memory.

4) NUMBER attribute .... The value itself.

The restrictions on combination of operators and the operands are summarized in the following two tables.

Table 3-3 Restrictions on combination of operands (Binary operators)

X; Illegal combination

Combination of operands		Operator and attribute of the result					
First-operand's attribute	Second-operand's attribute	+	-	*,/	SHR,SHL	OR,AND, XOR	
LT	LT	X	NUM.	X	X	X	
LT	FT	X	X	X	X	X	
LT	DM	X	X	X	X	X	
LT	NUM.	LT	LT	X	LT	LT	
FT	LT	X	X	X	X	X	
FT	FT	X	NUM.	X	X	X	
FT	DM	X	X	X	X	X	
FT	NUM.	FT	FT	X	FT	FT	
DM	LT	X	X	X	X	X	
DM	FT	X	X	X	X	X	
DM	DM	X	NUM.	X	X	X	
DM	NUM.	DM	DM	X	DM	DM	
NUM.	LT	LT	X	X	X	X	
NUM.	FT	FT	X	X	X	X	
NUM.	DM	DM	X	X	X	X	
NUM.	NUM.	NUM.	NUM.	NUM.	NUM.	NUM.	

NOTE; NUM.=NUMBER

Table 3-4 Restrictions on operands (unary operators)

X; Illegal combination

Operand's attribute	Operator and attribute of the result			
	+	-	NOT	
LT	LT	X	X	
FT	FT	X	X	
DM	DM	X	X	
NUMBER	NUMBER	NUMBER	NUMBER	

CHAPTER 4 STATEMENTS

This chapter explains each uPD7281 assembly language statement in detail.

4.1 Program structuring statements

There are two types of program structuring statements:

MODULE
START

4.1.1 MODULE

(1) Function

Defines the beginning of the Execution Section in a module and assigns a module name and a module number.

(2) Coding format

Statement field	Symbol field	Operand field
MODULE	module name	= expression ;

(3) Description

1) As mentioned in Section 3.1, a program consisting of a group of statements for one uPD7281 is called a module.

The MODULE statement defines the beginning of the Execution Section for one module, and it assigns the module name and the module number corresponding to the MN (Module Number) of the uPD7281. The value of the expression specified in the operand field is the Module Number.

2) More than one module can be coded in a source module, but modules with the same name cannot be coded.

3) The module number specified in the operand field must have its value within the range of 1 to 14. When 0 or 15 is specified, an error message (warning) will be displayed. (0 is the number assigned to the Host, and the number 15 is assigned to the VANISH token.)

4) A symbol can be coded as an operand of an expression, but the symbol must have the NUMBER attribute. Furthermore, a symbol coded at this point must be defined before this statement.

Examples:

1. The module name is defined to be 'AFFIN' and the module number is defined to be '8'.

```
      :
      :
MODULE  AFFIN=8;
      :
      :
```

2. This is an example where a symbol is used as the operand of an expression. (Symbol 'LOWMN' is assumed to have the NUMBER attribute, and its value is assumed to be '6'.) 'EXAM' therefore is assigned the value '8'.

```
      :
      :
MODULE  EXAM=LOWMN+2;
      :
      :
```

4.1.2 START

(1) Function

Defines the beginning of the Data Section.

(2) Coding format

Statement field	Symbol field	Operand field	
START			;

(3) Description

1) The START statement defines the beginning of the Data Section for a program. The DATA statements immediately follow the START statement.

2) This statement is coded only once in a source module.

3) Statements other than DATA statement or END statement cannot be coded after START statement.

Example:

```
    :
    :
START;
DATA    EXEC(EXAM,ARC1,2);
DATA    EXEC(EXAM,ARC2,3);
DATA    EXEC(NEXT,THEID,HEXDATA);
    :
    :
```

4.2 Name definition statements

There are four types of name definition statements:

NAME
LITERAL
EQUATE
ADDRESS

4.2.1 NAME

(1) Function

Defines the module name of the output object module.

(2) Coding format

Statement field	Symbol field	Operand field
NAME	module name	;

(3) Description

1) The NAME statement defines the name of the object module output. The object module is produced by assembling a user source module.

2) The NAME statement is coded only once in the Execution Section of a source module. Furthermore, if the NAME statement is not coded, a blank will be assigned to the output module name.

Example:

```
   :
   :
MODULE EXAM=8;
NAME   MAIN;
   :
   :
```

4.2.2 LITERAL

(1) Function

Defines a name and assigns a character string.

(2) Coding format

```
Statement field   Symbol field   Operand field
---------------   ------------   -----------------
   LITERAL            name      = 'character string';
```

(3) Description

1) The LITERAL statement defines a name and assigns a string of characters to the name. If the LITERAL assigned name is used anywhere in the following source program, the use of the name will have the same effect as the use of the character string.

   If a frequently used symbol, operator, statement or constant has a long string of characters, the LITERAL statement can be used to shorten the long string of characters thereby increasing the efficiency in source program writing.

2) The assigned character string must be enclosed by apostrophes ('). When an apostrophe (') itself is to be coded in a character string, two consecutive apostrophes must be coded.

3) Names defined by this statement are not output to the object module file.

   Example:

```
LITERAL FUNC = 'FUNCTION';
FUNC FADD = ADD; <---------- It has the same meaning as
                             FUNCTION FADD =ADD;
```

4.2.3 EQUATE

(1) Function

Defines a name and assigns it both the value and the attribute of the expression specified in the operand field.

(2) Coding format

```
Statement field  Symbol field   Operand field
---------------  ------------   -------------
    EQUATE       name         = expression    ;
```

(3) Description

1) The EQUATE statement defines a name and assigns it both the value and the attribute of the expression specified in the operand field. The expression to be coded in

the operand field must have an absolute value which must be determinable by the assembler.

2) The name defined by this statement can be used in the operand fields of the following statements within a source module. If the EQUATE defined name is used in any of the following statements, its use has the same effect as the use of the expression.

Examples:

```
EQUATE ZERO = 0; <--------- Assigns 0 to symbol 'ZERO'.
EQUATE NAME01 = SYM01; <--- Assigns the value of symbol
                            'SYM01' to symbol 'NAME01'.
                            'SYM01' is assumed to be
                            absolute and defined before
                            this statement.
```

4.2.4 ADDRESS

(1) Function

Defines a symbol name, and assigns both the value of the expression and the memory attribute specified in the operand field to the defined symbol.

(2) Coding format

Statement field	Symbol field	
ADDRESS	symbol name	=

Operand field

```
{ LT                                }
{ FT { (PU)    }                    }   AT  expression ;
{    { (OUT)   }                    }
{    { (GE)    }                    }
{    { (AG&FC) }                    }
{ DM                                }
```

(3) Description

1) The ADDRESS statement defines a symbol name for a location in one of the uPD7281's internal memories (LT, FT, or DM), and assigns to the symbol both the memory attribute and the value (address) of the expression specified after 'AT' in the operand field.

2) The value of the expression to be coded in 'AT expression' must have the attribute type NUMBER. Furthermore, the value of the expression must be within the range indicated in Table 4-1, according to the specified memory attribute.

Table 4-1

Type of memory	Memory attribute	Value of the expression	Memory configuration
LT	LT	0 ~ 127	16bits x 128
FT	FT	0 ~ 63	40bits x 64
DM	DM	0 ~ 511	18bits x 512

3) After a symbol name is defined for a location in a uPD7281's memory (LT, FT, or DM), this location can be referenced by the following statements using the defined symbol.

Examples:

```
        :
        :
ADDRESS LTSYM01=LT AT 0; <------ Defines 'LTSYM01' to
                                 address 0 of the Link
                                 Table.
ADDRESS DMSYM01=DM AT 100H; <--- Defines 'DMSYM01' to
        :                        address 100H of the
        :                        Data Memory.
LINK    AAA    =AFUNC(BBB,DMSYM01+2); <------ Reference
        :                                     using symbol
        :                                     'DMSYM01'.
```

Caution !!!

GE instructions use the first 128 DM locations to store the incremental/decremental values. Therefore, user should be very careful when using the first 128 DM locations for constants or tables.

For a GE instruction, the internal constant known as 'delta' is stored within the first 128 locations of DM. The actual DM address used for a particular GE instruction is equal to the ID value of the token accessing the GE instruction.

4.3 Input/Output statements

There are two types of Input/Output statements:

INPUT
OUTPUT

4.3.1 INPUT

(1) Function

Defines a symbol name as a LT symbol and reserves a LT memory location for the symbol. Since an output token from a uPD7281 may be the input token to another uPD7281, the INPUT statement refers to a symbol declared in the OUTPUT statements in other modules.

(2) Coding format

Statement field	Symbol field	Operand field
INPUT		symbol name[AT expression][,...];

(3) Description

1) The INPUT statement defines the symbol name coded in the operand field as a LT attribute symbol and reserves a LT memory location for the symbol. Furthermore, the statement refers to other symbols declared by the OUTPUT statements coded in other modules of the same source program.

2) If the 'AT expression' is coded after the symbol name, the symbol is associated with an absolute LT memory location pointed to by the value of the expression. The value of the expression must be within 0 and 127 and is restricted to NUMBER type attribute only.

3) The symbol name defined by this statement must be referenced by either the symbol-name3* or the symbol-name4* in a LINK statement coded in the same module. Furthermore, a symbol defined by this statement cannot be referenced by any statements other than LINK statements in the same module.

(*) Refer to the coding formats in Section 4.5.1.

```
Example:
        :
        :
INPUT ATKN AT 10H,BTKN; <----- Defines symbols  'ATKN'
        :                      and 'BTKN' and reserves
        :                      a LT memory area.
        :                      'ATKN' is assigned to a
        :                      LT memory address of
        :                      10H.
        :
LINK  CTKN=AFUNC(ATKN,BTKN);
        :
```

4.3.2 OUTPUT

(1) Function

Declares that a LT symbol is an output token so that it may be referenced by an INPUT statement of another module of the same source program. The declared LT symbol must be defined using a LINK statement in the same module where the 'OUTPUT symbol-name' statement resides.

(2) Coding format

Statement field	Symbol field	Operand field
OUTPUT		symbol name[,...];

(3) Description

1) The OUTPUT statement declares the symbol in the operand field to be referenced by other modules. The symbol declared as an output link must also be defined by a LINK statement in the same module. Once a symbol is declared as an output link, the symbol can be referenced by an INPUT statement in other module.

2) A symbol defined by the OUTPUT statement must also be defined as 'symbol-name1'* of a LINK statement in the same module. Furthermore, an OUT instruction (either OUT1 or OUT2 instruction) must use the OUPUT defined symbol in the operand field of the FUNCTION statement.

```
Example:
      :
      :
MODULE    EXAM1=8; <---- Assigns the module number '8'.
      :
      :
OUTPUT    CTKN; <------- Defines/declares symbol 'CTKN'.
LINK      CTKN=FOUT(ATKN,BTKN); <---- 'CTKN' is coded
                                       as 'symbol-name1'.
FUNCTION FOUT=OUT1(9,CTKN); <- An example where OUT1 is
      :                        using the 'CTKN' in the
      :                        operand field of a
      :                        FUNCTION statement.
MODULE    EXAM2=9; <---------- Assigns the module number
      :                        '9' to another module.
      :
INPUT     CTKN;<---------------'CTKN' being referenced
      :                        by an INPUT statement in
      :                        other module.
      :
      :
END;
```

(*) Refer to the coding format in Section 4.5.1.

4.4 Memory allocation statements

There are four types of statements used to allocate memory areas:

LOCATE
MEMORY
DMSETLT
DMSETFT

4.4.1 LOCATE

(1) Function

Defines the memory areas for each memory (LT, FT, or DM) to be used by a module.

(2) Coding format

Statement field	Symbol field	Operand field
LOCATE		{LT / FT / DM} (expression1, expression2)[,...];

(3) Description

1) The LOCATE statement allocates memory areas (in LT, FT or DM) for the symbols defined by LINK, FUNCTION and MEMORY statements. The address of the lower-bound in the defined memory area is specified by 'expression-1', and the upper-bound address is specified by 'expression-2'.

2) Only NUMBER attribution type expressions are allowed for the 'expression-1' and 'expression-2' for the LOCATE statements.

Table 4-2 shows the ranges in which expression-1 and expression-2 may be specified along with the default values.

Table 4-2

MEMORY	Permitted value ranges for the expression-1 and the expression-2	Default values expr-1	Default values expr-2
LT memory	$0 \leq$ expr-1 $\leq$ expr-2 ≤ 127	0	127
FT memory	$0 \leq$ expr-1 $\leq$ expr-2 ≤ 63	0	63
DM memory	$0 \leq$ expr-1 $\leq$ expr-2 ≤ 511	0	511

3) Areas of different memories can be coded in a single LOCATE statement. There is no restriction on the order in which memory statements may appear.

Example:

```
 :
 :
LOCATE LT(0,3FH),FT(0,1FH),DM(0,7FH); <-- Areas of different
 :                                        memories can be
 :                                        coded.
```

4.4.2 MEMORY

(1) Function

Defines a symbol name as a DM symbol and reserves a DM area for the symbol.

(2) Coding format

Statement field	Symbol field	
MEMORY	symbol name	=

Operand field

```
{    expression-1                          }
{AREA (expression-2[,expression-3])}

      [,...][  AT  expression-4];
```

(3) Description

1) The MEMORY statement defines the symbol name in the symbol field as a DM attribute symbol, and it reserves a DM area for the symbol.

2) When expression-1 is specified in the operand field of a MEMORY statement, one location in the DM is reserved for the symbol. When several expression-1's are specified, as many memory locations as the number of 'expression-1's specified are reserved in DM. Furthermore, only the first address of the reserved area(called DM Base address) is assigned to the defined symbol. The reserved memory locations are initialized with the values in the 'expression-1's.

3) When expressions are used with AREA in the operand field, the number of DM locations equal to the value of expression-2 are reserved, and the value of expression-3 is used to initialized all reserved memory locations. In addition, only the first address(DM Base address) is assigned to the symbol.

   When ',expression-3' is omitted, the number of DM areas corresponding to the value of expression-2 are reserved, but the contents of the memory are not initialized. (That is the same case as where '?' is coded in 'expression-3'.)

4) When ' AT expression-4' is coded in the operand field, a DM area is reserved for the symbol beginning at the absolute starting memory address of 'expression-4', and the starting address is assigned to the defined symbol.

5) The following lists the restrictions on the use of expression-1,-2, -3, and -4.

   a. In expression-1 and expression-3, the constants used must have NUMBER type attribute, and its value must be 65535 or less. If '?' is specified for expression-3, a memory area is reserved, but its contents are not initialized.

   b. Any constant used as an 'expression-2' must have its value between 1 and 65535.

   c. The constant used as an expression-4 must be a number whose value is within the range of 0 and 511, and its attribution type is NUMBER.

```
  Examples:
      :
      :
MEMORY AREA1=1; <---------------------- One DM location is
                                        reserved and
MEMORY QUE1=AREA(10,?) AT 10H; <-----┐  initialized to the
                                     │  value of '1'.
MEMORY QUE2=AREA(10); <------------┐ └──┐
                                  │    └─ 10 words of
MEMORY XYZ=4,5,6; <-----------┐   └─────┐  DM area are
                              └─────────┐  reserved
MEMORY WWW=AREA(2,1),AREA(4,3),6,7; <─┐ │  starting from
      :                               │ │  address 10H.
      :                               │ │  Their contents
                                      │ │  are not
                                      │ │  initialized.
                                      │ └─ 10 words of DM area are reserved.
                                      │    Their contents are not initialized.
                                      └──── 3 words of DM area are reserved,
                                            and the values 4, 5, and 6 are
                                            initialized to 3 consecutive DM
                                            memory locations.
                                      └──── 8 words of DM area are reserved,
                                            and the values, 1,1,3,3,3,3,6, and
                                            7 are initialized to 8 consecutive
                                            locations starting at the symbolic
                                            location "WWW".
```

4.4.3 DMSETLT

(1) Function

Specifies the address of a LT memory location to be used for setting data into DM.

(2) Coding format

Statement field	Symbol field	Operand field
DMSETLT		expression ;

(3) Description

1) Since there is no specialized token format to set a DM location(s), a program must be written to initialize any DM memory locations used as constants, variables and tables before the actual program execution. Fortunately, however, the uPD7281 assembler is capable of generating the necessary program tokens to set a location (or locations) in DM memory, and these DM

setting program tokens are automatically included in the assembler generated object file when a source file is assembled.

The DMSETLT statement reserves a LT memory location to initialize the DM during the program download to uPD7281. The address of the LT memory location used for this purpose is specified in the operand field.

2) When this statement is omitted in the source module, the last LT memory location specified by the LOCATE statement is reserved for DM memory setting purpose.

3) A valid expression to be coded in the operand field is restricted to those having NUMBER type attribute and the value within 0 and 127.

Example:

```
     :
     :
DMSETLT 127;
     :
     :
```

4.4.4 DMSETFT

(1) Function

Specifies the address of a FT memory location to be used for setting data into DM.

(2) Coding format

Statement field	Symbol field	Operand field
DMSETFT		expression ;

1) Since there is no specialized token format to set a DM location(s), a program must be written to initialize any DM memory locations used as constants, variables and tables before the actual program execution. Fortunately, however, the uPD7281 assembler is capable of generating the necessary program tokens to set a location (or locations) in DM memory, and these DM setting program tokens are automaically included in the assembler generated object file when a source file is assembled.

The DMSETFT statement reserves a FT memory location to initialize the DM during the program download to uPD7281. The address of FT memory location used for this purpose is specified in the operand field.

2) When this statement is omitted in the source module, the last FT memory location specified by the LOCATE statement is reserved for DM memory setting purpose.

3) A valid expression to be coded in the operand field is restricted to those having NUMBER type attribute and the value within 0 and 63.

Examples:

```
     :
     :
DMSETFT 63;
     :
```

4.5 Execution statements

There are three types of Execution statements:

LINK
FUNCTION
DEFINE

4.5.1 LINK

(1) Function

Reserves a LT memory area, and generates object code(s) for the LINK statement.

(2) Coding format

Statement field	symbol field
LINK	[LT-symboll[,...] =

Operand field

```
FT-symbol(LT-symbol2 [,LT-symbol3])[ AT expression];
```

(3) Description

1) The LINK statement defines a LT-symboll to be a LINK TABLE symbol, and it reserves a LINK TABLE location for the symbol. The contents of the reserved LT location is defined by the parameters specified in the operand field. The LT-symboll is an optional parameter if the link(or arc) specified by the operand field needs to be deleted by the uPD7281 internally.

Also, there may be more than one LT-symboll in a LINK statement, since more than one link may be

output from a function. For the multiple LT-symboll's in a LINK statement, the assembler reserves as many consecutive LT memory locations as the number of LT-symboll's.

The absolute LT memory locations for the LT-symboll's can be specified using ' AT expression' in the LINK statement where the 'expression' is the address of the first LT-symboll location. The actual number of LT-symboll's that can be included in a LINK statement is dependent upon the associated FUNCTION statement. Table 4-3 lists the number of LT-symboll's which can be defined in a LINK statement depending on the associated FUNCTION statement used.

2) A FT-symbol specified in the operand field must be defined using a FUNCTION statement in the same source program module.

Both the LT-symbol2 and the LT-symbol3 are optional parameters. However, at least one of these parameters must be specified for a LINK statement, and also depending on the associated FUNCTION statement used, the both or only one of them may be required for a LINK statement.

The LT-symbol2 corresponds to a FTRC=0 bit token, and the LT-symbol3 corresponds to a FTRC=1 bit token in a flow graph.

3) The uPD7281 user's manual lists each ImPP instruction with illustrative examples using LINK statements and FUNCTION statements along with example flow graphs. Please refer to the uPD7281 user's manual for further information.

TABLE 4-3

	Instruction used in the associated FUNCTION statement		The number of LT-symboll's to be defined
Out Inst.	Regardless of whether it is a single OUT instruction or a combined instruction (OUT instruction+AG&FC instruction).	With OUT1	1
		With OUT2	2
PU Instructions	Single PU instruction	When BRC=0 with X,Y specified	1
		When BRC=1 with X,Y specified	2
		With XX, XY specified	2
	PU instruction + AG&FC instruction*	---------	(Single PU instruction + AG&FC instruction - 1)

(Table 4-3, continued)

		Instruction used in the associated FUNCTION statement	The number of LT-symboll's to be defined
GE Instructions	Single COPYBK	---	2
		With COPYBK + CNTGE	3
	COPYBK+AG&FC instruction	With COPYBK + (AG&FC instruction other than CNTGE)	(Single COPYBK +AG&FC instruction-1)
	Single COPYM, SETCTL	---	(Number of first parameters in the parameter list.)
	COPYM, SETCTL+AG&FC	---	(Single COPYM, SETCTL+AG&FC instruction-1)
AG & FC Instructions	RDCYCS RDCYCL **	With FTRC=0	1
		With FTRC=1	2
		With FTRC=0,1	2
	WRCYCS WRCYCL **	With FTRC=0	0
		With FTRC=1	1
		With FTRC=0,1	1
	RDWR RDIDX	---	1
	PICKUP COUNT **	With FTRC=0	2
		With FTRC=1	0
		With FTRC=0,1	2
	CONVO	---	3
	CNTGE	---	3

(Table 4-3 continued)

	Instruction used in the associated FUNCTION statement		The number of LT-symboll's to be defined
AG & FC Instructions	DIVCYC **	With FTRC=0	2
		With FTRC=1	0
		With FTRC=0,1	2
	DIV **	With FTRC=0	2
		With FTRC=1	0
		With FTRC=0,1	2
	DIST	---------	Number of first parameters
	SAVE	With FTRC=0	1
		With FTRC=1	0
		With FTRC=0,1	1
	CUT **	With FTRC=0	1
		With FTRC=1	0
		With FTRC=0,1	1
	QUEUE	---------	1

(*) When the PU instruction is ACC, the number of LT-symboll's which may be defined in the LINK statement is 1 or 2, regardless which AG&FC instruction is used along with an ACC instruction. If there are more than two LT-symboll's defined in a LINK statement and the associated FUNCTION staement is an ACC instruction, then it is treated as an error.

(**) FTRC=0 .... when LT-symbol2 is specified in the LINK statement.
FTRC=1 .... when LT-symbol3 is specified in the LINK statement.

Examples:

```
     :
     :
LINK     X1,X2,X3=FGNE(X46,STATTK);  <---------
FUNCTION  FGNE=COPYBK(1,32),CNTGE(4);
LINK     X4,X5,X6=FCOPY3(X1);  <-----------
FUNCTION FCOPY3=COPYM(3,0);
LINK    X7=FBR(X4);  <--------------
FUNCTION FBR=MUL,RDCYCS(CNTB,1);
MEMORY    CNTB=2000H;
    :
    :
LINK     A1,A2,A3=FUNC2(,B2) AT 10; <--
    :
    :
    :
```

- (LINK X1,X2,X3=...) Symbols X1,X2, and X3 are defined, and 3 LT memory locations are reserved.
- (LINK X4,X5,X6=...) Symbols X4,X5, and X6 are defined, and 3 LT memory locations are reserved.
- (LINK X7=...) Symbol X7 is defined, and 1 LT memory location is reserved.
- (LINK A1,A2,A3=...) 3 symbols are defined, and the corresponding areas of LT memory starting from address 10 are reserved.

4.5.2 FUNCTION

(1) Function

Reserves a FT memory location and generates object code for the FT memory location reserved.

(2) Coding format

```
Statement field  Symbol field
---------------  ------------
   FUNCTION      symbol name  =

Operand Field
-----------------------------------------------------------
[ (OUT instruction (parameters),) ]
[ {PU  instruction (parameters),} ]  [AG&FC instruction(parameters),]
[ (GE  instruction (parameters),) ]

Operand Field (cont.)
------------------------
[FTT]  [ AT expression];
```

(3) Description

1) The FUNCTION statement defines a symbol specified in the symbol field as a FT symbol, and reserves a FT memory location for the symbol. In the case where ' AT expression' is specified in the operand field, a FT memory address location equal to the value specified in the 'expression' parameter is reserved.

2) An object code for the instruction(s) specified in the operand field is generated by the assembler. And the object code is set into the FT memory location reserved for the symbol during the program download.

3) The operand field contains the instruction(s) for the reserved symbol. There are two insruction fields in the operand field. The first instruction field is set aside for a PU, OUT or GE type instruction, and the second instruction field is only for the AG&FC type instruction. Each instruction may be followed by a parameter field enclosed in a pair of parentheses.

For a FUNCTION statement, both instruction fields or only one instruction field may be specified. However, at least one instruction field must be specified in the operand field. For a detailed explanation on how to code each instruction, please refer to the Appendix 4 of the assembler part of this manual.

The format of object code for a FUNCTION statement is shown in Fig. 4-1.

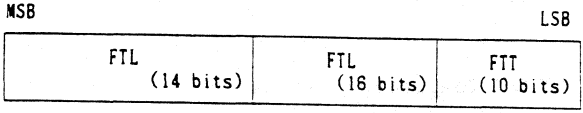


Fig. 4-1 FT object code format

Explanations on FTL, FTR, and FTT fields shown in Fig. 4-1 are given below.

a) FTL (Function Table Left) field

The object code for this field is generated when the instruction specified in the first instruction field is an OUT, a PU, or a GE instruction. The FTL field format is shown in Fig. 4-2.

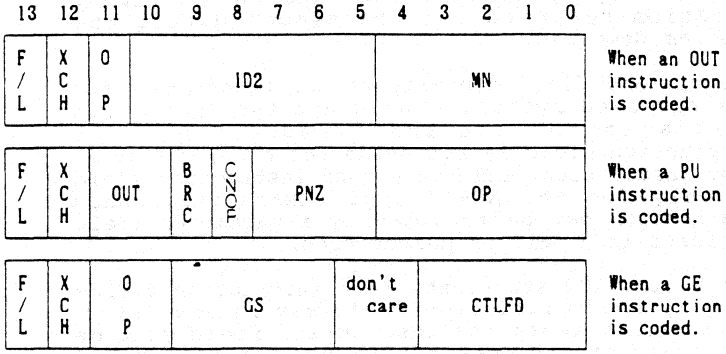


Fig. 4-2 Structure of FTL field

F/L ; If F/L=1 then both FTL and FTR fields of FT are to be executed at the same time.
If F/L=0, then either FTL or FTR field of FT is executed.

XCH ; If XCH=1 then 2 pieces of data are exchanged.
If XCH=0 then the 2 pieces of data are not exchanged.

b) FTR(FT Right), FTT(FT Temporary) fields

These fields are generated when an AG&FC instruction is specified along with either an OUT, a PU or a GE instruction in the operand field for a FUNCTION statement. Formats for these fields are shown in Table 4-4. The field formats vary with each of the AG&FC instructions. For detailed explanations of the fields

and the syntax, please refer to the uPD7281 User's Manual.

Table 4-4 FTR, FTT field formats

Instruction name (AG&FC instruction)	FTR (FT RIGHT FIELD) 15 ~ 12	11 ~ 8	7 ~ 4	3 ~ 0	FTT(FT TEMPORARY FIELD) 9	8	7	6	5	4	3	2	1	0
QUEUE	0 0 1 1	DM BASE ($x2^1$)		QUEUE SIZE	A/B	W/R	READ COUNTER						WRITE COUNTER	
RDCYCS	0 0 0 0	DM BASE ($x2^1$)		BUFFER SIZE									READ COUNTER	
WRCYCS	0 0 0 1	DM BASE ($x2^1$)		BUFFER SIZE									WRITE COUNTER	
RDCYCL	1 0 0 0	DM BASE ($x2^5$)		BUFFER SIZE			READ COUNTER							
WRCYCL	1 0 0 1	DM BASE ($x2^5$)		BUFFER SIZE			WRITE COUNTER							
RDWR	0 1 0 0	DM BASE ($x2^1$)					ADDRESS REGISTER							
RDIDX	0 1 0 1	DM BASE ($x2^1$)					ADDRESS REGISTER							

Table 4-4 FTR, FTT field formats (continued)

Instruction name (AG&FC instruction)	FTR (FT RIGHT FIELD) 15 ~ 12	11 ~ 8	7 ~ 4	3 ~ 0	FTT(FT TEMPORARY FIELD) 9	8	7	6	5	4	3	2	1	0
PICKUP	1 1 0 0		COUNT SIZE			COUNTER								
COUNT	1 1 0 1		COUNT SIZE			COUNTER								
CONVO	1 1 1 1		COUNT SIZE			COUNTER								
CNTGE	1 1 1 0		COUNT SIZE		W/D	COUNTER								
DIVCYC	1 0 1 0		COUNT SIZE	DIVIDE SIZE		COUNTER					COUNTER			
DIV	1 0 1 1		COUNT SIZE		S/F	COUNTER								
DIST	0 0 1 0		△ID SIZE			△ID								
SAVE	0 1 1 0					ID STACK REGISTER								
CUT	0 1 1 1		COUNT SIZE		S/F	COUNTER								

NOTE; During the actual execution of a program, the values of QUEUE SIZE, BUFFER SIZE, COUNT SIZE and △ID SIZE are stored as (SIZE-1).

```
Examples:
   :
   :
FUNCTION FT1=RDCYCS(ZERO,1); <---------------- Defines symbol
FUNCTION FT2=SHR,RDCYCS(ONE,1); <-┐             'FT1' and
FUNCTION FT3=OR,RDCYCS(LSB3,1); <-┤             reserves a FT
FUNCTION FT4=COPYBK(1,0);         └---------┐   memory location
FUNCTIONFT5=QUEUE(QUE14,1);                 |   for the symbol.
FUNCTION FT6=MUL(AA,BB),QUEUE(QUE1,16) AT 10;<  The 'RDCYCS
   :                                        |   (ZERO,1)' is
   :                                    ----┘   an AG&FC
                                               instruction.
                                    └-- Examples of combined
                                        instructions.
```

4.5.3 DEFINE

(1) Function

Declares a LT symbol(s) and reserves a LT memory location or a LT memory area for the symbol(s). These symbols are to be referenced by LINK statement(s) in the same module within a source program.

(2) Coding format

Statement field	Symbol field	Operand field
DEFINE		symbol name [AT expression][,...];

(3) Description

1) The DEFINE statement defines the symbol names specified in the operand field as LT symbols, reserves corresponding LT memory areas, and assigns certain memory addresses to the reserved areas.

2) When ' AT expression' is specified after 'symbol name', the address pointed to by the value of the expression is assigned to the reserved LT memory area. When more than one symbol name is specified and ' AT expression' is used, the address pointed by the value of the expression is assigned to the bottom (lower-limit) of the reserved memory area.

The value of the expression must be within 0 and 127, and it must have a NUMBER type attribute.

3) Symbols defined by this statement can only be referenced by 'LT-symbol2'* or 'LT-symbol3'* in LINK statements within the same module. (Reference cannot be made by statements other than a LINK statement.)

Examples:

```
     :
     :
DEFINE TKN1 AT 20H,TKN2; <---------- Defines symbols 'TKN1'
LINK  TKN3=AFUNC(TKN1,TKN2); <-┐      and 'TKN2' and reserves
     :                         │      corresponding LT memory
     :                         │      area. The first address
     :                         │      of the area is 20H.
                               │
                               └---- Refers to the above-
                                     defined symbols with a
                                     LINK statement.
```

(*) Refer to coding format in Section 4.5.1.

4.6 Data definition statements

4.6.1 DATA

(1) Function

Generates input data tokens (32 bits) for the uPD7281.

(2) Coding format

Statement field	Symbol field	Operand field
DATA		Type [(parameter list)][,...];

(3) Description

1) The DATA statement generates a 32-bit input data token (high- order 16 bits and low-order 16 bits) for the uPD7281 processor.

This statement can only be specified in the Data Section within a source program.

After downloading the assembler generated object code to a uPD7281, the 32-bit input data tokens are sent to the uPD7281 initiating the program execution.

2) Depending on the 'Type' of token specified in the operand field in a DATA statement, different parameters in the parameter list field are required. Table 4-5 lists different types of input data token formats. In addition, the actual field structures for these input tokens are shown in Table 4-6.

Table 4-5 Input Data Token Format

TYPE	Parameter list
SETLT	(Module number, LT address, LT WRITE DATA)
SETFTL	(Module number, FT address, FT LEFT FIELD WRITE DATA)
SETFTR	(Module number, FT address, FT RIGHT FEILD WRITE DATA)
SETFTT	(Module number, FT address, FT TEMPORARY FIELD WRITE DATA)

Table 4-5 (continued)

TYPE	Parameter list
RDLT	(Module number, LT address)
RDFTL RDFTR RDFTT	(Module number, FT address)
CRESET	(Module number)
SETMD	(Module number, constant*, Refresh Register Set Data, Input Inhibit Register Set Data)
SETBRK	(Module number, ID, constant**, count)
DUMP	(Module number, dump mode***)
CBRK	----------
VAN	----------
PASS	(Module number, ID, DATA)
EXEC	(Module number, ID, DATA)
DMS	(Module number, ID, first address to DM, data list)****
DMR	(Module number, ID, first address to DM, number of reads)****

(*) Bit configuration of the value specified in this constant is shown below.

Bit configuration		Explanation
D1	D0	
0	0	Input not inhibited according to the DQ size.
0	1	Input inhibited when DQ>24.
1	0	Input limited by Timer when DQ>24.
1	1	Illegal setting.

(**) 0 or 1 can be specified for the constant.
0 ; Break occurs when specified number of tokens access the LT location pointed to by the ID. The number of tokens accessed before a break occurs is specified by 'count'.
1 ; Break occurs when the number of cycles indicated in 'count' are executed after the first data arrives at LT location pointed by ID.

(***) The values that can be specified in 'dump mode' are shown below.

Specified value	Dump items specification
000	DQ, GQ size.
001	LT Input Latch content (U, ID, CTLF)
010	LT Input Latch content (DATA)
011	QUEUE Input Latch content (U,ID,SEL, Cb, Sb, Ca, Sa)
100	QUEUE Input Latch content (FTL, low-order 12 bits)
101	QUEUE Input Latch content (DATAa)
110	QUEUE Input Latch content (DATAb)
111	ID held by FT token when it was refering to LT.

(****) When DMS or DMR command is specified in the operand field of the DATA statement, a program which sets initial values into DM memory must already be set into certain LT, FT memory areas. Note that these memory areas are reserved by the DMSETLT and DMSETFT statements in the EXECUTION section of a program module.

When LT and FT memory areas are not specified by DMSETLT and DMSETFT, the default values are assumed.

Table 4-6 Field structure in command name

TYPE	HIGH					LOW									
	Module number	Z	I	ID	C	T	L	F							
SETLT	M	0	LT address	1	1	0	0	LT WRITE DATA							
SETFTL	M	0	FT address	1	1	1	0	FT LEFT FIELD WRITE DATA							
SETFTR	M	0	FT address	1	1	0	1	FT RIGHT FIELD WRITE DATA							
SETFTT	M	0	FT address	1	1	1	1	FT TEMPORARY FIELD WRITE DATA							
RDLT	M	0	LT address	1	0	0	0	*							
RDFTL	M	0	FT address	1	0	1	0	*							
RDFTR	M	0	FT address	1	0	0	1	*							
RDFTT	M	0	FT address	1	0	1	1	*							
CRESET	M	0	*******	0	1	0	0	*							
SETMD	M	0	*******	0	1	0	1	0	0	D_1	D_0	NOTE 1	NOTE 2		
SETBRK	M	0	ID	0	1	1	0	MB				Count Data			
DUMP	M	0	****|dump mode	0	1	1	1	*							
CBRK	0000	0	*******	0	1	0	0	*							
VAN	1111	0	*******	*	*	*	*	*							
PASS	M	0	*******	*	*	*	*	*							

NOTE1 ; Refresh Register Set Data
NOTE2 ; Input Inhibit Register Set Data
* ; don't care bit
MB ; Refer to note (**) in Table 4.5

Table 4-6 Field structure in command name (continued)

TOKEN TYPE	HIGH										LOW
	Module number		Z		ID		CTLF				
EXEC	M		0		ID		0 0 C S				Data
DMS	M		0		ID		0 0 C S				Data
DMR	M		0		ID		0 0 C S				Bias

3) For a token specified using DATA statement, there are certain restrictions on the parameter listed in the operand field. These restrictions are shown below according to each parameter.

a. Module number

A module name is specified in 'module number'.

b. LT address or ID

A constant, or a symbol with a NUMBER type attribute, or LT attribute is specified for the 'LT address' or ID.

c. FT address

A constant, or a symbol with a NUMBER type attribute, or FT attribute is specified for the 'FT address'.

d. First address to DM

A constant, or a symbol with a NUMBER type attribute, or DM attribute is specified for the 'first address to DM'.

e. Other parameters

A constant or a symbol with a NUMBER type attribute is specified for the parameters other than those listed above.

4.7 Assembly end statement

The 'END' statement is used to terminate a source program file. Since this statement is used to terminate the assembly, it must be the last statement in a source program.

4.7.1 END

(1) Function

Terminates assembly.

(2) Coding format

Statement field	Symbol field	Operand field
END		;

(3) Description

1) The END statement terminates the source module. Nothing should be specified in the symbol field or the operand field of this statement.

2) This statement can be specified in the source module only once.

Example:

```
      .
      .
      .
MODULE AFIN=8;
      .
      .
      .
START;
DATA  EXEC(EXAM,ARC1,2);
DATA  EXEC(EXAM,ARC2,3);
END;
```

CHAPTER 5 INPUT/OUTPUT FILES

This chapter explains the organization of input/output files of the assembler and other listing files.

5.1 Input/Output files handled by the Assembler

The relationships between the input/output files handled by the assembler are shown in Fig. 5-1.

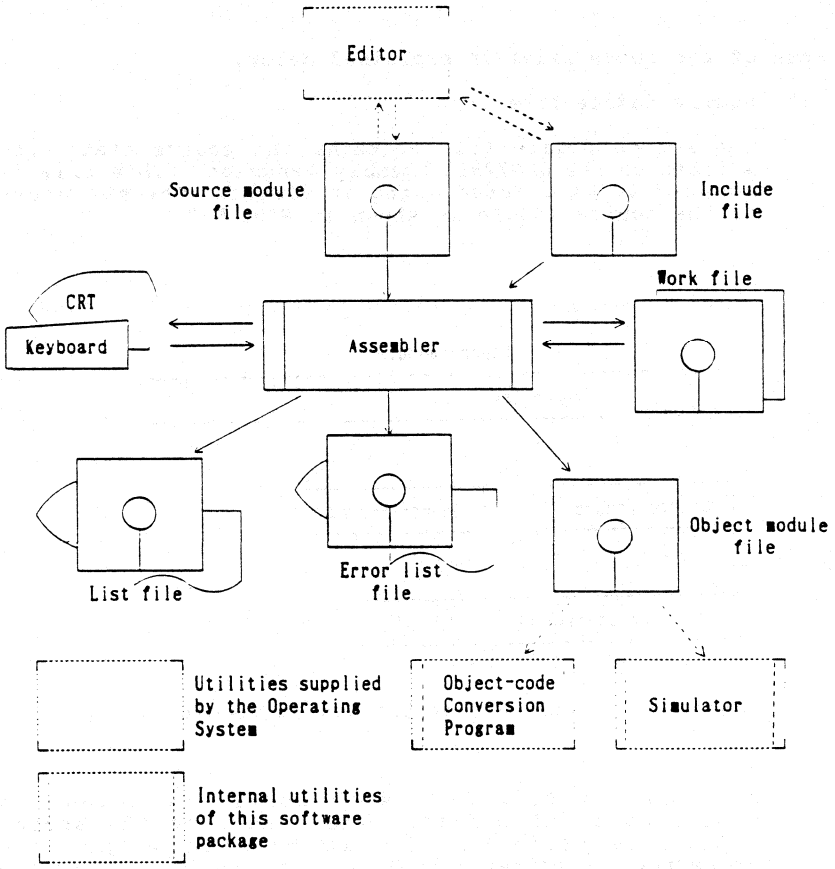


Fig. 5-1 Relationships between input/output files

5.1.1 File Types

There are six different types of files handled by the assembler.

1) Source module file
2) Object module file
3) List file
4) Error list file
5) Include file
6) Work file

Each of the above files is explained below.

(1) Source module file

The source module file contains the source statements written in the uPD7281 assembly language. This file is assumed to be created using an editor. The structure of the source module is shown in Fig. 5-2.

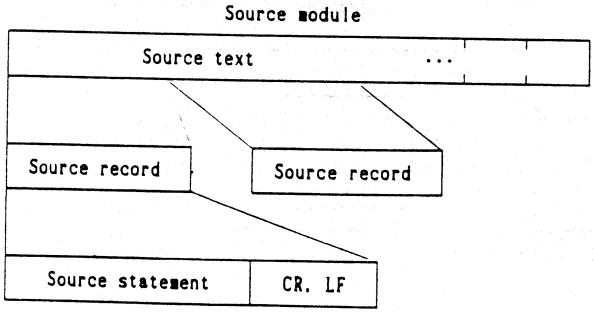


Fig. 5-2 Structure of the source module

As shown in Fig. 5-2, each source statement is separated by CR and LF. The length of the source record is variable, but it must be within 116 characters. Chapter 3 of Part 2 of this manual (on the assmbler) contains detailed explanations about the source statement formats.

(2) Object module file

The object module file contains the machine object code for the uPD7281. The object file is generated by the assembler using a source program file as an input module. The object file is obtained by specifying the 'OBJECT' control* at assembly starting time.

An object module file in HEX-format or ASCII-data-format can be obtained by inputting an object module file into the Object-code Conversion Program. Additionally, the object module file may be input to the uPD7281 software simulator for program simulation purposes.

(3) List file

A list file may include a source listing, object listing, cross reference listing and error messages resulting from the assembly. This file can be obtained by specifying 'PRINT' control* at the time of assembly.

(4) Error list file

If there exist any syntax errors in the source module during the time of assembly, the lines containing these assembly errors and the respective error messages are written into the error list file. This file can be obtained by specifying 'ERRORLIST' control at the time assembly.

(5) Include file

The include file may be specified if there exists source statements in other files to be included during the assmbly of current source file's statements. The structure of the include file must conform to any other source mudule. The include file may be included by specifying 'INCLUDE' control* in the source module file.

(6) Work file

This is a file in which the assembler temporarily stores its intermediate results during the assembly. The assembler itself creates the work files and deletes them after the assembly is done. File names of work files are shown below.

```
------------------
  d:AS7281.$$n
------------------
```

```
d:      ; disk drive name specified by 'WORKFILES'
          control*.
n       ; a workfile sequence number.
```

(*) for more information on controls, refer to Section 6.2.

5.1.2 Input/Output classification and media

Input/output classification and usable media are shown in table 5-1 for each differnt type of file.

Table 5-1 Input/Output classification and media

File	Input/Output	Medium
Source module file	Input	FD/HD
Object module file	Output	FD/HD
List file	Output	FD/HD/CON/LST
Error list file	Output	FD/HD/CON/LST
Include file	Input	FD/HD
Work file	Input/Output	FD/HD

```
FD  ; Floppy disk
HD  ; Hard disk
CON ; Console
LST ; Printer
```

5.2 Output lists

The following five types of listing files may be output by the assembler.

1) Assembly list
2) Error list
3) Mnemonic-format object code list
4) HEX-format object code list
5) Symbol cross-reference list

The lists 1, 3, 4, and 5 are all output to the file specified by 'PRINT' control, while list 2 is output to the file specified by 'ERRORLIST' control.

These five different types of listing files are explained below.

5.2.1 Assembly list

The result of assembling a source module file is output to this list. When there are any errors in the source module file, the error messages are also output to the assembly list.

(1) Format

```
**  UPD7281 ASSEMBLER, Vx.x ** PROGRAM NAME program name
                                          date PAGE page
```

The program name is printed on the first page, and the title is printed on the subsequent pages (from the second page).

```
SOURCE FILE   ;  source_module_name

OBJECT FILE   ;  object_module_name

LIST FILE     ;  list_file_name

ERROR FILE    ;  error_file_name

COMMAND       ;  command at the time of assembler invocation
                 (When no control is specified, only the
                 source module file name is printed.)
```

```
LNO  STNO   SOURCE

XXXX  XXXX  X XX-----------------------------------------XX
 :     :    :                    :
 :     :    :                    :
XXXX  XXXX  X XX-----------------------------------------XX
```

line number — statement number — source statement

------ When the input file of the source statement is the one specified in the 'INCLUDE' control, '=' is printed.

---------- Printed only on the first line of each statement.

5.2.2 Error list

During the source program assembly, if the assembler detects any error in a source statement, the source statement with the error and corresponding error message(s) is output to the error list file (or device).

(1) Format

```
**  UPD7281 ASSEMBLER, Vx.x ** PROGRAM NAME program name
                                          date PAGE page
```

The program name is printed on the first page, and the title is printed on the subsequent pages (from the second on).

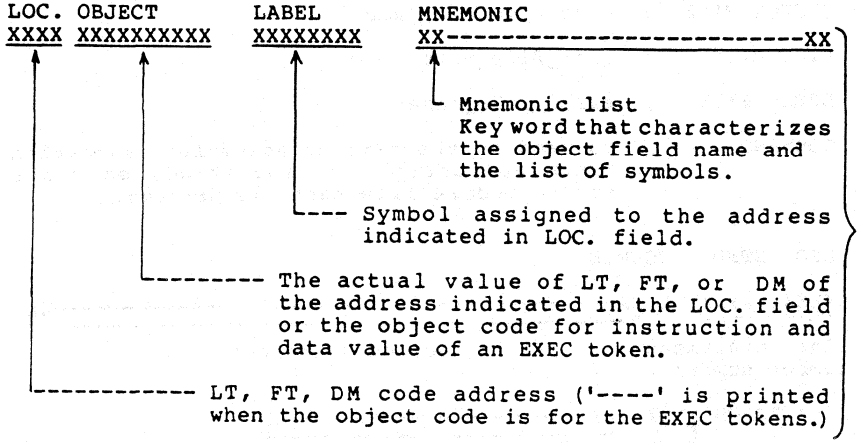
```
SOURCE FILE   ;  source_module_name

OBJECT FILE   ;  object_module_name

LIST FILE     ;  list_file_name

ERROR FILE    ;  error_file_name

COMMAND       ;  command at the time of assembler invocation
                 (When no control is specified, only the
                 source module file name is printed.)

LNO  STNO   SOURCE

XXXX  XXXX  X XX----------------------------------------XX
line  statement              source statement
number number

*** ERROR XXXX, XX----------------------XX
                          error message
             error number
          {W}
          {F}

XXXX  XXXX  X XX----------------------------------------XX

*** ERROR XXXX, XX----------------------XX

      :
      :
```

5.2.3 Mnemonic-format object code list

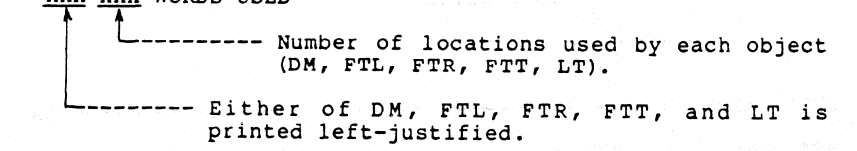
The object code resulting from the source program file assembly is listed in this file in mnemonic format.

(1) Format

```
**  UPD7281 ASSEMBLER, Vx.x ** PROGRAM NAME program name
                                        date PAGE page

OBJECT CODE LIST (MNEMONIC)     MODULE NAME XXXXXXXX NO. XX
                                            1)          2)

  ** XXXX SECTION **
```




```
** XXX XXX WORDS USED **
```



1) Module name is printed in this field. However, when the object code is for the Declaration Section or Data Section, nothing is printed in this field.

2) Module number is printed in this field. However, when the object code is for the Declaration Section or Data Section, nothing is printed.

Print formats for

DM Section
LT Section
FT Section
DATA Section

are shown below in paragraphs from (2) to (5).

(2) DM Section print format

```
LOC. OBJECT       LABEL       MNEMONIC

       1)         2)          3)       4)
XXXX XXXX         XXXXXXXX:   XXX      XXXXX
```

1) The object code is printed in this field. However, when the mnemonic is ORG, DS or DC, nothing is printed here.

2) Label (symbol) is printed in this field. However, when the mnemonic is ORG or DC, nothing is printed here.

3) A mnemonic (ORG, DS, or DC) is printed here.

4) The value to be set in DM is printed in this field.
   ORG ..... address value
   DS ..... area size
   DC ..... constant

(3) LT Section print format

```
LOC. OBJECT       LABEL       MNEMONIC

                              1)       2)          3)          4)
XXXX XXXX         XXXXXXXX:   XXX      XXXXXXXX,   XXXXXXXX,  X
```

1) The type of instruction (OUT, PU, GE or AG&FC) is printed in this field.

2) The destination ID is printed in this field.

3) The FT symbol name to be referenced is printed in this field.

4) FTRC of LT template is printed in this field.

(4) FT Section print format

```
LOC. OBJECT         LABEL        MNEMONIC

                                              3)                      4)
    1)                          2)      /XX---------------XX\
XXXX XXXX          XXXXXXXX:  XXXXXXXX {                     } [,FULL]
                                        \[XX,]XXXXH[,XXXXH]/
                                              5)    6)     7)

XXXX    XXXXXXXX  XXXXXXXX:  XXXXXXXX   [XXXXXX][,XXXXH]
8)      9)                    10)          11)        12)
```

1) The object code for the FTL field is printed in this field.

2) When the instruction is an OUT, a GE or a PU instruction, the instruction mnemonic is printed in this field.

3) When the instruction is one of the GE instructions, the number for the GS(Generation Size) is printed in this field. When the instruction is one of PU instructions, output specification, XCH, PNZ, (EQ, NE, ...) and CNOP are printed in this field. (The default values are not printed, however.)

4) 'FULL' is printed here when the F/L flag in FTL field is '1'.

5) The output destination module number is printed in this field when the instruction is an OUT instruction.

6) The destination ID is printed in this field.

7) The second destination ID is printed here, in the case of 2 outputs.

8) The address of FTR/FTT field is printed in this field (blank in the case of packed format.)

9) The object codes for the FTR and FTT fields are printed in this field.

10) The instruction mnemonic is printed here when the instruction is one of the AG&FC instructions.

11) The DM address is printed in this field.

12) The SIZE is printed in this field.

(5) Data Section print format

```
LOC.  OBJECT        LABEL        MNEMONIC

      1)                         2)         3)          4)
----  XXXXXXXX                   XXXX       XXXXXXXX,XXXXXX
```

1) Normally, the object code for an EXEC token is printed here. However, if the type of the input token is RDLT, RDFTL, RDFTR, CRESET, DUMP, CBRK, VAN, PASS, or DMR, the lower 4 hex digits are not printed.

2) The input token type is printed in this field.

3) When the input token type is DUMP, the dump mode is printed in this field; when the input token type is CRESET, SETMD, CBRK, VAN, or PASS, nothing is printed here. For those token types other than the types listed above, the destination ID is printed in this field.

4) When the input token type is RDLT, RDFTL, RDFTR, CRESET, DUMP, CBRK, VAN, PASS, or DMR, nothing is printed here, otherwise, the data field of the token is printed in this field.

5.2.4 HEX-format object code list

The object code resulting from an assembly may be output to the list file in HEX-format. This is an optional feature.

(1) Format

```
**  UPD7281 ASSEMBLER, Vx.x **  PROGRAM NAME program name
                                              date PAGE page

OBJECT CODE LIST (HEXADECIMAL)   MODULE NAME  XXXXXXXX NO. XX
                                               1)        2)

  ** XXXX SECTION **
       └------------ LT, FT, DM or DATA is printed left-
                     justified.

LOC. OBJECT CODE
XXXX XXXX:XXXX XXXX:XXXX XXXX:XXXX
                :
                :
                :
                :
XXXX XXXX:XXXX XXXX:XXXX XXXX:XXXX
  |     └------------ Up to 3 locations' object-code may be
  |                   printed in a line. The object-code for
  |                   each location conforms to the I/O token
  |                   formats, thus containing 32 binary bits.
  |                   The 32 bit object-code for a token is
  |                   dislayed in 8 hex digits, with the upper
  |                   4 digits and the lower 4 digits
  |                   separated by a colon.
  |
  |                  ( LT Section .....    1 token
  |                  | FT Section ..... 1~3 tokens
  |                  | DM Section ..... 1~2 tokens
  |                  ( DATA Section ...    1 token
  |
  └------------------ The address location of LT, FT or DM,
                      to be set.
```

1) The module name is printed in this field. However, nothing is printed here when the object code belongs to the Declaration Section or the Data Section.

2) The module number is printed in this field. However, nothing is printed here when the object code belongs to the Declaration Section or the Data Section.

5.2.5 Symbol cross-reference list

If the cross reference option is specified during the time of assmbler invocation, the cross reference list of symbols is output to the listing file. The cross reference list shows where each symbol is defined and referenced by other statements in the same source program file. It also lists the type of attribute assigned to the symbol along with its value.

(1) Format

```
**  UPD7281 ASSEMBLER, Vx.x ** PROGRAM NAME  program name
                                              date PAGE page

SYMBOL XREF LISTING              MODULE NAME   XXXXXXXX NO. XX
                                                   1)       2)
-------------------

IDENTIFIER  ATTR.    VALUE    XREF

XXXXXXXX    XXXXXXX  XXXXXXX  XXXX  XXXX  XXXX  ....  XXXX
                       :
                       :
                       :
                       :
XXXXXXXX    XXXXXXX  XXXXXXX  XXXX  XXXX  XXXX  ....  XXXX
   3)          4)       5)                  6)

ASSEMBLY COMPLETE, XXXX ERROR(S), XXXX WARNING(S) FOUND
```

1) The module name is printed in this field.

2) The module number is printed in this field. However, nothing is printed here in the case of the Declaration Section or the Data Section.

3) The symbol name is printed in this field.

4) The type of attribute for the symbol is printed in this field. There are five different types of attributes that can be printed in the field.

   EQUATE
   LITERAL
   LT SYM
   FT SYM
   DM SYM

5) The value of the symbol is printed in this field.

6) The line number where a statement defined or referred to a symbol is printed. In the case of a definition, '#' is printed after the line number.

CHAPTER 6 THE ASSEMBLER OPERATION PROCEDURES

6.1 Sequence of operations

The sequence of the operations for using the assembler is shown in Fig. 6-1.

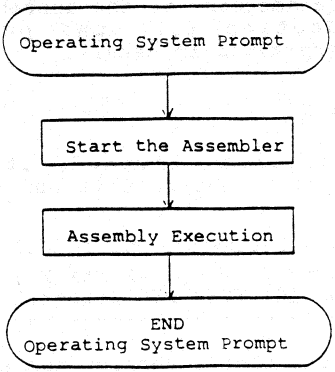


Fig. 6-1 Sequence of operations

Each operation in Fig. 6-1 is explained below.

(1) Wait for the operating system's prompt sign

Only if the Operating System is ready to accept a command by displaying the prompt sign 'A>' on the console, then the uPD7281 assembler may be invoked.

(2) Starting the assembler

The assembler may be invoked by entering the following command after the operating system's prompt sign. The syntax for invoking the assembler is

```
A> AS7281  d:source_module_file_name  [control list](CR)
```

d: ; The disk drive number containing the source module file. (If d: is omitted, then it is assumed that the default drive has the source file.)

AS7281 ; The command file name of the assembler. All of the assembler and its overlay files must exist in the default drive.

control list ; This field contains optional control parameters for the assembler ouputs. In Section 6.2, the parameters are explained in detail.

(3) Assembly execution

Once the assembler is invoked, it displays the following messages on the console and then begins the execution.

```
I UPD7281 ASSEMBLER Vx.y [XX XXX XX]     I
I     Copyright (C) YYYY NEC Corporation I
```

Vx.y ; Assembler's version number.

XX XXX XX ; The date of the Vx.y version assembler is created.

YYYY ; The year this version of assembler is created.

(4) Assembly termination

When the assembly is finished or terminated, the assembler displays the following message on the console and returns the control to operating system.

1) In the case of normal termination

```
ASSEMBLY COMPLETE,nnnnERROR(S),mmmmWARNING(S) FOUND
```

nnnn ; number of errors
mmmm ; number of warnings

2) In the case of abnormal termination

```
--------------------
  PROGRAM ABORTED
--------------------
```

6.2 Controls

This section contains instructions on how to use the assembler with its optional controls. The available control parameters for the uPD7281 assembler can be classified into the following two types.

1) Basic controls
2) General controls

6.2.1 Basic controls

The basic controls specify the necessary information prior to the assembler's execution. The parameters are specified at the time of assembler invocation.

(1) Coding format

The coding format for basic controls is shown below.

```
-------------------------------------------------------
A> AS7281 B:source module file name
                        [ basic control  ....[&]](CR)
-------------------------------------------------------
```

Up to 80 characters (including CR, LF) can be entered in a line for the assembler invocation command. If more than 80 characters are needed to specify the basic control parameters, it can be continued on the next line (continuation line) by writing '&(CR)' at the end of the line to be continued. As '&(CR)' is entered, '**' is displayed on the next line, and then subsequent control parameters may be entered.

Example:

```
A> AS7281 B:EXAMPL.SRC PRINT(LST:)& (CR)
** SYMBOLS DATE(7/22/85) (CR)
```

The assembler ignores any strings of characters between '&' and '(CR)', which are used for specifying continuing lines. Therefore, a comment may be written between '&' and '(CR)'.

Example:

```
A> AS7281 C:AFIN.SRC PRINT(LST:) & CHECK PROGRAM (CR)
                  It is regarded as a comment.
```

When specifying more than one basic control parameter in the command line, the order in which the control parameters are specified is arbitrary, except for the

specification of the source module file. It must follow the AS7281 command.

The following two assembler invocation command examples have the same meaning.

Examples:

```
1)  A> AS7281 B:TEST.SRC PRINT(LST:) SYMBOLS (CR)
2)  A> AS7281 B:TEST.SRC SYMBOLS PRINT(LST:) (CR)
```

(2) Types of basic controls

The assembler provides the following 4 types of basic control.

- Controls for specifying the Input/Output files
- Controls for specifying the output list items
- Controls for specifying the output list lines
- Controls for specifying the assembly date.

A classification list of basic controls is shown in Table 6-1. Included are the control names, possibility of omission, specification format and default values.

Table 6-1 List of basic controls

Classification		Control name	Abbreviated format	May be omitted?	Default value
Input/Output File Specifications	Source module file	Source module file name	----	NO	----
	Object module file	OBJECT [(file name)]/ NOOBJECT	OJ NOOJ	YES	OJ[(source module file name)]. File type is OBJ.
	List file	PRINT [(file name)]/ NOPRINT	PR NOPR	YES	PR[(source module file name)]. File type is LST.
	Work file	WORKFILES (drive name)	WF	YES	WF(drive name of source module file)
	Error list file	ERRORLIST[(file name)]/ NOERRORLIST	EL NOEL	YES	NOEL

Table 6-1 List of basic controls (continued)

Classification	Control name	Abbreviated format	Maybe omitted?	Default value
Object code list / HEX-format	CODE NOCODE	CD NOCD	YES	NOCD
Object code list / Mnemonic -format	MNEMONIC NOMNEMONIC	MN NOMN	YES	MN
Symbol list / Symbol information	SYMBOLS NOSYMBOLS	SB NOSB	YES	NOSB
Cross reference	XREF NOXREF	XR NOXR	YES	NOXR
Output list Specification / Number of columns in a line	PAGEWIDTH (number of columns)	PW	YES	PW(120). When console (CON:) is specified with ERRORLIST or PRINT control, PW(79) is assumed.
Output list line Specification / Number of lines in a page	PAGELENGTH (number of lines)	PL	YES	PL(66)
Processing date specification	DATE (character string)	DA	YES	DA (9 blanks)

The following describes each control parameter and explains its usage.

1) Controls for specifying the input/output files

There are five types of controls for specifying the input/output files.

a) for source module file specification
b) for object module file specification
c) for list file specification
d) for work file specification
e) for error list file specification

a) Source module file specification (cannot be omitted)

Format

```
------------------
  [d:]file name        where d is the drive number
------------------
```

Description

This is the source program file input to the uPD7281 assembler. It must be specified immediately following the AS7281 command. If the drive name d: is omitted, it is assumed that the input file is on the default disk drive. The medium for the source module file is restricted to disk files. (For file name formats, refer to Section 6.3, Rules on file names.)

Examples:

1) A> AS7281 B:AFFINE.SRC (CR)

2) B> AS7281 MATRIX.SRC (CR)

b) Object module file specification (can be omitted)

Format

If an object file output is desired,

```
----------------------   -------------------
OBJECT[(file name)]   or   OJ[(file name)]
----------------------   -------------------
```

If an object file output is not desired,

```
------------   --------
NOOBJECT    or    NOOJ
------------   --------
```

Description

Specifies whether or not the object module file is to be output after assembly, and if the output is desired, the control specifies an output file or an output device.

A disk file must be specified in 'file name'. When drive name (d:) is omitted, the default drive name is assumed. The output medium is restricted to a disk file.

When the object module file specification is omitted, the following specification is assumed

by default.

```
---------------------------
  OBJECT(primary name.LNK)
---------------------------
```

NOTE: The primary name is the same as that of the source module file.

Example:

```
A> AS7281 B:AFFINE.SRC OBJECT(C:AFIN.LNK) (CR)
```

; When it is entered, the source module file AFFINE.SRC on disk drive B: is assembled, and the object module file is output to a file on disk drive C: with the file name 'AFIN.LNK'.

c) List file specification (can be omitted)

Format

If a list file is to be output

```
----------------------    -------------------
  PRINT[(file name)]   or   PR[(file name)]
----------------------    -------------------
```

If no list file is to be output

```
-----------    --------
  NOPRINT   or   NOPR
-----------    --------
```

Description

Specifies whether or not a list file is to be output. If a list file is desired, the control also specifies the output file or device.

A disk file is to be specified in 'file name'. If a drive name is omitted, the same drive name specified for the source module file is assumed by default.

The output medium is restricted to a disk file, LST:, or CON:. When the list file is specified and if the file name is omitted, the following is assumed to be specified. (For file name formats, refer to Section 6.3.)

```
---------------------------
  PRINT(primary name.PRN)
---------------------------
```

NOTE: The primary name is the same as the source module file.

Examples:

1) A> AS7281 AFFINE.SRC PRINT(LST:) (CR)
   ; Outputs the list file to the printer after assembly.

2) A> AS7281 AFFINE.SRC PRINT(B:AFIN.PRN) (CR)
   ; Outputs the list file to the disk drive B: with file name AFIN.PRN.

d) Work file specification (can be omitted)

Format

WORKFILES(drive name)	or	WF(drive name)

Description

Specifies where the assembler should create the work files. The work files are used to temporarily store the intermediate results during assembly. The medium to be specified is restricted to disk files only. The work files are deleted after the assembly terminates.

When this specification is omitted, the following specification is assumed by default.

WORKFILES(drive name specified for source modulefile)

NOTE: As shown above, the default drive name is same as the drive name specified for the source module file.

Example:

A> AS7281 B:EXAMPL.SRC WORKFILES(D:) (CR)
   ; Creates work file on disk drive D:.

e) Error list file specification (can be omitted)

Format

In case the error list file is to be output

```
ERRORLIST[(file name)]   or   EL[(file name)]
```

In case the error list file is not to be output

```
NOERRORLIST    or    NOEL
```

Description

Specifies whether or not the error list file is to be output.

When the file name specified in 'PRINT' control and the file name specified in 'ERRORLIST' control are the same, the latter is ignored, and the 'PRINT' control becomes effective. The file name parameter for the ERRORLIST may be omitted.

When 'ERRORLIST' control is specified but the file name is omitted, the following specification is assumed by default.

```
ERRORLIST(CON:)
```

Furthermore, when this control is omitted, 'NOERRORLIST' is assumed by default.

Examples:

1) A> AS7281 B:EXAMPL.SRC ERRORLIST (CR)
   ; Displays the error list file on the console.

2) A> AS7281 B:EXAMPL.SRC ERRORLIST(C:EXAMPL.ERR)(CR)
   ; Outputs the error list file to the disk drive C: with the file name 'EXAMPL.ERR'.

2) Controls for specifying the output list

This control is used for specifying the type of list to be output to the list file. The relationships between the specification of this control and the output lists, please refer to Chapter 5, 'Input/Output files'.

There are four types of controls for specifying output list:

a) for object code list (HEX)
b) for object code list (mnemonic)
c) for symbol cross-reference list (symbol information)
d) for symbol cross-reference list (cross reference)

When 'PRINT' control is specified, the assembly list is output unconditionally.

a) Object code list specification (HEX) (can be omitted)

Format

If a HEX-format object code list is to be output

CODE	or	CD

If a HEX-format object code list is not to be output

NOCODE	or	NOCD

Description

Specifies whether or not the list of the object code converted into HEX-format is to be output after the assembly.

When 'NOPRINT' control is specified, the 'CODE' control is ignored (even if it is specified). When this control is omitted, 'NOCODE' is assumed.

Examples:

1) A> AS7281 B:EXAMPL.SRC PRINT(LST:) CODE (CR)
   ; The HEX-format object code list is output to the printer.

2) A> AS7281 B:EXAMPL.SRC NOCODE (CR)
   ; The HEX-format object code list is not output to the printer. (The same is true even when 'NOCODE' is omitted.)

b) Object code list specification (mnemonic) (can be omitted)

Format

If the mnemonic-format object code list is to be output

MNEMONIC	or	MN

If the mnemonic-format object code list is not to be output

NOMNEMONIC	or	NOMN

Description

Specifies whether or not the mnemonic-format object code list is to be output after the assembly.

When 'NOPRINT' control is specified, 'MNEMONIC' control is ignored even when it is specified.

When this control is omitted, the default specification is 'MNEMONIC'.

Examples:

1) A> AS7281 B:EXAMPL.SRC NOMNEMONIC (CR)
   ; The mnemonic-format object code list is not output.

2) A> AS7281 B:EXAMPL.SRC MNEMONIC (CR)
   ; The mnemonic-format object code list is output. (The same is true even when 'MNEMONIC' is omitted.)

c) Symbol list specification (can be omitted)

Format

If the symbol list is to be output

SYMBOLS	or	SB

If the symbol list is not to be output

NOSYMBOLS	or	NOSB

Description

Specifies whether or not the symbol reference list is to be output after the assembly. When this control is omitted, 'NOSYMBOLS' is assumed by default.

Examples:

1) A> AS7281 B:EXAMPL.SRC PRINT(LST:) SYMBOLS (CR)
   ; Symbol reference list is output to the printer.

2) A> AS7281 B:EXAMPL.SRC NOSYMBOLS (CR)
   ; Symbol reference list is not output to the printer. (The same is true even when 'NOSYMBOLS' is omitted.)

d) Symbol cross-reference list specification (can be omitted)

Format

If the symbol cross-reference list is to be output

XREF	or	XR

If the symbol cross-reference list is not to be output

NOXREF	or	NOXR

Description

Specifies whether or not the symbol cross-reference list is to be output after the assembly.

When 'XREF' is specified, the above mentioned 'SYMBOLS' is regarded as specified, even when it is not specified explicitly. When this control is omitted, 'NOXREF' is assumed by default.

Examples:

1) A> AS7281 B:EXAMPL.SRC (LST:) XREF (CR)
   ; Symbol cross-reference list (both cross reference and symbol information) is output to the printer.

2) A> AS7281 B:EXAMPL.SRC NOXREF (CR)
; Symbol cross-reference list is not output. (The same is true even when 'NOXREF' is omitted.)

3) Controls for specifying the number of lines in the list file or device.

This control parameter may be used to design the page format in the listing file by specifying the number of columns and the number of lines per page to be used.

The two specification items are a) the maximum number of columns and b) the maximum number of lines per page.

These specifications are effective for all list files.

a) Specification for the maximum number of columns (can be omitted)

Format

PAGEWIDTH(number of columns)	or	PW(number of columns)

Description

Specifies the maximum number of characters (number of columns) to be printed in one line when list file is output after the assembly.

The number to be specified in 'number of columns' is restricted to a decimal number between 72 and 132. When this control is omitted, the following specification is assumed by default.

```
PAGEWIDTH(120)
```

NOTE: When this control is omitted and the console (CON:) is specified as an output file with 'PRINT' and 'ERRORLIST' control parameters, the number of columns is assumed to be 79.

Example:

A> AS7281 B:EXAMPL.SRC PRINT(LST:) PAGEWIDTH(132)(CR)
; Sets the width of a printing line of all lists to 132 characters (columns).

b) Specification for the number of lines in a page (can be omitted)

Format

PAGELENGTH(number of lines)	or	PL(numberof lines)

Description

Specifies the maximum number of lines to be printed in one page of a list. The number to be specified in 'number of lines' is restricted to a decimal number from 20 to 127. When this control is omitted, the following specification is assumed by default.

```
PAGELENGTH(66)
```

Example:

```
A> AS7281 B:EXAMPL.SRC PRINT(LST:) PAGELENGTH(80)(CR)
      ; Sets the number of lines to be printed in one
        page to 80 characters.
```

4) Control for specifying the assembly date

A date may be specified at the time of assembler invocation so that it can be printed out as a part of header information on listing files. This is an optional parameter. The same date also will be logged on to the object file as the assembly date. If the DATE control is omitted, 9 blank spaces are placed in the DATE field in the list.

Format

DATE(date)	or	DA(date)

Example:

```
A> AS7281 B:EXAMPL.SRC PRINT(LST:) DATE(7/22/85)(CR)
      ; The date on the list file is set to 7/22/85.
```

6.2.2 General controls

General control parameters are specified within a source file, and they are executed during the actual assembly process.

(1) Coding format

The coding format for all general control commands is as follows.

```
-----------------------------------
  $[  ]control name[  operand] (CR)
-----------------------------------
  └first column
```

A '$' must always be placed in the first column of a general control. A general control must be specified within a line (continuation line is not allowed for the general controls). Furthermore, it is not possible to specify more than one control in the same line. An optional blank can be inserted between '$' and 'control name'.

Example:

```
          .
          .
     statement
          .
          .
$ INCLUDE (B:SMAC.INC)
          .
          .
     statement
          .
          .
```

(2) Types of general control

The assembler provides two types of general control.

Control for including source module files
Control for outputting lists

The Table 6-2 summarizes the general control with a list of control names, abbreviations and coding formats.

Table 6-2 List of general controls

Classification	Control name	Abbreviation	Coding format
For source module file	INCLUDE	IC	$ INCLUDE(file name)
	TITLE	TT	$ TITLE (character string)
For control of output lists	EJECT	EJ	$ EJECT
	LIST	LI	$ LIST
	NOLIST	NOLI	$ NOLIST

Each control is explained below.

1) Control for including a source module file

The 'INCLUDE' control is used to include a source module file that already exist outside of the current source program file during the time of assembly.

Format

```
$[  ]INCLUDE[  ](d:source module file name)
```

Description

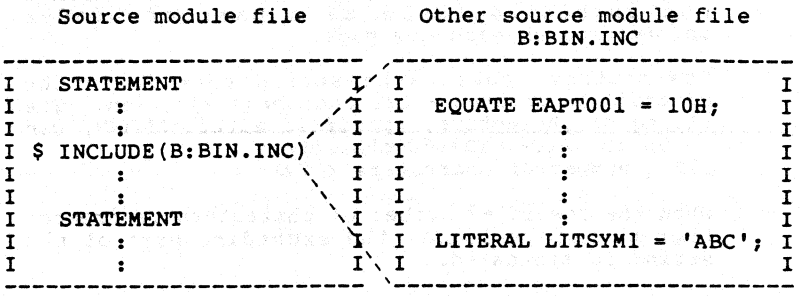
This control is used for including a group of source module statements from other source module files during the time of assembly.

When drive name (in front of the source module file name) is omitted, the current default drive number is assumed. (For the file name format, refer to Section 6.3, Rules on file names, and for the structure of the source module file, refer to Section 5.1, 'Input/Output files handled by the assembler'.)

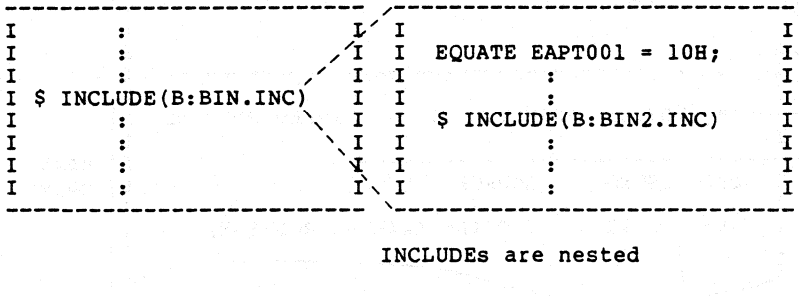
The 'INCLUDE' control cannot be nested. In other words, the 'INCLUDE' control cannot be used in a file that is being included. Optional blanks can be inserted before and after 'INCLUDE'.

Examples:

1) Correct use



2) Incorrect use



INCLUDEs are nested

2) Control for outputting lists

There are three different controls used for controlling the output formats of assembly lists.

For title printing
For page ejection
For starting/stopping list output

a) Control for title printing

Format

```
$[ ]TITLE[ ](character string)
```

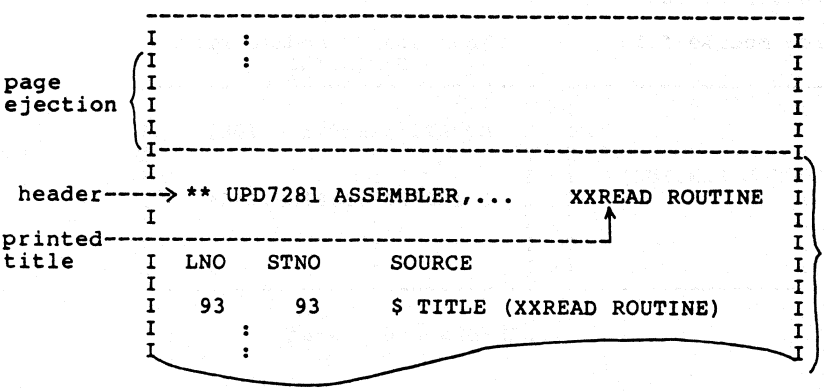
Description

Whenever this control appears in a source module, the assembler performs page ejection and prints the specified title as a part of header information on each new page.

The control itself (the string that forms the control) is printed after each page ejection. The string of characters, specified after 'TITLE', can be up to PAGEWIDTH*-60 characters.
(12 ≤ number of characters ≤ 72)

When the specified number of characters is greater than the above limit, the exceeding part of the string is truncated.

Example; $ TITLE(XXREAD ROUTINE)



(*) PAGEWIDTH = number of columns specified by 'PAGEWIDTH' control.

b) Control for page ejection

Format

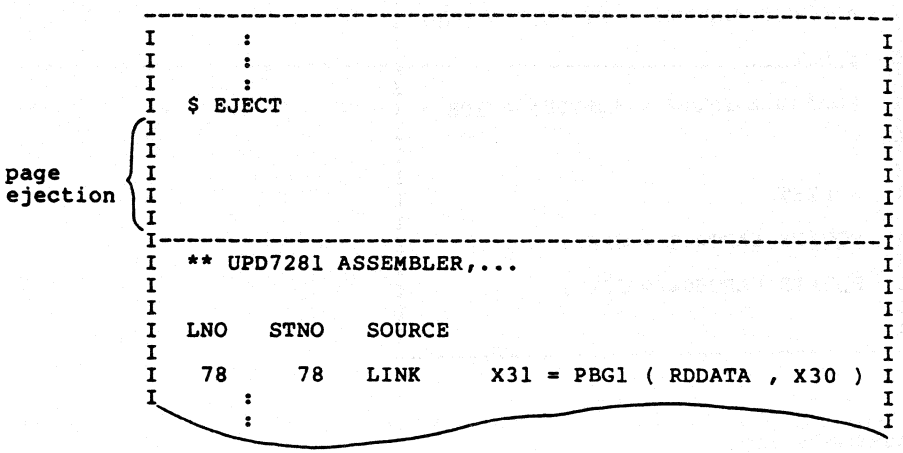
```
$[ ]EJECT
```

Description

Whenever this control appears in the source module, the assembler outputs a form feed(FF)

ASCII character to the printer for a new page. The character string of the control itself is printed at the page before ejection. Optional blanks can be inserted between '$' and 'EJECT'.

Example:



c) Control for starting/stopping list output

Format

If the assembly list output is desired

```
$[ ]LIST
```

If the assembly list output is not desired

```
$[ ]NOLIST
```

Description

Specifies either to start or to stop sending the assembly list to the list file. This control is used only for the assembly ouput control. Therefore, an assembly may be continued even after the '$ NOLIST' has been specified. Initially, the assembler assumes '$ LIST' has been specified. Optional blanks can be inserted after '$'.

Example:

Source module

```
                :
                :
  ADDRESS ADPT001 = PT(PU) AT OAH;

  $ NOLIST

  EQUATE EAPT002 = ADPT001 + 10H ;

  ;

  $ LIST

  EQUATE EAPT003 = 0FH ;

  EQUATE EAPT004 = 20H ;
                :
                :
```

Assembly list

```
                :
                :
  ADDRESS ADPT001 = PT(PU) AT OAH;

  $ NOLIST

  $ LIST

  EQUATE EAPT003 = 0FH ;

  EQUATE EAPT004 = 20H ;
                :
                :
```

6.3 Rules on file names

Some rules on file names handled by the assembler are explained below. The file names are classified into disk files and non-disk files.

(1) Disk files

Format for disk file names is shown below.

```
------------------------------
     d:primary name.ext
------------------------------
```

d: ; Name of the disk drive where the file is or to be located.

primary name ; A string of characters with 8 or less number of characters.

ext ; A string of characters with 3 or less number of characters.

1) When drive name is omitted

When the specified file is an input file, the current default drive name is assumed. If it is an output file, the same disk drive name as the input file is assumed.

2) When file type is omitted

When 'ext' (file type extension) is omitted, one of the default extension shown in Table 6-3 is assumed, depending on the file type.

Table 6-3 Default file type

File	Default file type
Source module file	-------
Object module file	LNK
List file	PRN

Examples;

1) A:MDMAIN.SRC ... Drive name is A:, primary name is MDMAIN, and file extension is SRC.

2) SUB1 ..... Drive name and file type are omitted. Primary name is SUB1.

(2) Non-disk files

format

```
-----------------
  device name:
-----------------
```

One of the following can be specified as a device name.

```
CON ..... console
LST ..... printer
```

Examples;

1) CON: ..... Outputs assembly list on console.

2) PRINT(LST:) ... Outputs assembly list to the printer.

CHAPTER 7 EXECUTION EXAMPLE

```
TYPE NOT.SRC
;**************************************************************
;
;               N O T   O P E R A T I O N
;
;-------------------------------------------------------------
;
MODULE  IPP     =       8                                       ;
;
EQUATE  H       =       32                                      ;
EQUATE  V       =       32                                      ;
EQUATE  L       =       64                                      ;
;
EQUATE  HOST    =       0                                       ;
EQUATE  READ    =       4                                       ;
EQUATE  WRITE   =       5                                       ;
;
EQUATE  STARTS  =       0                                       ;
EQUATE  STARTD  =       32                                      ;
;
;**************************************************************
;
;               INPUT - OUTPUT
;
;-------------------------------------------------------------
;
INPUT   LSA0,   LDA0,   RDATA AT 0                              ;
;
OUTPUT  RDAT,   WDAT,   WADR,   PEND                            ;
;
;**************************************************************
;
;               LINK TABLE
;
;-------------------------------------------------------------
;
LINK    MSA0,   RSA0,   RSA1    =  FGEN1 (LSA1, LSA0)           ;
LINK    IMR1,   RSA2,   RSA3    =  FGEN2 (MSA1, MSA0)           ;
LINK    LSA1                    =  FQUE1 (RSA0, RSA3)           ;
LINK    PEND                    =  FOUTE (RSA1, RDA1)           ;
LINK    RDAT                    =  FIMRED(IMR1)                 ;
LINK    MSA1,   MDA1            =  FQNP  (RSA2, RDA2)           ;
LINK    IMW0                    =  FNOT  (RDATA)                ;
LINK    MDA0,   RDA0,   RDA1    =  FGEN3 (LDA1, LDA0)           ;
LINK    IMW1,   RDA2,   RDA3    =  FGEN4 (MDA1, MDA0)           ;
LINK    LDA1                    =  FQUE2 (RDA0, RDA3)           ;
LINK    WDAT,   WADR            =  FIMWRT(IMW0, IMW1)           ;
;
;**************************************************************
;
;               FUNCTION TABLE
;
;-------------------------------------------------------------
;
FUNCTION    FIMRED =  OUT1 (READ, 0)                            ;
FUNCTION    FIMWRT =  OUT2 (WRITE, 20H, 0),QUEUE (QUEW,16)      ;
FUNCTION    FOUTE  =  OUT1 (HOST, 0), QUEUE (QUEE, 1)           ;
FUNCTION    FGEN1  =  COPYBK ( 1, 1), CNTGE (H)                 ;
FUNCTION    FGEN2  =  COPYBK (16, L), CNTGE (V)                 ;
FUNCTION    FGEN3  =  COPYBK ( 1, 1), CNTGE (H)                 ;
FUNCTION    FGEN4  =  COPYBK (16, L), CNTGE (V)                 ;
FUNCTION    FQNP   =  NOP     (XY),     QUEUE (QUEN, 1)       ;
FUNCTION    FNOT   =  NOT     (X)                               ;
FUNCTION    FQUE1  =  QUEUE   (QUE1, 1)                         ;
FUNCTION    FQUE2  =  QUEUE   (QUE2, 1)                         ;
```

```
;
;*************************************************  ************
;
;                DATA MEMORY
;
;-------------------------------------------------------------
;
MEMORY   QUE1 =  AREA ( 1)                                    ;
MEMORY   QUE2 =  AREA ( 1)                                    ;
MEMORY   QUEN =  AREA ( 1)                                    ;
MEMORY   QUEW =  AREA (16)                                    ;
MEMORY   QUEE =  AREA ( 1)                                    ;
;
;*************************************************************
;
;                START
;
;-------------------------------------------------------------
;
START
;
DATA     EXEC (IPP, LSA0, STARTS)                             ;
DATA     EXEC (IPP, LDA0, STARTD)                             ;
;
END
```

```
A>A:7281 B:NOT.SRC

UPD7281 ASSEMBLER E1.0i [30 Jan 85]
   Copyright (C) 1985 NEC Corporation


ASSEMBLY COMPLETE,     0 ERROR(S),     0 WARNING(S) FOUND

A>TYPE B:NOT.PRN
```

```
** UPD7281 ASSEMBLER, V1.0 **      PROGRAM NAME

SOURCE FILE ; B:NOT.SRC

OBJECT FILE ; B:NOT.LNK

LIST FILE   ; B:NOT.PRN

ERROR FILE  ;

COMMAND     ; B:NOT.SRC

LNO   STNO     SOURCE

  1      1     ;*************************************************************
  2      2     ;
  3      3     ;                  N O T   O P E R A T I O N
  4      4     ;
  5      5     ;-------------------------------------------------------------
  6      6     ;
  7      7     MODULE    IPP     =      8                                  ;
  8      8     ;
  9      9     EQUATE    H       =      32                                 ;
 10     10     EQUATE    V       =     32                                  ;
 11     11     EQUATE    L       =     64                                  ;
 12     12     ;
 13     13     EQUATE    HOST    =     0                                   ;
 14     14     EQUATE    READ    =     4                                   ;
 15     15     EQUATE    WRITE   =     5                                   ;
 16     16     ;
 17     17     EQUATE    STARTS  =     0                                   ;
 18     18     EQUATE    STARTD  =     32                                  ;
 19     19     ;
 20     20     ;*************************************************************
 21     21     ;
 22     22     ;                  INPUT - OUTPUT
 23     23     ;
 24     24     ;-------------------------------------------------------------
 25     25     ;
 26     26     INPUT     LSA0,    LDA0,    RDATA AT 0                      ;
 27     27     ;
 28     28     OUTPUT    RDAT,    WDAT,    WADR,    PEND                   ;
 29     29     ;
 30     30     ;*************************************************************
 31     31     ;
 32     32     ;                  LINK TABLE
 33     33     ;
 34     34     ;-------------------------------------------------------------
 35     35     ;
 36     36     LINK      MSA0,    RSA0,    RSA1     =  FGEN1 (LSA1, LSA0)  ;
 37     37     LINK      IMR1,    RSA2,    RSA3     =  FGEN2 (MSA1, MSA0)  ;
 38     38     LINK      LSA1                       =  FQUE1 (RSA0, RSA3)  ;
 39     39     LINK      PEND                       =  FOUTE (RSA1, RDA1)  ;
 40     40     LINK      RDAT                       =  FIMRED(IMR1)        ;
 41     41     LINK      MSA1,    MDA1              =  FQNP  (RSA2, RDA2)  ;
 42     42     LINK      IMW0                       =  FNOT  (RDATA)       ;
 43     43     LINK      MDA0,    RDA0,    RDA1     =  FGEN3 (LDA1, LDA0)  ;
 44     44     LINK      IMW1,    RDA2,    RDA3     =  FGEN4 (MDA1, MDA0)  ;
```

** ASSEMBLER, V1.0 **

```
LNO   STNO    SOURCE

  45     45   LINK    LDA1                    =  FQUE2 (RDA0, RDA3)      ;
  46     46   LINK    WDAT,   WADR            =  FIMWRT(IMW0, IMW1)      ;
  47     47   ;
  48     48   ;************************************************************
  49     49   ;
  50     50   ;               FUNCTION TABLE
  51     51   ;
  52     52   ;------------------------------------------------------------
  53     53   ;
  54     54   FUNCTION     FIMRED =  OUT1 (READ, 0)                        ;
  55     55   FUNCTION     FIMWRT =  OUT2 (WRITE, 20H, 0),QUEUE (QUEW,16);
  56     56   FUNCTION     FOUTE  =  OUT1 (HOST, 0), QUEUE (QUEE, 1)      ;
  57     57   FUNCTION     FGEN1  =  COPYBK ( 1, 1), CNTGE (H)            ;
  58     58   FUNCTION     FGEN2  =  COPYBK (16, L), CNTGE (V)            ;
  59     59   FUNCTION     FGEN3  =  COPYBK ( 1, 1), CNTGE (H)            ;
  60     60   FUNCTION     FGEN4  =  COPYBK (16, L), CNTGE (V)            ;
  61     61   FUNCTION     FQNP   =  NOP     (XY),   QUEUE (QUEN, 1)      ;
  62     62   FUNCTION     FNOT   =  NOT     (X)                          ;
  63     63   FUNCTION     FQUE1  =  QUEUE   (QUE1, 1)                    ;
  64     64   FUNCTION     FQUE2  =  QUEUE   (QUE2, 1)                    ;
  65     65   ;
  66     66   ;************************************************************
  67     67   ;
  68     68   ;               DATA MEMORY
  69     69   ;
  70     70   ;------------------------------------------------------------
  71     71   ;
  72     72   MEMORY  QUE1 =  AREA ( 1)                                   ;
  73     73   MEMORY  QUE2 =  AREA ( 1)                                   ;
  74     74   MEMORY  QUEN =  AREA ( 1)                                   ;
  75     75   MEMORY  QUEW =  AREA (16)                                   ;
  76     76   MEMORY  QUEE =  AREA ( 1)                                   ;
  77     77   ;
  78     78   ;************************************************************
  79     79   ;
  80     80   ;               START
  81     81   ;
  82     82   ;------------------------------------------------------------
  83     83   ;
  84     84   START;
  85     85   ;
  86     86   DATA    EXEC (IPP, LSA0, STARTS)                            ;
  87     87   DATA    EXEC (IPP, LDA0, STARTD)                            ;
  88     88   ;
  89     89   END;
```

```
** UPD7281 ASSEMBLER, V1.0 **      PROGRAM NAME

OBJECT CODE LIST ( MNEMONIC )      MODULE NAME   IPP        NO.  8

  ** DM   SECTION **

LOC.  OBJECT          LABEL      MNEMONIC

0000                  QUEE     : DS          0001H
0001                             DC          0001H
0002                  QUEN     : DS          0001H
0004                             ORG         0004H
0004                             DC          0040H
0006                             ORG         0006H
0006                  QUE1     : DS          0001H
0007                             DC          0001H
0008                  QUE2     : DS          0001H
000A                             ORG         000AH
000A                             DC          0040H
000C                             ORG         000CH
000C                  QUEW     : DS          0010H

** DM   24 WORDS USED **


  ** LT   SECTION **

LOC.  OBJECT          LABEL      MNEMONIC

0000  0099           RDATA    : PU          IMW0     , FNOT     ,  0
0001  1826           MDA0     : GE          IMW1     , FGEN4    ,  1
0002  0483           RDA0     : AG&FC       LDA1     , FQUE2    ,  0
0003  0804           RDA1     : OUT         PEND     , FOUTE    ,  1
0004  2104           IMW1     : OUT         0020H    , FIMWRT   ,  1
0005  1C6D           RDA2     : PU          MSA1     , FQNP     ,  1
0006  0487           RDA3     : AG&FC       LDA1     , FQUE2    ,  1
0007  1056           MSA0     : GE          IMR1     , FGEN2    ,  1
0008  0093           RSA0     : AG&FC       LSA1     , FQUE1    ,  0
0009  0800           RSA1     : OUT         PEND     , FOUTE    ,  0
000A  0400           IMR1     : OUT         PEND     , FIMRED   ,  0
000B  1C69           RSA2     : PU          MSA1     , FQNP     ,  0
000C  0097           RSA3     : AG&FC       LSA1     , FQUE1    ,  1
000D  1052           MSA1     : GE          IMR1     , FGEN2    ,  0
000E  1822           MDA1     : GE          IMW1     , FGEN4    ,  0
000F  140E           LDA0     : GE          MDA0     , FGEN3    ,  1
0010  140A           LDA1     : GE          MDA0     , FGEN3    ,  0
0011  0C3E           LSA0     : GE          MSA0     , FGEN1    ,  1
0012  0C3A           LSA1     : GE          MSA0     , FGEN1    ,  0
0013  2100           IMW0     : OUT         0020H    , FIMWRT   ,  0

** LT   20 WORDS USED **
```

```
** UPD7281 ASSEMBLER, V1.0 **      PROGRAM NAME

OBJECT CODE LIST ( MNEMONIC )      MODULE NAME   IPP        NO.  8


  ** FT   SECTION **

LOC.  OBJECT          LABEL      MNEMONIC

0000  000C            FNOT     : NOT
0000    3030 0000     FQUE1    : QUEUE      QUE1     ,0001H
0001  0004            FIMRED   : OUT1        4,0000H
0001    3040 0000     FQUE2    : QUEUE      QUE2     ,0001H
0002  2000            FOUTE    : OUT1        0,0000H,FULL
        3000 0000                QUEUE      QUEE     ,0001H
0003  2000            FGEN1    : COPYBK     0001H,FULL
        E01F 0000                CNTGE      0020H
0004  23C0            FGEN2    : COPYBK     0010H,FULL
        E01F 0000                CNTGE      0020H
0005  2000            FGEN3    : COPYBK     0001H,FULL
        E01F 0000                CNTGE      0020H
0006  23C0            FGEN4    : COPYBK     0010H,FULL
        E01F 0000                CNTGE      0020H
0007  2C1B            FQNP     : NOP        XY ,FULL
        3010 0000                QUEUE      QUEW     ,0001H
0008  2805            FIMWRT   : OUT2        5,0020H,0000H,FULL
        306F 0000                QUEUE      QUEW     ,0010H

** FTL    9 WORDS USED **
** FTR    9 WORDS USED **
** FTT    9 WORDS USED **

OBJECT CODE LIST ( MNEMONIC )      MODULE NAME              NO.


  ** DATA SECTION **

LOC.  OBJECT          LABEL      MNEMONIC

----  81100000                   EXEC       LSA0     ,0000H
----  80F00020                   EXEC       LDA0     ,0020H

ASSEMBLY COMPLETE,     0 ERROR(S),     0 WARNING(S) FOUND
```

APPENDIX 1 LIST OF RESERVED WORDS

(1) Statements

ADDRESS	AT	DATA	DEFINE	DMSETLT
DMSETFT	END	EQUATE	FUNCTION	INPUT
LINK	LITERAL	LOCATE	MEMORY	MODULE
NAME	OUTPUT	START		

(2) Instructions

ACC	ADD	ADDSC	ADJL	AND
ANDMSK	ANDNOT	CLR1	CMP	CMPNOM
CMPXCH	CNTGE	CONVO	COPYBK	COPYC
COPYM	COUNT	CUT	CVT2AB	CVTAB2
DEC	DIST	DIV	DIVCYC	GET1
INC	MUL	MULSC	NOP	NOPSC
NOT	OR	ORMSK	OUT1	OUT2
PICKUP	QUEUE	RDCYCL	RDCYCS	RDIDX
RDWR	SAVE	SET1	SETCTL	SHL
SHLBRV	SHR	SHRBRV	SUB	SUBSC
WRCYCL	WRCYCS	XOR		

(3) Input tokens

CBRK	CRESET	DMR	DMS	DUMP
EXEC	PASS	RDFTL	RDFTR	RDFTT
RDLT	SETBRK	SETFTL	SETFTR	SETFTT
SETLT	SETMD	VAN		

(4) Parameters of PU instructions

BRC	CNOP	EQ	GE	GT
LE	LT	NE	OVF	X
XCH	XX	XY	Y	

(5) Operators

AND	NOT	OR	SHL	SHR
XOR				

(6) Others

AREA	C	DM	FT	LT

APPENDIX 2 COMMAND NAMES AND PARAMETER LIST FORMAT FOR DATA STATEMENT

Command name	Parameter list format
SETLT	(Module number, LT address, LT write data)
SETFTL	(Module number, FT address, FT Left-field write data)
SETFTR	(Module number, FT address, FT Right-field write data)
SETFTT	(Module number, FT address, FT Temporary-field write data)
RDLT	(Module number, LT address)
RDFTL	
RDFTR	(Module number, FT address)
RDFTT	
CRESET	(Module number)
SETMD	(Module number, constant, Refresh Register data, Input Inhibit Register data)
SETBRK	(Module number, ID, constant, count data)
DUMP	(Module number, dump mode)
CBRK	----------
VAN	----------
PASS	(Module number, ID, data)
EXEC	(Module number, ID, data)
DMS	(Module number, ID, first address to DM, list of data)
DMR	(Modulenumber,ID,first address to DM, number of reads)

APPENDIX 3 ERROR MESSAGES

There are two types of error messages output by the assembler; the error messages displayed on the console and the messages printed out on the assembly list.

Error messages displayed on console

There are four types of error messages that are displayed on console.

a) Error messages related to the assembler invocation commands.
b) Error messages related to the assembler's internal processing.
c) Error messages related with fatal file-I/O errors.
d) Error messages related to source program assembly.

The error messages related to assembly errors (d) are displayed on the console even when the assembly list is not to be output (e.g. when 'NOPRINT' control is specified at the time of assembler invocation). For the case of type a),b), and c) type of errors, the assembler returns control to the Operating System after outputting the error messages.

The following is the list of error messages mentioned above.

a) Error messages related to the assembler invocation commands

Format; *** ERROR error number error message
PROGRAM ABORTED

Error number F001	Message	MISSING FILE SPECIFICATION
	Cause	Source module file is not specified.
	User's operation	Specify the source module file and restart the assembler.
Error number F002	Message	ILLEGAL FILE SPECIFICATION - file name
	Cause	Incorrect file name.
	User's operation	Specify thet correct file name and restart the assembler.

	Message	FILE SPECIFICATION CONFLICTED - file name
Error number F003	Cause	There is a conflict between either input file name and output file name or among output file names.
	User's operation	Specify correct file name(s) and restart the assembler.
	Message	FILE NOT FOUND - file name
Error number F005	Cause	Source module file does not exist.
	User's operation	Specify the correct source module file name and restart the assembler.
	Message	WRITE PROTECTED FILE - file name
Error number F006	Cause	The output file is write-protected.
	User's operation	Either remove the write protection or specify another output file and then restart the assembler.
	Message	ILLEGAL OR MISSING PARAMETER - parameter
Error number F007	Cause	Either a parameter (file name) is not specified for the control, or it is specified incorrectly.
	User's operation	Specify the correct parameter and restart the assembler.
	Message	CONTROL IS NOT RECOGNIZED
Error number F008	Cause	Improper character string specified for control.
	User's operation	Specify correct control and restart the assembler.

b) Error messages related to the internal processing of the assembler.

Format; *** ERROR error number error message
PROGRAM ABORTED

Error number	Message	WORKING TABLE SPACE EXHAUSTED
A901	Cause	Work area (memory) is not large enough.
	User's operation	Reduce symbol definitions.
Error number	Message	INVALID FILE SYNTAX - file name
A902	Cause	Source module file is incorrectly formatted.
	User's operation	Correct the source module file by using editor, etc.

c) Error messages related with fatal file-I/O errors

Format; *** ERROR error number error message
PROGRAM ABORTED

Error number	Message	FATAL I/O ERROR - file name
A903	Cause	File-read error File-write error Assembler's overlay files (AS7281.OMn) do not exist File specified by INCLUDE control does not exist
	User's operation	Verify the presence of file, its attributes, remaining area on disk, etc.

d) Error messages that are printed out on the assembly list

These error messages are printed out on the line immediately after the source statement containing the error.

Format; *** ERROR error number error message

*I&P stands for Interpretation and Processing.

Error number F010	Message	ILLEGAL GENERAL CONTROL
	Cause	General control is coded incorrectly.
	Program I&P	The command is ignored and processing continues.
Error number F011	Message	SYNTAX ERROR
	Cause	Syntax error in the statement being evaluated.
	Program I&P	Program locations impossible to be evaluated are ignored, and processing continues.
Error number F012	Message	ILLEGAL ATTRIBUTE OF EXPRESSION
	Cause	Attribute of the expression being evaluated mismatches the expected type.
	Program I&P	The expression is evaluated to 0(zero) and processing continues.
Error number F013	Message	BALANCE ERROR
	Cause	Mismatched parentheses or quotation marks.
	Program I&P	In evaluation of expression, its value is set to 0 (zero) and processing continues. In other cases, processing goes on while evaluation is possible; places that cannot be evaluated are ignored, and processing continues.
Error number F014	Message	ILLEGAL CHARACTER IN NUMERIC CONSTANT
	Cause	Presence of illegal character in numeric constant.
	Program I&P	The numeric constant is evaluated as 0 (zero) and processing continues.

Error number	Message	FORMAT ERROR
F015	Cause	Coding format mismatches the expected.
	Program I&P	Processing goes on while evaluation is possible; places that cannot be evaluated are ignored and processing continues.
Error number	Message	SYMBOL FIELD ALREADY DEFINED
F016	Cause	Symbol is already defined in another place.
	Program I&P	Statement attempts to redefine the symbol. It is ignored.
Error number	Message	UNDEFINED SYMBOL
F017	Cause	Undefined symbol is being referenced.
	Program I&P	Undefined symbol is not evaluated, and processing continues.
Error number	Message	STACK OVERFLOW IN EXPRESSION EVALUATION
F018	Cause	Stack for evaluation of expressions has overflowed.
	Program I&P	Evaluation value is set to zero and processing continues.
Error number	Message	ILLEGAL CHARACTER
F019	Cause	Illegal character was coded.
	Program I&P	Processing goes on while it is possible; places that cannot be processed are ignored, and processing continues.
Error number	Message	ATTRIBUTE OF SYMBOL UNDEFINED
F020	Cause	Attribute of symbol is not specified in ADDRESS statement.
	Program I&P	The statement is ignored.

Error number	Message	ADDRESS EXCEEDS THE RANGE
W101	Cause	Address assignment for symbols exceeded both upper and lower bounds.
	Program I&P	Symbol is assigned to the specified address and processing continues.
Error number	Message	ILLEGAL PRESET OF THE ADDRESS RANGE
W102	Cause	Illegal upper and lower bounds for address assignment.
	Program	Default values are assumed for address assignment range and processing continues.
Error number	Message	ADDRESS DEFINED TO SYMBOL OF OUTPUT STATEMENT
W103	Cause	'AT expression' was used in LINK statement for symbol coded in OUTPUT statement.
	Program I&P	'AT expression' is ignored and processing continues.
Error number	Message	ILLEGAL DESCRIPTION OF AT EXPRESSION
W104	Cause	'AT expression' was used when no symbol was coded in LINK statement.
	Program I&P	'AT expression' is ignored and processing continues.
Error number	Message	RAM ADDRESS OVERLAP
W105	Cause	The same address as the default for DMSETLT, DMSETFT was specified. The same address was specified for different symbols within a module.
	Program I&P	Overlapping address is assigned and processing continues.
Error number	Message	ATTRIBUTE OF SYMBOL IN EXPRESSION UNMATCHED
W106	Cause	Attribute of the first operand mismatches that of the second operand in a expression.
	Program I&P	The attribute of the first operand is set to NUMBER and processing continues.

Error number	Message	REFERENCED SYMBOL NOT DEFINED
W107	Cause	Symbol that is not referenced by MEMORY statement, LINK statement or FUNCTION statement was defined.
	Program I&P	This error message is output and normal processing is carried out.

Error number	Message	ILLEGAL MODULE NUMBER
W108	Cause	0 or a number greater than 15 was specified to module number.
	Program I&P	This error message is output and normal processing is carried out. (For all numbers greaterthan 15 the module number is setto 15.)

Error number	Message	START STATEMENT ALREADY APPEARED
W109	Cause	The coded START statement is overlapped.
	Program I&P	The START statements from the second on are ignored and processing continues.

Error number	Message	END STATEMENT MISSING
W110	Cause	END statement is not coded.
	Program I&P	An END statement is assumed to be present at the end of the source module and the processing continues.

Error number	Message	ILLEGAL STATEMENT
F111	Cause	Statement that is neither general control nor name definition statement was coded in declaration section. Statement that is not a DATA statement was coded in DATA section.
	Program I&P	The statement is ignored and processing continues.

Error number	Message	MISSING PNZ CONDITION
W112	Cause	Expected PNZ is not specified for comparison instruction of PU instruction (CMPNOM, CMP or CMPXCH).
	Program I&P	Default (PNZ=0) is assumed and processing continues.

Error number	Message	ILLEGAL PNZ PRESET
W113	Cause	PNZ was specified in instruction in which PNZ cannot be specified.
	Program I&P	The specified value is set to PNZ and processing continues.
Error number	Message	ILLEGAL BRC PRESET
W114	Cause	'BRC=1' was preset when 2 outputs were specified for a PU instruction.
	Program I&P	The specified value is set and processing continues.
Error number	Message	ILLEGAL PRESET IN ACC INSTRUCTION
W115	Cause	A value other than ACC default value is specified for ACC instruction of PU instruction.
	Program I&P	Processing continues with the specified value.
Error number	Message	MORE ERRORS DETECTED, NOT REPORTED
W116	Cause	More than 4 errors were detected in the statement being evaluated.
	Program I&P	The errors after the 5th (included) are not output and processing continues.
Error number	Message	ARITHMETIC OVERFLOW IN NUMERIC CONSTANT
W117	Cause	The size specified by AG&FC instruction is larger than the allowed range.
	Program I&P	The largest usable DM range is assumed and processing continues.
Error number	Message	ILLEGAL DEFINED DATA MEMORY ADDRESS
W118	Cause	Base address of the area reserved in DM does not meet the conditions required by AG&FC instruction.
	Program I&P	The specified base address is assumed and processing continues.

Error number	Message	NAME STATEMENT ALREADY APPEARED
W119	Cause	The coded NAME statement was overlapped.
	Program I&P	NAME statements after the second (included) are ignored and processing continues.
Error number	Message	CHARACTER STRING OF LITERAL EXCEEDS LIMITS
W120	Cause	LITERAL statement's line being expanded exceeded 116 characters (including CR and LF codes).
	Program I&P	Up to the first 116 characters are evaluated (includingCR and LF codes) and the rest is ignored.
Error number	Message	IMPOSSIBLE ALLOCATION OF MEMORY SPACE
W121	Cause	Memory address can not not be assigned due to impossible memory allocation.
	Program I&P	The address is set to 0 and number of area locations impossible to be allocated is counted.
Error number	Message	OUTPUT ID. MISS MATCHED
W122	Cause	Destination ID specified in FUNCTION statement mismatches both module number and destination ID from INPUT and OUTPUT statements.
	Program I&P	Specified module number and destination ID are assumed and processing continues.
Error number	Message	TOO MANY SYMBOLS IN LINK STATEMENT
W123	Cause	The number of symbols defined in LINK statement exceeds the number required by the instruction coded in FUNCTION statement.
	Program I&P	Only coded symbol are processed.

Error number W124	Message	TOO FEW SYMBOLS IN LINK STATEMENT
	Cause	The number of symbols defined in LINK statement is less than the number required by the instruction coded in FUNCTION statement.
	Program I&P	Only coded symbols are processed.
Error number W125	Message	DIFFERENT INNER CONSTANT AT SAME ADDRESS IN DM MEMORY
	Cause	The same DM address is referenced from more than one place with different values.
	Program I&P	Processing is carried out as specified.
Error number W126	Message	AREA SIZE MUST BE POSITIVE
	Cause	Negative size was specified in MEMORY statement.
	Program I&P	Reserved area size is assumed to be 0 and processing goes on.

APPENDIX 4 CODING FORMAT OF INSTRUCTIONS

The instructions to be coded in FUNCTION statements are classified into four groups: OUT instructions, PU instructions, GE instructions, and AG&FC instructions.

The coding formats for all instructions are explained below. (For details of instructions, refer to uPD7281 User's manual.)

1. OUT instructions

There are two OUT instructions: OUT1 and OUT2.

1.1 OUT1

(1) Coding format

```
OUT1(module number, destination ID[,XCH])
                    [,AG&FC instruction]
```

(2) Explanation of each item

1) Module number

A numeric constant or a module name. (It is set to MN.)

2) Destination ID

Destination ID of the first output.

3) XCH

Specification for exchanging two pieces of data. (It causes XCH flag to be set to 1.)

1.2 OUT2

(1) Coding format

```
OUT2(module number, first destination ID,
    second destination ID[,XCH])[,AG&FC instruction]
```

(2) Explanation of each item

1) Second destination ID

Destination ID of the second output.

2) Other items are the same as in OUT1.

2. PU instructions

The PU instructions are further classified into the following groups. (For the instruction names, refer to Table 4-1).

- o Logical operations
- o Arithmetic operations
- o Shift operations
- o Comparison operations
- o Accumulation
- o C-bit copy
- o Double precision adjust
- o Bit manipulation
- o Data exchange
- o Check bits

(1) Coding format

```
Op-code[([output specification][,BRC][,CNOP*][,PNZ]
                                [,XCH])][AG&FC instruction]
```

(*) Cannot be specified in C-bit copy instruction.

(2) Explanation of each item

1) Op-code
Table 4-1 (supplement) shows Op-codes and functions of each PU instruction.

Table 4-1 (supplement) Groups of operations and coding formats of PU instructions' Op-codes

Operations	Instruction name	Op-code	Function
Logical operations	AND	00000	Logical product
	OR	00001	Logical sum
	XOR	00010	Logical EXCLUSIVE OR
	ANDNOT	00011	Logical A B
	NOT	01100	Complement

Arithmetic operations	ADD	11000	Add
	SUB	11001	Subtract
	MUL	11010	Multiply
	NOP	11011	No operation
	ADDSC	11100	Add and shift count
	SUBSC	11101	Subtract and shift count
	MULSC	11110	Multiply and shift count
	NOPSC	11111	NOP and shift count
	INC	01010	Increment
	DEC	01011	Decrement
Shift operations	SHL	00100	Shift left
	SHLBRV	00101	Shift left with bit reverse
	SHR	00110	Shift right
	SHRBRV	00111	Shift right with bit reverse
Comparison operation	CMPNOM	01000	Compare and normalize
	CMP	01001	Compare
	CMPXCH	10001	Compare and exchange
Accumulation	ACC	10010	Accumulate
Copy C-bit	COPYC	10011	Copy control bit
Double precision adjust	ADJL	10100	Adjust long (for double precision numbers)
Bit manipulation	GET1	10101	Get one bit
	SET1	10110	Set one bit
	CLR1	10111	Clear one bit

Data exchange	CVT2AB	01110	Convert 2's complement to sign-magnitude
	CVTAB2	01111	Convert sign-magnitude to 2's complement
Bit check	ORMSK	10000	Mask with logical AND
	ANDMSK	01101	Mask with logical OR

2) Output specification

Either X, Y, XX, or YY is specified.

X	X	OUT=00
Y	Y	OUT=01
XX	XX	OUT=10
YY	YY	OUT=11

3) BRC

Coding this item causes BRC to be set to 1 (BRC=1).

4) CNOP

Coding this item causes NOP to be set to 1 (NOP=1).

5) PNZ

One of the following items is specified.

		PNZ
EQ	EQ	001
NE	NE	110
LE	LE	011
LT	LT	010
GE	GE	101
GT	GT	100
OVF	OVF	111

6) XCH

Specifies two operand data to be exhanged before a PU operation . (XCH flag is set to 1.)

(3) Default parameters in PU instructions

For each different type of PU instructions, the default values for omitting the parameters are shown in Table 4-2.

Table 4-2 (Supplement) Default parameters for each group of instructions

Operation group	Default value
Logical operations	NORMAL
Arithmetical operations	
Shift operations	
Comparison operations	NORMAL
Accumulation	ACC
C-bit copy	NORMAL
Double precision adjust	NORMAL
Bit manipulation	NORMAL
Data exchange	NORMAL
Check bit	CHECK BIT

1) NORMAL
OUT = 00
BRC = 0
CNOP = 0
PNZ = 0

2) ACC
OUT = 00
BRC = 1
CNOP = 0
PNZ = 0

3) CHECK BIT
OUT = 00
BRC = 0
CNOP = 0
PNZ ---- ORMSK = 110
ANDMSK = 001

3. GE instruction

There are three GE instructions: COPYBK, COPYM, and SETCTL.

3.1 COPYBK, COPYM

(1) Coding format

```
{COPYBK}
{COPYM } (constant1, internal constant[,XCH])
                     [,AG&FC instruction]
```

(2) Explanation on each item

1) Constant1

The number of generations is coded in this item.

a. For COPYBK instruction, constant1 must be within 1 and 16.
b. In the case of COPYM, constant1 must be within 2 and 17.

2) Internal constant

For a GE instruction, it is necessary to store a incremental/decremental value (or an internal constant) known "delta' in a DM location. The actual DM address used for a GE instruction is equal to the ID value of the token accessing the GE instruction.

Since the ID field is only 7 bits wide, only the first 128 DM locations can be used to store the internal constants. Thus, user should exercise caution when using the first 128 DM locations for program constants or tables.

3) XCH

It is the same as in OUT1 (Section 1.1, paragraph 3)).

3.2 SETCTL

(1) Coding format

```
SETCTL(constant1, internal constant, constant2[,XCH])
                                  [,AG&FC instruction]
```

(2) Explanation of each item

1) Constant1, internal constant, XCH

These items are the same as described in paragraphs 1)

and 2) of Section 3.1, COPYM.

2) Internal Constant

See explanation for COPYBK and COPYM instruction.

3) Constant2

Data to be set in CTLF is coded in this item.

4. AG&FC instructions

There are sixteen AG&FC instructions.

4.1 QUEUE

(1) Coding format

QUEUE(DM address, size)[,FTT field contents]

(2) Explanation of each item

1) DM address

This is the base address of the DM area to be used by QUEUE instruction.

2) Size ($1 \leq$ size ≤ 16)

This specifies the size of the DM area to be used as a queue.

3) FTT field contents

Either 'constant' or '?' is specified here. Both can be omitted.

```
a. ?          ; FTT field is not initialized.
b. constant   ; FTT field is initialized with the
                specified constant.
c. In the case where this item is omitted
              ; 0 is set to FTT field by default.
```

4.2 RDCYCS

(1) Coding format

RDCYCS(DM address, size)[,FTT field contents]

(2) Explanation of each item

1) DM address

This is the base address of the DM area to be used.

2) Size ($1 \leq$ size ≤ 16)

This specifies the size of the area to be used.

3) FTT field contents

It is the same as in Section 4.1, QUEUE.

4.3 RDCYCL

(1) Coding format

RDCYCL(DM address, size)[,FTT field contents]

(2) Explanation of each item

1) DM address

This specifies the base address of the DM area.

2) Size ($1 \leq$ size ≤ 256)

This specifies the size of the area to be used.

3) FTT field contents

It is the same as in Section 4.1, QUEUE.

4.4 WRCYCS

(1) Coding format

WRCYCS(DM address, size)[,FTT field contents]

(2) Explanation of each item

Each item is the same as in Section 4.2, RDCYCS.

4.5 WRCYCL

(1) Coding format

WRCYCL(DM address, size)[,FTT field contents]

(2) Explanation of each item

It is the same as in Section 4.3, RDCYCL.

4.6 RDWR

(1) Coding format

RDWR(DM address)[,FTT field contents]

(2) Explanation of each item

1) DM address

This is the DM address to be used.

2) FTT field content

It is the same as in Section 4.1, QUEUE.

4.7 RDIDX

(1) Coding format

RDIDX(DM address)[,FTT field contents]

(2) Explanation of each item

It is the same as in Section 4.6, RDWR.

4.8 PICKUP

(1) Coding format

PICKUP(size)[,FTT field contents]

(2) Explanation of each item

1) Size ($1 \leq$ size ≤ 256)

The Count Size is coded.

2) FTT field contents

It is the same as in Section 4.1, QUEUE.

4.9 COUNT

(1) Coding format

COUNT(size)[,FTT field contents]

(2) Explanation of each item

It is the same as in Section 4.8, PICKUP.

4.10 CUT

(1) Coding format

CUT(size)[,FTT field contents]

(2) Explanation of each item

It is the same as in Section 4.8, PICKUP

4.11 CONVO

(1) Coding format

CONVO(size)[,FTT field contents]

(2) Explanation of each item

It is the same as in Section 4.8, PICKUP.

4.12 CNTGE

(1) Coding format

CNTGE(size)[,FTT field contents]

(2) Explanation of each item

a. Size (1 $\leq$ size $\leq$ 256)

The Count Size is coded.

b. FTT field contents

It is the same as in Section 4.1, QUEUE.

4.13 DIVCYC

(1) Coding format

DIVCYC(sizel, size2)[,FTT field contents]

(2) Explanation of each item

a. Sizel

The Count Size is coded.

b. Size2

The Divide Size is coded.

c. The values of sizel and size2 are restricted to the following relations.

$1 \leq sizel \leq size2 \leq 16$

d. FTT field contents

It is the same as in Section 4.1, QUEUE.

4.14 DIV

(1) Coding format

DIV(size)[,FTT field contents]

(2) Explanation of each item

a. Size ($1 \leq size \leq 256$)

This specifies the count size.

b. FTT field contents

It is the same as in Section 4.1, QUEUE.

4.15 DIST

(1) Coding format

DIST(size)[,FTT field contents]

(2) Explanation of each item

a. Size ($1 \leq size \leq 16$)

The ID size is coded.

b. FTT field contents

It is the same as in Section 4.1, QUEUE.

4.16 SAVE

(1) Coding format

SAVE[,FTT field contents]

(2) Explanation of each item

a. FTT field contents

It is the same as in Section 4.1, QUEUE.

PART III

USAGE OF THE SIMULATOR

CHAPTER 1 OUTLINE OF THE SOFTWARE

1.1 Outline of Simulator functions

The purpose of the software simulator is to aid in writing programs for the uPD7281 and to take part in debugging the program during the program development cycle.

One important advantage of using the software simulator is the ability to fully simulate an entire image processing sub-system which may include many uPD7281s, a uPD9305, and a block of image memory. Being able to simulate a system is not only a good program debugging tool, but also is a way to evaluate system performance without actually building the hardware. Moreover, upon analyzing the simulation results, the user may even be able to optimize the uPD7281 software for a specific application given a system set-up.

The simulator also provides a powerful set of commands to debug uPD7281 programs interactively. The simulator accepts object files created by the assembler directly, and therefore, the user can actually use the same set of symbols created by the assembler, when using the simulator. Utilizing the fully symbolic debugger, break pointers, and the macro commands, the simulator can fully interact with the user, or it can accept a batch of commands from an include file.

One note to mention here is that the user needs to be somewhat familiar with the architecture of the uPD7281 in order to fully utilize the debugging capabilities of the simulator. However, the simulator may also serve as an educational tool for those who do not understand how the ImPP processes data internally.

1.1.1 The Simulator Models

The Simulator can fully simulate the inner-workings of the uPD7281, the Image Pipelined Processor. Additionally, it can also simulate a small block of external memory and the uPD7281's companion chip called 'MAGIC' (uPD9305). Therefore, an image processing sub-system consisting of uPD7281(s), a uPD9305 and an image memory can be simulated using the software simulator. Of course, the system may or may not include the image memory and/or the uPD9305. However, it must include at least one uPD7281 in a system. Since there may be many variations to actual system design, the software simulator offers the following three different system models to cover the majority of possible image processing sub-system configurations.

(1) A system model with both MAGIC and image memory

Fig. 1-1 shows one possible structure of a system model. The system model includes a peripheral LSI 'MAGIC' (Memory Access and General bus-Interface Chip), a block of image memory, and a chain of cascaded uPD7281's. The MAGIC's function in this model is to interface the uPD7281s to the image meory so that the ImPPs can read and write to the external memory. For more details on this model, refer to Section 2.3.1, 'Model of IM block at MAGIC mode'.

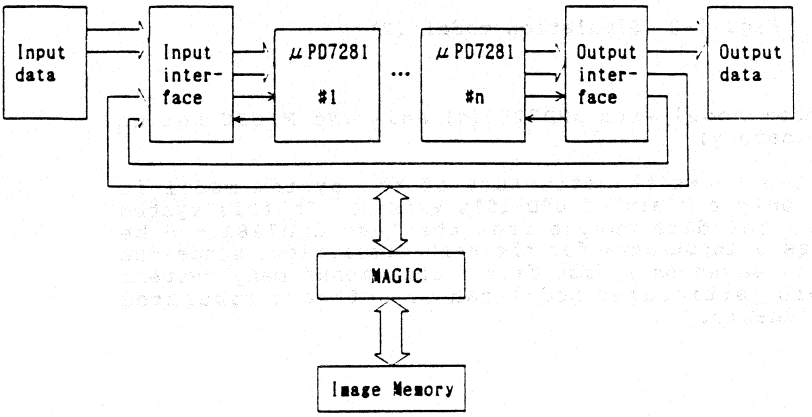


Fig. 1-1 Simulation model (1)

(2) A system model with an image memory (without MAGIC)

Fig. 1-2 shows the structure of the second system model which includes the image memory only. This model assumes that the user has interfaced the uPD7281s to the image memory using a user-specific memory interface. For more details on this model, please refer to Section 2.3.2, 'Model of IM block at NON-MAGIC mode'.

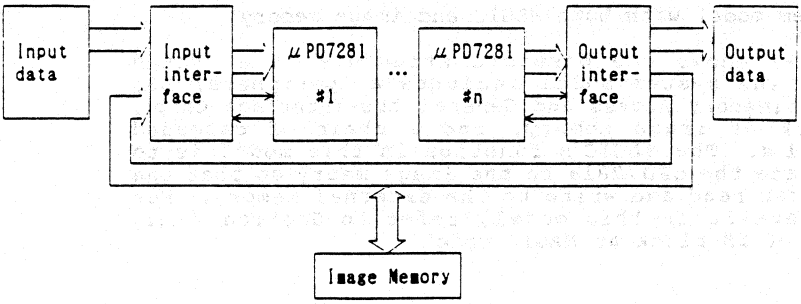


Fig. 1-2 Simulation model (2)

(3) A system model with uPD7281(s) only (No MAGIC and no image memory)

Fig. 1-3 shows the structure of the system model in which only a chain of uPD7281s exists. In this system model, the data output from the last uPD7281 can be used as a input data for the next simulation, since the data is saved on a disk file. This means many systems of this particlular model can be software simulated individually.

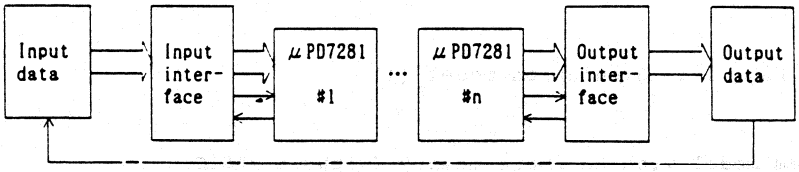


Fig. 1-3 Simulation model (3)

1.1.2 Symbolic debug

The simulator provides fully symbolic debugging capability. The same symbols created and used in source program file may be used in the simulator also, because the assembler generated object file contains the information on symbols. With the simulator the user can define, use or delete any LT symbols, FT symbols, DM symbols, and NUMBER symbols. Chapter 5 contains more information on symbolic debugging commands.

1.2 Organization of files handled by the Simulator

The simulator handles many different types of files. The relationships between the simulator and these files are shown in Fig. 1-4.

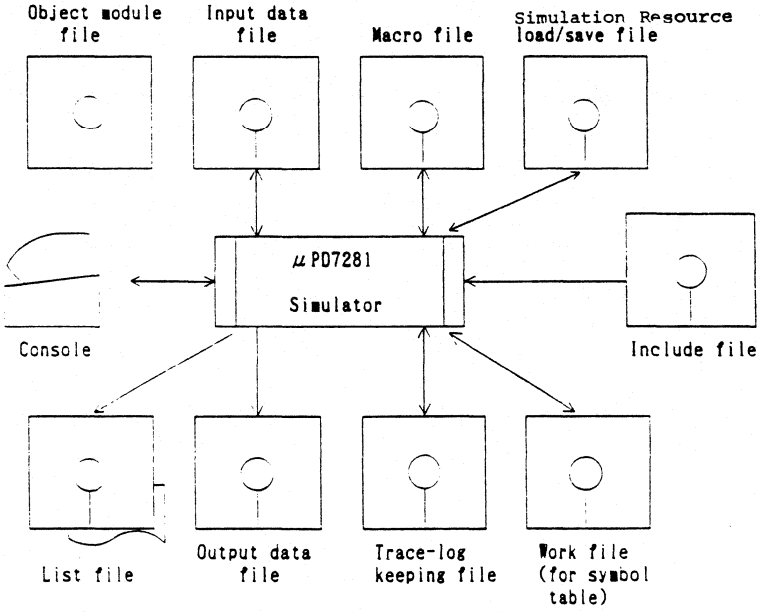


Fig. 1-4 Organization of files handled by the simulator

1.2.1 Work file

The simulator requires work space on a disk. For this purpose, the simulator automatically creates the work files on a disk specified by the user, and deletes them after the simulation. The main use of the work files is for the symbol tables.

The user may specify the work file name using the WORK subcommand in 'ENVIRON command'. However, if it is omitted, the work files are created on the default disk drive with the default file names given below. Since any file on the default drive with the default work file name is deleted by the Simulator at the end of the simulation, the user must not use the file name for other purposes.

primary name	file type
SM7281	.$$n (n is a sequence number)

CHAPTER 2 DETAILS OF SIMULATION MODELS

The majority of image processing sub-systems configured using ImPPs can be simulated using the software simulator. The three different models of system structures have been discussed in the previous chapter. Choosing one of the three hardware equivalent simulation models, the user is able to software simulate the hardware and also is able to debug/try any application specific algorithm before actually building the hardware.

This chapter explains the software simulator in detail from the software system models' point of view. Fig. 2-1 shows the block diagram of the simulation model for uPD7281 systems.

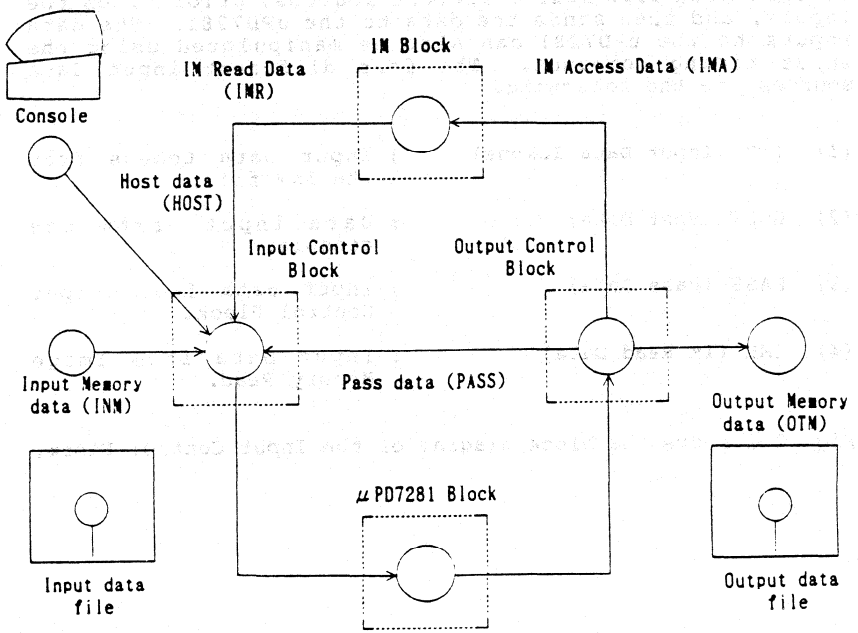


Fig. 2-1 Simulation model (block diagram)

2.1 Interactions between blocks

As shown in Fig. 2-1, there are the following four blocks in the simulation model.

1) Input Control Block
2) Output Control Block
3) IM Block
4) uPD7281 Block

The function of each block and the interactions between the blocks are explained below.

2.1.1 Input Control Block

The function of the Input Control Block is to control the tokens going into the uPD7281. The Input Control Block accepts data from four different sources, prioritizes the inputs, and then sends the data to the uPD7281. The data inputs to the uPD7281 can also be manipulated using the input timing commands. The four different input data sources are the following.

(1) INM (Input Data Tokens) ; Input data tokens from the INM file.

(2) HOST (Host Data) ; Data input from the console.

(3) PASS (Pass Data) ; Input data from Output Control Block.

(4) IMR (IM Read Data) ; Input Data from Image Memory Read.

Fig. 2-2 shows the block diagram of the Input Control Block.

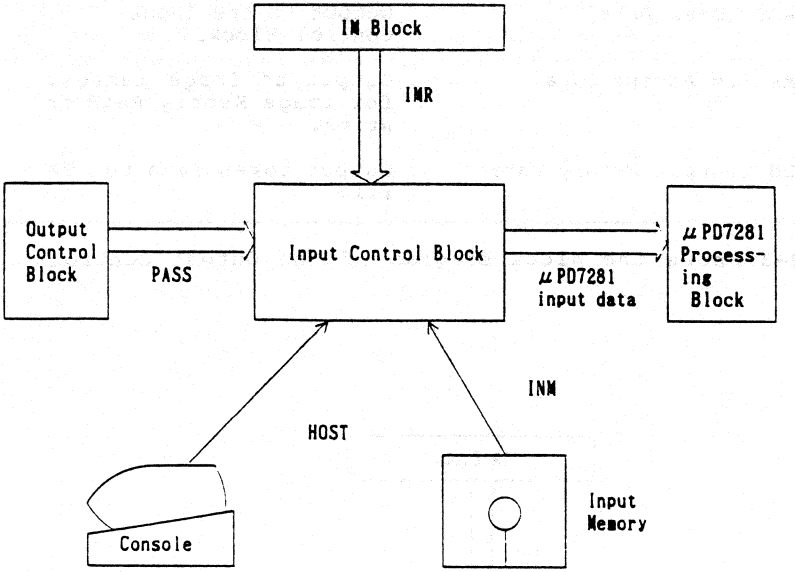


Fig. 2-2 Relation diagram of the Input Control Block

INM ..... Input Data Tokens from INM file
HOST ..... Host Data from the console
PASS ..... Pass Data from Output Control Block
IMR ..... Image Memory Read Data

2.1.2 Output Control Block

The function of the Output Control Block is to receive the data from the uPD7281 and then route the data to the proper destinations. There are three different destinations to which an output data token from the uPD7281 may be routed.

(1) PASS (Pass Data) ; Output to the Input Control Block.

(2) IMA (IM Access Data) ; Output to image memeory for Image Memory Read or Write.

(3) OTM (Output Memory Data) ; Output token data to OTM file.

Fig. 2-3 shows the block diagram of the Output Control Block.

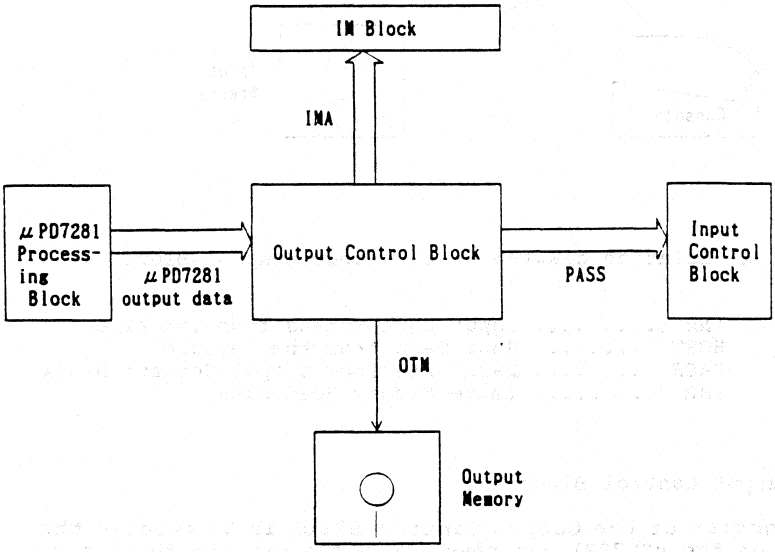


Fig. 2-3 Relation diagram of the Output Control Block

PASS ..... Pass Data to Input Control Block
IMA ..... IM Access Data to Image Memory
OTM ..... Output tokens to OTM file

2.1.3 IM Block

The function of the IM Block is to simulate the system's Image Memory. One note of caution is that not all of the image memory locations addressable by the MAGIC are simulated. In fact, only two planes of 256 memory locations are simulated using the software IM block, and therefore, great care must be given when IM block is used in the software simulation. Fig. 2-4 shows the block diagram of the IM Block.

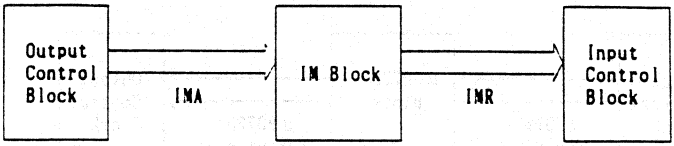


Fig. 2-4 Relation diagram of the IM Block

IMA .... Image Memory Access Token Data
IMR .... Image Memory Read Token Data

2.1.4 uPD7281 Block

The function of the uPD7281 Block is to simulate the inner workings of the uPD7281 hardware itself. The uPD7281 block accepts data tokens from the Input Control Block, processes the data as if the actual hardware were there, and then outputs the processed data tokens to the Output Control Block.

Since the uPD7281 Block literally software simulates the uPD7281 hardware, the user must be somewhat familiar with the architecture of the uPD7281 in order to understand the software simulation.

The only functional block within the uPD7281 that is not completely software simulated is the Refresh Controller which is used to generate Refresh Tokens for internal dynamic RAMs. Since the Refresh Controller in the uPD7281 hardware is totally transparent to the user, the software simulation of the Refresh Controller may not be required in most cases.

Fig. 2-5 shows the relations between the uPD7281 block and other software simulation blocks. Fig. 2-6 shows the internal architecture of the uPD7281.

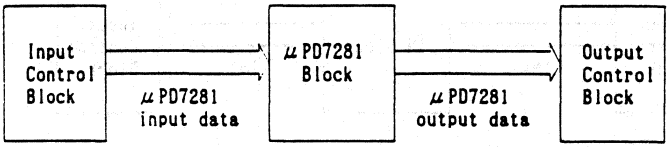


Fig. 2-5 Relation diagram of uPD7281 Block

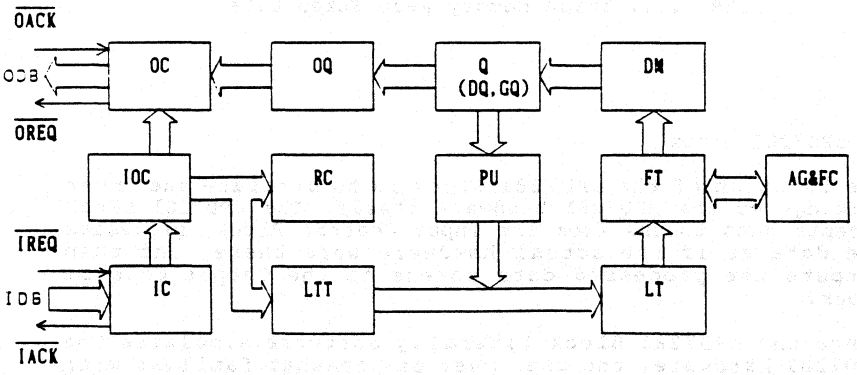


Fig. 2-6 Internal Architecture of the uPD7281.

2.2 The uPD7281 simulation model

The uPD7281 software simulation model used is almost equivalent to the actual uPD7281 hardware. Since the uPD7281 adequately explains the internal architecture, this manual will not duplicate the effort. However, there are five functional blocks that are not exact equivalents of the uPD7281 hardware. The following sub-sections illustrate and explain the software models used for these five special cases.

The figures in the following sub-sections use signal names related to data contained in a counter, a latch, a register, etc.. Those signal names carrying the actual data in the figures are designated with the prefix symbol of '@' sign.

2.2.1 IC Block (Input Controller)

As Fig. 2-7 shows, the IC Block accepts a 32-bit input token from the Input Control Block. The actual token transfers from the @INB to other functional blocks are done in accordance with the internal flags and the handshake signals.

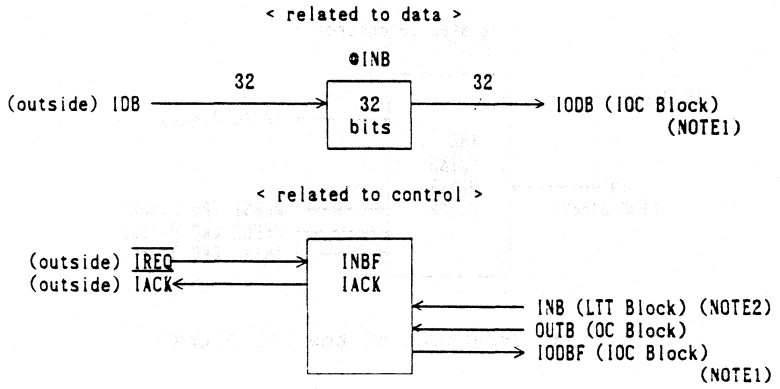


Fig. 2-7 Structure of the IC Block

@INB ..... 32-bit Data Input Latch
INBF ..... Empty/Full Flag of @INB
IACK ..... Input Acknowledge Flag

NOTES: 1. The Simulator provides an IOC Block to control input and output. For details, refer to Section 2.2.5.

2. This is the Link Table Transfer Block. For details, refer to Section 2.2.2.

2.2.2 LTT Block (Link Table Transfer)

The LTT Block receives an input from the IC Block and output the data to the Link Table(LT). This transfer is conditional in accordance with the internal handshake signals. Fig. 2-8 shows the structure of the LTT Block.

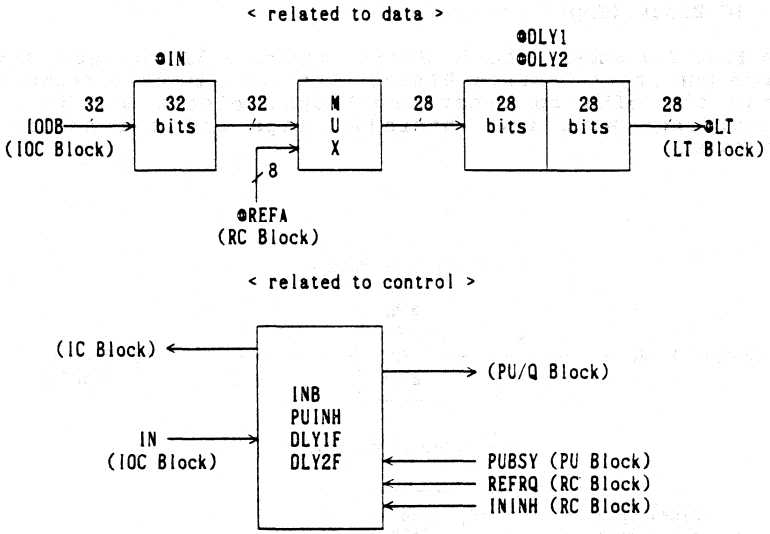


Fig. 2-8 Structure of the LTT Block

@IN	32-bit IN-Data Latch
INB	Empty/Full Flag of @IN
@DLY1, @DLY2	2 pipeline step delay circuit
DLY1F, DLY2F	Empty/Full Flag of @DLY1, @DLY2
MUX	Input Data multiplexer
PUINH	PU Block Inhibit Flag

2.2.3 OC Block (Output Controller)

The OC Block receives its usual inputs from the I/O Control block and the Output Queue(OQ) block. The other inputs may exist only if the uPD7281 encounters hardware error condition. Again, the data transfers are done in accordance with the internal flags and handshake lines. Fig. 2-9 shows the structure of the OC Block.

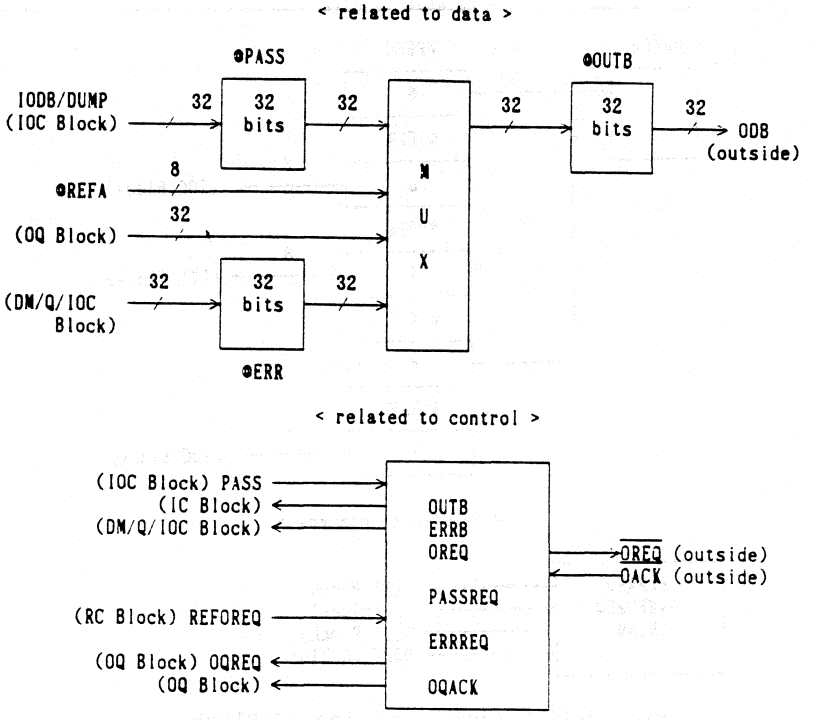


Fig. 2-9 Structure of the OC Block

@PASS ..... PASS/DUMP Data Output Latch (32 bits)
@ERR ..... ERR Data Output Latch (32 bits)
@OUTB ..... Output Data Latch (32 bits)
OUTB ..... Empty/full Flag of @OUTB
ERRB ..... Empty/full Flag of @ERR
OREQ ..... Data Output Request Flag
PASSREQ ..... PASS Data Output Request Flag
ERRREQ ..... ERR Data Output Request Flag
OQACK ..... OQ Block Data Acknowledge Flag

2.2.4 RC Block (Refresh Controller)

Fig. 2-10 shows the structure of the RC Block. Even though the simulator keeps track of refresh counter contents, it does not output refresh tokens to the Link Table. However, since the actual Refresh Controller hardware is totally transparent to the user, the software simulation of the Refresh Controller may not be necessary in most cases.

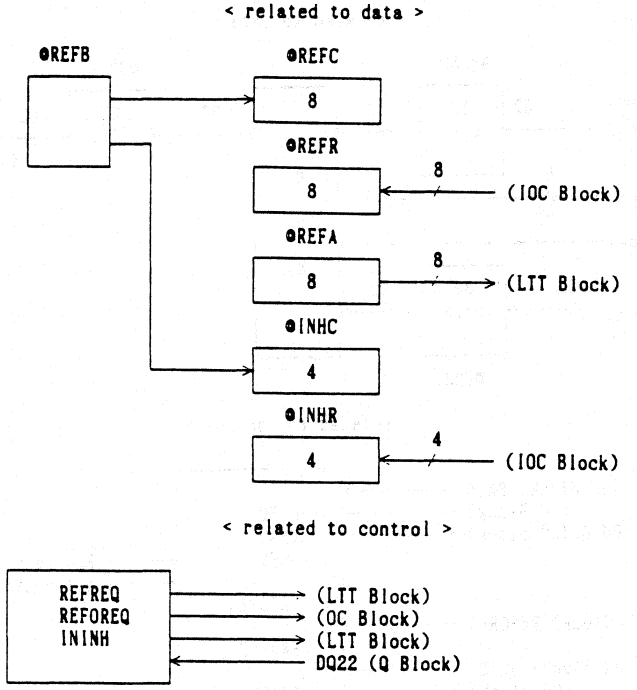


Fig. 2-10 Structure of the RC Block

@REFB	.....	3-bit Refresh Base Counter
@REFC	.....	8-bit Refresh Counter
@REFR	.....	8-bit Refresh Register
@REFA	.....	8-bit Refresh Address Register
@REFREQ	...	Refresh Data Output Request Flag
REFOREQ	...	External Refresh Data Output Request Flag
@INHC	.....	4-bit Input Inhibit Counter
@INHR	.....	4-bit Input Inhibit Register
ININH	.....	Input Inhibit Flag

2.2.5 IOC Block (Input/Output Controller)

This block is the one that actually controls the input and output of tokens to and from a uPD7281. It accepts a 32-bit token and makes a decision as to whether or not the token is to be routed to the Link Table or to be passed directly to the Output Controller Block. The IOC Block also accepts hardware operational mode related tokens such as SETMD and SETBRK tokens. Additionally, it keeps the module number for the uPD7281 that it is simulating for the case where DUMPD token needs to be output. Fig. 2-11 shows the structure of the IOC Block.

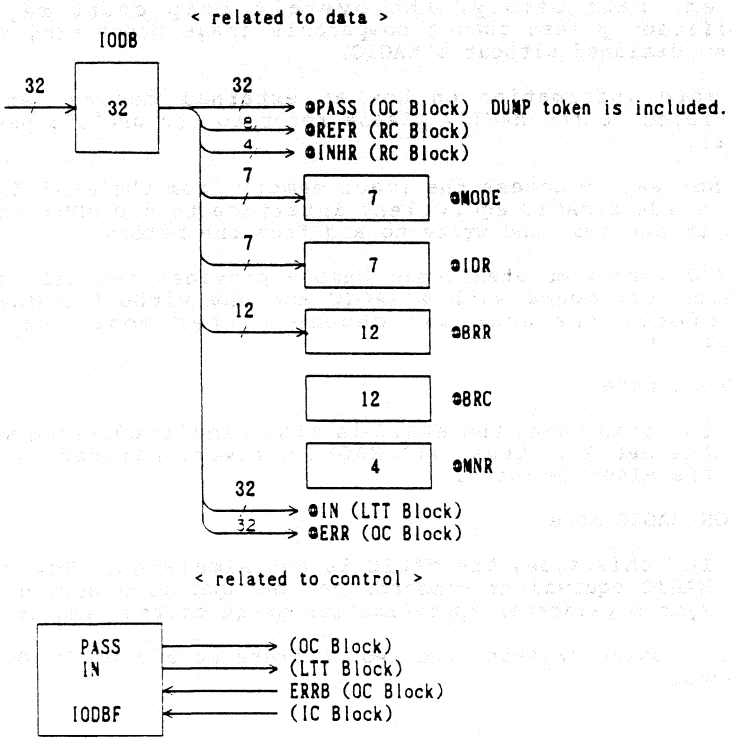


Fig. 2-11 Structure of the IOC Block

@MODE ..... Mode Register
@IDR ..... ID Register
@BRR ..... Break Register
@BRC ..... Break Counter
@MNR ..... Module Number Register

2.3 IM Block model

As mentioned before, the simulator is capable of simulating the external image memory which can be addressed from the uPD7281s. A typical way of accomplishing the external memory accesses from a UPD7281 is to use the companion chip called 'MAGIC' or uPD9305.

Utilizing the MAGIC, an image memory and a chain of uPD7281s can easily be interfaced to almost any 8-bit or 16-bit common microprocessor. Moreover, if an image processing sub-system is designed using a MAGIC, a chain of uPD7281s, and an image memory, the overall chip count may be significantly less than a comparable image processing sub-system designed without a MAGIC.

For more information on how an external memory can be interfaced to the MAGIC, please refer to the uPD9305 User's Manual.

Another way to access the image memory from the uPD7281 is to provide a MAGIC equivalent interface to the uPD7281 so that it can read and write to and from the memory.

The software simulated Image Memory provides two different models: one model with a MAGIC and one without a MAGIC. Therefore, the user may choose either model during simulation.

1) MAGIC mode

   In this mode, the MAGIC is also simulated along with the uPD7281, thus, all MAGIC hardware related tokens are also simulated.

2) NON-MAGIC mode

   In this mode, the MAGIC is not simulated. However, MAGIC equivalent commands can be specified during the system parameter specification stage of the simulation.

The following explains the two software models of IM Block in detail.

2.3.1 Model of IM Block in MAGIC mode

Fig. 2-12 shows the block diagram of IM Block when the MAGIC mode is chosen for software simulation of image memory accesses.

Initially, the IM Block is initialized to 0. When plane-1 is relocated, its 256 words are also initialized to 0.

NOTE: When this mode is used, the access method must be specified by MAGIC, RHASEL, and CSSEL subcommands in ENVIRON command (to be explained in Chapter 5).

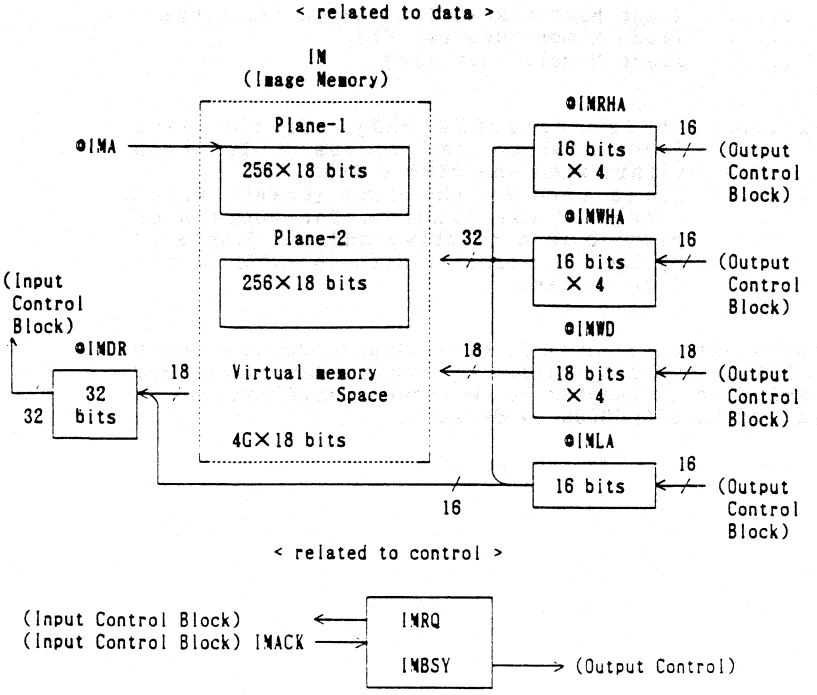


Fig. 2-12 IM Block model in MAGIC mode

IM	Image Memory (256 words X 18 bits X 2 planes). 4G words of virtual space can be mapped.
@IMA	Plane-1 Start Address Latch (32 bits)
@IMRHA	Image Memory Read High Address Latch (16 bits x 4)
@IMWHA	Image Memory Write High Address Latch (16 bits x 4)
@IMWD	Image Memory Write Data Latch (18 bits x 4)
@IMLA	Image Memory Read/Write Low Address Latch (18bits x 4)
@IMDR	Image Memory Read Data Latch (32 bits)
IMRQ	Image Memory Request Flag
IMBSY	Image Memory Busy Flag
Plane-1	It is available only for the data (token) which can access a location within @IMA and @IMA + 255.
Plane-2	It is used by the data (token) which cannot access plane-1. The address of plane-2 is a relative address displaced by the low-order 8 bits (0 255) of the data (token).

NOTE: The start address in Plane-1 Start Address Latch (@IMA) must be specified by Modify Memory command explained in Chapter 5. Without specification, it is set to 00000000H by default.

2.3.2 Model of IM Block in NON-MAGIC mode

Fig. 2-13 shows the block diagram of the IM Block used in the NON-MAGIC mode for image memory accesses.

NOTE; In this mode, the access method must be specified by WRITE, READ, ID, MN subcommands in ENVIRON command (explained in Chapter 5).

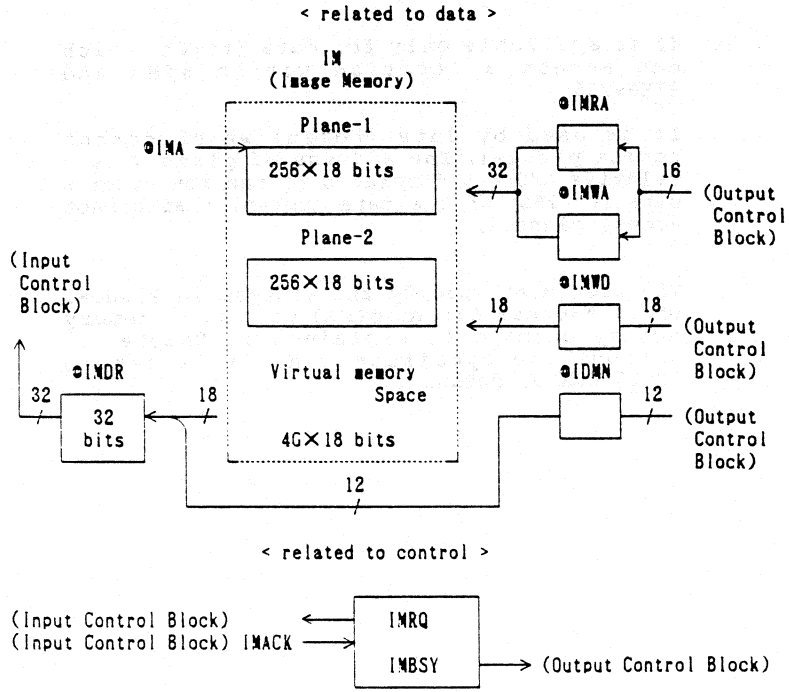


Fig. 2-13 IM Block model in NON-MAGIC mode

IM	Image Memory (256 words x 18 bits x 2 planes) 4G words of virtual space can be mapped.
@IMA	Plane-1 Start Address Latch (32 bits)
@IMRA	Image Memory Read High Address Latch (32 bits)
@IMWA	Image Memory Write High Address Latch (32 bits)
@IMWD	Image Memory Write Data Latch (18 bits)
@IDMN	Image Memory Read ID and MN Latch (18 bits)
@IMDR	Image Memory Read Data Latch (32 bits)
IMRQ	Image Memory Request Flag
IMBSY	Image Memory Busy Flag

Plane-1 .... It is available only for data (token) which can access a location within @IMA and @IMA+255

Plane-2 .... It is used by data (token) which cannot access plane-1. The address of plane-2 is a relative address displaced by the low-order 8 bits (0 255) of the data (token) that cannot access plane-1.

NOTE: The user must specify the address in Plane-1 Start Address Latch (@IMA) with the Memory Modify command as explained in Chapter 5. Without the specification, it is set to 00000000H by default.

CHAPTER 3 INPUT/OUTPUT FILE FORMAT

There are nine types of files handled by the simulator.

1) Object module file
2) Input data file
3) Macro file
4) CPU resource load/save file
5) Include file
6) List file
7) Output data file
8) Trace-log keeping file
9) Work file (for symbol table)

The function and usage of each different file is explained in the following sections. However, explanation on the work file is omitted, since it has been explained in Section 1.2.1.

3.1 File name

The following shows the format of file names (general form under the operating system).

(1) Case of disk files

```
-----------------------------------
      [d:] primary name [.ext]
-----------------------------------
```

d: ; Specifies the disk drive name. If omitted, the current default drive name is assumed.

primary name ; A string of up to 8 characters, specifying files name.

ext ; The extension on file name up to 3 characters.

(2) Case of non-disk files

```
-----------------
  device name:
-----------------
```

Console and printer can be specified as devices.

Console ..... CON:
Printer ..... LST:

3.2 Rules on creating files

The following rules should be observed when creating input data files and macro files.

3.2.1 Character code

(1) Use 7-bit ASCII code for character code.

(2) Delimit one line with a CR (carriage return) and a LF (line feed).

(3) The number of characters (including a CR and a LF) in one line must be 128 or less.

3.2.2 Number

(1) The number of pipeline steps is coded with 5 decimal digits or less.

(2) Input data for simulation is coded in hexadecimal digits without a radix suffix.

3.3 Input files

The simulator uses the following 5 files as input files. The file(s) used as an input during a simulation must already have been created properly and exist on a disk.

(1) Object module file
(2) Input data file
(3) Macro file
(4) Simulation resource load file
(5) Include file

3.3.1 Object module file

The object module file is output by the uPD7281 Assembler. It contains the object code and symbol information.

3.3.2 Input data file

This file holds the data tokens to be input to the uPD7281. After invoking the uPD7281 simulator, the input data file is specified in order to input tokens to the uPD7281 being simulated. The data tokens to be input to the uPD7281 are those tokens specified in the DATA section of program source file.

If an input data file is specified after loading the object file into the simulator, the simulator creates the input data file with an INM file name extension using the

information stored in the object file. The data tokens contained in the input data file are then input to the software uPD7281 model once the simulation is initiated.

For flexibility in system simulation, the simulator allows the input data file to be an output data file from other simulation. Using this feature, almost any large system configuration may be simulated using the uPD7281 software simulator.

The format in which the INM file must follow is shown below. The simulator generated INM file always conform to the format, and therefore, it may not require the user's attention. However, if the user is generating the INM file with an editor, the format of the INM file must meet the specifications of the following.

Format

```
I INM(CR)    (or OTM(CR))                  I
I                                          I
I datal  data2  [  n][  ;comment](CR)       I
I           .                              I
I           .                              I
I           .                              I
I END(CR)                                  I
```

INM (or OTM) ..... Indicates whether the following data tokens are for input (INM) to or output (OTM) from the simulator.

datal, data2 ..... The datal and data2 both forms an input data token to a uPD7281. The datal is the upper half, and the data2 is the lower half of a token. Each half of the token is specified with 4 hexadecimal digits. The input sequence of the data follows the order in which they are coded.

n ..... Represents the data input timing. It is coded with 5 decimal digits or less. The number 'n' is the pipeline clock cycle at which the token is to be input to the uPD7281. Specifying this parameter for each input data token, the user may has complete control over token input to a uPD7281. The data input timing can also be specified by the 'TIMING' command. If there is no need to specify the data input timing in advance, the 'n' parameter is omitted. There must be a space before and after 'n'.

END ..... Represents the end of input data for the simulation.

3.3.3 Macro file

This file contains the information on macro command definitions. Any convenient macro commands are defined by the user in this file. The macro command definitions stored in this file come into effect if the simulator loads in the file using 'LOAD MACRO' command in the simulator.

Format

```
-----------------------------
I MACRO   macro namel (CR) I
I command (CR)             I
I      .                   I
I      .                   I
I      .                   I
I END (CR)                 I
I      .                   I
I      .                   I
I      .                   I
I MACRO   macro namen (CR) I
I command (CR)             I
I      .                   I
I      .                   I
I      .                   I
I END (CR)                 I
I (EOF)                    I
-----------------------------
```

MACRO ..... Indicates that the file is a macro file and that it is the beginning of a macro command definition.

macro name ... It is the name of the macro.

command ..... One of the commands explained in Chapter 5 is coded.

END ..... Indicates the end of a macro command definition.

(EOF) ..... Represents the physical end of file mark. (The simulator or the operating system inserts it automatically.)

3.3.4 Simulation resource load file

This input file contains the information on the uPD7281 simulations. The purpose of this file is to be able to save all of the resources from a simulation, and then to re-load the simulation resources at a later in time to continue the original simulation. Thus, this file is not user creatable.

The saved resource file may be re-loaded into the simulator by 'LOAD RESOURCE' command to resume the simulation. 'SAVE RESOURCE' command is used to save the simulation resources on a disk file.

3.3.5 Include file

This file holds a sequence of commands for the Simulator. It can be referenced with the 'INCLUDE' command when the simulator is invoked. The sequence of commands is stored in the file line by line, and each line is read by the simulator and executed sequentially in the order of appearances.

The purpose of this file is to submit a batch of simulator commands in a file. It is quite useful when specifying system environments, specifying I/O file parameters, etc. prior to a program simulation using one command. Therefore, an 'INCLUDE' file with a typical system environment and parameters may be used over and over again for simulations.

The following is the format which an 'INCLUDE' file must follow.

Format

```
----------------
I command (CR) I
I    .         I
I    .         I
I    .         I
I (EOF)        I
----------------
```

command ..... One of the the commands explained in Chapter 5 is to be coded.

(EOF) ..... Represents the physical end of file marker. (The simulator or the operating system inserts it automatically.)

3.4 Output files

The simulator uses four types of files as output files.

(1) Macro file
(2) Simulation resource save file
(3) Output data file
(4) Trace-log file

3.4.1 Macro file

This file is output when macro definitions currently in effect are to be saved in a file. This can be accomplished by the use of 'PUT MACRO' command. For the file format, refer to Section 3.3.3, 'Macro file'.

3.4.2 Simulation resource save file

This file is output when the uPD7281 simulation resources are saved by the use of 'SAVE RESOURCE' command.

3.4.3 Output data file

This file is output by the simulator during a simulation. As the file format is the same as that of the input data file, refer to Section 3.3.2, 'Input data file'.

3.4.4 Trace-log keeping file

This is a binary file that keeps a trace-log of the simulation results. In order to keep the trace-log, either a block of memory or a disk file name must be specified with the 'MAP command'. (Refer to Section 5.5, 'Commands for Trace-log'.)

The trace-log can be displayed on the console by using the 'PRINT' command (refer to Section 5.5). Items to be displayed on the console must be specified using the 'DEFINE TRACE' command. (Refer to Section 5.11, 'Set/Display/Cancel Trace Item commands'.)

Table 3-1 shows trace-log items.

Table 3-1 Trace items

Trace item	meaning
Module Number	Module Number
CLOCK	Number of pipeline steps
@INB	Input Data Buffer
@IN	LT Transfer Latch
@LT	LT Input Latch
@FT	FT Input Latch
@DM	DM Input Latch
@Q	Q Input Latch
@OQ	OQ Input Latch
@PU	PU Input Latch
@PASS	Pass Data Latch
@OUTB	Output Data Buffer
@BRC	Break Counter
@INHC	Input Inhibit Counter
@REFC	Refresh Counter
DQSIZE	DQ size
GQSIZE	GQ size
OQSIZE	OQ size

NOTE: When more than one uPD7281 are software simulated, each trace item must be prefixed by the module number of the uPD7281. (EX. #8@LT)

3.4.6 Trace-log display format

The trace-log kept in the trace-log keeping file can be displayed on the console with the 'PRINT' command (refer to Section 5.5, 'Trace-log commands').

(1) For every pipeline clock cycle, the simulator displays all of the items to be traced in one horizontal line. Thus, any two traced items displayed in the same horizontal line are the simulation results that exist simultaneously at that pipeline clock cycle.

Since many trace items are to be displayed in a horizontal line, there may be many occasions where the trace items exceeds more than 132 character spaces. In such cases, the trace display is partitioned into a number of blocks, each of which takes 132 or less column spaces. If the partition falls on the middle of a trace item display, the item is displayed on the next partitioned block. The figure below illustrates the partitioning of trace-log display.

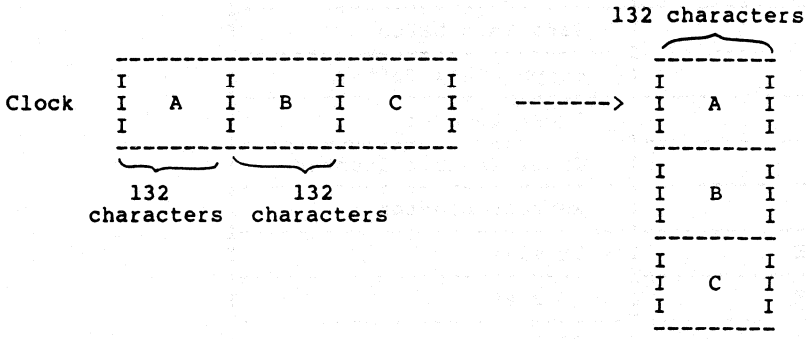


Fig. 3-1 and Fig. 3-2 show examples of the partitioned display format.

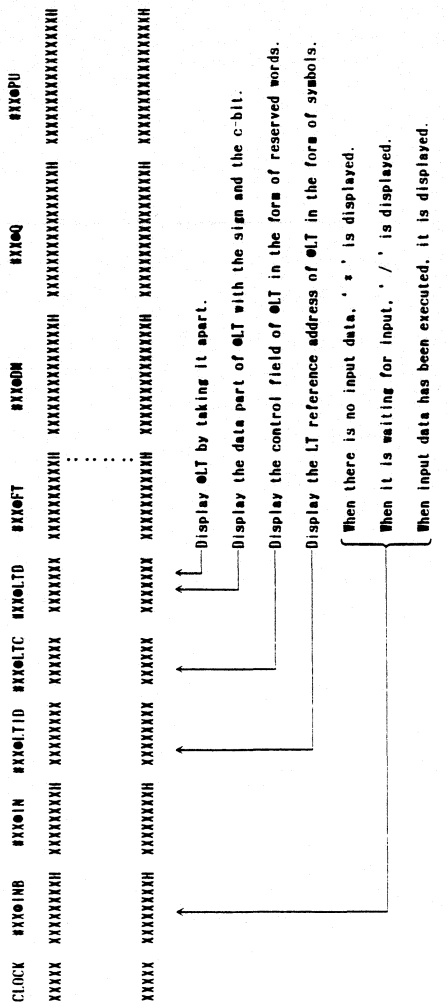


Fig. 3-1 Trace-log display format (1)

```
CLOCK  #XX●OQ             #XX●PASS   #XX●OUTB   #XX●BRC  #XX●INHC  #XX●REFC  #XXDQSZ  #XXGQSZ  #XXDQSZ

XXXXX  XXXXXXXXXXXXXXXXH  XXXXXXXXH  XXXXXXXXH  XXXXH    XXH       XXH       XXH      XXH      XXH
                                                          :
                                                          :
XXXXX  XXXXXXXXXXXXXXXXH  XXXXXXXXH  XXXXXXXXH  XXXXH    XXH       XXH       XXH      XXH      XXH
                                     ↑
                                     |   ┌ When there is no output data, ' * ' is displayed.
                                     └───┤ When it is waiting for output, ' / ' is displayed.
                                         └ When data has been output, it is displayed.
```

Fig. 3-2 Trace-log display format (2)

(2) The items to be displayed in the trace-log can be selectively specified using the 'DEFINE TRACE' command (refer to Section 5.11, 'Set/Display/Cancel Trace Item commands'). If the specification is omitted, the simulator displays @INB, @LT, @OUTB of current module.

Table 3-2 shows each trace item for trace-log display along with explanation of the item.

Table 3-2 Items of the trace display format

XX ; represents a module number (in decimal).

Display item	meaning
CLOCK	The pipeline clock cycle is displayed in 5 decimal digits.
#XX@INB	The data latched in the Input Buffer is displayed in 8 hexadecimal digits.
#XX@IN	The data latched in the LT Transfer Latch is displayed in 8 hexadecimal digits.
#XX@LTID	The ID part of the data latched in the LT Input Latch is displayed either as a symbol name if the symbol exists, or in 4 hexadeciamal digits if the symbol does not exist.
#XX@LTC	The control field of the data latched in the LT Input Latch is displayed as a reserved word.
#XX@LTD	The data part latched in the LT Input Latch is displayed in 4 hexadecimal digits. If the latched data is an EXEC data, it is displayed with the sign bit and the C-bit.
#XX@FT	The data latched in the FT Input Latch is displayed in 10 hexadecimal digits.
#XX@DM	The datalatched in the DM InputLatchis displayed in 14 hexadecimal digits.
#XX@Q	The data latched in the Q Input Latch is displayed in 16 hexadecimal digits.
#XX@PU	The data latched in the PU Input Latch is displayed in 16 hexadecimal digits.

(cont.)

Items	Meaning
#XX@OQ	The data latched in the OQ Input Latch is displayed in 16 hexadecimal digits.
#XX@PASS	The data latched in the Pass Data Latch is displayed in 8 hexadecimal digits.
#XX@OUTB	The data latched in the Output Buffer is displayed in 8 hexadecimal digits.
#XX@BRC	The Break Counter is displayed in 4 hexadecimal digits.
#XX@INHC	The Input Inhibit Counter is displayed in 2 hexadecimal digits.
#XX@REFC	The Refresh Counter is displayed in 2 hexadecimal digits.
#XX@DQSZ	The DQ size isdisplayed in 2 hexadecimal digits.
#XX@GQSZ	The GQ size is displayed in 2 hexadecimal digits.
#XX@OQSZ	The OQ size is displayed in 2 hexadecimal digits.

CHAPTER 4 BASIC USAGE OF THE SIMULATOR

4.1 Character set

Characters available for entering simulator commands or their operands are shown below.

(1) Alphabetic characters (upper-case and lower-case letters)

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

(2) Numeric digits

```
0 1 2 3 4 5 6 7 8 9
```

(3) Special characters

```
#  $  %  '  (  )  *  +  ,  -  .  /  :
;  ?  @  _  blank  !
```

(4) Control characters

```
DEL  CR  ESC  CTRL-X  CTRL-S  CTRL-Q
```

4.1.1 Use of special characters

Character	Name	Main use
#	sharp mark	Identification mark for module number.
$	dollar mark	This is ignored by the Simulator.
%	percent mark	Identification mark for macro formal parameter.
'	single quote	Beginning mark or end mark of character strings.
(.....)	left/right parenthesis	For change in precedence of operations.
*	asterisk	Multiply operator.
+	plus	Positive sign or add operator.
,	comma	Delimiter for operands.
-	minus	Negative sign or substract operator.
.	period	Identification mark for c-bit or symbol reference mark.
/	slash	Divide operator.
:	colon	Macro reference mark.
;	semicolon	Beginning mark for comments.

?	question mark	Alphabet-equivalent character.
@	unit-price mark	Alphabet-equivalent character.
_	underscore	Alphabet-equivalent character.
!	exclamation mark	Symbol reference mark.

4.1.2 Use of control characters

The use of control characters such as DEL, ESC, CTRL-X, CTRL-S, and CTRL-Q, is shown in Table 4-1.

Table 4-1 The use of control characters

Character	Meaning
DEL (rubout)	Deletes a character entered the last. (While entering commands)
CTRL-X	Deletes all characters that have been entered so far. (While entering commands)
CTRL-S	Suspends output to the console.
CTRL-Q	Resumes output to the console.
ESC (escape)	Breaks a process in execution and returns control to the command mode.

4.2 Expressions and Operators

There are five different types of operators used by the uPD7281 simulator. The operators in each category are listed below, and the following sections explain the uses of the operators.

1) Arithmetic operators
   +, -, *, /

2) Logical operators
   OR, AND, NOT, XOR

3) Shift operators
   SHR, SHL

4) Relational operators
   =, <>, <, <=, >, >=

5) Other operators
   (,)

The operators, such as logical operators and shift operators which use string of characters as operators, must be preceded and followed by blank spaces in order to distinguish them from symbols representing operands.

The general forms of both unary operators and binary operators are shown below.

General form of unary operators

```
--------------------
 operator  operand
--------------------
```

Examples:
```
-5
NOT SYM1   (SYM1 is a symbol.)
```

General form of binary operators

```
-------------------------------------------
 first operand  operator   second operand
-------------------------------------------
```

Examples:
```
0010H AND SYM1
SYM1 <= 01FFH
```

Operators and their order of precedence are explained below.

4.2.1 Order of precedence of operators

The order of precedence for operators is predetermined. An expression containing more than one operator is evaluated according in the order of precedence. In the case where operators of the same precedence exist within an expression, the evaluation proceeds from the left to the right.

Table 4-2 shows the order of precedence of operators. (The highest precedence is 1.)

Table 4-2 Order of precedence of operators

Precedence	Operator
1	(,)
2	NOT, + (unary operator), - (unary operator)
3	*, /, SHL, SHR
4	+ (binary operator), - (binary operator)
5	AND
6	OR, XOR
7	=, <>, <, <=, >, >=

4.2.2 Arithmetic operators

Using the arithmetic operators as binary operators, addition, subtraction, multiplication and division operations can be performed in expressions.

All arithmetic operations are performed on 17-bit data consisting of a sign bit and a 16-bit absolute value. The control bit (C-bit) of the operation's result is the same as that of the first operand.

(1) +

This operator performs an addition operation using the first operand and the second operand. The operator may also be used as an unary operator.

Examples: 2+3 (The result is 5.)
+2

(2) -

This operator performs a subtraction operation on the first operand using the second operand. When it is used as an unary operator, it returns a value with the sign bit inverted.

Examples: 3-2 (The result is 1.)
-1

(3) *

This operator multiplies the first operand by the second one.

Example: 3*2 (The result is 6.)

(4) /

This operator divides the first operand by the second one and returns the integer part of the result.

Example: 5/2 (The result is 2.)

4.2.3 Logical operators

A logical operator performs bitwise logical operations using the first and the second operands. Both the control bit (C-bit) and the sign bit (S-bit) of the result become the same as those of the first operand.

(1) OR

This operator performs bitwise logical "OR" operations using the two operands and returns the logical sum of the two operands.

Example: 1234H OR 5678H (The result is 567CH.)

(2) AND

This operator performs bitwise logical "AND" operations using the two operands and returns the logical product of the operands.

Example: 1234H AND 5678H (The result is 1230H.)

(3) NOT

This operator returns the logical negation of the operand.

Example: NOT 1234H (The result is 0EDCBH.)

(4) XOR

This operator performs bitwise logical "XOR" operations and returns the EXCLUSIVE-ORed sum of the two operands.

Example: 1234H XOR 5678H (The result is 444CH.)

4.2.4 Shift operators

A shift operator performs a shift operation on the bit pattern stored in the first operand. The number of bits to be shifted is contained in the second operand. Both the control bit (C-bit) and the sign bit (S-bit) remain the same as those of the first operand.

(1) SHR (SHIFT RIGHT)

This operator performs a shift right operation on the first operand using the number of shifts specified by the second operand. Bit positions at the MSB side corresponding to the number of shifts are set to zeros.

Example: 1234H SHR 4 (The result is 0123H.)

(2) SHL (SHIFT LEFT)

This operator performs a shift left operation on the first operand using the number of the shifts specified by the second operand. Bit positions at the LSB side

corresponding to the number of shifts are set to zeros.

Example: 1234H SHL 4 (The result is 2340H.)

4.2.5 Relational operators

These operators compare the first operand to the second operand. If the result of the comparison is true, it returns 0FFFFH, and if the result is false, it returns 0000H. Both the control bit (C-bit) and the sign bit (S-bit) of the result become 0.

(1) = (EQUAL)

If the first operand is equal to the second operand, 0FFFFH is returned. Otherwise, 0000H is returned.

Example: 1234H = 3456H (The result is 0000H.)

(2) <> (NOT EQUAL)

If the first operand is not equal to the second operand, 0FFFFH is returned. Otherwise, 0000H is returned.

Example: 1234H <> 3456H (The result is 0FFFFH.)

(3) < (LESS THAN)

If the first operand is less than the second operand, 0FFFFH is returned. Otherwise, 0000H is returned.

Example: 1234H < 5678H (The result is 0FFFFH.)

(4) <= (LESS THAN OR EQUAL)

If the first operand is less than or equal to the second operand, 0FFFFH is returned. Otherwise 0000H is returned.

Example: 1234H <= 5678H (The result is 0FFFFH.)

(5) > (GREATER THAN)

If the first operand is greater than the second operand, 0FFFFH is returned. Otherwise 0000H is returned.

Example: 1234H > 5678H (The result is 0000H.)

(6) >= (GREATER THAN OR EQUAL)

If the first operand is greater than or equal to the second operand, 0FFFFH is returned. Otherwise 0000H

is returned.

Example: 1234H >= 5678H (The result is 0000H.)

4.2.6 Other operators

(1) (,)

This operator has the highest precedence among all of the operators. It causes the expression between '(' and ')' to be evaluated first. When '()'s are nested, evaluation proceeds from the innermost '()' to the outermost '()' in an expression.

Example: '2+3+4*4+3' equals '24', but '(2+3+4)*(4+3)' equals '63'.

(((1+2)*3)+5)/7 (The result is 2.)

4.3 The Simulator Operation Sequence

There are four different simulator operations that need to be provided in sequence by the user. These operations in sequence are shown in the following Fig. 4-1.

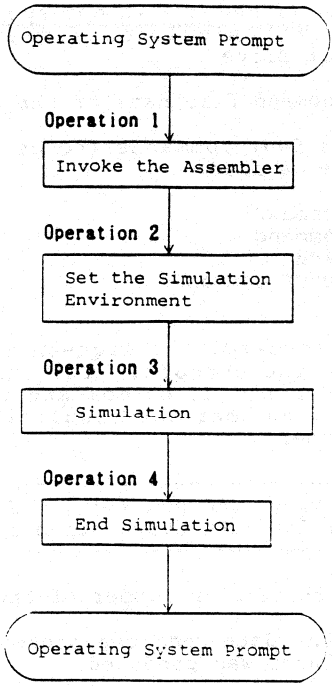


Fig. 4-1 Sequence of operations

Each operation starting from operation-1 to operation-4 are explained below.

(Operation 1) Invoke the simulator

To invoke the simulator, the following must be entered after the operating system prompt sign.

```
---------------------------------
  A> [d:]SM7281 [ command] (CR)
---------------------------------
```

A> ; This is the operating system prompt sign.

d: ; This is the disk drive that contains the SM7281.CMD file plus all overlay files (SM7281.OMn). If the 'd:' is not specified, it is assumed that all simulator files are on the current default dirve.

SM7281 ; This is the command file name of the simulator.

command ; The following four commands (to be explained in Chapter 5) may be used.

1) ENVIRON command
2) INCLUDE command
3) LIST command
4) LOAD RESOURCE command

When the simulator is invoked, the sign-on message shown below is displayed on the console. Then the specified command, if any, is executed. If no command is specified, '*' is displayed indicating that the simulator is waiting for command input from user.

```
-----------------------------------------
I UPD7281 SIMULATOR Vx.y  [dd Mmm YY]     I
I      Copyright (C) 1984 NEC Corporation I
-----------------------------------------
```

Vx.y ; This is the version number of the Simulator.

dd Mmm YY ; This is the date when the present version of the simulator was produced.

(Operation 2) Setting the simulation environment

Before a simulation can start, the hardware system environment to be simulated must be specified. For more information on how to set the environment, please see Chapter 5.

(Operation 3) Start the simulation

After setting the system simulation environment, the system hardware may be software simulated using many simulation commands available. For more detail, refer to Chapter 5.

(Operation 4) Terminating the simulation

To terminate simulation, enter the EXIT command. The control will then be returned to the operating system.

```
---------------
I      .       I
I      .       I
I      .       I
I  *EXIT (CR)  I
I  A>          I
---------------
```

4.4 The Simulator Command Format

After invoking the uPD7281 simulator, a chain of simulator commands may be input for controlling the software simulator. The following shows the input format for the simulator commands and explains certain rules to be followed.

In general, commands are entered in the following format.

```
-----------------------------------------------
*command_name  operand_list  [;comment] (CR)
-----------------------------------------------
```

(1) '*' is the command input request mark (prompt) from the simulator.

(2) command name
There are numerous commands available for the simulator. For the available command names, refer to Section 4.5 or Chapter 5.

(3) Operand list
There are five types of data which can be used in the operand list.

- o reserved words
- o symbols
- o numeric constants
- o expressions
- o character strings, character constants.

1) Reserved words

There are many words reserved for the simulator which can not be used as a user symbol. For reserved words, refer to Appendix 1, 'List of Reserved Words'.

2) Symbols

The symbols used in the simulator may come directly from the assembler generated object file. In this case, all symbols defined and used in the source program file may also be used in the simulator. Additionally, the user may define symbols within the simulator for symbolic debugging purposes.

3) Numeric constants

Numeric constants may be binary constants, octal constants, decimal constans or hexadecimal constants.

a) Binary constants

Binary constants are composed of binary digits with a suffix 'B'.

Examples:

```
011B .....    Represents the decimal number 3.
-101B .....   Represents the decimal number -5.
```

b) Octal constants

Octal constants are composed of octal digits with a suffix 'O'.

Examples:

```
037O   ......  Represents the decimal number 31.
-11O  .......  Represents the decimal number -9.
```

c) Decimal constants

Decimal constants are composed of decimal digits with a suffix 'D'. If no radix suffix is attached to a numeric constants, it is assumed to be a decimal number.

Examples:

```
123D
      .......  Both represent the decimal number
0123           123.

-86   .......  Represents the decimal number -86.
```

d) Hexadecimal constants

Hexadecimal constants are composed of hexadecimal digits with a suffix 'H'. Additionally, if a hexadecimal constant begins with a hexadecimal digit within 'A F', a prefix '0' must be placed in front of the number.

Examples:

13H ..... Represents the decimal number 19.
0FFH ..... Represents the decimal number 255.

When the C-bit (control bit) is to be added to a numeric constant, the option '.C' must be appended at the end of the numeric constant. If the option '.C' is omitted, C-bit is assumed to be 0.

4) Expressions

An expression is a collection of constants, symbols and operators. The order in which these items may appear is explained in Section 4.2.

5) Character strings, character constants

A string of characters used for character constants should be enclosed by a pair of apostrophes. If an apostrophe itself is to be used as a character, it must be coded as two consecutive apostrophes.

Examples:

'FUNCTION' ... FUNCTION is a character string.
'A''' ..... A' is a character string.

(4) comment

Any string of characters inserted between a ';' (semicolon) after a command and a 'CR' on a command line is regarded as a comment.

4.5 A Set of Simulator Commands

The commands available for the simulator are categorized according to their functions.

Command set

Set/Display Simulation Environment commands

```
RESET           ; External reset.  It closes opened files.
ENVIRON         ; Sets simulation environment.
STATUS          ; Displays simulation environment.
```

Set/Display Input Timing commands

```
TIMING=timing specification
                ; Set input timing.
TIMING          ; Displays input timing.
```

Load Object command

```
LOAD OBJECT     ; Load an object module.
```

Simulation Input/Output Data File Specification commands

```
DATA INM        ; Specifies input data file.
DATA OTM        ; Specifies output data file.
```

Trace-log commands

```
MAP             ; Sets trace-log area.
PRINT           ; Displays trace-log.
```

Host Data Input command

```
HOST            ; Inputs  host data.
```

Simulation execution commands

```
GO              ;  Starts simulation.
STEP            ;  Simulates 1 pipeline step.
CONTINUE        ;  Resumes a suspended simulation.
```

Set/Display/Cancel GR Break Condition commands

```
GR=GR break condition
                ;  Sets GR break condition.
GR              ;  Displays GR break condition.
NOGR            ;  Cancels GR break condition.
```

Set/Display/Cancel Conditional Breakpoint Register commands

```
  BRn=break condition
                  ; Sets a break condition for the
                    Conditional Breakpoint Register.
  BR[n]           ; Displays the content of the Conditional
                    Breakpoint Register.
  NOBR[n]         ; Cancels a break condition of the
                    Conditional Breakpoint Register.
```

Set/Display/Cancel Unconditional Breakpoint Register commands

```
  BPn=break condition
                  ; Sets a break condition for the
                    Unconditional Breakpoint Register.
  BP[n]           ; Displays the content of the Unconditional
                    Breakpoint Register.
  NOBP[n]         ; Cancels a break condition of the
                    Unconditional Breakpoint Register.
```

Set/Display/Cancel Trace Item commands

```
  DEFINE TRACE    ; Sets trace items.
  TRACE           ; Displays trace items.
  NOTRACE         ; Cancels trace items.
  TON=trace condition
                  ; Sets the trace start condition.
  TON             ; Displays the trace start condition.
  NOTON           ; Cancels the trace start condition.
  TOFF=trace condition
                  ; Sets the trace stop condition.
  TOFF            ; Displays the trace stop condition.
  NOTOFF          ; Cancels the trace stop condition.
```

Symbolic debug commands

```
  CALCULATE       ; Evaluates an expression and displays its
                    value.
  DEFINE SYMBOL   ; Defines a symbol.
  NOSYMBOL        ; Deletes a symbol.
  .symbol=        ; Modifies the value of a symbol.
  address expression=
                  ; Modifies the contents of memory, latch,
                    or register.
  SYMBOLS         ; Displays the contents of the symbol
                    table.
  .symbol         ; Displays the value of a symbol.
  address expression
                  ; Displays the content of memory, latch,
                    or register.
```

Assemble commands

ASSEMBLE	; Interprets assembler codes and stores them on specified memory (or file).
DISASSEMBLE	; Displays memory contents of a specified file in a symbolic format.

Macro function commands

REPEAT	; Processes a sequence of commands repeatedly.
COUNT	; Processes a sequence of commands repeatedly for a specified number of times.
IF	; Evaluates a condition and processes a sequence of commands.
MACRO	; Macro definition, reference, display, etc. of commands.
INCLUDE	; Inputs a sequence of commands from a file and executes it.
WRITE	; Evaluates a specified operand and displays it.
ECHO	; Displays a sequence of commands at macro execution.
NOECHO	; Sequence of commands is not displayed at macro execution.

Display Pipeline Step command

CLOCK	; Displays the content of the Pipeline Step Counter.

Set/Display Current Module Number commands

#=	; Sets the current module number.
#	; Displays the current module number.

Evaluation commands

LTEVAL	; Displays the LT's operating ratio.
PUEVAL	; Displays the PU's operating ratio.
IMEVAL	; Displays the IM's operating ratio.
NOEVAL	; Clears the values of all operating ratio counters.

Load/Save Simulation Resource commands

LOAD RESOURCE	; Loads simulation resource information from a file.
SAVE RESOURCE	; Saves simulation resource information into a file.

Set/Cancel Listing Function commands

```
  LIST          ; Sets the output data (that is output to
                  the console) to also be output to a
                  specified file.
  NOLIST        ; Cancels the listing function.
```

Dump ASCII-file command

```
  TYPE          ; Displays the content of an ASCII file.
```

End Simulation command

```
  EXIT          ; Ends simulation.
```

CHAPTER 5 THE SIMULATOR COMMANDS

In this chapter, all available simulator commands are explained in detail along with their format.

5.1 Set/Display Simulation Environment commands

There are three types of Set/Display Simulation Environment commands as follows.

1) RESET (Reset command)
2) ENVIRON (Set Simulation Environment command)
3) STATUS (Display Simulation Environment command)

5.1.1 RESET (Reset) command

(1) Input format

```
RESET (CR)
```

(2) Function

This command performs an external reset operation on the software simulator and closes all opened files.

(3) Description

The RESET command resets the uPD7281 simulation resources. When this command is entered, all of uPD7281 simulation resources become ineffective and all files currently being used by the simulator are closed.

(4) Application Example

```
*RESET (CR)
*
```

5.1.2 ENVIRON (Set Simulation Environment command)

(1) Input format

```
ENVIRON (CR)
 MODULE  n (CR)                                          .....1)
 MN  n[,...] (CR)                                        .....2)
 [ [ WRITE  { HIGH | LOW | DATA [ { COMMAND | CMD } ] }  expression (CR)
   [ READ   { HIGH | LOW | [ { COMMAND | CMD } ] }       expression (CR)
   [ MN FIELD expression LENGTH expression
            OFFSET expression PADDING
            expression (CR)                              .....3)
   [ ID  FIELD  expression  LENGTH  expression
            OFFSET expression PADDING
            expression (CR)
   [ DELAY   expression (CR)
   [ MAGIC (CR)
   [ RHASEL  expression (CR)
   [ CSSEL   expression (CR)
   [ DELAY   expression (CR)
   [ PRIORITY { HOST | INM | IM | PASS } [,...] (CR)     .....4)
 [  WORK  [d:] [file name]]                              .....5)
 END (CR)
```

(2) Function

This command sets the simulation environment.

(3) Description

The ENVIRON command sets the hardware system environment to be simulated by the software simulator before the simulation. Since a simulation cannot proceed without its system environment defined, this is a required statement for a simulation.

As the command 'ENVIRON (CR)' is entered, '**' will be displayed. Subsequently, the simulation environment conditions are set in sequence according to the command input format.

The command is completed by entering 'END (CR)' at the end of ENVIRON commands. All subcommands available within the ENVIRON command are explained below in the order of their appearance in the input format (as indicated by the numbers appended in the input format).

1) MODULE

This subcommand sets the number of uPD7281 modules to be simulated by the simulator. The number 'n' must be within 1 and 14.
(1 ≤ n (number of modules) ≤ 14)

Example:

**MODULE 3(CR) <----- 3 modules will be simulated.

2) MN

This subcommand declares the module numbers of the uPD7281s to be simulated. Since different module numbers must be assigned for each module, the total of module numbers declared must be equal to the number specified by the MODULE subcommand in 1). The number 'n' must be within 1 and 14. (This restriction for module numbers also applies to the following commands.)
(1 ≤ expression (module number) ≤ 14)

Example:

**MN 3,14,2(CR) <----- Modules of numbers 3, 14, and 2 are to be simulated.

3) [WRITE, READ, MN, ID, DELAY] for non-MAGIC mode
[MAGIC, RHASEL, CSSEL, DELAY] for MAGIC mode

These subcommands are used to simulate the image memory accesses by the uPD7281s. When a system model with an Image Memory is to be simulated, one of the following two sets of subcommands must be used. Regardless of which mode is used, all subcommands in a set must be entered with parameters.

Simulation of a system with an Image Memory using the MAGIC mode uses

MAGIC, RHASEL, CSSEL, DELAY.

Simulation of a system with an Image Memory using the NON-MAGIC mode uses

WRITE, READ, MN, ID, DELAY.

NON-MAGIC mode

For non-MAGIC mode, the simulator model assumes that the user has interface hardware between the uPD7281s and the image memory. The hardware configuration of this image memory interface is shown in Fig. 2-13.

This interface hardware is functionally similar to the MAGIC, since it must provide image memory access capability from a uPD7281 and also the token passing capability. The ways in which IMREAD, IMWRITE and PASS tokens are handled are very similar to the actual MAGIC implementation. Thus, for a better understanding of how to use simulator in the non-MAGIC mode, please refer to the uPD9305 User's Manual.

The following subcommands are for the non-MAGIC mode of image access only. Please note that an entirely different set of subcommands must be specified for the MAGIC mode. Since there are no default values available for non-MAGIC mode, all sub-commands along with appropriate parameters must be specified.

The following are explanations of the sub-commands that are used to define the token formats for image memory accesses. The tokens coming out of the Output Control Block of the simulator model are examined to determine their destinations. If token is related with Image Memory operations, it is directed to the IM Block of the simulator model.

a. WRITE HIGH

This subcommand is used to specify the bit pattern for the upper half of a token to be recognized as an Image Memory Address Latch (@IMWA) set token. The lower 16-bit half of the same token is used to set the upper 16-bit half of the 32-bit @IMWA.

In the following example, X designates the "don't care" logic value. Thus, if the bit pattern of the upper half of a token matches the specified WRITE HIGH bit pattern, the token will be recognized as a @IMWA upper half set token.

Example:

**WRITE HIGH XXXX0100XXXXXXXXB (CR)

b. WRITE LOW

This subcommand is used to specify the bit pattern for the upper half of a token to be also recognized as an Image Memory Address Latch (@IMWA) set token. However, this token is used to set the lower half of the 32-bit @IMWA. Thus, if the bit pattern of the upper half of a token matches the specified WRITE LOW bit pattern, the token will be recognized as a @IMWA lower half set token.

Example:

**WRITE LOW XXXX0001XXXXXXXXB (CR)

c. WRITE DATA

This subcommand is used to speicfy the bit pattern for the upper half of a token to be recognized as the Image Memory Write Data latch (@IMWD) set token. The lower half of the token contains the data to be set in the @IMWD latch. The last two bits in the upper half of the token contain the contol-bit and the sign-bit of the data. @IMWD latch is 18-bit wide.

Example:

**WRITE DATA XXXX0110XXXXXXXXB (CR)

d. WRITE COMMAND

This subcommand is used to specify the bit pattern for the upper half of a token to be recognized as the actual write command. When this token is received by the IM Block, the data stored in the @IMWD is written into the image memory location pointed to by the 32-bit address in the @IMWA.

Example:

**WRITE COMMAND XXXX0011XXXXXXXXB (CR)

e. READ HIGH

This subcommand is used to specify the bit pattern for the upper half of a token to be recognized as an Image Memory Read Address (@IMRA) latch set token. The lower 16-bit half of the same token is used to set the upper 16-bit half of the @IMRA latch which is 32-bit wide.

Example:

**READ HIGH XXXX1100XXXXXXXXB (CR)

f. READ LOW

This subcommand is used to specify the bit pattern for the upper half of a token to also be recognized as a @IMRA latch set token. However, the lower 16-bit half of the token is used to set the lower 16-bit half the @IMRA latch.

Example:

** READ LOW XXXX1001XXXXXXXXB (CR)

g. READ COMMAND

This subcommand is used to specify the bit pattern for the upper half of a token to be recognized as the actual read command. If a token with its upper half token's bit pattern is equal to the bit pattern specified by the READ COMMAND, then the simulator performs a read operation using the 32-bit image memory location specified by the @IMRA latch. The data read from the image memory location is placed on the lower half of the READ COMMAND token. The control-bit and the sign-bit are also included in the READ COMMAND token in its C and S field of the token.

Example:

** READ COMMAND XXXX1011XXXXXXXXB (CR)

h. MN

This MN subcommand is used for image memory READ COMMAND token only. After a READ COMMAND token performs an image memory read operation, the token containing the READ COMMAND token must be assigned a new Module Number and a new ID in order to direct the token to a proper ImPP accessing a proper Link Table location.

The MN command fills exactly that function but only for the MN field of a token. The ID field of the token exiting from the IM Block will be specified by the following ID subcommand.

The MN subcommand has four parameters to be speified. They are FIELD, LENGTH, OFFSET, and PADDING specifications.

For the explanation of the four parameters, the

example shown in Fig. 5-1 will be used. In this example, the first token is the READ COMMAND token which enters the IM Block, and the second token is the token which exits the IM Block after an image memory read operation. The MN specifications for this particular example is

Example:

**MN FIELD 5 LENGTH 3 OFFSET 13 PADDING 0000B (CR)

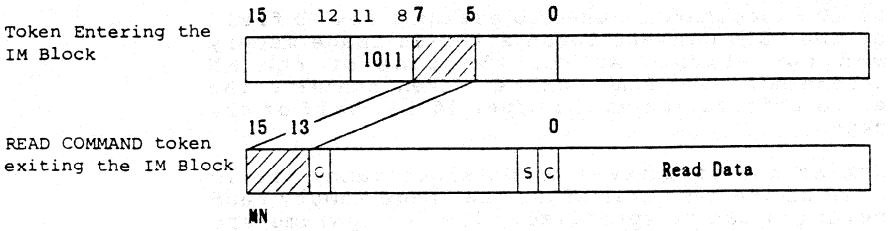


Fig. 5-1

MN subcommand specifies which four bits in the READ COMMAND token are to be used for the Module Number for the token exiting the IM Block.

The Field parameter specifies a bit location in the upper 16-bit half of the READ COMMAND token. The bit points to the lowest bit position of the field that contains the new Module Number. This new Module Number will be used as a Module Number when the READ COMMAND token exits the IM Block.

The LENGTH parameter specifies the length of the Module Number field in the READ COMMAND token. Since the length of the Module Number field for the uPD7281 is four, this LENGTH specification should not exceed four.

The OFFSET parameter specifies the location where the new MN field in the READ COMMAND token is placed when a new token is assembled and output after the image memory read operation. The OFFSET parameter points to the lowest bit location of the MN field in the new assembled token.

The PADDING parameter value specified takes care of the instances where the LENGTH parameter specified for the MN FIELD is less than four. In this case, the unspecified bits in the new token's MN field will be filled in by the 4-bit value under the PADDING parameter. In the example, the last bit in the MN field of the new token has been filled in with a zero using the PADDING parameter value, since the bit was not specified by the READ COMMAND token.

i. ID

The ID subcommand is used to assign a new ID field for the READ COMMAND token after an image memory read operation. As for the case of the MN subcommand, the READ COMMAND token contains the new ID information on the upper 16-bit half of the token.

Just as the parameters of MN subcommand, the ID field of the new token after the image memory read operation can be specified. The four parameters for the ID subcommand are FIELD, LENGTH, OFFSET and PADDING. The function of each parameter is the same as for the case of the MN subcommand, except that the length of the ID field can be seven bits long.

The following example illustrates the use of ID subcommand. In this example, the first token is the READ COMMAND token which enters the IM Block, and the second token is the token which exits the IM Block after an image memory read operation. The ID specifications for this particular example is

Example:

**ID FIELD 13 LENGTH 3 OFFSET 7 PADDING 1000010B (CR)

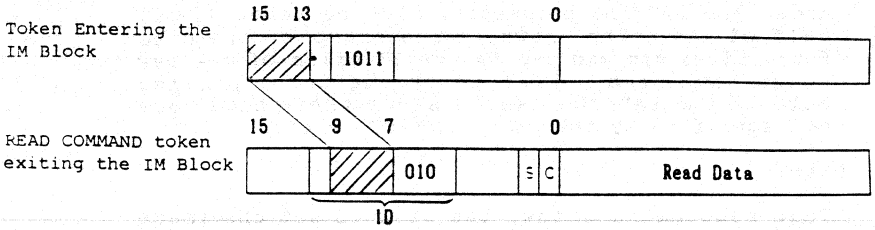


Fig. 5-2

The ID subcommand specifies which seven bits in the READ COMMAND token are to be used as a new ID value for the token exiting the IM Block.

The FIELD parameter specifies a bit location in the upper 16-bit half of the READ COMMAND token. The bit points to the lowest bit position of the field that contains the new ID field value. This value will be used as a new ID field of the token that exits the IM Block after a memory read operation.

The LENGTH parameter specifies the length of the new ID field in the READ COMMAND token. Since the length of the ID field for the uPD7281 is seven, this LENGTH specification should not exceed seven. However, the LENGTH can be less than seven bits.

The OFFSET parameter specifies the bit location where the new ID field in the READ COMMAND token is placed when a new token is assembled and output after the image memory read operation. The OFFSET points to the lowest bit location of the ID field in the new assembled token.

The PADDING parameter value takes care of the instances where the LENGTH specified for the ID FIELD is less than seven. In this case, the unspecified bits in the new token's ID field will be filled in by the corresponding bits specified

under the PADDING parameter. In the upper 16-bit half of the token shown in the example, the bits four, five, six and ten in the ID field of the new token have been filled in by the corresponding bits in the PADDING value, since these bits were not specified by the READ COMMAND token.

j. DELAY

This subcommand allows the user to set the image memory read cycle time. This memory read cycle time is the time interval between the time a READ COMMAND token enters the IM Block and the time a token with an image memory data exits the IM Block. The time delay between the two tokens is specified in terms of pipeline clock cycles. The delay value should be within a specified range.

$(0 \leq \text{delay value} \leq 65535)$

Example:

**DELAY 3 (CR)

MAGIC mode

Since MAGIC user's manual provides extensive coverage on the use of the uPD9305, explanations about image memory accesses using MAGIC will be minimal here. However, for a better understanding of the simulator model, the user should refer to Fig. 2-2, which shows the block diagram of the simulator IM block model of MAGIC mode.

a. MAGIC

This subcommand specifies that all image memory accesses are to be executed through MAGIC.

b. RHASEL

This subcommand specifies the state of the RHASEL bit in the mode register of the MAGIC. Either 0 or 1 may be specified.

Examples:

**RHASEL 0 (CR)
or
**RHASEL 1 (CR)

c. CSSEL

This subcommand specifies the state of the CSSEL

bit in the mode register of the MAGIC. Either 0 or 1 may be specified.

Examples:

**CSSEL 0 (CR)
or
**CSSEL 1 (CR)

e. DELAY

This subcommand is the same as that in NON-MAGIC mode. Refer to 'DELAY' mentioned above.

4) PRIORITY

This subcommand specifies the priorities among the input tokens from the HOST, input tokens from INM file, the token from image memory access, and the PASS token. In some instances, all four types of tokens may try to access the Input Control Block of the simulator. If there exists any timing conflict among the four type of tokens, the order in which the tokens may enter the Input Control Block is determined by the assigned priority. The priorities among the four different types of tokens are assigned by listing the token types in the decending order of priority.

Example:

**PRIORITY HOST, INM, IM, PASS (CR)

In the above example, the tokens from the HOST CPU have the highest priority.

5) WORK

This subcommand specifies the drive name, primary name, and file name extension for the simulator work file. The work files are used temporarily to store the symbols used for simulation.

If this subcommand is omitted, the simulator creates work file on the default disk drive with default file name. The default file name assigned for the work file is 'SM7281.$$0'.

If only the drive name is specified, a work file with the default file name will be created on the disk drive specified.

Examples:

1) Only the drive name is specified

```
**WORK A: (CR)      ; The  work file SM7281.$$0 is
                      created on disk drive A:.
```

2) Only the file name is specified

```
**WORK WORKF.SYM(CR) ;The work file WORKF.SYM is
                      created on current default
                      disk drive.
```

(4) Application example

```
*ENVIRON (CR)
**MODULE 1 (CR)
**MN 8 (CR)
**WRITE HIGH         01010001XXXXXXXXB (CR)
**WRITE LOW          01010000XXXXXXXXB (CR)
**WRITE DATA         01010010XXXXXXXXB (CR)
**WRITE COMMAND      01010000XXXXXXXXB (CR)
**READ  HIGH         01000111XXXXXXXXB (CR)
**READ  LOW          010000XXXXXXXXXXB (CR)
**READ  COMMAND      010000XXXXXXXXXXB (CR)
**MN FIELD 8 LENGTH 3 OFFSET 12 PADDING 100B (CR)
**ID FIELD 4 LENGTH 4 OFFSET  4 PADDING 0000000B (CR)
**DELAY 2 (CR)
**WORK B: (CR)
**END (CR)
*
```

5.1.3 STATUS (Display Simulation Environment command)

(1) Input format

```
STATUS (CR)
```

(2) Function

This command displays the simulation environment.

(3) Description

The STATUS command displays all specified input and output file names and the environment conditions set by the ENVIRON command.

(4) Application example

```
*STATUS (CR)
INM     =B:AFFIN4.INM  ┐
OTM     =B:AFFIN4.OTM  │
OBJECT  =A:AFFIN4.LNK  ├     file name
LIST    =LST:          │
MAP     =B:AFFIN4.MAP  │
WORK    =B:SM7281.$$0  │
MACRO   =              ┘
MODULE 01H    <------- number of modules (hexadecimal number)
MN 0BH        <------- module number (hexadecimal number)
WRITE HIGH     01010001********B ┐
WRITE LOW      01010000********B │
WRITE DATA     01010010********B ├ mask patterns (16 binary
WRITE COMMAND  01010000********B │  digits)
READ HIGH      01000111********B │
READ LOW       010000**********B │
READ COMMAND   010000**********B ┘
MN FIELD 8 LENGTH 3 OFFSET 12 PADDING 1000B
ID FIELD 4 LENGTH 4 OFFSET  4 PADDING 0000000B
PRIORITY HOST, INM, IM,PASS
*
```

5.2 Set/Display Input Timing commands

There are two Set/Display Input Timing commands. One command is used to set the timing pattern for the input tokens going into the uPD7281. The other command is used to display the current timing pattern.

1) TIMING = timing specification (Set Input Timing command)

2) TIMING (Display Input Timing command)

5.2.1 TIMING = timing specification (Set Input Timing command)

(1) Input format

```
TIMING = [expression-2] (expression-1 [  ...]) [  ...] (CR)
```

(2) Function

This command sets the timing pattern for the input data tokens entering the uPD7281.

(3) Description

In order to control the token flow into the uPD7281, the TIMING command is used to specify the timing pattern in terms of ImPP pipeline clock cycles. The timing pattern specified determines when an output token from Input Control Block may be input to the uPD7281 block in the software simulation model described in Fig. 2-1.

Expression-1 specifies the timing intervals at which tokens may be input to the uPD7281 in terms of pipeline clock. Expression-2 specifies the number of repetitions of timing operation specified in expression-1. Multiple expression-1 specifications must be separated by blank spaces.

The timing specifications given by this command are effective only when the input data tokens have no timing specifications attached to them. If the input timing is not specified, the default timing interval is every 2 pipeline clock cycles.

(4) Application examples

```
*TIMING=10(1 2 3 4) (CR) <----- Data tokens are input to the
                                uPD7281 at intervals of 1,
                                2, 3 and 4 pipeline clock,
                                and the input pattern is
                                repeated for 10 times (total
                                of 40 input data tokens).
                                After the 40 tokens, the
                                data tokens are input at
                                every 2 pipeline clock
                                intervals.

*TIMING=0AH(SYM1-3) (CR) <--┐
                            │
*TIMING=5(1 3 7) (CR)       │
                            │
                            └-- The symbol 'SYM1' is used in
                                this example.
```

5.2.2 TIMING (Display Input Timing command)

(1) Input format

TIMING (CR)

---

(2) Function

This command displays the current timing specifications for input tokens entering the uPD7281.

(3) Description

The TIMING command displays the input timing specified by input timing set command.

(4) Application examples

```
*TIMING (CR)
10(1 2 3 4)
*TIMING=5(1 3 7) (CR)  <-----  Timing is modified.
*TIMING (CR)
5(1 3 7)
*
```

5.3 Load Object Module command

LOAD OBJECT (Load Object Module command) is used to load all necessary resources for a simulation. The resources loaded include an object file created by the assembler, all symbols used in the source program file, and the INM extension file which contains tokens in the START section of source program file.

5.3.1 LOAD OBJECT (Load Object Module command)

(1) Input format

```
LOAD {OBJECT} file name (CR)
     {OBJ   }
[{ {ALLMODULE}          }]
 { {ALLM     } (CR)     }                                    .....1)
 { MODULE module name[,...] (CR) }]
[{ {ALLSYMBOL}      [{LT}]      }]
 { {ALLS     }      [{FT}] (CR) }                            .....2)
 { {SYMBOL   } [{LT}] [{DM}]     }
 { {SYM      } [{FT}] module name[,...] (CR) }]
 { [{DM}]                        }
[DATA[ file name [ APPEND]] (CR)]                           .....3)
END (CR)
```

(2) Function

This command loads an object file created by the assembler for a simulation. It also loads the symbol table information and the input data file (INM file).

(3) Description

The LOAD OBJECT command inputs the object module (an output file from the assembler) specified by 'file name'.

As 'LOAD OBJECT file name(CR)' is entered, '**' is displayed. After this, the object modules, symbol information, and input data file (INM file) are loaded into the simulator sequentially.

Entering 'END' at the end of the Load Object Module command terminates the command.

Subcommands are explained below in the order of their appearance.

1) ALLMODULE, MODULE

This subcommand specifies which modules from the object file should be loaded into the simulator. Normally, all of the modules in the object file are loaded into the simulator before a simulation. However, it is a user's option to selectively load the modules required. It should be noted that, within an object file, there are as many modules as the original number of uPD7281 in the source program.

The modules to be loaded into the simulator are specified by their module names. The module name must have been declared using the MODULE statement in the source program file. When a module is loaded into the simulator, only the object information for the execution section for that module can be used as input information for the simulator.

a) ALLMODULE (or ALLM)

All modules in the object file are loaded into the simulator.

Example:

**ALLMODULE (CR)

b) MODULE

Only those object code corresponding to the specified modules are loaded into the simulator. More than one module name can be specified using the MODULE subcommand.

Example:

**MODULE FUNC1, FUNC2 (CR)

If neither ALLMODULE nor MODULE is specified, no object information will be loaded into the simulator.

2) ALLSYMBOL, SYMBOL

There are three different types of symbols with different attrubutes generated by the assembler. They are LT, FT and DM attribute type symbols. Using either ALLSYMBOL or SYMBOL command, it is possible to include all or only a portion of the symbols generated by the assembler for a simulation.

a) ALLSYMBOL (or ALLS)

When ALLSYMBOL is used, all symbol information contained in the object module is input regardless of the types of attributes. However, if a symbol attribute type is specified along with 'ALLSYMBOL', only those symbols of the specified attribute type are input to the simulator.

Examples:

**ALLSYMBOL (CR)
**ALLS LT (CR) <----- The abbreviation (ALLS) is used.

b) SYMBOL (or SYM)

When SYMBOL is used, only those symbols in the modules specified in the subcommand line are loaded into the simulator. However, if a symbol attribute is used along with 'SYMBOL' subcommand, only those symbols with the specified attribute type within the specified modules are input to the simulator.

Examples:

```
**SYMBOL FUNC01,FUNC02(CR)
**SYM FT FUNC01,FUNC02(CR) <----- The abbreviation
                                  (SYM) is used.
```

If neither ALLSYMBOL nor SYMBOL is specified, no symbol information will be loaded in to the simulator.

3) DATA

The DATA command is used to create the input data token file (INM file) for the simulator. The object code for the DATA section in a source program is used to create the input data token file (INM file).

If a file name is specified with the DATA command, the object code for the DATA section of a source program is written into the file name specified. However, if no file name is specified with the DATA command, then the simulator creates an input data token file with a default file name.

The default file name assigned to the input data token file is the primary file name from the OBJECT file specified previously plus 'INM' extension. The file is created on the same disk where the OBJECT file is stored.

Additionally, if APPEND is tagged onto the DATA command, the input data tokens from the OBJECT file are appended to the end of the file specified. Since the input data tokens are appended to the file, the file must be previously opened before the APPEND operation.

Examples:

```
**DATA (CR)
**DATA B:DATAF.INM (CR) <----- The input data file is
                               created with the name
                               B:DATAF.INM.
**DATA B:DATAF.INM APPEND (CR) <--- This append data to
                                    the file B:DATAF.INM,
                                    opened already.
```

The order in which MODULE, SYMBOL and DATA subcommands are specified is arbitrary.

(4) Application example

```
*LOAD OBJECT B:AFFIN4.OBJ (CR)
**ALLSYMBOL (CR)
**ALLMODULE (CR)
**DATA B:AFFIN4.DAT (CR)
**END (CR)
AFFINE  <----- Module names (assigned by the MODULE
*              statement) of the loaded object module are
               displayed.
```

5.3.2 SAVE OBJECT (Save Object Module command)

(1) Input format

```
OBJECT
  SAVE        file name (CR)
OBJ
```

(2) Function

This command saves templates of uPD7281 to disk file.

(3) Description

This command saves current template of uPD7281 to disk file.
This command is used for keeping changed template or
for inputting to an object converter in case of you have
changed template by ASSEMBLE command.
It outputs all template on memory and contents of current
input data file in the format of object module file.
Caution:
FTT values are initialized zero.

(4) Application examples

```
*SAVE OBJECT D:IMPP.LNK (CR)
*SAVE OBJ D:IMPP LNK > (CR)
*
```

5.4 Simulation Input/Output Data File Specification commands

In the simulator model shown in Fig. 2-1, there is basically one input file to the simulator model and one output file from the simulator model. These two files are specified using the following two commands.

1) DATA INM (Input Data File Specification command)

2) DATA OTM (Output Data File Specification command)

5.4.1 DATA INM (Input Data File Specification command)

(1) Input format

```
DATA  INM  file name (CR)
```

(2) Function

This command specifies the input data file for the simulator model.

(3) Description

The DATA INM command specifies the input data file used for the simulator model. The file which can be specified by this command must be a file created by the simulator using the LOAD OBJECT command (see Section 5.3) or a file that has the same format as the INM file created by the simulator.

(4) Application examples

```
*DATA INM B:DATAF.INM (CR) <----The  file B:DATAF.INM is
*DATA INM B:AFFIN4.DAT (CR)      specified as the input data
*                                file.
```

5.4.2 DATA OTM (Output Data File Specification command)

(1) Input format

```
DATA  OTM  file name (CR)
```

(2) Function

This command specifies the output data file for the simulator model.

(3) Description

The DATA OTM command specifies the output data file used during simulation. When this command is entered, a file with the specified file name is created on the disk drive specified. Usually, the data output to the HOST side at simulation is written into this file.

The user must be aware that, if there exists a file with the same file name specified as an output data file, the file will be overwritten by the output data from the simulator, thus destroying the file with the same name.

(4) Application example

```
*DATA OTM B:DATA.OTM (CR)
*
```

5.5 Trace-log commands

There are two trace-log related commands. One is used to specify the size and the device in which the trace-log is kept, and the other is used to dislay the simulation trace on the console.

1) MAP (Set Trace-log Area command)

2) PRINT (Display Trace-log command)

5.5.1 MAP (Set Trace-log Area command)

(1) Input format

```
MAP { MEMORY          } number of steps (CR)
    { FILE file name  }
```

(2) Function

This commands reserves a trace-log area.

(3) Description

The MAP command reserves a trace-log* area in order to record the simulation traces. The amount of information to be stored is specified by the number of pipeline clock steps. The device where the trace-log can be kept is either on the development system memory or in a disk file.

Unless the trace-log area is reserved using this command, the trace-log information can not be recorded.

(4) Application examples

```
*MAPMEMORY 20 (CR) <-----Reserves trace-log area for 20
*MAP FILE B:TRACE.LOG 100 (CR)<-┐      steps on memory.
*MAP MEMORY 200 (CR)            └---Reserves trace-log area
                                    for 100 steps on file
                                    B:TRACE.LOG.

*** ERROR A901 WORKING TABLE SPACE EXHAUSTED
                  ↑
                  └----This message will be displayed when
                       reservation is not possible. (Not
                       enough free memory spaces or not
                       enough free disk spaces.)
```

NOTE; If the trace-log area cannot be reserved on the system memory, either specify a disk file as a trace-log device or decrease the number of definitions of macros and symbols.

---

*Trace-log; information produced as a result of simulation.

5.5.2 PRINT (Display Trace-log command)

(1) Input format

#nn , ...] PRINT [ALL]		[number of steps]	(CR)

(2) Function

This command displays simulation trace-log information on the console.

(3) Description

The PRINT command displays the simulation trace-log information on the console. The trace-log dislayed must have been previously recorded in the trace-log area specified.

The amount of information to be displayed is specified in terms of the number of pipeline clock steps. If the PRINT command contains 'n' as a number of steps, the 'n' last simulation steps will be displayed. However, if the number of steps is not specified, then all of the information currently stored in the trace-log area will be displayed.

For display format, please refer to 'Trace-log Display Format' in Chapter 3.

(4) Application example

```
*DEFINE TRACE (CR)      <----- Refer to Section 5.11.1
**#8 @LT (CR)
**END (CR)
*; (CR)
*PRINT 20 (CR)          <----- Displays 20 steps (clocks).
CLOCK  # 8@LTID  # 8@LTD  # 8@LTD
    1      *         *         *
    2      *         *         *
    3      *         *         *
    4  WRBASE       EXEC      0020H
    5      *         *         *
    6  STATTK       EXEC      0000H
    7      *         *         *
    8      *         *         *
    9      *         *         *
   10      *         *         *
   11  X43          EXEC      0020H
   12  X44          EXEC      0021H
   13  X1           EXEC      0000H
   14  X2           EXEC      0020H
   15      *         *         *
   16      *         *         *
   17      *         *         *
   18  X39          EXEC      0020H
   19  X40          EXEC      0021H
   20  X4           EXEC      0000H
END
*
```

5.6 Host Data Input command

As Fig. 2-1 shows, the input data tokens from the host can also be input to the Input Control Block in the simulation model. The tokens entered into the Input Control Block from the host are output to the uPD7281 block. These tokens from the host are simulated by HOST command.

5.6.1 HOST

(1) Input format

```
HOST  expression-1, expression-2 (CR)
```

(2) Function

This command is used to simulate the input data token from the HOST CPU.

(3) Description

The HOST command simulates the input data tokens going into the uPD7281 from the HOST CPU. Usually, such a token is used to initiate a program execution. Or it may also be used for debugging purposes.

Since the command simulates inputting a data token from the host CPU, 'expression-1' and 'expression-2' must conform to the I/O token format of the uPD7281. Hence, 'expression-1' corresponds to the first 16-bit half of a token, and 'expression-2' corresponds to the second 16-bit half of a token. The values specified in 'expression-1' and 'expression-2' must be in the range of 0 to 65535.

(4) Application example

```
*HOST 1007H,0000H (CR)
*
```

5.7 Simulation Execution commands

There are three Simulation Execution commands available for the simulator as follows.

1) GO (Continuous Execution command)

2) STEP (Single Step Execution command)

3) { CONTINUE / CONT } (Resume Execution command)

5.7.1 GO (Continuous Execution command)

(1) Input format

```
GO [  GR break condition] (CR)
```

(2) Function

This command initates a simulation (or re-initiates if a simulation is stopped due to a break condition).

(3) Description

The GO command starts a simulation. The simulation is executed until a break condition occurs as a result of one of the following events.

- o Input of ESC key
- o Occurrence of a non-fatal error (an error that outputs an error message and returns the control to the prompt mode '*').
- o Meeting one of the preset break conditions.

There are three types of registers where break conditions can be set.

GO Register (GR)

This register sets a condition which breaks the simulation in progress. The break condition for the GO Register is set using Set GR command explained in Section 5.8. Another way to interrupt a simulation in progress is to use the GO command followed by parameters for the break condition desired.

Break Registers (BR0 ~ BR3)

These four registers are used to store break conditions for the simulator. These are merely break condition registers that store the specified break conditions, and they do not have the capability to stop a simulation unless they are used along with the GO command.

Using a combination of Break Registers, a very sophisticated break condition can be specified for the GO command. The Break conditions are set into registers BR0 ~ BR3 with the Set BR command (refer to Section 5.9).

Breakpoint Registers (BP0 ~ BP3)

These registers are used to set conditions where the simulator breaks during a simulation. After break conditions are set into the BP0~BP3 registers, a simulation may be interrupted if, during a simulation, one of the preset break conditions in the BP's is satisfied. After a simulation break due to one of the BP registers, the CONTINUE command (see Section 5.7.3) may be used to resume the simulation.

There are eight different ways to set the GR break conditions. The following shows the GR command formats and the examples.

1)

```
[#n ] { LT } { address expression                                          }
      { FT } { address expression TO address expression                    }
      { DM } { address expression LENGTH address range expression          }

      { ACCESSED } [ m TIMES]
      { ACCE     }
```

a. n

Specifies one of the module numbers of the uPD7281s being simulated. (If n is omitted, the current module number is assumed by default.)

b. LT, FT, DM

Specify either LT, FT, or DM.

c. address expression

Specifies either LT, FT, or DM address. The parameter TO or LENGTH is used to specify a memory address range. 'TO' is used to specify the lower and upper bound memory address range, while 'LENGTH' is used to specify a block of memory with the lower bound address and the length from it.

For LT memory .... $0 \leq$ address expression ≤ 127
For FT memory .... $0 \leq$ address expression ≤ 63
For DM memory .... $0 \leq$ address expression ≤ 511

d. ACCESSED

Indicates that the break condition is a memory access.

e. m TIMES

The number of times the specified condition has to be met before a break in a simulation occurs. (Here it means the number of memory accesses.) If m is omitted, it is assumed to be '1'.

Using this command, it is possible to break a simulation when tokens access a particular memory location or a range of memory locations a specified number of times.

Example:

```
*GO #1 LT 30H ACCESSED 2 TIMES (CR)
   ; After a simulation is started (or re-initiated after an
     interruption), the simulation will break if address
     30H of the LT memory in module-1 is accessed 2 times.
```

2) [#n] { @LT / @FT / @DM } DATA mask data [m TIMES]

a. @LT, @FT, @DM

Specify either the LT Latch, FT Latch, or DM Latch.

b. DATA

Indicates that the break condition is a data value latched in one of the latches.

c. mask data

Specifies the break condition data value to be compared against the data stored in the specified latch.

d. Other items are the same as in 1).

Using this command, a simulation may be interrupted if the content of the token latched in the specified latch is equal to the data value specified 'm' number of times.

The mask data value for the command has a certain flexibility. The format of the mask data specification is as follows.

Comparative value expression [FIELD bit position expression LENGTH bit size expression]

In the mask data format, the number after the FIELD statement points to the lowest bit position of the data to

be compared, and the number after the LENGTH statement specifies the size of the data field to be compared against the incoming tokens.

Example:

```
*GO #1 @LT DATA 30H FIELD 0 LENGTH 10H 3 TIMES (CR)
   ; The simulation will break if the 16-bit (10H) value
     starting from the bit 0 of the LT Latch of the module
     number 1 matches the value of 30H three times.
```

3)

[#n] { DQ / GQ / OQ } SIZE size expression [m TIMES]

a. DQ, GQ, OQ

Specifies either the Data Queue, Generator Queue, or Output Queue.

b. SIZE

Indicates that the break condition is a queue size level.

c. size expression

Specifies the value of the queue level size.

d. Other items are the same as in 1).

Using this command, a simulation may be interrupted if the queue level matches the specified level size 'm' times. Only one of the three queues may be selected per command.

Example:

```
*GO #1 DQ SIZE 8 5 TIMES (CR)
   ; A simulation will be interrupted if the DQ level of the
     uPD7281 with MN=1 matches eight, five times.
```

4)

[#n] { INPUT / OUTPUT } [m TIMES]

a. INPUT, OUTPUT

Specifies either an input operation or an output operation.

b. Other items are the same as in 1).

Using this command, a simulation may be interrupted if a certain number of tokens is input to the #n uPD7281, or output from the #n uPD7281.

Example:

```
*GO #1 INPUT 100 TIMES (CR)
   ; After a simulation is started or re-started after an
     interruption, it will break if exactly 100 tokens are
     input to the uPD7281 with MN=1.
```

5)

BRn [{AND / OR} BRn] [m TIMES]

a. BRn

Specifies one of the break registers, BR0 BR3.

b. AND, OR

Specifies either a logical product or a logical sum of the break conditions set in BRn.

c. Other items are the same as in 1).

Using this command, a very sophisticated break condition may be set. After an individual BR has been set, a combination (a logical OR or AND combination) of the Break Registers can be used to specify even more complicated break conditions at which the simulator may be interrupted.

Example:

```
*GO BR0 AND BR1 2 TIMES (CR)
        ; The example specifies that the simulation should
          be interrupted if the break conditions set in BR0
          and BR1 are simutaneously met 2 times.
```

6) expression STEP

a. expression

Specifies the number of pipeline clock steps.

b. STEP

Indicates that the break condition is a number of pipeline clock steps.

After a simulation is started, the simulation will break immediately after the specified number of the pipeline clock steps is reached.

Example:

```
*GO 100 STEPS (CR)
   ; After a simulation started, the simulation will be
     interrupted immediately after 100 pipeline clock steps.
```

7) IM { READ / WRITE } [m TIMES]

a. IM

Stands for Image Memory.

b. READ, WRITE

Specifies either reading from or writing to the image memory.

c. Other items are the same as in 1).

Using this command, it is possible to break the simulation on the number of times the image memory is read or written.

Example:

```
*GO IM READ 10 TIMES (CR)
   ; The simulation will be interrupted immediately after
     the 10th image meory read has been taken place.
```

8) { INM / OTM } [m TIMES]

a. INM, OTM

Specifies either an INPUT file or an OUTPUT file to and from the simulator.

Using this command, it is possible to break the simulation on the number of times a token is read out from the input file (INM file), or the number of times a token is written into the output file (OTM file).

Example:

```
*GO OTM 20 TIMES (CR)
   ; The simulation will be interrupted immediately after
     the simulator outputs  20 tokens to the output file
     (OTM file).
```

(4) Application examples

```
*GO #8 LT 30H ACCESSED 2 TIMES (CR)
BREAK BY ESC, SIMULATION TO INTERRUPT;
                                  CLOCK=259 <-┐  Execution was
*GO #8 @LT DATA 30H FIELD 0 LENGTH 10H        │  interrupted by
                            3 TIMES (CR)      ├--an ESC key
BREAK BY ESC, SIMULATION TO INTERRUPT;        │  entry.
                                  CLOCK=494 <-┘
*PRINT 6 (CR) <----- Displays the trace-log.
CLOCK  # 8@LTID  # 8@LTC  # 8@LTD
  489  X16         EXEC     0040H
  490  X15         EXEC     0060H
  491  X16         EXEC     0060H
  492  X17         EXEC     0000H
  493  X20         EXEC     0000H
  494  X16         EXEC     0080H
END
*
```

5.7.2 STEP (Single Step Execution command)

(1) Input format

STEP (CR)

---

(2) Function

This command executes only one pipeline step of simulation.

(3) Description

The STEP command executes only one pipeline clock cycle of simulation.

(4) Application examples

```
*BR0=10 STEPS (CR)    <-----  See Section 5.9.1.
*; (CR)
*BP0=5 STEPS (CR)     <-----  See Section 5.10.1.
*CLOCK (CR)
CLOCK =    100        <-----  See Section 5.15.1.
*; (CR)
*GO BR0 (CR) <----- Executes simulation until condition set
                    in BR0 is met.
BREAK BY BP0, SIMULATION TO INTERRUPT; CLOCK =    105
                                          ↑—The condition set
*STEP (CR) <--┐                           in BP0 was met.
*STEP (CR) <--┴---Executes one pipeline
                  step at a time.
*CLOCK (CR)
CLOCK =    107 <---The pipeline clock step counter has
                   been incremented by 2.
*STEP (CR) <--┐
*STEP (CR) <--┤
*STEP (CR) <--┴---- Executesl pipeline step ata time.
BREAK BY END, SIMULATION TO STOP; CLOCK =    110
*                          ↑
*                          └-- The condition in BR0 was met.
```

5.7.3 CONTINUE (Continue Execution command)

(1) Input format

```
{CONTINUE} (CR)
{CONT     }
```

(2) Function

This command is used to resume a simulation after an interruption.

(3) Description

The CONTINUE command can resume a simulation that has been interrupted during an execution of a GO command as a result of one of the following events:

- o Input of ESC key
- o Occurrence of a non-fatal error (an error that outputs an error message and then returns the control to the simulator's prompt mode '*').
- o Meeting one of the break conditions set in the Unconditional Break point Registers (BP0 ~ BP3).

A simulation break due to meeting a break condition set in one of the four Break Registers (BR0 ~ BR3) cannot be resumed by entering the CONTINUE command.

(4) Application examples

```
*BR0=10 STEPS (CR) <----- See Section 5.9.1.
*BP0=5 STEPS (CR) <----- See Section 5.10.1.
*CLOCK (CR)
CLOCK =     0
*GOBR0 (CR) <----- The simulation is executed until the
                       condition set in BR0 is met.

BREAK BY BP0 , SIMULATION TO INTERRUPT;
                       CLOCK =     5 <----- The condition in
                                            BP0 was met first.
*CONTINUE (CR)<--Simulation is resumed.
BREAK BY BP0 , SIMULATION TO INTERRUPT;
                       CLOCK =    10
*CONTINUE (CR)
BREAK BY BP0 , SIMULATION TO INTERRUPT;
                       CLOCK =    15

*BR2=4 STEPS (CR)<--- Sets a break condition in BR2.
*; (CR)
```

```
*GO BR2 (CR) <------- The simulation is executed until the
                      condition set in BR2 is satisfied.
BREAK BY END , SIMULATION TO STOP; CLOCK =     19
*CONTINUE (CR)
*** ERROR F005 ILLEGAL COMMAND<--The CONTINUE command cannot
*                                resume  simulation in case
                                 of a break in BR2.(The same
                                 is applied to BR0, BR1, and
                                 BR3.)
```

5.8 Set/Display/Cancel GR Break Condition commands

There are three separate commands to Set/Display/Cancel GR Break Condition, as shown below.

1) GR=[GRbreakcondition] (Set GR BreakConditioncommand)

2) GR (Display GR Break Condition command)

3) NOGR (Cancel GR Break Condition command)

5.8.1 GR=GR break condition (Set GR Break Condition command)

(1) Input format

GR=GR break condition (CR)

---

(2) Function

This command sets the GR (Go Register) break condition.

(3) Description

The Set GR Break Condition command sets a condition in the GR (Go Register) to break a simulation. Section 5.7.1 explains how to use the GO command after setting the GO register.

(4) Application examples

```
*GR=BRO (CR) <--------  Sets the GO register break condition
                        equal to the condition set in the
                        BR0.

*GR=BRO AND BR1 02 TIMES (CR)<--- Sets the GO register break
                        condition to a particular instance
                        when the BR1 condition and the BR0
                        condition are simutaneously met 2
                        times.
*
```

5.8.2 GR (Display GR Break Condition command)

(1) Input format

```
GR (CR)
```

(2) Function

This command displays GR break condition.

(3) Description

This command displays the current break condition set in GR (Go Register). If no condition is set, 'NOTHING' is displayed.

(4) Application examples

```
*GR (CR)
BR0 AND BR1 00002 TIMES
*GR=BR0 (CR) <---- Changes the set condition.
*GR (CR)
BR0         1TIMES
*NOGR (CR) <----- See Section 5.8.3
*GR (CR)
NOTHING    <----- No condition is set.
*
```

5.8.3 NOGR (Cancel GR Break Condition command)

(1) Input format

```
NOGR (CR)
```

(2) Function

This command cancels GR break condition.

(3) Description

The NOGR command cancels the break condition currently set in GR (Go Register).

(4) Application example

```
*GR=BR0 (CR)
*GR (CR)
BR0
*NOGR (CR) <----- Cancels GR break condition.
*GR (CR)
NOTHING
*
```

5.9 Set/Display/Cancel Conditional Breakpoint Register commands

There are three separate commands to Set/Display/Cancel the Conditional Breakpoint Register, as shown below.

1) BRn=break condition (Set Conditional Breakpoint Register command)

2) BR[n] (Display Conditional Breakpoint Register command)

3) NOBR[n] (Cancel Conditional Breakpoint Register command)

($0 \leq n \leq 3$)

5.9.1 BRn=break condition (Set Conditional Breakpoint Register command)

(1) Input format

BRn=break condition (CR) (0 ≤ n ≤ 3)

(2) Function

This command sets a break condition in a Conditional Breakpoint Register.

(3) Description

The Set Conditional Breakpoint Register command sets a break condition in the specified Conditional Breakpoint Register (one of the BR0 BR3). The break conditions set in BRn are not effective unless the Break Registers (BR ~ BR3) are used along with the GO command.

(4) Application examples

```
*BR0=#1 OQ SIZE 8 5 TIMES (CR)
*BR2=#8 INPUT 2 TIMES (CR)
*BR3=IM READ 50 TIMES (CR)
*
```

5.9.2 BR[n] (Display Conditional Breakpoint Register command)

(1) Input format

```
BR[n] (CR)     (0 ≤ n ≤ 3)
```

(2) Function

This command displays the break condition set in a Conditional Breakpoint Register.

(3) Description

This command displays the break condition set in the specified Conditional Breakpoint Register (one of the BR0 ~BR3) or the break conditions set in all of them. If the register number n is omitted, the contents of all Conditional Breakpoint Registers are displayed. If no break condition is set in BR0~BR3, 'NOTHING' is displayed when the command is entered.

(4) Application examples

```
*BR0 (CR)
BR0=01000 STEPS
*BR (CR)
BR0=01000 STEPS                    ┐
BR1=#1 INPUT 00001 TIMES           } Displays conditions set in
BR2=IM WRITE 00100 TIMES           } each register.
BR3=#1 OQ SIZE 08 00010 TIMES      ┘
*
```

5.9.3 NOBR[n] (Cancel Conditional Breakpoint Register command)

(1) Input format

NOBR[n] (CR) ($0 \leq n \leq 3$)

(2) Function

This command cancels the break condition set in a Conditional Breakpoint Register.

(3) Description

The NOBR[n] command cancels the break condition set in either one of the specified Conditional Breakpoint Register (one of BR0 BR3) or all of BR0 BR3. If the register number n is omitted, the break conditions of all Conditional Breakpoint Registers are cancelled.

(4) Application examples

```
*NOBR0 (CR) <----- Cancels the break condition set in the
                   BR0 only.
*BR (CR)
BR0 = NOTHING
BR1 = #1 INPUT 00001 TIMES
BR2 = IM WRITE 00100 TIMES
BR3 = #1 OQ SIZE 08 00010 TIMES
*NOBR (CR) <-----  Cancels the break conditions set in all
                   Break Registers.
*BR (CR)
BR0 = NOTHING
BR1 = NOTHING
BR2 = NOTHING
BR3 = NOTHING
*
```

5.10 Set/Display/Cancel Unconditional Breakpoint Register commands

There are three separate commands to Set/Display/Cancel Unconditional Breakpoint Register (BP), as shown below.

1) BPn=break condition (Set Unconditional Breakpoint Register command)

2) BP[n] (Display Unconditional Breakpoint Register command)

3) NOBP[n] (Cancel Unconditional Breakpoint Register command)

$(0 \leq n \leq 3)$

5.10.1 BPn=break condition (Set Unconditional Breakpoint Register command)

(1) Input format

BPn=break condition (CR) (0 ≤ n ≤ 3)

(2) Function

This command sets a break condition in an Unconditional Breakpoint Register.

(3) Description

The Set Unconditional Breakpoint Register command sets a break condition in a specified Unconditional Breakpoint Register (one of BP0 BP3). However, no BPn can be specified on the right side of the '=' sign as a break condition (BPn's cannot be nested). For the methods of setting break conditions, refer to Section 5.7.1, 'GO command'.

The break conditions set in all Break Point Registers are always checked during a simulation in the case where the simulation is executed by a GO command. If one of the break conditions set in the four Break Point Registers (BP0 ~ BP3) is satisfied during a simulation, the simulation will be interrupted and the current pipeline clock count will be displayed. When a simulation is interrupted due to satisfying one of the BP0 ~ BP3, the simulation may be resumed by entering the CONTINUE command.

(4) Application examples

```
*BP0=#1 OQ SIZE 8 5 TIMES (CR) <----- Sets a break condition
                                      in BP0.
*BP1=#8 INPUT 2 TIMES (CR) <-----  Sets a break condition in
*                                  BP1.
```

5.10.2 BP[n] (Display Unconditional Breakpoint Register command)

(1) Input format

BP[n] (CR) ($0 \leq n \leq 3$)

(2) Function

This command displays the break condition set in an Unconditional Breakpoint Register(s).

(3) Description

The BP[n] command displays either a break condition set in the specified Unconditional Breakpoint Register (one of BP0 ~ BP3) or all of the conditions in BP0 ~ BP3. If the register number 'n' is omitted, the contents of all Unconditional Breakpoint Registers are displayed.

(4) Application examples

```
*BP0 (CR)
BP0 = 01000 STEPS (CR)
*BP (CR)
BP0 = 01000 STEPS
BP1 = #1 INPUT 00001 TIMES
BP2 = IM WRITE 00100 TIMES
BP3 = #1 OQ SIZE 08 00010 TIMES
*
```

5.10.3 NOBP[n] (Cancel Unconditional Breakpoint Register command)

(1) Input format

NOBP[n] (CR)

(2) Function

This command cancels a break condition set in an Unconditional Breakpoint Register(s).

(3) Description

The NOBP[n] command cancels either a break condition set in the specified Unconditional Breakpoint Register (one of BP0~BP3) or the break conditions in all of them. If the register number 'n' is omitted, the break conditions in all Unconditional Breakpoint Registers are cancelled.

(4) Application examples

```
*NOBP0 (CR) <----- Cancels only the break condition set in
                   BP0.
*BP0 (CR) <----- Displays the content of BP0.
BP0 = NOTHING
*NOBP(CR)  <-------  Cancels break conditions in all BP
*BP (CR)             registers.
BP0 = NOTHING
BP1 = NOTHING
BP2 = NOTHING
BP3 = NOTHING
*
```

5.11 Set/Display/Cancel Trace Item commands

There are nine different commands to Set/Display/Cancel simulation trace items.

1) DEFINE TRACE (Set Trace Item command)

2) TRACE (Display Trace Item command)

3) NOTRACE (Cancel Trace Item command)

4) TON=trace condition (Set Trace Start Condition command)

5) TON (Display Trace Start Condition command)

6) NOTON (Cancel Trace Start Condition command)

7) TOFF=trace condition (Set Trace End Condition command)

8) TOFF (Display Trace End Condition command)

9) NOTOFF (Cancel Trace End Condition command)

5.11.1 DEFINE TRACE (Set Trace Item command)

(1) Input format

```
DEFINE TRACE (CR)
command (CR)    [...]
END (CR)
```

(2) Function

This command declares the items to be traced during a simulation.

(3) Description

The DEFINE TRACE command specifies the items to be traced during a simulation. Once the trace items are declared using the command, all declared trace items are recorded in the simulator's trace-log area specified. It should be noted that the states of all trace items are recorded into the trace-log area at each pipeline clock steps. If DEFINE TRACE command is not used to specify the items to be traced, all traceable items are recorded in the trace-log area.

As 'DEFINE TRACE (CR) ' is entered, '**' is displayed. Following this, the items to be traced in the trace-log area are entered sequentially. Entering these items is equal to the 'command' in the input format shown above. The items that can be kept in trace-log are shown in Table 5-1.

Table 5-1. Command group (trace items)

Trace item	Meaning
CLOCK	Number of pipeline steps
@INB	Input Data Buffer
@IN	LT Transfer Latch
@LT	LT Input Latch
@PT	FT Input Latch
@DM	DM Input Latch
@Q	Q Input Latch
@OQ	OQ Input Latch
@PU	PU Input Latch
@PASS	Pass Data Latch
@OUTB	Output Data Buffer
@BRC	Break Counter
@INHC	Input Inhibit Counter
@REFC	Refresh Counter
DQSIZE	DQ size
GQSIZE	GQ size
OQSIZE	OQ size

(4) Application example

```
*DEFINE TRACE (CR)
**#8 @LT (CR)          Sets trace items.
**#8 @DM (CR)
**END (CR)
*; (CR)
*GO 10 STEPS (CR) <----- Executes simulation.
BREAK BY BP0 , SIMULATION TO INTERRUPT; CLOCK = 142
*PRINT 10 (CR)
CLOCK    # 8@LTID    # 8@LTC    # 8@LTD        # 8@DM
  133    X23          EXEC       0004H      01305D00680180H
  134    X23          EXEC       0005H      014E0D00680180H
  135    X23          EXEC       0006H      21622A80600004H
  136    X17          EXEC       0024H      21622B80600005H
  137    X20          EXEC       0000H      21622C80600006H
  138    X15          EXEC       01A0H      01340E00600024H
  139    X16          EXEC       01A0H      015408006000000H
  140    X17          EXEC       0028H      01305C006801A0H
  141    X15          EXEC       01C0H      014E0C006801A0H
  142    X16          EXEC       01C0H      01340E00600028H
END
*
```

5.11.2 TRACE (Display Trace Item command)

(1) Input format

```
TRACE (CR)
```

(2) Function

This command displays trace items.

(3) Description

The TRACE command displays all trace items specified by the DEFINE TRACE command.

(4) Application examples

```
*TRACE (CR)
CLOCK
#1 @LT
#1 @DM
END
*TRACE (CR)
END <----- If no trace item is set, only 'END' is displayed.
*
```

5.11.3 NOTRACE (Cancel Trace Item command)

(1) Input format

```
NOTRACE (CR)
```

(2) Function

This command cancels all trace item.

(3) Description

The NOTRACE command cancels all trace items specified by the DEFINE TRACE command. After the trace items are cancelled, the execution of PRINT command causes display of all traceable items.

(4) Application example

```
*DEFINE TRACE (CR) <----- Sets trace items.
**CLOCK (CR)
**@LT (CR)
**@FT (CR)
**@DM (CR)
**END (CR)
*TRACE (CR) <----- Displays trace items.
CLOCK
@LT
@FT
@DM
END
*NOTRACE (CR) <----- Cancels trace items.
*TRACE (CR)
END       <----- All trace items have been cancelled.
*
```

5.11.4 TON=trace condition (Set Trace Start Condition command)

(1) Input format

```
TON=trace condition (CR)
```

(2) Function

This command sets the trace display start condition.

(3) Description

The Set Trace Start Condition command specifies the condition when the trace information is to be displayed on the console. Thus, without the use of PRINT command, specified trace information can be displayed on the console when the simulation is initiated using the GO command.

The items to be displayed on the console are the trace items previously declared using the DEFINE TRACE command. The 'TON' command merely specifies when the trace items are to be displayed on the console. Thus, the 'TON' command has no effects on the items being recorded in the trace-log area.

There is a complementary command of the 'TON' called 'TOFF' which is used to specify when the display of the trace items is to be stopped.

In addition to those break conditions explained in Section 5.7.1, 'GO command', one of the conditions shown below can be used as a trace condition in the above input format.

NOWON .... After executing the GO command, the trace information is always displayed on console.

NOWOFF .... Even when the GO command is executed, the trace information is not displayed on console.

(4) Application examples

```
*DEFINE TRACE (CR)
**#8 @LT (CR)
**#8 @DM (CR)
**END (CR)
*; (CR)
*TON=2 STEPS (CR) <----- The trace condition is set.
*GO 20 STEPS (CR) <----- The GO command is executed.
*#8 @LT
@LT      2A 0 0000          }  After 2 steps of execution, the
*#8 @DM                     }  trace information (@LT, @DM) is
@DM      02201010000000     }  displayed.
*END
*#8 @LT
@LT      1A 0 001C          }
*#8 @DM                     }  The same is repeated.
@DM      015A6E00180000     }
*END
*#8 @LT
@LT      2D 0 0000
*#8 @DM
@DM      01386E0018001C
*END
*#8 @LT
@LT      15 0 0100
BREAK BY ESC , SIMULATION TO INTERRUPT; CLOCK = 109
*TON=NOWON (CR) <----- NOWON is set as a trace condition.
*GO 20 STEPS (CR)
*#8 @LT                     )
@LT      27 0 0000          )
*#8 @DM                     }  Trace information is always
@DM      02201010000000     )  displayed.
*END                        )
:
:
```

5.11.5 TON (Display Trace Start Condition command)

(1) Input format

```
TON (CR)
```

(2) Function

This command displays the trace start condition.

(3) Description

The TON command displays the trace start condition which is set by the Set Trace Start Condition command. If no condition is set, 'NOTHING' is displayed.

(4) Application examples

```
*TON (CR)
TON = 100 STEPS
*TON (CR)
TON = NOWON
*TON (CR)
TON = NOTHING <----- No trace start condition has been set.
*
```

5.11.6 NOTON (Cancel Trace Start Condition command)

(1) Input format

```
NOTON (CR)
```

(2) Function

This command cancels the trace start condition.

(3) Description

The NOTON command cancels the trace start condition set by the Set Trace Start Condition command.

(4) Application example

```
*TON (CR) <-----Trace start condition is to be displayed.
TON = 100 STEPS
*NOTON (CR) <-----Trace start condition is to be cancelled.
*TON (CR)
TON = NOTHING <----Trace start condition has been cancelled.
*
```

5.11.7 TOFF=trace condition (Set Trace End Condition command)

(1) Input format

```
TOFF=trace condition (CR)
```

(2) Function

This command sets the trace end condition.

(3) Description

The Set Trace End Condition command specifies the condition when the display of the trace information on the console is to be terminated. When the trace condition specified by this command is met, the trace information being displayed on the console will be stopped.

The conditions that can be specified in 'trace condition' are the same as those in Section 5.11.4, 'Set Trace Start Condition command', with the exception that NOWON and NOWOFF cannot be specified.

(4) Application example

```
*TOFF=00200 STEPS (CR)
*
```

5.11.8 TOFF (Display Trace End Condition command)

(1) Input format

```
TOFF (CR)
```

(2) Function

This command displays the trace end condition.

(3) Description

The TOFF command displays the trace end condition specified by the Set Trace End Condition command. If no condition is specified, 'NOTHING' is displayed.

(4) Application examples

```
*TOFF (CR)
TOFF = NOTHING    <----- No condition has been specified.
*TOFF=100 STEPS (CR)  <----- Sets a condition.
*TOFF (CR)
TOFF = 100 STEPS
*
```

5.11.9 NOTOFF (Cancel Trace End Condition command)

(1) Input format

```
NOTOFF (CR)
```

(2) Function

This command cancels the trace end condition.

(3) Description

The NOTOFF command cancels the trace end condition set by the Set Trace End Condition command.

(4) Application example

```
*TOFF (CR)
TOFF = 100 STEPS
*NOTOFF (CR) <----- Cancels the trace end condition.
*TOFF (CR)
TOFF = NOTHING <----- The condition has been cancelled.
*
```

5.12 Symbolic Debug commands

There are eight symbolic debugging commands for the simulator, as shown below.

1) CALCULATE (Calculate/Display Symbol command)

2) DEFINE SYMBOL (Define Symbol command)

3) NOSYMBOL (Delete Symbol command)

4) .symbol= (Modify Symbol command)

5) address expression= (Modify Memory command)

6) SYMBOLS (Display Symbol Table command)

7) .symbol (Display Symbol command)

8) address expression (Display Memory command)

5.12.1 CALCULATE (Calculate/Display Symbol command)

(1) Input format

```
CALCULATE expression [[ #n] symbol type SYMBOLIC] (CR)
```

(2) Function

This command evaluates a specified expression and displays its value.

(3) Description

The CALCULATE command evaluates a specified expression and displays the value of the result in its 2's complement form in binary, in octal, in decimal and in hexadecimal. Additionally, using the CALCULATE command it is possible to display or define a symol in various form. The format in which the symbol information is displayed depends upon the type of symbol and the way it was previously defined using the DEFINE SYMBOL command.

For detail explanations of display format, please refer to Section 5.12.2, and for the symbol types, see APPENDIX 2.

(4) Application examples

```
*CALCULATE 127+127 (CR)
BIN = 0000000011111110  OCT = 000376   DEC =   254   HEX = 00FE
      1111111100000010        177402         65282         FF02
*CALCULATE (1+2+3)*10/5 (CR)
BIN = 0000000000001100  OCT = 000014   DEC =    12   HEX = 000C
      1111111111110100        177764         65524         FFF4
*CALCULATE 63 #1 LT SYMBOLIC (CR)
#01 LT 63=        LINK63    (NOTE)
*
```

NOTE; Refer to Section 5.12.2.

5.12.2 DEFINE SYMBOL (Define Symbol command)

(1) Input format

```
DEFINE  { SYMBOL }  symbol name (CR)
        { SYM    }

[#n ]symbol type[  address expression][  field specification](CR)

END (CR)
```

(2) Function

This command defines a symbol.

(3) Description

The DEFINE SYMBOL command defines the specified symbol and enter the symbol into the symbol table. When 'DEFINE SYMBOL symbol name (CR)' is entered, the simulator displays '**' as a prompt sign and expects a symbol definition to be entered. The symbol definition is composed of the module number '#n', 'symbol type', 'address expression', and 'field specification'. After entering the symbol definition line plus a carriage-return, the DEFINE SYMBOL command must be terminated by an END statement followed by a carriage-return.

The module number 'n' must be within one and fourteen. Module numbers not specified by the ENVIRON command can also be used. If the symbol to be defined is for one of the registers or latches listed in APPENDIX 2, then the 'address expression' must not be specified since an address will have no meaning. If the symbol to be defined is memory related such as LT and FT memory and DM, then the 'address expression' must be specified. If the symbol to be defined is for a queue type symbol, then all items must be specified.

The 'field specification' defines what the symbol should represent and the format of the symbol. Two different field specification formats are available for the simulator, and the correct field specification must be used for the type of a symbol to be defined.

1) Specification of memory (LT, FT or DM), latch or register as a symbol type

   In this case, the symbol should be defined with the following field specification format.

FIELD bitposition expression LENGTH bit size expression

The symbol to be defined using the DEFINE SYMBOL command can represent either the whole value or a portion of the value contained in the specified memory location, latch or register. The field specification needs to be specified only if a portion of the specified memory location, latch or register is to be represented by the symbol being defined.

If a portion of the whole field is to be represented by the symbol being defined, then the 'bit position expression' in the above format points to the lowest bit position of the portion to be represented by the symbol. The 'bit size expression' is the width of the bit field being defined. Fig. 5-3 shows an example of the FIELD specification.

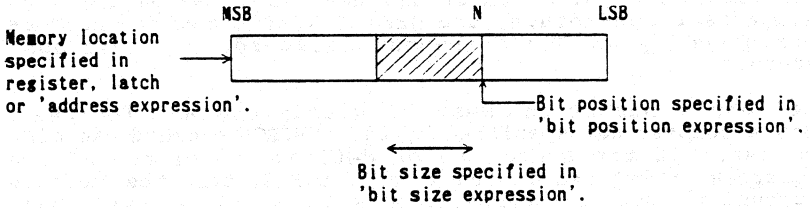


Fig. 5-3

2) Specification of a queue as a symbol type

In this case, the symbol is assigned to a DM area which represents a Data Memory Queue (DMQ). Since a DMQ specification requires several parameters to be completely defined, the following format should be used for defining any QUEUE type symbols.

```
-------------------------------------------------------
I BASE  base field symbol  FPTR first pointer symbol   I
I    LPTR last pointer symbol  SIZE  queue size symbol I
-------------------------------------------------------
```

BASE, SIZE ; The BASE specifies the symbol for the DMQ's base address. The SIZE specifies the size of the DMQ.

FPTR, LPTR ; The FPTR (First Pointer) and the LPTR (Last Poniter) represent the two pointers required for a queue. The FPTR points to the lower address location and the LPTR points to the upper address location within the DMQ size specified. The address location specified by the FPTR and the LPTR are relative the the DMQ base address specified in the DEFINE SYMBOL command as 'address expression'.

Fig. 5-4 illustrates the DMQ specification.

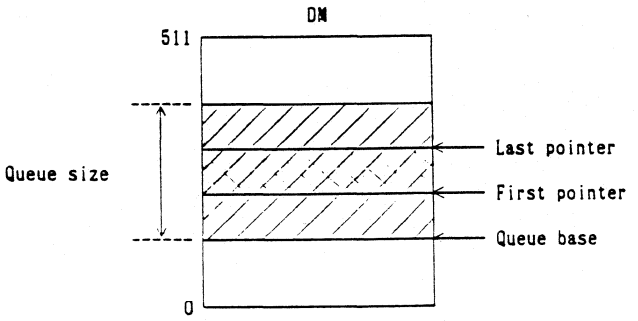


Fig. 5-4

(4) Application examples

```
*DEFINE SYMBOL SYMMNO (CR)
**#1 @IN FIELD 12 LENGTH 4 (CR)
**END (CR)
*
*DEFINE SYM DMQUEUE (CR)<---'SYMBOL' is coded in abbreviated
                           form.
**#1 QUEUE 20H BASE QUEBASE FPTR QUEFPTR LPTR QUELPTR SIZE
                                                 QUESIZE (CR)
**END (CR)
*
```

5.12.3 NOSYMBOL (Delete Symbol command)

(1) Input format

```
NOSYMBOL [ #n] [ symbol] (CR)
```

(2) Function

This command deletes a previously defined symbol or all symbols in a module.

(3) Description

The NOSYMBOL command deletes a specified symbol from the symbol table. This command is used to delete unnecessary symbols from the table. If the module number (#n) is omitted, the specified symbol is deleted from the current module's symbol table. If 'symbol' specification is omitted, all symbols in the symbol table of the specified module are deleted. (When the module number is omitted, the 'current module' is assumed.)

(4) Application examples

```
*NOSYMBOL#1 SYMMNO (CR) <-----  Only 'SYMMNO' in the module
                                number 1 is deleted.
*NOSYMBOL #1 (CR) <----- All symbols in the module number 1
                         are deleted.
*NOSYMBOL (CR) <----- All symbols in the current module are
                      deleted.
```

5.12.4 .symbol= (Modify Symbol command)

(1) Input format

```
                                  ( address expression          )
                                  | FIELD  bit position expression |
                                  | LENGTH  bit size expression  |
.[#n  ]symbol=                    | BASE  base field symbol      | (CR)
                                  | FPTR  first pointer symbol   |
                                  | LPTR  last pointer symbol    |
                                  ( SIZE  queue size symbol      )
```

(2) Function

This command modifies paramerters of a symbol already registered in the symbol table.

(3) Description

The Modify Symbol command modifies the parameters of a symbol already registered in the symbol table of the module specified by #n. The parameter of the symbol to be modified is specified on the right-hand side of '='. The meaning of each specification format in the right-hand side is the same as those in Section 5.12.2, 'DEFINE SYMBOL'.

(4) Application example

```
*DEFINE SYMBOL SYM01 (CR) <---The symbol 'SYM01' is defined.
**#8 @IN FIELD 12 LENGTH 4 (CR)
**END (CR)
*
*.#8 SYM01=FIELD 10 (CR) <----- The bit position of 'SYM01'
*                                is modified.
```

5.12.5 address expression= (Modify Memory command)

(1) Input format

```
[#n ][type ] [address expression [{TO address expression}]]
                                 [{LENGTH size expression}]
                                 = value [,...] (CR)
```

(2) Function

This command modifies the content of either memory, a latch, or a register.

(3) Description

The Modify Memory command modifies the content of either memory (LT or FT memory or DM), a latch, or a register, specified by the 'type' in the above format. The content of the specified memory, latch or register is replaced by the the value specified in the right-hand side of '='. The '#n' specifies the module number ($1 \leq n \leq 14$).

The 'type' is specified with one of the symbol types shown in Appendix 2. However, a memory area cannot be specified with 'address expression', 'TO', 'LENGTH', and 'size expression' when the symbol type is either a latch or a register.

The ' TO address expression' is used to specify a memory area with its lower- and upper-bound addresses, and when specifying the memory area with a lower-bound address and a length (size) from it, ' LENGTH size expression' is used.

The 'type' can be omitted only when a symbol is used in 'address expression' (after 'type' in the input format). Additionally, the 'address expression' can be omitted only when 'type' is specified. However, in this case, all the memory area specified by the 'type' must be modified by the 'value' specified in the right-hand side of the expression.

NOTE: 'Type' and 'address expression' specifying an area in the input format cannot be omitted at the same time.

(4) Application examples

```
*#1 LT 10H TO 12H=0000H,0100H,0200H (CR) <---The contents of
*@REFB=0000H (CR)<-------------------+      addresses 10H, 11H, and
*                                    |      12H of the LT memory
                                     |      are modified to 0000H,
                                     |      0100H, and 0200H,
                                     |      respectively.
                                     |
                                     |
The value of the Refresh Base        |
Counter is modified to 0000h.        |
```

5.12.6 SYMBOLS (Display Symbol Table command)

(1) Input format

```
[#n ] { SYMBOLS } (CR)
      { SYM     }
```

(2) Function

This command displays the contents of the symbol table.

(3) Description

The SYMBOLS command displays all the contents of the present symbol table. If the module number '#n' is specified, the contents of the specified module's symbol table are displayed. If the module number is omitted, the contents of the current module's symbol table are displayed.

(4) Application example

```
*DEFIN SYMBOL SYM01 (CR) <----- Symbol 'SYM01' is defined.
**#8 @IN FIELD 12 LENGTH 4 (CR)
**END (CR)
*SYMBOLS (CR) <---- The contents of the symbol table is
# 8 SYMBOL TABLE    displayed.

 SYMBOL  TYPE  VALUE  FIELD  LENGTH  LPTR  SIZE
                      /BASE  /FPTR
SYM01    @IN   0000       12      4
*
```

5.12.7 .symbol (Display Symbol command)

(1) Input format

```
.[#n ] symbol (CR)
```

(2) Function

This command displays the value represented by the symbol specified.

(3) Description

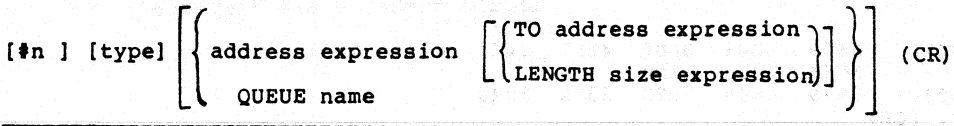
The Display Symbol command displays the value of a specified symbol. If the specified symbol is a QUEUE type, then the DMQ's BASE address, FPTR, LPTR and the SIZE are also displayed. If the module number is omitted, the current module number is assumed.

(4) Application example

```
*.#8 NONA (CR) <----- The value of the symbol 'NONA' in
# 8 SYMBOL TABLE       module number 8 is displayed.

 SYMBOL  TYPE  VALUE  FIELD  LENGTH  LPTR  SIZE
                      /BASE  /FPTR
NONA     @IN   0000       10      4
*
```

5.12.8 Address expression (Display Memory command)

(1) Input format



(2) Function

This command displays the content of memory, a latch, or a register.

(3) Description

The Display Memory command displays the content of the specified memory, a latch, or a register. Since the specification format for the 'type', 'address expression', etc. is the same as that in Modify Memory command, refer to Section 5.12.5.

If a QUEUE name is specified, the effective data of the present QUEUE are displayed.

The 'type' can be omitted only if the 'address expression' (after 'type' in the input format) is specified by a symbol. Additionally, the 'address expression' can be omitted if the 'type' is specified.

NOTE: The 'type' and 'address expression' specifying a memory area in the above format cannot be omitted at the same time.

(4) Application examples

```
*#8 LT 10H TO 20H (CR) <----- The contents of addresses from
# 8 MEMORY DUMP                10H to 20H of the LT memory of
                               module number 8 are displayed.

0010:  6409  4D01  3D00  48F7  24C1  4539  54F9  40CF  58D1
       5C6A  20E1  11BD
001C:  7579  083A  7D22  2892  3949
*; (CR)
*#8 FT 0 TO 10 (CR) <----- The contents of addresses from 0
# 8 MEMORY DUMP              to 10 of the FT memory of module
                             number 8 are displayed.

      FTL  FTR  FTT  FTL  FTR  FTT  FTL  FTR  FTT  FTL  FTR  FTT
0000: 2000 E000 000  0004 D00F 000  2000 E003 302  2000 3030 000
0004: 2018 313F 000  2018 0310 000  2004 01BF 000  2015 30BF 802
0008: 2006 0370 000  201A 02E0 000  0400 0340 000
*; (CR)
*#8 FT 0 LENGTH 4 (CR) <----- LENGTH is used.
# 8 MEMORY DUMP
      FTL  FTR  FTT  FTL  FTR  FTT  FTL  FTR  FTT  FTL  FTR  FTT
0000: 2000 E000 000  0004 D00F 000  2000 E003 302  2000 3030 000
*
```

5.13 Assemble/Disassemble commands

The are two different commands for the on-line assembly and disassembly.

1) { ASSEMBLE / ASM } (On-line Assemble command)

2) { DISASSEMBLE / DASM } (Disassemble command)

5.13.1 ASSEMBLE (On-line Assemble command)

(1) Input format

```
[#n ] { ASSEMBLE } { LT [  address expression]                                  }
      { ASM      } { FT [  address expression]                                  } (CR)
                   { INM file name [ data location expression]                 }

assembly codes (CR)

END (CR)
```

(2) Function

This command assembles the uPD7281 assembly codes and stores them in the specified memory (or file).

(3) Description

The ASSEMBLE command interprets (assembles) specified assembly codes and translates them into the uPD7281 executable code. It then stores the codes in the specified memory (or file) location of the specified module.

The LT memory, FT memory, and the input data token file (INM file) can be the input for the ASSEMBLE command. If the input to the ASSEMBLE command is an input data token, the translated code is stored in the specified input data token file(an INM extension file).

The 'address expression' specifies the location in the LT or FT memory where the translated code is to be stored. If the object is an input data token, the file name and the 'data location expression', that specifies the location where the data is to be stored in, must be specified. Once all parameters are specified, the input data token stored at the 'data location expression' in the specified file is replaced by the translated code.

When the ASSEMBLE command and its subcommands are entered, '**' is displayed. After this, assembly codes are entered, followed by an 'END (CR)' that indicates the end of the command.

NOTE; For the input format of the assembly codes, refer to Appendix 3.

(4) Application examples

```
*DASM LT 0 LENGTH 4 (CR) <----- At first, 4 words beginning
                                 from address 0 of the LT
                                 memory are disassembled and
                                 displayed.

LOC.  OBJECT        LABEL     MNEMONIC

0000  1D61                   PU      002CH   ,0007H   ,0
0001  7C26                   GE      0004H   ,001FH   ,1
0002  5187                   AG&FC   0030H   ,0014H   ,1
0003  0C10                   OUT     0002H   ,0003H   ,0

*#8 ASM LT 0 (CR) <----- The ASSEMBLE command is executed
**PU 0018H,0009H,0 (CR)    from address 0 of the LT memory.
**PU 0027H,0011H,0 (CR)
**PU 001FH,0015H,0 (CR)
**AG&FC 0019H.0010H,1 (CR)
**END (CR) <----- The ASSEMBLE command is terminated.

*#8 DASM LT 0 LENGTH 6 (CR)
LOC.  OBJECT        LABEL     MNEMONIC

0000  24C1                   PU      0018H   ,0009H   ,0
0001  4539                   PU      0027H   ,0011H   ,0
0002  54F9                   PU      001FH   ,0015H   ,0
0003  40CF                   AG&FC   0019H   ,0010H   ,1
0004  3D04                   OUT     0020H   ,000FH   ,1
0005  48F3                   AG&FC   001EH   ,0012H   ,0
                                      ↑
                              The modification is confirmed.
```

5.13.2 DISASSEMBLE (Disassemble command)

(1) Input format

```
[#n ] {DISASSEMBLE}
      {DASM       }

{ LT [ address expression ] [{TO     address expression}]]
                            {LENGTH size expression   }]
{ FT [addressexpression  ] [{TO     address expression}]]
                            {LENGTH size expression   }]
{ INM file name [data location [{TO     data location expression}]]]
                  expression    {LENGTH size expression        }]]
                                                                  } (CR)
```

(2) Function

This command disassembles and displays the content of the specified memory, symbolically.

(3) Description

The DISASSEMBLE command disassembles and displays the content of the specified memory (file) area in a symbolic form. (For the display format, refer to Appendix 3.)

The LT memory, FT memory or an input data token file (INM fipe) can be used as an object for the DISASSEMBLE command.

When a LT or FT memory area is to be specified by a lower-bound address and an upper-bound address, ' TO address expression' is used. Alternately, when a memory area is to be specified with its lower-bound address and the length (size) from this address, ' LENGTH size expression' is used. The same format is used for specifying a portion of an input data token file. However, in this case , 'data location expression' and a file name must be specified.

(4) Application examples

```
*#8 DISASSEMBLE LT 0 TO 10 (CR) <---- The content of the LT
                                      memory from address 0 to
                                      address 10 (0AH) is
                                      disassembled and displayed.

LOC.  OBJECT      LABEL    MNEMONIC

0000  0D61         RDDATA   PU        X31    ,PBG1      ,0
0001  7C26         X43      GE        X39    ,PGEN5     ,1
0002  5187         X44      AG&FC     X42    ,PQ6       ,1
0003  0C10         X45      OUT       X44    ,PEND      ,0
0004  3D04         X39      OUT       X29    ,FWRITE    ,1
0005  48F3         X40      AG&FC     X38    ,PQ2       ,0
0006  5183         X41      AG&FC     X42    ,PQ6       ,0
0007  2C52         X1       GE        X4     ,FW1       ,0
0008  60EB         X2       AG&FC     X46    ,PQ7       ,0
0009  0C14         X3       OUT       X44    ,PEND      ,1
000A  6919         X4       PU        X7     ,FBR       ,0
*; (CR)
*#8 DASM FT 10 LENGTH 6 (CR) <---- Six words beginning from
                                   address 10 (0AH) of the FT
                                   memory are disassembled and
                                   displayed.

LOC.  OBJECT      LABEL      MNEMONIC

000A  0400        FRD2     : COPYM    0001H
000A    0340 000  FRD2     : RDCYCS CNTO      ,0001H
000B  0440        FW1      : COPYM    0002H
000B    1040 000  FW1      : WRCYCS WIBUF     ,0001H
000C  2006        PSHD     : SHR      X ,FULL
        0370 000             RDCYCS SF        ,0001H
000D    1070 000  FW2      : WRCYCS W2BUF     ,0001H
000E  3019        FSUB     : SUB      X ,XCH,FULL
        02D0 000             RDCYCS C15
000F  2805        FWRITE   : OUT2     0005H,FULL
        323F 323             QUEUE  QUE4      ,0010H
*
```

5.14 Macro Function commands

There are eigth different Macro Function commands available for the uPD7281 simulator, as follows.

1) REPEAT (REPEAT command)

2) COUNT (COUNT command)

3) IF (IF command)

4) { MACRO / MAC } (MACRO command)

5) { INCLUDE / INC } (INCLUDE command)

6) WRITE (WRITE command)

7) ECHO (Display Macro Expansion command)

8) NOECHO (Suppress Macro Expansion Display command)

5.14.1 REPEAT command

(1) Input format

```
REPEAT (CR)

[{command                                  } (CR)] [...]
 {WHILE  condition expression              }
 {UNTIL  condition expression              }

END (CR)
```

(2) Function

This command processes the specified sequence of commands repeatedly.

(3) Description

The REPEAT command processes the sequence of commands enclosed within REPEAT and END, repeatedly.

If the 'WHILE condition expression' is specified, the 'WHILE' statement first evaulates the 'condition expression' before executing the statement(s) following the 'WHILE condition expression'. If the 'condition expression' is false (LSB of the expression value = 0), the execution of the statement(s) following the 'WHILE condition expression' ends. However, as long as the 'condition expression' is true (LSB of the expression value = 1), the statement(s) is executed repeatedly.

If 'UNTIL condition expression' is specified, the statement(s) following the 'UNTIL condition expression' is executed repeatedly until the specified condition expression is true.

Once the simulator is in the prompt (*) mode, the execution of the REPEAT command is initiated immediately after the corresponding END statement is entered.

(4) Application example

```
*REPEAT (CR) <----- REPEAT command is entered.
**STEP (CR)
**WHILE @Q AND 01H (CR)
**END (CR)
*STEP <----Just after 'END' is entered, the execution begins.
*WHILE @Q AND 01H
*    <----- As the evaluation of '@Q AND 01H' resulted false,
            the processing ends.
```

5.14.2 COUNT command

(1) Input format

```
COUNT  expression (CR)

[ { command                                  } (CR) ]  [...]
[ { WHILE  condition expression              }      ]
[ { UNTIL  condition expression              }      ]

END (CR)
```

(2) Function

This command executes a sequence of commands repeatedly, the specified number of times.

(3) Description

The COUNT command executes the sequence of commands enclosed within COUNT and END the number of times specified in 'expression', repeatedly.

If the 'WHILE condition expression' is specified, the 'WHILE' statement first evaulates the 'condition expression' before executing the statement(s) following the 'WHILE condition expression'. If the 'condition expression' is false (LSB of the expression value = 0), the execution of the statement(s) following the 'WHILE condition expression' ends. However, as long as the 'condition expression' is true (LSB of the expression value = 1), the statement(s) is executed repeatedly.

If 'UNTIL condition expression' is specified, the statement(s) following the 'UNTIL condition expression' is executed repeatedly until the specified condition expression is true.

Once the simulator is in the prompt (*) mode, the execution of the COUNT command is initiated immediately after the corresponding END statement is entered.

(4) Application example

```
*COUNT 10 (CR)  <----- The COUNT command is entered.  The
**STEP (CR)             number of repetitions is 10.
**UNTIL @Q AND 01H (CR)
**CLOCK (CR)
**END (CR)
*STEP <----- Just after END is entered, execution starts.
*UNTIL @Q AND 01H
*CLOCK
CLOCK = 1131
*END
*STEP              <----- Since the condition is not met,
*UNTIL @Q AND 01H         the execution proceeds.
*CLOCK
CLOCK = 1132
*END
*STEP             <----- Since the condition  expression is
*UNTIL @Q AND 01H      not satisfied, the execution proceeds.
*    <----- Since the condition  expression  is  satisfied,
            the execution is stopped.
```

5.14.3 IF command

(1) Input format

```
IF  condition expression-1 [  THEN](CR)

[command a (CR)] [...]

[ORIF  condition expression-2 [  THEN](CR)                    ]
[[command b(CR) ] [...]                                 [...] ]

[ELSE(CR)                                                     ]
[[command c(CR) ] [...]                                       ]

END (CR)
```

(2) Function

This command controls the command execution flow based on the value of the 'condition expression'.

(3) Description

The IF statement evaluates the specified 'condition expression' and executes the sequence of commands following the specified condition. If the 'condition expression-1' is true (LSB of the expression value = 1), the sequence of 'command a' between the IF statement and either one of the next ORIF, ELSE, or END statement is executed. In this case, the sequence of commands between either the ORIF or the ELSE statement (following the sequence of command-a) and the END statement is ignored.

For the ORIF statement, if the 'condition expression-1' is false (LSB of the expression value = 0) and the condition expression-2 is true, the sequence of 'command b' between the ORIF and the next ORIF, ELSE, or END command is executed. In this case, the sequence of commands between either the ORIF or the ELSE statement (following the sequence of command-b) and the END statement is ignored.

When ELSE statement is specified and all of the condition expressions described by the IF and/or the ORIF statements are false, the sequence of 'command c' between the ELSE and the END statement is executed. Fig. 5-5 shows the processing of the IF command when the command is entered in the following way.

```
-------------------------------------
I IF condition expression-1 THEN(CR) I
I      command a(CR)                 I
I ORIF  condition expression-2(CR)   I
I      command b(CR)                 I
I ELSE(CR)                           I
I      command c(CR)                 I
I END(CR)                            I
-------------------------------------
```

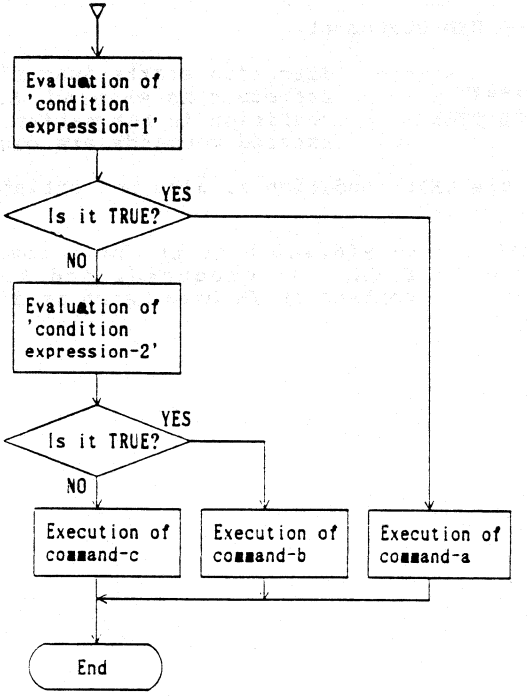


Fig. 5-5 Example of IF command processing

ORIF and ELSE must be specified in the order as shown in the input format. The execution of the IF statement starts immediately after the END command is entered.

(4) Application example

```
*IF @Q AND 04H THEN (CR) <----- IF statement is entered.
**@Q (CR)
**WRITE '****OK****' (CR) <----- (See Section 5.14.6)
**ORIF @OQ AND 01H THEN (CR) <----- ORIF statement
**@OQ (CR)
**ORIF @LT=10H THEN (CR) <----- ORIF statement
**@FT (CR)
**@DM (CR)
**ELSE (CR) <----- ELSE statement
**@IN
**END (CR) <----- END statement

*@Q                  <-----   Execution starts just after END
*WRITE   ****OK****           statement is entered. Since IF
*ORIF @OQ AND 01H THEN        condition is not satisfied, the
*@OQ    <---┐                 entered commands are displayed.
            │
*ELSE       └-- The ORIF condition is also not satisfied.
*@IN
# 8 MEMORY DUMP
@IN    = ******** <----- @IN,which is the next command to
*END                 ELSE, is  executed, and then the
*                    content of IN Data Latch is displayed.
```

5.14.4 MACRO command

(1) Input format

```
DEFINE {MACRO} macro name (CR)                                  .....1)
       {MAC  }
[command (CR)] [. . .]
END (CR)
```

```
:macro name [ actual parameter [, . . .]] (CR)                  .....2)
```

```
{MACRO}
{MAC  } [ macro name [, . . .]] (CR)                            .....3)
```

```
{DIRECTORY} {MACRO}
{DIR      } {MAC  } (CR)                                        .....4)
```

```
NOMACRO [ macro name [, . . .]] (CR)                            .....5)
```

```
PUT file name {MACRO} [[ APPEND]                                .....6)
              {MAC  }
                [ macro name[, . . .]]] (CR)
```

```
GET file name {MACRO} [ macro name [, . . .]] (CR)              .....7)
              {MAC  }
```

(2) Function

This command performs macro definition, reference, display and deletion of commands.

(3) Description

The Macro commands are sub-divided into the following seven types of commands as shown in the input format.

Define Macro
Refer to Macro (Expansion and execution)
Display Macro Body
Display Macro Name
Delete Macro
Save Macro
Load Macro

A higher debugging efficiency can be obtained by an effective use of the macro commands mentioned above.

Each macro command is explained in detail, below.

1) Define Macro

Input format

```
---------------------------------------
  DEFINE {MACRO} macro name (CR)
         {MAC  }

  [command (CR)][...]

  END (CR)
---------------------------------------
```

The Define Macro command is used to define a macro command. It assigns a sequence of the uPD7281 simulator commands to a specified 'macro name' for it's convenient use during a simulation. A sequence of commands represented by a 'macro name' may include certain variable parameters, and these parameters will be referred as 'formal parameters' hereafter.

When 'DEFINE MACRO macro name(CR) ' is entered, '**' is displayed. Then simulator execution commands are entered sequentially, followed by an 'END(CR)' that indicates an end of the command input.

The formal parameters can be coded in a sequence of commands. Up to ten formal parameters can be used, and they are expressed with %0 %9. The use of formal parameters is quite useful when a sequence of commands is used repeatedly with some minor changes in command parameters. During the execution of 'Refer to MACRO command' (to be explained later), the formal parameters are substituted sequentially by the 'actual parameters' specified.

Example:

```
**DEFINE MACRO EXEC1(CR) <---- The macro 'EXEC1' is
                               defined.
**LIST LST:(CR)
**GO %0(CR) <----- A formal parameter }  A sequence
                   is used.           }  of commands
**END(CR)
*
```

2) Refer to Macro

Input format

```
-----------------------------------------------
  :macro name [  actual parameter[,...]](CR)
-----------------------------------------------
```

The Refer to Macro command expands and executes the sequence of commands of the specified macro. To refer to (i.e., to expand and execute) a defined macro, its name must be entered preceded by a ':' (colon) and followed by '(CR)'.

When using the Refer to Macro command, the number of the actual parameters specified must be equal to the number of formal parameters specified during the macro definition stage.

When specifying several actual parameters, each one must be delimited with ',' (comma). When an actual parameter such as '100 STEPS' is composed of more than two character strings, they must be enclosed with a pair of apostrophes.

During the macro command execution, the actual parameters specified are used to substitute the formal parameters, %0, %1, ... , in the same order as they are specified.

Example:

```
*:EXEC1  '100 STEPS' (CR)
```

3) Display Macro Body

Input format

```
---------------------------------------
  {MACRO}
  {    }  [  macro name[,...]] (CR)
  {MAC }
---------------------------------------
```

The Display Macro Body command displays the macro body of the specified macro. When displaying several macro bodies, each one must be delimited with ',' (comma). If the macro name is omitted, all macro bodies are displayed.

Example:

```
*MACRO EXEC1 (CR)
LIST :LP:
GO %0
END
*
```

4) Display Macro Name

Input format

```
-----------------------------
  {DIRECTORY}   {MACRO}
  {DIR      }   {MAC  }
-----------------------------
```

The Display Macro Name command displays all macro names presently defined.

Example:

```
*DIRECTORY MACRO (CR)
EXEC1
EXEC2
TEST
*
```

5) Delete Macro

Input format

```
-------------------------------------
  NOMACRO[  macro name[,...]] (CR)
-------------------------------------
```

The Delete Macro command deletes the macro specified by 'macro name'. If the specification of macro name is omitted, all macros are deleted.

Examples:

```
*NOMACROEXEC1(CR)<--Only macro 'EXEC1' is deleted.
*NOMACRO     <------  All macros are deleted.
*
```

6) Save Macro

Input format

```
PUT  file name  { MACRO }  [[  APPEND][  macro name[,...]]](CR)
                { MAC   }
```

The Save Macro command saves presently defined macro(s) in a specified file. If the specification of 'macro name' is omitted, all defined macros are saved.

If APPEND and macro name are specified, only the specified macro is appended to a existing macro file. In this case, each appending macro name must be delimited by ',' (comma).

Example:

```
*PUT B:MACFIL.MAC MACRO APPEND EXEC1 (CR)
*
```

7) Load Macro

Input format

```
GET  file name  { MACRO }  [  macro name[,...]](CR)
                { MAC   }
```

The Load Macro command loads the specified macro stored in a specified macro file into the development system's memory. If the specification of macro name is omitted, all macros in the macro file are loaded. When loading several macros, each specified 'macro name' must be delimited by ',' (comma).

Macros loaded by this command can be referenced expanded, and executed.

Example:

```
*GET B:CHK1.MAC MACRO EXEC1 (CR)
```

(4) Application examples

```
*DEFINE MACRO CHECK (CR) <---- The macro 'CHECK' is defined.
**#8 @LT (CR)
**#8 @FT (CR)
**#8 @DM (CR)
**#8 @IMRD (CR)
**END (CR)
*:CHECK (CR) <---- The defined macro is executed.
# 8 MEMORY DUMP
@LT       = 2D00000
*#8 @FT
# 8 MEMORY DUMP
@FT       = 0056980000
*#8 @DM
# 8 MEMORY DUMP
@DM       = 01386E00180014
*#8 @IMRD
# 8 MEMORY DUMP
@IMRD     = 00000000
*END
*MACRO CHECK (CR) <---- The content of the macro 'CHECK' is
MACRO CHECK             displayed.
#8 @LT
#8 @FT
#8 @DM
#8 @IMRD
END
*DIR MAC (CR) <-----  Names of presently defined macros are
PARA1                 displayed.
PARA
CHECK
END
*; (CR)
*PUT B:MACFIL.MAC MACRO (CR)<-----  All defined macros are
                                    saved on drive B:.
                                    The file name is
                                    'MACFIL.MAC'.

*NOMACRO (CR) <----- All macros are deleted.
*; (CR)
*DIR MAC (CR) <-- Macro names are specified to be displayed,
END               but since no macro is defined, nothing is
                  displayed.

*GET B:MACFIL.MAC MACRO (CR)<-----  Macros are loaded.
*; (CR)
*DIR MAC (CR)
CHECK
PARA      <----- Three macros have been loaded.
PARA1
END
*DEFINE MACRO EXEC3 (CR) <--- A macro is defined with formal
**#%0 @%1 (CR)                parameters.
```

```
**#%0 @%2 (CR)
**#%0 @%3 (CR)
**END
*; (CR)
*:EXEC3 8,LT,FT,DM (CR) <---Actual parameters(4 parameters)
*#8 @LT                    are specified, and the macro is
# 8 MEMORY DUMP            executed.
@LT       = 2D00000
*#8 @FT
# 8 MEMORY DUMP
@FT       = 0056980000
*#8 @DM
# 8 MEMORY DUMP
@DM       = 01386E00180014
*END
*
```

```
**#%0 @%1 (CR)              parameters.
**#%0 @%2 (CR)
**#%0 @%3 (CR)
**END
*; (CR)
*:EXEC3 8,LT,FT,DM (CR) <---Actual parameters(4 parameters)
*#8 @LT                    are specified, and the macro is
# 8 MEMORY DUMP            executed.
@LT       = 2D00000
*#8 @FT
# 8 MEMORY DUMP
@FT       = 0056980000
*#8 @DM
# 8 MEMORY DUMP
@DM       = 01386E00180014
*END
*
```

5.14.5 INCLUDE command

(1) Input format

```
{INCLUDE}
{INC    }  file name (CR)
```

(2) Function

This command inputs a sequence of commands from the specified file and executes the commands in the order appearance.

(3) Description

The INCLUDE command inputs simulator commands sequentially from the specified file (from an INC extension file already in existence) and executes them. An include file must be created in advance according to the include file format explained in Chapter 3.

The use of an include file is strongly recommended since all system environment parameters can be declared using the file.

(4) Application example

```
*INCLUDE A:FIN5.INC (CR) <---The include file 'AFIN5.INC'
*ENVIRON       <----┐          is input from drive A:.
**MODULE 1          └--- The commands in the file are executed
**MN 8                  sequentially.
**WRITE HIGH          01010001XXXXXXXXB
**WRITE LOW           01010000XXXXXXXXB
**WRITE DATA          01010010XXXXXXXXB
**WRITE COMMAND       01010000XXXXXXXXB
**READ  HIGH          01000111XXXXXXXXB
**READ  LOW           010000XXXXXXXXXXB
**READ  COMMAND       010000XXXXXXXXXXB
**MN FIELD 8 LENGTH 3 OFFSET 12 PADDING 1000B
**ID FIELD 4 LENGTH 4 OFFSET  4 PADDING 0000000B
**DELY 2
**WORK B:
**END
*LOAD OBJECT A:AFFIN4.OBJ
**ALLSYMBOL
**ALLMODULE
**DATA B:AFFIN4.DAT
**END
AFFINE
*DATA INM B:AFFIN4.DAT
*DATA OTM B:AFFIN4.DST
*IM 0 = 1010101010101010B
*IM 1 = 1100110011001100B
*IM 2 = 0111000011110101B
*IM 3 = 1001101010000111B
END
*
*
EXIT INCLUDE <---Indicates the exit from the INCLUDE command.
*
```

5.14.6 WRITE command

(1) Input format

```
WRITE { expression       } [,...] (CR)
      { 'character string'}
```

(2) Function

This command evaluates the specified operand and displays the result, or it writes the literals specified in the 'character string'.

(3) Description

The WRITE command displays either the value resulting from the evaluation of the expression specified as the operand, or the specified character string itself. This command is effective for inputting messages or expression values in a Refer to Macro command or an INCLUDE command which executes simulation commands automatically.

When a string of characters is specified as the operand, it must be enclosed by apostrophes. Moreover, when several expressions or character strings are specified as the operands, each one must be delimited by commas.

(4) Application examples

```
*WRITE ' @PASS= ',FCRD+2 (CR) <--- ' @PASS= ' is  displayed,
@PASS =     0022H               and then 'FCRD+2' is evaluated
*; (CR)                         and  the result is displayed.
*; (CR)                         (FRCD is a symbol.)

*WRITE '*** MODULE #8 DEBUG START ***' (CR)<---The character
*** MODULE #8 DEBUG START ***                  string enclosed
*                                              by apostrophes
                                               is displayed.
```

5.14.7 ECHO (Display Macro Expansion command)

(1) Input format

ECHO (CR)

---

(2) Function

This command displays the sequence of commands on the console during a macro execution time.

(3) Description

After the ECHO command is entered, the sequence of commands, which is expanded when macro function commands (MACRO, IF, REPEAT, COUNT, INCLUDE) are executed, is displayed on the console. (ECHO is assumed by default.)

(4) Application example

```
*ECHO (CR)
*:CHECK (CR) <----- A macro is executed.
*#8 @LT      <-----------------------------  The sequence of commands
# 8 MEMORY DUMP                          |    itself, defined as the
@LT        = 2D00000                     |    macro body, is also
*#8 @FT      <---------------------------|    displayed.
# 8 MEMORY DUMP                          |
@FT        = 0056980000                  |
*#8 @DM      <---------------------------|
# 8 MEMORY DUMP                          |
@DM        = 01386E00180014              |
*#8 @IMRD    <---------------------------|
# 8 MEMORY DUMP
@IMRD      = 00000000
*END
*
```

5.14.8 NOECHO (Suppress Macro Expansion Display command)

(1) Input format

```
NOECHO (CR)
```

(2) Function

This command instructs the simulator not to display the sequence of commands on the console during a macro execution time.

(3) Description

After the NOECHO command is entered, the sequence of commands, which is expanded at macro execution time, is not displayed on the console.

(4) Application example

```
*NOECHO (CR) <----- NOECHO command is entered.
*; (CR)
*:CHECK (CR) <----- The same macro as that in the previous
                    section is executed.
# 8 MEMORY DUMP                  )
@LT       = 2D00000               )
# 8 MEMORY DUMP                  ) Only the results of command
@FT       = 0056980000            ) execution are displayed.
# 8 MEMORY DUMP                  ) (Neither  END is displayed.)
@DM       = 01386E00180014        )
# 8 MEMORY DUMP                  )
@IMRD     = 00000000              )
*
```

5.15 Display Pipeline Clock Step command

The following is the command for displaying pipeline clock count during a simulation.

CLOCK (Display Pipeline Clock Step command)

5.15.1 CLOCK (Display Pipeline Step command)

(1) Input format

```
CLOCK (CR)
```

(2) Function

This command displays the current content of the Pipeline Step Counter.

(3) Description

During a simulation, the number of pipeline clock steps executed so far can be displayed using the CLOCK command. When this command is entered, the content of the Pipeline Step Counter is displayed in decimal digits. The pipeline step counter is initialized when the RESET command is entered.

(4) Application examples

```
*REPEAT (CR)
**STEP (CR)
**PRINT 1 (CR)
**CLOCK (CR) <-----  This example shows a CLOCK command used
**END (CR)           in a REPEAT command.
*STEP       <----- Execution starts.
*PRINT 1
CLOCK   #  8@LTID   # 8@LTC   # 8@LTD
  104   X18          EXEC      001CH
END
*CLOCK
CLOCK =  104 <----- The value of the Pipeline Step Counter
*END                is displayed.
*STEP
*PRINT 1
CLOCK   #  8@LTID   # 8@LTC   # 8@LTD
  105       *          *         *
END
*CLOCK
CLOCK =   105 <---------------------------
*END
*STEP
*PRINT 1
CLOCK   #  8@LTID   # 8@LTC   # 8@LTD
  106   STATTK       EXEC      0000H
END
*CLOCK
CLOCK =   106 <---------------------------
*END
*STEP
```

```
BREAK BY ESC , SIMULATION TO INTERRUPT : CLOCK =    107
*
```

5.16 Set/Display Current Module Number commands

There are two commands to Set/Display Current Module Number, as shown below.

1) #= (Set Current Module Number command)

2) # (Display Current Module Number command)

5.16.1 #= (Set Current Module Number command)

(1) Input format

```
#= expression (CR)
```

(2) Function

This command sets the current module number.

(3) Description

This command sets the value of the expression (a module name is also permitted) specified as the current module number. Whenever the module number is omitted in a simulator command, it is assumed to be the current module number by default.

(4) Application examples

```
*#=8 (CR) <----- The current module number is set to '8'.
*# (CR) <----- Confirmation.   (See Section 5.16.2.)
# = 8
*#=2 (CR) <----- The current module number is set to '2'.
*# (CR)
# = 2
*
```

5.16.2 # (Display Current Module Number command)

(1) Input format

```
# (CR)
```

(2) Function

This command displays the current module number.

(3) Description

The '#' command displays the current module number or default module number. When several uPD7281 modules are in simulation, this command allows the current module to be known.

(4) Application example

```
*# (CR)
# = 2          The current module number is 2.
*
```

5.17 Evaluation commands

There are four evaluation commands, as shown below.

1) LTEVAL (Display LT Operating Ratio command)
2) PUEVAL (Display PU Operating Ratio command)
3) IMEVAL (Display IM Operating Ratio command)
4) NOEVAL (Clear Operating Ratio Counter command)

5.17.1 LTEVAL (Display LT Operating Ratio command)

(1) Input format

[#n]LTEVAL (CR)

---

(2) Function

This command displays the LT operating ratio.

(3) Description

The LTEVAL command displays the following items related to the specified module.

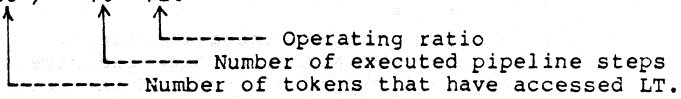
1) The number of simulation pipeline clock steps executed so far. (The value of the Pipeline Step Counter.)

2) The total number of LT tokens entered the LINK TABLE so far. (The value of the LT Operating Ratio Pipeline Step Counter.)

3) The operating ratio that is the ratio between the above two values (in percentage).

(4) Application example

```
*#8 LTEVAL (CR)
# 8 =     50 /    70   71%
*
```



5.17.2 PUEVAL (Display PU Operating Ratio command)

(1) Input format

```
[#n  ] PUEVAL (CR)
```

(2) Function

This command displays the PU operating ratio.

(3) Description

As to the specified module, the PUEVAL command displays the following items:

1) The number of simulation pipeline clock steps executed so far. (The value of the Pipeline Step Counter.)

2) The number of pipeline steps that PU actually operated. (The value of the PU Operating Ratio Pipeline Step Counter.)

3) The operating ratio that is the ratio between the above two values (in percentage).

(4) Application example

```
*#8 PUEVAL (CR)
# 8 =     50 /    70   71%
*
```

- `71%` — Operating ratio.
- `70` — Number of executed pipeline steps.
- `50` — Number of pipeline steps PU has operated.

5.17.3 IMEVAL (Display IM Operating Ratio command)

(1) Input format

```
IMEVAL (CR)
```

(2) Function

This command displays the Image Memory (IM) operating ratio.

(3) Description

The IMEVAL command displays the following items.

1) The number of simulation pipeline clock steps executed so far. (The value of the Pipeline Step Counter.)

2) The number of pipeline steps that the Image Memory has been accessed. (The value of the PU Operating Ratio Pipeline Step Counter.)

3) The operating ratio that is the ratio between the above two values (in percentage).

(4) Application example

```
*IMEVAL (CR)
IM =      0 /    70    0%
*
```

- `0%` — Operating ratio.
- `70` — Number of executed pipeline steps.
- `0` — Number of pipeline steps IM has been accessed.

5.17.4 NOEVAL (Clear Operating Ratio Counter command)

(1) Input format

[#n] NOEVAL (CR)

---

(2) Function

This command clears the values of all Operating Ratio Pipeline Step Counters to 0.

(3) Description

The NOEVAL command resets the values of the LT, PU, and IM Operating Ratio Pipeline Step Counters to 0 (zero).

(4) Application examples

```
*#8 NOEVAL (CR)
*#8 LTEVAL (CR)
# 8 =      0 /    70    0%
*#8 PUEVAL (CR)
# 8 =      0 /    70    0%
*IMEVAL (CR)
IM =       0 /    70    0%
*
```

Since each Pipeline Step Counter is cleared to 0, the operating ratios are 0%.

5.18 Save/Load Simulation Resource commands

There are two commands for saving/loading simulation resources, as shown below.

1) SAVE { RESOURCE / RSRC } (Save Simulation Resource command)

2) LOAD { RESOURCE / RSRC } (Load Simulation Resource command)

5.18.1 SAVE RESOURCE (Save Simulation Resource command)

(1) Input format

```
SAVE { RESOURCE } file name (CR)
     { RSRC     }
```

(2) Function

This command saves the uPD7281 simulation resources into a specified file.

(3) Description

The SAVE RESOURCE command saves all the present uPD7281 simulation information into the specified file.

Although it was stated in Section 5.1.2. that 'simulation cannot be executed without specification of its environment' (ENVIRON command), when the simulation environment (Simulation resource information) is saved by the SAVE RESOURCE command, this environment can be reloaded using the Load Simulation Resource command (to be explained later). Thus, the simulation can be resumed from the same conditions when the simulation resources were saved.

(4) Application examples

```
*SAVE RESOURCE D:IMPP.SVF (CR)
*SAVE RSRC D:IMPP.SVF (CR) <---- Use of the abbreviated
*                                form, 'RSRC'.
```

5.18.2 LOAD RESOURCE (Load Simulation Resource command)

(1) Input format

```
LOAD { RESOURCE } file name [ WORK [d:] [file name]] (CR)
     { RSRC     }
```

(2) Function

This command reloads the simulation resources from the specified file which was created by the SAVE RESOURCE command.

(3) Description

The LOAD RESOURCE command reloads the saved simulation information from the specified file. However, this command can only be used immediatetly after invoking the simulator. ENVIRON command should not be used when the simulation information is loaded by the LOAD RESOURCE command. (If the ENVIRON command is executed, the loaded information is deleted.)

After a LOAD RESOURCE command, files such as INM, OTM, and MAP files, should be loaded prior to the simulation since they are not saved as a part of simulation resources.

The work file must be specified with ' WORK [d:][file name]'. The specification format is the same as that of the WORK subcommand in the ENVIRON command. If this specification is omitted, the same file name as where the simulation resources were saved will be used as the work file.

(4) Application example

```
*LOAD RESOURCE A:IMPP.SVF (CR) <------ Just after the start,
*; (CR)                  the CPU resource information is loaded.
*STATUS (CR) <--- The STATUS command displays the simulation
INM      =          environment.
OTM      =
OBJECT   =
LIST     = LST:
MAP      =
WORK     = B:SM7281.$$0
MACRO    =
MODULE 01H
MN 08H
WRITE HIGH     01010001********B
WRITE LOW      01010000********B
WRITE DATA     01010010********B
WRITE COMMAND  01010000********B
READ HIGH      01000111********B
READ LOW       010000**********B
READ COMMAND   010000**********B
MN FIELD  8 LENGTH  3 OFFSET 12 PADDING 1000B
ID FIELD  4 LENGTH  4 OFFSET  4 PADDING 0000000B
PRIORITY HOST, INM, IM,PASS
*
```

5.19 Set/Cancel Listing Function commands

There are two commands to set/cancel the listing function, as shown below.

1) LIST (Set Listing Function command)

2) NOLIST (Cancel Listing Function command)

5.19.1 LIST (Set Listing Function command)

(1) Input format

```
LIST  file name (CR)
```

(2) Function

This command sets the listing function.

(3) Description

The LIST command outputs all the data (input commands and execution results) displayed on the console to the file specified by 'file name'.

In order to change the file name with this command, enter the LIST command after entering the NOLIST command (to be explained in Section 5.19.2). The default file name is CON:.

(4) Application examples

```
*LIST LST: (CR) <----- The data for  display is also output
*; (CR)                 to the printer.
*NOLIST (CR) <----- (See Section 5.19.2)
*LIST B:CHECK.01 (CR) <---- The data for display is also
*                           ouput to drive B:.  The file
                            name is 'CHECK.01'.
```

5.19.2 NOLIST (Cancel Listing Function command)

(1) Input format

```
NOLIST (CR)
```

(2) Function

This command cancels the listing function.

(3) Description

The NOLIST command cancels the listing function set by the LIST command. After this command is entered, data are displayed only on the console (CON:) until a new LIST command is entered.

(4) Application example

```
*NOLIST (CR)
*
```

5.20 DUMP ASCII File command

The command shown below is used to dump ASCII files to the console.

```
TYPE     (Dump ASCII File command)
```

5.20.1 TYPE (Dump ASCII File command)

(1) Input format

TYPE file name (CR)

---

(2) Function

This command dumps all contents of an ASCII file.

(3) Description

The TYPE command displays all contents of the specified ASCII file on the console. Its use is effective for verification of the contents of source files, include files, input/output files, etc..

(4) Application example

```
*TYPE A:AFFIN4.SRC (CR) <--- The content of AFFIN4.SRC on
MODULE AFFINE = 8 ;          drive A: is displayed.
;*****************************************
;*     AFFINE TRANSFORMATION               *
;*          MAY 11 1984                    *
;*****************************************
;
;
INPUT STATTK, RDDATA AT 0, WRBASE;
OUTPUT READA, WRADD, WRDATA;
;
;
LINK      X1,X2,X3 = FGENE   ( X46    , STATTK )   ;
FUNCTION    FGENE  = COPYBK  ( 1 , 32)  ,CNTGE (4 ) ;
LINK      X4,X5,X6 = FCOPY3( X1                )   ;
FUNCTION    FCOPY3 = COPYM ( 3 , 0 )      ;
LINK      X7       = FBR    ( X4              )   ;
FUNCTION    FBR    = MUL        ,RDCYCS( CNTB ,1) ;
MEMORY      CONTB  = 2000H      ;
LINK        X8     = FCR    ( X7              )   ;
* <---- The display was interrupted by pressing the ESC key.
```

5.21 End Simulation command

The command for terminating a simulation is shown below.

```
EXIT     (End Simulation command)
```

5.21.1 EXIT (End Simulation command)

(1) Input format

```
EXIT [  M] (CR)
```

(2) Function

This command terminates the simulation and returns control to the operating system.

(3) Description

The EXIT command closes all the files presently opened and returns the control to the operating system. Furthermore, 'MAP file' is deleted at that time unless 'M' is specified in the command.

(4) Application example

```
*EXIT (CR)
A>          <----- The operating system's prompt sign, given
                   that the default drive is A:.
```

CHAPTER 6 EXECUTION EXAMPLES

```
B>SM7281 INCLUDE A:MEMTEST.INC

UPD7281 SIMULATOR E1.01 [13 Feb 85]
   Copyright (C) 1985 NEC Corporation

*LIST A:MEMTEST.TRC
*ENVIRON
**MODULE 1
**MN 8
**MAGIC
**RHASEL 0
**CSSEL 0
**DELAY 2
**WORK A:
**END
*LOAD OBJECT A:MEMTEST.LNK
**ALLSYMBOL
**ALLMODULE
**DATA A:MEMTEST.INM
**END
MEMTEST
*STATUS
INM    =
OTM    =
OBJECT = A:MEMTEST.LNK
LIST   = A:MEMTEST.TRC
MAP    =
WORK   = A:SM7281.$$0
MACRO  =
MODULE 01H
MN 08H
MAGIC
RHASEL = 0
CSSEL = 0
PRIORITY HOST, INM,  IM,PASS
*DASM INM A:MEMTEST.INM
LOC.  OBJECT          LABEL      MNEMONIC

----  17000000                   EXEC       0070     , 0000H
----  51000000                   EXEC       0010     , 0000H
----  80700000                   EXEC       0007     , 0000H
*DATA INM A:MEMTEST.INM
*DATA OTM A:MEMTEST.OTM
*MAP MEMORY 500
*BR0=IM WRITE 16 TIMES
*BR1=IM WRITE 4096 TIMES
*BR2=IM READ 16 TIMES
*BR3=IM READ 4096 TIMES
*
EXIT INCLUDE
*DASM LT 0 TO 20H
LOC.  OBJECT          LABEL      MNEMONIC

0000  288D            L20      : PU         L17      ,F9      ,1
0001  1449            L2       : PU         L7       ,F3      ,0
0002  100A            L3       : GE         L2       ,F1      ,0
0003  043B            L4       : AG&PC      L1       ,F12     ,0
0004  2079            L9       : PU         L13      ,F7      ,0
0005  1C22            L10      : GE         L9       ,F5      ,0
0006  085B            L11      : AG&PC      L32      ,F30     ,0
0007  100E            L1       : GE         L2       ,F1      ,1
0008  0869            L6       : PU         L30      ,F30     ,0
0009  1900            L7       : OUT        0020H    ,F4      ,0
000A  1904            L8       : OUT        0020H    ,F4      ,1
000B  1C26            L32      : GE         L9       ,F5      ,1
000C  0408            L12      : OUT        L2       ,F12     ,0
000D  0C59            L30      : PU         L32      ,F31     ,0
000E  0C5D            L31      : PU         L32      ,F31     ,1
```

```
0010  2400          L14     : OUT       L20       ,F8       ,0
0011  0007          L17     : AG&FC     L20       ,F10      ,1
0012  0000          L16     : OUT       L20       ,F10      ,0
0013  ****
0014  ****
0015  ****
0016  ****
0017  ****
0018  ****
0019  ****
001A  ****
001B  ****
001C  ****
001D  ****
001E  ****
001F  ****
0020  ****
*DASM FT 0 TO 10H
LOC.  OBJECT        LABEL     MNEMONIC

0000  0000          F10     : OUT1      0000H
0000    D0FF 000    F10     : COUNT     0000H
0001  0000          F12     : OUT1      0000H
0001    C00F 000    F12     : PICKUP    0010H
0002  081B          F30     : NOP       XX
0002    C00F 000    F30     : PICKUP    0010H
0003  2019          F31     : SUB       X ,FULL
        313F 000              QUEUE  BLOCK3      ,0010H
0004  23C0          F1      : COPYBK    0010H,FULL
        E0FF 000              CNTGE     0000H
0005  081B          F3      : NOP       XX
0006  2805          F4      : OUT2      0005H,FULL
        303F 000              QUEUE  BLOCK1      ,0010H
0007  23C0          F5      : COPYBK    0010H,FULL
        E0FF 000              CNTGE     0000H
0008  081B          F7      : NOP       XX
0009  0001          F8      : OUT1      0001H
000A  22C9          F9      : CMP       X ,BRC,NE ,FULL
        30BF 000              QUEUE  BLOCK2      ,0010H
000B  ********
000C  ********
000D  ********
000E  ********
000F  ********
0010  ********
*STATUS
INM     = A:MEMTEST.INM
OTM     = A:MEMTEST.OTM
OBJECT  = A:MEMTEST.LNK
LIST    = A:MEMTEST.TRC
MAP     =
WORK    = A:SM7281.$$0
MACRO   =
MODULE 01H
MN 08H
MAGIC
RHASEL = 0
CSSEL = 0
PRIORITY HOST, INM,  IM,PASS
```

```
*SYMBOLS
# 8 SYMBOL TABLE

 SYMBOL   TYPE   VALUE  FILED     LENGTH    LPTR      SIZE
                       /BASE      /FPTR

L4        LT     0003        0         0
L6        LT     0008        0         0
L7        LT     0009        0         0
L8        LT     000A        0         0
L9        LT     0004        0         0
STRTDAT   LT     0000        0         0
F10       FT     0000        0         0
F11       FT     0000        0         0
F30       FT     0002        0         0
F12       FT     0001        0         0
F31       FT     0003        0         0
WRITE     FT     0005        0         0
L10       LT     0005        0         0
L20       LT     0000        0         0
L11       LT     0006        0         0
L30       LT     000D        0         0
L12       LT     000C        0         0
L31       LT     000E        0         0
L13       LT     000F        0         0
L32       LT     000B        0         0
L14       LT     0010        0         0
L16       LT     0012        0         0
L17       LT     0011        0         0
F1        FT     0004        0         0
F2        FT     0001        0         0
F3        FT     0005        0         0
F4        FT     0006        0         0
F5        FT     0007        0         0
READ      FT     0001        0         0
F6        FT     0002        0         0
BLOCK1    DM     0006        0         0
L1        LT     0007        0         0
F7        FT     0008        0         0
BLOCK2    DM     0016        0         0
HOST      DM     0000        0         0
L2        LT     0001        0         0
F8        FT     0009        0         0
BLOCK3    DM     0026        0         0
L3        LT     0002        0         0
F9        FT     000A        0         0
*IM 0 TO 0FH
# 8 MEMORY DUMP
0000: ******* ******* ******* ******* ******* ******* ******* *******
0008: ******* ******* ******* ******* ******* ******* ******* *******
*IM 0 TO 0FH = 0000H
*IM 0 TO 0FH
# 8 MEMORY DUMP
0000:  0000  0000  0000  0000  0000  0000  0000  0000
0008:  0000  0000  0000  0000  0000  0000  0000  0000
*BR
BR0 = IM WRITE    16 TIMES
BR1 = IM WRITE  4096 TIMES
BR2 = IM READ     16 TIMES
BR3 = IM READ   4096 TIMES
*BP
BP0 = NOTHING
BP1 = NOTHING
BP2 = NOTHING
BP3 = NOTHING
*BP0=LT 1 ACCE 2 TIMES
```

```
*BP
BP0 = @8 LT 001H                ACCESSED     2 TIMES
BP1 =    10 STEPS
BP2 = NOTHING
BP3 = NOTHING
*GO BR0
BREAK BY BP1 , SIMULATION TO INTERRUPT  ; CLOCK =    10
*PRINT 10
CLOCK   # @@INB    # @@IN     # @@LTID  # @@LTC   # @@LTD     # @@FT          # @@DM          # @@Q               # @@PU
    1       *      ********      *        *         *       **********    **************    ****************    ****************
    2  17000000H ********      *        *         *       **********    **************    ****************    ****************
    3       *      ********      *        *         *       **********    **************    ****************    ****************
    4  51000000H ********      *        *         *       **********    **************    ****************    ****************
    5       *      ********      *        *         *       **********    **************    ****************    ****************
    6  80700000H  00070000H      *        *         *       **********    **************    ****************    ****************
    7       *      00070000H      *        *         *       **********    **************    ****************    ****************
    8       *      ********  L1          EXEC      0000H     **********    **************    ****************    ****************
    9       *      ********      *        *         *       00808900000H  **************    ****************    ****************
   10       *      ********      *        *         *       **********    0202010F000000H   ****************    ****************
END
CLOCK       # @@OQ          # @@PASS   # @@OUTB   # @@BRC  # @@INHC  # @@REFC  # @DQSZ  # @GQSZ  # @OQSZ
    1  ****************    ********      *        0000H     00H       00H       00H       00H       00H
    2  ****************    17000000H  17000000H   0000H     00H       00H       00H       00H       00H
    3  ****************    17000000H      *       0000H     00H       00H       00H       00H       00H
    4  ****************    51000000H  51000000H   0000H     00H       00H       00H       00H       00H
    5  ****************    51000000H      *       0000H     00H       00H       00H       00H       00H
    6  ****************    51000000H      *       0000H     00H       00H       00H       00H       00H
    7  ****************    51000000H      *       0000H     00H       00H       00H       00H       00H
    8  ****************    51000000H      *       0000H     00H       01H       00H       00H       00H
    9  ****************    51000000H      *       0000H     00H       01H       00H       00H       00H
   10  ****************    51000000H      *       0000H     00H       01H       00H       00H       00H
END
*NOBP
*BP
BP0 = NOTHING
BP1 = NOTHING
BP2 = NOTHING
BP3 = NOTHING
*GO BR0
BREAK BY END , SIMULATION TO STOP       ; CLOCK =    94
*IM 0 TO 0FH
# 8 MEMORY DUMP
0000:  0000     0001     0002     0003     0004     0005     0006     0007
0008:  0008     0009     000A     000B     000C     000D     000E     000F
*PRINT 84
CLOCK   # @@INB    # @@IN     # @@LTID  # @@LTC   # @@LTD     # @@FT          # @@DM          # @@Q               # @@PU
   11       *      ********      *        *         *       **********    **************    01013C00000000001H  ****************
   12       *      ********      *        *         *       **********    **************    ****************    ****************
   13       *      ********      *        *         *       **********    **************    ****************    01013C00000000001H
   14       *      ********      *        *         *       **********    **************    ****************    01013C00000000001H
   15       *      ********  L2          EXEC      0000H     **********    **************    ****************    01013C00000000001H
   16       *      ********  L2          EXEC      0001H     00448A0000H   **************    ****************    01013C00000000001H
   17       *      ********  L2          EXEC      0002H     00448A0001H   011209206C0000H    ****************    01013C00000000001H
   18       *      ********  L2          EXEC      0003H     00448A0002H   011209206C0001H    008981B0000000000H  01013C00000000001H
   19       *      ********  L2          EXEC      0004H     00448A0003H   011209206C0002H    008981B0000040000H  01013C00000000001H
   20       *      ********  L2          EXEC      0005H     00448A0004H   011209206C0003H    008981B0000080000H  01013C00000000001H
   21       *      ********  L2          EXEC      0006H     00448A0005H   011209206C0004H    008981B0000C0000H   01013C00000000001H
   22       *      ********  L2          EXEC      0007H     00448A0006H   011209206C0005H    008981B0001000000H  01013C00000000001H
   23       *      ********  L2          EXEC      0008H     00448A0007H   011209206C0006H    008981B0001400000H  01013C00000000001H
   24       *      ********  L2          EXEC      0009H     00448A0008H   011209206C0007H    008981B0001800000H  01013C00000000001H
   25       *      ********  L2          EXEC      000AH     00448A0009H   011209206C0008H    008981B0001C0000H   01013C00000000001H
   26       *      ********  L2          EXEC      000BH     00448A000AH   011209206C0009H    008981B0002000000H  01013C00000000001H
   27       *      ********  L2          EXEC      000CH     00448A000BH   011209206C000AH    008981B0002400000H  01013C00000000001H
   28       *      ********  L2          EXEC      000DH     00448A000CH   011209206C000BH    008981B0002800000H  01013C00000000001H
   29       *      ********  L2          EXEC      000EH     00448A000DH   011209206C000CH    008981B0002C0000H   01013C00000000001H
   30       *      ********  L2          EXEC      000FH     00448A000EH   011209206C000DH    008981B0003000000H  008981B0000000000H
   31       *      ********  L3          EXEC      0010H     00448A000FH   011209206C000EH    00898    0340000H  ****************
```


NEC

32	*	********	L7	EXEC	0000H	0080880010H	011209206C000FH	008981B000380000H	008981B000040000H
33	*	********	L8	EXEC	0000H	00100C0000H	0202010F000010H	008981B0003C0000H	****************
34	*	********	L7	EXEC	0001H	00100D0000H	204006A0140000H	01013C0000400001H	008981B000080000H
35	*	********	L8	EXEC	0001H	00100C0001H	00400660140000H	****************	****************
36	*	********	L7	EXEC	0002H	00100D0001H	204007A0140001H	00208050000000000H	008981B0000C0000H
37	*	********	L8	EXEC	0002H	00100C0002H	00400760140001H	****************	****************
38	*	********	L7	EXEC	0003H	00100D0002H	204008A0140002H	00208050000400001H	008981B000100000H
39	*	********	L8	EXEC	0003H	00100C0003H	00400860140002H	****************	****************
40	*	********	L7	EXEC	0004H	00100D0003H	204009A0140003H	00208050000800002H	008981B000140000H
41	*	********	L8	EXEC	0004H	00100C0004H	00400960140003H	****************	****************
42	*	********	L7	EXEC	0005H	00100D0004H	20400AA0140004H	00208050000C0003H	008981B000180000H
43	*	********	L8	EXEC	0005H	00100C0005H	00400A60140004H	****************	****************
44	*	********	L7	EXEC	0006H	00100D0005H	20400BA0140005H	00208050001000004H	008981B0001C0000H
45	*	********	L8	EXEC	0006H	00100C0006H	00400B60140005H	****************	****************
46	*	********	L7	EXEC	0007H	00100D0006H	20400CA0140006H	00208050001400005H	008981B000200000H
47	*	********	L8	EXEC	0007H	00100C0007H	00400C60140006H	****************	****************
48	*	********	L7	EXEC	0008H	00100D0007H	20400DA0140007H	00208050001800006H	008981B000240000H
49	*	********	L8	EXEC	0008H	00100C0008H	00400D60140007H	****************	****************
50	*	********	L7	EXEC	0009H	00100D0008H	20400EA0140008H	00208050001C0007H	008981B000280000H
51	*	********	L8	EXEC	0009H	00100C0009H	00400E60140008H	****************	****************
52	*	********	L7	EXEC	000AH	00100D0009H	20400FA0140009H	00208050002000008H	008981B0002C0000H
53	*	********	L8	EXEC	000AH	00100C000AH	00400F60140009H	****************	****************
54	*	********	L7	EXEC	000BH	00100D000AH	204010A014000AH	00208050002400009H	008981B000300000H
55	*	********	L8	EXEC	000BH	00100C000BH	004010601400AH	****************	****************
56	*	********	L7	EXEC	000CH	00100D000BH	204011A014000BH	0020805000028000AH	008981B000340000H
57	*	********	L8	EXEC	000CH	00100C000CH	004011601400BH	****************	****************
58	*	********	L7	EXEC	000DH	00100D000CH	204012A014000CH	002080500002C000BH	008981B000380000H
59	*	********	L8	EXEC	000DH	00100C000DH	004012601400CH	****************	****************
60	*	********	L7	EXEC	000EH	00100D000DH	204013A014000DH	0020805000030000CH	008981B0003C0000H
61	*	********	L8	EXEC	000EH	00100C000EH	004013601400DH	****************	****************
62	*	********	L7	EXEC	000FH	00100D000EH	204014A014000EH	0020805000034000DH	****************
63	*	********	L8	EXEC	000FH	00100C000FH	004014601400EH	****************	****************
64	*	********	*	*	*	00100D000FH	204015A014000FH	0020805000038000EH	****************
65	*	********	*	*	*	**********	004015601400FH	****************	****************
66	*	********	*	*	*	**********	**************	002080500003C000FH	****************
67	*	********	*	*	*	**********	**************	****************	****************
68	*	********	*	*	*	**********	**************	****************	****************
69	*	********	*	*	*	**********	**************	****************	****************
70	*	********	*	*	*	**********	**************	****************	****************
71	*	********	*	*	*	**********	**************	****************	****************
72	*	********	*	*	*	**********	**************	****************	****************
73	*	********	*	*	*	**********	**************	****************	****************
74	*	********	*	*	*	**********	**************	****************	****************
75	*	********	*	*	*	**********	**************	****************	01013C0000400001H
76	*	********	*	*	*	**********	**************	****************	01013C0000400001H
77	*	********	L2	EXEC	0010H	**********	**************	****************	01013C0000400001H
78	*	********	L2	EXEC	0011H	00448A0010H	**************	****************	01013C0000400001H
79	*	********	L2	EXEC	0012H	00448A0011H	011209206C0010H	****************	01013C0000400001H
80	*	********	L2	EXEC	0013H	00448A0012H	011209206C0011H	008981B000400003H	01013C0000400001H
81	*	********	L2	EXEC	0014H	00448A0013H	011209206C0012H	008981B000440003H	01013C0000400001H
82	*	********	L2	EXEC	0015H	00448A0014H	011209206C0013H	008981B000480003H	01013C0000400001H
83	*	********	L2	EXEC	0016H	00448A0015H	011209206C0014H	008981B0004C0003H	01013C0000400001H
84	*	********	L2	EXEC	0017H	00448A0016H	011209206C0015H	008981B000500003H	01013C0000400001H
85	*	********	L2	EXEC	0018H	00448A0017H	011209206C0016H	008981B000540003H	01013C0000400001H
86	*	********	L2	EXEC	0019H	00448A0018H	011209206C0017H	008981B000580003H	01013C0000400001H
87	*	********	L2	EXEC	001AH	00448A0019H	011209206C0018H	008981B0005C0003H	01013C0000400001H
88	*	********	L2	EXEC	001BH	00448A001AH	011209206C0019H	008981B000600003H	01013C0000400001H
89	*	********	L2	EXEC	001CH	00448A001BH	011209206C001AH	008981B000640003H	01013C0000400001H
90	*	********	L2	EXEC	001DH	00448A001CH	011209206C001BH	008981B000680003H	01013C0000400001H
91	*	********	L2	EXEC	001EH	00448A001DH	011209206C001CH	008981B0006C0003H	01013C0000400001H
92	*	********	L2	EXEC	001FH	00448A001EH	011209206C001DH	008981B000700003H	****************
93	*	********	L3	EXEC	0020H	00448A001FH	011209206C001EH	008981B000740003H	008981B000400003H
94	*	********	*	*	*	0080880020H	011209206C001FH	008981B000780003H	****************

END

r	8@OQ	8@PASS	8@OUTB	8@BRC	8@INHC	8@BEEC	8DQSZ	8GQSZ	8OQSZ
11	**********	51000000H	*	0000H	00H	01H	00H	00H	00H
12	**********	51000000H	*	0000H	00H	01H	00H	01H	00H
13	**********	51000000H	*	0000H	00H	01H	00H	00H	00H
14	**********	51000000H	*	0000H	00H	01H	00H	00H	00H
15	**********	51000000H	*	0000H	00H	01H	00H	00H	00H
16	**********	51000000H	*	0000H	00H	02H	00H	00H	00H
17	**********	51000000H	*	0000H	00H	02H	00H	00H	00H
18	**********	51000000H	*	0000H	00H	02H	00H	00H	00H
19	**********	51000000H	*	0000H	00H	02H	01H	00H	00H
20	**********	51000000H	*	0000H	00H	02H	02H	00H	00H
21	**********	51000000H	*	0000H	00H	02H	03H	00H	00H
22	**********	51000000H	*	0000H	00H	02H	04H	00H	00H
23	**********	51000000H	*	0000H	00H	02H	05H	00H	00H
24	**********	51000000H	*	0000H	00H	03H	06H	00H	00H
25	**********	51000000H	*	0000H	00H	03H	07H	00H	00H
26	**********	51000000H	*	0000H	00H	03H	08H	00H	00H
27	**********	51000000H	*	0000H	00H	03H	09H	00H	00H
28	**********	51000000H	*	0000H	00H	03H	0AH	00H	00H
29	**********	51000000H	*	0000H	00H	03H	0BH	00H	00H
30	**********	51000000H	*	0000H	00H	03H	0BH	00H	00H
31	**********	51000000H	*	0000H	00H	03H	0CH	00H	00H
32	**********	51000000H	*	0000H	00H	04H	0CH	00H	00H
33	**********	51000000H	*	0000H	00H	04H	0DH	00H	00H
34	**********	51000000H	*	0000H	00H	04H	0DH	00H	00H
35	**********	51000000H	*	0000H	00H	04H	0DH	01H	00H
36	**********	51000000H	*	0000H	00H	04H	0CH	01H	00H
37	**********	51000000H	*	0000H	00H	04H	0DH	01H	00H
38	**********	51000000H	*	0000H	00H	04H	0CH	01H	00H
39	**********	51000000H	*	0000H	00H	04H	0DH	01H	00H
40	**********	51000000H	*	0000H	00H	05H	0CH	01H	00H
41	**********	51000000H	*	0000H	00H	05H	0DH	01H	00H
42	**********	51000000H	*	0000H	00H	05H	0CH	01H	00H
43	**********	51000000H	*	0000H	00H	05H	0DH	01H	00H
44	**********	51000000H	*	0000H	00H	05H	0CH	01H	00H
45	**********	51000000H	*	0000H	00H	05H	0DH	01H	00H
46	**********	51000000H	*	0000H	00H	05H	0CH	01H	00H
47	**********	51000000H	*	0000H	00H	05H	0DH	01H	00H
48	**********	51000000H	*	0000H	00H	06H	0CH	01H	00H
49	**********	51000000H	*	0000H	00H	06H	0DH	01H	00H
50	**********	51000000H	*	0000H	00H	06H	0CH	01H	00H
51	**********	51000000H	*	0000H	00H	06H	0DH	01H	00H
52	**********	51000000H	*	0000H	00H	06H	0CH	01H	00H
53	**********	51000000H	*	0000H	00H	06H	0DH	01H	00H
54	**********	51000000H	*	0000H	00H	06H	0CH	01H	00H
55	**********	51000000H	*	0000H	00H	06H	0DH	01H	00H
56	**********	51000000H	*	0000H	00H	07H	0CH	01H	00H
57	**********	51000000H	*	0000H	00H	07H	0DH	01H	00H
58	**********	51000000H	*	0000H	00H	07H	0CH	01H	00H
59	**********	51000000H	*	0000H	00H	07H	0DH	01H	00H
60	**********	51000000H	*	0000H	00H	07H	0CH	01H	00H
61	00208050000000000H	51000000H	*	0000H	00H	07H	0CH	01H	00H
62	00208050000040001H	51000000H	*	0000H	00H	07H	0DH	01H	02H
63	00208050000080002H	51000000H	52000000H	0000H	00H	07H	0BH	01H	03H
64	00208050000C0003H	51000000H	50000000H	0000H	00H	08H	0AH	01H	04H
65	**********	51000000H	52000001H	0000H	00H	08H	0BH	01H	05H
66	00208050001000004H	51000000H	50000001H	0000H	00H	08H	0AH	01H	04H
67	**********	51000000H	52000002H	0000H	00H	08H	0BH	01H	05H
68	00208050001400005H	51000000H	50000002H	0000H	00H	08H	0AH	01H	04H
69	**********	51000000H	52000003H	0000H	00H	08H	0AH	01H	05H
70	00208050001800006H	51000000H	50000003H	0000H	00H	08H	09H	01H	04H
71	**********	51000000H	52000004H	0000H	00H	08H	09H	01H	05H
72	00208050001C0007H	51000000H	50000004H	0000H	00H	09H	08H	01H	04H

```
 73 ****************  51000000H  52000C     0000H  00H  08H  01H  05H
 74 0020805000200008H  51000000H  5006300C8H  0000H  00H  09H  00H  04H
 75 ****************  51000000H  52000006H  0000H  00H  06H  00H  05H
 76 0020805000240009H  51000000H  50000006H  0000H  00H  09H  00H  04H
 77 ****************  51000000H  52000007H  0000H  00H  06H  00H  05H
 78 0020805000280000AH  51000000H  50000007H  0000H  00H  09H  00H  04H
 79 ****************  51000000H  52000008H  0000H  00H  06H  00H  05H
 80 0020805002C000BH  51000000H  50000008H  0000H  00H  0AH  00H  04H
 81 ****************  51000000H  52000009H  0000H  00H  0AH  00H  05H
 82 0020805000300000CH  51000000H  50000009H  0000H  00H  0AH  00H  04H
 83 ****************  51000000H  5200000AH  0000H  00H  0AH  00H  05H
 84 0020805000340000DH  51000000H  5000000AH  0000H  00H  0AH  00H  04H
 85 ****************  51000000H  5200000BH  0000H  00H  0AH  00H  05H
 86 0020805000380000EH  51000000H  5000000BH  0000H  00H  0AH  00H  04H
 87 ****************  51000000H  5200000CH  0000H  00H  0AH  00H  05H
 88 0020805000300000FH  51000000H  5000000CH  0000H  00H  00H  00H  04H
 89 ****************  51000000H  5200000DH  0000H  00H  00H  00H  05H
 90 ****************  51000000H  5000000DH  0000H  00H  00H  00H  04H
 91 ****************  51000000H  5200000EH  0000H  00H  00H  00H  03H
 92 ****************  51000000H  5000000EH  0000H  00H  00H  00H  02H
 93 ****************  51000000H  5200000FH  0000H  00H  00H  00H  01H
 94 ****************  51000000H  5000000FH  0000H  00H  00H  00H  00H
END
*EXIT

B>
```

APPENDIX 1 LIST OF RESERVED WORDS

1) Commands

ACCE	ACCESSED	ALLM	ALLMODULE	
ALLS	ALLSYMBOL	AND	APPEND	
ASM	ASSEMBLE	BASE	BP	
BP0	BP1	BP2	BP3	
BR	BR0	BR1	BR2	
BR3	CAL	CALCULATE	CLOCK	
CMD	COMMAND	CONT	CONTINUE	
COUNT	CSSEL	DASM	DATA	
DEFINE	DELAY	DIR	DIRECTORY	
DISASSEM	DM	DQ	DQSIZE	
ECHO	ELSE	END	ENVIRON	
EXIT	FIELD	FILE	FPTR	
FT	FTL	FTR	FTT	
GET	GO	GQ	GQSIZE	
GR	HIGH	HOST	ID	
IF	IM	IMEVAL	INC	
INCLUDE	INM	INPUT	LENGTH	
LIST	LOAD	LOW	LPTR	
LT	LTEVAL	MAC	MACRO	
MAGIC	MAP	MEMORY	MN	
MODULE	NOBP	NOBR	NOECHO	
NOEVAL	NOGR	NOLIST	NOMACRO	
NOSYMBOL	NOT	NOTOFF	NOTON	
NOTRACE	NOWOFF	NOWON	NSYM	
NTRC	OBJ	OBJECT	OFFSET	
OQ	OQSIZE	OR	ORIF	
OTM	OUTPUT	PADDING	PASS	
PRINT	PRIO	PRIORITY	PUEVAL	
PUT	QUEUE	READ	RESET	
REPEAT	RESOURCE	PHASEL	RSRC	
SAVE	SHL	SHR	SIZE	
STATUS	STEP	STEPS	SYM	
SYMBOL	SYMBOLIC	SYMBOLS	THEN	
TIMES	TIMING	TO	TOFF	
TON	TRACE	TYPE	UNTIL	
WHILE	WORK	WRITE	XOR	

2) Latches

@DM	@FT	@IDMN	@IMA	
@IMRA	@IMRD	@IMRHA1	@IMRHA2	
@IMRHA3	@IMRHA4	@IMWA	@IMWD	
@IMWD1	@IMWD2	@IMWD3	@IMWD4	
@IMWHA1	@IMWHA2	@IMWHA3	@IMWHA4	
@IN	@INB	@LT	@OQ	
@OUTB	@PASS	@PU	@Q	

3) Registers

I	@BRC	@BRR	@IDR	@INHC	I
I	@INHR	@MNR	@MODE	@REFA	I
I	@REFB	@REFC	@REFR		I

APPENDIX 2 LIST OF SYMBOL TYPES

Specification format types	Meaning
LT	LT symbol
FT	FT symbol
FTL	FT LEFT FIELD symbol
FTR	FT RIGHT FIELD symbol
FTT	FT TEMPORARY FIELD symbol
DM	DM symbol
DQ	DQ symbol
GQ	GQ symbol
OQ	OQ symbol
IM	IM symbol
@LT	LT Input Latch
@FT	FT Input Latch
@DM	DM Input Latch
@Q	Q Input Latch
@OQ	OQ Input Latch
@PU	PU Input Latch
@INB	Input Data Buffer
@OUTB	Output Data Buffer
@IN	IN Data Latch
@PASS	PASS Data Buffer
@IMA	IM Base-Address Latch
@IMRA	IM Read-Address Latch
@IMWA	IM Write-Address Latch
@IMRD	IM Read Data Latch

(cont.)

Specification format types	Meaning
@IMWD	IM Write Data Latch
@IDMN	ID, MN Data Latch
@IMRHA1	IM Read High-order Address Latch-1 (MAGIC mode)
@IMRHA2	IM Read High-order Address Latch-2 (MAGIC mode)
@IMRHA3	IM Read High-order Address Latch-3 (MAGIC mode)
@IMRHA4	IMRead High-order Address Latch-4 (MAGIC mode)
@IMWHA1	IM Write High-order Address Latch-1 (MAGIC mode)
@IMWHA2	IM Write High-order Address Latch-2 (MAGIC mode)
@IMWHA3	IM Write High-order Address Latch-3 (MAGIC mode)
@IMWHA4	IM Write High-order Address Latch-4 (MAGIC mode)
@IMWD1	IM Write Data Latch-1 (MAGIC mode)
@IMWD2	IM Write Data Latch-2 (MAGIC mode)
@IMWD3	IM Write Data Latch-3 (MAGIC mode)
@IMWD4	IM Write Data Latch-4 (MAGIC mode)
@REFB	Refresh Base Counter
@REFA	Refresh Address Counter
@REFR	Refresh Register
@REFC	Refresh Counter
@INHR	Input Inhibit Register
@INHC	Input Inhibit Counter
@MODE	Mode Register

(cont.)

Specification format types	Meaning
@IDR	ID Register
@BRC	Break Counter
@BRR	Break Register
@MNR	Module Number Register
QUEUE	Queue
DQSIZE	DQ size
GQSIZE	GQ size
OQSIZE	OQ size

APPENDIX 3 SPECIFICATION/DISPLAY FORMAT OF ASSEMBLY CODE

(1) LT memory

1. Specification format

{OUT / PU / GE / AG&FC} Destination ID, FT reference address, {0 / 1} ←1)

1) ..... FT Right field Control (FTRC)

2. Display format

XXXX XXXX XXXXXXXX: {OUT / PU / GE / AG&FC} Destination ID,
1) 2) 3)

FT reference address, {0 / 1}

1)..... LT address
2)..... memory contnt (object code)
3)..... symbol name

(2) FT memory

1. Specification format

<1> OUT1 instruction

OUT1 (module number, destination ID [, XCH][, AG&FC instruction]

<2> OUT2 instruction

OUT2 (module number, first destination ID, second destination ID[, XCH][, AG&FC instruction]

<3> PU instructions

Op-code [([output specification][,BRC][,CNOP][,PNZ][,XCH]) [,AG&FC instruction]

(For the Op-code and coding format, refer to (4) List of PU instruction opcodes)

(cont.)

```
I       <4> COPYBK, COPYM instructions                              I
I  COPYBK                                                           I
I  COPYM  (constant1[,XCH]                                          I
I                 [,AG&FC instruction]                              I
I-------------------------------------------------------------------I
I       <5> SETCTL instruction                                      I
I SETCTL(constant1, constant2[,XCH])                                I
I                 [,AG&FC instruction].                             I
I-------------------------------------------------------------------I
I       <6> QUEUE instruction                                       I
I QUEUE(DM expression, size)[,FTT field content]                    I
I-------------------------------------------------------------------I
I       <7> RDCYCS instruction                                      I
I RDCYCS(DM expression, size)[,FTT field content]                   I
I-------------------------------------------------------------------I
I       <8> RDCYCL instruction                                      I
I RDCYCL(DM expression, size)[,FTT field content]                   I
I-------------------------------------------------------------------I
I       <9> WRCYCS instruction                                      I
I WRCYCS(DM expression, size)[,FTT field content]                   I
I-------------------------------------------------------------------I
I       <10> WRCYCL instruction                                     I
I WRCYCL(DM expression, size)[,FTT field content]                   I
I-------------------------------------------------------------------I
I       <11> RDWR instruction                                       I
I RDWR(DM expression)[,FTT field content]                           I
I-------------------------------------------------------------------I
I       <12> RDIDX instruction                                      I
I RDIDX(DM expression)[,FTT field content]                          I
I-------------------------------------------------------------------I
I       <13> PICKUP instruction                                     I
I PICKUP(size)[,FTT field content]                                  I
I-------------------------------------------------------------------I
I       <14> COUNT instruction                                      I
I COUNT(size)[,FTT field content]                                   I
I-------------------------------------------------------------------I
I       <15> CUT instruction                                        I
I CUT(size)[,FTT field content]                                     I
I-------------------------------------------------------------------I
I       <16> CONVC instruction                                      I
I CONVC(size)[,FTT field content]                                   I
I-------------------------------------------------------------------I
I       <17> CNTGE instruction                                      I
I CNTGE(size) [,FTT field content]                                  I
I-------------------------------------------------------------------I
I       <18> DIVCYC instruction                                     I
I DIVCYC(size1, size2)[,FTT field content]                          I
I-------------------------------------------------------------------I
I       <19> DIV instruction                                        I
I DIV(size) [,FTT field content]                                    I
-------------------------------------------------------------------
```

(cont.)

<20> DIST instruction

DIST(size) [,FTT field content]

<21> SAVE instruction

SAVE[,FTT field content]

2. Display format

```
     1)                              2)
XXXX XXXX          XXXXXXXX:    XXXXXXXX

                                   3)                    4)
                                  {XX---------------XX}
                                  {                   } [,FULL]
                                  {[XX,]XXXXH[,XXXXH] }
                                   5)    6)     7)

XXXX   XXXXXXXX   XXXXXXXX:    XXXXXXXX    [XXXXXX][,XXXXH]
8)     9)                       10)          11)       12)
```

1)...The object code of FTL field is printed.

2)...When the instruction is either an OUT, a GE, or a PU instruction, its name is printed.

3)...When the instruction in MNEMONIC is a GE instruction, the generation size is printed, and when it is a PU instruction, the output specification, XCH, PNZ, (EQ, NE, ...), and CNOP are printed. (Default values are not printed, however.)

4)...'FULL' is printed when F/L flag in FTL field is '1'.

5)...The module number of output destination is printed, in the case of an OUT instruction.

6)...The destination ID is printed.

7)...The destination ID of the second output (in the case of two outputs) is printed.

8)...The address of FTR, FTT fields is printed (blank in the case of packed format.)

9)...The object code for the FTR and FTT fields are printed.

10)...The instruction name is printed when the instruction is an AG&FC instruction.

11)...DM address is printed.

12)...SIZE is printed.

(3) Input Data Tokens

1. Specification format
<1> SETLT SETLT module number, LT address, LT write data
<2> SETFTL SETFTL module number, FT address, FTL write data
<3> SETFTR SETFTR module number, FT address, FTR write data
<4> SETFTT SETFTT module number, FT address, FTT write data
<5> RDLT RDLT module number, LT address
<6> RDFTL, RDFTR, RDFTT {RDFTL / RDFTR / RDFTT} module number, FT address
<7> CRESET CRESET module number
<8> SETMD SETMD module number, constant, Referesh Register data, Input Inhibit Register data
<9> SETBRK SETBRK module number, ID, constant, count data
<10> DUMP DUMP module number, dump mode
<11> CBRK CBRK
<12> VAN VAN
<13> PASS PASS module number, ID, data
<14> EXEC EXEC module number, ID, data
<15> DMS DMS module number, ID, first address to DM, data list
<16> DMR DMR module number, ID, first address to DM, number of reads

2. Display format

```
 ----  XXXXXXXX      XXXX   XXXXXXXX,XXXXXX
       1)            2)     3)        4)

1)... Data (32 bits)
2)... Mnemonic code
3)... Destination ID (dump mode in the case of DUMP
       instruction)
4)... Data to be set
```

(4) List of op-codes (operation codes) of PU instructions

Classification	Specification format	Function
Logical operations	AND	Logical AND
	OR	Logical OR
	XOR	Logical EXCLUSIVE-OR
	ANDNOT	Logical $\overline{A} \cdot B$
	NOT	Negation
Arithmetic operations	ADD	Addition
	SUB	Subtraction
	MUL	Multiplication
	NOP	No operation
	ADDSC	Addition and shift count
	SUBSC	Subtraction and shift count
	MULSC	Multiplication and shift count
	NOPSC	No OP and Shift count
	INC	Increment
	DEC	Decrement
Shift operations	SHL	Shift left
	SHLBRV	Shift left with bit reverse
	SHR	Shift right
	SHRBRV	Shift right with bit reverse
Comparison operations	CMPNOM	Compare normalized
	CMP	Compare
	CMPXCH	Compare and exchange

Classification	Specification format	Function
Accumulation	ACC	Accumulate
Copy C-bit	COPYC	Copy C-bit
Double precision adjust	ADJL	Adjust input data sign
Bit manipulation	GET1	Get 1 bit
	SET1	Set 1 bit
	CLR1	Clear 1 bit
Data Exchange	CVT2AB	Convert 2's complement to sign-magnitude
	CVTAB2	Convert sign-magnitude to 2's complement
Bit Check	ORMSK	Mask a word with logical OR
	ANDMSK	Mask a word with logical AND

APPENDIX 4 LIST OF ERROR MESSAGES OF THE SIMULATOR

Error display format: *** ERROR error number error message

Error number F001	Message	ILLEGAL FILE SPECIFICATION - file name
	Cause	File name is coded incorrectly.
	Program handling	Returns control to command input mode.
	User's procedure	Specify correct file name and enter the command.
Error number F002	Message	CONFLITING FILE SPECIFICATION - file name
	Cause	There is a conflict between either input file name and output file name, among input file names, or among output file names.
	Program handling	Returns control to command input mode.
	User's procedure	Specify correct file name(s) and enter the command.
Error number F003	Message	FILE NOT FOUND - file name
	Cause	File specified by file name does not exist.
	Program handling	Returns control to command input mode.
	User's procedure	Specify the correct file name and enter the command.
Error number F004	Message	WRITE PROTECTED FILE - file name
	Cause	The output file is write-protected.
	Program handling	Returns control to command input mode.
	User's procedure	Either remove the write protection or specify another file name and then enter the command.

Error Number	Message	ILLEGAL COMMAND
F005	Cause	Command is coded incorrectly.
	Program handling	Returns control to command input mode.
	User's procedure	Enter the correct command.
Error Number	Message	COMMAND SYNTAX ERROR
F006	Cause	Operand of the command is coded incorrectly.
	Program handling	Returns control to command input mode.
	User's procedure	Specify the correct operand and enter the command.
Error Number	Message	INVALID NUMERIC
F007	Cause	Number specified incorrectly in operand of the command.
	Program handling	Returns control to command input mode.
	User's procedure	Specify correct number in the operand and enter the command.
Error Number	Message	SYMBOL NOT FOUND
F008	Cause	Symbol specified as an operand of the command does not exist.
	Program handling	Returns control to command input mode.
	User's procedure	Specify correct symbol to the operand and enter the command.

Error number	Message	MACRO NOT FOUND
F009	Cause	Macro specified as the operand of the command does not exist.
	Program handling	Returns control to command input mode.
	User's procedure	Specify correct macro in the operand and enter the command.
Error number	Message	ILLEGAL COMMAND SEQUENCE
F010	Cause	Command entered in incorrect order.
	Program handling	Returns control to command input mode.
	User's procedure	Enter the command in the correct order.
Error number	Message	SYMBOL DOUBLY DEFINED
F011	Cause	The same symbol is defined more than once.
	Program handling	Returns control to command input mode.
	User's procedure	Specify a different symbol name and enter the command.
Error number	Message	MACRO DOUBLY DEFINED
F012	Cause	The same macro name is defined more than once.
	Program handling	Returns control to command input mode.
	User's procedure	Specify a different macro name and enter the command.
Error number	Message	NO SYSTEM GENERATION
F013	Cause	Simulation environment is not preset.
	Program handling	Breaks execution and returns control to command input mode.
	User's procedure	Set the simulation environment.

Error number F014	Message	DATA SPECIFIED AT INHIBITED FIELD IN TOKEN NAME
	Cause	Value that exists inhibited field in token is set
	Program handling	Returns control to command input made
	User's procedure	Confirm token configuration and correct data
Error number F015	Message	INVALID VALUE IN TOKEN NAME
	Cause	Token value is out of available area
	Program handling	Returns control to command input mode
	User's procedure	Confirm available area of each field value and correct data

Error number	Message	TOO MANY OPERANDS,TRUNCATED
W021	Cause	There are too many operands.
	Program handling	Assumes that the permitted maximum number of operands have been specified and continues processing.
	User's procedure	Specify the ignored operand and reenter the command.
Error number	Message	COMMAND DOUBLY INPUT
W022	Cause	The same command has entered overlapped.
	Program handling	The latter command is validated.
	User's procedure	Nothing.
Error number	Message	HIGH/LOW DATA CHECK ERROR
F101	Cause	HIGH/LOW error detected during HIGH/LOW data check mode.
	Program handling	Breaks execution and returns control to command input mode.
	User's procedure	Correct the source data and simulate again.
Error number	Message	DMQ OVERFLOW ERROR #n FT-ADDR
F102	Cause	DMQ overflow
	Program handling	Breaks execution and returns control to command input mode.
	User's procedure	Either correct the source module and recreate the object module by assembling or correct the object module by using assembly commands.

Error number F103	Message	GQ OVERFLOW ERROR #n
	Cause	GQ overflow
	Program handling	Breaks execution and returns control to command input mode.
	User's procedure	Either correct the source module and recreate the object module by assembling or correct the object module by using assembly commands.
Error number F104	Message	DQ OVERFLOW ERROR #n
	Cause	DQ overflow
	Program handling	Breaks execution and returns control to command input mode.
	User's procedure	Either correct the source module and recreate the object module by assembling or correct the object module by using assembly commands.
Error number F105	Message	ILLEGAL EXECUTE INSTRUCTION #n @XX
	Cause	Incorrect PU instruction has been executed.
	Program handling	Breaks execution and returns control to command input mode.
	User's procedure	Either correct the source module and recreate the object module by assembling or correct the object module by using assembly commands.
Error number F107	Message	LT BREAK #n LT-ADDR
	Cause	LT break condition was satisfied in test mode.
	Program handling	Breaks execution and returns control to command input mode.
	User's procedure	Either correct the source module and recreate the object module by assembling or correct the object module by using assembly commands.

Error number	Message	READ FROM UNUSED AREA #n ADDR
F108	Cause	Area with no data was referenced.
	Program handling	Breaks execution and returns control to command input mode.
	User's procedure	Nothing.
Error number	Message	WORKING TABLE SPACE EXHAUSTED
A901	Cause	Not enough free space for symbol table
	Program handling	Returns control to command input mode.
	User's procedure	Reduce definitions of macros and symbols.
Error number	Message	INVALID FILE SYNTAX - file name
A902	Cause	File incorrectly formatted.
	Program handling	Returns control to command input mode.
	User's procedure	Recreate the file.
Error number	Message	FATAL I/O ERROR - file name
A903	Cause	1)File with physical damage. 2)Simulator's overlay file not present on the disk drive holding the activated file. 3)No free area in the directory of the disk specified for output. 4)Free area exhausted on output-proceeding disk drive.
	Program handling	Returns control to command input mode.
	User's procedure	1)Recreate the file. 2)Change the file's medium. 3)Set the overlay file on the disk drive holding the activation file. 4)Specify (change) output to a disk with sufficient free area.

PART IV

USAGE OF THE OBJECT-CODE CONVERSION PROGRAM

CHAPTER 1 OUTLINE OF THE SOFTWARE

1.1 Outline of the Functions of the Object-code Conversion Program

The Object-code Conversion Program reads an object module produced by the assembler and converts it into either a HEX-format or an ASCII-data-format object module. The flow of operations from the assembly to the object code conversion is shown in Fig. 1-1.

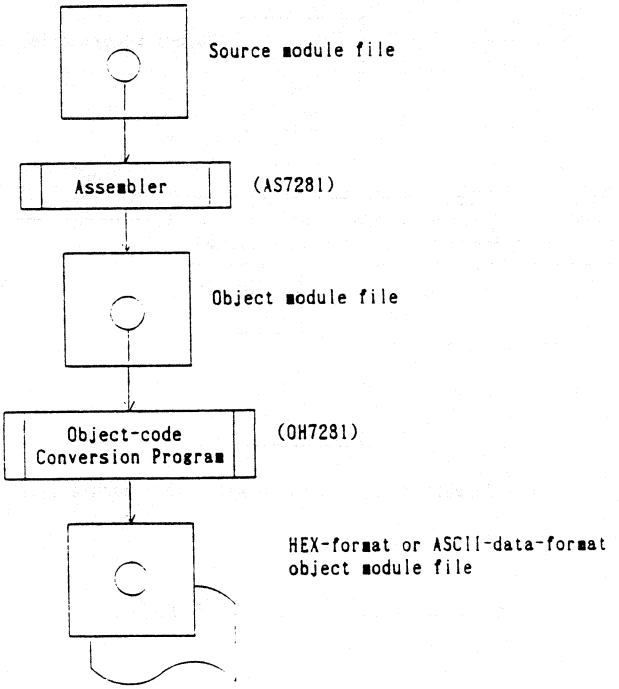


Fig. 1-1 Flow of operations from assembly to object conversion

1.2 Files handled by the Object-code Conversion Program and their relationships

The Object-code Conversion Program also creates a symbol table module file in addition to the HEX-format or ASCII-data format object module file mentioned above. For creating the symbol table module file, 'DEBUG' control must be specified at the start of conversion. For details, refer to Section 3.2.

The contents of the symbol table module file are the basic symbol information. These files and their relationship is shown in Fig. 1-2.

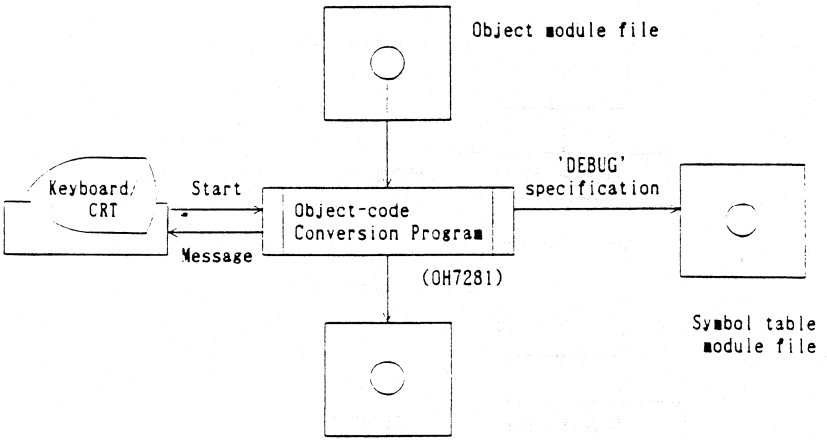


Fig. 1-2 Organization of files

CHAPTER 2 INPUT/OUTPUT FILES

This chapter explains input/output files handled by the Object-code Conversion Program.

2.1 Types of files

There are four types input/output files handled by the Object-code Conversion Program, as listed below.

1) Object module file (input)
2) HEX-format object module file (output)
3) ASCII-data-format object module file (output)
4) Symbol table module file (output)

Classification between input and output, and usable storage medium for each file are shown in Table 2-1.

Table 2-1 Input/output classification and storage medium

File Type	Input/Output	Medium
Object module file	input	FD/HD
HEX-format object module file	output	FD/HD/PT
ASCII-data-format object module file	output	FD/HD/PT
Symbol table module file	output	FD/HD/PT

FD; floppy disk HD; hard disk
PT; paper tape

2.2 File formats

The following explains the file formats and other detailed information about each file.

2.2.1 Object module file

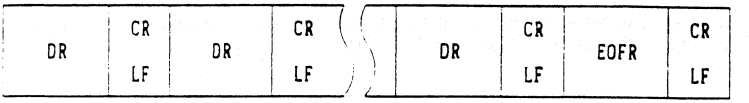
This is a binary file which is output by the assembler as an object file.

2.2.2 HEX-format object module file

The Object-code Conversion Program accepts a binary file (the object file produced by the assembler) and outputs a HEX-format object module file. This particular HEX-format object file is produced when the Object-code Conversion

Program is invoked with a command containing TYPE(1) or TYPE(2) control specification. (For more information on the control specifications, refer to Section 3.2.)

The structure of the HEX-format object module file is shown in Fig. 2-1.



DR ;data record
EOFR ;last record

Fig. 2-1 Structure of HEX-format object module file

(1) The organization of records

: XX XXXX XX DD.....D SS
a b c d e f

a)...Record mark. It indicates the beginning of the record.
b)...Represents the number of bytes in the object data. The last record of the file has the value of '00'.
c)...Represents the address of the first object data in the record. In the last record, its value is '0000'. The start address of the module can be designated with 'ADDRESS' control. (For more information on the control specifications, refer to Section 3.2.)
d)...Record type. '00' identifies a data record and '01' identifies the last record.
e)...This portion of a record contains the actual object code data. This part does not exist in the last record.
f)...Check sum. (See NOTE.)

NOTE: Check sum:

Check sum is the value obtained from successive subtractions of each piece of data except the record mark (:) from the preceding result, starting from the initial value of '00H'. The value of the check sum expressed is in two hexadecimal digits.

How check sum is calculated

To show the procedures for calculating check sum, the following HEX-format object module record is used as an example. The subtraction is assumed to be carried out in modulo-256 form.

:100780003CAFCD3A07.....FF8B

1. The value '10H' representing the number of bytes is subtracted from the initial value '00H'.

```
    0000 0000
-)  0001 0000
 -----------
    1111 0000
```

2. The value '07H' representing the high-order 8 bits of the address is subtracted from the above result.

```
    1111 0000
-)  0000 0111
 -----------
    1110 1001
```

3. The value '80H' representing the low-order 8 bits of the address is subtracted from the preceding result.

```
    1110 1001
-)  1000 0000
 -----------
    0110 1001
```

4. The same procedure repeats up to the last data 'FFH'.

5. The result from the subtraction of the last data 'FFH' is the check sum. (In this example, it is 8BH.)

(2) The contents of the object data

There are two types of formats for the object data (decribed as the item 5 of 'Organization of Records') in a HEX-format object module file.

a. Format in which the object code for the Execution Section per each uPD7281 module is separated into individual object code modules.

b. Format in which the object code for the Execution Section per each uPD7281 module is not separated into individual object code modules.

These two formats are explained below.

1) Format in which the object code for the Execution Section is separated into modules.

When 'TYPE(1)' control is specified, the converted object code is seperated into as many object code data modules as the number of the Execution Sections included in the original source program.

The internal organization of the converted object data is shown in Fig. 2-2.

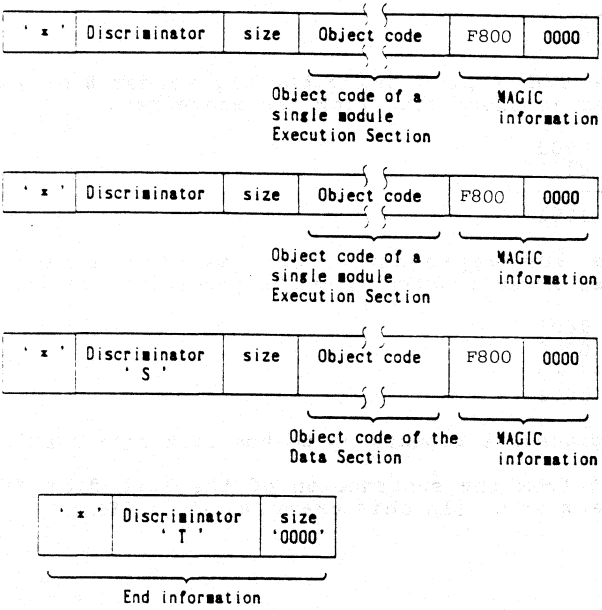


Fig. 2-2

- o '*' .... Beginning mark. Indicates the beginning of an object code record for an Execution Section module.

- o discriminator .... Discriminates the object code.
  '0'-'F'; One of the 16 hexadecimal digits in ASCII form is used as a discriminator for the Execution Section's object code. The indicated ASCII number is the module number for a uPD7281 of which the Execution Section's object code is included in the record.

  'S' ; Indicates that the object code is for the Data Section.

  'T' ; Indicates the end object code record.

- o size .... Represents the number of bytes in the object code record. This value is '0000' in the last object code record.

- o object code .... This is the object code itself.

- o MAGIC information ... Added when 'MAGIC' control is specified. For details of this control, refer to Section 3.2.6.

Examples:

```
                Beginning mark (*)
                Discriminator              Object code
                Size  (module               Check sum
                       number 8)

:100000002A38046487FCFC0383FD900083FF000012
:1000100087F0000087F0000187F0000083FD90006A
:1000200083FF000487F0000183FD900083FF00063A
:1000300087F0000087F0002087F0000083FD90002B
:1000400083FF000A87F0000083FD90FF83FF000C10
:1000500087F0000087F0002087F0000083FD90000B
:1000600083FF001087F0000083FD900083FF0012E3
                    :
                    :
                    :
```

2) The format in which the object code of the Execution Section is lumped into one object code module.

When 'TYPE(2)' control is specified, the object data will consist of one lumped object code for all modules in the Execution Section in a sequential format without separations. The contents of this object data is shown in Fig. 2-3.

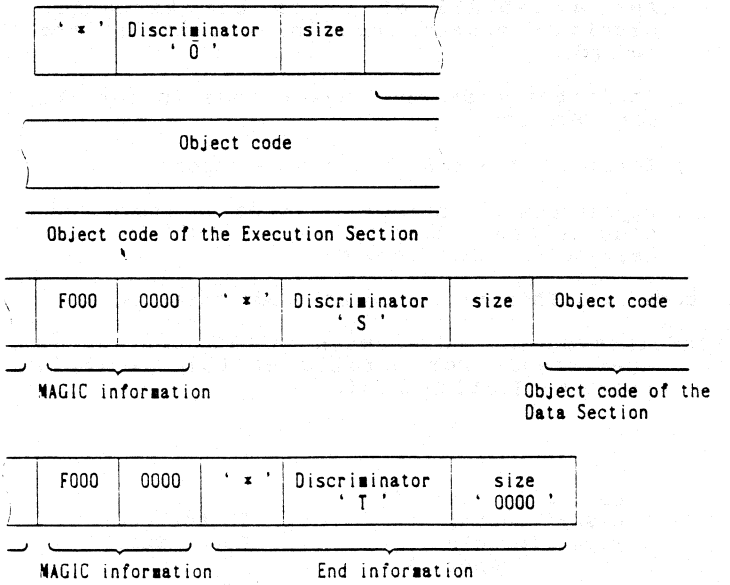


Fig. 2-3

o '*' .... Beginning mark. It indicates the beginning of an object code record for the entire Execution Section module.

o discriminator .... Discriminates the object code.
'O' ; A discriminator for object code for the Execution Sections.

'S' ; Indicates that the object code is for the Data Section.

'T' ; Indicates the end object code record.

- o size .... Represents the number of bytes in the object code record. This value is '0000' in the last object code record.
- o object code .... This is the object code itself.
- o MAGIC information ... Added when 'MAGIC' control is specified. For details of this control, refer to Section 3.2.6.

Examples:

```
:100000002A4F046487FCFC0383FD900083FF00001A
:1000100087F0000087F0000187F0000083FD90006A
:1000200083FF000487F0000183FD900083FF00063A
:1000300087F0000087F0002087F0000083FD90002B
:1000400083FF000A87F0000083FD90FF83FF000C10
:1000500087F0000087F0002087F0000083FD90000B
:1000600083FF001087F0000083FD900083FF0012E3
:1000700087F0000083FD900083FF001487F00000EC
                    :
                    :
                    :
```

NOTE; For explanations about the format, refer to page 4-8.

2.2.3 ASCII-data-format object module files

The Object-code Conversion Program is also capable of outputting an ASCII-data-format object module file when it is invoked using a command with the 'TYPE(3)' control. (For the control specifications, refer to Section 3.2.)

The structure of the ASCII-data-format object module file is shown in Fig. 2-4.

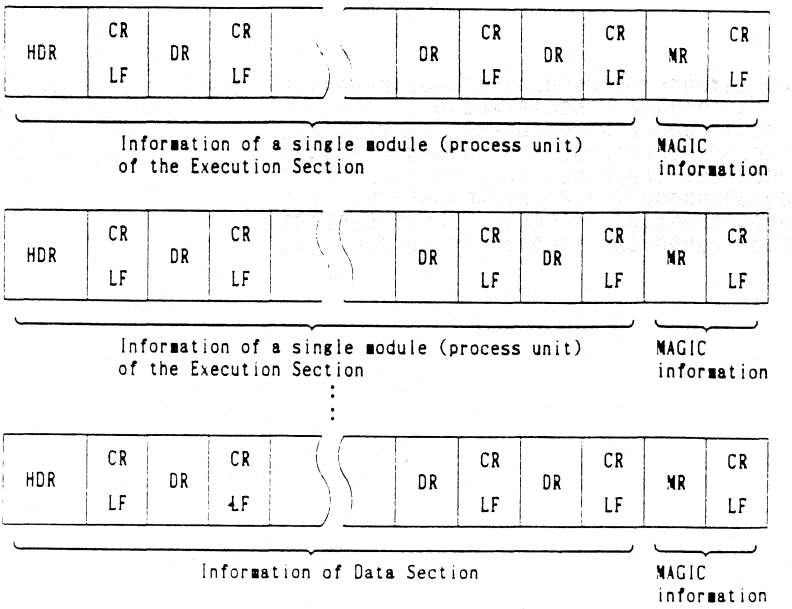


Fig. 2-4 Structure of ASCII-data-format object module

(1) Organization of the record

1) Header record

The internal organization of the header record differs depending upon whether the subsequent DR (data record) is for the Execution Section or the Data Section. The header records are shown in Fig. 2-5.

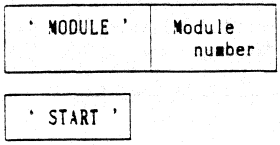


Fig. 2-5

a) MODULE

The character string 'MODULE' is used as the header information for the execution section's object code.

b) Module number

The module number is in two digit decimal form. The module number indicates that the following data records are for a particular uPD7281 with the specified module number.

c) START

The character string 'START' is used as the header information for the data section's object code.

2) Data record

The internal organization of a data record is shown in Fig. 2-6.

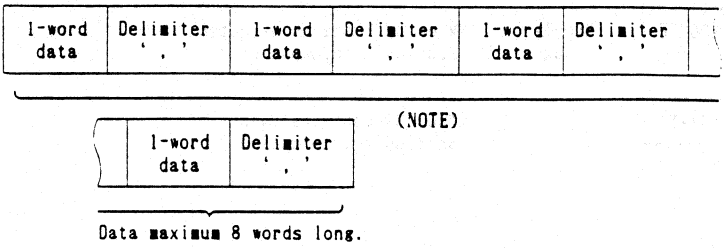


Fig. 2-6

a) 1-word data

The content of a 1-word data (16-bit data) is shown in the following format.

0XXXXH (6 ASCII characters)

1-word data (16 bits) is represented with 4 hexadecimal digits.

b) Delimiters (',')

Represents a delimiter between the two 1-word data.

NOTE: Delimiter is not added after the last 1-word data of the last data record of the module.

3) MAGIC information record

Added when 'MAGIC' control is specified. The content of the record is '0F000H, 00000H'.

Examples:

```
MODULE 08
087FCH,0FC03H,083FDH,09000H,083FFH,00000H,087F0H,00000H,
087F0H,00001H,087F0H,00000H,083FDH,09000H,083FFH,00004H,
087F0H,00001H,083FDH,09000H,083FFH,00006H,087F0H,00000H,
087F0H,00020H,087F0H,00000H,083FDH,09000H,083FFH,0000AH,
087F0H,00000H,083FDH,090FFH,083FFH,00000H,087F0H,00000H,
087F0H,00020H,087F0H,00000H,083FDH,09000H,083FFH,00010H,
087F0H,00000H,083FDH,09000H,083FFH,00012H,087F0H,00000H,
083FDH,09000H,083FFH,00014H,087F0H,00000H,083FDH,0900FH,
                              :
                              :
```

2.2.4 Symbol table module file

The Object-code Conversion Program outputs a symbol table file when it is invoked with 'DEBUG' or 'DB' control. (For more information on the control specifications, refer to Section 3.2.)

The organization of the symbol table file is shown in Fig. 2-7.

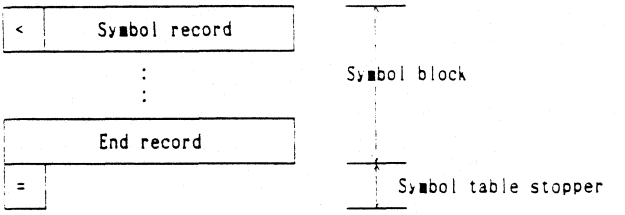


Fig. 2-7

(1) Symbol block

A symbol block begins with a symbol block header (<), followed by a set of symbol records. Furthermore,

symbol records are sorted in the ASCII-code order using the symbol name. The end record is appended at the end of the symbol block.

(2) Symbol table stopper

A symbol table stopper is appended at the end of the symbol table.

CHAPTER 3 OPERATION PROCEDURES OF THE OBJECT-CODE CONVERSION PROGRAM

This chapter explaines the operational procedures for the Object-code Conversion Program in detail.

3.1 Operating Procedures of the Object-code Conversion Program

3.1.1 How to invoke the Object-code Conversion Program

(1) Start

After the operating system's prompt sign, the command to invoke the Object-code Conversion Program may be entered. The command input format to start the coverter program is shown below.

```
------------------------------------------------------------
A> [d:]OH7281  input object module file name
  [TO output object module file name] [control list](CR)
------------------------------------------------------------
```

A> ..... This is the operating system's prompt sign. In this case, the default disk drive is A:.

[ ] .... This means that the enclosed item is optional.

d: .... This specifies the disk drive name which holds the OH7281 program. (When it is omitted, the current default disk drive is assumed.)

Input object module file name
.... This specifies the object code file output by the assembler.

Output object module file name
.... This specifies the output file name which will hold the converted object code. Besides a disk file, either a paper-tape pucher or a printer may be specified as an output device. (For more detail, refer to Section 3.3.)

Control list
.... This specifies whether or not various information is to be output after an execution of the Object-code Conversion Program. (For more information, refer to Section 3.2.)

Example:

A> B:OH7281 EXAMPL.LNK TO E:AFIN.HEX DEBUG (CR)

When the Object-code Conversion Program is invoked, the following message is displayed on the console.

```
-------------------------------------------
  UPD7281 OBJECT CONVERTER Vx.y [dd Mmm YY]
       Copyright (C) YYYY NEC Corporation
-------------------------------------------
```

Vx.y	Version number of the Object-code Conversion Program.
dd Mmm YY	The date when the Object-code Conversion Program is created.
YYYY	The year when the Object-code Conversion Program is created.

(2) Specification of the continuation line at the time of the program invocation

Up to 80 characters (including the CR and LF) can be entered in a line for the Object Converter invocation command. If more than 80 characters are needed to specify the list control parameters, it can be continued in the next line (continuation line) by writing '&(CR)' at the end of the line to be continued. As the '&(CR)' is entered, '**' is displayed on the next line and subsequent control parameters may be entered.

Additionally, the Object Converter ignores any string of characters between '&' and '(CR)', which are used for specifying continuation line. Therefore, a comment may be written in between '&' and "(CR)'.

Example:

```
A> OH7281 B:CONV01.LNK TO B:CONV02.HEX DEBUG &COMMENT (CR)

UPD7281 OBJECT CONVERTER Vx.y [dd Mmm YY]
   Copyright (C) YYYY NEC Corporation

** ADDRESS(1000H) (CR)
```

3.1.2 The termination of the Object-code Conversion Program

After the Object-code Conversion Program terminates, the following message is displayed on the console, and the control is returned to the operating system.

(1) In the case of a normal termination, the following messages are displayed.

```
-------------------------------------------------
  OBJECT FILE = input object module file name
  HEXA   FILE = output object module file name
  [SYMBOL FILE = symbol table module file name] *
  OBJECT CONVERT COMPLETE
-------------------------------------------------
```

(*) This message is displayed only if the 'DEBUG' control is specified.

(2) In the case of an abnormal termination, the program terminates with the following message displayed on the console.

```
----------------------
  PROGRAM TERMINATED
----------------------
```

3.2 Control Specification

There are six different controls which can be specified in the 'control list' mentioned above.

These controls are entered in abbreviated form during the Object Converter invocation. The controls are used to specify whether or not various information are to be output after a Object-code Conversion Program execution.

The list of control names, their abbreviated forms and default conditions are shown in Table 3-1.

Table 3-1 List of controls

Class	Control name	Abbreviated form	Default
Module information output specification	MODULE NOMODULE	MD NOMD	MODULE
Data information output specification	DATA NODATA	DT NODT	DATA
Symbol table module output specification	DEBUG [(fn)] NODEBUG	DB [(fn)] NODB	NODEBUG
Object module output format specification	TYPE [(n)]	TP [(n)]	TYPE(1)
HEX-format object module start address specification	ADDRESS [(sa)]	AD [(sa)]	ADDRESS (0000)
MAGIC information adding specification	MAGIC NOMAGIC	MG NOMG	MAGIC

[ ] ; Means that the enclosed item can be omitted.
n ; $1 \leq n \leq 3$
fn ; Stands for 'file name'.
sa ; Stands for 'start address'.

Each control is explained below.

3.2.1 Module information output specification

(1) Function

Specifies whether or not the object code for the Execution Section is to be converted and output to the specified output file.

(2) Description

If MODULE or MD is specified, the object code for the Execution Section is output. However, If NOMODULE or NOMD is specified, the object code for the Execution Section is not output to the specifed output file.

NOTE: NOMODULE and NODATA should not be specified at the same time.

3.2.2 Data information output specification

(1) Function

Specifies whether the object code for the Data Section is to be converted and output to the output object module file or not.

(2) Description

If DATA or DT is specified, the object code for the Data Section is output. However, if NODATA or NODT is specified, the object code for the Data Section is not output to the specified output file.

NOTE; NOMODULE and NODATA should not be specified at the same time.

3.2.3 Symbol table module output specification

(1) Function

Specifies whether or not the symbol information is to be output to the symbol table module file.

(2) Description

a. In the case where {DEBUG / DB} [(file name)] is specified,

; The symbol information is output.

b. In the case where NODEBUG or NODB is specified,
; The symbol information is not output.

The 'file name' specified after DEBUG or DB is the file name of the symbol table module. If the file name is omitted, the primary name of this file name will be the same as that of the output object module file. However the 'SYM' is assigned as the file extension. If the drive name is omitted, it will be the same drive as that of the output object module file.

3.2.4 Object module output format specification

(1) Function

Specifies the format of the object module output.

(2) Description

a. If {TYPE / TP} (1) is specified.

; The object code for each Execution Section is output in separate object code module units.

b. If {TYPE / TP} (2) is specified.

; The object code for all of the Execution Sections is output in one lumped object code output.

c. If {TYPE / TP} (3) is specified

; The converted object file is output in 16-bit data format.

d. If {TYPE / TP} (4) is specified.

; The object code without '*' discriminator is output continuously

When 'TYPE specification' is omitted, 'TYPE(1)' is assumed by default.

3.2.5 HEX-format object module start address specification

(1) Function
Specifys the start address of byte address for loading at HEX format object module output.
Start address can be specified per module as executive part.
So this command is available when you specify TYPE (4).

(2) Description

{ADDRESS / AD} { MODULE / module name } (start address) [,Data (start address)] (CR)

The value to be specified in 'start address' must be within 0H and FFFFH, and it must be a hexadecimal number. If this control is omitted, a default start address of (0000) is assumed. However, this control is ignored when 'TYPE(3)' is specified.

3.3 Rules on file names

The rules on naming files are the same as the rules specified in the PART I of this manual.

Here, the rules on naming files under the Object-code Conversion Program are shown below.

(1) For a disk file name

```
file name      [drive name]primary name.ext

drive name     ; The name of the disk drive (A: P:) in
                 which the specified file is or is to be
                 stored.

primary name   ; A string of 8 or less characters.

ext            ; A string of 3 or less characters.
```

The list of defaults for naming files under the Object-code Conversion Program is shown in Table 3-2.

Table 3-2 Defaults for file names

File name		Default
Input object module file	drive name	The current default disk
	primary name	Cannot be omitted.
	file type	----------
Output object module file	drive name	The same as for the input object module file.
	primary name	The same as for the input object module file.
	file type	HEX
Symbol table module file	drive name	The same as for the output object module file.
	primary name	The same as for the output object module file.
	file type	SYM

3.2.6 MAGIC information adding specification

(1) Function

Specifies whether or not the information about the MAGIC is to be added to the output object module.

(2) Description

If MAGIC or MG is specified, the additional information (2 words) for the MAGIC functions is included in the output. However, if NOMAGIC or NOMG is specified, the additional information for the MAGIC functions is not included in the output.

The following describes the location where the additional information for the MAGIC functions is included.

- If the TYPE-1 or TYPE-3 is specified, the information for the MAGIC functions is appended at the end of each object code module for each Execution Section and the Data Section.
- If the TYPE-2 is specified, the information for the MAGIC functions is appended at end of the object code module for all Execution Sections and at the end of the Data Section.

When this control is omitted, 'NOMAGIC' is assumed by default.

For the MAGIC functions, refer to the simulation models in Chapter 2 of PART III, 'USAGE OF THE SIMULATOR' in this manual.

Examples:

1) A> OH7281 E:MDMAIN.LNK(CR)
; The drive name of the input object module file is E:; its primary name is MDMAIN, and its extention is LNK.

2) A> OH7281 SUB1(CR)
; Both the drive name and the file type of the input object module file are omitted, and the primary name is SUB1. (The SUB1 file is assumed to be on the current default drive and there is no file extension.)

(2) For device names

Only the following devices can be specified as I/O devices.

LST:	printer
CON:	console
PUN:	high-speed, paper-tape puncher

Examples:

1) A> OH7281 B:AFFIN4.LNK TO LST:(CR)
; The converted object module file is output to the printer.

2) A> OH7281 B:AFFIN.LNK TO PUN:(CR)
; The converted object module file is output to the paper-tape puncher.

3.4 Execution examples

The following practical examples illustrate the invocation of the Object-code Conversion Program through to its termination. (The user's key entries are underlined for the clarity.)

1) OH7281 held in drive B: inputs the object module file 'AFFIN4.LNK' held in drive A:, converts it, and outputs an object module file named 'AFFIN4.HEX'.

```
B> OH7281 A:AFFIN4.LNK (CR) <----OH7281 is started.

UPD7281 OBJECT CONVERTER VX.X [XX XXX XX]
   Copyright (C) 1984 NEC Corporation


OBJECT FILE = A:AFFIN4.LNK
HEXA   FILE = A:AFFIN4.HEX
OBJECT CONVERT COMPLETE
B>                          <----- Control returns to the
                                   operating system's
                                   prompt mode.
```

2) The conversion is performed in the same way as above, but in this example, a symbol table module file is created using the 'DEBUG' control.

```
B> OH7281 A:AFFIN4.LNK DEBUG()(CR)

UPD7281 OBJECT CONVERTER VX.X [XX XXX XX]
   Copyright (C) 1984 NEC Corporation

OBJECT FILE = A:AFFIN4.LNK
HEXA   FILE = A:AFFIN4.HEX
SYMBOL FILE = A:AFFIN4.SYM <----The symbol table file
OBJECT CONVERT CONPLETE           is created.
B>
```

3) The conversion is performed in the same way as in the above example 1), but in this example the converted object module file is output with the name 'AFIN4.HHH' to drive B:.

```
B> OH7281 A:AFFIN4.LNK TO AFIN4.HHH (CR)

UPD7281 OBJECT CONVERTER VX.X [XX XXX XX]
   Copyright (C) 1984 NEC Corporation

OBJECT FILE = A:AFFIN4.LNK
HEXA   FILE = A:AFFIN4.HHH
OBJECT CONVERT COMPLETE
```

4) The conversion is performed in the same way as in example 1), but the converted object module file is output with the file name 'AFF.HEX' to the drive B:. Furthermore, by specifying the control 'DB' (abbreviated form of DEBUG), a symbol table file is output with the name 'AF.SSS' to drive E:.

```
B> OH7281 A:AFFIN4.LNK TO AFF.HEX DB(E:AF.SSS) (CR)

UPD7281 OBJECT CONVERTER VX.X [XX XXX XX]
   Copyright (C) 1984 NEC Corporation

OBJECT FILE = A:AFFIN4.LNK
HEXA   FILE = A:AFF.HEX
SYMBOL FILE = E:AF.SSS
OBJECT CONVERT COMPLETE
B>
```

APPENDIX 1 LIST OF ERROR MESSAGES

This appendix explains error messages related to the object converter invocation command and those related errors occurred during the execution.

(1) Error messages related to the invocation command

If there exists any errors at the time the object converter invocation command, the error message(s) is output to the console.

1) Output format

```
-----------------------------------------
  *** ERROR error number   error message
-----------------------------------------
```

2) Output destination

Console

3) Processing after the message is output

After all the input commands are analyzed, the control is returned to the Operating System.

The list of error messages related to the object converter invocation command is given below.

LIST OF ERROR MESSAGES RELATED TO INVOCATION COMMAND

Error number F001	Message	MISSING FILE SPECIFICATION
	Cause	The input file name is not specified.
	User's procedure	Specify the input file name and restart the program.
Error number F002	Message	ILLEGAL FILE SPECIFICATION - file name
	Cause	File name specification does not satisfy concerning rules.
	User's procedure	Specify correct file name and restart the program.
Error number F003	Message	FILE SPECIFICATION CONFLICTED-file name
	Cause	There is a conflict between either input file name and output file name or among output file names.
	User's procedure	Specify correct file name(s) and restart the program.
Error number F004	Message	HEXA OBJECT OUTPUT IS NOT SPECIFIED
	Cause	There is no output specification for output object module.
	User's procedure	Specify the output for output module file and restart the program.
Error number F005	Message	FILE NOT FOUND - file name
	Cause	Input file does not exist.
	User's procedure	Specify the correct input file name and restart the program.
Error number F006	Message	WRITE PROTECTED FILE - file name
	Cause	The output file is write-protected.
	User's procedure	Either remove the write protection or specify another output file and then restart the program.

Error number	Message	ILLEGAL OR MISSING PARAMETER - parameter
F007	Cause	Either a parameter (file name) is not specified for the control requiring it, or it is specified incorrectly.
	User's procedure	Specify the correct parameter (file name) and restart the program.
Error number	Message	CONTROL IS NOT RECOGNIZED - control
F008	Cause	Improper character string specified for the control.
	User's procedure	Specify the correct control and restart the program.

(2) Errors occurring during execution time

If an error condition arises during the execution of the Object-Code Conversion Program, the following error message is output.

1) Output format

```
*** ERROR error number   error message
```

2) Output destination

Console

3) Processing after the message is output

The control is returned to the Operating System.

The list of error messages for the Object-code Conversion Program is given below.

LIST OF ERROR MESSAGES OF THE OBJECT-CODE CONVERSION PROGRAM

Error number		
A901	Message	WORKING TABLE SPACE EXHAUSTED
	Cause	not enough free space.
	User's procedure	------
A902	Message	INVALID FILE SINTAX - file name
	Cause	Input object module is incorrectly formatted.
	User's procedure	Recreate the object module.
A903	Message	FATAL I/O ERROR - file name
	Cause	A file I/O error has occurred.
	User's procedure	Verify the file's condition.

PART V

VAX/UNIX BASED SOFTWARE

VAX/UNIX BASED SOFTWARE

The following describes the differences between PC based software and that which is available on the VAX (UNIX-BASED). Unless otherwise indicated both version use the smae command format.

The ImPP Assembler

VAX/UNIX-BASED	PC-BASED
PPROGRAM NAME:	
ra7281	AS7281
OUTPUT FILES.	
object module (.rel)	load module (.lnk)
symbol table (.nmf)	-
(linker needs both files)	
FILE TYPES	
default: lower case	: upper case
.asm	.ASM
.rel	.LNK
.nmf	-
.prn	.PRN
INVOCATION	
%ra7281 fn.asm ['controls']	AS7281 FN.ASM [CONTROLS]
(fn.asm and FN.ASM are source file names)	
LETTERS (in sorce code)	
allowed: nul, sp	
not allowed: !, cr, ff, del	
NEW RESERVED WORDS	
PU, OUT, AG&FC	

BASIC CONTROLS

-p / -np	PRINT / NOPRINT
-m / -nm	-
MNEMONIC / NOMNEMONIC	-
-x / -nx	XFER / NOXREF
-w(n)	PAGEWIDTH(n)
-l(n)	PAGELENGTH(n)
-d(date)	DATE(date)

ERROR MESSAGES
(vax-based only)

Cannot open file (fn)
Cannot open include file
Duplicate OM name
Expression error
Illegal character (char)
Illegal control (cntl)
Illegal keyword in FTT
Illegal MN number
Illegal number (number)
Ileegal option
Illegal suffix (suffix)
Improper control format
Internal (CDlocate)
Mismatch number of output
Mismatch number of output values
NUMBER expected
Overwrite DEFINE definition
Overwrite definition
Overwrite EQUATE definition
Overwrite LITERAL definition
Redefined DMSETFT
Redefined DMSETLT
Redefined symbol
Syntax error
Too few arguments
Too many errors
Type clash in AT-expr
Unevaluable expression type
Unexpected AG&FC instruction
Unexpected argument in OUT
Unexpected argument in GE
Unexpected EOL in literal
Unexpected keyword argument in PU

FATAL ERROR MESSAGES
(vax-based only)

Cannot open .rel or .nmf file
Exhausted node area
Exhausted save node area
Exhausted space for strings
Literal expansion loop
LT pair table overflow
Relocation table overflow
Rout stack overflow
Symbol table overflow
System error in fixrel

The ImPP Simulator

COMMANDS

```
ENVIRON
 ---
MN n [,..]
[WRITE parameters        ]
[READ  parameters        ]
[MN FIELD expression ...]
[ID FIELD expression ...]
[DELAY expression        ]
[MAGIC                   ]
[RHASEL expression       ]
[CSSEL expression        ]
[PRIORITY expression     ]
 ---
END

 ---

 ---

LOAD OBJECT fn
LOAD OBJ fn
     ---
     regard allmodule
     allsymbol

[ALLSYMBOL      [LT]      ]
[               [FT]      ]
[ALLS           [DM]      ]

[SYMBOL         [LT]      ]
[               [FT] module name[]]
[SYM            [DM]      ]

[DATA fn [APPEND] ]

END

DATA INM fn
DATA OTM fn

 ---
MAP FILE fn
NOMAP

PRINT [n steps]

HOST exprl, expr2
```

```
ENVIRON
MODULE n
MN n [,...]
[WRITE parameters        ]
[READ parameters         ]
[MN FIELD expression...]
[ID FIELD expression...]
[DELAY expression        ]
[MAGIC                   ]
[RHASEL expression       ]
[CSSEL expression        ]
[PRIORITY expression     ]
[WORK d:                 ]

STATUS

TIMING parameters

LOAD OBJECT fn
LOAD OBJ fn
     ALLM
     ALLMODULE
     MODULE module name[]

[DATA fn [APPEND] ]

END

DATA INM fn
DATA OTM fn

MAP MEMORY n steps
MAP FILE fn
NOMAP

PRINT [n steps]

HOST exprl, expr2
```

GO [GR break condition]	GO [GR break condition]
(break conditions)	
LT [#n] FT addr expr	LT [#n] FT addr expr
DM ACCESSED [n]	DM ACCESSED [n]
DM ACCE [n]	DM ACCE [n TIMES]
@LT [#n] @FT DATA mask data	@LT [#n] @FT DATA mask data
@DM [n]	@DM [n TIMES]
DQ [#n] GQ SIZE size expr	DQ [#n] GQ SIZE size expr
OQ [n]	OQ [n TIMES]
[#n] INPUT [m]	[#n] INPUT [m]
OUTPUT	OUTPUT
---	BRn [AND BRn] [n TIMES] [OR BRn]
expression	expression STEPS
IM READ [n]	IM READ [n TIMES]
IM WRITE [n]	IM WRITE [n TIMES]
INM [n]	INM [n TIMES]
OTM	OTM
STEP	STEP
CONTINUE	CONTINUE
CONT	CONT
---	GR=GR break condition
---	GR
---	NOGR
---	BRn=break condition
---	BR[n]
---	NOBR[n]
---	BPn=break condition
---	BP[n]
---	NOBP[n]

```
DEFINE TRACE
 command [...]
END

TRACE
NOTRACE

 ---
 ---
 ---
 ---

 ---

 ---
END

 ---

 .[#n] symbol
 ---
[#n] SYMBOLS/SYM

 ---
 ---

REPEAT
 parameters
END

COUNT expr
 command
WHILE conditional expr
UNTIL conditional expr
     [...]
END

IF parameters

 ---

INCLUDE/INC fn

 ---

 ---
 ---

CLOCK

#=expression

#
```

```
DEFINE TRACE
 command [...]
END

TRACE
NOTRACE

TON all style
TOFF all style
NOTON
NOTOFF

CALCULATE parameters

DEFINE SYMBOL/SYM symb


NOSYMBOL [#n] [symb]

.[#n]symbol=parameters
[#n][type] parameters
 [#n] SYMBOLS/SYM

(assemble command)
(disassemble command)

REPEAT
 parameters
END

COUNT expr
 command
WHILE conditional expr
UNTIL conditional expr
     [...]
END

 IF parameters

(MACRO command)

INCLUDE/INC fn

WRITE parameters

ECHO
NOECHO

CLOCK

#=expression

#
```

[#n]LTEVAL	[#n]LTEVAL
[#n]PUEVAL	[#n]PUEVAL
[#n]IMEVAL	[#n]IMEVAL
[#n]NOEVAL	[#n]NOEVAL
---	SAVE RESOURCE/RSRC fn
---	LOAD RESOURCE/RCSC fn [.]
---	LIST fn
---	NOLIST
EXIT [M]	EXIT [M]

IM (Image Memory) data command

(This command displays IM values from addr1 to addr2. If addr2 is omitted then only data at addr1 is displayed).

IM addr1 TO addr2

IM set data command

(This command sets Image Memory data values n1,n2,... to IM from addr1 to addr2).

IM addr1 [TO addr2] = n1,n2, ...

IM Display Command (Vax-based)

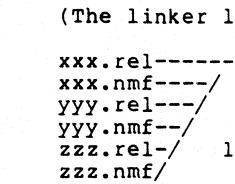
(This command displays IM data values like a bit pattern image).

IMDISP expr1, expr2, expr3, expr4

expression1: number of words in IM display line
expression2: start address for displaying data
expression3: number of words for display field's line
expression4: number of lines for display field

LINKER (Vax-based)

(The linker links plural files created by the assembler)



lk7281 (linker program name)

Input files: two types (which must be pairs)

object module file	(xxx.rel)
symbol table module file	(xxx.nmf)

Output files: two types

load module file	(xxx.lnk)
link list file	(xxx.prn)

(.lnk and .prn are the same file formats as on the PC-Based)

Invocation

%lk7281 xx yy zz to aa ["controls"]

controls:		
	-p / -np	make list file
	-m / -nm	make link map
	-c / -nc	make code list
	-e / -ne	make allocation error list
	-w(d) / -l(m)	PAGEWIDTH(n) PAGELENGTH(m)
	-d	input date to list file

MISC. NOTES:

assembler and simulator do not display opening messages unless the -v option is used.

ex. %as7281 -v

EUROPEAN DISTRIBUTORS

AUSTRIA
A & D
ABRAHAMCZIK & DEMEL
GES. MBH. & CO KG
EICHENSTRASSE 58-64/1
1120 WIEN
TEL.: (222) 85 76 61
TLX.: 134 273

BELGIUM
INTRA ELECTRONICS P.V.B.A.
BOSUIL 80, BUS 1
2100 DEURNE
TEL.: (03) 3 25 23 20
TLX.: 31102

MALCHUS ELECTRONICS P.V.B.A.
PLANTIN EN MORETUSLEI 172
2000 ANTWERPEN
TEL.: (03) 2 35 32 72
TLX.: 33 637

DENMARK
MER-EL A/S
VED KLAEDEBO 18
2970 HOERSHOLM
TEL.: (2) 57 10 00
TLX.: 37 360

FINLAND
OY FERRADO A/B
P.O. BOX 54
VALIMONTIE 1
00380 HELSINKI 38
TEL.: (90) 55 00 02
TLX.: 122 214

FRANCE
ASAP
2 AVENUE DES CHAUMES
78180 MONTIGNY LE BRETONNEUX
TÉL.: (1) 30 43 82 33
TLX.: 698 887

C.C.I.
5, RUE MARCELIN BERTHELOT
B.P. 92
92164 ANTONY
TÉL.: (1) 46 66 21 82
TLX.: 203 881

CGE COMPOSANTS
32 RUE GRANGE DAME ROSE
92360 MEUDON
TÉL.: (1) 46 30 24 25
TLX.: 632 118

SERTRONIQUE
60 RUE SAGEBIEN
CEDEX 43
72040 LE MANS
TÉL.: (16) 43 84 24 60
TLX.: 720 019

GERMANY
BIT-ELECTRONIC AG
DINGOLFINGER STRASSE 6
8000 MÜNCHEN 80
TEL.: (0 89) 41 80 07-0
TLX.: 5 212 931

GLEICHMANN + CO ELECTRONICS GMBH
WORMSER STRASSE 34
6710 FRANKENTHAL
TEL.: (0 62 33) 2 50 56
TLX.: 4 65 270

GLYN GMBH
SCHÖNE AUSSICHT 30
6272 NIEDERNHAUSEN
TEL.: (0 61 27) 80 77
TLX.: 4 186 911

H3W ELEKTRONIK VERTRIEBS GMBH
STAHLGRUBERRING 12
8000 MÜNCHEN 82
TEL.: (0 89) 42 92 71
TLX.: 5 214 514

MICROSCAN GMBH
ÜBERSEERING 31
2000 HAMBURG 60
TEL.: (0 40) 6 32 00 30
TLX.: 213 288

REIN ELEKTRONIK GMBH
LÖTSCHERWEG 66
4054 NETTETAL 1
TEL.: (0 21 53) 73 31 11
TLX.: 8 54 251

SYSTEM ELEKTRONIK VERTRIEBS GMBH
HEESFELD 4
3300 BRAUNSCHWEIG
TEL.: (05 31) 31 40 95
TLX.: 9 52 351

ULTRATRONIK GMBH
GEWERBESTRASSE 4
8036 HERRSCHING
TEL.: (0 81 52) 37 09-0
TLX.: 5 26 459

UNIELECTRONIC VERTRIEBS GMBH
LISE-MEITNER-STRASSE 8
6072 DREIEICH 1 B. FRANKFURT
TEL.: (0 61 03) 3 51 75
TLX.: 4 11 213

ITALY
ADELSY S.R.L.
VIA DEL FONDITORE, 5
LOCALITA ROVERI
40127 BOLOGNA
TEL.: (51) 532119
TLX.: 510 226

CLAITRON S.P.A.
VIA GALLARATE, 211
20151 MILANO
TEL.: (2) 3010091
TLX.: 313 843

MELCHIONI S.P.A.
VIA COLLETTA, 37
20135 MILANO
TEL.: (2) 57941
TLX.: 315 293

NETHERLANDS
INNOCIRCUIT
MALCHUS ELECTRONICA
ADVIESGROEP
MALCHUS B.V.
FOKKERSTRAAT 511-513
3125 BD SCHIEDAM
TEL.: (10) 4277777
TLX.: 21598

INTRA ELECTRONICS B.V.
GULBERG 33
5674 TE NUENEN
TEL.: (0 40) 83 80 09
TLX.: 59418

NORWAY
JAKOB HATTELAND ELECTRONIC A/S
P.B. 25
5578 NEDRE VATS
TEL.: (47) 63 111
TLX.: 428 50

PORTUGAL
AMPEREL S.A.
AV. FONTES PEREIRA DE MELO 47, 4D
1000 LISBOA
TEL.: (1) 53 26 98
TLX.: 18588

SPAIN
COMELTA SA
EMILIO MUNOZ 41, NAVE 1-1-2
28037 MADRID
TEL.: (1) 7 54 30 77
TLX.: 42 007

VENCO
CALLE GALILEO 249
08028 BARCELONA
TEL.: (3) 330 97 51
TLX.: 98 266

SWEDEN
NAX AB KOMPONENTBOLAGET
BOX 4115
17104 SOLNA
TEL.: (8) 98 51 40
TLX.: 17912

SWITZERLAND
MEMOTEC AG
GASWERKSTRASSE 32
4901 LANGENTHAL
TEL.: (63) 28 11 22
TLX.: 9 82 550

UNITED KINGDOM
ANZAC COMPONENTS LIMITED
822, YEOVIL ROAD
SLOUGH TRADING ESTATE
SLOUGH
BERKSHIRE
TEL.: (62 86) 44 11
TLX.: 847 949

CELDIS LIMITED
37, LOVEROCK ROAD
READING
BERKSHIRE
RG3 1EL
TEL.: (7 34) 56 51 71
TLX.: 848 370

DIALOGUE DISTRIBUTION LIMITED
WICAT HOUSE
403, LONDON ROAD
CAMBERLEY
SURREY
GO15 3HL
TEL.: (2 76) 68 20 01
TLX.: 858 944

IMPULSE ELECTRONICS LIMITED
HAMMOND HOUSE
CATERHAM
SURREY
CR3 6XG
TEL.: (8 83) 4 64 33
TLX.: 291 496

S.T.C. MULTICOMPONENTS LIMITED
EDINBURGH WAY
HARLOW
ESSEX
CM20 2DF
TEL.: (2 79) 44 29 71
TLX.: 818 763

V.S.I. ELECTRONICS (UK) LIMITED
ROYDONBURY INDUSTRIAL PARK
HORSECROFT ROAD
HARLOW
ESSEX
CM19 5BY
TEL.: (2 79) 2 96 66
TLX.: 81 387

NEC OFFICES

NEC Electronics (Europe) GmbH, Oberrather Str. 4, 4000 Düsseldorf 30, W. Germany,
Tel. (02 11) 65 03 01, Telex 8 58 996-0

NEC Electronics (Germany) GmbH, Oberrather Str. 4, 4000 Düsseldorf 30,
Tel. (02 11) 65 03 02, Telex 8 58 996-0
- Königstr. 12, 3000 Hannover 1, Tel. (05 11) 31 60 91, Telex 9 230 109
- Arabellastr. 17, 8000 München 81, Tel. (0 89) 92 10 03-0, Telex 5 22 971
- Heilbronner Str. 314, 7000 Stuttgart 30, Tel. (07 11) 89 09 10, Telex 7 252 220

NEC Electronics (BNL) - Boschdijk 187a, NL-5612 HB Eindhoven, Tel. (0 40) 44 58 45,
Telex 51 923

NEC Electronics (Scandinavia) - Box 4039, S-18304 Täby, Tel. (08) 73 28 200,
Telex 13 839

NEC Electronics (France) S.A., 9, rue Paul Dautier, B.P. 187,
F-78142 Velizy Villacoublay Cedex, Tél. (1) 39 46 96 17, Télex 699 499

NEC Electronics (France) S.A., Representacion en Espana, Edificio «La Caixa»,
Paseo de la Castellana 51, E-28046 Madrid, Tél. (1) 41 94 150, Télex 41 316

NEC Electronics Italiana S.R.L., Via Fabio Filzi, 25A, I-20124 Milano, Tel. (02) 67 09 108,
Telex 315 355
- Rome Office, Via Monte Cervialto, 131, I-00139 Rome,
Tel. (06) 8 11 12 91, Telex 623 323

NEC Electronics (UK) Ltd., Cygnus House, Sunrise Park Way, Milton Keynes, MK14 6NP,
Tel. (09 08) 69 11 33, Telex 777 565
- Dublin Office, 34/35 South William Street, Dublin 2, Ireland, Tel. (00 01) 71 02 00