SPRU031

Third-Generation TMS320 User's Guide

Digital Signal Processor Products

TEXAS INSTRUMENTS

Manual Update

Document Title: Third-Generation TMS320 User's Guide, SPRU031

Document Number: SPRZ048 ECN Number: 526635

This Manual Update should be appended to the *Third-Generation TMS320 User's Guide*. Changes should be made as indicated on the designated pages.

Page **Change or Add**

2-2 Table 2-1:

Line	Function (Now)	Function (Should Be)
1	X11	XA11
2	X12	XA12
5	XOD2	XD2
20	IOA5	XA5
26	IOD23	XD23
27	IOD24	XD24
28	IOD25	XD25
28	VSUBS	SUBS
29	IOD26	XD26
30	IOD27	XD27
31	IOD28	XD28
32	IOD29	XD29
33	IOD30	XD30
34	IOD31	XD31
35	$\overline{\text{IORDY}}$	$\overline{\text{XRDY}}$

2-6 Table 2-2. Insert the following at the end of the table.

LOCATOR (1 PIN)			
NONE	1	NC	Reserved. See Table 2-1 and Figure 2-1.

7-9 Line 7: *src* should be *dst*.

A-5 Table A-5: Characteristics (13), (14), (15), (16), (17), and (18) change (IO) to (X) in name and description.

A-6 Figure A-4: Change (IO)R/$\overline{\text{W}}$ to (X)R/$\overline{\text{W}}$, (IO)A to (X)A, (IO)D to (X)D, and (IO)$\overline{\text{RDY}}$ to (X)$\overline{\text{RDY}}$.

A-6 Table A-6: All characteristics change (IO) to (X) in name and description.

A-7 Figure A-5: Change (IO)R/$\overline{\text{W}}$ to (X)R/$\overline{\text{W}}$, (IO)A to (X)A, (IO)D to (X)D, and (IO)$\overline{\text{RDY}}$ to (X)$\overline{\text{RDY}}$.

A-8 Figure A-6: Change IOR/$\overline{\text{W}}$ to XR/$\overline{\text{W}}$, IOA to XA, IOD to XD, and $\overline{\text{IORDY}}$ to $\overline{\text{XRDY}}$. Change $\overline{\text{(M)STRB}}$ in title to $\overline{\text{IOSTRB}}$.

A-9 Table A-7: Characteristics (22), (14.1), (15.1), (16.1), (17.1), and (18.1) change IO to X in name and description.

A-9 Table A-8: All characteristics change IO to X in name and description.

The changes shown in this Manual Update will be included in the next revision of the *Third-Generation TMS320 User's Guide.*

Third-Generation TMS320 User's Guide

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to or to discontinue any semiconductor product or service identified in this publication without notice. TI advises its customers to obtain the latest version of the relevant information to verify, before placing orders, that the information being relied upon is current.

TI warrants performance of its semiconductor products to current specifications in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed.

TI assumes no liability for TI applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Copyright © 1988, Texas Instruments Incorporated

Introduction 1

Pinout and Signal Descriptions 2

Architectural Overview 3

CPU Registers, Memory, and Cache 4

Data Formats and Floating-Point Operation 5

Addressing 6

Program Flow Control 7

External Bus Operation 8

Peripherals 9

Pipeline Operation 10

Assembly Language Instructions 11

Software Applications 12

Hardware Applications 13

Appendices A-D

Contents

Section		*Page*
1	**Introduction**	**1-1**
1.1	General Description	1-3
1.2	Key Features	1-4
1.3	Typical Applications	1-5
1.4	How To Use This Manual	1-6
1.5	References	1-8
2	**Pinout and Signal Descriptions**	**2-1**
2.1	Signal Descriptions	2-3
3	**Architectural Overview**	**3-1**
3.1	Central Processing Unit (CPU)	3-3
3.1.1	Multiplier	3-5
3.1.2	Arithmetic Logic Unit (ALU)	3-5
3.1.3	Auxiliary Register Arithmetic Units (ARAUs)	3-5
3.1.4	CPU Register File	3-5
3.2	Memory Organization	3-8
3.2.1	RAM, ROM, and Cache	3-8
3.2.2	Memory Maps	3-10
3.2.3	Memory Addressing Modes	3-12
3.2.4	Instruction Set Summary	3-12
3.3	Internal Bus Operation	3-20
3.4	External Bus Operation	3-21
3.5	Peripherals	3-22
3.5.1	Timers	3-23
3.5.2	Serial Ports	3-23
3.6	Direct Memory Access (DMA)	3-24
4	**CPU Registers, Memory, and Cache**	**4-1**
4.1	CPU Register File	4-2
4.1.1	Extended-Precision Registers (R0-R7)	4-3
4.1.2	Auxiliary Registers (AR0-AR7)	4-3
4.1.3	Data Page Pointer (DP)	4-3
4.1.4	Index Registers (IR0, IR1)	4-4
4.1.5	Block Size Register (BK)	4-4
4.1.6	System Stack Pointer (SP)	4-4
4.1.7	Status Register (ST)	4-4
4.1.8	CPU/DMA Interrupt Enable Register (IE)	4-7
4.1.9	CPU Interrupt Flag Register (IF)	4-8
4.1.10	I/O Flags Register (IOF)	4-9
4.1.11	Repeat Counter (RC) and Block Repeat Registers (RS, RE)	4-10
4.1.12	Program Counter (PC)	4-10
4.1.13	Reserved Bits and Compatibility	4-10
4.2	Memory	4-11
4.2.1	Memory Maps	4-11
4.2.2	Peripheral Bus Map	4-13
4.2.3	Reset/Interrupt/Trap Vector Map	4-13
4.3	Instruction Cache	4-15

4.3.1 Cache Architecture 4-15
4.3.2 Cache Algorithm 4-16
4.3.3 Cache Control Bits 4-17

5 Data Formats and Floating-Point Operation 5-1
5.1 Integer Formats 5-2
5.1.1 Short Integer Format 5-2
5.1.2 Single-Precision Integer Format 5-2
5.2 Unsigned-Integer Formats 5-3
5.2.1 Short Unsigned-Integer Format 5-3
5.2.2 Single-Precision Unsigned-Integer Format 5-3
5.3 Floating-Point Formats 5-4
5.3.1 Short Floating-Point Format 5-4
5.3.2 Single-Precision Floating-Point Format 5-5
5.3.3 Extended-Precision Floating-Point Format 5-6
5.3.4 Conversion Between Floating-Point Formats 5-7
5.4 Floating-Point Multiplication 5-9
5.5 Floating-Point Addition and Subtraction 5-13
5.6 Normalization Using the NORM Instruction 5-17
5.7 Rounding: The RND Instruction 5-20
5.8 Floating-Point to Integer Conversion 5-22
5.9 Integer to Floating-Point Conversion Using the FLOAT Instruction 5-24

6 Addressing 6-1
6.1 Types of Addressing 6-2
6.1.1 Register Addressing 6-2
6.1.2 Direct Addressing 6-3
6.1.3 Indirect Addressing 6-4
6.1.4 Short-Immediate Addressing 6-16
6.1.5 Long-Immediate Addressing 6-17
6.1.6 PC-Relative Addressing 6-17
6.2 Groups of Addressing Modes 6-18
6.2.1 General Addressing Modes 6-18
6.2.2 Three-Operand Addressing Modes 6-19
6.2.3 Parallel Addressing Modes 6-20
6.2.4 Long-Immediate Addressing Mode 6-21
6.2.5 Conditional-Branch Addressing Modes 6-21
6.3 Circular Addressing 6-22
6.4 Bit-Reversed Addressing 6-27
6.5 System and User Stack Management 6-28
6.5.1 Stacks 6-28
6.5.2 Queues and Deques 6-30

7 Program Flow Control 7-1
7.1 Repeat Modes 7-2
7.1.1 Repeat Mode Initialization 7-2
7.1.2 RPTB Initialization 7-3
7.1.3 RPTS Initialization 7-3
7.1.4 Repeat Mode Operation 7-4
7.2 Delayed Branches 7-7
7.3 Interlocked Operations 7-8
7.4 Reset Operation 7-13
7.5 Interrupts 7-16

8 External Bus Operation **8-1**
8.1 External Interface Control Registers 8-2
8.1.1 Primary Bus Control Register 8-3
8.1.2 Expansion Bus Control Register 8-4
8.2 External Interface Timing 8-5
8.2.1 Primary Bus Cycles 8-5
8.2.2 Expansion Bus I/O Cycles 8-10
8.3 Programmable Wait States 8-19
8.4 Programmable Bank Switching 8-21

9 Peripherals **9-1**
9.1 Timers 9-2
9.1.1 Timer Global Control Register 9-3
9.1.2 Timer Period and Counter Registers 9-5
9.1.3 Timer Pulse Generation 9-6
9.1.4 Timer Operation Modes 9-7
9.2 Serial Ports 9-9
9.2.1 Serial Port Global Control Register 9-11
9.2.2 FSX/DX/CLKX Port Control Register 9-14
9.2.3 FSR/DR/CLKR Port Control Register 9-15
9.2.4 Receive/Transmit Timer Control Register 9-16
9.2.5 Receive/Transmit Timer Counter Register 9-18
9.2.6 Receive/Transmit Timer Period Register 9-19
9.2.7 Data Transmit Register 9-19
9.2.8 Data Receive Register 9-19
9.2.9 Serial Port Operation Configurations 9-20
9.2.10 Serial Port Timing 9-23
9.2.11 Serial Port Interrupt Sources 9-26
9.2.12 Serial Port Functional Operation 9-26
9.3 DMA Controller 9-33
9.3.1 DMA Global Control Register 9-34
9.3.2 Destination and Source Address Registers 9-36
9.3.3 Transfer Counter Register 9-36
9.3.4 CPU/DMA Interrupt Enable Register 9-36
9.3.5 DMA Memory Transfer Operation 9-38
9.3.6 Synchronization of DMA Channels 9-42

10 Pipeline Operation **10-1**
10.1 Pipeline Structure 10-2
10.2 Pipeline Conflicts 10-4
10.2.1 Branch Conflicts 10-4
10.2.2 Register Conflicts 10-6
10.2.3 Memory Conflicts 10-8
10.3 Resolving Memory Conflicts 10-14
10.4 Clocking Of Memory Accesses 10-16
10.4.1 Program Fetches 10-16
10.4.2 Data Loads and Stores 10-16

11 Assembly Language Instructions 11-1
11.1 Instruction Set . . . 11-2
11.1.1 Load and Store Instructions . . . 11-2
11.1.2 Two-Operand Instructions . . . 11-3
11.1.3 Three-Operand Instructions . . . 11-4
11.1.4 Program Control Instructions . . . 11-4
11.1.5 Interlocked Operations Instructions . . . 11-5
11.1.6 Parallel Operations Instructions . . . 11-5
11.2 Condition Codes and Flags . . . 11-8
11.3 Individual Instructions . . . 11-11
11.3.1 Symbols and Abbreviations . . . 11-11
11.3.2 Optional Assembler Syntaxes . . . 11-13
11.3.3 Individual Instruction Descriptions . . . 11-15

12 Software Applications 12-1
12.1 Processor Initialization . . . 12-3
12.2 Program Control . . . 12-7
12.2.1 Subroutines . . . 12-7
12.2.2 Software Stack . . . 12-9
12.2.3 Interrupt Service Routines . . . 12-10
12.2.4 Delayed branches . . . 12-15
12.2.5 Repeat Modes . . . 12-16
12.2.6 Computed GOTO's . . . 12-18
12.3 Logical and Arithmetic Operations . . . 12-20
12.3.1 Bit Manipulation . . . 12-20
12.3.2 Block Moves . . . 12-22
12.3.3 Bit-Reversed Addressing . . . 12-22
12.3.4 Integer and Floating-point Division . . . 12-23
12.3.5 Square Root . . . 12-29
12.3.6 Extended-Precision Arithmetic . . . 12-32
12.3.7 Floating-point Format Conversion: IEEE to/from TMS320C30 . . . 12-35
12.4 Application-Oriented Operations . . . 12-45
12.4.1 Companding . . . 12-45
12.4.2 FIR, IIR, and Adaptive Filters . . . 12-49
12.4.3 Matrix-Vector Multiplication . . . 12-60
12.4.4 Fast Fourier Transforms (FFT) . . . 12-63
12.4.5 Lattice Filters . . . 12-79
12.5 Programming Tips . . . 12-86
12.5.1 C-Callable Routines . . . 12-86
12.5.2 Hints for Assembly Coding . . . 12-86

13 Hardware Applications 13-1
13.1 System Configuration Options Overview . . . 13-2
13.1.1 Categories of Interfaces on the TMS320C30 . . . 13-2
13.1.2 Typical System Block Diagram . . . 13-3
13.2 Primary Bus Interface . . . 13-4
13.2.1 Zero Wait-State Interface To RAMs . . . 13-4
13.2.2 Ready Generation . . . 13-10
13.2.3 Bank Switching Techniques . . . 13-14
13.3 Expansion Bus Interface . . . 13-17
13.4 System Control Functions . . . 13-21
13.4.1 Clock Oscillator Circuitry . . . 13-21
13.4.2 Reset Signal Generation . . . 13-23
13.5 XDS1000 Target Design Considerations . . . 13-26

A	TMS320C30 Timing Specifications & Dimensions	A-1
B	Development Support/Part Order Information	B-1
C	Instruction Opcodes	C-1
D	Quality and Reliability	D-1

Illustrations

Figure *Page*

1-1 TMS320 Device Evolution 1-1
2-1 TMS320C30 Pin Assignments 2-1
3-1 TMS320C30 Block Diagram 3-2
3-2 Central Processing Unit (CPU) 3-4
3-3 Memory Organization 3-9
3-4 Memory Maps 3-11
3-5 Peripheral Modules 3-22
3-6 DMA Controller 3-24
4-1 Extended-Precision Register Floating-Point Format 4-3
4-2 Extended-Precision Register Integer Format 4-3
4-3 Status Register 4-5
4-4 CPU/DMA Interrupt Enable Register (IE) 4-7
4-5 CPU Interrupt Flag Register (IF) 4-9
4-6 I/O Flag Register (IOF) 4-9
4-7 Memory Maps 4-12
4-8 Peripheral Bus Memory Map 4-13
4-9 Reset, Interrupt, and Trap Vector Locations 4-14
4-10 Instruction Cache Architecture 4-15
4-11 Address Partitioning for Cache Control Algorithm 4-16
5-1 Short Integer Format and Sign Extension of Short Integer 5-2
5-2 Single-Precision Integer Format 5-2
5-3 Short Unsigned-Integer Format and Zero Fill 5-3
5-4 Single-Precision Unsigned-Integer Format 5-3
5-5 Generic Floating-Point Format 5-4
5-6 Short Floating-Point Format 5-4
5-7 Single-Precision Floating-Point Format 5-5
5-8 Extended-Precision Floating-Point Format 5-6
5-9 Flowchart for Floating-Point Multiplication 5-10
5-10 Flowchart for Floating-Point Addition 5-14
5-11 Flowchart for NORM Instruction Operation 5-18
5-12 Flowchart for Floating-Point Rounding by the RND Instruction 5-21
5-13 Flowchart for Floating-Point to Integer Conversion by FIX Instructions 5-23
5-14 Flowchart for Integer to Floating-Point Conversion Using the FLOAT Instruction 5-24
6-1 Direct Addressing 6-4
6-2 Encoding for General Addressing Modes 6-18
6-3 Encoding for Three-Operand Addressing Modes 6-19
6-4 Encoding for Parallel Addressing Modes 6-20
6-5 Encoding for Long-Immediate Addressing Mode 6-21
6-6 Encoding for Conditional-Branch Addressing Modes 6-21
6-7 Flowchart for Circular Addressing 6-23
6-8 Circular Buffer Implementation 6-24
6-9 Circular Addressing Example 6-25
6-10 Data Structure for FIR Filters 6-25
6-11 FIR Filter Code Using Circular Addressing 6-26
6-12 Bit-Reversed Addressing Example 6-27
6-13 System Stack Configuration 6-28
6-14 Implementions of High-to-Low Memory Stacks 6-29
6-15 Implementions of Low-to-High Memory Stacks 6-29

7-1 Repeat Mode Control Algorithm 7-4
7-2 Multiple TMS320C30s Sharing Global Memory 7-10
7-3 Zero-Logic Interconnect of TMS320C30s 7-11
7-4 Interrupt Logic Functional Diagram 7-16
7-5 Interrupt Processing 7-19
8-1 Memory-Mapped External Interface Control Registers 8-2
8-2 Primary Bus Control Register 8-3
8-3 Expansion Bus Control Register 8-4
8-4 Read-Read-Write for $\overline{(M)STRB}$ = 0 8-6
8-5 Write-Write-Read for $\overline{(M)STRB}$ = 0 8-7
8-6 Use of Wait States for Read for $\overline{(M)STRB}$ = 0 8-8
8-7 Use of Wait States for Write for $\overline{(M)STRB}$ = 0 8-9
8-8 Read and Write for $\overline{IOSTRB}$ = 0 8-10
8-9 Read with One Wait-State for $\overline{IOSTRB}$ = 0 8-11
8-10 Write with One Wait-State for $\overline{IOSTRB}$ = 0 8-12
8-11 Memory Read and I/O Write for Expansion Bus 8-13
8-12 I/O Write and Memory Read for Expansion Bus 8-14
8-13 Memory Write and I/O Read for Expansion Bus 8-15
8-14 Inactive Bus States for $\overline{IOSTRB}$ 8-16
8-15 Inactive Bus States for $\overline{STRB}$ and $\overline{MSTRB}$ 8-17
8-16 $\overline{HOLD}$ and $\overline{HOLDA}$ Timing 8-18
8-17 BNKCMP Example 8-21
8-18 Bank Switching Example 8-23
9-1 Timer Block Diagram 9-2
9-2 Memory-Mapped Timer Locations 9-3
9-3 Timer Global Control Register 9-5
9-4 Timer Timing 9-6
9-5 Timer I/O Port Configurations 9-7
9-6 Timer Modes as Defined by CLKSRC and FUNC 9-8
9-7 Serial Port Block Diagram 9-10
9-8 Memory-Mapped Locations for the Serial Port 9-11
9-9 Serial Port Global Control Register 9-14
9-10 FSX/DX/CLKX Port Control Register 9-15
9-11 FSR/DR/CLKR Port Control Register 9-16
9-12 Receive/Transmit Timer Control Register 9-18
9-13 Receive/Transmit Timer Counter Register 9-18
9-14 Receive/Transmit Timer Period Register 9-19
9-15 Transmit Buffer Shift Operation 9-19
9-16 Receive Buffer Shift Operation 9-20
9-17 Serial Port Clocking in I/O Mode 9-21
9-18 Serial Port Clocking in Serial Port Mode 9-22
9-19 Data Word Format in Handshake Mode 9-25
9-20 Single Zero Sent as an Acknowledge 9-25
9-21 Direct Connection Using Handshake Mode 9-26
9-22 Fixed Burst Mode 9-27
9-23 Fixed Continuous Mode With Frame Synch 9-28
9-24 Fixed Continuous Mode Without Frame Synch 9-29
9-25 Exiting Fixed Continuous Mode Without Frame Synch, FSX Internal 9-30
9-26 Variable Burst Mode 9-30
9-27 Variable Continuous Mode With Frame Synch 9-31
9-28 Variable Continuous Mode Without Frame Synch 9-32
9-29 Memory-Mapped Locations for a DMA Channel 9-33
9-30 DMA Global Control Register 9-34
9-31 CPU/DMA Interrupt Enable Register 9-37
9-32 Timing and Number of Cycles for DMA Transfers When Destination is On-Chip 9-39

9-33 DMA Timing When Destination is a Primary Bus 9-39
9-34 DMA Timing When Destination is an Expansion Bus 9-40
9-35 No DMA Synchronization 9-43
9-36 DMA Source Synchronization 9-44
9-37 DMA Destination Synchronization 9-45
9-38 DMA Source and Destination Synchronization 9-46
10-1 TMS320C30 Pipeline Structure 10-2
10-2 Two-Operand Instruction Word 10-17
10-3 Three-Operand Instruction Word 10-17
10-4 A Multiply or CPU Operation with a Parallel Store 10-18
10-5 Two Parallel Stores 10-18
10-6 Parallel Multiplies and Adds 10-19
12-1 Data Memory Organization For a FIR Filter 12-50
12-2 Data Memory Organization For a Single Biquad 12-52
12-3 Data Memory Organization For N Biquads 12-55
12-4 Data Memory Organization for Matrix-Vector Multiplication 12-61
12-5 Structure of the Inverse Lattice Filter 12-80
12-6 Data Memory Organization for Lattice Filters 12-80
12-7 Structure of the (Forward) Lattice Filter 12-83
13-1 External Interfaces on the TMS320C30 13-2
13-2 Possible System Configurations 13-3
13-3 Ram Interface - No $\overline{OE}$ 13-5
13-4 Interface Read Timing 13-6
13-5 Interface Write Timing 13-6
13-6 RAM Interface - $\overline{OE}$ 13-7
13-7 Read Operations Timing 13-8
13-8 Write Operations Timing 13-9
13-9 Circuit For Generation of 0, 1, or 2 Wait States For Multiple Devices 13-13
13-10 Bank Switching For Cyprus Semiconductors CY7C185 13-15
13-11 Timing For Read Operations Using Bank Switching 13-16
13-12 Expansion Bus Interface to A/D Converter 13-18
13-13 Timing of Expansion Bus Interface 13-19
13-14 Crystal Oscillator Circuit 13-21
13-15 Magnitude of the Impedance of the Oscillator LC Network 13-22
13-16 Reset Circuit 13-23
13-17 Voltage on the TMS320C30 Reset Pin 13-24
13-18 12 Pin Header Signals 13-26
13-19 Typical Setup For Using the Emulation Connection of the XDS1000 13-27
A-1 Test Load Circuit A-3
A-2 X2/CLKIN Timing A-4
A-3 H1/H3 Timing A-4
A-4 Memory ($\overline{(M)STRB}$ = 0) Read A-6
A-5 Memory ($\overline{(M)STRB}$ = 0) Write A-7
A-6 Memory ($\overline{(M)STRB}$ = 0) Read A-8
A-7 Memory ($\overline{IOSTRB}$ = 0) Write A-10
A-8 Timing for XF0 and XF1 When Executing a LDFI or LDII A-11
A-9 Timing for XF0 When Executing a STFI or STII A-12
A-10 Timing for XF0 and XF1 When Executing SIGI A-13
A-11 Timing for Loading XF Register When Configured as an Output Pin A-14
A-12 Change of XF From Output to Input Mode A-15
A-13 Change of XF From Input to Output Mode A-16
A-14 $\overline{RESET}$ Timing A-17
A-15 $\overline{RESET}$ and $\overline{INT}$(3-0) Response Timing A-19
A-16 $\overline{IACK}$ Timing A-20
A-17 TRAP Response Timing A-21
A-18 Fixed Data Rate Mode A-22

A-19 Variable Data Rate Mode A-23
A-20 HOLD/HOLDA Timing A-25
A-21 TMS320C30 180 Pin PGA Dimensions A-26
B-1 TMS320C30 Development Environment B-2
B-2 TMS320C30 Simulator User Interface B-8
B-3 TMS320C30 XDS1000 Development Environment B-10
B-4 TMS320 Device Nomenclature B-15

Tables

Table *Page*

1-1 Typical Applications of the TMS320 Family 1-5
2-1 TMS320C30 Pin Function Assignments 2-2
2-2 TMS320C30 Signal Descriptions 2-3
3-1 CPU Registers 3-6
3-2 Instruction Set Summary 3-13
4-1 CPU Registers 4-2
4-2 Status Register Bits Summary 4-6
4-3 IE Register Bits Summary 4-8
4-4 IF Register Bits Summary 4-9
4-5 IOF Register Bits Summary 4-10
4-6 Combined Effect of the CE and CF Bits 4-18
6-1 CPU Register/Assembler Syntax and Function 6-3
6-2 Indirect Addressing 6-5
6-3 Index Steps and Bit-Reversed Addressing 6-27
7-1 Repeat Mode Registers 7-2
7-2 Interlocked Operations 7-8
7-3 Pin Operation at Reset 7-13
7-4 Reset and Interrupt Vector Locations 7-18
8-1 Primary Bus Control Register Bits Summary 8-3
8-2 Expansion Bus Control Register Bits Summary 8-4
8-3 Wait-State Generation When SWW = 0 0 8-20
8-4 Wait-State Generation When SWW = 0 1 8-20
8-5 Wait-State Generation When SWW = 1 0 8-20
8-6 Wait-State Generation When SWW = 1 1 8-20
8-7 BNKCMP and Bank Size 8-21
9-1 Timer Global Control Register Bits Summary 9-4
9-2 Result of a Write of Specified Values of GO and HLD 9-5
9-3 Serial Port Global Control Register Bits Summary 9-12
9-4 FSX/DX/CLKX Port Control Register Bits Summary 9-14
9-5 FSR/DR/CLKR Port Control Register Bits Summary 9-15
9-6 Receive/Transmit Timer Control Register 9-17
9-7 Global Control Register Bits 9-34
9-8 START Bits and Operation of the DMA 9-35
9-9 STAT Bits and Status of the DMA 9-35
9-10 SYNCH Bits and Synchronization of the DMA 9-35
9-11 CPU/DMA Interrupt Enable Register Bits 9-37
9-12 Maximum DMA Transfer Rates When $C_r = C_W = 0$ 9-41
9-13 Maximum DMA Transfer Rates When $C_r = 1$, $C_W = 0$ 9-41
9-14 Maximum DMA Transfer Rates When $C_r = 1$, $C_W = 1$ 9-41
10-1 One Program Fetch and One-Data Access for Maximum Performance 10-14
10-2 One Program Fetch and Two Data Accesses for Maximum Performance 10-15

11-1 Load and Store Instructions 11-2
11-2 Two-Operand Instructions 11-3
11-3 Three-Operand Instructions 11-4
11-4 Program Control Instructions 11-5
11-5 Interlocked Operations Instructions 11-5
11-6 Parallel Instructions 11-6
11-7 Output Value Formats 11-8
11-8 Condition Codes and Flags 11-10
11-9 Instruction Symbols 11-12
12-1 TMS320C30 FFT Timing Benchmarks 12-79
13-1 Bank Switching Interface Timing 13-16
A-1 Absolute Maximum Ratings Over Specified Temperature Range A-2
A-2 Recommended Operating Conditions A-2
A-3 Electrical Characteristics Over Specified Free-Air Temperature Range A-3
A-4 Switching Characteristics for CLKIN, H1, and H3 ..... A-5
A-5 Switching Characteristics for a memory ($\overline{(M)STRB}$ = 0) read A-5
A-6 Switching Characteristics for a memory ($\overline{(M)STRB}$ = 0) Write A-6
A-7 Switching Characteristics for a Memory ($\overline{IOSTRB}$ = 0) Read A-9
A-8 Switching Characteristics for a Memory ($\overline{IOSTRB}$ = 0) Write A-9
A-9 Information for Figure A-8 ..... A-11
A-10 Information for Figure A-9 ..... A-12
A-11 Information for Figure A-10 ..... A-13
A-12 Information for Figure A-11 ..... A-14
A-13 Information for Figure A-12 ..... A-15
A-14 Information for Figure A-13 ..... A-16
A-15 Information for Figure A-14 ..... A-18
A-16 Information for Figure A-15 ..... A-19
A-17 Information for Figure A-16 ..... A-20
A-18 Serial Port Timing as Shown in Figures A-18 and A-19 ..... A-24
A-19 Information for Figure A-20 ..... A-25
B-1 TMS320C30 Digital Signal Processor Part Numbers B-13
B-2 TMS320C30 Support Tool Part Numbers B-13
C-1 TMS320C30 Instruction Opcodes C-1
D-1 Microprocessor and Microcontroller Tests D-5
D-2 TMS320C30 Transistors D-5

Introduction 1

Pinout and Signal Descriptions 2

Architectural Overview 3

CPU Registers, Memory, and Cache 4

Data Formats and Floating-Point Operation 5

Addressing 6

Program Flow Control 7

External Bus Operation 8

Peripherals 9

Pipeline Operation 10

Assembly Language Instructions 11

Software Applications 12

Hardware Applications 13

Appendices A-D

Section 1

Introduction

The TMS320C30 (third-generation) Digital Signal Processor (DSP) is a high-performance CMOS 32-bit device in the TMS320 family of single-chip digital signal processors. Since 1982 when the TMS32010 was introduced, the TMS320 family has established itself as the industry standard for digital signal processing. Powerful instruction sets, high-speed number-crunching capabilities, and innovative architectures have made this high-performance family of processors ideal for DSP applications.

The TMS320 family consists of three generations of processors: TMS320C1x, TMS320C2x, and TMS320C3x (see Figure 1-1). The family has expanded to include enhancements of earlier generations and more powerful new generations of digital signal processors.

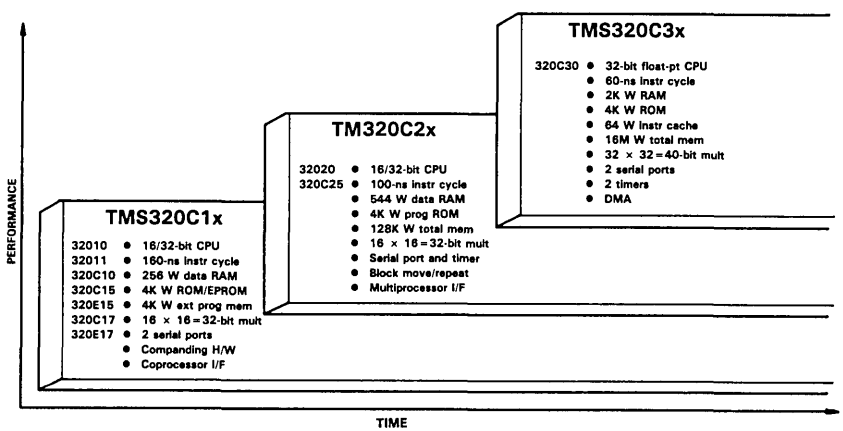


Figure 1-1. TMS320 Device Evolution

1

This document discusses the third-generation device, TMS320C30, within the TMS320 family. The 60-ns cycle time of the TMS320C30 allows it to execute operations at a performance rate previously available only on a supercomputer. Even higher performance is gained through its large on-chip memories, concurrent DMA controller, and instruction cache.

This section presents the following major topics:

- General Description (Section 1.1 on page 1-3)
- Key Features (Section 1.2 on page 1-4)
- Typical Applications (Section 1.3 on page 1-5)
- How To Use This Manual (Section 1.4 on page 1-6)
- References (Section 1.5 on page 1-8)

1.1 General Description

The TMS320's internal busing and special digital signal processing (DSP) instruction set provide speed and flexibility.This combination produces a processor family capable of executing up to 33 MFLOPS (million floating-point operations per second). The TMS320 family optimizes speed by implementing functions in hardware that other processors implement through software or microcode. This hardware-intensive approach provides the design engineer with power previously unavailable on a single chip.

The TMS320C30, the third-generation device in the TMS320 family, can perform parallel multiply and ALU operations on integer or floating-point data in a single cycle. The processor also possesses a general-purpose register file, program cache, dedicated auxiliary register arithmetic units (ARAU), internal dual-access memories, one DMA channel supporting concurrent I/O, and a short machine-cycle time. High performance and ease of use are achieved through greater parallelism, greater accuracy, and general-purpose features.

General-purpose applications are greatly enhanced by the large address space, multiprocessor interface, internally and externally generated wait states, two timers, two serial ports, and multiple interrupt structure. The TMS320C30 supports a wide variety of system applications from host processor to dedicated coprocessor.

The emphasis on total system cost has resulted in a less-expensive processor that can be designed into systems currently using costly bit-slice processors. High-level language is more easily supported through a register-based architecture, large address space, powerful addressing modes, flexible instruction set, and support of floating-point arithmetic.

1.2 Key Features

Some key features of the TMS320C30 are listed below.

- 60-ns single-cycle instruction execution time
  - 33.3 MFLOPS (million floating-point operations per second)
  - 16.7 MIPS (million instructions per second)
- One 4K x 32-bit single-cycle dual-access on-chip ROM block
- Two 1K x 32-bit single-cycle dual-access on-chip RAM blocks
- 64 x 32-bit instruction cache
- 32-bit instruction and data words, 24-bit addresses
- 40/32-bit floating-point/integer multiplier and ALU
- 32-bit barrel shifter
- Eight extended-precision registers (accumulators)
- Two address generators with eight auxiliary registers and two auxiliary register arithmetic units
- On-chip Direct Memory Access (DMA) controller for concurrent I/O and CPU operation
- Integer, floating-point, and logical operations
- Two- and three-operand instructions
- Parallel ALU and multiplier instructions in a single cycle
- Block repeat capability
- Zero-overhead loops with single-cycle branches
- Conditional calls and returns
- Interlocked instructions for multiprocessing support
- Two serial ports to support 8/16/32-bit transfers
- Two 32-bit timers
- Two general-purpose external flags, four external interrupts
- 180-pin grid array (PGA) package; 1 μ m CMOS

1.3 Typical Applications

The TMS320 family's unique versatility and realtime performance offer flexible design approaches in a variety of applications. In addition, TMS320 devices can simultaneously provide the multiple functions often required in those complex applications. Table 1-1 lists typical TMS320 family applications.

Table 1-1. Typical Applications of the TMS320 Family

GENERAL-PURPOSE DSP	GRAPHICS/IMAGING	INSTRUMENTATION
Digital Filtering Convolution Correlation Hilbert Transforms Fast Fourier Transforms Adaptive Filtering Windowing Waveform Generation	3-D Rotation Robot Vision Image Transmission/ Compression Pattern Recognition Image Enhancement Homomorphic Processing Workstations Animation/Digital Map	Spectrum Analysis Function Generation Pattern Matching Seismic Processing Transient Analysis Digital Filtering Phase-Locked Loops
VOICE/SPEECH	**CONTROL**	**MILITARY**
Voice Mail Speech Vocoding Speech Recognition Speaker Verification Speech Enhancement Speech Synthesis Text-to-Speech Neural Networks	Disk Control Servo Control Robot Control Laser Printer Control Engine Control Motor Control Kalman Filtering	Secure Communications Radar Processing Sonar Processing Image Processing Navigation Missile Guidance Radio Frequency Modems Sensor Fusion
TELECOMMUNICATIONS		**AUTOMOTIVE**
Echo Cancellation ADPCM Transcoders Digital PBXs Line Repeaters Channel Multiplexing 1200 to 19200-bps Modems Adaptive Equalizers DTMF Encoding/Decoding Data Encryption	FAX Cellular Telephones Speaker Phones Digital Speech Interpolation (DSI) X.25 Packet Switching Video Conferencing Spread Spectrum Communications	Engine Control Vibration Analysis Antiskid Brakes Adaptive Ride Control Global Positioning Navigation Voice Commands Digital Radio Cellular Telephones
CONSUMER	**INDUSTRIAL**	**MEDICAL**
Radar Detectors Power Tools Digital Audio/TV Music Synthesizer Toys and Games Solid-State Answering Machines	Robotics Numeric Control Security Access Power Line Monitors Visual Inspection Lathe Control CAM	Hearing Aids Patient Monitoring Ultrasound Equipment Diagnostic Tools Prosthetics Fetal Monitors MR Imaging

1

1.4 How To Use This Manual

The purpose of this user's guide is to serve as a reference book for the TMS320C30 digital signal processor. This document is designed to provide information that assists managers and hardware/software engineers in application development. The first group of sections provides specific information about the architecture and hardware operation of the device. Later sections describe the software operation. Specific software and hardware applications are provided in Sections 12 and 13, respectively. Electrical specifications and mechanical data can be found in the data sheet (Appendix A).

The following table lists each section and briefly describes the section contents.

Section 2. Pinout and Signal Descriptions. Drawing of the PGA package for the TMS320C30. Functional listing of the signals, their pin locations, and descriptions.

Section 3. Architectural Overview. Functional block diagram. TMS320C30 design description, hardware components, and device operation. Instruction set summary.

Section 4. CPU Registers, Memory, and Cache. Description of the registers in the CPU register file. Memory maps provided and instruction cache architecture, algorithm, and control bits explained.

Section 5. Data Formats and Floating-Point Operations. Description of signed and unsigned integer and floating-point formats. Discussion of floating-point multiplication, addition, subtraction, normalization, rounding, and conversions.

Section 6. Addressing. Operation, encoding, and implementation of addressing modes. Format descriptions. System stack management.

Section 7. Program Flow Control. Software control of program flow with repeat modes and branching. Interlocked operations. Reset and interrupts.

Section 8. External Bus Operation. Description of primary and expansion interfaces. External interface timing diagrams. Programmable wait-states and bank switching.

Section 9. Peripherals. Description of the DMA controller, timers, and serial ports.

Section 10. Pipeline Operation. Discussion of the pipelining of operations on the TMS320C30.

Section 11. Assembly Language Instructions. Functional listing of instructions. Condition codes defined. Alphabetized individual instruction descriptions with examples.

Section 12. Software Applications. Software application examples for the use of various TMS320C30 instruction set features.

Section 13. Hardware Applications. Hardware design techniques and application examples for interfacing to memories, peripherals, or other microcomputers/microprocessors.

Four appendices are included to provide additional information.

Appendix A. TMS320C30 Data Sheet. Electrical specifications, timing, and mechanical data.

Appendix B. Development Support/Part Order Information. Listings of the hardware and software available to support the TMS320C30 device.

Appendix C. Instruction Opcodes. List of the opcode fields for all the TMS320C30 instructions.

Appendix D. Quality and Reliability. Discussion of Texas Instruments quality and reliability criteria for evaluating performance.

1

1.5 References

The following reference list contains useful information regarding functions, operations, and applications of digital signal processing. These books also provide other references to many useful technical papers. The reference list is organized into categories of general DSP, speech, image processing, and digital control theory, and alphabetized by author.

General Digital Signal Processing:

Antoniou, Andreas, *Digital Filters: Analysis and Design*. New York, NY: McGraw-Hill Company, Inc., 1979.

Brigham, E. Oran, *The Fast Fourier Transform*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1974.

Burrus, C.S. and Parks, T.W., *DFT/FFT and Convolution Algorithms*. New York, NY: John Wiley and Sons, Inc., 1984.

Digital Signal Processing Applications with the TMS320 Family, Texas Instruments, 1986; Prentice-Hall, Inc., 1987.

Gold, Bernard and Rader, C.M., *Digital Processing of Signals*. New York, NY: McGraw-Hill Company, Inc., 1969.

Hamming, R.W., *Digital Filters*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1977.

IEEE ASSP DSP Committee (Editor), *Programs for Digital Signal Processing*. New York, NY: IEEE Press, 1979.

Jackson, Leland B., *Digital Filters and Signal Processing*. Hingham, MA: Kluwer Academic Publishers, 1986.

Jones, D.L. and Parks, T.W., *A Digital Signal Processing Laboratory Using the TMS32010*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

Lim, Jae and Oppenheim, Alan V. (Editors), *Advanced Topics in Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988.

Morris, L. Robert, *Digital Signal Processing Software*. Ottawa, Canada: Carleton University, 1983.

Oppenheim, Alan V. (Editor), *Applications of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

Oppenheim, Alan V. and Schafer, R.W., *Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

Oppenheim, Alan V. and Willsky, A.N. with Young, I.T., *Signals and Systems*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983.

Parks, T.W. and Burrus, C.S., *Digital Filter Design*. New York, NY: John Wiley and Sons, Inc., 1987.

Rabiner, Lawrence R., Gold and Bernard *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

Treichler, J.R., Johnson, Jr., C.R., and Larimore, M.G., *Theory and Design of Adaptive Filters*. New York, NY: John Wiley and Sons, Inc., 1987.

Speech:

Gray, A.H. and Markel, J.D., *Linear Prediction of Speech*. New York, NY: Springer-Verlag, 1976.

Jayant, N.S. and Noll, Peter, *Digital Coding of Waveforms*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Papamichalis, Panos, *Practical Approaches to Speech Coding*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

Rabiner, Lawrence R. and Schafer, R.W., *Digital Processing of Speech Signals*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

Image Processing:

Andrews, H.C. and Hunt, B.R., *Digital Image Restoration*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1977.

Gonzales, Rafael C. and Wintz, Paul, *Digital Image Processing*. Reading, MA: Addison-Wesley Publishing Company, Inc., 1977.

Pratt, William K., *Digital Image Processing*. New York, NY: John Wiley and Sons, 1978.

Digital Control Theory:

Jacquot, R., *Modern Digital Control Systems*. New York, NY: Marcel Dekker, Inc., 1981.

Katz, P., *Digital Control Using Microprocessors*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

Kuo, B.C., *Digital Control Systems*. New York, NY: Holt, Reinholt and Winston, Inc., 1980.

Moroney, P., *Issues in the Implementation of Digital Feedback Compensators*. Cambridge, MA: The MIT Press, 1983.

Phillips, C. and Nagle, H., *Digital Control System Analysis and Design*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Introduction 1

Pinout and Signal Descriptions 2

Architectural Overview 3

CPU Registers, Memory, and Cache 4

Data Formats and Floating-Point Operation 5

Addressing 6

Program Flow Control 7

External Bus Operation 8

Peripherals 9

Pipeline Operation 10

Assembly Language Instructions 11

Software Applications 12

Hardware Applications 13

Appendices A-D

Section 2

Pinout and Signal Descriptions

The TMS320C30 (third-generation TMS320) digital signal processor is available in a 180-pin grid array (PGA) package. The pinout of this package (Figure 2-1), and a functional listing of the signals, pin locations, and descriptions are provided in this section. Electrical specifications and mechanical data are given in the data sheet (Appendix A).

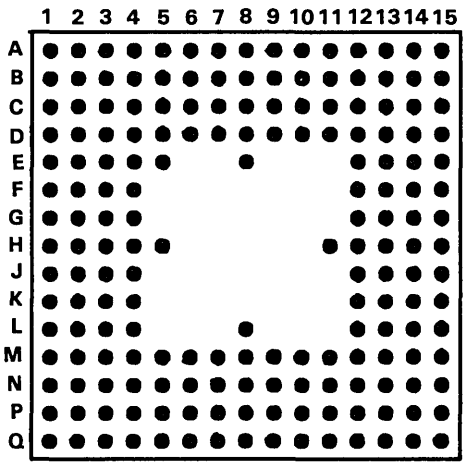


Figure 2-1. TMS320C30 Pin Assignments

Table 2-1. TMS320C30 Pin Function Assignments

Function	Pin	Function	Pin	Function	Pin	Function	Pin	Function	Pin
A0	F15	EMU0	F14	D19	A9	$\overline{IACK}$	G1	X11	D14
A1	G12	EMU1	E15	D20	B9	$\overline{INT0}$	H2	X12	E13
A2	G13	CLKR0	N4	D21	C9	$\overline{INT1}$	H1	XD0	Q4
A3	G14	CLKR1	L4	D22	A10	$\overline{INT2}$	J1	XD1	P5
A4	G15	CLKX0	M5	D23	D9	$\overline{INT3}$	J2	XOD2	N6
A5	H15	CLKX1	N2	D24	B10	RSV0	J3	XD3	Q5
A6	H14	D0	C4	D25	A11	RSV1	J4	XD4	P6
A7	J15	D1	D5	D26	C10	RSV2	K1	XD5	M7
A8	J14	D2	A2	D27	B11	RSV3	K2	XD6	Q6
A9	J13	D3	A3	D28	A12	RSV4	L1	XD7	N7
A10	K15	D4	B4	D29	D10	RSV5	K3	XD8	P7
A11	J12	D5	C5	D30	C11	RSV7	K4	XD9	Q7
A12	K14	D6	D6	D31	B12	RSV9	L3	XD10	P8
A13	L15	D7	A4	DR0	Q1	RSV10	M2	XD11	Q8
A14	K13	D8	B5	DR1	N1	XA0	A13	XD12	Q9
A15	L14	D9	C6	DX0	Q3	XA1	A14	XD13	P9
A16	M15	D10	A5	DX1	P2	XA2	D11	XD14	N9
A17	K12	D11	B6	FSR0	P3	XA3	C12	XD15	Q10
A18	L13	D12	D7	FSR1	M3	XA4	B13	XD16	M9
A19	M14	D13	A6	FSX0	Q2	IOA5	A15	XD17	P10
A20	N15	D14	C7	FSX1	P1	XA6	B15	XD18	Q11
A21	M13	D15	B7	H1	B3	XA7	C14	XD19	N10
A22	L12	D16	A7	H3	A1	XA8	E12	XD20	P11
A23	N14	D17	A8	$\overline{HOLD}$	F3	XA9	D13	XD21	Q12
EMU5	C1	D18	B8	$\overline{HOLDA}$	E2	XA10	C15	XD22	M10
IOD23	N11	LOCATOR	E5	TCLK1	N5	ADVDD	D12	VSS	N8
IOD24	P12	EMU4	F12	VBBP	D3	ADVDD	H11		
IOD25	Q13	MC/$\overline{MP}$	D15	VSUBS	E4	DDVDD	D4	CVSS	B2
IOD26	Q14	$\overline{MSTRB}$	E3	X1	C2	DDVDD	E8	CVSS	P14
IOD27	M11	EMU6	M6	X2	B1	IODVDD	L8	DVSS	C3
IOD28	N12	$\overline{RDY}$	E1	XF0	G2	IODVDD	M12	DVSS	C13
IOD29	P13	$\overline{RESET}$	F1	XF1	G3	MDVDD	H5	DVSS	N3
IOD30	Q15	R/$\overline{W}$	G4			PDVDD	M4	DVSS	N13
IOD31	P15	EMU2	F13	VDD	D8			IVSS	B14
$\overline{IORDY}$	D2	EMU3	E14	VDD	H4	VSS	C8		
IOR/$\overline{W}$	D1	$\overline{STRB}$	F2	VDD	H12	VSS	H3	RSV6	L2
$\overline{IOSTRB}$	F4	TCLK0	P4	VDD	M8	VSS	H13	RSV8	M1

NOTE:

1) ADVDD, DDVDD, IODVDD, MDVDD, and PDVDD pins (D4, D12, E8, H5, H11, L8, M4, and M12) are on a common plane internal to the device.

2) VDD pins (D8, H4, H12, and M8) are on a common plane internal to the device.

3) VSS, CVSS, and INSS pins (B2, B14, C8, H3, H13, N8, and P14) are on a common plane internal to the device.

4) DVSS pins (C3, C13, N3, and N13) are on a common plane internal to the device.

2.1 Signal Descriptions

The signal descriptions for the TMS320C30 device in the microprocessor mode are provided in this section. Table 2-2 lists each signal, the number of pins, function, and operating mode(s), i.e., input, output, or high-impedance state as indicated by I, O, or Z. All pins labelled 'NC' are not to be connected by the user. A line over a signal name (e.g., $\overline{\text{RESET}}$) indicates that the signal is active low true at a logic '0' level. The signals in Table 2-2 are grouped according to function.

Table 2-2. TMS320C30 Signal Descriptions

SIGNAL	# PINS	I/O/Z†	DESCRIPTION
PRIMARY BUS INTERFACE (61 PINS)			
D(31-0)	32	I/O/Z	32-bit data port of the primary bus interface.
A (23-0)	24	O/Z	24-bit address port of the primary bus interface.
R/$\overline{\text{W}}$	1	O/Z	Read/write signal for primary bus interface. This pin is high when a read is performed and low when a write is performed over the parallel interface.
$\overline{\text{STRB}}$	1	O/Z	External access strobe for the primary bus interface.
$\overline{\text{RDY}}$	1	I	Ready signal. This pin indicates that the external device is prepared for a primary bus interface transaction to complete. As long as $\overline{\text{RDY}}$ is a logic high, the data and address buses of the primary bus interface remain valid.
$\overline{\text{HOLD}}$	1	I	Hold signal for primary bus interface. When $\overline{\text{HOLD}}$ is a logic low, any ongoing transaction is completed. The A(23-0), D(31-0), $\overline{\text{STRB}}$,, and R/$\overline{\text{W}}$ signals are placed in a high-impedance state, and all transactions over the primary bus interface are held until $\overline{\text{HOLD}}$ becomes a logic high.
$\overline{\text{HOLDA}}$	1	O	Hold acknowledge signal for primary bus interface. This signal is generated in response to a logic low on $\overline{\text{HOLD}}$. It signals that A(23-0), D(31-0), $\overline{\text{STRB}}$, and R/$\overline{\text{W}}$ are placed in a high-impedance state and all transactions over the bus will be held. $\overline{\text{HOLDA}}$ will be high in response to a logic high of $\overline{\text{HOLD}}$.
EXPANSION BUS INTERFACE (49 PINS)			
XD (31-0)	32	I/O/Z	32-bit data port of the expansion bus interface.
XA (12-0)	13	O/Z	13-bit address port of the expansion bus interface.
XR/$\overline{\text{W}}$	1	O/Z	Read/write signal for expansion bus interface. When a read is performed, this pin is held high; when a write is performed, this pin is low.
$\overline{\text{MSTRB}}$	1	O/Z	External memory access strobe for the expansion bus interface.
$\overline{\text{IOSTRB}}$	1	O/Z	External I/O access strobe for the expansion bus interface.
$\overline{\text{XRDY}}$	1	I	Ready signal. This pin indicates that the external device is prepared for an expansion bus interface transaction to complete. As long as $\overline{\text{XRDY}}$ is high, the data and address buses of the expansion bus interface remain valid.

† Input, Output, High-impedance state.

Table 2-2. TMS320C30 Signal Descriptions (Continued)

SIGNAL	# PINS	I/O/Z†	DESCRIPTION
CONTROL SIGNALS (9 PINS)			
$\overline{\text{RESET}}$	1	I	Reset. When this pin is a logic low, the device is placed in the reset condition. When reset becomes a logic high, execution begins from the location specified by the reset vector.
$\overline{\text{INT}}$(3-0)	4	I	External interrupts.
$\overline{\text{IACK}}$	1	O	Interrupt acknowledge signal. $\overline{\text{IACK}}$ goes low during execution of an IACK instruction. This can be used to indicate the beginning or end of an interrupt service routine.
MC/$\overline{\text{MP}}$	1	I	Microcomputer/microprocessor mode pin.
XF(1-0)	2	I/O	External flag pins. These pins are formatted as I/O through a program instruction, and latched internally when used as output pins. They are used as general-purpose I/O pins or to support interlocked processor instructions.
SERIAL PORT 0 SIGNALS (6 PINS)			
CLKX0	1	I/O	Serial port 0 transmit clock. This pin serves as the serial shift clock for the serial port 0 transmitter.
DX0	1	O/Z	Data transmit output. Serial port 0 transmits serial data on this pin.
FSX0	1	I/O	Frame synchronization pulse for transmit. The FSX0 pulse initiates the transmit data process over pin DX0.
CLKR0	1	I/O	Serial port 0 receive clock.This pin serves as the serial shift clock for the serial port 0 receiver.
DR0	1	I	Data receive. Serial port 0 receives serial data via the DR0 pin.
FSR0	1	I	Frame sychronization pulse for receive. The FSR0 pulse initiates the receive data process over DR0.
SERIAL PORT 1 SIGNALS (6 PINS)			
CLKX1	1	I/O	Serial port 1 transmit clock. This pin serves as the serial shift clock for the serial port 1 transmitter.
DX1	1	O/Z	Data transmit output. Serial port 1 transmits serial data on this pin.
FSX1	1	I/O	Frame synchronization pulse for transmit. The FSX1 pulse initiates the transmit data process over pin DX1.
CLKR1	1	I/O	Serial port 1 receive clock. This pin serves as the serial shift clock for the serial port 1 receiver.
DR1	1	I	Data receive. Serial port 1 receives serial data via the DR1 pin.
FSR1	1	I	Frame sychronization pulse for receive. The FSR1 pulse initiates the receive data process over DR1.

† Input, Output, High-impedance state.

Table 2-2. TMS320C30 Signal Descriptions (Continued)

SIGNAL	# PINS	I/O/Z†	DESCRIPTION
TIMER 0 SIGNALS (1 PIN)			
TCLK0	1	I/O	Timer clock. As an input, TCLK0 is used by timer 0 to count external pulses. As an output pin, TCLK0 outputs pulses generated by timer 0.
TIMER 1 SIGNALS (1 PIN)			
TCLK1	1	I/O	Timer clock. As an input, TCLK1 is used by timer 1 to count external pulses. As an output pin, TCLK1 outputs pulses generated by timer 1.
SUPPLY AND OSCILLATOR SIGNALS (29 PINS)			
V_{DD}(3-0)	4	I	Four +5 V supply pins.
$IODV_{DD}$(1,0)	2	I	Two +5 V supply pins.
ADV_{DD}(1,0)	2	I	Two +5 V supply pins.
PDV_{DD}	1	I	One +5 V supply pin.
DDV_{DD}(1,0)	2	I	Two +5 V supply pins.
MDV_{DD}	1	I	One +5 V supply pin.
V_{SS}(3-0)	4	I	Four ground pins.
DV_{SS}(3-0)	4	I	Four ground pins.
CV_{SS}(1,0)	2	I	Two ground pins.
IV_{SS}	1	I	One ground pin.
V_{BBP}	1	NC	V_{BB} pump oscillator output.
SUBS	1	I	Substrate pin. Tie to ground.
X1	1	O	Output pin from the internal oscillator for the crystal. If a crystal is not used, this pin should be left unconnected.
X2/CLKIN	1	I	Input pin to the internal oscillator from the crystal or a clock.
H1	1	O	External H1 clock. This clock has a period equal to twice CLKIN.
H3	1	O	External H3 clock. This clock has a period equal to twice CLKIN.

† Input, Output, High-impedance state.

Table 2-2. TMS320C30 Signal Descriptions (Concluded)

SIGNAL	# PINS	I/O/Z†	DESCRIPTION
RESERVED (18 PINS)			
EMU(0-2)	3	I	Reserved. Use pull-ups to +5 volts. See Section 13.5
EMU3	1	O	Reserved. See Section 13.5
EMU4	1	I	Reserved. Tie to +5 volts.
EMU(5,6)	2	NC	Reserved.
RSV(0-10)	11	I	Reserved. Tie to +5 volts.

† Input, Output, High-impedance state.

The user must follow the connections specified for the reserved pins. All pull-up resistors must be 20 k ohms. All +5 volt supply pins must be connected to a common supply plane and all ground pins must be connected to a common ground plane.

Introduction 1

Pinout and Signal Descriptions 2

Architectural Overview 3

CPU Registers, Memory, and Cache 4

Data Formats and Floating-Point Operation 5

Addressing 6

Program Flow Control 7

External Bus Operation 8

Peripherals 9

Pipeline Operation 10

Assembly Language Instructions 11

Software Applications 12

Hardware Applications 13

Appendices A-D

Section 3

Architectural Overview

Emphasis on hardware and software system solutions to demanding arithmetic algorithms has resulted in the TMS320C30 architecture shown in Figure 3-1. High system performance is achieved through the accuracy and precision of the floating-point units, large on-chip memory, a high degree of parallelism, and the DMA controller.

This section provides an architectural overview of the TMS320C30 processor. Major areas of discussion are listed below.

- Central Processing Unit (CPU) (Section 3.1 on page 3-3)
  - Floating-point/integer multiplier
  - ALU for floating-point, integer, and logical operations
  - Auxiliary register arithmetic units (ARAUs)
  - CPU register file
- Memory Organization (Section 3.2 on page 3-7)
  - RAM, ROM, and cache
  - Memory maps
  - Memory addressing modes
  - Instruction set summary
- Internal Bus Operation (Section 3.3 on page 3-18)
- External Bus Operation (Section 3.4 on page 3-19)
- Peripherals (Section 3.5 on page 3-20)
  - Timers
  - Serial ports
- Direct Memory Access (DMA) (Section 3.6 on page 3-21)

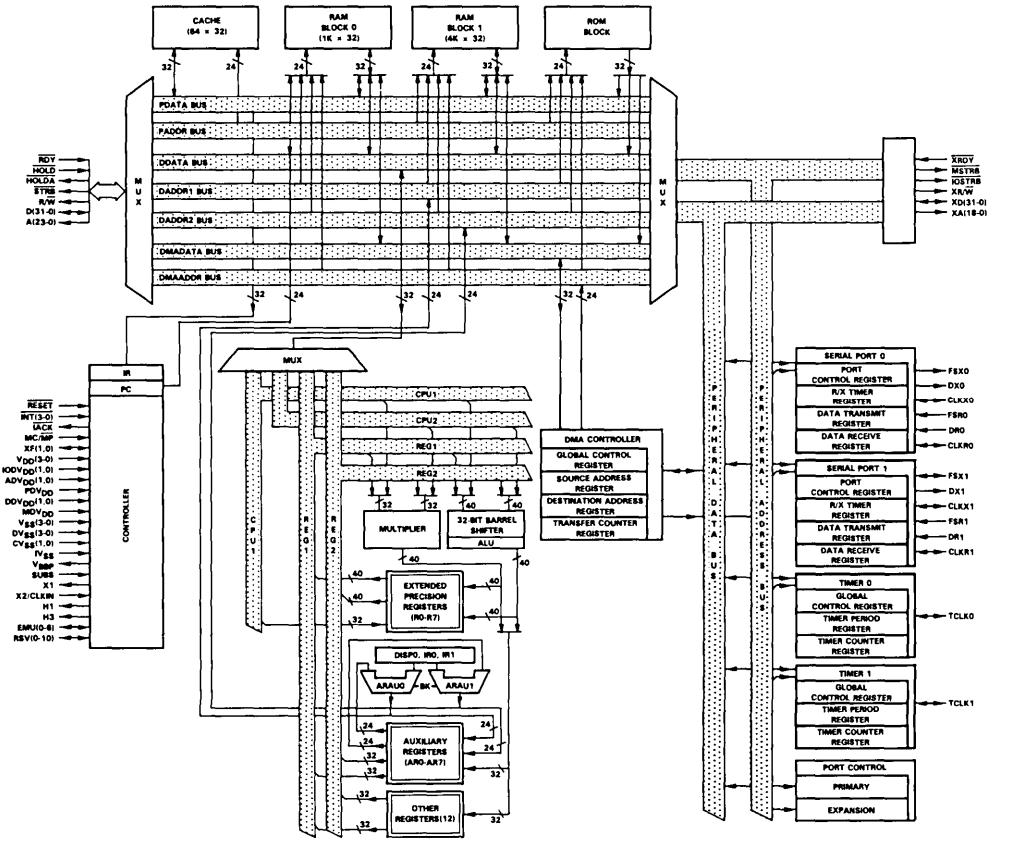


Figure 3-1. TMS320C30 Block Diagram

3.1 Central Processing Unit (CPU)

The TMS320C30 has a register-based CPU architecture. The CPU consists of the following components:

- Floating-point/integer multiplier
- ALU for performing arithmetic (floating-point, integer)and logical operations
- 32-bit barrel shifter
- Internal buses (CPU1/CPU2 and REG1/REG2)
- Auxiliary register arithmetic units (ARAUs)
- CPU register file.

Figure 3-2 shows the various CPU components that are discussed in the succeeding subsections.

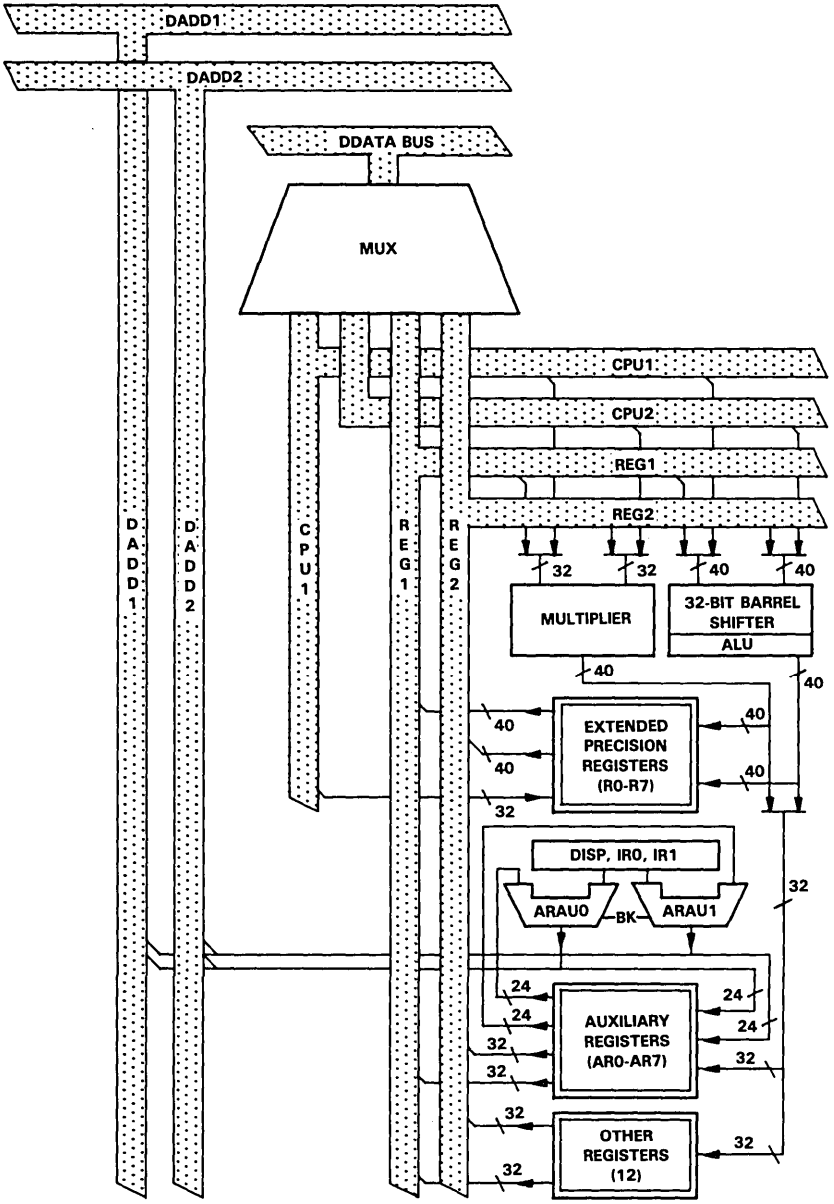


Figure 3-2. Central Processing Unit (CPU)

3.1.1 Multiplier

The multiplier performs single-cycle multiplications on 24-bit integer and 32-bit floating-point values. The TMS320C30 implementation of floating-point arithmetic allows for floating-point operations at fixed-point speeds via a 60-ns instruction cycle and a high degree of parallelism. To gain even higher throughput, a multiply and ALU operation can be performed in a single cycle by using parallel instructions.

When performing floating-point multiplication, the inputs are 32-bit floating-point numbers, and the result is a 40-bit floating-point number. When performing integer multiplication, the input data is 24 bits and yields a 32-bit result. Refer to Section 5 for detailed information on data formats and floating-point operation.

3.1.2 Arithmetic Logic Unit (ALU)

The ALU performs single-cycle operations on 32-bit integer, 32-bit logical, and 40-bit floating-point data, including single-cycle integer and floating-point conversions. Results of the ALU are always maintained in 32-bit integer or 40-bit floating-point formats. The barrel shifter is used to shift up to 32 bits left or right in a single cycle.

Internal buses, CPU1/CPU2 and REG1/REG2, carry two operands from memory and two operands from the register file, thus allowing parallel multiplies and adds/subtracts on four integer or floating-point operands in a single cycle.

3.1.3 Auxiliary Register Arithmetic Units (ARAUs)

Two auxiliary register arithmetic units (ARAU0 and ARAU1) can generate two addresses in a single cycle. The ARAUs operate in parallel with the multiplier and ALU. They support addressing with displacements, index registers (IR0 and IR1), and circular and bit-reversed addressing. Refer to Section 6 for a description of addressing modes.

3.1.4 CPU Register File

The TMS320C30 provides 28 registers in a multiport register file that is tightly coupled to the CPU. All of these registers can be operated upon by the multiplier and ALU, and can be used as general-purpose registers. However, the registers also have some special functions for which they are more suited than others. For example, the eight extended-precision registers are especially suited for maintaining extended-precision floating-point results. The eight auxiliary registers support a variety of indirect addressing modes and can be used as general-purpose 32-bit integer and logical registers. The remaining registers provide system functions such as addressing, stack management, processor status, interrupts, and block repeat. Refer to Section 6 for detailed information and examples of stack management and register usage.

The registers names and assigned functions are listed in Table 3-1. Following the table, the function of each register or group of registers will be briefly described. Refer to Section 4 for detailed information on each of the CPU registers.

Table 3-1. CPU Registers

REGISTER NAME	ASSIGNED FUNCTION
R0	Extended-precision register 0
R1	Extended-precision register 1
R2	Extended-precision register 2
R3	Extended-precision register 3
R4	Extended-precision register 4
R5	Extended-precision register 5
R6	Extended-precision register 6
R7	Extended-precision register 7
AR0	Auxiliary register 0
AR1	Auxiliary register 1
AR2	Auxiliary register 2
AR3	Auxiliary register 3
AR4	Auxiliary register 4
AR5	Auxiliary register 5
AR6	Auxiliary register 6
AR7	Auxiliary register 7
DP	Data page pointer
IR0	Index register 0
IR1	Index register 1
BK	Block size
SP	System stack pointer
ST	Status register
IE	CPU/DMA interrupt enable
IF	CPU interrupt flags
IOF	I/O flags
RS	Repeat start address
RE	Repeat end address
RC	Repeat counter
PC	Program Counter

The **extended-precision registers (R0-R7)** are capable of storing and supporting operations on 32-bit integer and 40-bit floating-point numbers. Any instruction that assumes the operands are floating-point numbers uses bits 39-0. If the operands are either signed or unsigned integers, only bits 31-0 are used, bits 39-32 remain unchanged. This is true for all shift operations. Refer to Section 4 for extended-precision register formats for floating-point and integer numbers.

The 32-bit **auxiliary registers (AR0-AR7)** can be accessed by the CPU and modified by the two Auxiliary Register Arithmetic Units (ARAUs). The primary function of the auxiliary registers is the generation of 24-bit addresses. They can also be used to perform a variety of functions, such as loop counters or as 32-bit general-purpose registers that can be modified by the multiplier and ALU. Refer to Section 6 for detailed information and examples of the use of auxiliary registers in addressing.

The **data page pointer (DP)** is a 32-bit register. The eight LSBs of the data page pointer are used by the direct addressing mode as a pointer to the page of data being addressed. Data pages are 64 k words long with a total of 256 pages.

The 32-bit **index registers (IR0 and IR1)** are used by the Auxiliary Register Arithmetic Unit (ARAU) for indexing the address. Refer to Section 6 for examples of the use of index registers in addressing.

The 32-bit **block size register (BK)** is used by the ARAU in circular addressing to specify the data block size.

The **system stack pointer (SP)** is a 32-bit register that contains the address of the top of the system stack. The SP always points to the last element pushed onto the stack. A push performs a preincrement and a pop, a postdecrement of the system stack pointer. The SP is manipulated by interrupts, traps, calls, returns, and the PUSH and POP instructions. Refer to Section 6.5 for information about system stack management.

The **status register (ST)** *contains global information relating to the state* of the CPU. Typically, operations set the condition flags of the status register according to whether the result is zero, negative, etc. This includes register load and store operations as well as arithmetic and logical functions. When the status register is loaded, however, a bit-for-bit replacement is performed on the current contents with the contents of the source operand regardless of the state of any bits in the source operand. Therefore, following a load, the contents of the status register are identically equal to the contents of the source operand. This allows the status register to be easily saved and restored. See Table 4.2 for a list and definitions of the status register bits.

The **CPU/DMA interrupt enable register (IE)** is a 32-bit register. The CPU interrupt enable bits are in locations 10-0. The DMA interrupt enable bits are in locations 26-16. A 1 in a CPU/DMA interrupt enable register bit enables the corresponding interrupt. A 0 disables the corresponding interrupt. Refer to Section 4.1 for bit definitions.

The **CPU interrupt flag register (IF)** is also a 32-bit register (see Section 4.1). A 1 in a CPU interrupt flag register bit indicates that the corresponding interrupt is set. A 0 indicates that the corresponding interrupt is not set.

The **I/O flags register (IOF)** controls the function of the dedicated external pins, XF0 and XF1. These pins may be configured for input or output, and they may also be read from and written to. See Section 4.1 for detailed information.

The **repeat counter (RC)** is a 32-bit register used to specify the number of times a block of code is to be repeated when performing a block repeat. When operating in the repeat mode, the 32-bit **repeat start address register (RS)** contains the starting address of the block of program memory to be repeated and the 32-bit **repeat end address register (RE)** contains the ending address of the block to be repeated.

The **program counter (PC)** is a 32-bit register containing the address of the next instruction to be fetched. Although the PC is not part of the CPU register file, it is a register that can be modified by instructions that modify the program flow.

3.2 Memory Organization

The total memory space of the TMS320C30 is 16M (million) 32-bit words. Program, data, and I/O space are contained within this 16M-word address space, thus allowing tables, coefficients, program code, or data to be stored in either RAM or ROM. In this way, memory usage can be maximized and memory space allocated as desired.

3.2.1 RAM, ROM, and Cache

Figure 3-3 shows how the memory is organized on the TMS320C30. RAM blocks 0 and 1 are each 1K x 32 bits. The ROM block is 4K x 32 bits. Each RAM and ROM block is capable of supporting two accesses in a single cycle. The separate program buses, data buses, and DMA buses allow for parallel program fetches, data reads and writes, and DMA operations. For example: the CPU can access two data values in one RAM block and perform an external program fetch in parallel with the DMA loading another RAM block, all within a single cycle.

A 64 x 32-bit instruction cache is provided to store often repeated sections of code, thus greatly reducing the number of off-chip accesses necessary. This allows for code to be stored off-chip in slower, lower-cost memories. The external buses are also freed for use by the DMA, external memory fetches, or other devices in the system.

Refer to Section 4 for detailed information about the memory and instruction cache.

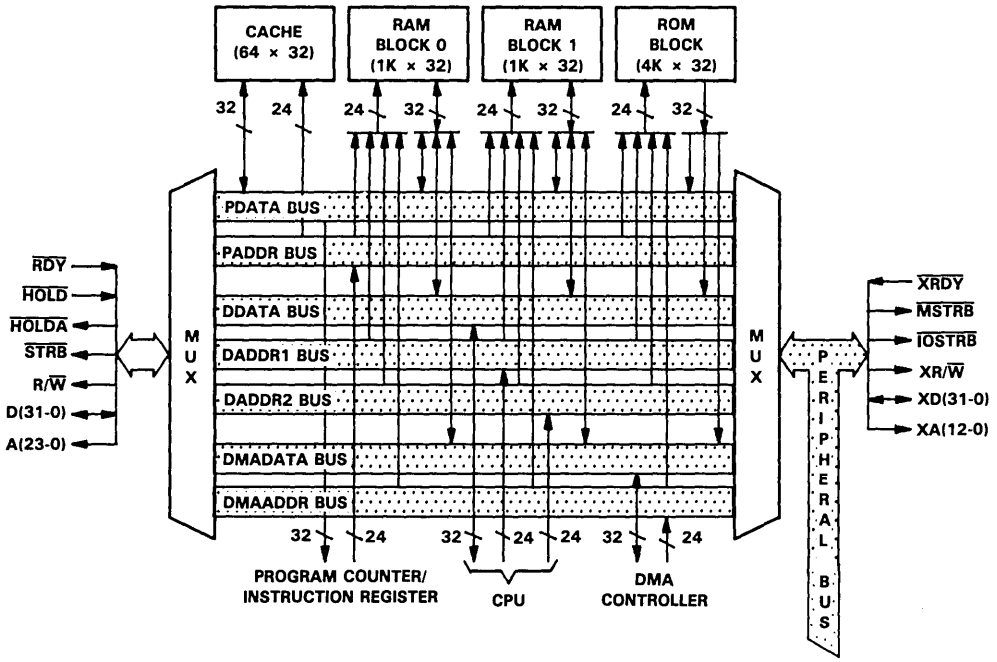


Figure 3-3. Memory Organization

3.2.2 Memory Maps

The memory map is dependent upon whether the processor is running in the microprocessor mode (MC/$\overline{MP}$ = 0) or the microcomputer mode (MC/$\overline{MP}$ = 1). The memory maps for these modes are very similar (see Figure 3-4). Locations 800000h through 801FFFh are mapped to the expansion bus. When this region is accessed, $\overline{MSTRB}$ is active. Locations 802000h through 803FFFh are reserved. Locations 804000h through 805FFFh are mapped to the expansion bus. When this region is accessed, $\overline{IOSTRB}$ is active. Locations 806000h through 807FFFh are reserved. All of the memory-mapped peripheral registers are in locations 808000h through 8097FFh. In both modes, RAM block 0 is located at addresses 809800h through 809BFFh, and RAM block 1 is located at addresses 809C00h through 809FFFh. Locations 80A000h through 0FFFFFFh are accessed over the external memory port ($\overline{STRB}$ active).

In microprocessor mode, the 4K on-chip ROM is not mapped into the TMS320C30 memory map. Locations 0h through 3Fh consist of interrupt vector, trap vector, and reserved locations, all of which are accessed over the external memory port ($\overline{STRB}$ active). Locations 40h through 7FFFFFh are also accessed over the external memory port.

In microcomputer mode, the 4K on-chip ROM is mapped into locations 0h through 0FFFh. There are 192 locations (0h through BFh) within this block for interrupt vectors, trap vectors, and a reserved space. Locations 1000h through 7FFFFFh are accessed over the external memory port ($\overline{STRB}$ active).

Section 4.2 describes the memory maps in greater detail. The peripheral bus map and the vector locations for reset, interrupts, and traps are also given.

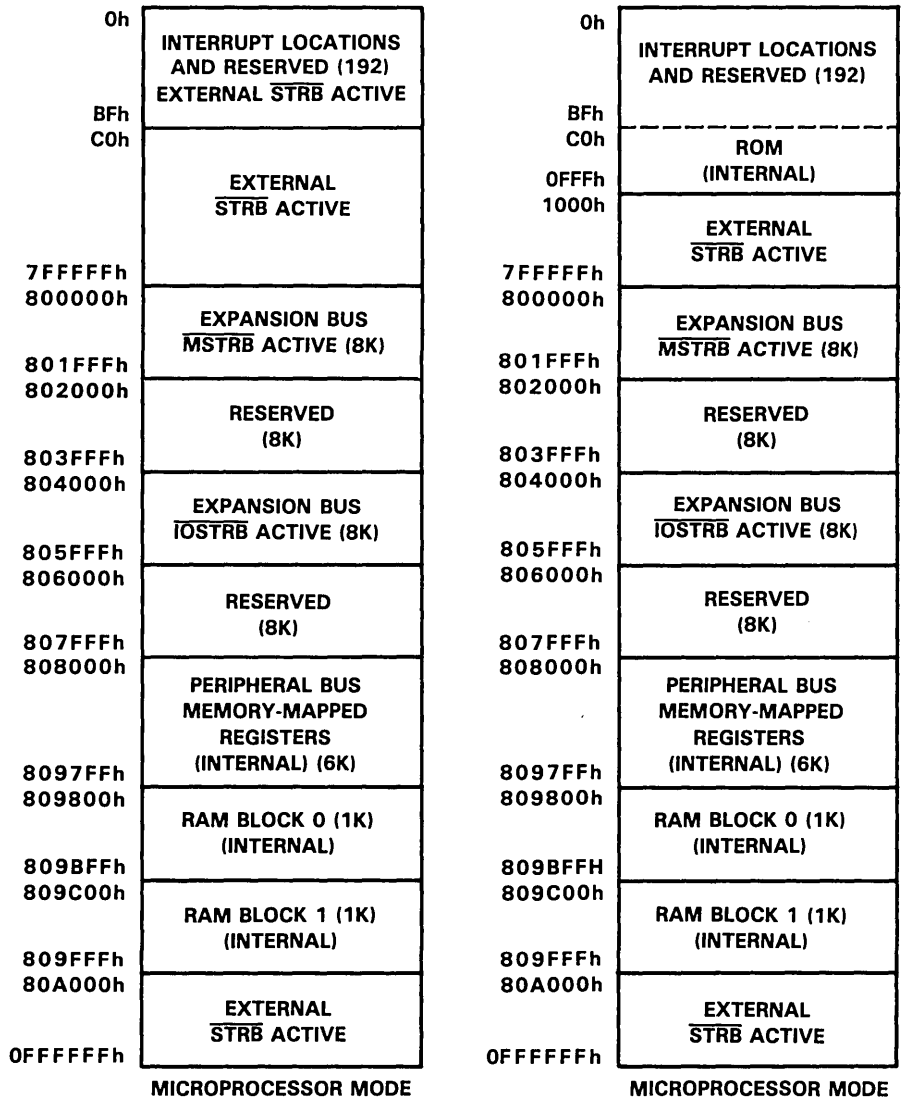


Figure 3-4. Memory Maps

3.2.3 Memory Addressing Modes

The TMS320C30 supports a base set of general-purpose instructions as well as arithmetic-intensive instructions that are particularly suited for digital signal processing and other numeric-intensive applications. Refer to Section 6 for detailed information on addressing.

Five groups of addressing modes are provided on the TMS320C30. Six types of addressing may be used within the groups, as shown in the following list:

- General addressing modes:
  - Register. The operand is a CPU register.
  - Short immediate. The operand is a 16-bit immediate value.
  - Direct. The operand is the contents of a 24-bit address.
  - Indirect. An auxiliary register indicates the address of the operand.
- Three-operand addressing modes:
  - Register. Same as for general addressing mode.
  - Indirect. Same as for general addressing mode.
- Parallel addressing modes:
  - Register. The operand is an extended-precision register.
  - Indirect. Same as for general addressing mode.
- Long-immediate addressing mode.
  - Long immediate. The operand is a 24-bit immediate value.
- Conditional branch addressing modes:
  - Register. Same as for general addressing mode.
  - PC-relative. A signed 16-bit displacement is added to the PC.

3.2.4 Instruction Set Summary

Table 3-2 lists the TMS320C30 instruction set in alphabetical order. Each table entry shows the instruction mnemonic, description, and operation. Refer to Section 11 for a functional listing of the instructions and individual instruction descriptions.

Table 3-2. Instruction Set Summary

MNEMONIC	DESCRIPTION	OPERATION
ABSF	Absolute value of a floating-point number	|src| → Rn
ABSI	Absolute value of an integer	|src| → Dreg
ADDC	Add integers with carry	src + Dreg + C → Dreg
ADDC3	Add integers with carry (3-operand)	src1 + src2 + C → Dreg
ADDF	Add floating-point values	src + Rn → Rn
ADDF3	Add floating-point values (3-operand)	src1 + src2 → Rn
ADDI	Add integers	src + Dreg → Dreg
ADDI3	Add integers (3-operand)	src1 + src2 + → Dreg
AND	Bitwise logical-AND	Dreg AND src → Dreg
AND3	Bitwise logical-AND (3-operand)	src1 AND src2 → Dreg
ANDN	Bitwise logical-AND with complement	Dreg AND $\overline{\text{src}}$ → Dreg
ANDN3	Bitwise logical-ANDN (3-operand)	src1 AND $\overline{\text{src2}}$ → Dreg
ASH	Arithmetic shift	If count ≥ 0: (Shift Dreg left by count) → Dreg Else: (Shift Dreg right by |count|) → Dreg
ASH3	Arithmetic shift (3-operand)	If count ≥ 0: (Shift src left by count) → Dreg Else: (Shift src right by |count|) → Dreg
B*cond*	Branch conditionally (standard)	If cond = true: If Csrc is a register, Csrc → PC If Csrc is a value, Csrc + PC → PC Else, PC + 1 → PC
B*cond*D	Branch conditionally (delayed)	If cond = true: If Csrc is a register, Csrc → PC If Csrc is a value, Csrc + PC + 3 → PC Else, PC + 1 → PC

LEGEND:

src – general addressing modes
src1 – three-operand addressing modes
src2 – three-operand addressing modes
Csrc – conditional-branch addressing modes
Sreg – register address (any register)
count – shift value (general addressing modes)
SP – stack pointer
GIE – global interrupt enable register
RM – repeat mode bit
TOS – top of stack

Dreg – register address (any register)
Rn – register address (R0-R7)
Daddr – destination memory address
ARn – auxiliary register n (AR0-AR7)
addr – 24-bit immediate address (label)
cond – condition code (see Section 11)
ST – status register
RE – repeat interrupt register
RS – repeat start register
PC – program counter

Table 3-2. Instruction Set Summary (Continued)

MNEMONIC	DESCRIPTION	OPERATION
BR	Branch unconditionally (standard)	Value → PC
BRD	Branch unconditionally (delayed)	Value → PC
CALL	Call subroutine	PC + 1 → TOS Value → PC
CALL*cond*	Call subroutine conditionally	If cond = true: PC + 1 → TOS If Csrc is a register, Csrc → PC If Csrc is a value, Csrc + PC → PC Else, PC + 1 → PC
CMPF	Compare floating-point values	Set flags on Rn - src
CMPF3	Compare floating-point values (3-operand)	Set flags on src1 - src2
CMPI	Compare integers	Set flags on Dreg - src
CMPI3	Compare integers (3-operand)	Set flags on src1 - src2
DB*cond*	Decrement and branch conditionally (standard)	ARn - 1 → ARn If cond = true and ARn ≥ 0: If Csrc is a register, Csrc → PC If Csrc is a value, Csrc + PC → PC Else, PC + 1 → PC
DB*cond*D	Decrement and branch conditionally (delayed)	ARn - 1 → ARn If cond = true and ARn ≥ 0: If Csrc is a register, Csrc → PC If Csrc is a value, Csrc + PC + 3 → PC Else, PC + 1 → PC
FIX	Convert floating-point value to integer	Fix (src) → Dreg
FLOAT	Convert integer to floating-point value	Float(src) → Rn
IDLE	Idle until interrupt	PC + 1 → PC Idle until next interrupt
LDE	Load floating-point exponent	src(exponent) → Rn(exponent)
LDF	Load floating-point value	src → Rn

LEGEND:

src – general addressing modes
src1 – three-operand addressing modes
src2 – three-operand addressing modes
Csrc – conditional-branch addressing modes
Sreg – register address (any register)
count – shift value (general addressing modes)
SP – stack pointer
GIE – global interrupt enable register
RM – repeat mode bit
TOS – top of stack
Dreg – register address (any register)
Rn – register address (R0-R7)
Daddr – destination memory address
ARn – auxiliary register n (AR0-AR7)
addr – 24-bit immediate address (label)
cond – condition code (see Section 11)
ST – status register
RE – repeat interrupt register
RS – repeat start register
PC – program counter

Table 3-2. Instruction Set Summary (Continued)

MNEMONIC	DESCRIPTION	OPERATION
LDF*cond*	Load floating-point value conditionally	If cond = true, src → Rn Else, Rn is not changed
LDFI	Load floating-point value, interlocked	Signal interlocked operation src → Rn
LDI	Load integer	src → Dreg
LDI*cond*	Load integer conditionally	If cond = true, src → Dreg Else, Dreg is not changed
LDII	Load integer, interlocked	Signal interlocked operation src → Dreg
LDM	Load floating-point mantissa	src(mantissa) → Rn(mantissa)
LSH	Logical shift	If count ≥ 0: (Dreg left-shifted by count) → Dreg Else: (Dreg right-shifted by |count|) → Dreg
LSH3	Logical shift (3-operand)	If count ≥ 0: (src left-shifted by count) → Dreg Else: (src right-shifted by |count|) → Dreg
MPYF	Multiply floating-point values	src × Rn → Rn
MPYF3	Multiply floating-point values (3-operand)	src1 × src2 → Rn
MPYI	Multiply integers	src × Dreg → Dreg
MPYI3	Multiply integers (3-operand)	src1 × src2 → Dreg
NEGB	Negate integer with borrow	0 - src - C → Dreg
NEGF	Negate floating-point value	0 - src → Rn
NEGI	Negate integer	0 - src → Dreg
NOP	No operation	Modify src if specified
NORM	Normalize floating-point value	Normalize (src) →Rn
NOT	Bitwise logical-complement	$\overline{\text{src}}$ → Dreg
OR	Bitwise logical-OR	Dreg OR src → Dreg
OR3	Bitwise logical-OR (3-operand)	src1 OR src2 → Dreg

LEGEND:

src – general addressing modes
src1 – three-operand addressing modes
src2 – three-operand addressing modes
Csrc – conditional-branch addressing modes
Sreg – register address (any register)
count – shift value (general addressing modes)
SP – stack pointer
GIE – global interrupt enable register
RM – repeat mode bit
TOS – top of stack
Dreg – register address (any register)
Rn – register address (R0-R7)
Daddr – destination memory address
ARn – auxiliary register n (AR0-AR7)
addr – 24-bit immediate address (label)
cond – condition code (see Section 11)
ST – status register
RE – repeat interrupt register
RS – repeat start register
PC – program counter

Table 3-2. Instruction Set Summary (Continued)

MNEMONIC	DESCRIPTION	OPERATION
POP	Pop integer from stack	*SP--→ Dreg
POPF	Pop floating-point value from stack	*SP--→ Rn
PUSH	Push integer on stack	Sreg →*++ SP
PUSHF	Push floating-point value on stack	Rn → *++ SP
RETI*cond*	Return from interrupt conditionally	If cond = true or missing: *SP--→ PC 1 → ST (GIE) Else, continue
RETS*cond*	Return from subroutine conditionally	If cond = true or missing: *SP-- → PC Else, continue
RND	Round floating-point value	Round (src) → Rn
ROL	Rotate left	Dreg rotated left 1 bit → Dreg
ROLC	Rotate left through carry	Dreg rotated left 1 bit through carry → Dreg
ROR	Rotate right	Dreg rotated right 1 bit → Dreg
RORC	Rotate right through carry	Dreg rotated right 1 bit thru carry → Dreg
RPTB	Repeat block of instructions	src → RE 1 → ST (RM) Next PC → RS
RPTS	Repeat single instruction	src → RC 1 → ST (RM) Next PC → RS Next PC → RE
SIGI	Signal, interlocked	Signal interlocked operation Wait for interlock acknowledge Clear interlock
STF	Store floating-point value	Rn → Daddr
STFI	Store floating-point value, interlocked	Rn → Daddr Signal end of interlocked operation
STI	Store integer	Sreg → Daddr
STII	Store integer, interlocked	Sreg → Daddr Signal end of interlocked operation

LEGEND:

src – general addressing modes
src1 – three-operand addressing modes
src2 – three-operand addressing modes
Csrc – conditional-branch addressing modes
Sreg – register address (any register)
count – shift value (general addressing modes)
SP – stack pointer
GIE – global interrupt enable register
RM – repeat mode bit
TOS – top of stack

Dreg – register address (any register)
Rn – register address (R0-R7)
Daddr – destination memory address
ARn – auxiliary register n (AR0-AR7)
addr – 24-bit immediate address (label)
cond – condition code (see Section 11)
ST – status register
RE – repeat interrupt register
RS – repeat start register
PC – program counter

Table 3-2. Instruction Set Summary (Continued)

MNEMONIC	DESCRIPTION	OPERATION
SUBB	Subtract integers with borrow	Dreg - src - C → Dreg
SUBB3	Subtract integers with borrow (3-operand)	src1 - src2 - C → Dreg
SUBC	Subtract integers conditionally	If Dreg - src ≥ 0: [(Dreg-src) << 1] OR 1 → Dreg Else, Dreg << 1 → Dreg
SUBF	Subtract floating-point values	Rn - src → Rn
SUBF3	Subtract floating-point values (3-operand)	src1 - src2 → Rn
SUBI	Subtract integers	Dreg - src → Dreg
SUBI3	Subtract integers (3-operand)	src1 - src2 → Dreg
SUBRB	Subtract reverse integer with borrow	src - Dreg - C → Dreg
SUBRF	Subtract reverse floating-point value	src - Rn → Rn
SUBRI	Subtract reverse integer	src - Dreg → Dreg
SWI	Software interrupt	Perform emulator interrupt sequence
TRAP*cond*	Trap conditionally	If cond = true or missing: Next PC → * ++ SP Trap vector N → PC 0 → ST (GIE) Else, continue
TSTB	Test bit fields	Dreg AND src
TSTB3	Test bit fields (3-operand)	src1 AND src2
XOR	Bitwise exclusive-OR	Dreg XOR src → Dreg
XOR3	Bitwise exclusive-OR (3-operand)	src1 XOR src2 → Dreg

LEGEND:

src – general addressing modes
src1 – three-operand addressing modes
src2 – three-operand addressing modes
Csrc – conditional-branch addressing modes
Sreg – register address (any register)
count – shift value (general addressing modes)
SP – stack pointer
GIE – global interrupt enable register
RM – repeat mode bit
TOS – top of stack

Dreg – register address (any register)
Rn – register address (R0-R7)
Daddr – destination memory address
ARn – auxiliary register n (AR0-AR7)
addr – 24-bit immediate address (label)
cond – condition code (see Section 11)
ST – status register
RE – repeat interrupt register
RS – repeat start register
PC – program counter

Table 3-2. Instruction Set Summary (Continued)

MNEMONIC	DESCRIPTION	OPERATION
PARALLEL ARITHMETIC WITH STORE INSTRUCTIONS		
ABSF || STF	Absolute value of a floating-point	|src2| → dst1 || src3 → dst2
ABSI || STI	Absolute value of an integer	|src2| → dst1 || src3 → dst2
ADDF3 || STF	Add floating-point	src1 + src2 → dst1 || src3 → dst2
ADDI3 || STI	Add integer	src1 + src2 → dst1 || src3 → dst2
AND3 || STI	Bitwise logical-AND	src1 AND src2 → dst1 || src3 → dst2
ASH3 || STI	Arithmetic shift	If count ≥ 0: src2 << count → dst1 || src3 → dst2 Else: src2 >> |count| → dst1 || src3 → dst2
FIX || STI	Convert floating-point to integer	Fix(src2) → dst1 || src3 → dst2
FLOAT || STF	Convert integer to floating-point	Float(src2) → dst1 || src3 → dst2
LDF || STF	Load floating-point	src2 → dst1 || src3 →dst2
LDI || STI	Load integer	src2 → dst1 || src3 → dst2
LSH3 || STI	Logical shift	If count ≥ 0: src2 << count → dst1 || src3 → dst2 Else: src2 >> |count| → dst1 || src3 → dst2
MPYF3 || STF	Multiply floating-point	src1 × src2 → dst1 || src3 → dst2
MPYI3 || STI	Multiply integer	src1 x src2 → dst1 || src3 → dst2
NEGF || STF	Negate floating-point	0- src2 → dst1 || src3 → dst2

LEGEND:

src1 – register addr (R0-R7)
src2– indirect addr (disp = 0, 1, IR0, IR1)
src3 – register addr (R0-R7)
src4 – indirect addr (disp = 0, 1, IR0, IR1)
dst1 – register addr (R0-R7)
dst2 – indirect addr (disp = 0, 1, IR0, IR1)

Table 3-2. Instruction Set Summary (Concluded)

MNEMONIC	DESCRIPTION	OPERATION
PARALLEL ARITHMETIC WITH STORE INSTRUCTIONS (Concluded)		
NEGI || STI	Negate integer	0 - src2 → dst1 || src3 → dst2
NOT3 || STI	Complement	src1 → dst1 || src3 → dst2
OR3 || STI	Bitwise logical-OR	src1 OR src2 → dst1 ||src3 → dst2
STF || STF	Store floating-point	src1 → dst1 ||src3 → dst2
STI || STI	Store integer	src1 → dst1 || src3 → dst2
SUBF3 || STF	Subtract floating-point	src1 - src2 → dst1 || src3 → dst2
SUBI3 || STI	Subtract integer	src1 - src2 → dst1 || src3 → dst2
XOR3 || STI	Bitwise exclusive-OR	src1 XOR src2 → dst1 || src3 → dst2
PARALLEL LOAD INSTRUCTIONS		
LDF || LDF	Load floating-point	src2 → dst1 || src4 → dst2
LDI || LDI	Load integer	src2 → dst1 || src4 → dst2
PARALLEL MULTIPLY AND ADD/SUBTRACT INSTRUCTIONS		
MPYF3 || ADDF3	Multiply and add floating-point	op1 x op2 → op3 || op4 + op5 → op6
MPYF3 || SUBF3	Multiply and subtract floating-point	op1 x op2 → op3 || op4 - op5 → op6
MPYI3 || ADDI3	Multiply and add integer	op1 x op2 → op3 || op4 + op5 → op6
MPYI3 || SUBI3	Multiply and subtract integer	op1 x op2 → op3 || op4 - op5 → op6

LEGEND:

src1 – register addr (R0-R7)
src2 – indirect addr (disp = 0, 1, IR0, IR1)
src3 – register addr (R0-R7)
src4 – indirect addr (disp = 0, 1, IR0, IR1)
dst1 – register addr (R0-R7)
dst2 – indirect addr (disp = 0, 1, IR0, IR1)
op3 – register addr (R0 or R1)
op6 – register addr (R2 or R3)

op1,op2,op4,op5 - Two of these operands must be specified using register addr. and two must be specified using indirect

3.3 Internal Bus Operation

A large portion of the TMS320C30's high performance is due to the internal busing and the parallelism possible because of this busing. The separate program buses (PADDR and PDATA), data buses (DADDR1, DADDR2, and DDATA), and DMA buses (DMAADDR and DMADATA) allow for parallel program fetches, data accesses, and DMA accesses. These buses connect all of the physical spaces (on-chip memory, off-chip memory, and on-chip peripherals) supported by the TMS320C30.

The program counter (PC) is connected to the 24-bit program address bus (PADDR). The instruction register (IR) is connected to the 32-bit program data bus (PDATA). These buses can fetch a single instruction word every machine cycle.

The 24-bit data address buses (DADDR1 and DADDR2) and the 32-bit data data bus (DDATA) support two data memory accesses every machine cycle. The DDATA bus carries data to the CPU over the CPU1 and CPU2 buses. The CPU1 and CPU2 buses can carry two data memory operands to the multiplier, ALU, and register file every machine cycle. Also internal to the CPU are register buses REG1 and REG2 that can carry two data values from the register file to the multiplier and ALU every machine cycle.

The DMA controller is supported with a 24-bit address bus (DMAADDR) and a 32-bit data bus (DMADATA). These buses allow the DMA to perform memory accesses in parallel with the memory accesses occurring from the data and program buses.

3.4 External Bus Operation

The TMS320C30 provides two external interfaces: the primary bus and expansion bus. Both consist of a 32-bit data bus and a set of control signals. The primary bus has a 24-bit address bus, whereas the expansion bus has a 13-bit address bus. Both buses can be used to address external program/data memory or I/O space. The buses also have an external $\overline{RDY}$ signal for wait-state generation. Additional wait states may be inserted under software control. Refer to Section 8 for detailed information on external bus operation.

The TMS320C30 supports four external interrupts ($\overline{INT3}$-$\overline{INT0}$), a number of internal interrupts, and a nonmaskable external $\overline{RESET}$ signal. Two external I/O flags, XF0 and XF1, can be configured as input or output pins under software control. These pins are also used by the interlocked operations of the TMS320C30. The interlocked-operations instruction group supports multiprocessor communication (see Section 7 for examples of the use of interlocked instructions).

3.5 Peripherals

All TMS320C30 peripherals are controlled through memory mapped registers on a dedicated peripheral bus, composed of a 32-bit data bus and a 24-bit address bus. This peripheral bus permits straightforward communication to the peripherals. The TMS320C30 peripherals include two timers and two serial ports. Figure 3-5 shows the peripherals with associated buses and signals. Refer to Section 9 for detailed information on the peripherals.

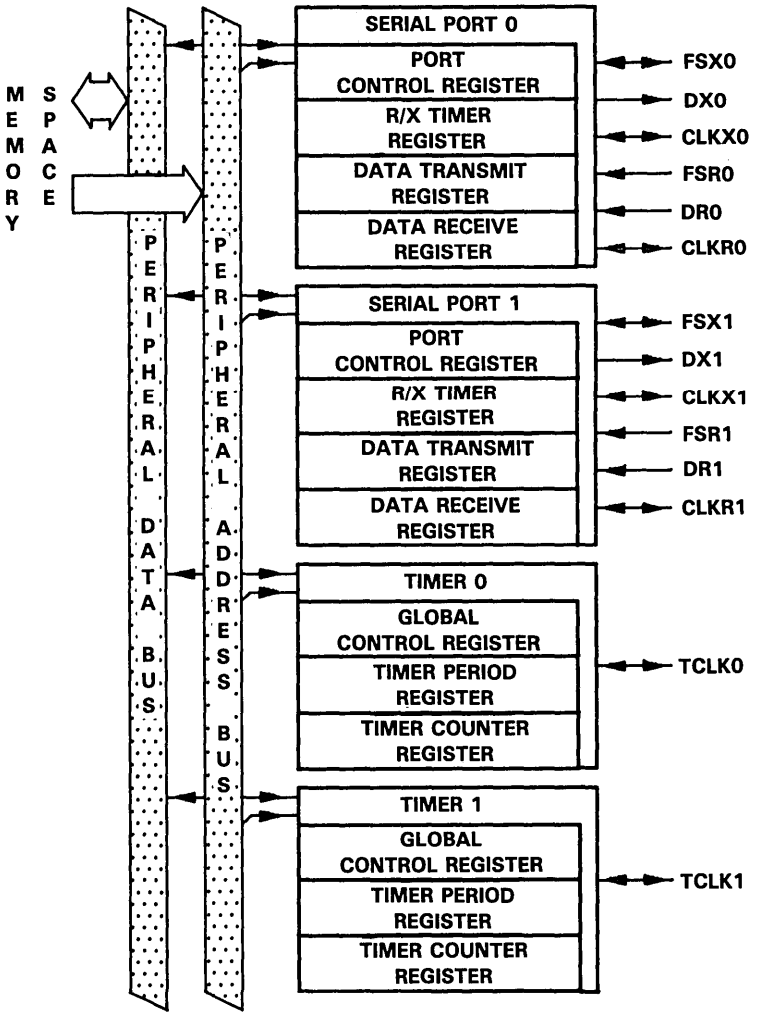


Figure 3-5. Peripheral Modules

3.5.1 Timers

The two timer modules are general-purpose 32-bit timer/event counters, with two signaling modes and internal or external clocking. Each timer has an I/O pin that can be used as an input clock to the timer or as an output signal driven by the timer. The pin may also be configured as a general-purpose I/O pin.

3.5.2 Serial Ports

The two serial ports are totally independent. They are identical with a complementary set of control registers controlling each one. Each serial port can be configured to transfer 8, 16, 24, or 32 bits of data per word. The clock for each serial port can originate either internally or externally. An internally generated divide-down clock is provided. The serial port pins are configurable as general-purpose I/O pins. The serial ports can also be configured as timers. A special handshake mode allows TMS320C30s to communicate over their serial ports with guaranteed synchronization.

3.6 Direct Memory Access (DMA)

The on-chip Direct Memory Access (DMA) controller can read from or write to any location in the memory map without interfering with the operation of the CPU. Therefore, the TMS320C30 can interface to slow external memories and peripherals without reducing throughput to the CPU. The DMA controller contains its own address generators, source and destination registers, and transfer counter. Dedicated DMA address and data buses allow for minimization of conflicts between the CPU and the DMA controller. A DMA operation consists of a block or single-word transfer to or from memory. Refer to Section 9 for detailed information on the DMA. Figure 3-6 shows the DMA controller with associated buses.

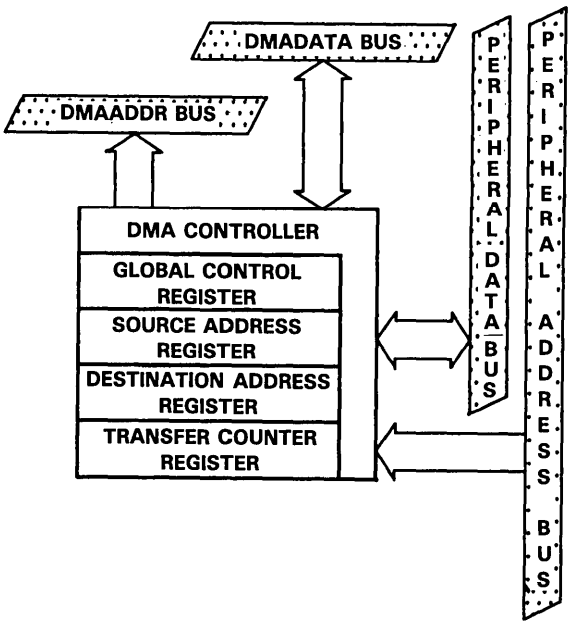


Figure 3-6. DMA Controller

In summary, the TMS320C30 is a powerful DSP system because of its integration of a powerful CPU, large memories, and sufficient buses to support its speed. These along with peripherals such as a DMA controller, two serial

ports, and two timers are all contained on a single chip. The total system real estate and price have been reduced, providing the user with a true single-chip solution.

Introduction 1

Pinout and Signal Descriptions 2

Architectural Overview 3

CPU Registers, Memory, and Cache 4

Data Formats and Floating-Point Operation 5

Addressing 6

Program Flow Control 7

External Bus Operation 8

Peripherals 9

Pipeline Operation 10

Assembly Language Instructions 11

Software Applications 12

Hardware Applications 13

Appendices A-D

Section 4

CPU Registers, Memory, and Cache

The CPU register file contains 28 registers that can be operated upon by the multiplier and ALU (arithmetic logic unit). Included in the register file are the auxiliary registers, extended-precision registers, and index registers. The registers in the CPU register file support addressing, floating-point/integer operations, stack management, processor status, block repeats, and interrupts.

The TMS320C30 provides a total memory space of 16M (million) 32-bit words. Program, data, and I/O space are contained within this 16M-word address space. Two RAM blocks of 1K x 32 bits each and a ROM block of 4K x 32 bits permit two accesses in a single cycle. The memory maps for the microcomputer and microprocessor modes are similar, except that the on-chip ROM is not used in microprocessor mode.

A 64 x 32-bit instruction cache stores often repeated sections of code. This greatly reduces the number of off-chip accesses necessary and allows code to be stored off-chip in slower, lower-cost memories. Three bits are provided in the CPU status register to control the clear, enable, or freeze of the cache.

This section describes in detail each of the CPU registers, the memory maps, and the instruction cache. Major topics in this section are as follows:

- CPU Register File (Section 4.1 on page 4-2)
  - Extended-precision registers (R0-R7)
  - Auxiliary registers (AR0-AR7)
  - Index registers (IR0, IR1)
  - Block size register (BK)
  - Data page pointer (DP)
  - System stack pointer (SP)
  - Status register (ST)
  - CPU/DMA interrupt enable register (IE)
  - CPU interrupt flag register (IF)
  - I/O flags register (IOF)
  - Repeat counter (RC) and block repeat registers (RS, RE)
  - Program counter (PC)
- Memory (Section 4.2 on page 4-11)
  - Memory maps
  - Peripheral bus map
  - Reset/interrupt/trap map
- Instruction Cache (Section 4.3 on page 4-15)
  - Cache architecture
  - Cache algorithm
  - Cache control bits

4.1 CPU Register File

The TMS320C30 provides 28 registers in a multiport register file that is tightly coupled to the CPU. The PC is not included in the 28 registers. All of these registers can be operated upon by the multiplier and ALU, and can be used as general-purpose 32-bit registers. However, the registers also have some special functions for which they are more suited than others. For example, the eight extended-precision registers are especially suited for maintaining extended-precision floating-point results. The eight auxiliary registers support a variety of indirect addressing modes and can be used as general-purpose 32-bit integer and logical registers. The remaining registers provide system functions such as addressing, stack management, processor status, interrupts, and block repeat. Refer to Section 6 for detailed information and examples of the use of CPU registers in addressing.

The registers names and assigned function are listed in Table 4-1.

Table 4-1. CPU Registers

REGISTER NAME	ASSIGNED FUNCTION
R0	Extended-precision register 0
R1	Extended-precision register 1
R2	Extended-precision register 2
R3	Extended-precision register 3
R4	Extended-precision register 4
R5	Extended-precision register 5
R6	Extended-precision register 6
R7	Extended-precision register 7
AR0	Auxiliary register 0
AR1	Auxiliary register 1
AR2	Auxiliary register 2
AR3	Auxiliary register 3
AR4	Auxiliary register 4
AR5	Auxiliary register 5
AR6	Auxiliary register 6
AR7	Auxiliary register 7
DP	Data page pointer
IR0	Index register 0
IR1	Index register 1
BK	Block size
SP	System stack pointer
ST	Status register
IE	CPU/DMA interrupt enable
IF	CPU interrupt flags
IOF	I/O flags
RS	Repeat start address
RE	Repeat end address
RC	Repeat counter
PC	Program counter

4.1.1 Extended-Precision Registers (R0-R7)

The eight extended-precision registers (R0-R7) are capable of storing and supporting operations on 32-bit integer and 40-bit floating-point numbers. These registers consist of two separate and distinct regions. Bits 39-32 of the extended-precision registers are dedicated to the storage of the exponent (*e*) of the floating-point number. Bits 31-0 store the mantissa of the floating-point number. Bit 31 is the sign (*s*) bit, bits 30 - 0 are the fraction (*f*). Any instruction that assumes the operands are floating-point numbers uses bits 39-0. Figure 4-1 illustrates the storage of 40-bit floating-point numbers in the extended-precision registers.

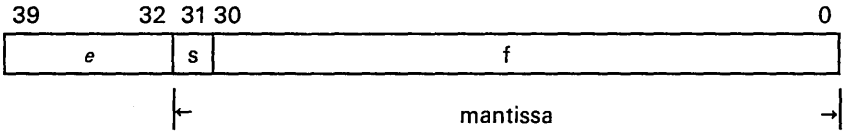


Figure 4-1. Extended-Precision Register Floating-Point Format

For integer operations, bits 31-0 of the extended-precision registers contain the integer (signed or unsigned). Any instruction that assumes the operands are either signed or unsigned integers uses only bits 31-0. Bits 39-32 remain unchanged. This is true for all shift operations. The storage of 32-bit integers in the extended-precision registers is shown in Figure 4-2.

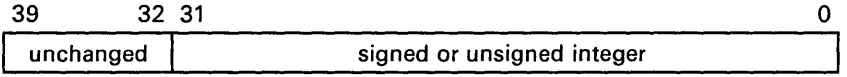


Figure 4-2. Extended-Precision Register Integer Format

4.1.2 Auxiliary Registers (AR0-AR7)

The eight 32-bit auxiliary registers (AR0-AR7) can be accessed by the CPU and modified by the two Auxiliary Register Arithmetic Units (ARAUs). The primary function of the auxiliary registers is the generation of 24-bit addresses. However, they can also be used to perform a variety of functions, such as loop counters in indirect addressing or as 32-bit general-purpose registers that can be modified by the multiplier and ALU. Refer to Section 6 for detailed information and examples of the use of auxiliary registers in addressing.

4.1.3 Data Page Pointer (DP)

The data page pointer (DP) is a 32-bit register. The eight LSBs of the data page pointer are used by the direct addressing mode as a pointer to the page of data being addressed. Data pages are 64 k words long with a total of 256 pages. Bits 31 - 8 are reserved and should always be kept zero by the user.

4.1.4 Index Registers (IR0, IR1)

The 32-bit index registers (IR0 and IR1) are used by the Auxiliary Register Arithmetic Unit (ARAU) for indexing the address. Refer to Section 6 for detailed information and examples of the use of index registers in addressing.

4.1.5 Block Size Register (BK)

The 32-bit block size register (BK) is used by the ARAU in circular addressing to specify the data block size (see Section 6.3).

4.1.6 System Stack Pointer (SP)

The system stack pointer (SP) is a 32-bit register that contains the address of the top of the system stack. The SP always points to the last element pushed onto the stack. The SP is manipulated by interrupts, traps, calls, returns, and the PUSH, PUSHF, POP, and POPF instructions. Pushes and pops of the stack perform pre-increment and post-decrement on all 32 bits of the stack pointer. However, only the 24 LSBs are used as an address. Refer to Section 6.5 for information about system stack management.

4.1.7 Status Register (ST)

The status register (ST) contains global information relating to the state of the CPU. Typically, operations set the condition flags of the status register according to whether the result is zero, negative, etc. This includes register load and store operations as well as arithmetic and logical functions. When the status register is loaded, however,a bit-for-bit replacement is performed of the current contents with the contents of the source operand regardless of the state of any bits in the source operand. Therefore, following a load, the contents of the status register are identically equal to the contents of the source operand. This allows the status register to be easily saved and restored. At system reset, 0 is written to this register.

The format of the status register is shown in Figure 4-3. Table 4-2 defines the status register bits, their names and functions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	GIE	CC	CE	CF	xx	RM	OVM	LUF	LV	UF	N	Z	V	C
		R/W	R/W	R/W	R/W		R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

NOTE: xx = reserved bit.
R = read, W = write.

Figure 4-3. Status Register

Table 4-2. Status Register Bits Summary

BIT	NAME	FUNCTION
0	C	Carry flag
1	V	Overflow flag
2	Z	Zero flag
3	N	Negative flag
4	UF	Floating-point underflow flag
5	LV	Latched overflow flag
6	LUF	Latched floating-point underflow flag
7	OVM	Overflow mode flag. This flag affects only the integer operations. If OVM = 0, the overflow mode is turned off; integer results that overflow are treated in no special way. If OVM = 1, integer results overflowing in the positive direction are set to the most positive 32-bit two's-complement number (7FFFFFFFh). If OVM = 1, integer results overflowing in the negative direction are set to the most negative 32-bit two's-complement number (80000000h). Note that the function of V and LV is independent of the setting of OVM.
8	RM	Repeat mode flag. If RM = 1, the PC is being modified in either the repeat block or repeat-single mode.
9	Reserved	Read as 0.
10	CF	Cache Freeze. When CF = 1, the cache is frozen. If the cache is enabled (CE = 1), fetches from the cache are allowed, but no modification of the state of the cache is performed. This function can be used to save frequently used code resident in the cache. At reset, 0 is written to this bit. Cache clearing (CC=1) is allowed when CF=0.
11	CE	Cache Enable. CE = 1 enables the cache, allowing the cache to be used according to the LRU cache algorithm. CE = 0 disables the cache; no update or modification of the cache can be performed. No fetches are made from the cache. This function is useful for system debug. At system reset, 0 is written to this bit. Cache clearing (CC = 1) is allowed when CE=0.
12	CC	Cache Clear. CC = 1 invalidates all entries in the cache. This bit is always cleared after it is written to and thus always read as 0. At reset, 0 is written to this bit.
13	GIE	Global interrupt enable. If GIE = 1, the CPU responds to an enabled interrupt. If GIE = 0, the CPU does not respond to an enabled interrupt.
14-15	Reserved	Read as 0.
16-31	Reserved	Value undefined.

4.1.8 CPU/DMA Interrupt Enable Register (IE)

The CPU/DMA interrupt enable register (IE) is a 32-bit register (see Figure 4-4). The CPU interrupt enable bits are in locations 10-0. The DMA interrupt enable bits are in locations 26-16. A 1 in a CPU/DMA interrupt enable register bit enables the corresponding interrupt. A 0 disables the corresponding interrupt. At reset, 0 is written to this register. Table 4-3 defines the register bits, the bit names, and the bit functions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
xx	xx	xx	xx	xx	EDINT (DMA)	ETINT1 (DMA)	ETINT0 (DMA)	ERINT1 (DMA)	EXINT1 (DMA)	ERINT0 (DMA)	EXINT0 (DMA)	EINT3 (DMA)	EINT2 (DMA)	EINT1 (DMA)	EINT0 (DMA)
					R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	xx	xx	xx	EDINT (CPU)	ETINT1 (CPU)	ETINT0 (CPU)	ERINT1 (CPU)	EXINT1 (CPU)	ERINT0 (CPU)	EXINT0 (CPU)	EINT3 (CPU)	EINT2 (CPU)	EINT1 (CPU)	EINT0 (CPU)
					R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

NOTE: xx = reserved bit, read as 0.
R = read, W = write.

Figure 4-4. CPU/DMA Interrupt Enable Register (IE)

Table 4-3. IE Register Bits Summary

BIT	NAME	FUNCTION
0	EINT0	Enable external interrupt 0 (CPU)
1	EINT1	Enable external interrupt 1 (CPU)
2	EINT2	Enable external interrupt 2 (CPU)
3	EINT3	Enable external interrupt 3 (CPU)
4	EXINT0	Enable serial port 0 transmit interrupt (CPU)
5	ERINT0	Enable serial port 0 receive interrupt (CPU)
6	EXINT1	Enable serial port 1 transmit interrupt (CPU)
7	ERINT1	Enable serial port 1 receive interrupt (CPU)
8	ETINT0	Enable timer 0 interrupt (CPU)
9	ETINT1	Enable timer 1 interrupt (CPU)
10	EDINT	Enable DMA controller interrupt (CPU)
11-15	Reserved	Value undefined
16	EINT0	Enable external interrupt 0 (DMA)
17	EINT1	Enable external interrupt 1 (DMA)
18	EINT2	Enable external interrupt 2 (DMA)
19	EINT3	Enable external interrupt 3 (DMA)
20	EXINT0	Enable serial port 0 transmit interrupt (DMA)
21	ERINT0	Enable serial port 0 receive interrupt (DMA)
22	EXINT1	Enable serial port 1 transmit interrupt (DMA)
23	ERINT1	Enable serial port 1 receive interrupt (DMA)
24	ETINT0	Enable timer 0 interrupt (DMA)
25	ETINT1	Enable timer 1 interrupt (DMA)
26	EDINT	Enable DMA controller interrupt (DMA)
27-32	Reserved	Value undefined

4.1.9 CPU Interrupt Flag Register (IF)

The 32-bit CPU interrupt flag register (IF) is shown in Figure 4-5. A 1 in a CPU interrupt flag register bit indicates that the corresponding interrupt is set. The IF bits are set to 1 when an interrupt occurs. They may also be set to 1 through software to cause an interrupt. A 0 indicates that the corresponding interrupt is not set. If a 0 is written to an interrupt flag register bit, the corresponding interrupt is cleared. At reset, 0 is written to this register. Table 4-4 lists the bit fields, bit field names, and bit field functions of the CPU interrupt flag register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	xx	xx	xx	DINT	TINT1	TINT0	RINT1	XINT1	RINT0	XINT0	INT3	INT2	INT1	INT0
					R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

NOTE: xx = reserved bit, read as 0.
R = read, W = write.

Figure 4-5. CPU Interrupt Flag Register (IF)

4

Table 4-4. IF Register Bits Summary

BIT	NAME	FUNCTION
0	INT0	External interrupt 0 flag
1	INT1	External interrupt 1 flag
2	INT2	External interrupt 2 flag
3	INT3	External interrupt 3 flag
4	XINT0	Serial port 0 transmit interrupt flag
5	RINT0	Serial port 0 receive interrupt flag
6	XINT1	Serial port 1 transmit interrupt flag
7	RINT1	Serial port 1 receive interrupt flag
8	TINT0	Timer 0 interrupt flag
9	TINT1	Timer 1 interrupt flag
10	DINT0	DMA channel interrupt flag
11-31	Reserved	Value undefined

4.1.10 I/O Flags Register (IOF)

The I/O flags register (IOF) controls the function of the dedicated external pins, XF0 and XF1. These pins may be configured for input or output (see Table 4-5). They may also be read from and written to. At reset, 0 is written to this register. The bit fields, bit field names, and bit field functions are shown in Table 4-5.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	xx	xx	xx	xx	xx	xx	INXF1	OUTXF1	$\overline{I}$/OXF1	xx	INXF0	OUTXF0	$\overline{I}$/OXF0	xx
								R	R/W	R/W		R	R/W	R/W	

NOTE: xx = reserved bit, read as 0.
R = read, W = write.

Figure 4-6. I/O Flag Register (IOF)

Table 4-5. IOF Register Bits Summary

BIT	NAME	FUNCTION
0	Reserved	Read as 0.
1	$\overline{I}$/OXF0	If $\overline{I}$/OXF0 = 0, XF0 is configured as a general-purpose input pin. If $\overline{I}$/OXF0 = 1, XF0 is configured as a general-purpose output pin.
2	OUTXF0	Data output on XF0.
3	INXF0	Data input on XF0. A write has no effect.
4	Reserved	Read as 0.
5	$\overline{I}$/OXF1	If $\overline{I}$/OXF1 = 0, XF1 is configured as a general-purpose input pin. If $\overline{I}$/OXF1 = 1, XF1 is configured as a general-purpose output pin.
6	OUTXF1	Data output on XF1.
7	INXF1	Data input on XF1. A write has no effect.
8-31	Reserved	Read as 0.

4.1.11 Repeat Counter (RC) and Block Repeat Registers (RS, RE)

The repeat counter (RC) is a 32-bit register used to specify the number of times a block of code is to be repeated when performing a block repeat.

The repeat start address register (RS) is a 32-bit register containing the starting address of the block of program memory to be repeated when operating in the repeat mode.

The 32-bit repeat end address register (RE) contains the ending address of the block of program memory to be repeated when operating in the repeat mode.

4.1.12 Program Counter (PC)

The program counter (PC) is a 32-bit register containing the address of the next instruction to be fetched. While the program counter is not part of the CPU register file, it is a register that can be modified via instructions that modify the program flow.

4.1.13 Reserved Bits and Compatibility

In order to retain compatibility with future members of the TMS320C3X family of microprocessors, reserved bits that are read as zero must be written as zero. Reserved bits that have an undefined value must not have their current value modified. In other cases, the user should maintain the reserved bits as specified.

4.2 Memory

The total memory space of the TMS320C30 is 16M (million) 32-bit words. Program, data, and I/O space are contained within this, allowing tables, coefficients, program code, or data to be stored in either RAM or ROM. In this way, memory usage can be maximized and memory space allocated as desired.

RAM blocks 0 and 1 are each 1K x 32 bits. The ROM block is 4K x 32 bits. Each RAM and ROM block is capable of supporting two accesses in a single cycle. The separate program buses, data buses, and DMA buses allow for parallel program fetches, data reads/writes, and DMA operations. This is covered in detail in Section 10.3.

4.2.1 Memory Maps

The memory map is dependent upon whether the processor is running in the microprocessor mode (MC/$\overline{MP}$ = 0) or the microcomputer mode (MC/$\overline{MP}$ = 1). The memory maps for these modes are very similar (see Figure 4-7). Locations 800000h through 801FFFh are mapped to the expansion bus. When this region is accessed, $\overline{MSTRB}$ is active. Locations 802000h through 803FFFh are reserved. Locations 804000h through 805FFFh are mapped to the expansion bus. When this region is accessed, $\overline{IOSTRB}$ is active. Locations 806000h through 807FFFh are reserved. All of the memory-mapped peripheral registers are in locations 808000h through 8097FFh. In both modes, RAM block 0 is located at addresses 809800h through 809BFFh, and RAM block 1 is located at addresses 809C00h through 809FFFh. Memory locations 80A000h through 0FFFFFFh are accessed over the external memory port ($\overline{STRB}$ active).

In microprocessor mode, the 4K on-chip ROM is not mapped into the TMS320C30 memory map. Locations 0h through 3Fh consist of interrupt vector, trap vector, and reserved locations, all of which are accessed over the external memory port ($\overline{STRB}$ active). Locations 40h through 7FFFFFh are also accessed over the external memory port.

In microcomputer mode, the 4K on-chip ROM is mapped into locations 0h through 0FFFh. There are 192 locations (0h through BFh) within this block for interrupt vectors, trap vectors, and a reserved space. Locations 1000h through 7FFFFFh are accessed over the external memory port ($\overline{STRB}$ active).

Reserved portions of the TMS320C30 memory space and reserved peripheral bus addresses should not be read and written by the user. Doing so may cause the TMS320C30 to halt operation and require a system reset to restart.

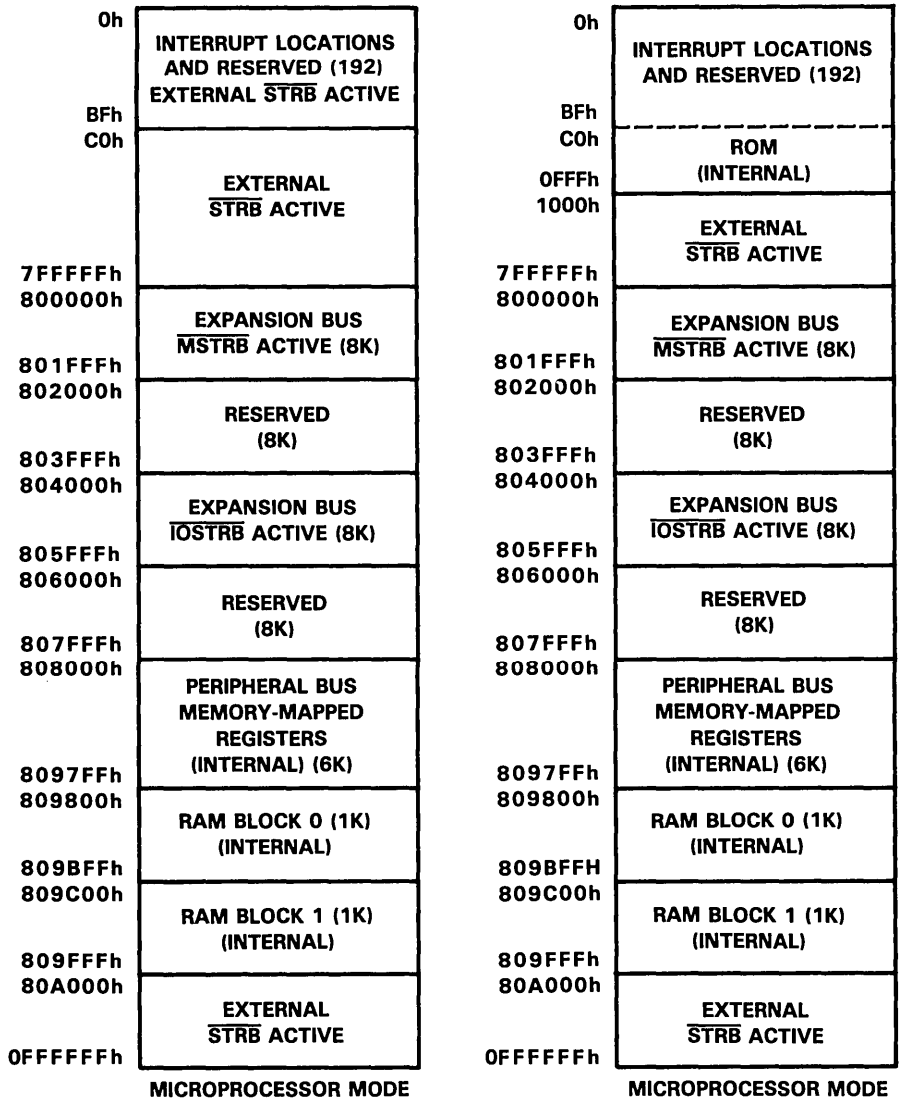


Figure 4-7. Memory Maps

4.2.2 Peripheral Bus Map

The memory-mapped peripheral registers are located starting at address 808000h. The peripheral bus memory map is shown in Figure 4-8. Each peripheral occupies a 16-word region of the memory map. Locations 808010h through 80801Fh and locations 808070h through 8097FFh are reserved.

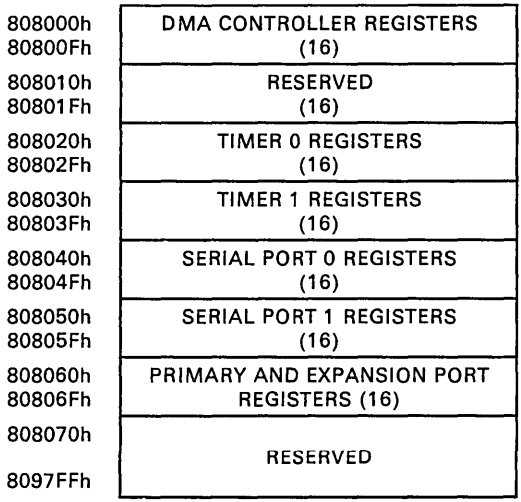


Figure 4-8. Peripheral Bus Memory Map

4.2.3 Reset/Interrupt/Trap Vector Map

The addresses for the reset, interrupt, and trap vectors are 0h through 3Fh, as shown in Figure 4-9. The vectors stored in these locations are the addresses of the start of the respective reset, interrupt, and trap routines. For example, at reset, the contents of memory location 0h (the reset vector) are loaded into the PC and execution begins from that address.

Traps 28-31 are reserved and should not be used by the user.

00h	RESET
01h	INT0
02h	INT1
03h	INT2
04h	INT3
05h	XINT0
06h	RINT0
07h	XINT0
08h	RINT1
09h	TINT0
0Ah	TINT1
0Bh	DINT
0Ch 1Fh	RESERVED
20h	TRAP 0
	. . .
3Bh	TRAP 27
3Ch	TRAP 28 (Reserved)
3Dh	TRAP 29 (Reserved)
3Eh	TRAP 30 (Reserved)
3Fh	TRAP 31 (Reserved)

Figure 4-9. Reset, Interrupt, and Trap Vector Locations

4.3 Instruction Cache

A 64 x 32-bit instruction cache allows for maximum system performance with minimal system cost. The instruction cache stores sections of code that can be fetched when repeatedly accessing time-critical code. This greatly reduces the number of off-chip accesses necessary and allows for code to be stored off-chip in slower, lower-cost memories. The external buses are also freed from program fetches, so they can be used by the DMA or other system elements.

The cache can operate in a completely automatic fashion without the need for user intervention. A form of the LRU (least-recently-used) cache update algorithm is used (see Section 4.3.2).

4.3.1 Cache Architecture

The instruction cache (see Figure 4-10) contains 64 32-bit words of RAM. The cache is divided into two 32-word segments. Associated with each segment is a 19-bit segment start address (SSA) register. For each word in the cache, there is a corresponding single-bit: Present (P) flag.

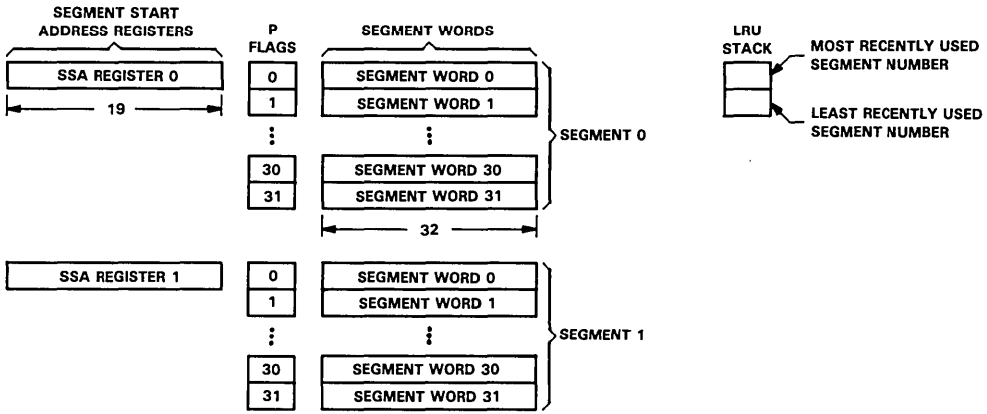


Figure 4-10. Instruction Cache Architecture

When the CPU requests an instruction word from external memory, a check is made to determine if the word is already contained in the instruction cache. The partitioning of an instruction address as used by the cache control algorithm is shown in Figure 4-11. The 19 most-significant bits of the instruction address are used to select the segment and the 5 least-significant bits define the address of the instruction word within the pertinent segment. The 19 MSBs of the instruction address are compared with the two segment start

address (SSA) registers. If a match is found, a check is made of the relevant P flag. The P flag indicates whether or not the word within a particular segment is already present in cache memory.

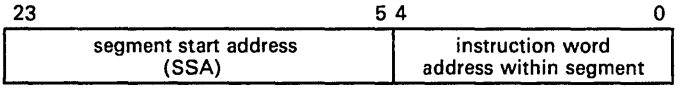


Figure 4-11. Address Partitioning for Cache Control Algorithm

If there is no match, one of the segments must be replaced by the new data. The segment replaced in this circumstance is determined by the LRU (least-recently-used) algorithm. The LRU stack (see Figure 4-10) is maintained for this purpose.

The LRU stack tracks which of the two segments qualifies as the least-recently-used after each access to the cache, therefore the stack contains either 0,1 or 1,0. Each time a segment is accessed, its segment number is removed from the LRU stack and pushed on the top of the LRU stack. Therefore, the number at the top of the stack is the most-recently-used segment number and the number at the bottom of the stack is the least-recently-used segment number.

At system reset, the LRU stack is initialized with 0 at the top, 1 at the bottom, and all P flags in the instruction cache are cleared. If both SSA registers are equal (due to system reset conditions) and a cache hit occurs, the instruction word is fetched from the most recently used segment.

When a replacement is necessary, the least-recently-used segment is selected for replacement. Also, the 32 P flags for the segment to be replaced are set to 0, and the segment's SSA register is replaced with the 19 MSBs of the instruction address. ,*

4.3.2 Cache Algorithm

When the TMS320C30 requests an instruction word from external memory, two possible actions occur: a cache hit or a cache miss. These are described in the following list:

- **Cache Hit.** The requested instruction is contained within the cache and the following actions occur:
  1) The instruction word is read from the cache.
  2) The segment number of the segment within which the word is contained is removed from the LRU stack and pushed to the top of the LRU stack, thus moving the other segment number to the bottom of the stack.

- **Cache Miss.** The instruction is not contained in the cache. Types of cache miss are:
  1) Word Miss. The segment address register matches the instruction address, but the relevant P flag is not set. The following actions occur in parallel:

- The instruction word is read from memory and copied into the cache.
- The segment number of the segment within which the word is contained is removed from the LRU stack and pushed to the top of the LRU stack, thus moving the other segment number to the bottom of the stack.
- The relevant P flag is set.

2) Segment Miss. Neither of the segment addresses matches the instruction address. The following actions occur in parallel:
   - The least-recently-used segment is selected for replacement. The P flags for all 32 words are cleared.
   - The SSA register for the selected segment is loaded with the 19 MSBs of the address of the requested instruction word.
   - The instruction word is fetched and copied into the cache. It goes into the appropriate word of the least-recently-used segment. The P flag for that word is set 1.
   - The segment number of the segment containing the instruction word is removed from the LRU stack and pushed to the top of the LRU stack, thus moving the other segment number to the bottom of the stack.

Only instructions may be fetched from the program cache. All reads and writes of data in memory bypass the cache. Program fetches from internal memory do not modify the cache and will not generate cache hits or misses. The program cache is a single-access memory block. Dummy program fetches (i.e., following a branch) are treated by the cache as valid program fetches and can generate cache misses and cache updates.

Care should be taken when using self-modifying code. If an instruction resides in cache and the corresponding location in primary memory is modified, the copy of the instruction in cache is not modified.

More efficient use of the cache can be made by aligning program code on 32 word address boundaries. This can be done using the ALIGN directive when coding assembly language.

4.3.3 Cache Control Bits

Three cache control bits are located in the CPU status register: the cache clear bit (CC), cache enable bit (CE), and the cache freeze bit (CF).

Cache Clear Bit (CC). Writing a 1 to the cache clear bit (CC) invalidates all entries in the cache. All P flags in the cache are cleared. The CC bit is always cleared after the cache is cleared. It is therefore always read as a 0. At reset the cache is cleared and 0 is written to this bit.

Cache Enable Bit (CE). Writing a 1 to this bit enables the cache. When enabled, the cache is used according to the previously described cache algorithm. Writing a 0 to the cache enable bit disables the cache; no updates or modification of the cache can be performed. Specifically, no SSA register updates are performed, no P flags are modified (unless CC = 1), and the LRU stack is not modified. Writing a 1 to CC when the cache is disabled will clear the cache, and thus the P flags. No fetches are made from the cache when the cache is disabled. At reset, 0 is written to this bit.

Cache Freeze Bit (CF). When CF = 1, the cache is frozen. If, in addition, the cache is enabled, fetches from the cache are allowed, but no modification of the state of the cache is performed. Specifically, no SSA register updates are performed, no P flags are modified (unless CC = 1), and the LRU stack is not modified. This function can be used to keep frequently used code resident in the cache. Writing a 1 to CC when the cache is frozen will clear the cache, and thus the P flags. At reset, 0 is written to this bit.

Table 4-6 defines the effect of the CE and CF bits used in combination.

Table 4-6. Combined Effect of the CE and CF Bits

CE	CF	EFFECT
0	0	Cache not enabled
0	1	Cache not enabled
1	0	Cache enabled and not frozen
1	1	Cache enabled and frozen

Introduction 1

Pinout and Signal Descriptions 2

Architectural Overview 3

CPU Registers, Memory, and Cache 4

Data Formats and Floating-Point Operation 5

Addressing 6

Program Flow Control 7

External Bus Operation 8

Peripherals 9

Pipeline Operation 10

Assembly Language Instructions 11

Software Applications 12

Hardware Applications 13

Appendices A-D

Section 5

Data Formats and Floating-Point Operation

Data is organized in the TMS320C30 architecture to provide three fundamental data types: integer, unsigned-integer, and floating-point. Note that the terms, integer and signed-integer, are considered to be equivalent. The TMS320C30 supports short and single-precision formats for signed and unsigned integers. It also supports short, single-precision and extended-precision formats for floating-point data.

Floating-point operations provide convenient and trouble-free computations while maintaining accuracy and precision. The TMS320C30 implementation of floating-point arithmetic allows for floating-point operations at integer speeds. The floating-point capability can prevent problems with overflow, operand alignment, and other burdensome tasks common in integer operations.

This section discusses in detail the data formats and floating-point operations supported on the TMS320C30. Major topics in this section are as follows:

- Integer Formats (Section 5.1 on page 5-2)
- Unsigned-Integer Formats (Section 5.2 on page 5-3)
- Floating-Point Formats (Section 5.3 on page 5-4)
- Floating-Point Multiplication (Section 5.4 on page 5-9)
- Floating-Point Addition and Subtraction (Section 5.5 on page 5-13)
- Normalization (Section 5.6 on page 5-17)
- Rounding (Section 5.7 on page 5-20)
- Floating-Point to Integer Conversions (Section 5.8 on page 5-22)
- Integer to Floating-Point Conversions (Section 5.9 on page 5-24)

5.1 Integer Formats

The TMS320C30 supports two integer formats: a 16-bit short integer format and a 32-bit single-precision integer format. When extended-precision registers are used as integer operands only bits 31-0 are used; bits 39-32 remain unchanged and unused.

5.1.1 Short Integer Format

The short integer format is a 16-bit two's-complement integer format, used for immediate integer operands. For those instructions that assume integer operands, this format is sign-extended to 32 bits (see Figure 5-1). The range of an integer *si*, represented in the short integer format, is $-2^{15} \leq si \leq 2^{15} - 1$. In Figure 5-1, *s*=signed bit.

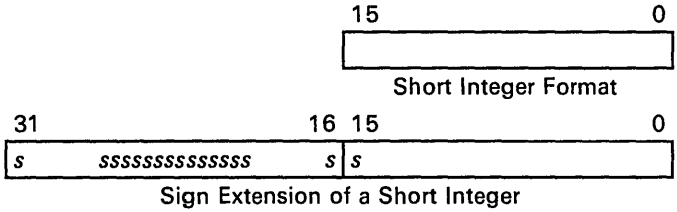


Figure 5-1. Short Integer Format and Sign Extension of Short Integer

5.1.2 Single-Precision Integer Format

In the single-precision integer format, the integer is represented in two's-complement notation. The range of an integer *sp*, represented in the single-precision integer format, is $-2^{31} \leq sp \leq 2^{31} - 1$. Figure 5-2 shows the single-precision integer format.

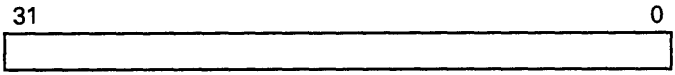


Figure 5-2. Single-Precision Integer Format

5.2 Unsigned-Integer Formats

Two unsigned-integer formats are supported on the TMS320C30: a 16-bit short format and a 32-bit single-precision format. In extended-precision registers, the unsigned-integer operands use only bits 31-0; bits 39-32 remain unchanged.

5.2.1 Short Unsigned-Integer Format

Figure 5-3 shows the 16-bit short unsigned-integer format, used for immediate unsigned-integer operands. For those instructions that assume unsigned-integer operands, this format is zero-filled to 32 bits. In Figure 5-3 below, X = MSB (1 or 0).

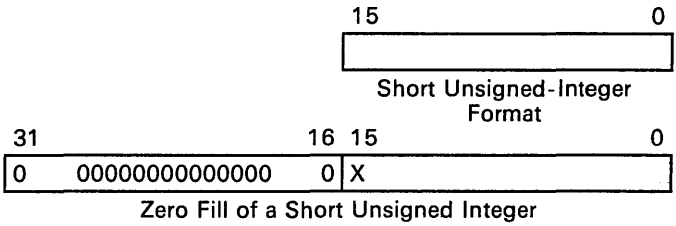


Figure 5-3. Short Unsigned-Integer Format and Zero Fill

5.2.2 Single-Precision Unsigned-Integer Format

In the single-precision unsigned-integer format, the number is represented as a 32-bit value, as shown in Figure 5-4.

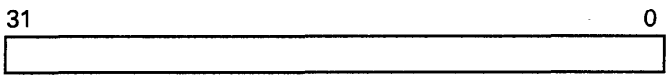


Figure 5-4. Single-Precision Unsigned-Integer Format

5.3 Floating-Point Formats

All TMS320C30 floating-point formats consist of three fields: an exponent field (*e*), a single sign-bit field (*s*), and a fraction field (*f*). These are stored as shown in Figure 5-5. The exponent field is a two's-complement number. The sign field and fraction field may be considered as one unit and referred to as the mantissa field (*man*). The mantissa is used to represent a normalized two's-complement number. In a normalized representation, a most-significant nonsign bit is implied, thus providing an additional bit of precision. The value of a floating-point number *x* as a function of the fields *e*, *s*, and *f* is given as

$$x = \begin{cases} 01.f \times 2^e & \text{if } s = 0 \\ 10.f \times 2^e & \text{if } s = 1 \\ 0 & \text{if } e = \text{most negative two's-complement value for the specified exponent field width.} \end{cases}$$

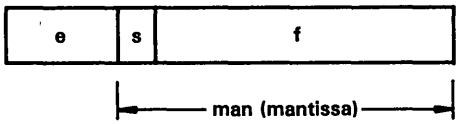


Figure 5-5. Generic Floating-Point Format

Three floating-point formats are supported on the TMS320C30. The first is a short floating-point format for immediate floating-point operands, consisting of a 4-bit exponent, 1 sign bit, and an 11-bit fraction. The second is a single-precision format consisting of an 8-bit exponent, 1 sign bit, and a 23-bit fraction. The third is an extended-precision format consisting of an 8-bit exponent, 1 sign bit, and a 31-bit fraction.

5.3.1 Short Floating-Point Format

In the short floating-point format, floating-point numbers are represented by a two's-complement 4-bit exponent field (*e*) and a two's-complement 12-bit mantissa field (*man*) with an implied most-significant nonsign bit.

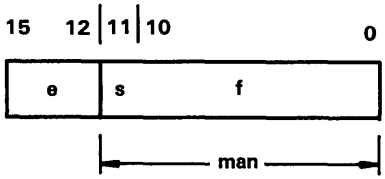


Figure 5-6. Short Floating-Point Format

Operations are performed with an implied binary point between bits 11 and 10. When the implied most-significant nonsign bit is made explicit, it is located to the immediate left of the binary point. The floating-point two's-complement number x in the short floating-point format is given by

$$x = \begin{cases} 01.f \times 2^e & \text{if } s = 0 \\ 10.f \times 2^e & \text{if } s = 1 \\ 0 & \text{if } e = -8,\ s = 0,\ f = 0 \end{cases}$$

The following reserved values must be used to represent zero in the short floating-point format:

$$e = -8$$
$$s = 0$$
$$f = 0$$

The following examples illustrate the range and precision of the short floating-point format:

Most Positive: $x = (2 - 2^{-11}) \times 2^{7} = 2.5594 \times 10^{2}$

Least Positive: $x = 1 \times 2^{-7} = 7.8125 \times 10^{-3}$

Least Negative: $x = (-1-2^{-11}) \times 2^{-7} = -7.8163 \times 10^{-3}$

Most Negative: $x = -2 \times 2^{7} = -2.5600 \times 10^{2}$

5.3.2 Single-Precision Floating-Point Format

In the single-precision format, the floating-point number is represented by an 8-bit exponent field (e) and a two's-complement 24-bit mantissa field (*man*) with an implied most-significant nonsign bit.

Operations are performed with an implied binary point between bits 23 and 22. When the implied most-significant nonsign bit is made explicit, it is located to the immediate left of the binary point. The floating-point number x is given by:

$$x = \begin{cases} 01.f \times 2^e & \text{if } s = 0 \\ 10.f \times 2^e & \text{if } s = 1 \\ 0 & \text{if } e = -128,\ s = 0,\ f = 0 \end{cases}$$

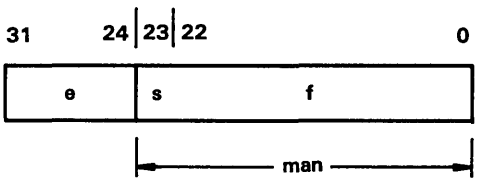


Figure 5-7. Single-Precision Floating-Point Format

The following reserved values must be used to represent zero in the single-precision floating-point format:

$$e = -128$$
$$s = 0$$
$$f = 0$$

The following examples illustrate the range and precision of the single-precision floating-point format.

Most Positive: $x = (2 - 2^{-23}) \times 2^{127} = 3.4028234 \times 10^{38}$

Least Positive: $x = 1 \times 2^{-127} = 5.8774717 \times 10^{-39}$

Least Negative: $x = (-1-2^{-23}) \times 2^{-127} = -5.8774724 \times 10^{-39}$

Most Negative: $x = -2 \times 2^{127} = -3.4028236 \times 10^{38}$

5.3.3 Extended-Precision Floating-Point Format

In the extended-precision format, the floating-point number is represented by an 8-bit exponent field (*e*) and a 32-bit mantissa field (*man*) with an implied most-significant nonsign bit.

5

Operations are performed with an implied binary point between bits 31 and 30. When the implied most-significant nonsign bit is made explicit, it is located to the immediate left of the binary point. The floating-point number *x* is given by

$$x = \begin{cases} 01.f \times 2^e & \text{if } s = 0 \\ 10.f \times 2^e & \text{if } s = 1 \\ 0 & \text{if } e = -128,\ s = 0,\ f = 0 \end{cases}$$

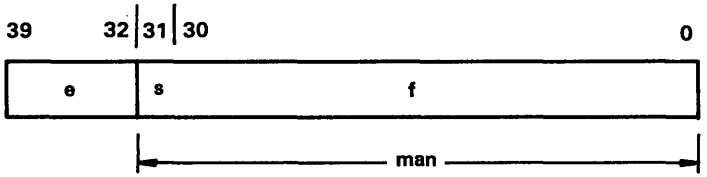


Figure 5-8. Extended-Precision Floating-Point Format

The following reserved values must be used to represent zero in the extended-precision floating-point format:

$e = -128$
$s = 0$
$f = 0$

The following examples illustrate the range and precision of the extended-precision floating-point format:

Most Positive: $x = (2 - 2^{-31}) \times 2^{127} = 3.4028236683 \times 10^{38}$

Least Positive: $x = 1 \times 2^{-127} = 5.8774717541 \times 10^{-39}$

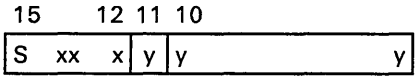
Least Negative: $x = (-1-2^{-31}) \times 2^{-127} = -5.8774717569 \times 10^{-39}$

Most Negative: $x = -2 \times 2^{127} = -3.4028236691 \times 10^{38}$

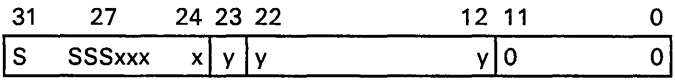
5.3.4 Conversion Between Floating-Point Formats

Floating-point operations assume several different formats for inputs and outputs. These formats often require conversion from one floating-point format to another (e.g., short floating-point format to extended-precision floating-point format). Format conversions automatically occur in hardware, with no overhead, as a part of the floating-point operations. The four conversions are shown below with examples of the conversion. When a floating-point format zero is converted to a greater-precision format, it is always converted to a valid representation of zero in that format. In the below figures, S = sign bit of the exponent.

- **Short floating-point format conversion to single-precision floating-point format.**



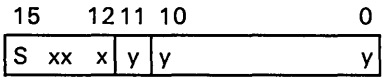
Short Floating-Point Format



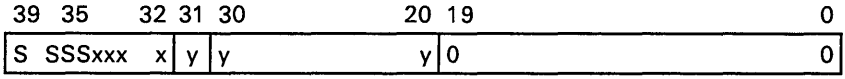
Single-Precision Floating-Point Format

In this format, the exponent field is sign-extended and the fraction field filled with zeros.

- **Short floating-point format conversion to extended-precision floating-point format.**



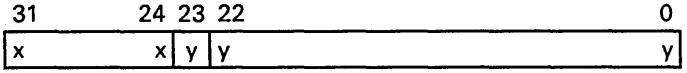
Short Floating-Point Format



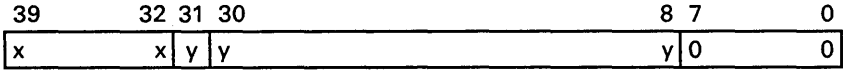
Extended-Precision Floating-Point Format

The exponent field in this format is sign-extended and the fraction field filled with zeros.

- **Single-precision floating-point format conversion to extended-precision floating-point format.**



Single-Precision Floating-Point Format

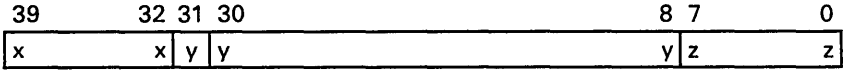


Extended-Precision Floating-Point Format

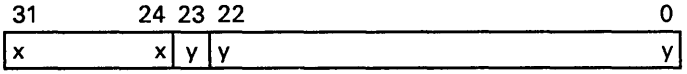
The fraction field is filled with zeros.

5

- **Extended-precision floating-point format conversion to single-precision floating-point format.**



Extended-Precision Floating-Point Format



Single-Precision Floating-Point Format

The fraction field is truncated.

5.4 Floating-Point Multiplication

A floating-point number *a* can be written in floating-point format as the following formula, where a(*man*) is the mantissa and a(*exp*) is the exponent.

$$a = a(man) \times 2^{a(exp)}$$

The product of a and b is c, defined as

$$c = a \times b = a(man) \times b(man) \times 2^{(a(exp)+b(exp))}$$

$$c(man) = a(man) \times b(man)$$
$$c(exp) = a(exp) + b(exp)$$

When performing floating-point multiplication, source operands are always assumed to be in the single-precision floating-point format. If the source of the operands is in short floating-point format, it is extended to the single-precision floating-point format. If the source of the operands is in extended-precision floating-point format, it is truncated to single-precision format. These conversions automatically occur in hardware with no overhead. All results of floating-point multiplications are in the extended-precision format. These multiplications occur in a single cycle.

A flowchart for floating-point multiplication is shown in Figure 5-9. In step 1, the 24-bit source operand mantissas are multiplied, producing a 50-bit result c(*man*). (Note that input and output data are always represented as normalized numbers.) In step 2, the exponents are added, yielding c(*exp*). Steps 3 through 6 check for special cases. Step 3 checks for whether c(*man*) in extended-precision format is equal to zero. If c(*man*) is zero, step 7 sets c(*exp*) to -128, thus yielding the representation for zero.

Steps 4 and 5 normalize the result. If a right shift of one is necessary, then in step 8, c(*man*) is right-shifted one bit and one is added to c(*exp*). If a right shift of two is necessary, then in step 9, c(*man*) is right-shifted two bits and two is added to c(*exp*). Step 6 occurs when the result is normalized.

In step 10, c(*man*) is set in the extended-precision floating-point format. Steps 11 through 18 check for special cases of c(*exp*). In step 14, if c(*exp*) has overflowed (step 11) in the positive direction, then c(*exp*) is set to the most-positive extended-precision format value. If c(*exp*) has overflowed in the negative direction, then c(*exp*) is set to the most-negative extended-precision format value. If c(*exp*) has underflowed (step 12), then c is set to zero (step 15); i.e., c(*man*) = 0 and c(*exp*) = -128.

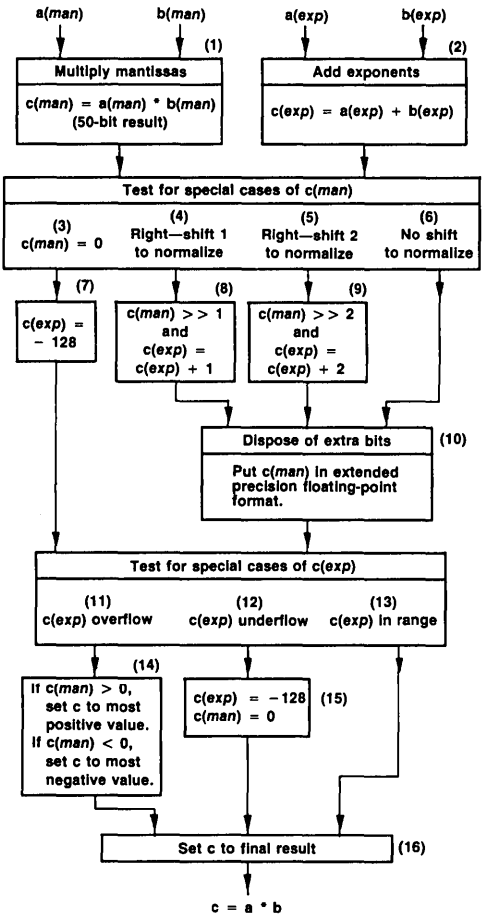


Figure 5-9. Flowchart for Floating-Point Multiplication

The following examples illustrate how floating-point multiplication is performed on the TMS320C30. For these examples, the implied most-significant nonsign bit is made explicit.

Example 5-1. Floating-Point Multiply (Both Mantissas = -2.0)

Let

$$a = -2.0 \times 2^{a(exp)} = 10.00000000000000000000000 \times 2^{a(exp)}$$
$$b = -2.0 \times 2^{b(exp)} = 10.00000000000000000000000 \times 2^{b(exp)}$$

where a and b are both represented in binary form according to the normalized single-precision floating-point format. Then

$$\begin{array}{l} 10.00000000000000000000000 \times 2^{a(exp)} \\ \times\ 10.00000000000000000000000 \times 2^{b(exp)} \\ \hline 0100.00000000000000000000000000000000000000000000000 \times 2^{(a(\ exp) + b(exp))} \end{array}$$

To place this number in the proper normalized format, it is necessary to shift the mantissa two places to the right and add two to the exponent. This yields

$$\begin{array}{l} 10.00000000000000000000000 \times 2^{a(exp)} \\ \times\ 10.00000000000000000000000 \times 2^{b(exp)} \\ \hline 01.00000000000000000000000000000000000000000000000 \times 2^{(a(\ exp) + b(exp)+2)} \end{array}$$

In floating-point multiplication, the exponent of the result may overflow. This can occur when the exponents are initially added or when the exponent is modified during normalization.

Example 5-2. Floating-Point Multiply (Both Mantissas = 1.5)

Let

$$a = 1.5 \times 2^{a(exp)} = 01.10000000000000000000000 \times 2^{a(exp)}$$
$$b = 1.5 \times 2^{b(exp)} = 01.10000000000000000000000 \times 2^{b(exp)}$$

where a and b are both represented in binary form according to the single-precision floating-point format. Then

$$\begin{array}{l} 01.10000000000000000000000 \times 2^{a(exp)} \\ \times\ 01.10000000000000000000000 \times 2^{b(exp)} \\ \hline 0010.01000000000000000000000000000000000000000000000 \times 2^{(a(exp) + b(exp))} \end{array}$$

To place this number in the proper normalized format, it is necessary to shift the mantissa one place to the right and add one to the exponent. This yields

$$\begin{array}{l} 01.10000000000000000000000 \times 2^{a(exp)} \\ \times\ 01.10000000000000000000000 \times 2^{b(exp)} \\ \hline 01.00100000000000000000000000000000000000000000000 \times 2^{(a(exp) + b(exp) + 1)} \end{array}$$

Example 5-3. Floating-Point Multiply (Both Mantissas = 1.0)

Let

$$a = 1.0 \times 2^{a(exp)} = 01.00000000000000000000000 \times 2^{a(exp)}$$
$$b = 1.0 \times 2^{b(exp)} = 01.00000000000000000000000 \times 2^{b(exp)}$$

where a and b are both represented in binary form according to the single-precision floating-point format. Then

$$\begin{array}{l} \quad 01.00000000000000000000000 \times 2^{a(exp)} \\ \times \quad 01.00000000000000000000000 \times 2^{b(exp)} \\ \hline 0001.00000000000000000000000000000000000000000000000 \times 2^{(a(exp) + b(exp))} \end{array}$$

This number is in the proper normalized format. Therefore, no shift of the mantissa or modification of the exponent is necessary.

5

These examples have shown cases where the product of two normalized numbers can be normalized with a shift of zero, one, or two. For all normalized inputs with the floating-point format used by the TMS320C30, a normalized result can be produced by a shift of zero, one, or two.

Example 5-4. Floating-Point Multiply Between Positive and Negative Numbers

Let

$$a = 1.0 \times 2^{a(exp)} = 01.00000000000000000000000 \times 2^{a(exp)}$$
$$b = 2.0 \times 2^{b(exp)} = 10.00000000000000000000000 \times 2^{b(exp)}$$

Then

$$\begin{array}{l} \quad 01.00000000000000000000000 \times 2^{a(exp)} \\ \times \quad 10.00000000000000000000000 \times 2^{b(exp)} \\ \hline 1110.00000000000000000000000000000000000000000000000 \times 2^{(a(exp) + b(exp))} \end{array}$$

The result is $c = -2.0 \times 2^{(a(exp) + b(exp))}$

Example 5-5. Floating-Point Multiply by Zero

All multiplications by a floating-point zero yield a result of zero ($f = 0$, $s = 0$, and $exp = -128$).

5.5 Floating-Point Addition and Subtraction

In floating-point addition and subtraction, two floating-point numbers a and b can be defined as

$$a = a(man) \times 2^{a(exp)}$$
$$b = b(man) \times 2^{b(exp)}$$

The sum (or difference) of a and b can be defined as

$$c = a \pm b$$
$$= (a(man) \pm (b(man) \times 2^{-(a(exp)-b(exp))})) \times 2^{a(exp)}, \quad \text{if } a(exp) \geq b(exp)$$
$$= ((a(man) \times 2^{-(b(exp)-a(exp))}) \pm b(man)) \times 2^{b(exp)}, \quad \text{if } a(exp) < b(exp)$$

The flowchart for floating-point addition is shown in Figure 5-10. Since this flowchart assumes signed data, it is also appropriate for floating-point subtraction. In this figure, it is assumed that $a(exp) \leq b(exp)$. In step 1, the source exponents are compared, and c(*exp*) is set equal to the largest of the two source exponents. In step 2, d is set to the difference of the two exponents. In step 3, the mantissa with the smallest exponent, in this case a(*man*), is right-shifted d bits in order to align the mantissas. After the mantissas have been aligned, they are added (step 4).

Steps 5 through 7 check for a special case of c(*man*). If c(*man*) is zero (step 5), then c(*exp*) is set to its most-negative value (step 8) to yield the correct representation of zero. If c(*man*) has overflowed c (step 6), then in step 9, c(*man*) is right-shifted one bit and one is added to c(*exp*) In step 10, the result is normalized. In steps 11 and 12, special cases of c(*exp*) are tested. If c(*exp*) has overflowed, then c is set to the most-positive extended-precision value if it is positive; otherwise, it is set to the most-negative extended-precision value.

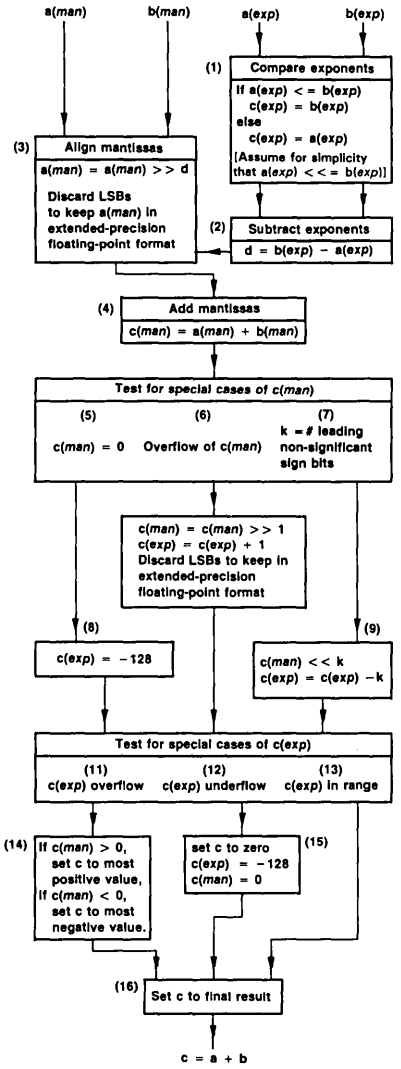


Figure 5-10. Flowchart for Floating-Point Addition

The following examples describe the floating-point addition and subtraction operations. It is assumed that the data is in the extended-precision floating-point format.

Example 5-6. Floating-Point Addition

In the case of two normalized numbers to be summed, let

$$a = 1.5 = 01.10000000000000000000000000000000 \times 2^{0}$$
$$b = 0.5 = 01.00000000000000000000000000000000 \times 2^{-1}$$

It is necessary to shift b to the right by one so that a and b have the same exponent. This yields

$$b = 0.5 = 00.10000000000000000000000000000000 \times 2^{0}$$

Then

$$\begin{array}{rl} & 01.10000000000000000000000000000000 \times 2^{0} \\ + & 00.10000000000000000000000000000000 \times 2^{0} \\ \hline & 010.00000000000000000000000000000000 \times 2^{0} \end{array}$$

As in the case of multiplication, it is necessary to shift the binary point one place to the left and to add one to the exponent. This yields

$$\begin{array}{rl} & 01.10000000000000000000000000000000 \times 2^{0} \\ + & 00.10000000000000000000000000000000 \times 2^{0} \\ \hline & 01.00000000000000000000000000000000 \times 2^{1} \end{array}$$

Example 5-7. Floating-Point Subtraction

A subtraction is performed in this example. Let

$$a = 01.00000000000000000000000000000001 \times 2^{0}$$
$$b = 01.00000000000000000000000000000000 \times 2^{0}$$

The operation to be performed is a - b. The mantissas are already aligned since the two numbers have the same exponent. The result is a large cancellation of the upper bits, as shown below.

$$\begin{array}{rl} & 01.00000000000000000000000000000001 \times 2^{0} \\ - & 01.00000000000000000000000000000000 \times 2^{0} \\ \hline & 00.00000000000000000000000000000001 \times 2^{0} \end{array}$$

The result must be normalized. In this case, a left-shift of 31 is required. The exponent of the result is modified accordingly. The result is

$$\begin{array}{rl} & 01.00000000000000000000000000000001 \times 2^{0} \\ - & 01.00000000000000000000000000000000 \times 2^{0} \\ \hline & 01.00000000000000000000000000000000 \times 2^{-31} \end{array}$$

Example 5-8. Floating-Point Addition with a 32-Bit Shift

This example illustrates a situation where a full 32-bit shift is necessary to normalize the result. Let

$$a = 01.11111111111111111111111111111111 \times 2^{127}$$
$$b = 10.00000000000000000000000000000000 \times 2^{127}$$

The operation to be performed is a + b.

$$\begin{array}{rl} & 01.11111111111111111111111111111111 \times 2^{127} \\ + & 10.00000000000000000000000000000000 \times 2^{127} \\ \hline & 11.11111111111111111111111111111111 \times 2^{127} \end{array}$$

Normalizing the result requires a left-shift of 32 and a subtraction of 32 from the exponent. The result is

$$\begin{array}{rl} & 01.11111111111111111111111111111111 \times 2^{127} \\ + & 10.00000000000000000000000000000000 \times 2^{127} \\ \hline & 10.00000000000000000000000000000000 \times 2^{95} \end{array}$$

Example 5-9. Floating-Point Addition/Subtraction and Zero

When floating-point addition and subtraction is performed with a floating-point 0, , the following identities are satisfied:

$$a \pm 0 = a \quad (a \neq 0)$$
$$0 \pm 0 = 0$$
$$0 - a = -a \quad (a \neq 0)$$

5.6 Normalization Using the NORM Instruction

The NORM instruction takes an extended-precision floating-point number, assumed to be unnormalized, and normalizes it. Since the number is assumed to be unnormalized, no implied most-significant nonsign bit is assumed. The NORM instruction executes the following three steps:

1) Locate the most-significant nonsign bit of the floating-point number.
2) Left-shift to normalize the number.
3) Adjust the exponent.

Given the extended-precision floating-point value *a* to be normalized, the normalization (norm ()) is performed as shown in Figure 5-11.

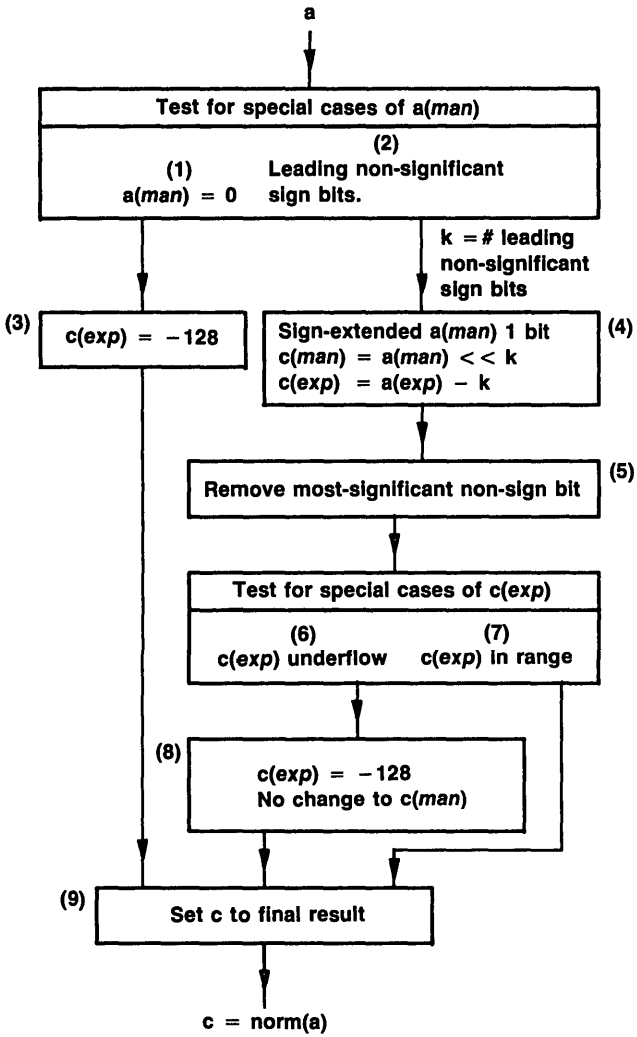


Figure 5-11. Flowchart for NORM Instruction Operation

Example 5-10. NORM Instruction

Assume that an extended-precision register contains the value

man = 00000000000000000001000000000001, *exp* = 0

When the normalization is performed on a number assumed to be unnormalized, the binary point is assumed to be

man = 0.00000000000000000001000000000001, *exp* = 0

This number is then sign-extended one bit so that the mantissa contains 33 bits.

man = 00.00000000000000000001000000000001, *exp* = 0

The intermediate result after the most-significant nonsign bit is located and the shift performed is

man = 01.00000000000100000000000000000000, *exp* = -19

The final 32-bit value output after removing the redundant bit is

man = 00000000000100000000000000000000, *exp* = -19

The NORM instruction is useful for counting the number of leading zeros or leading ones in a 32-bit field. If the exponent is initially zero, the absolute value of the final value of the exponent is the number of leading ones or zeros. This instruction is also useful for manipulating unnormalized floating-point numbers.

5.7 Rounding: The RND Instruction

The RND instruction rounds a number from the extended-precision floating-point format to the single-precision floating-point format. Rounding is similar to floating-point addition. Given the number *a* to be rounded, the following operation is performed first.

$$c = a(man) \times 2^{a(exp)} + (1 \times 2^{(a(exp)-24)})$$

Next a conversion from extended-precision floating-point to single-precision floating-point format is performed. Given the extended-precision floating-point value, the rounding (rnd()) is performed as shown in Figure 5-12.

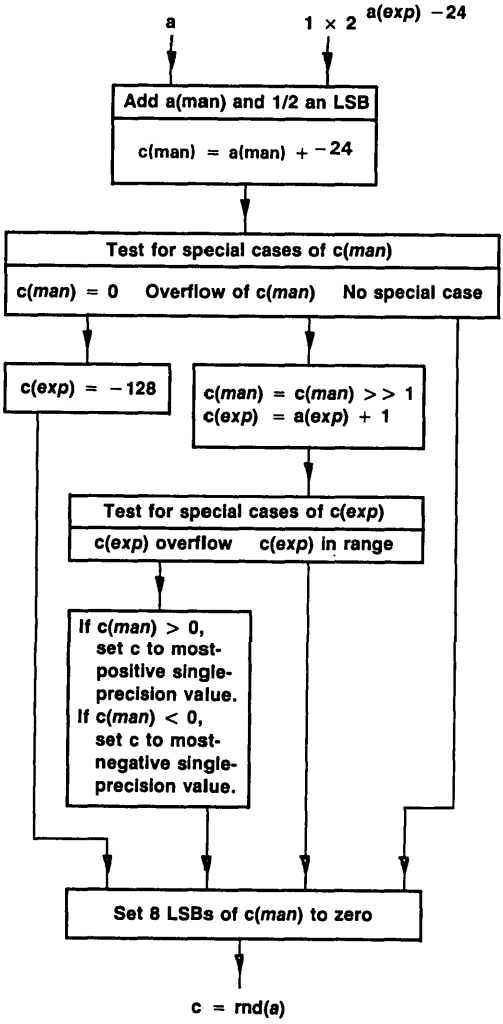


Figure 5-12. Flowchart for Floating-Point Rounding by the RND Instruction

5.8 Floating-Point to Integer Conversion

Floating-point to integer conversion, using the FIX instructions, allow extended-precision floating-point numbers to be converted to single-precision integers in a single cycle. The floating-point to integer conversion of the value x will be referred to here as fix(x). The conversion will not overflow if *a*, the number to be converted, is in the range:

$$-2^{31} \leq a \leq 2^{31} - 1$$

First, it is necessary to be certain that

$$a(exp) \leq 30$$

If these bounds are not met, an overflow occurs. If an overflow occurs in the positive direction, the output is the most positive integer. If an overflow occurs in the negative direction, the output is the most negative integer. If a(*exp*) is within the valid range, then a(*man*), with implied bit included, is sign-extended and right-shifted (rs) by the amount

$$rs = 31 - a(exp)$$

This right-shift (rs) shifts out those bits corresponding to the fractional part of the mantissa. For example:

If $0 \leq x < 1$, then fix(x) = 0.
If $-1 \leq x < 0$, then fix(x) = -1.

The flowchart for the floating-point to integer conversion is shown in Figure 5-13.

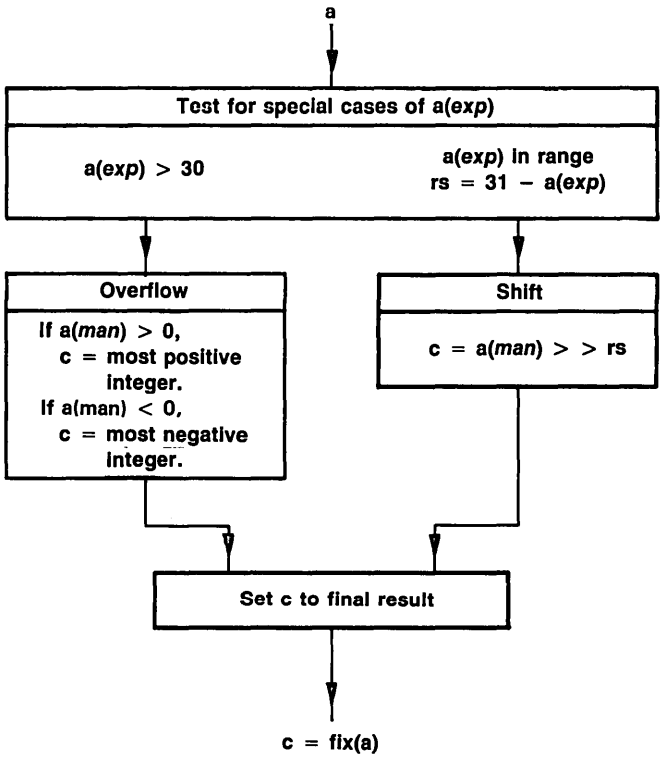


Figure 5-13. Flowchart for Floating-Point to Integer Conversion by FIX Instructions

5.9 Integer to Floating-Point Conversion Using the FLOAT Instruction

Integer to floating-point conversion, using the FLOAT instruction, allows single-precision integers to be converted to extended-precision floating-point numbers. The flowchart for this conversion is shown in Figure 5-14.

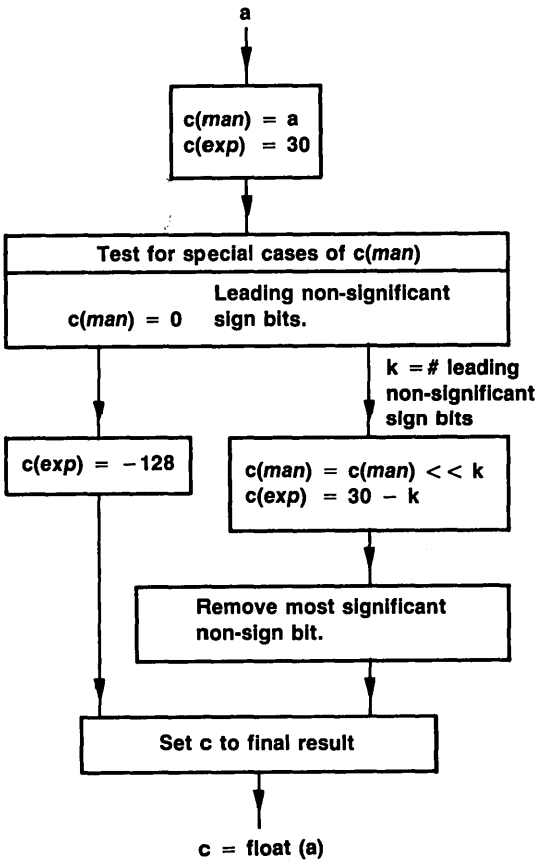


Figure 5-14. Flowchart for Integer to Floating-Point Conversion Using the FLOAT Instruction

Introduction 1

Pinout and Signal Descriptions 2

Architectural Overview 3

CPU Registers, Memory, and Cache 4

Data Formats and Floating-Point Operation 5

Addressing 6

Program Flow Control 7

External Bus Operation 8

Peripherals 9

Pipeline Operation 10

Assembly Language Instructions 11

Software Applications 12

Hardware Applications 13

Appendices A-D

Section 6

Addressing

The TMS320C30 supports five groups of powerful addressing modes. Six types of addressing may be used within the groups, which allow access of data from memory, registers, and the instruction word. This section details the operation, encoding, and implementation of the addressing modes. Also discussed is the management of system stacks, queues, and deques in memory. The major topics in this section are:

- Types of Addressing (Section 6.1 on page 6-2)
  - Register
  - Direct
  - Indirect
  - Short-immediate
  - Long-immediate
  - PC-relative
- Groups of Addressing Modes (Section 6.2 on page 6-18)
  - General addressing modes
  - Three-operand addressing modes
  - Parallel addressing modes
  - Long-immediate addressing mode
  - Conditional-branch addressing modes
- Circular Addressing (Section 6.3 on page 6-22)
- Bit-Reversed Addressing (Section 6.4 on page 6-26)
- System Stack Management (Section 6.5 on page 6-27)

6.1 Types of Addressing

Six types of addressing allow access of data from memory, registers, and the instruction word. They are:

- Register
- Direct
- Indirect
- Short-immediate
- Long-immediate
- PC-relative

Some types of addressing are appropriate for some instructions and not others. For this reason, the types of addressing are used in the five different groups of addressing modes as follows:

- General addressing modes (G):
  - Register
  - Direct
  - Indirect
  - Short-immediate
- Three-operand addressing modes (T):
  - Register
  - Indirect
- Parallel addressing modes (P):
  - Register
  - Indirect
- Long-immediate addressing mode
  - Long-immediate
- Conditional-branch addressing modes (B):
  - Register
  - PC-relative

The six types of addressing will be discussed first, followed by the five groups of addressing modes.

6.1.1 Register Addressing

In register addressing, the operand is contained in a CPU register, as shown in the example below.

```
ABSF   R1          ; R1 = |R1|
```

The syntax for the CPU registers, the assembler syntax, and the assigned function for those registers are listed in Table 6-1.

Table 6-1. CPU Register/Assembler Syntax and Function

CPU REGISTER ADDRESS	ASSEMBLER SYNTAX	ASSIGNED FUNCTION
00h	R0	Extended-precision register
01h	R1	Extended-precision register
02h	R2	Extended-precision register
03h	R3	Extended-precision register
04h	R4	Extended-precision register
05h	R5	Extended-precision register
06h	R6	Extended-precision register
07h	R7	Extended-precision register
08h	AR0	Auxiliary register
09h	AR1	Auxiliary register
0Ah	AR2	Auxiliary register
0Bh	AR3	Auxiliary register
0Ch	AR4	Auxiliary register
0Dh	AR5	Auxiliary register
0Eh	AR6	Auxiliary register
0Fh	AR7	Auxiliary register
10h	DP	Data page pointer
11h	IR0	Index register 0
12h	IR1	Index register 1
13h	BK	Block size
14h	SP	*Active stack pointer*
15h	ST	Status register
16h	IE	CPU/DMA interrupt enable
17h	IF	CPU interrupt flags
18h	IOF	I/O flags
19h	RS	Repeat start address
1Ah	RE	Repeat end address
1Bh	RC	Repeat counter

6.1.2 Direct Addressing

In direct addressing, the data address is formed by the concatenation of the eight least-significant bits of the data page pointer (DP) with the 16 least-significant bits of the instruction word (expr). This results in 256 pages (64 K words per page), giving the programmer a large address space without needing to change the page pointer. The syntax and operation for direct addressing are listed below.

Syntax: @expr

Operation: address = DP concatenated with expr

Figure 6-1 shows the formation of the data address. Example 6-1 gives an instruction example with data before and after instruction execution.

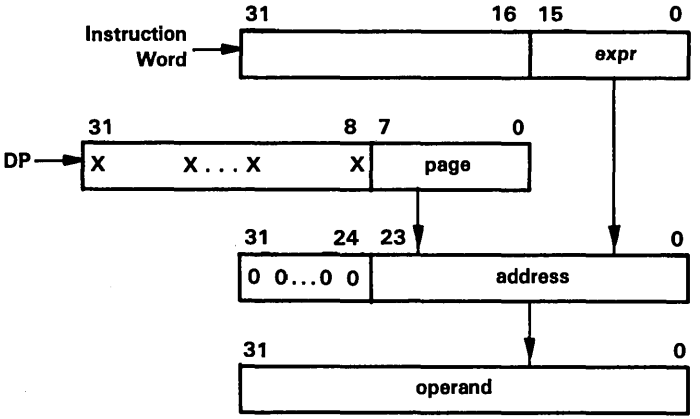


Figure 6-1. Direct Addressing

Example 6-1. Direct Addressing

```
ADDI  @0BCDEh,R7
```

Before Instruction:

DP = 8Ah
R7 = 0h
Data at 8ABCDEh = 12345678h

After Instruction:

DP = 8Ah
R7 = 12345678h
Data at 8ABCDEh = 12345678h

6.1.3 Indirect Addressing

Indirect addressing is used to specify the address of an operand in memory through the contents of an auxiliary register, optional displacements, and index registers. Only the 24 least-significant bits of the auxiliary registers and index registers are used in indirect addressing. This arithmetic is performed by the auxiliary register arithmetic units (ARAUs) on these lower 24 bits and is unsigned. The upper eight bits are unmodified.

The flexibility of indirect addressing is possible because the ARAUs on the TMS320C30 are used to modify auxiliary registers in parallel with operations within the main CPU. Indirect addressing is specified by a five-bit field in the instruction word, referred to as the mod field. A displacement is either an explicit unsigned 8-bit integer contained in the instruction word or an implicit displacement of one. Two index registers, IR0 and IR1, can also be used in indirect addressing. In some cases, an addressing scheme using circular or bit reversed addressing is optional. The mechanism for generating addresses in circular addressing is discussed in Section 6.3, bit reversed in Section 6.4.

Table 6-2 lists the various kinds of indirect addressing, along with the value of the modification (mod) field, assembler syntax, operation, and function for each. The succeeding 18 examples show the operation for each kind of indirect addressing.

Table 6-2. Indirect Addressing

MOD FIELD	SYNTAX	OPERATION	DESCRIPTION
INDIRECT ADDRESSING WITH DISPLACEMENT			
00000	*+ARn(disp)	addr = ARn + disp	With pre-displacement add
00001	*-ARn(disp)	addr = ARn - disp	With pre-displacement subtract
00010	*++ARn(disp)	addr = ARn + disp ARn = ARn + disp	With pre-displacement add and modify
00011	*--ARn(disp)	addr = ARn - disp ARn = ARn - disp	With pre-displacement subtract and modify
00100	*ARn++(disp)	addr = ARn ARn = ARn + disp	With post-displacement add and modify
00101	*ARn--(disp)	addr = ARn ARn = ARn - disp	With post-displacement subtract and modify
00110	*ARn++(disp)%	addr = ARn ARn = circ(ARn + disp)	With post-displacement add and circular modify
00111	*ARn--(disp)%	addr + ARn ARn = circ(ARn - disp)	With post-displacement subtract and circular modify
INDIRECT ADDRESSING WITH INDEX REGISTER IR0			
01000	*+ARn(IR0)	addr = ARn + IR0	With pre-index (IR0) add
01001	*-ARn(IR0)	addr = ARn - IR0	With pre-index (IR0) subtract
01010	*++ARn(IR0)	addr = ARn + IR0 ARn = ARn + IR0	With pre-index (IR0) add and modify
01011	*--ARn(IR0)	addr = ARn - IR0 ARn = ARn - IR0	With pre-index (IR0) subtract and modify
01100	*ARn++(IR0)	addr = ARn ARn = ARn + IR0	With post-index (IR0) add and modify
01101	*ARn--(IR0)	addr = ARn ARn = ARn - IR0	With post-index (IR0) subtract and modify
01110	*ARn++(IR0)%	addr = ARn ARn = circ(ARn + IR0)	With post-index (IR0) add and circular modify
01111	*ARn--(IR0)%	addr = ARn ARn = circ(ARn - IR0)	With post-index (IR0) subtract and circular modify

LEGEND:

addr = memory address
ARn = auxiliary register AR0 - AR7
IRn = index register IR0 or IR1
disp = displacement
++ = add and modify
-- = subtract and modify
circ() = address in circular addressing
% = where circular addressing is performed
B = where bit-reversed addressing is performed

Table 6-2. Indirect Addressing (Concluded)

MOD FIELD	SYNTAX	OPERATION	DESCRIPTION
INDIRECT ADDRESSING WITH INDEX REGISTER IR1			
10000	*+ARn(IR1)	addr = ARn + IR1	With pre-index (IR1) add
10001	*-ARn(IR1)	addr = ARn - IR1	With pre-index (IR1) subtract
10010	*++ARn(IR1)	addr = ARn + IR1 ARn = ARn + IR1	With pre-index (IR1) add and modify
10011	*--ARn(IR1)	addr = ARn - IR1 ARn = ARn - IR1	With pre-index (IR1) subtract and modify
10100	*ARn++(IR1)	addr = ARn ARn = ARn + IR1	With post-index (IR1) add and modify
10101	*ARn--(IR1)	addr = ARn ARn = ARn - IR1	With post-index (IR1) subtract and modify
10110	*ARn++(IR1)%	addr = ARn ARn = circ(ARn + IR1)	With post-index (IR1) add and circular modify
10111	*ARn--(IR1)%	addr = ARn ARn = circ(ARn - IR1)	With post-index (IR1) subtract and circular modify
INDIRECT ADDRESSING (SPECIAL CASES)			
11000	*ARn	addr = ARn	Indirect
11001	*ARn++(IR0)B	addr = ARn ARn = B(ARn + IR0)	With post-index (IR0) add and bit-reversed modify

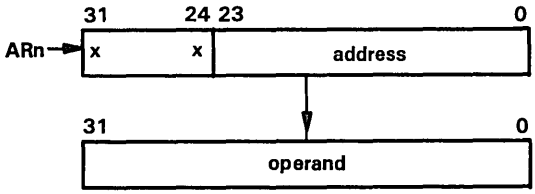
LEGEND:

addr = memory address
ARn = auxiliary register AR0 - AR7
IRn = index register IR0 or IR1
disp = displacement
++ = add and modify
-- = subtract and modify
circ() = address in circular addressing
% = where circular addressing is performed
B = where bit-reversed addressing is performed

Example 6-2. Auxiliary Register Indirect

The address of the operand to be fetched is contained in an auxiliary register (ARn).

Operation:	operand address = AR*n*
Assembler Syntax:	*AR*n*
Modification Field:	11000



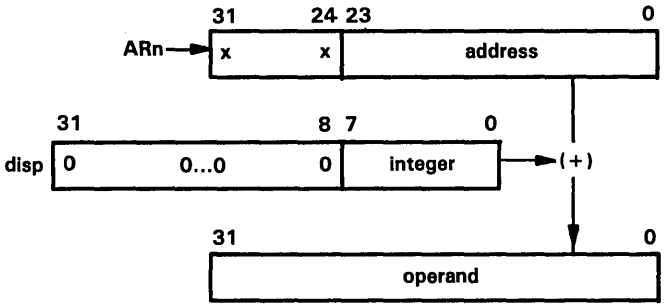
Example 6-3. Indirect with Pre-Displacement Add

The address of the operand to be fetched is the sum of an auxiliary register (ARn) and the displacement (disp). The displacement is either an eight-bit unsigned integer contained in the instruction word or an implied value of 1.

Operation: operand address = AR*n*+disp

Assembler Syntax: *+AR*n*(disp)

Modification Field: 00000



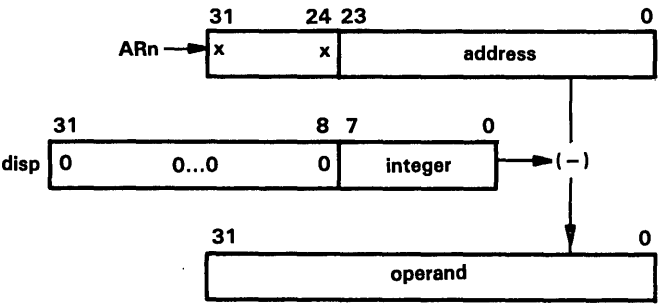
Example 6-4. Indirect with Pre-Displacement Subtract

The address of the operand to be fetched is the contents of an auxiliary register (ARn) minus the displacement (disp). The displacement is either an eight-bit unsigned integer contained in the instruction word or an implied value of 1.

Operation: operand address = AR*n*–disp

Assembler Syntax: *–AR*n*(disp)

Modification Field: 00001



Example 6-5. Indirect with Pre-Displacement Add and Modify

The address of the operand to be fetched is the sum of an auxiliary register (ARn) and the displacement (disp). The displacement is either an eight-bit unsigned integer contained in the instruction word or an implied value of 1. After the data is fetched, the auxiliary register is updated with the address generated.

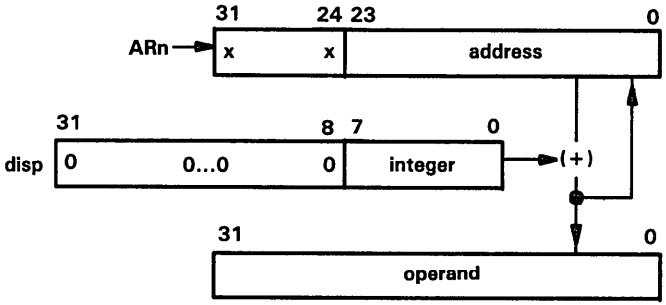
1

Operation: operand address = AR*n*+disp

ARn=AR*n*+disp

Assembler Syntax: *++AR*n*(disp)

Modification Field: *00010*



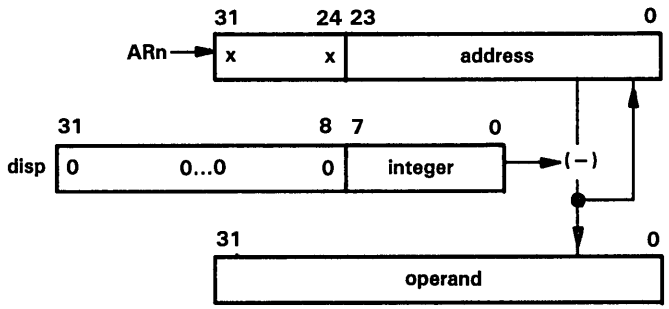
Example 6-6. Indirect with Pre-Displacement Subtract and Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn) minus the displacement (disp). The displacement is either an eight-bit unsigned integer contained in the instruction word or an implied value of 1. After the data is fetched, the auxiliary register is updated with the address generated.

Operation: operand address = AR*n*-disp
ARn=AR*n*+disp

Assembler Syntax: *--AR*n*(disp)

Modification Field: 00010



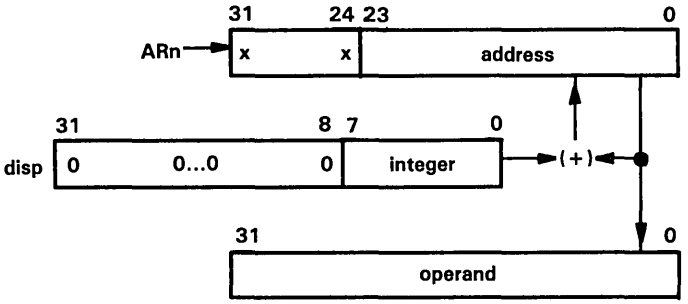
Example 6-7. Indirect with Post-Displacement Add and Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn). After the operand is fetched, the displacement (disp) is added to the auxiliary register. The displacement is either an eight-bit unsigned integer contained in the instruction word or an implied value of 1.

Operation: operand address = AR*n*
AR*n*=AR*n*+disp

Assembler Syntax: *AR*n*++(disp)

Modification Field: 00100



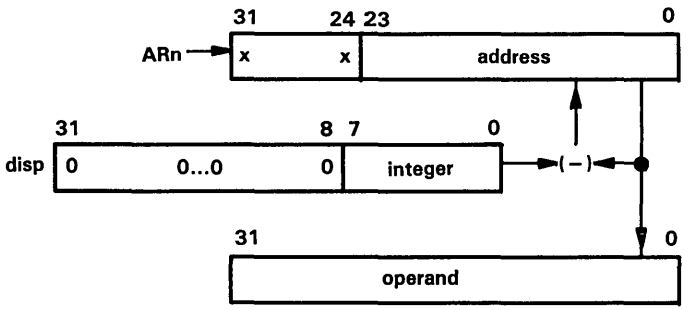
Example 6-8. Indirect with Post-Displacement Subtract and Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn). After the operand is fetched, the displacement (disp) is subtracted from the auxiliary register. The displacement is either an eight-bit unsigned integer contained in the instruction word or an implied value of 1.

Operation: operand address = AR*n*
AR*n*=AR*n*-disp

Assembler Syntax: *AR*n*--(disp)

Modification Field: 00101



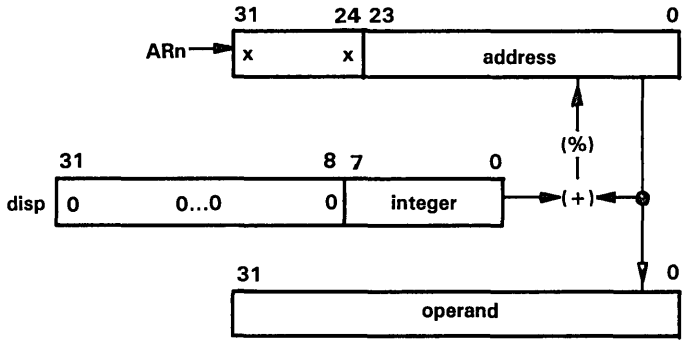
Example 6-9. Indirect with Post-Displacement Add and Circular Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn). After the operand is fetched, the displacement (disp) is added to the contents of the auxiliary register using circular addressing. This result is used to update the auxiliary register. The displacement is either an eight-bit unsigned integer contained in the instruction word or an implied value of 1.

Operation: operand address = AR*n*
AR*n*=circ(AR*n*+disp)

Assembler Syntax: *AR*n*++(disp)%

Modification Field: 00110



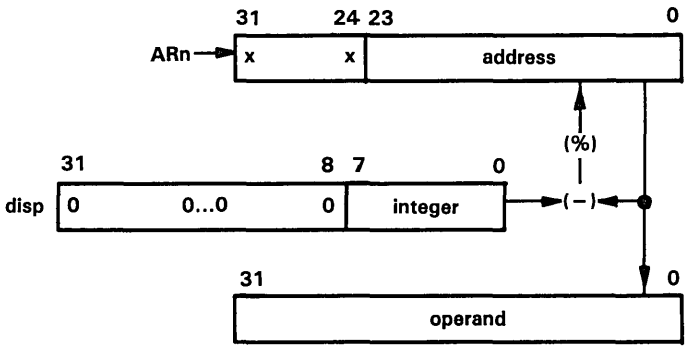
Example 6-10. Indirect with Post-Displacement Subtract and Circular Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn). After the operand is fetched, the displacement (disp) is subtracted from the contents of the auxiliary register using circular addressing. This result is used to update the auxiliary register. The displacement is either an eight-bit unsigned integer contained in the instruction word or an implied value of 1.

Operation: operand address = AR*n*
AR*n*=circ(AR*n*-disp)

Assembler Syntax: *AR*n*--(disp)%

Modification Field: 00111



Example 6-11. Indirect with Pre-Index Add

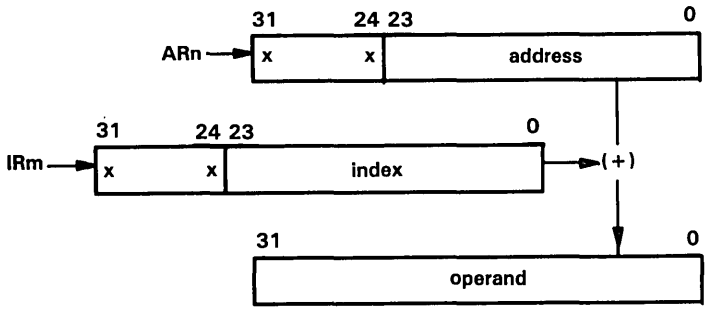
The address of the operand to be fetched is the sum of an auxiliary register (ARn) and an index register (IR0 or IR1). generated.

Operation: operand address = AR*n*+IR*m*

Assembler Syntax: *+AR*n*(IR*m*)

Modification Field:

01000	if m=0
10000	if m=1



Example 6-12. Indirect with Pre-Index Subtract

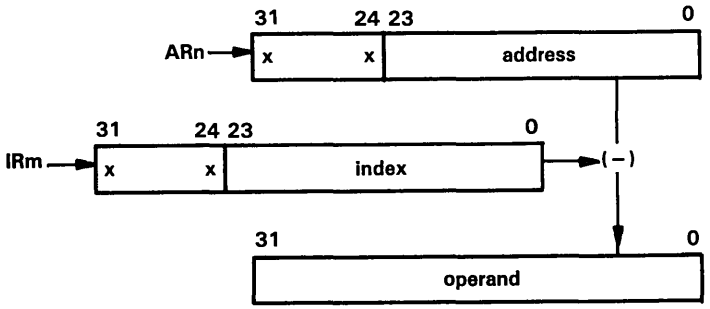
The address of the operand to be fetched is the difference of an auxiliary register (ARn) and an index register (IR0 or IR1).

Operation: operand address = AR*n*-IR*m*

Assembler Syntax: *-AR*n*(IR*m*)

Modification Field:

01001	if m=0
10001	if m=1



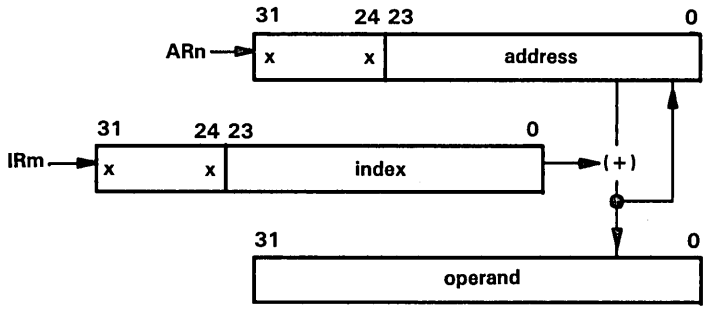
Example 6-13. Indirect with Pre-Index Add and Modify

The address of the operand to be fetched is the sum of an auxiliary register (ARn) and an index register (IR0 or IR1). After the data is fetched, the auxiliary register is updated with the address generated.

Operation: operand address = AR*n*+IR*m*
AR*n*=AR*n*+IR*m*

Assembler syntax: *++AR*n*(IR*m*)

Modification Field: 01010 if m=0
10010 if m=1



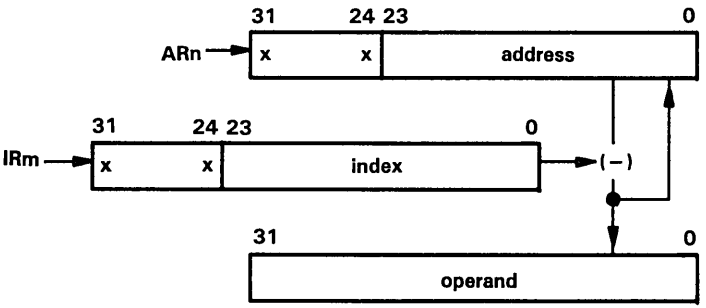
Example 6-14. Indirect with Pre-Index Subtract and Modify

The address of the operand to be fetched is the difference of an auxiliary register (ARn) and an index register (IR0 or IR1). The resulting address becomes the new contents of the auxiliary register.

Operation: operand address = AR*n*-IR*m*
AR*n*=AR*n*-IR*m*

Assembler Syntax: *--AR*n*(IR*m*)

Modification Field: 01011 if m=0
10011 if m=1



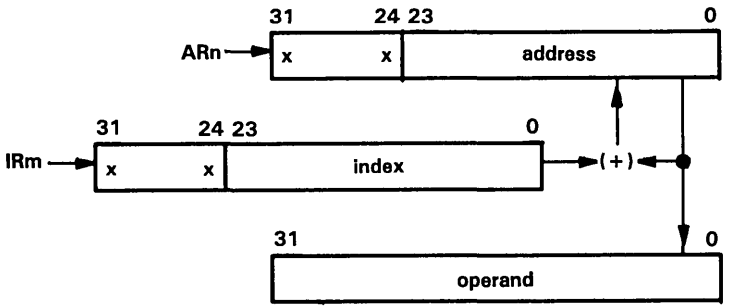
Example 6-15. Indirect with Post-Index Add and Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn). After the operand is fetched, the index register (IR0 or IR1) is added to the auxiliary register.

Operation: operand address = AR*n*
AR*n*=AR*n*+IR*m*

Assembler Syntax: *AR*n*++(IR*m*)

Modification Field: 01100 if m=0
10100 if m=1



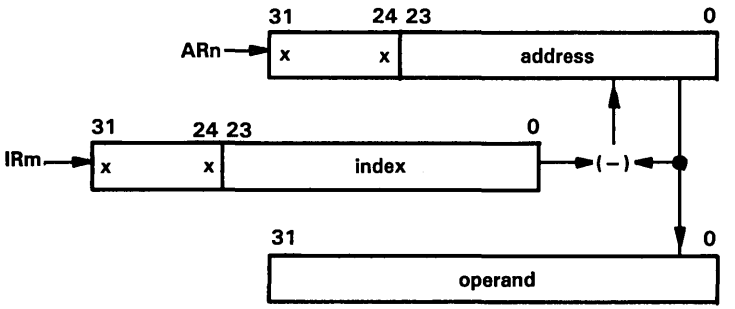
Example 6-16. Indirect with Post-Index Subtract and Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn). After the operand is fetched, the index register (IR0 or IR1) is subtracted from the auxiliary register.

Operation: operand address = AR*n*
AR*n*=AR*n*-IR*m*

Assembler Syntax: *AR*n*--(IR*m*)

Modification Field: 01101 if m=0
10101 if m=1



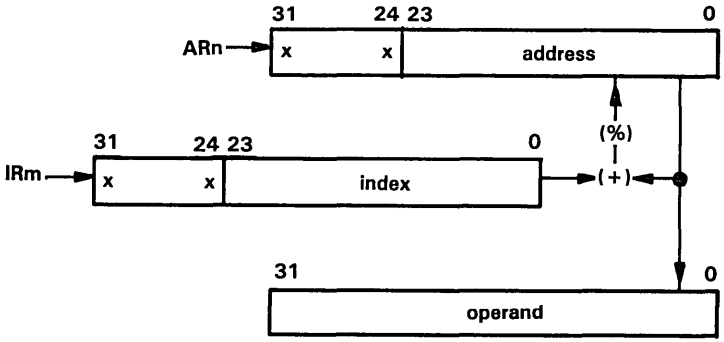
Example 6-17. Indirect with Post-Index Add and Circular Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn). After the operand is fetched, the index register (IR0 or IR1) is added to the auxiliary register. This value is evaluated using circular addressing and replaces the contents of the auxiliary register.

Operation: operand address = AR*n*
AR*n*=circ(AR*n*+IR*m*)

Assembler Syntax: *AR*n*++(IR*m*)%

Modification Field: 01110 if m=0
10110 if m=1



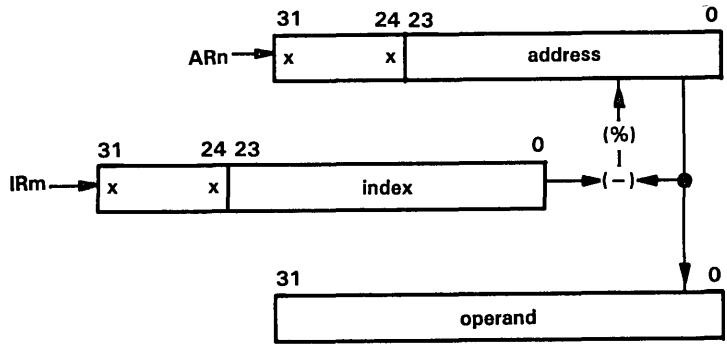
Example 6-18. Indirect with Post-Index Subtract and Circular Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn). After the operand is fetched, the index register (IR0 or IR1) is subtracted from the auxiliary register. This value is evaluated using circular addressing and replaces the contents of the auxiliary register.

Operation: operand address = AR*n*
AR*n*=circ(AR*n*-IR*m*)

Assembler Syntax: *AR*n*--(IR*m*)%

Modification Field: 01111 if m=0
10111 if m=1



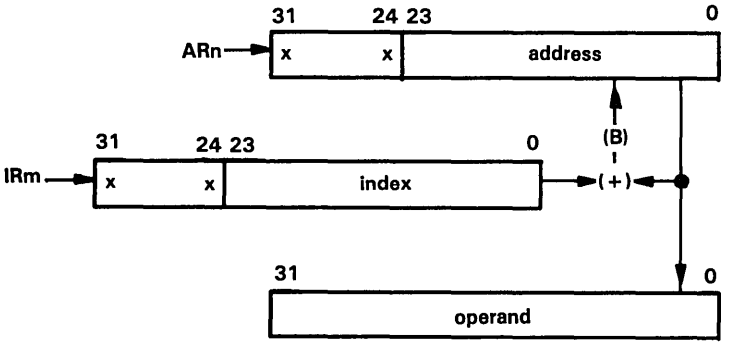
Example 6-19. Indirect with Post-Index Add and Bit-Reversed Modify

The address of the operand to be fetched is the contents of an auxiliary register (ARn). After the operand is fetched, the index register (IR0) is added to the auxiliary register. This addition is performed with a reverse-carry propagation and can be used to yield a bit-reversed (B) address. This value replaces the contents of the auxiliary register.

Operation: operand address = AR*n*
AR*n*=B(AR*n*+IR0)

Assembler Syntax: *AR*n*++(IR0)B

Modification Field: 11001



6.1.4 Short-Immediate Addressing

In short-immediate addressing, the operand is a 16-bit immediate value contained in the 16 least-significant bits of the instruction word (expr). Depending upon the data types assumed for the instruction, the short-immediate operand may be a two's-complement integer, an unsigned integer, or a floating-point number. The syntax for this mode is listed below.

Syntax: expr

Example 6-20 gives an instruction example with before and after instruction data.

Example 6-20. Short-Immediate Addressing

```
SUBI  1,R0
```

Before Instruction: R0 = 0h

After Instruction: R0 = 0FFFFFFFFh

6.1.5 Long-Immediate Addressing

In long-immediate addressing, the operand is a 24-bit immediate value contained in the 24 least-significant bits of the instruction word (expr). The syntax for this mode is listed below.

Syntax: expr

Example 6-21 gives an instruction example with before and after instruction data.

Example 6-21. Long-Immediate Addressing

```
BR    8000h
```

Before Instruction:	**After Instruction:**
PC = 0h	PC = 8000h

6.1.6 PC-Relative Addressing

PC-relative addressing is used for branching. It replaces the value of the PC based upon the contents of the 16 least significant bit of the instruction word. The assembler takes the src (a label or address) specified by the user and generates a displacement. If the branch is a standard branch, this displacement is equal to the label - (PC+1). If the branch is a delayed branch, this displacement is equal to the label - (PC+3).

The displacement is stored as a 16-bit signed integer in the least significant bits of the instruction word.

Syntax: expr

Example 6-22 gives an instruction example with before and after instruction data.

Example 6-22. PC-Relative Addressing

```
BU    NEWPC          ; pc=1, NEWPC=5, displacement=3
```

Before Instruction:	**After Instruction:**
PC = 1h	PC = 5h

6.2 Groups of Addressing Modes

The types of addressing are used to form the following five groups of addressing modes:

- General addressing modes (G)
- Three-operand addressing modes (T)
- Parallel addressing modes (P)
- Long-immediate addressing mode
- Conditional-branch addressing modes (B)

These groups of addressing modes are discussed in the following sections.

6.2.1 General Addressing Modes

Instructions that use the general addressing modes are general-purpose instructions, such as ADDI, MPYF, and LSH. Such instructions are usually of the form:

dst operation *src* → *dst*

where the destination operand is signified by *dst*, the source operand by *src*, and 'operation' defines an operation to be performed using the general addressing modes to specify certain operands. Bits 31-29 are zero, indicating general addressing mode instructions. Bits 22 and 21 specify the general addressing mode (G) field, which defines how bits 15 through 0 are to be interpreted for addressing the *src* operand.

Options for bits 22 and 21 (G field) are as follows:

0 0 register (all CPU registers unless specified otherwise)
0 1 direct
1 0 indirect
1 1 immediate

If the *src* and *dst* fields contain register specifications, the value in these fields contains the CPU register addresses as defined by Table 6-1. For the general addressing modes, the following values of ARn are valid:

ARn, $0 \leq n \leq 7$

Figure 6-2 shows the encoding for the general addressing modes. The notation 'modn' indicates the modification field that goes with the ARn field. Refer to Table 6-2 for further information.

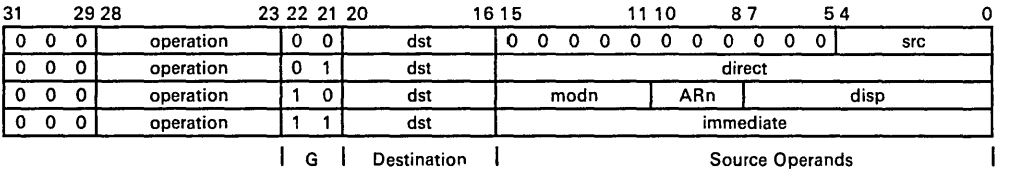


Figure 6-2. Encoding for General Addressing Modes

6.2.2 Three-Operand Addressing Modes

Instructions that use the three-operand addressing modes, such as ADDI3, LSH3, CMPF3. or XOR3, are usually of the form:

SRC1 operation SRC2 → *dst*

where the destination operand is signified by *dst*, the source operands by SRC1 and SRC2, and 'operation' defines an operation to be performed. Note that the '3' can be omitted from three-operand instructions.

Bits 31-29 are set to the value of 001, indicating three-operand addressing mode instructions. Bits 22 and 21 specify the three-operand addressing mode (T) field, which defines how bits 15-0 are to be interpreted for addressing the SRC operands. Bits 15-8 are used to define the SRC1 address, and bits 7-0 to define the SRC2 address. Options for bits 22 and 21 (T) are as follows:

T	SRC1	SRC2
0 0	Register	Register
0 1	Indirect	Register
1 0	Register	Indirect
1 1	Indirect	Indirect

Figure 6-3 shows the encoding for three-operand addressing. If the SRC1 and SRC2 fields use the same auxiliary register, both addresses are correctly generated. However, only the value created by the SRC1 field is saved in the auxiliary register specified. The assembler issues a warning if this condition is specified by the user.

The following values of ARn and ARm are valid:

$$ARn,0 \leq n \leq 7$$
$$ARm,0 \leq m \leq 7$$

The notation "modm" or "modn" indicates the modification field goes with the ARm or ARn field respectively. Refer to Table 6-2 for further information.

In indirect addressing of the three-operand addressing mode, displacements (if used) are allowed of 0 or 1, and the index registers (IR0 and IR1) can be used. The displacement of 1 is implied and is not explicitly coded in the instruction word.

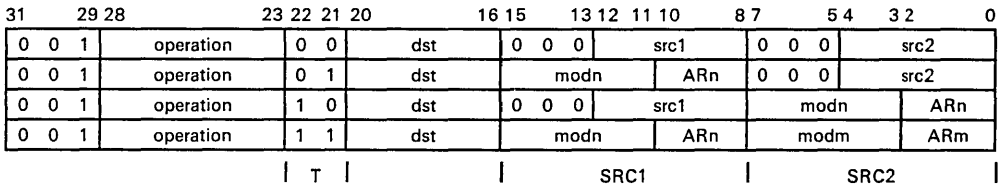


Figure 6-3. Encoding for Three-Operand Addressing Modes

6.2.3 Parallel Addressing Modes

Instructions that use parallel addressing (indicated by || (two vertical bars)) allow for the greatest amount of parallelism possible. The destination operands are indicated as d1 and d2, signifying *dst*1 and *dst*2, respectively (see Figure 6-4). The source operands, signified by *src*1 and *src*2, use the extended-precision registers. The parallel operation to be performed is notated as 'operation'.

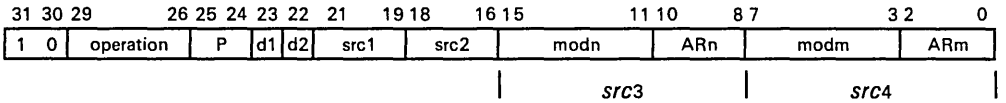


Figure 6-4. Encoding for Parallel Addressing Modes

The parallel addressing mode (P) field specifies how the operands are to be used, i.e., whether they are source or destination. The specific relationship between the P field and the operands is detailed in the description of the individual parallel instructions (see Section 11). However, the operands are always encoded in the same way. Bits 31 and 30 are set to the value of 10, indicating parallel addressing mode instructions. Bits 25 and 24 specify the parallel addressing mode (P) field, which defines how bits 21-0 are to be interpreted for addressing the *src* operands. Bits 21-19 are used to define the *src*1 address, bits 18-16 to define the *src*2 address, bits 15-8 the *src*3 address, and bits 7-0 the *src*4 address. The notation 'modn' and 'modm' indicate which modification field goes with which ARn or ARm (auxiliary register) field, respectively. The parallel addressing operands are listed below.

src1 $0 \leq src1 \leq 7$ (extended-precision registers R0-R7)
src2 $0 \leq src2 \leq 7$ (extended-precision registers R0-R7)
d1 If 0, *dst1* is R0. If 1, *dst1* is R1.
d2 If 0, *dst2* is R2. If 1, *dst2* is R3.
P $0 \leq P \leq 3$
src3 indirect (disp = 0, 1, IR0, IR1)
src4 indirect (disp = 0, 1, IR0, IR1)

As in the three-operand addressing mode, indirect addressing in the parallel addressing mode allows for displacements of 0 or 1 and the use of the index registers (IR0 and IR1). The displacement of 1 is implied and is not explicitly coded in the instruction word.

In the encoding shown for this mode in Figure 6-4, if the *src*3 and *src*4 fields use the same auxiliary register, both addresses are correctly generated, but only the value created by the *src*3 field is saved in the auxiliary register specified. The assembler issues a warning if this condition is specified by the user.

6.2.4 Long-Immediate Addressing Mode

The long-immediate addressing mode is used to encode the program control instructions (BR, BRD, and CALL), for which it is useful to have a 24-bit absolute address contained in the instruction word. The unconditional branches, BR (standard) and BRD (delayed), use the long-immediate addressing mode. Bits 31-25 are set to the value of 0110000, indicating long-immediate addressing mode instructions. Selection of bit 24 determines the type of branch: D = 0 for a standard branch or D = 1 for a delayed branch. The long-immediate operand is the 24-bit *src*. These instructions are encoded as shown in Figure 6-5.

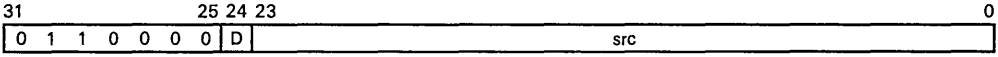


Figure 6-5. Encoding for Long-Immediate Addressing Mode

6.2.5 Conditional-Branch Addressing Modes

Instructions using the conditional-branch addressing modes (B*cond*, B*cond*D, CALL*cond*, DB*cond*, and DB*cond*D) can perform a variety of conditional operations. Bits 31-27 are set to the value of 01101, indicating conditional-branch addressing mode instructions. Bit 26 is set to 0 or 1, the former selects DB*cond*, the latter B*cond*. Selection of bit 25 determines the conditional-branch addressing mode (B). If B = 0, register addressing is used; if B = 1, PC-relative addressing is used. Selection of bit 21 sets the type of branch: D = 0 for a standard branch or D = 1 for a delayed branch. The condition field (*cond*) specifies the condition checked to determine what action to take, i.e., whether or not to branch (see Section 11 for a list of condition codes). Figure 6-6 shows the encoding for conditional-branch addressing.

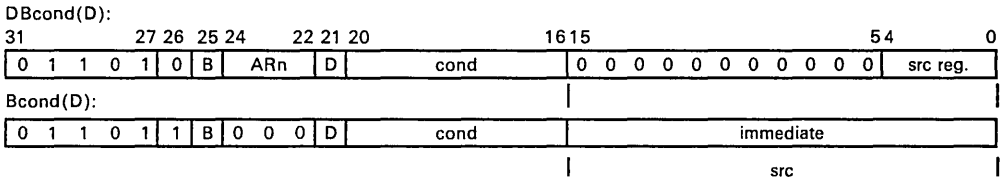


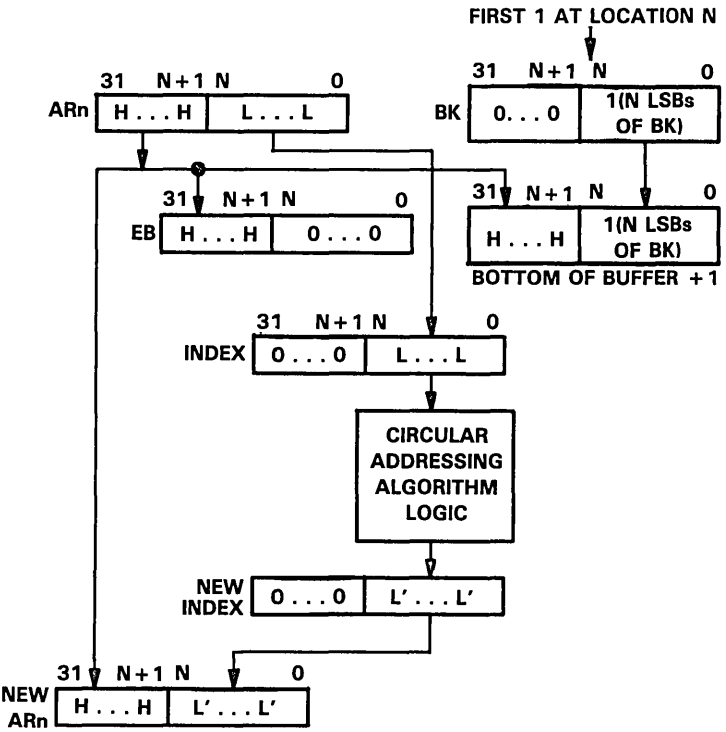
Figure 6-6. Encoding for Conditional-Branch Addressing Modes

6.3 Circular Addressing

Many algorithms, such as convolution and correlation, require the implementation of a circular buffer in memory. In convolution and correlation, the circular buffer is used to implement a sliding window which contains the most recent data to be processed. As new data is brought in, the new data overwrites the oldest data. Key to the implemention of a circular buffer is the implementation of a circular addressing mode. This section describes the circular addressing mode of the TMS320C30.

The blocksize register (BK) specifies the size of the circular buffer. Information concerning the lower 16 bits of the BK register plus a user-selected auxiliary register (ARn) are used to specify the bottom of the circular buffer. The information concerning the BK register is the location of the the first 1 bit, counting from the most-significant bit to the least-significant bit, in the lower 16 bits. With the location of the first 1 bit specified as bit N, the address at the top of the buffer is referred to as the effective base (EB) and is equal to bits 31 through (N+1) of ARn with bits N through 0 of EB being zero.

Figure 6-7 illustrates the relationships between the blocksize register (BK), the auxiliary registers (ARn), the bottom of the circular buffer, the top of the circular buffer, and the index into the circular buffer.



LEGEND:

ARn	= auxiliary register n	L	= low-order bits
BK	= blocksize register	L'	= new low-order bits
EF	= effective base	LSB	= least-significant bit
H	= high-order bits	N	= bit value

Figure 6-7. Flowchart for Circular Addressing

In circular addressing, 'index' refers to the N LSBs of the auxiliary register selected, and 'step' is the quantity being added to or subtracted from the auxiliary register. The following two rules must be followed when using circular addressing:

- The step used must be less than or equal to the blocksize.
- The first time the circular queue is addressed, the auxiliary register must be pointing to an element in the circular queue.

The algorithm for circular addressing is as follows:

```
If 0 ≤ index + step < BK:
    index = index + step.
Else if index + step ≥ BK:
    index = index + step - BK.
Else if index + step < 0:
    index = index + step + BK.
```

Figure 6-8 shows how the circular buffer is implemented. It illustrates the relationship of the quantities generated and the elements in the circular buffer.

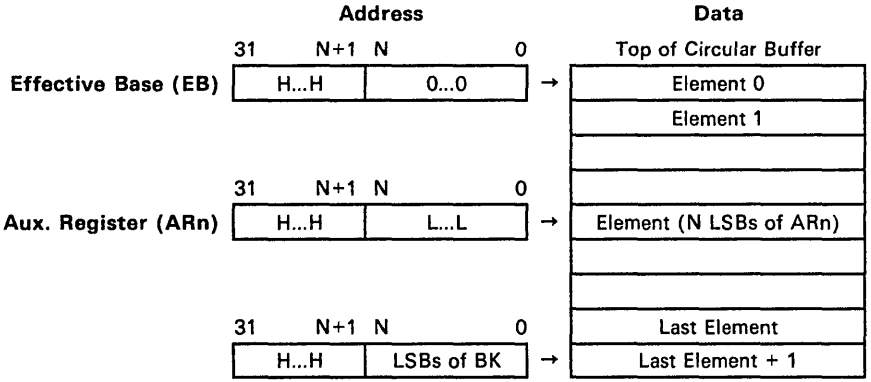


Figure 6-8. Circular Buffer Implementation

Figure 6-9 provides an example that shows the operation of circular addressing. Assuming that all ARs are four bits, let AR0 = 0000,and BK = 0110 (blocksize of 6). This example shows a sequence of modifications and the resulting value of AR0. It also shows how the pointer steps through the circular queue, with a variety of step sizes (both incrementing and decrementing).

```
*AR0          ; AR0 = 0 (0th value)
*AR0++(5)%    ; AR0 = 5 (1st value)
*AR0++(2)%    ; AR0 = 1 (2nd value)
*AR0--(3)%    ; AR0 = 4 (3rd value)
*AR0++(6)%    ; AR0 = 4 (4th value)
*AR0--%       ; AR0 = 3 (5th value)
```

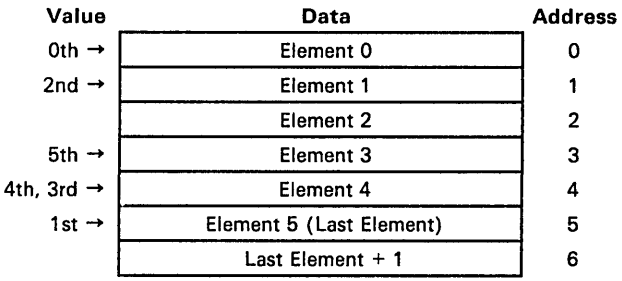


Figure 6-9. Circular Addressing Example

Circular addressing is especially useful for the implementation of FIR filters. Figure 6-10 shows one possible data structure for FIR filters. Note that the initial value of AR0 points to h(N-1), and the initial value of AR1 points to x(0). Circular addressing is used in the TMS320C30 code for the FIR filter shown in Figure 6-11.

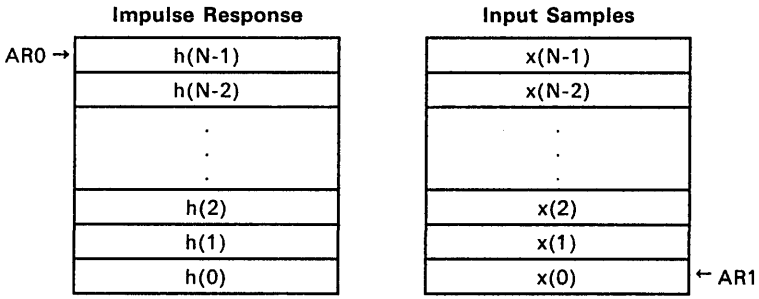


Figure 6-10. Data Structure for FIR Filters

```
* Initialization
*
     LDI    N,BK              ; Load block size.
     LDI    H,AR0             ; Load pointer to impulse response.
     LDI    X,AR1             ; Load pointer to bottom of input
*                             ; sample buffer.
*
TOP  LDF    IN, R3            ; Read input sample.
     STF    R3,*AR1++%        ; Store with other samples.
                              ; and point to top of buffer.
     LDF    0,R0              ; Initialize R0.
     LDF    0,R2              ; Initialize R2.
*
*    Filter
*
     RPTS   N-1               ; Repeat next instruction.
     MPYF3  *AR0++%,*AR1++%,R0
||   ADDF3  R0,R2,R2          ; Multiply and accumulate.
     ADDF   R0,R2             ; Last product accumulated.
*
     STF    R2,Y              ; Save result.
     B      TOP               ; Repeat.
```

Figure 6-11. FIR Filter Code Using Circular Addressing

6.4 Bit-Reversed Addressing

Bit-reversed addressing on the TMS320C30 is useful in FFT algorithms using a variety of radices. One auxiliary register is used as a pointer to the physical location of a data value. IR0 is used to specify the size of the FFT; e.g., the value contained in IR0 must be equal to 2^n where n is an integer. By adding IR0 to the auxiliary register using bit-reversed addressing, addresses are generated in a bit-reversed fashion.

To illustrate this kind of addressing, assume eight-bit auxiliary registers. Let AR2 contain the value 0110 0000 (96). This is the base address of the data in memory. Let IR0 contain the value 0000 1000 (8). Figure 6-12 shows a sequence of modifications of AR2 and the resulting values of AR2.

```
*AR2              ; AR2 = 0110 0000 (0th value)
*AR2++(IR0)B      ; AR2 = 0110 1000 (1st value)
*AR2++(IR0)B      ; AR2 = 0110 0100 (2nd value)
*AR2++(IR0)B      ; AR2 = 0110 1100 (3rd value)
*AR2++(IR0)B      ; AR2 = 0110 0010 (4th value)
*AR2++(IR0)B      ; AR2 = 0110 1010 (5th value)
*AR2++(IR0)B      ; AR2 = 0110 0110 (6th value)
*AR2++(IR0)B      ; AR2 = 0110 1110 (7th value)
```

Figure 6-12. Bit-Reversed Addressing Example

Table 6-3 shows the relationship of the index steps and the four LSBs of AR2. It can be seen that the four LSBs can be found by reversing the bit pattern of the steps.

Table 6-3. Index Steps and Bit-Reversed Addressing

STEP	BIT PATTERN	BIT-REVERSED PATTERN	BIT-REVERSED STEP
0	0000	0000	0
1	0001	1000	8
2	0010	0100	4
3	0011	1100	12
4	0100	0010	2
5	0101	1010	10
6	0110	0110	6
7	0111	1110	14

6.5 System and User Stack Management

The TMS320C30 provides a dedicated system stack pointer (SP) for building stacks in memory. The auxiliary registers can also be used to build a variety of more general linear lists. This section discusses the implementation of the following types of linear lists:

Stack A linear list for which all insertions and deletions are made at one end of the list.

Queue A linear list for which all insertions are made at one end of the list, and all deletions are made at the other end.

Deque A 'double-ended queue' linear list for which insertions and deletions are made at the either end of the list.

The system stack pointer (SP) is a 32-bit register that contains the address of the top of the system stack. The system stack fills from low-memory address to high-memory address (see Figure 6-13). The SP always points to the last element pushed onto the stack. A push performs a preincrement; and a pop, a postdecrement of the system stack pointer.

The program counter is pushed on the system stack on subroutine calls, traps, and interrupts. It is popped from the system stack on returns. The system stack can be pushed and popped using the PUSH, POP, PUSHF, and POPF instructions.

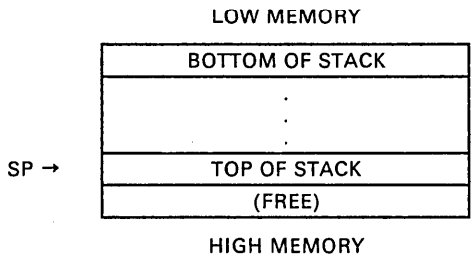


Figure 6-13. System Stack Configuration

6.5.1 Stacks

Stacks can be built from low to high memory or high to low memory. Two cases for each type of stack are shown. Stacks can be built using the preincrement/decrement and postincrement/decrement modes of modifying the auxiliary registers (AR). Stack growth from high-to-low memory can be implemented in two ways:

CASE 1: Stores to memory using *--ARn to push data on the stack and reads from memory using *ARn++ to pop data off the stack.

CASE 2: Stores to memory using *ARn-- to push data on the stack and reads from memory using * ++ARn to pop data off the stack.

Figure 6-14 illustrates these two cases. The only difference is that in using case 1, the AR always points to the top of the stack, and in case 2, the AR always points to the next free location on the stack.

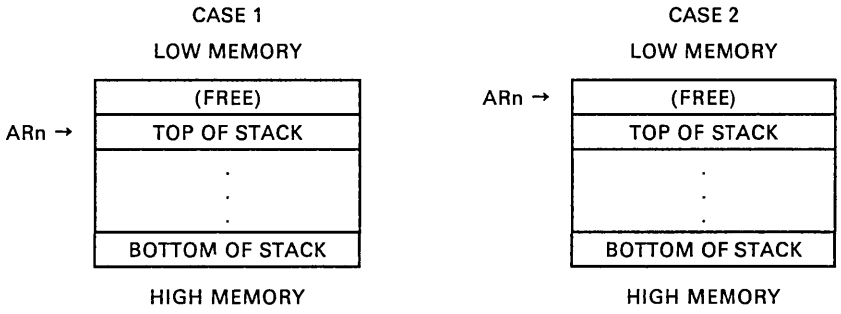


Figure 6-14. Implementions of High-to-Low Memory Stacks

Stack growth from low-to-high memory can be implemented in two ways:

CASE 3: Stores to memory using *++ARn to push data on the stack and reads from memory using *ARn-- to pop data off the stack.

CASE 4: Stores to memory using *ARn++ to push data on the stack and reads from memory using *--ARn to pop data off the stack.

Figure 6-15 shows these two cases. In the case 3, the AR always points to the top of the stack. In case 4, the AR always points to the next free location on the stack.

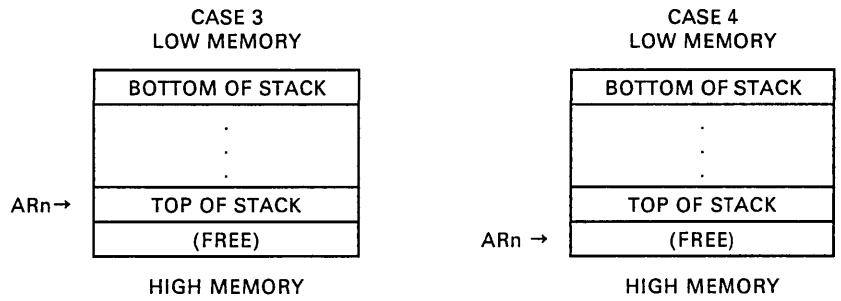


Figure 6-15. Implementions of Low-to-High Memory Stacks

6.5.2 Queues and Deques

The implementations of queues and deques is based upon the manipulation of the auxiliary registers for user stacks. For queues, two auxiliary registers are used, one to mark the front of the queue from which data is popped and the other to mark the rear of the queue where data is pushed.

For deques, two auxiliary registers are also necessary. One is used to mark one end of the deque, and the other is used to mark the other end. Data can be popped or pushed from either end.

Introduction 1

Pinout and Signal Descriptions 2

Architectural Overview 3

CPU Registers, Memory, and Cache 4

Data Formats and Floating-Point Operation 5

Addressing 6

Program Flow Control 7

External Bus Operation 8

Peripherals 9

Pipeline Operation 10

Assembly Language Instructions 11

Software Applications 12

Hardware Applications 13

Appendices A-D

Section 7

Program Flow Control

The TMS320C30 provides a complete set of flexible and powerful constructs that allow for software control of the program flow. These consist of two main types: repeat modes and branching (standard and delayed). When programming includes a combination of repeat modes, standard branches, and delayed branches, the type best suited for a particular application can be selected.

Several interlocked operations instructions provide a flexible means of multiprocessor support. Through the use of external signals, these instructions allow for powerful synchronization mechanisms. They also guarantee the integrity of the communication and result in a high-speed operation.

The TMS320C30 supports a nonmaskable external reset signal and a number of internal and external interrupts. These functions can be programmed for a particular application.

Major topics discussed in this section include:

- Repeat Modes (Section 7.1 on page 7-2)
  - Initialization
  - Operation
- Delayed Branches (Section 7.2 on page 7-7)
- Interlocked Operations (Section 7.3 on page 7-8)
- Reset Operation (Section 7.4 on page 7-12)
- Interrupts (Section 7.5 on page 7-15)

7.1 Repeat Modes

The repeat modes of the TMS320C30 allow for the implementation of zero-overhead looping. For many algorithms, there is an inner kernel of code where most of the execution time is spent. Using the repeat modes allows these time-critical sections of code to be executed in the shortest possible time.

The TMS320C30 provides two instructions to support zero-overhead looping: RPTB (repeat a block of code) and RPTS (repeat a single instruction). RPTB allows for a block of code to be repeated a specified number of times. RPTS allows a single instruction to be repeated a number of times and reduces the bus traffic by fetching the instruction only once.

Three registers (RS, RE, and RC) are associated with the updating of the program counter when updated in a repeat mode. Table 7-1 describes these registers.

Table 7-1. Repeat Mode Registers

REGISTER	FUNCTION
RS	Repeat Start Address Register. Holds the address of the first instruction of the block of code to be repeated.
RE	Repeat End Address Register. Holds the address of the last instruction of the block of code to be repeated.
RC	Repeat Count Register. Contains one less than the number of times the block remains to be repeated.

7.1.1 Repeat Mode Initialization

There are two bits that are very important to the operation of RPTB and RPTS, the RM and S bits.

The RM (repeat mode flag) bit in the status register specifies if the processor is running in the repeat mode. If RM = 0, fetches are not made in repeat mode, if RM = 1, fetches are made in repeat mode.

The S bit is hidden from the user, but is necessary to fully describe the operation of RPTB and RPTS. If S = 0, the CPU is not performing fetches in the repeat-single mode. If S = 1 and RM = 1, the CPU is performing fetches in the repeat-single mode.

The correct operation of the repeat modes requires that all of the above registers and status register fields be initialized correctly. The RPTB and RPTS instructions perform this initialization in slightly different ways (see Sections 7.1.2 and 7.1.3).

7.1.2 RPTB Initialization

When RPTB *src* is executed, the following operations take place:

1) PC + 1 → RS
2) *src* → RE
3) 1 → RM status register bit
4) 0 → S bit.

Step 1 loads the start address of the block into RS. Step 2 loads the *src* into the RE (end address of the block). The *src* operand is a 24-bit value contained in the instruction word. Step 3 sets the status register to indicate the repeat mode of operation. Step 4 indicates that this is the repeat block mode of operation.

The last bit of information required is the number of times to repeat the block. The value is determined by properly initializing the RC (repeat count) register. Since the execution of RPTB does not load the RC, this register must be loaded explicitly by the user. The typical setup of the block repeat operation is shown below.

```
LDI     15,RC  ; 15 → RC
RPTB    LOOP   ; LOOP → RE, PC + 1 → RS, 1 → RM, 0 → S
```

The repeat modes repeat a block of code at least once in a typical operation. The repeat counter should be loaded with one less than the number of times to repeat the block; i.e., a value of 0 in RC repeats the block of code one time. All block repeats initiated by RPTB can be interrupted.

7.1.3 RPTS Initialization

When RPTS *src* is executed, the following sequence of operations occurs:

1) PC + 1 → RS
2) PC + 1 → RE
3) 1 → RM status register bit
4) 1 → S bit
5) *src* → RC

The RPTS instuction loads all registers and mode bits necessary for the operation of the single instruction repeat mode. Step 1 loads the start address of the block into RS. Step 2 loads the end address into the RE (end address of the block). Since this is a repeat of a single instruction, the start address and the end address are the same. Step 3 sets the status register to indicate the repeat mode of operation. Step 4 indicates that this is the repeat single-instruction mode of operation. The operand *src* is loaded into RC.

Repeats of a single instruction initiated by RPTS are not interruptible, since the RPTS fetches the instruction word only once and then keeps it in the instruction register for reuse. An interrupt would cause the instruction word to be lost. The refetching of the instruction word from the instruction register reduces memory accesses and, in effect, acts as a one-word program cache. If it is necessary to have a single instruction that is repeatable and interruptible, the RPTB instruction may be used on this single instruction.

7.1.4 Repeat Mode Operation

The information in the repeat mode registers and associated control bits is used to control the modification of the PC when the fetches are being made in repeat mode. The repeat modes compare the contents of the RE register with the program counter (PC). If they match and the repeat counter is non-negative, the repeat counter is decremented, the PC is loaded with the repeat start address, and the processing continues. The fetches and appropriate status bits are modified as necessary. Note that the repeat counter (RC) is never modified when RM is 0. The maximum number of repeats occurs when RC = 080000000h. This will result in 080000001h repetitions. The detailed algorithm for the update of the PC is described in Figure 7-1.

```
if RM == 1                                  ; If in repeat mode (RPTB or RPTS)
        if S == 1                           ; If RPTS
                if first time through       ; If this is the first fetch
                    fetch instruction from memory   ; Fetch instruction from memory
                else                        ; If not the first fetch
                    fetch instruction from IR       ; Fetch instruction from IR

                RC - 1 → RC                 ; Decrement RC
                if RC < 0                   ; If RC is negative
                                            ; Repeat single mode completed
                    0 → ST(RM)              ; Turn off repeat mode bit
                    0 → S                   ; Clear S
                    PC + 1 → PC             ; Increment PC

        else if S == 0                      ; If RPTB
                fetch instruction from memory   ; Fetch instruction from memory
                if PC == RE                 ; If this is the end of the block
                    RC - 1 → RC             ; Decrement RC
                    if RC ≥ 0               ; If RC is not negative
                        RS → PC             ; Set PC to start of block
                    else if RC < 0          ; If RC is negative
                        0 → ST(RM)          ; Turn off repeat mode bits
                        0 → S               ; Clear S
                        PC + 1 → PC         ; Increment PC
```

Figure 7-1. Repeat Mode Control Algorithm

The RPTB and RPTS are four-cycle instructions. These four cycles of overhead are only incurred on the first pass through the loop. All subsequent passes through the loop are accomplished with zero cycles of overhead. In Example 7-1, the block of code from STLOOP to ENDLOP is repeated sixteen times.

Example 7-1. RPTB Operation

```
            LD      15,RC   ; Load repeat counter with 15
            RPTB    ENDLOP  ; Execute the block of code
STLOOP                      ; from STLOOP to ENDLOP 16 times
             .
             .
             .
ENDLOP
```

Using this mode of modifying the PC allows for a straightforward analysis of what would happen in the case of branches within the block. It is best to look at the operation from the point of view that the next value of the PC will be either PC + 1 or the contents of the RS register. It is thus apparent that this method of block repeat allows for any amount of branching within the repeated block. Execution can go anywhere within the user's code via interrupts, subroutine calls, etc. For proper modification of the loop counter, the last instruction of the loop must be fetched. The repeating of the loop can be stopped prior to completion by writing a 0 into the repeat counter or writing 0 into the RM bit of the status register.

Since the block repeat modes modify the program counter, other instructions cannot modify the program counter at the same time. Two rules apply here:

1) The last instruction in the block (or the only instruction in a block of size one) cannot be a B*cond*, BR, DB*cond*, CALL, CALL*cond*, TRAP*cond*, RETI*cond*, RETS*cond*, IDLE, RPTB, or RPTS. Example 7-2 shows an incorrectly placed standard branch.

2) None of the last four instructions from the bottom of the block (or the only instruction in a block of size one) can be a B*cond*D, BRD, or DB*cond*D. Example 7-3 shows an incorrectly placed delayed branch.

If either of these rules are violated, the PC will be undefined.

Example 7-2. Incorrectly Placed Standard Branch

```
          LD     15,RC    ; Load repeat counter with 15
          RPTB   ENDLOP   ; Execute block of code
STLOOP                    ; from STLOOP to ENDLOP 16 times
           .
          JCS
           .
ENDLOP    BR     OOPS     ; This branch violates rule 1
```

Example 7-3. Incorrectly Placed Delayed Branch

```
          LD     15,RC    ; Load repeat counter with 15
          RPTB   ENDLOP   ; Execute block of code
STLOOP                    ; from STLOOP to ENDLOP 16 times
           .
          GAF
           .
          BRD    OOPS     ; This branch violates rule 2
          ADDF
          MPYF
ENDLOP    SUBF
```

Block repeats (RPTB) are nestable. Since all of the control is defined by the RS, RE, RC, and ST registers, saving and restoring these registers allows for their nesting. The RM bit in the status register can be used to determine if the block repeat mode is active. For example, if an interrupt service routine is written which requires the use of RPTB, it is possible that the interrupt associated with the routine may occur during another block repeat. The interrupt service routine can check the RM bit. If this bit is set, the interrupt routine saves RS, RE, RC, and ST. The interrupt routine can then perform a block repeat. Before returning to the interrupted routine, the interrupt routine restores

RS, RE, RC, and ST. If the RM bit is not set, the save and restore of these registers is not necessary.

7.2 Delayed Branches

The branching capabilities of the TMS320C30 include two main types: standard and delayed branches. Standard branches empty the pipeline before performing the branch to guarantee correct management of the program counter. This results in a TMS320C30 branch taking four cycles. Included in this class are calls, returns, and traps.

Delayed branches on the TMS320C30 do not empty the pipeline, but rather guarantee that the next three instructions will be executed before the program counter is modified by the branch. The result is a branch that only requires a single cycle, thus making the speed of the delayed branch very close to the optimal block repeat modes of the TMS320C30. However, unlike block repeat modes, delayed branches may be used in situations other than looping. Every delayed branch has a standard branch counterpart that is used when a delayed branch cannot be used. The delayed branches of the TMS320C30 are B*cond*D, BRD, and DB*cond*D.

Conditional delayed branches use the conditions that exist at the end of the instruction immediately preceding the delayed branch. They do not depend upon the instructions following the delayed branch. Delayed branches are guaranteed to allow the three following instructions to be executed regardless of other pipeline conflicts.

When a delayed branch is fetched, it remains pending until the three following instructions are executed. None of the three instructions that follow a delayed branch can be B*cond*, B*cond*D, BR, BRD, DB*cond*, DB*cond*D, CALL, CALL*cond*, TRAP*cond*, RETI*cond*, RETS*cond*, RPTB, RPTS, or IDLE. (see Example 7-4).

Delayed branches disable interrupts until the three instructions following the delayed branch are completed. This is independent of whether or not the branch is taken.

If delayed branches are used incorrectly, the PC will be undefined.

Example 7-4. Incorrectly Placed Delayed Branches

```
B1:  BD    L1
     NOP
     NOP
B2:  B     L2          ; This branch is incorrectly placed
     NOP
     MJH
     NOP
       .
       .
       .
```

7.3 Interlocked Operations

One of the most common multiprocessing configurations is the sharing of global memory by multiple processors. In order to allow multiple processors to access this global memory and share data in a coherent manner, some sort of arbitration or handshaking is necessary. This requirement for arbitration is the purpose of the TMS320C30 interlocked operations.

The TMS320C30 provides a flexible means of multiprocessor support with five instructions, referred to as interlocked operations. Through the use of external signals, these instructions allow for powerful synchronization mechanisms. They also guarantee the integrity of the communication and result in a high-speed operation. The interlocked-operation instruction group is listed in Table 7-2.

Table 7-2. Interlocked Operations

MNEMONIC	DESCRIPTION	OPERATION
LDFI	Load floating-point value into a register, interlocked	Signal interlocked src → dst
LDII	Load integer into a register, interlocked	Signal interlocked src → dst
SIGI	Signal, interlocked	Signal interlocked Clear interlock
STFI	Store floating-point value to memory, interlocked	src → dst Clear interlock
STII	Store integer to memory, interlocked	src → dst Clear interlock

The interlocked operations use the two external flag pins, XF0 and XF1. XF0 must be configured as an output pin, and XF1 as an input pin. When configured in this manner, XF0 signals an interlock operation request, and XF1 acts as an acknowledge signal for the requested interlocked operation. In this mode, XF0 and XF1 are treated as active-low signals.

The external timing for the interlocked loads and stores are the same as standard load and stores. The interlocked loads and stores may be extended like standard accesses, by using the appropriate ready signal ($\overline{\text{RDY}}$ or $\overline{\text{XRDY}}$).

The LDFI and LDII instructions perform the following actions:

1) Simultaneously set XF0 to 0 and begin a read cycle. The timing of XF0 is similar to that of the address bus during a read cycle.
2) Execute an LDF or LDI instruction and extend the read cycle until XF1 is set to 0 and a ready ($\overline{\text{RDY}}$ or $\overline{\text{XRDY}}$) is signalled.
3) Leave XF0 set to 0 and end the read cycle.

The read/write operation is identical to any other read/write cycle except for the special use of XF0 and XF1. The *src* operand for LDFI and LDII is always a direct or indirect memory address. XF0 is set to 0 only if the *src* is located off-chip; i.e., ($\overline{\text{STRB}}$, $\overline{\text{MSTRB}}$ or $\overline{\text{IOSTRB}}$ is active), or the *src* is one of the on-chip peripherals. If on-chip memory is accessed, then XF0 is not asserted, and the operation is as an LDF or LDI from internal memory.

The STFI and STII instructions perform the following operations:

1) Simultaneously set XF0 to 1 and begin a write cycle. The timing of XF0 is similar to that of the address bus during a write cycle.
2) Execute an STF or STI instruction and extend the write cycle until a ready ($\overline{RDY}$ or $\overline{XRDY}$) is signalled.

As in the case for LDFI and LDII, the *dst* of STFI and STII affects XF0 if *dst* is located off-chip ($\overline{STRB}$, $\overline{MSTRB}$, or $\overline{IOSTRB}$ is active), or the *src* is one of the on-chip peripherals. If on-chip memory is accessed, then XF0 is not asserted and the operations are as an STF or STI to internal memory.

The SIGI instruction functions as follows:

1) Sets XF0 to 0.
2) Idles until XF1 is set to 0.
3) Sets XF0 to 1 and ends the operation.

While the LDFI, LDII, and SIGI instructions are waiting for XF1 to be set to 0, they may be interrupted. LDFI and LDII require a ready signal in order to be interrupted. This allows the user to implement protection mechanisms against deadlock conditions by interrupting an interlocked load that has taken too long. Upon return from the interrupt, the next instruction is executed. The STFI and STII instructions are not interruptible.

Interlocked operations can be used to implement a busy-waiting loop, to manipulate a multiprocessor counter, to implement a simple semaphore mechanism, or to perform synchronization between two TMS320C30s. The following examples illustrate the usefulness of the interlocked operations instructions. Example 7-5 shows the implementation of a busy-waiting loop. If location LOCK is the interlock for a critical section of code, and a nonzero means the lock is busy, the algorithm for a busy-waiting loop can be used as shown in Example 7-5.

Example 7-5. Busy-Waiting Loop

```
     LDI:   1,R0            ; Put 1 in R0
L1:  LDII   @LOCK,R1        ; Interlocked operation begun
                            ; Contents of LOCK → R1
     STII   R0,@LOCK        ; Put R0 (= 1) into LOCK, XF0 = 1
                            ; Interlocked operation ended
     BNZ    L1              ; Keep trying until LOCK = 0
```

Example 7-6 shows how a location COUNT may contain a count of the number of times a particular operation needs to be performed. This operation may be performed by any processor in the system. If the count is zero, the processor waits until it is nonzero before beginning processing. The algorithm for modifying COUNT correctly is shown in Example 7-6.

Example 7-6. Multiprocessor Counter Manipulation

```
CT:  OR     4,IOF          ; XF0 = 1
                           ; Interlocked operation ended
     LDII   @COUNT,R1      ; Interlocked operation begun
                           ; Contents of COUNT → R1
     BZ     CT             ; If COUNT = 0, keep trying
     SUBI   1,R1           ; Decrement R1(= COUNT)
     STII   R1,@COUNT      ; Update COUNT, XF0 = 1
                           ; Interlocked operation ended
```

Figure 7-2 illustrates multiple TMS320C30s sharing global memory and using the interlocked instructions as in Examples 7-7, 7-8, 7-9, and 7-10.

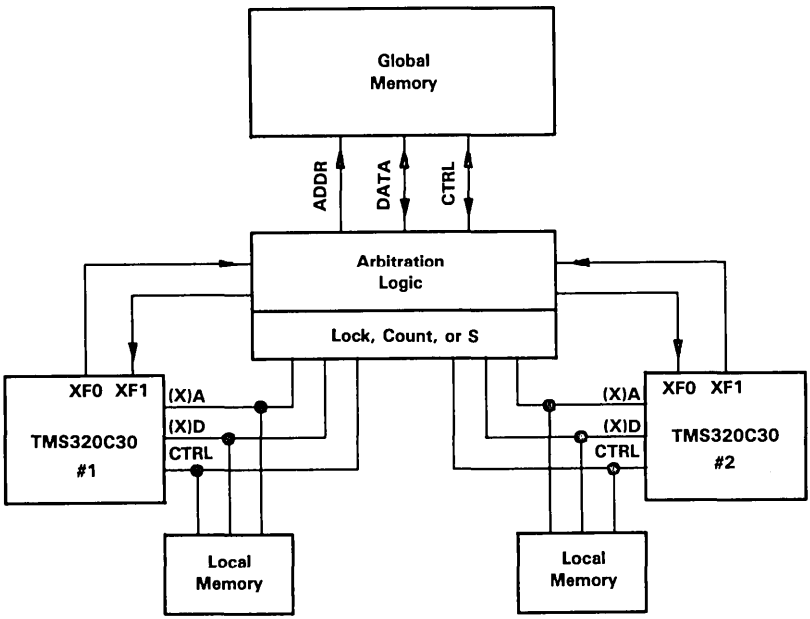


Figure 7-2. Multiple TMS320C30s Sharing Global Memory

Sometimes it may be necessary for several processors to access some shared data or other common resources. The portion of code which must access the shared data is called a critical section.

To ease the programming of critical sections, semaphores may be used. Semaphores are variables which can only take non-negative integer values. Two primitive, indivisible, operations are defined on semaphores, namely (with S being a semaphore):

```
V(S):       S + 1 → S
P(S):   P:  if (S == 0), go to P
            else S - 1 → S
```

Indivisibility of V(S) and P(S) means that when these processes access and modify the semaphore S, they are the only processes accessing and modifying S.

To enter a critical section, a P operation is performed on a common semaphore, say S (S is initialized to 1). The first processor performing P(S) will be able to enter its critical section. All other processors are blocked since S has become 0. After leaving its critical section, the processor performs a V(S), thus allowing another processor to execute P(S) successfully.

The TMS320C30 code for V(S) is shown in Example 7-7, and code for P(S) is shown in Example 7-8. Compare the code in Example 7-8 to the code in Example 7-6.

Example 7-7. Implementation of V(S)

```
V:   LDII   @S,R0    ; Interlocked read of S begins (XF0 = 0)
                     ; Contents of S → R0
     ADDI   1,R0     ; Increment R0 (= S)
     STII   R0,@S    ; Update S, end interlock (XF0 = 0)
```

Example 7-8. Implementation of P(S)

```
P:   OR     4,IOF    ; End interlock (XF0 = 1)
     LDII   @S,R0    ; Interlocked read of S begins
                     ; Contents of S → R0
     BZ     P        ; If S = 0, go to P and try again
     SUBI   1,R0     ; Decrement R0 (= S)
     STII   R0,@S    ; Update S, end interlock (XF0 = 1)
```

The SIGI operation may be used to synchronize, at an instruction level, multiple TMS320C30s. Consider two processors connected as shown in Figure 7-3 The code for the two processors is shown in Example 7-9

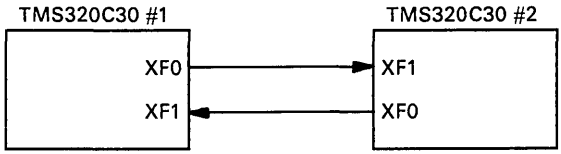
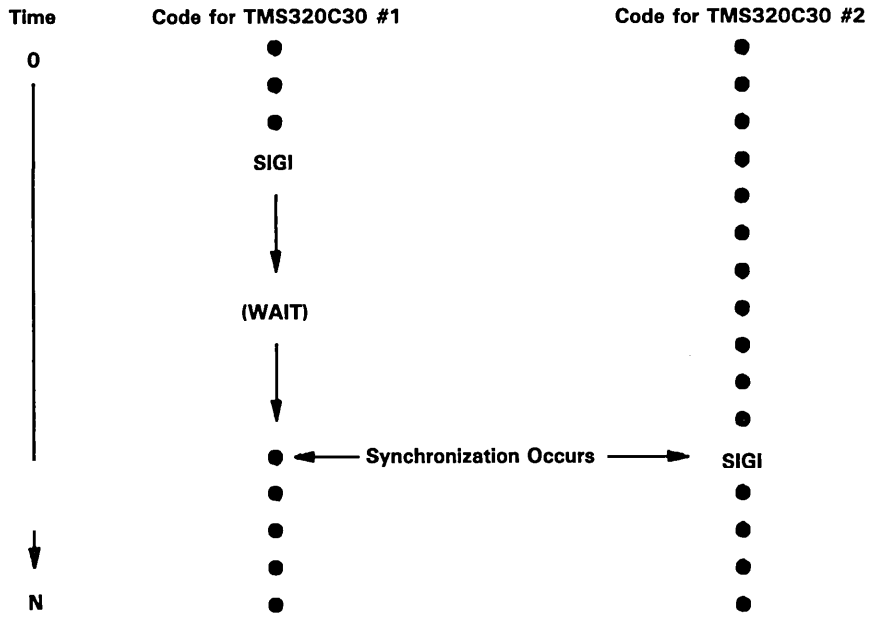


Figure 7-3. Zero-Logic Interconnect of TMS320C30s

Processor #1 runs until it executes the SIGI. It then waits until processor #2 executes a SIGI. At this point, the two processors have synchronized and continue execution.

Example 7-9. Code to Synchronize Two TMS320C30s at the Software Level



7.4 Reset Operation

The TMS320C30 supports a nonmaskable external reset signal ($\overline{\text{RESET}}$), which is used to perform system reset. This section discusses the reset operation.

At powerup, the state of the TMS320C30 processor is undefined. The $\overline{\text{RESET}}$ signal is used to place the processor in a known state. This signal must be asserted low for 10 or more H1 clock cycles to guarantee a system reset. H1 is an output clock signal generated by the TMS320C30 (see Appendix A for more information).

Reset affects the other pins on the device in either a synchronous or asynchronous manner. The synchronous reset is gated by the TMS320C30s internal clocks. The asynchronous reset directly affects the pins, and is faster than the synchronous reset. Table 7-3 shows the state of the TMS320C30s pins after $\overline{\text{RESET}}$ = 0. Each pin is described according to whether the pin is reset synchronously or asynchronously.

Table 7-3. Pin Operation at Reset

SIGNAL	# PINS	OPERATION AT RESET
PRIMARY INTERFACE (61 PINS)		
D(31-0)	32	Synchronous reset. Placed in high-impedance state.
A(23-0)	24	Synchronous reset. Placed in high-impedance state.
R/$\overline{\text{W}}$	1	Synchronous reset. Deasserted by going to a high level.
$\overline{\text{STRB}}$	1	Synchronous reset. Deasserted by going to a high level.
$\overline{\text{RDY}}$	1	Reset has no effect.
$\overline{\text{HOLD}}$	1	Reset has no effect.
$\overline{\text{HOLDA}}$	1	Reset has no effect.
EXPANSION INTERFACE (49 PINS)		
XD(31-0)	32	Synchronous reset. Placed in high-impedance state.
XA(12-0)	13	Synchronous reset. Placed in high-impedance state.
XR/$\overline{\text{W}}$	1	Synchronous reset. Deasserted by going to a high level.
$\overline{\text{MSTRB}}$	1	Synchronous reset. Deasserted by going to a high level.
$\overline{\text{IOSTRB}}$	1	Synchronous reset. Deasserted by going to a high level.
$\overline{\text{XRDY}}$	1	Reset has no effect.
CONTROL SIGNALS (9 PINS)		
$\overline{\text{RESET}}$	1	Reset input pin
$\overline{\text{INT}}$(3-0)	4	Reset has no effect.
$\overline{\text{IACK}}$	1	Synchronous reset. Deasserted by going to a high level.
MC/$\overline{\text{MP}}$	1	Reset has no effect.
XF(1-0)	2	Asynchronous reset. Placed in high-impedance state.

Table 7-3. Pin Operation at Reset (Continued)

SIGNAL	# PINS	OPERATION AT RESET
SERIAL PORT 0 SIGNALS (6 PINS)		
CLKX0	1	Asynchronous reset. Placed in high-impedance state.
DX0	1	Asynchronous reset. Placed in high-impedance state.
FSX0	1	Asynchronous reset. Placed in high-impedance state.
CLKR0	1	Asynchronous reset. Placed in high-impedance state.
DR0	1	Asynchronous reset. Placed in high-impedance state.
FSR0	1	Asynchronous reset. Placed in high-impedance state.
SERIAL PORT 1 SIGNALS (6 PINS)		
CLKX1	1	Asynchronous reset. Placed in high-impedance state.
DX1	1	Asynchronous reset. Placed in high-impedance state.
FSX1	1	Asynchronous reset. Placed in high-impedance state.
CLKR1	1	Asynchronous reset. Placed in high-impedance state.
DR1	1	Asynchronous reset. Placed in high-impedance state.
FSR1	1	Asynchronous reset. Placed in high-impedance state.
TIMER 0 SIGNAL (1 PIN)		
TCLK0	1	Asynchronous reset. Placed in high-impedance state.
TIMER 1 SIGNAL (1 PIN)		
TCLK1	1	Asynchronous reset. Placed in high-impedance state.
SUPPLY and OSCILLATOR SIGNALS (29 PINS)		
V_{DD}(3-0)	4	Reset has no effect.
IODV_{DD}(1,0)	2	Reset has no effect.
ADV_{DD}(1,0)	2	Reset has no effect.
PDV_{DD}	1	Reset has no effect.
DDV_{DD}(1,0)	2	Reset has no effect.
MDV_{DD}	1	Reset has no effect.
V_{SS}(3-0)	4	Reset has no effect.
DV_{SS}(3-0)	4	Reset has no effect.
CV_{SS}(1,0)	2	Reset has no effect.
IV_{SS}	1	Reset has no effect.
V_{BBP}	1	Reset has no effect.
SUBS	1	Reset has no effect.
X1	1	Reset has no effect.
X2/CLKIN	1	Reset has no effect.
H1	1	Synchronous reset. Will go to its initial state when $\overline{\text{RESET}}$ makes a 1 to 0 transition. See Appendix A.
H3	1	Synchronous reset. Will go to its initial state when $\overline{\text{RESET}}$ makes a 1 to 0 transition. See Appendix A.

Table 7-3. Pin Operation at Reset (Concluded)

SIGNAL	# PINS	OPERATION AT RESET
EMULATION, TEST, and RESERVED (18 PINS)		
EMU0	F14	Undefined.
EMU1	E15	Undefined.
EMU2	F13	Undefined.
EMU3	E14	Undefined.
EMU4	F12	Undefined.
EMU5	C1	Undefined.
EMU6	M6	Undefined.
RSV0	J3	Undefined.
RSV1	J4	Undefined.
RSV2	K1	Undefined.
RSV3	K2	Undefined.
RSV4	L1	Undefined.
RSV5	K3	Undefined.
RSV6	L2	Undefined.
RSV7	K4	Undefined.
RSV8	M1	Undefined.
RSV9	L3	Undefined.
RSV10	M2	Undefined.

At system reset, the following additional operations are performed:

- The peripherals are reset. This is a synchronous operation. The peripheral reset is described in Section 9.
- The following CPU registers are loaded with zero:
  - ST (CPU status register)
  - IE (CPU/DMA interrupt enable flags)
  - IF (CPU interrupt flags)
  - IOF (I/O flags)
- The reset vector is read from memory location 0h and loaded into the PC. This vector contains the start address of the system reset routine
- Execution begins. Refer to Section 12 an example of a processor initialization routine.

Multiple TMS320C30s driven by the same system clock may be reset and synchronized. When the 1 to 0 transition of $\overline{\text{RESET}}$ occurs, the processor is placed on a well-defined internal phase, and all of the TMS320C30s will come up on the same internal phase.

7.5 Interrupts

The TMS320C30 supports multiple internal and external interrupts, which can be used for a variety of applications. This section discusses the operation of these interrupts.

A functional diagram of the logic used to implement the external interrupt inputs is shown in Figure 7-4; the logic for internal interrupts is similar. Additional information regarding internal interrupts can be found in Section 9.

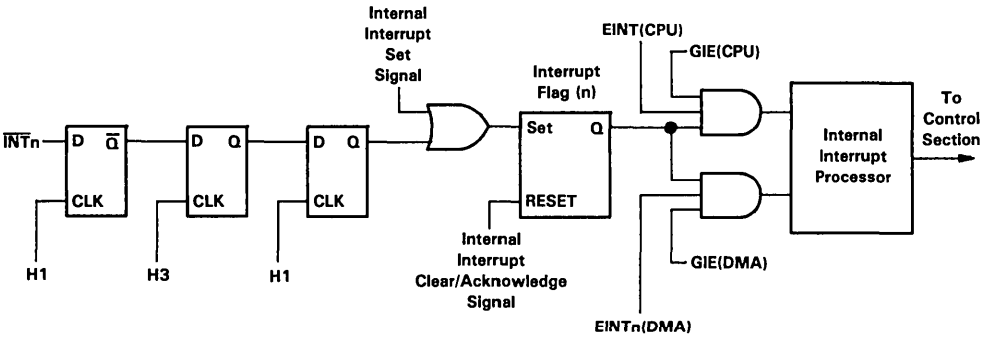


Figure 7-4. Interrupt Logic Functional Diagram

External interrupts are synchronized internally as illustrated by the three flip-flops clocked by H1 and H3. Once synchronized, the interrupt input will set the corresponding Interrupt Flag register (IF) bit if the interrupt is active.

External interrupts can be effectively either edge- or level-triggered, depending on the duration of the low level on the interrupt input. An external interrupt must be held low for at least one H1/H3 cycle to be recognized by the TMS320C30. If the interrupt is held low for between one and three cycles, then only one interrupt is recognized. If the interrupt is held low three or more cycles, more than one interrupt may be recognized depending on how rapidly interrupts are serviced.

When a particular interrupt is processed by the CPU or DMA controller, the corresponding interrupt flag bit is cleared by the internal interrupt acknowledge signal. It should be noted, however, that if $\overline{INTn}$ is still low when the interrupt acknowledge signal occurs, the interrupt flag bit will only be cleared for one cycle and then set again since $\overline{INTn}$ is still low. Accordingly, it is theoretically possible that, depending on when the IF register is read, this bit may be zero even though $\overline{INTn}$ is zero. When the TMS320C30 is reset, zero is written to the interrupt flag register, thereby clearing all pending interrupts.

The interrupt flag register bits may be read and written under software control. Writing a 1 to a IF register bit sets the associated interrupt flag to 1. Similiarly, writing a 0 resets the corresponding interrupt flag to 0. In this way, all interrupts may be triggered and/or cleared through software. Since the interrupts flags may be read, the interrupt pins may be polled in software when an interrupt-driven interface is not required.

Internal interrupts operate in a similar manner. The bit in the IF register corresponding to an internal interrupt may be read and written through software. Writing a 1 sets the interrupt latch, and writing a 0 clears it. All internal interrupts are one H1/H3 cycle in length.

The CPU global interrupt enable bit (GIE), located in the CPU status register (ST), controls all CPU interrupts. All DMA interrupts are controlled by the DMA global interrupt enable bit, which is not dependent upon ST(GIE) and is local to the DMA. The DMA global interrupt enable bit is dependent, in part, upon the state of the DMA SYNCH bits. It is not directly accessible through software (see Section 9). The AND of the interrupt flag bit and the interrupt enables is then connected to the interrupt processor.

To provide for maximum performance in servicing interrupts, the interrupt acknowledge (IACK) instruction is provided. IACK drives the $\overline{\text{IACK}}$ pin and performs a dummy read. The read is peformed from the address specified by the IACK instruction operand. When IACK is used, it typically is placed in the early portion of an interrupt service routine. For certain applications, it may be better suited at the end of the interrupt service routine or be totally unnecessary.

The CPU controls all prioritization of interrupts (see Table 7-4 for reset and interrupt vector locations and priorities). If the DMA is not using interrupts for synchronization of transfers, it will not be affected by the processing of the CPU interrupts. If the CPU is involved in a pipeline conflict (branch, register, or memory), it will not respond to the interrupts until that conflict is resolved. It is therefore possible to interrupt the CPU and DMA simultaneously with the same or different interrupts and, in effect, synchronize their activities. For example, it may be necessary to cause a high-priority DMA transfer that avoids bus conflicts with the CPU, i.e., make the DMA higher priority than the CPU. This may be accomplished using an interrupt that causes the CPU to trap to an interrupt routine that contains an IDLE instruction. Then if the same interrupt is used to synchronize DMA transfers, the DMA transfer counter can be used to generate an interrupt, and thus return control to the CPU following the DMA transfer.

Since the DMA and CPU share the same set of interrupt flags, the DMA may clear an interrupt flag before the CPU can respond to it. For example, if the CPU interrupts are disabled, the DMA can be responding to interrupts and thus clearing the associated interrupt flags.

Table 7-4. Reset and Interrupt Vector Locations

RESET OR INTERRUPT	VECTOR LOCATION	PRIORITY	FUNCTION
$\overline{\text{RESET}}$	0h	0	External reset signal input on the $\overline{\text{RESET}}$ pin.
$\overline{\text{INT0}}$	1h	1	External interrupt input on the $\overline{\text{INT0}}$ pin.
$\overline{\text{INT1}}$	2h	2	External interrupt input on the $\overline{\text{INT1}}$ pin.
$\overline{\text{INT2}}$	3h	3	External interrupt input on the $\overline{\text{INT2}}$ pin.
$\overline{\text{INT3}}$	4h	4	External interrupt input on the $\overline{\text{INT3}}$ pin.
XINT0	5h	5	Internal interrupt generated when serial port 0 transmit buffer is empty.
RINT0	6h	6	Internal interrupt generated when serial port 0 receive buffer is full.
XINT1	7h	7	Internal interrupt generated when serial port 1 transmit buffer is empty.
RINT1	8h	8	Internal interrupt generated when serial port 1 receive buffer is full.
TINT0	9h	9	Internal interrupt generated by timer 0.
TINT1	0Ah	10	Internal interrupt generated by timer 1.
DINT	0Bh	11	Internal interrupt generated by DMA controller 0.

If there is a delayed branch in the pipeline, interrupts are held pending until after the branch. If the interrupt occurs in the first cycle of the fetch of an instruction, the fetched instruction is discarded (not executed), and the address of that instruction is pushed to the top of the system stack. If the interrupt occurs after the first cycle of the fetch, in the case of a multicycle fetch due to wait states, that instruction is executed and the address of the next instruction to be fetched is pushed to the top of the system stack. If no program fetch is occurring, then no new fetch is performed. After the address of the appropriate instruction has been pushed, the interrupt vector is fetched, loaded into the PC, and execution continues.

The TMS320C30 allows the CPU and DMA to respond to and process interrupts in parallel. Figure 7-5 shows interrupt processing flow. The interrupts are polled and the CPU and DMA begin processing them. In the interrupt flow pertaining to the the CPU, the interrupt flag corresponding to the highest-priority enabled interrupt is cleared, and GIE is set to 0. The CPU completes all fetched instructions. The interrupt vector is fetched and loaded into the PC, and the CPU continues execution. The DMA cycle is similar to that for the CPU. After the pertinent interrupt flag is cleared, the DMA proceeds based upon the status of the SYNCH bits in the DMA global control register.

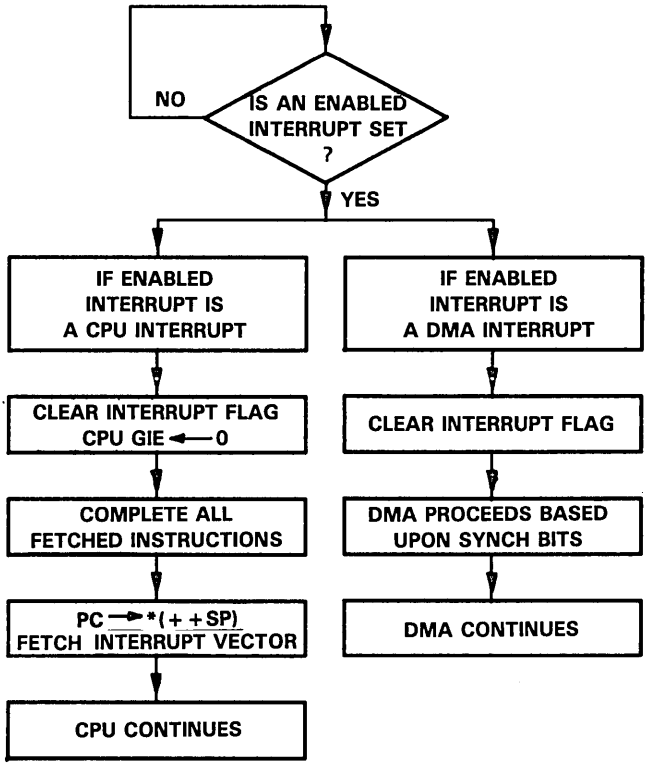


Figure 7-5. Interrupt Processing

Introduction 1

Pinout and Signal Descriptions 2

Architectural Overview 3

CPU Registers, Memory, and Cache 4

Data Formats and Floating-Point Operation 5

Addressing 6

Program Flow Control 7

External Bus Operation 8

Peripherals 9

Pipeline Operation 10

Assembly Language Instructions 11

Software Applications 12

Hardware Applications 13

Appendices A-D

Section 8

External Bus Operation

Two external interfaces are provided on the TMS320C30: the primary bus and the expansion bus. These are used to access memories and external peripheral devices. Software controlled wait states and an external input signal provide for wait state generation.

Major topics discussed in this hardware interface section are listed below.

- External Interface Control Registers (Section 8.1 on page 8-2)
  - Primary bus
  - Expansion bus
- External Interface Timing (Section 8.2 on page 8-5)
- Programmable Wait States (Section 8.3 on page 8-18)
- Programmable Bank Switching (Section 8.4 on page 8-20)

8.1 External Interface Control Registers

The TMS320C30 provides two external interfaces: the primary bus and the expansion bus. The primary bus consists of a 32-bit data bus, a 24-bit address bus, and a set of control signals. The expansion bus consists of a 32-bit data bus, a 13-bit address bus and a set of control signals. Both buses support software-controlled wait states and an external ready input signal. Both buses support data, program, and I/O accesses.

When a primary bus access is performed, $\overline{\text{STRB}}$ is low. The expansion bus supports two types of accesses. One is used primarily for memory accesses that are signalled by $\overline{\text{MSTRB}}$ low. The timing for a $\overline{\text{MSTRB}}$ access is the same as that of the $\overline{\text{STRB}}$ access on the parallel interface. The other type of expansion bus access is commonly used for access of external peripheral devices and is signalled by $\overline{\text{IOSTRB}}$ low.

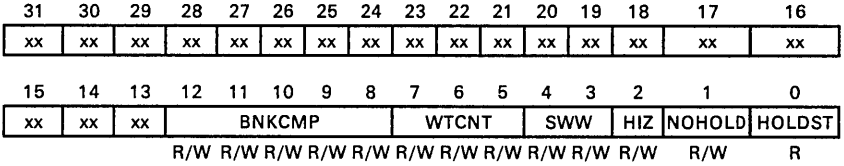
The primary bus and the expansion bus each have an associated control register. These registers are memory-mapped as shown in Figure 8-1.

REGISTER	PERIPHERAL ADDRESS
EXPANSION BUS CONTROL	808060h
RESERVED	808061h
RESERVED	808062h
RESERVED	808063h
PRIMARY BUS CONTROL	808064h
RESERVED	808065h
RESERVED	808066h
RESERVED	808067h
RESERVED	808068h
RESERVED	808069h
RESERVED	80806Ah
RESERVED	80806Bh
RESERVED	80806Ch
RESERVED	80806Dh
RESERVED	80806Eh
RESERVED	80806Fh

Figure 8-1. Memory-Mapped External Interface Control Registers

8.1.1 Primary Bus Control Register

The primary bus control register is a 32-bit register that contains the control bits for the primary bus (see Figure 8-2). Table 8-1 lists the register bits with the bit names and functions.



NOTE: xx = *reserved bit, read as 0.*
R = read, W = write.

Figure 8-2. Primary Bus Control Register

Table 8-1. Primary Bus Control Register Bits Summary

BIT	NAME	FUNCTION
0	HOLDST	Hold status bit. This bit signals if the port is being held (HOLDST = 1) or is not being held (HOLDST = 0). This status bit is valid whether the port has been held via hardware or software.
1	NOHOLD	Port hold signal. NOHOLD allows or disallows the port to be held by an external $\overline{\text{HOLD}}$ signal. When NOHOLD = 1, the TMS320C30 takes over the external bus and controls it regardless of requests by external devices. No hold acknowledge ($\overline{\text{HOLDA}}$) is asserted when a $\overline{\text{HOLD}}$ is received. However, it is asserted if an internal hold is generated (HIZ = 1). NOHOLD is set to 0 at reset.
2	HIZ	Internal hold. When set (HIZ = 1), the port is put in hold mode. This equivalent to the external $\overline{\text{HOLD}}$ signal. By forcing a three-state condition, the TMS320C30 can relinquish the external memory port through software. $\overline{\text{HOLDA}}$ goes low when the port is three-stated. HIZ is set to 0 at reset.
3-4	SWW	Software wait-state generation. In conjunction with WTCNT, this 2-bit field defines the mode of wait-state generation. It is set to 1 1 at reset.
5-7	WTCNT	Software wait mode. This 3-bit field specifies the number of cycles to use when in software wait mode for the generation of internal wait states. The range is zero (WTCNT = 0 0 0) to seven (WTCNT = 1 1 1) H1/H3 cycles. It is set to 1 1 1 at reset.
8-12	BNKCMP	Bank compare. This 5-bit field specifies the number of MSBs of the address to be used to define the bank size. It is set to 1 0 0 0 0 at reset.
13-31	Reserved	Read as 0.

8.1.2 Expansion Bus Control Register

The expansion bus control register is a 32-bit register that contains control bits for the expansion bus (see Figure 8-3 and Table 8-2).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	xx	xx	xx	xx	xx	xx	WTCNT			SWW		xx	xx	xx
								R/W	R/W	R/W	R/W	R/W			

NOTE: xx = reserved bit, read as 0.
R = read, W = write.

Figure 8-3. Expansion Bus Control Register

Table 8-2. Expansion Bus Control Register Bits Summary

BIT	NAME	FUNCTION
0-2	Reserved	Read as 0.
3-4	SWW	Software wait-state generation. In conjunction with the WTCNT, this 2-bit field defines the mode of wait-state generation. It is set to 1 1 at reset.
5-7	WTCNT	Software wait mode. This 3-bit field specifies the number of cycles to use when in software wait mode for the generation of internal wait states. The range is zero (WTCNT = 0 0 0) to seven (WTCNT = 1 1 1) H1/H3 clock cycles. It is set to 1 1 1 at reset.
8-31	Reserved	Read as 0.

8.2 External Interface Timing

This section discusses functional timing of operations on the primary bus and the expansion bus, the TMS320C30's two independent parallel buses. Detailed timing specifications for all TMS320C30 signals are contained in Appendix A, the TMS320C30 Data Sheet.

The parallel buses implement three mutually exclusive address spaces distinguished through the use of three separate control signals: $\overline{\text{STRB}}$,$\overline{\text{MSTRB}}$, and $\overline{\text{IOSTRB}}$. The $\overline{\text{STRB}}$ signal controls accesses on the primary bus, and the $\overline{\text{MSTRB}}$ and $\overline{\text{IOSTRB}}$ control accesses on the expansion bus. Since the two buses are independent, two accesses may be made in parallel.

With the exception of bank switching and the external HOLD function (discussed later in this section), timing of primary bus cycles and $\overline{\text{MSTRB}}$ expansion bus cycles are identical, and will be discussed collectively. The acronym $\overline{\text{(M)STRB}}$ will be used in references which pertain equally to $\overline{\text{STRB}}$ and $\overline{\text{MSTRB}}$. Similarly (X)R/$\overline{\text{W}}$, (X)A, (X)D, and (X)$\overline{\text{RDY}}$ are used to symbolize the equivalent primary and expansion bus signals. The $\overline{\text{IOSTRB}}$ expansion bus cycles are timed differently and will be discussed independently.

8.2.1 Primary Bus Cycles

All bus cycles comprise integral numbers of H1 clock cycles. One H1 cycle is defined to be from one falling edge of H1 to the next falling edge of H1. For full speed (zero wait state) accesses, reads take one H1 cycle, while writes take two cycles, unless the write follows a read, in which case the write takes three cycles. Recall that internally (from the perspective of the CPU and DMA) writes require only one cycle if no accesses to that interface are in progress. The following discussions pertain to zero wait state accesses unless otherwise specified.

The $\overline{\text{(M)STRB}}$ signal is low for the active portion of both reads and writes, which lasts one H1 cycle. Additionally before and after the active portion ($\overline{\text{(M)STRB}}$ low) of writes only, there is a transition cycle of H1. During this transition cycle, the following occur:

1) $\overline{\text{(M)STRB}}$ is high.
2) If required, (X)R/$\overline{\text{W}}$ changes state on H1 rising.
3) If required, addresses changes on H1 rising if the previous H1 cycle was the active portion of a write. If the previous H1 cycle was a read, address changes on H1 falling.

Figure 8-4 illustrates a read-read-write sequence for $\overline{\text{(M)STRB}}$ active and no wait states. The data is read as late in the cycle as possible to allow for the maximum access time from address valid. Note that although external writes take two cycles, internally (from the perspective of the CPU and DMA), they require only one cycle if no accesses to that interface are in progress. In the typical timing for all external interfaces, the (X)R/$\overline{\text{W}}$ strobe does not change until $\overline{\text{(M)STRB}}$ or $\overline{\text{IOSTRB}}$ goes inactive.

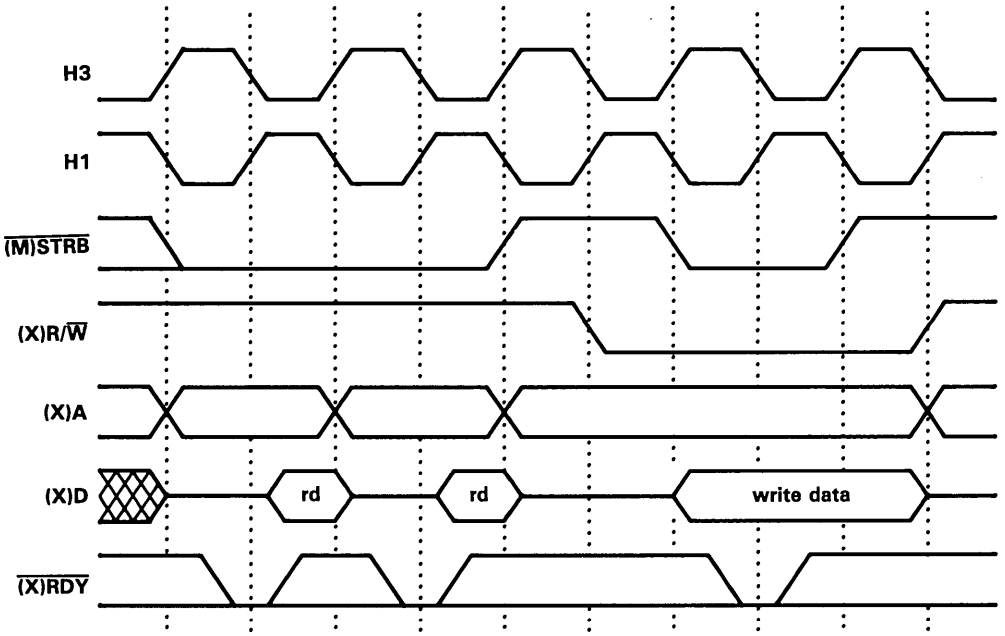


Figure 8-4. Read-Read-Write for (M)STRB = 0

Figure 8-5 Illustrates a write-write-read sequence for $\overline{(M)STRB}$ active and no wait states. The address and data written are held valid approximately one-half cycle after $\overline{(M)STRB}$ changes.

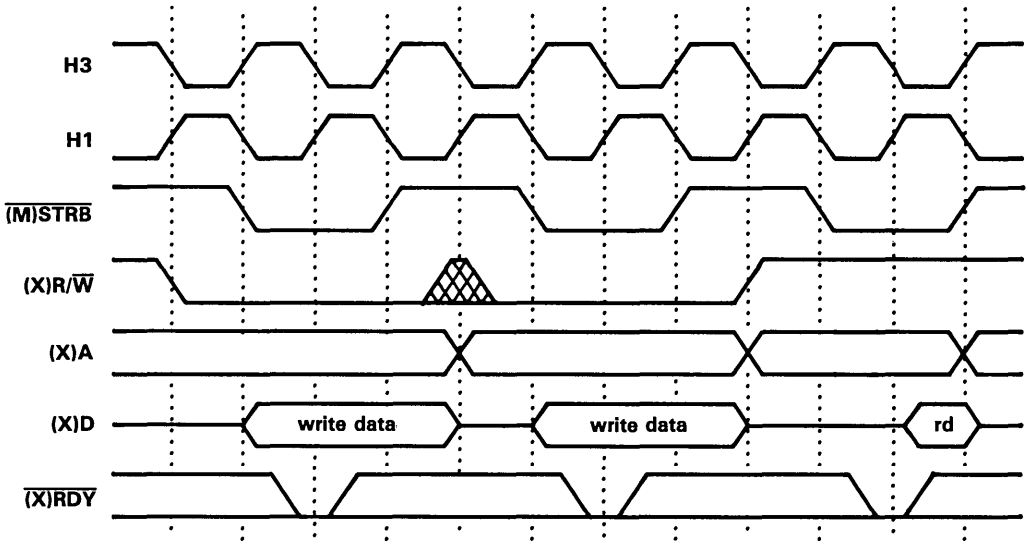


Figure 8-5. Write-Write-Read for $\overline{(M)STRB}$ = 0

Figure 8-6 illustrates a read cycle with one wait state. Since $\overline{(X)RDY}$ = 1, the read cycle is extended. $\overline{(M)STRB}$, (X)R/$\overline{W}$, and (X)A are also extended one cycle. The next time $\overline{(X)RDY}$ is sampled, it is 0.

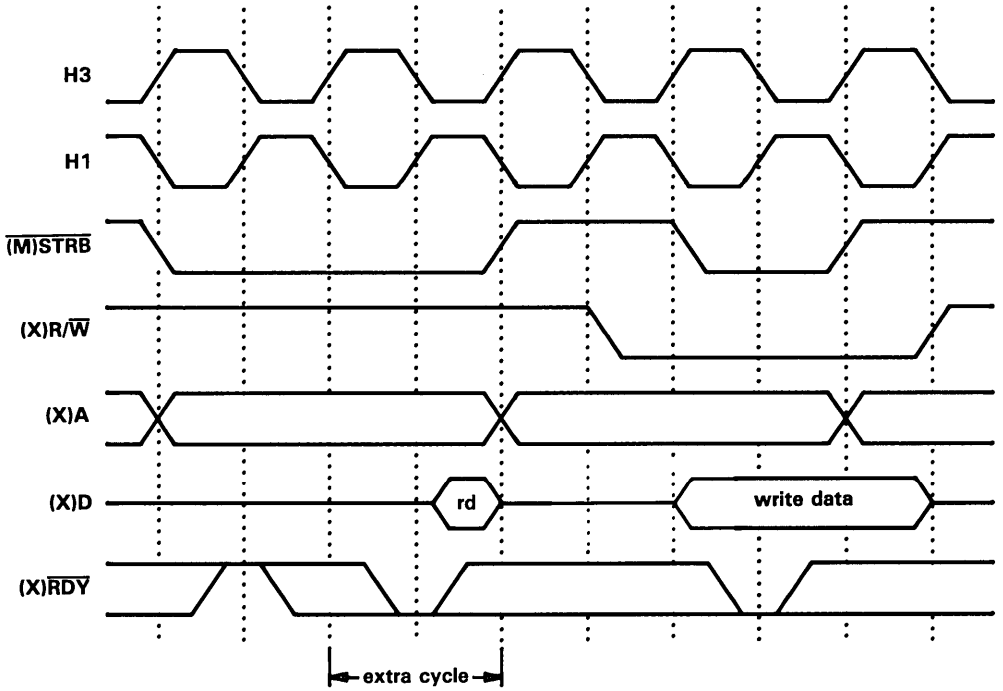


Figure 8-6. Use of Wait States for Read for $\overline{(M)STRB}$ = 0

Figure 8-7 illustrates a write cycle with one wait state. Since initially $\overline{(X)RDY}$ = 1, the write cycle is extended. $\overline{(M)STRB}$, (X)R/$\overline{W}$, and (X)A are extended one cycle. The next time $\overline{(X)RDY}$ is sampled, it is 0.

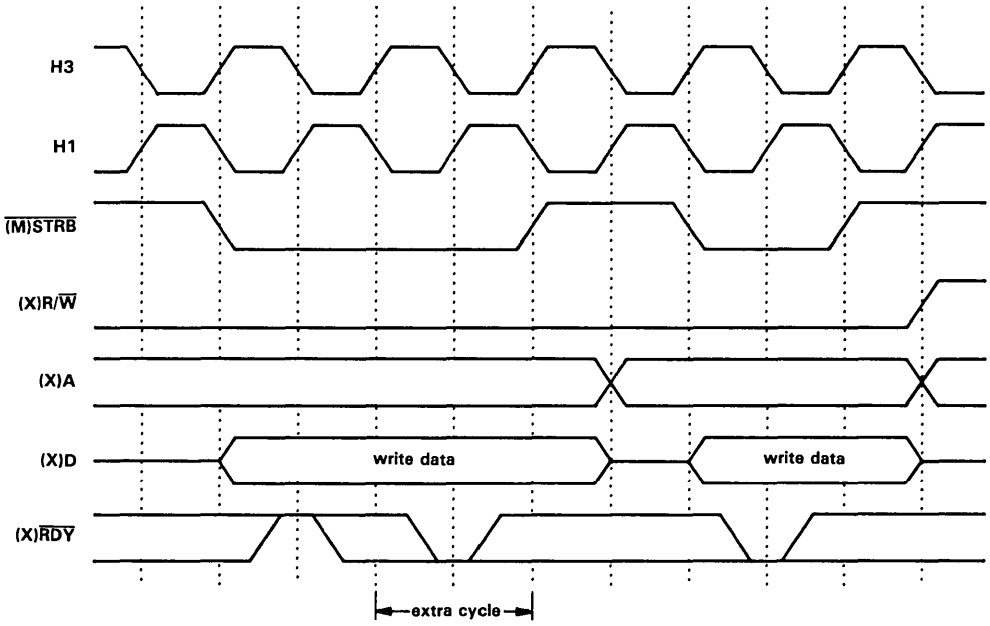


Figure 8-7. Use of Wait States for Write for $\overline{(M)STRB}$ = 0

8.2.2 Expansion Bus I/O Cycles

In contrast to primary bus and $\overline{\text{MSTRB}}$ cycles, $\overline{\text{IOSTRB}}$ reads and writes are both two cycles in duration (with no wait states) and exhibit the same timing. During these cycles, address always changes on the falling edge of H1, and $\overline{\text{IOSTRB}}$ is low from the rising edge of the first H1 cycle to the rising edge of the second H1 cycle. The $\overline{\text{IOSTRB}}$ signal always goes inactive (high) between cycles, and XR/$\overline{\text{W}}$ is high for reads and low for writes.

Figure 8-8 illustrates read and write cycles when $\overline{\text{IOSTRB}}$ is active and there are no wait states. For $\overline{\text{IOSTRB}}$ accesses, reads and writes require a minimum of two cycles. Some off-chip peripherals may change their status bits when read or written. Therefore, it is important that valid addresses be maintained when communicating with these peripherals. For reads and writes when $\overline{\text{IOSTRB}}$ is active, $\overline{\text{IOSTRB}}$ is completely framed by the address.

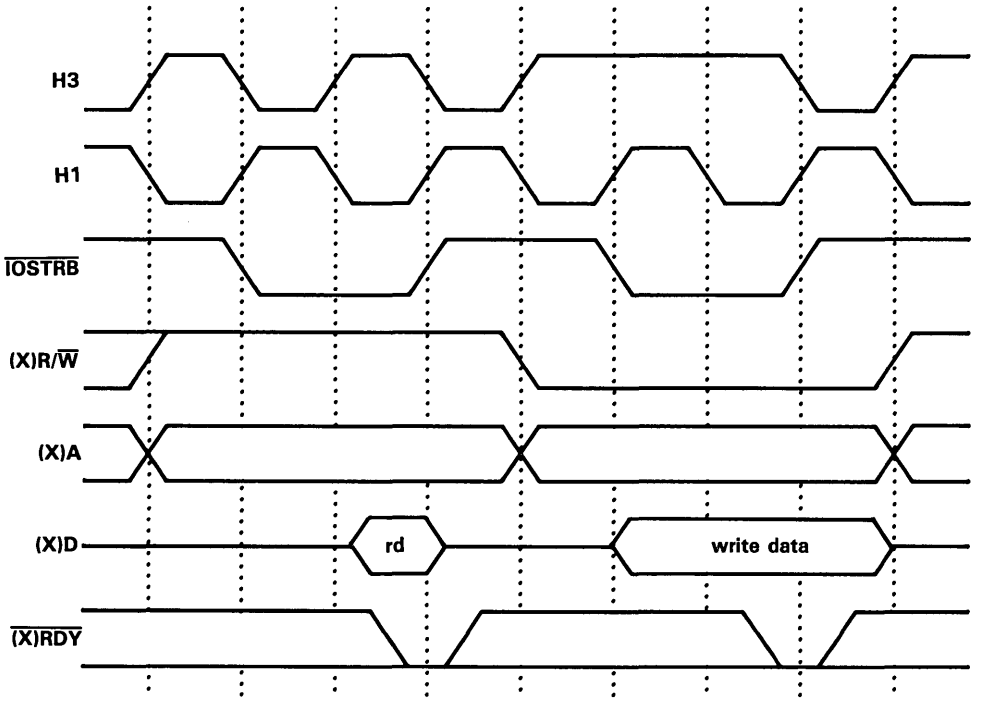


Figure 8-8. Read and Write for $\overline{\text{IOSTRB}}$ = 0

Figure 8-9 illustrates a read with one wait state when $\overline{\text{IOSTRB}}$ is active, and Figure 8-10 illustrates a write with one wait state when $\overline{\text{IOSTRB}}$ is active. For each wait state added, $\overline{\text{IOSTRB}}$, XR/$\overline{\text{W}}$, and XA are extended one clock cycle. Writes hold the data on the bus one additional cycle. The sampling of $\overline{\text{XRDY}}$ is repeated each cycle.

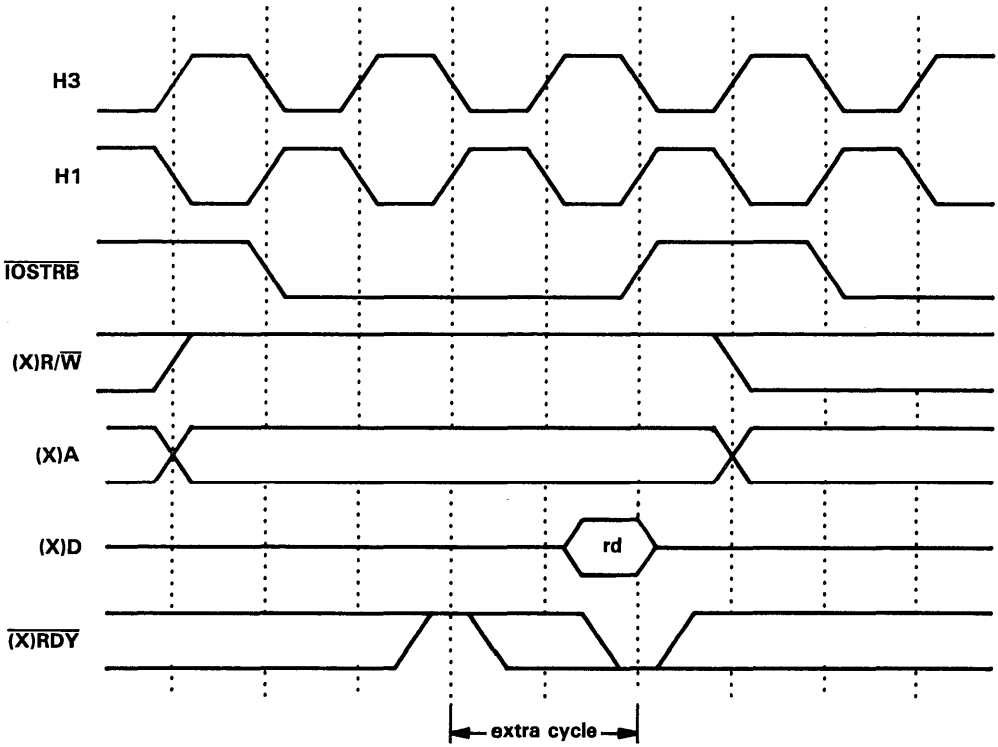


Figure 8-9. Read with One Wait-State for $\overline{\text{IOSTRB}}$ = 0

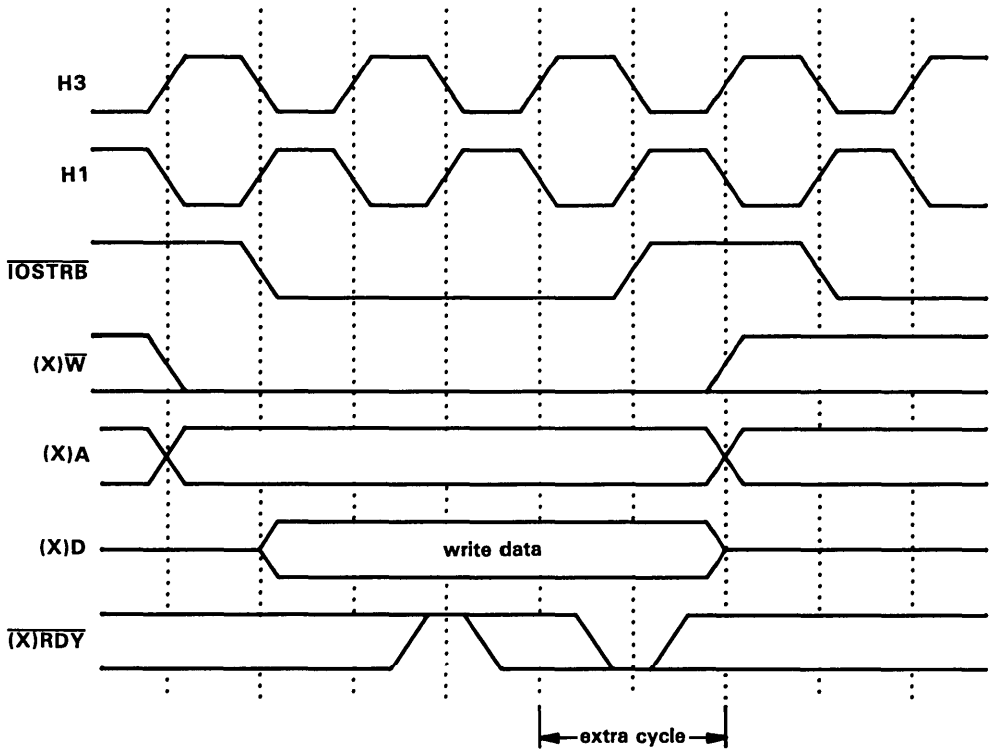


Figure 8-10. Write with One Wait-State for $\overline{\text{IOSTRB}}$ = 0

Figure 8-11 through Figure 8-13 illustrate the various transitions between memory reads and writes, and I/O writes over the expansion bis.

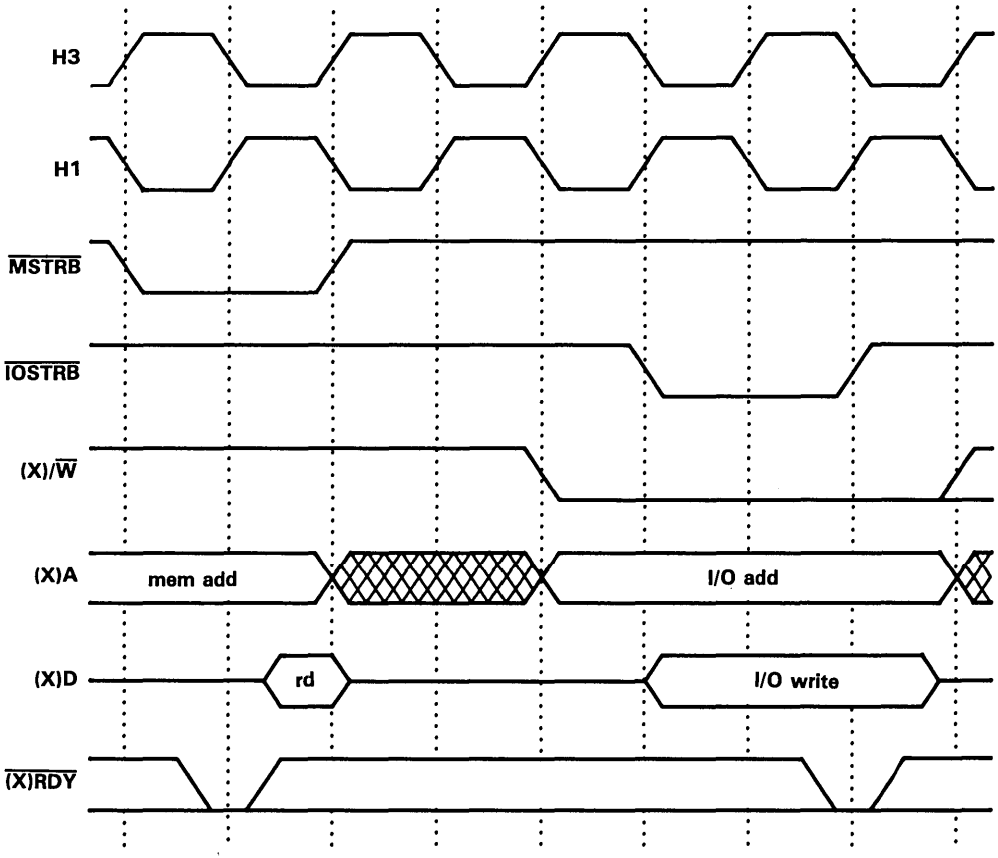


Figure 8-11. Memory Read and I/O Write for Expansion Bus

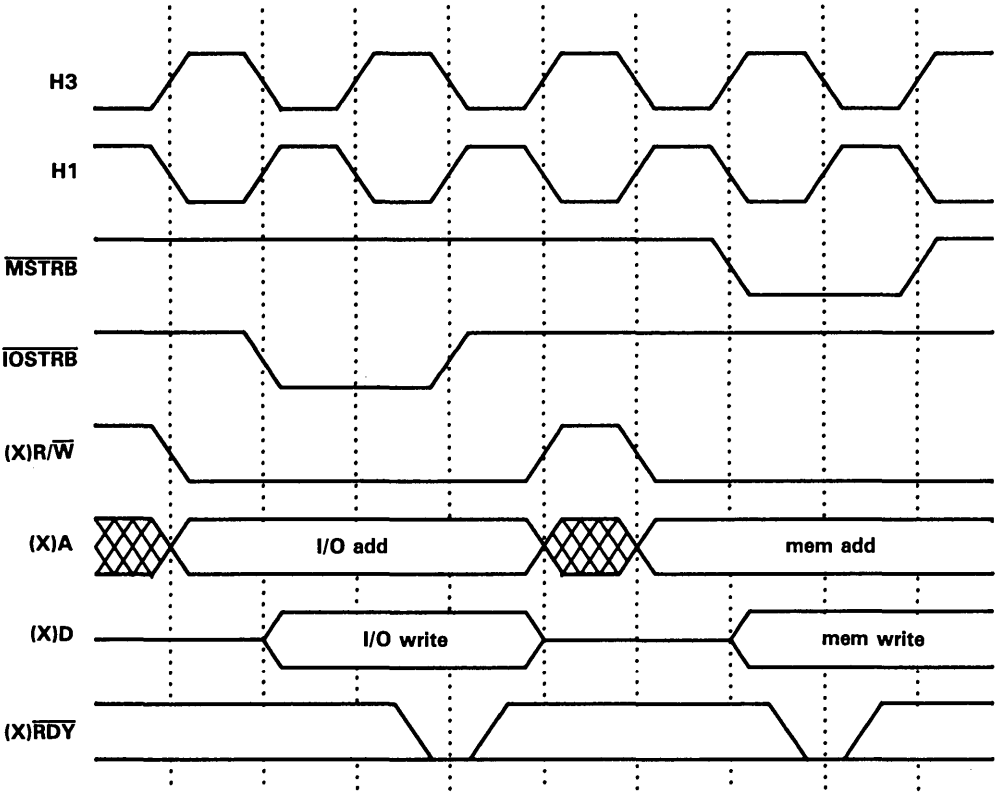


Figure 8-12. I/O Write and Memory Read for Expansion Bus

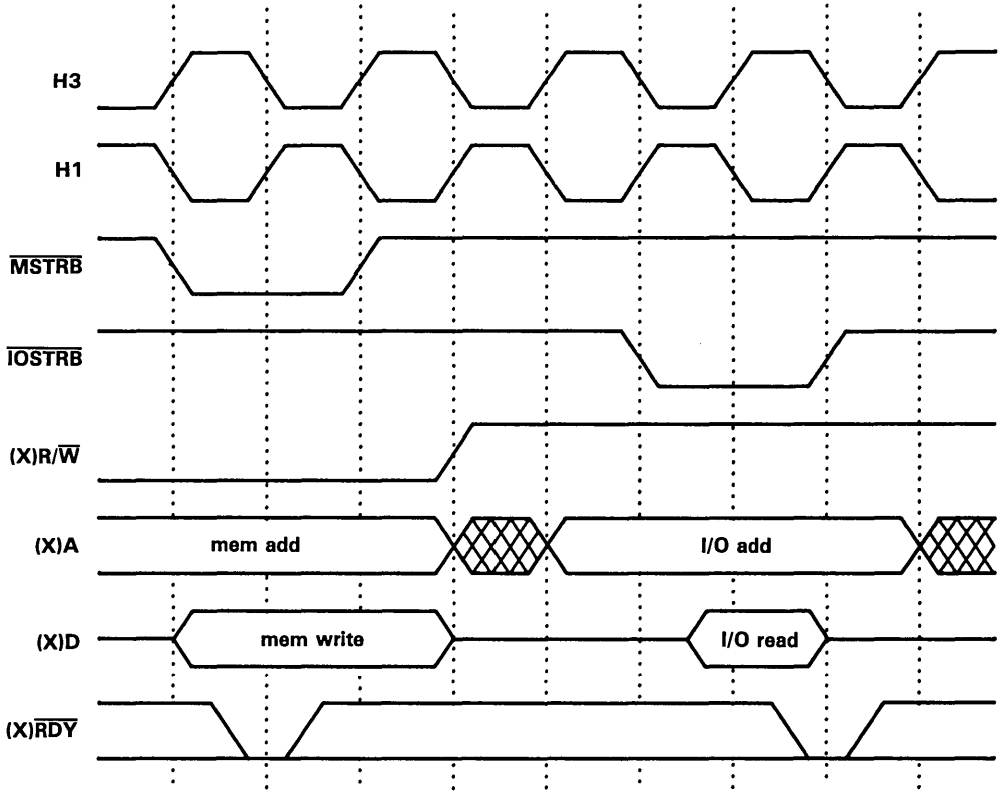


Figure 8-13. Memory Write and I/O Read for Expansion Bus

Figure 8-14 and Figure 8-15 illustrate the signal states when a bus is inactive (after a $\overline{\text{IOSTRB}}$ or $\overline{\text{(M)STRB}}$ access respectively). The strobes ($\overline{\text{STRB}}$, $\overline{\text{MSTRB}}$, $\overline{\text{IOSTRB}}$) and (X)R/$\overline{\text{W}}$ go to 1. The address is undefined, and the ready signal ($\overline{\text{XRDY}}$ or $\overline{\text{RDY}}$) is ignored.

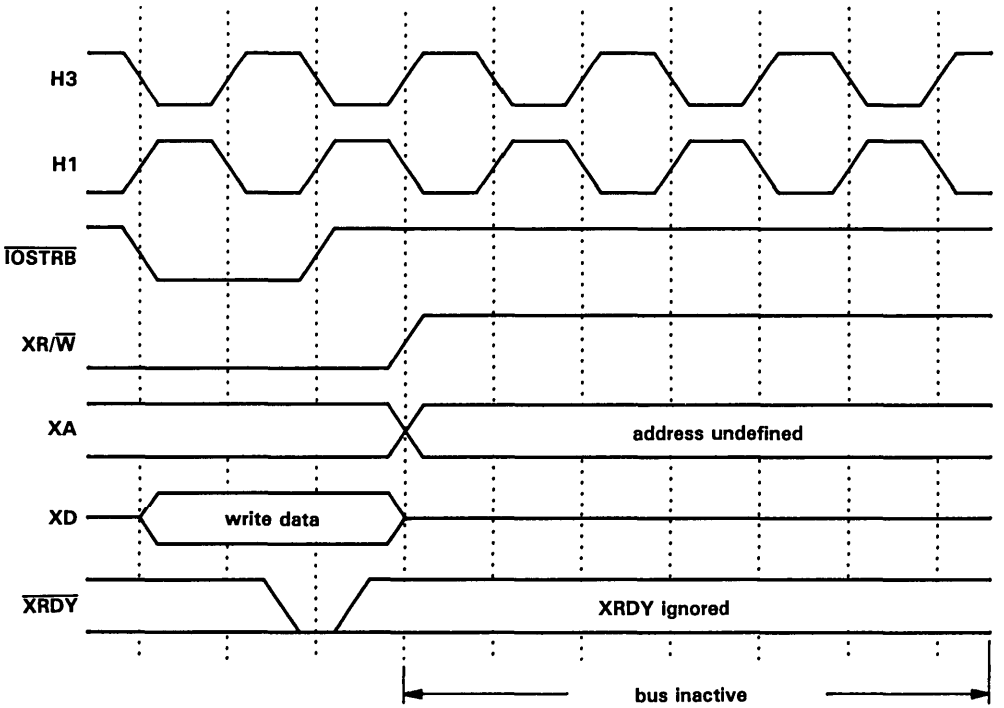


Figure 8-14. Inactive Bus States for $\overline{\text{IOSTRB}}$

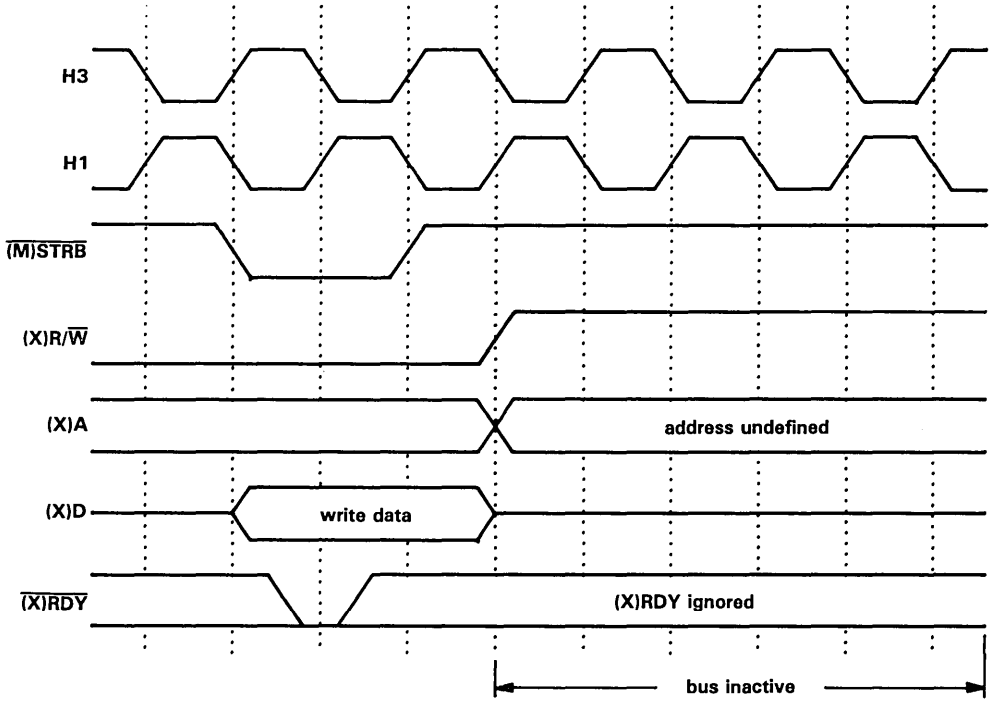


Figure 8-15. Inactive Bus States for STRB and MSTRB

Figure 8-16 illustrates the timing for $\overline{HOLD}$ and $\overline{HOLDA}$. $\overline{HOLD}$ is an external asynchronous input. There is a minimum of one cycle delay from when the processor recognizes $\overline{HOLD}$=0 until $\overline{HOLDA}$=0. When $\overline{HOLDA}$=0, the address, data buses, and associated strobes are placed in a high-impedance state. All accesses occurring over an interface are completed before a hold is acknowledged.

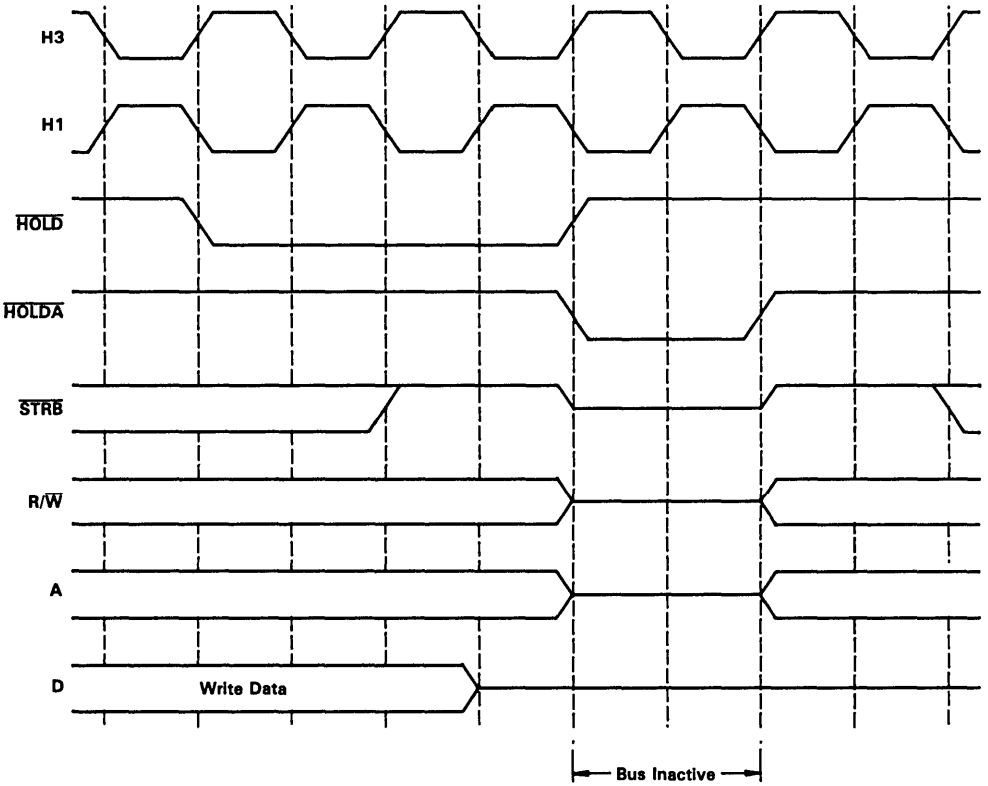


Figure 8-16. $\overline{HOLD}$ and $\overline{HOLDA}$ Timing

8.3 Programmable Wait States

Both the parallel and expansion interfaces allow the control of wait-state generation through the manipulation of their associated memory-mapped control registers. The SWW field is used to select the mode of wait-state generation, and the WTCNT field is used to load an internal timer used in the generation of wait states. The following four modes of wait-state generation can be used:

- External $\overline{RDY}$
- WTCNT-generated $\overline{RDY}_{wtcnt}$
- Logical-AND of $\overline{RDY}$ and $\overline{RDY}_{wtcnt}$
- Logical-OR of $\overline{RDY}$ and $\overline{RDY}_{wtcnt}$

These four modes are used in the generation of the internal ready signal that controls accesses, $\overline{RDY}_{int}$. As long as $\overline{RDY}_{int} = 1$, the current external access is delayed. When $\overline{RDY}_{int} = 0$, the current access completes. Since the use of programmable wait states for both external interfaces is identical, only the primary bus interface is described in the following paragraphs.

$\overline{RDY}_{wtcnt}$ is an internally generated ready signal. When an external access is begun, the value in WTCNT is loaded into a counter. WTCNT may be any value from 0 through 7. The counter is decremented every H1/H3 clock cycle until it becomes 0. Once the counter is set to 0, it remains set to 0 until the next access. While the counter is nonzero, $\overline{RDY}_{wtcnt} = 1$. While the counter is 0, $\overline{RDY}_{wtcnt} = 0$.

When SWW = 0 0, $\overline{RDY}_{int}$ is only dependent upon $\overline{RDY}$. $\overline{RDY}_{wtcnt}$ is ignored. The truth table for this mode is shown in Table 8-3.

Table 8-3. Wait-State Generation When SWW = 0 0

$\overline{RDY}$	$\overline{RDY}_{wtcnt}$	$\overline{RDY}_{int}$
0	0	0
0	1	0
1	0	1
1	1	1

When SWW = 0 1, $\overline{RDY}_{int}$ is only dependent upon $\overline{RDY}_{wtcnt}$. $\overline{RDY}$ is ignored. Table 8-4 shows the truth table for this mode.

Table 8-4. Wait-State Generation When SWW = 0 1

$\overline{RDY}$	$\overline{RDY}_{wtcnt}$	$\overline{RDY}_{int}$
0	0	0
0	1	1
1	0	0
1	1	1

When SWW = 1 0, $\overline{RDY}_{int}$ is the logical-OR (electrical-AND, since these signals are low true) of $\overline{RDY}$ and $\overline{RDY}_{wtcnt}$ (see Table 8-5).

Table 8-5. Wait-State Generation When SWW = 1 0

$\overline{RDY}$	$\overline{RDY}_{wtcnt}$	$\overline{RDY}_{int}$
0	0	0
0	1	0
1	0	0
1	1	1

When SWW = 1 1, $\overline{RDY}_{int}$ is the logical-AND (electrical-OR, since these signals are low true) of $\overline{RDY}$ and $\overline{RDY}_{wtcnt}$ The truth table for this mode is shown in Table 8-6.

Table 8-6. Wait-State Generation When SWW = 1 1

$\overline{RDY}$	$\overline{RDY}_{wtcnt}$	$\overline{RDY}_{int}$
0	0	0
0	1	1
1	0	1
1	1	1

8.4 Programmable Bank Switching

Programmable bank switching provides the capability of switching between external memory banks without the need for externally inserting wait states due to memories requiring several cycles to turn off. Bank switching is implemented on the primary bus and not on the expansion bus.

The size of a bank is determined by the number of bits specified to be examined. For example (see Figure 8-17), if BNKCMP =16, the 16 MSBs of the address are used to define a bank. Since addresses are 24 bits, the bank size is specified by the 8 LSBs, yielding a bank size of 256 words in this case. If BNKCMP $\geq$ 16, only the 16 MSBs are compared. Banksizes from 2^8 = 256 to 2^{24} = 16M are allowed. Table 8-7 summarizes the relationship between BNKCMP, the address bits used to define a bank, and the resulting bank size.

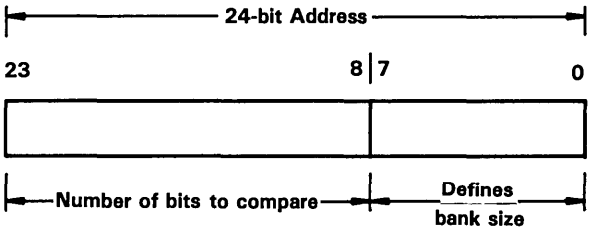


Figure 8-17. BNKCMP Example

Table 8-7. BNKCMP and Bank Size

BNKCMP	MSBs DEFINING A BANK	BANK SIZE (32-BIT WORDS)
00000	None	2^{24} = 16M
00001	23	2^{23} = 8M
00010	23-22	2^{22} = 4M
00011	23-21	2^{21} = 2M
00100	23-20	2^{20} = 1M
00101	23-19	2^{19} = 512K
00110	23-18	2^{18} = 256K
00111	23-17	2^{17} = 128K
01000	23-16	2^{16} = 64K
01001	23-15	2^{15} = 32K
01010	23-14	2^{14} = 16K
01011	23-13	2^{13} = 8K
01100	23-12	2^{12} = 4K
01101	23-11	2^{11} = 2K
01110	23-10	2^{10} = 1K
01111	23-9	2^{9} = 512
10000	23-8	2^{8} = 256
10001 through 11111	Reserved	Undefined

Internal to the TMS320C30 is a register that contains the MSBs (as defined by the BNKCMP field) of the last address used for a read or write over the primary interface. At reset, the register bits are set to zero. If the MSBs of the address being used for the current primary interface read do not match those contained in this internal register, a read cycle is not asserted for one H1/H3 clock cycle. During this extra clock cycle, the address bus switches over to the new address, but $\overline{\text{STRB}}$ is inactive (high). The contents of the internal register are replaced with the MSBs being used for the current read of the current address. If the MSBs of the address being used for the current read match the bits in the register, a normal read cycle takes place.

If repeated reads are performed from the same memory bank, no extra cycles are inserted. When reading from a different memory bank, memory conflicts are avoided by the insertion of an extra cycle. This feature can be disabled by setting BNKCMP to 0. The insertion of the extra cycle occurs only when a read is performed. The changing of the MSBs in the internal register occurs for all reads and writes over the primary interface.

Figure 8-18 illustrates the addition of an inactive cycle when switches between banks of memory occur.

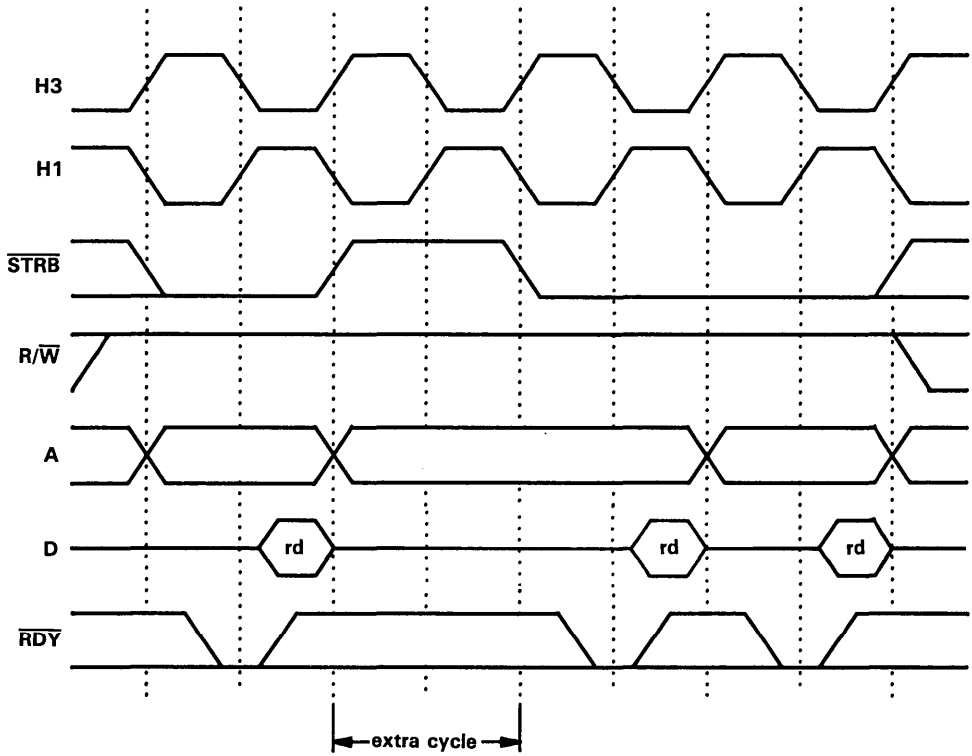


Figure 8-18. Bank Switching Example

Introduction 1

Pinout and Signal Descriptions 2

Architectural Overview 3

CPU Registers, Memory, and Cache 4

Data Formats and Floating-Point Operation 5

Addressing 6

Program Flow Control 7

External Bus Operation 8

Peripherals 9

Pipeline Operation 10

Assembly Language Instructions 11

Software Applications 12

Hardware Applications 13

Appendices A-D

Section 9

Peripherals

The TMS320C30 provides two timers, two serial ports, and an on-chip Direct Memory Access (DMA) controller. These peripheral modules are manipulated through memory-mapped registers located on the dedicated peripheral bus.

The DMA controller is used to perform input/output operations without interfering with the operation of the CPU. Therefore, it is possible to interface the TMS320C30 to slow external memories and peripherals (A/D's, serial ports, etc.) without reducing the computational throughput of the CPU. The result is improved system performance and decreased system cost.

Major topics discussed in this section on peripherals are listed below.

- Timers (Section 9.1 on page 9-2)
  - Registers
  - Pulse generation
  - Operation modes
- Serial Ports (Section 9.2 on page 9-9)
  - Registers
  - Operation configurations
  - Timing
- DMA Controller (Section 9.3 on page 9-26)
  - Registers
  - VA memory transfer operation
  - Synchronization of DMA channels

9.1 Timers

The TMS320C30 timer modules are general-purpose 32-bit timer/event counters, with two signalling modes and internal or external clocking (see Figure 9-1). The timer modules can be used to signal to the TMS320C30 or the external world at specified intervals, or to count external events. With an internal clock, the timer can be used to signal an external A/D converter to start a conversion, or it can interrupt the TMS320C30 DMA controller to begin a data transfer. With an external input, the timer can count external events and interrupt the CPU after a specified number of events. Available to each timer is an I/O pin that can be used either as an input clock to the timer, an output clock signal, or a general-purpose I/O pin.

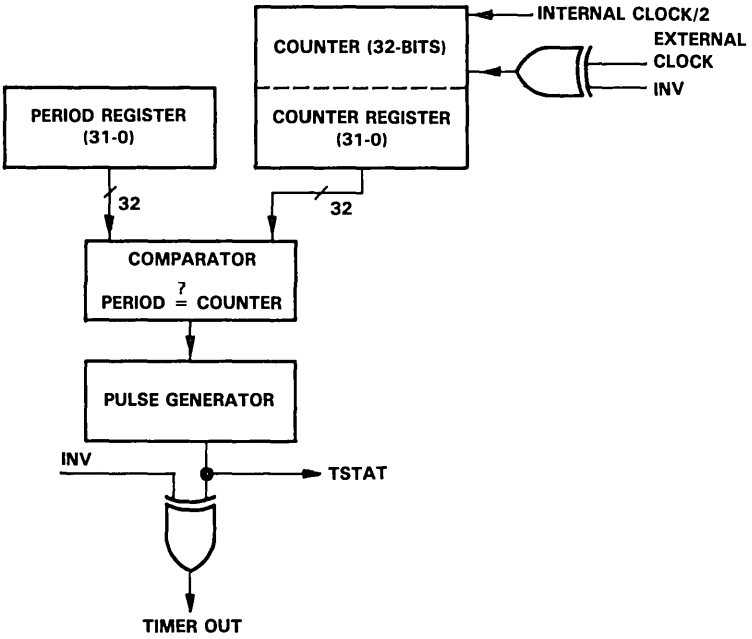


Figure 9-1. Timer Block Diagram

Three memory-mapped registers are used by each timer. They are:

- Global control register
- Period register
- Counter register

The global control register determines the operating mode of the timer, monitors the timer status, and controls the function of the I/O pin of the timer. The period register specifies the timer's signalling frequency. The counter register

contains the current value of the incrementing counter. The timer can be incremented on the rising edge or the falling edge of the input clock. The counter is zeroed whenever its value equals that in the period register. The pulse generator generates two types of external clock signals: pulse or clock. The memory map for the timer modules is shown in Figure 9-2.

Register	Peripheral Address	
	Timer 0	Timer 1
TIMER GLOBAL CONTROL REGISTER	808020h	808030h
RESERVED	808021h	808031h
RESERVED	808022h	808032h
RESERVED	808023h	808033h
TIMER COUNTER REGISTER	808024h	808034h
RESERVED	808025h	808035h
RESERVED	808026h	808036h
RESERVED	808027h	808037h
TIMER PERIOD REGISTER	808028h	808038h
RESERVED	808029h	808039h
RESERVED	80802Ah	80803Ah
RESERVED	80802Bh	80803Bh
RESERVED	80802Ch	80803Ch
RESERVED	80802Dh	80803Dh
RESERVED	80802Eh	80803Eh
RESERVED	80802Fh	80803Fh

Figure 9-2. Memory-Mapped Timer Locations

9.1.1 Timer Global Control Register

The timer global control register is a 32-bit register that contains the global and port control bits for the timer module. Table 9-1 defines the register bits, names, and functions. Bits 3-0 are the port control bits. Bits 11-6 are the timer global control bits. Figure 9-3 shows the 32-bit register. Note that at reset, all bits are set to 0 except for DATIN (set to the value read on TCLK).

Table 9-1. Timer Global Control Register Bits Summary

BITS	NAME	FUNCTION
0	FUNC	FUNC controls the function of TCLK. If FUNC = 0, TCLK is configured as a general-purpose digital I/O port. If FUNC = 1, TCLK is configured as a timer pin (see Figure 9-6) for a description of the relationship between FUNC and CLKSRC.
1	$\overline{I}$/O	If FUNC = 0 and CLKSRC = 0, TCLK is configured as a general-purpose I/O pin. In this case, if $\overline{I}$/O = 0, TCLK is configured as a general-purpose input pin. If $\overline{I}$/O = 1, TCLK is configured as a general-purpose output pin.
2	DATOUT	DATOUT drives TCLK when in I/O port mode. DATOUT can also be used as an input to the timer.
3	DATIN	Data input on TCLK or DATOUT. A write has no effect.
4-5	Reserved	Read as 0.
6	GO	The GO bit resets and starts the timer counter. When GO = 1 and the timer is not held, the counter is zeroed and begins incrementing on the next rising edge of the timer input clock. The GO bit is cleared on the same rising edge. GO = 0 has no effect on the timer.
7	$\overline{HLD}$	Counter hold signal. When this bit is zero, the counter is disabled and held in its current state. If the timer is driving TCLK, the state of TCLK is also held. The internal divide-by-two counter is also held so that the counter can continue where it left off when $\overline{HLD}$ is set to 1. The timer registers can be read and modified while the timer is being held. $\overline{RESET}$ has priority over $\overline{HLD}$. Table 9-2 shows the effect of writing to GO and $\overline{HLD}$.
8	C/$\overline{P}$	Clock/Pulse mode control. When C/$\overline{P}$ = 1, clock mode is chosen and the signalling of the status flag and external output will have a 50 percent duty cycle. When C/$\overline{P}$ = 0, the status flag and external output will be active for one H1 cycle during each timer period (see Figure 9-4).
9	CLKSRC	Specifies the source of the timer clock. When CLKSRC = 1, an internal clock with frequency equal to one-half the H1 frequency is used to increment the counter. The INV bit has no effect on the internal clock source. When CLKSRC = 0, an external signal from the TCLK pin can be used to increment the counter. The external clock is synchronized internally, thus allowing external asynchronous clock sources not exceeding the specified maximum allowable external clock frequency. This will be less than f(H1)/2. (See Figure 9-6 for a description of the relationship between FUNC and CLKSRC).
10	INV	Inverter control bit. If an external clock source is used and INV = 1, the external clock is inverted as it goes into the counter. If the output of the pulse generator is routed to TCLK and INV = 1, the output is inverted before it goes to TCLK (see Figure 9-1). If INV = 0, no inversion is performed on the input or output of the timer. The INV bit has no effect, regardless of its value, when TCLK is used in I/O port mode.
11	TSTAT	This bit indicates the status of the timer. It tracks the output of the uninverted TCLK pin. This flag sets a CPU interrupt on a transition from 0 to 1. A write has no effect.
12-31	Reserved	Read as 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	xx	xx	TSTAT	INV	CLKSRC	C/$\overline{P}$	$\overline{HLD}$	GO	xx	xx	DATIN	DATOUT	$\overline{I}$/O	FUNC
				R/W	R/W	R/W	R/W	R/W	R/W			R	R/W	R/W	R/W

NOTE: xx = Reserved bit, read as 0.
R = read, W = write.

Figure 9-3. Timer Global Control Register

Table 9-2 shows the result of a write using specified values of the GO and $\overline{HLD}$ bits in the global control register.

Table 9-2. Result of a Write of Specified Values of GO and HLD

GO	$\overline{HLD}$	RESULT
0	0	All timer operations are held. No reset is performed.
0	1	Timer proceeds from state before write.
1	0	All timer operations are held, including zeroing of the counter. The GO bit is not cleared until the timer is taken out of hold.
1	1	Timer reset and started.

9.1.2 Timer Period and Counter Registers

The 32-bit timer period register is used to specify the frequency of the timer signalling. The timer counter register is a 32-bit register, which is reset to zero whenever it increments to the value of the period register. Both registers are set to 0 at reset.

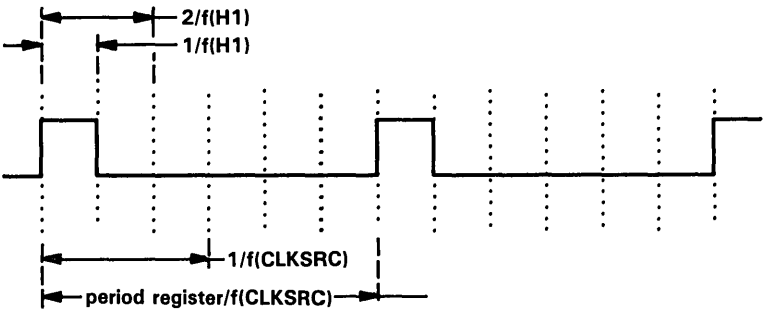
Certain boundary conditions affect timer operation, such as a zero in the period register and overflowing the counter. These conditions are listed as follows:

- When the period and counter registers are zero, the operation of the timer is dependent upon the C/$\overline{P}$ mode selected. In pulse mode (C/$\overline{P}$ = 0), TSTAT is set and remains set. In clock mode (C/$\overline{P}$ = 1), the width of the cycle is 2/f(H1) and the external clocks are ignored.
- When the counter register is not 0 and the period register = 0, the counter will count, roll over to 0, and then behave as described above.
- When the counter register is set to a value greater than the period register, the counter may overflow when being incremented. Once the counter reaches its maximum 32-bit value (0FFFFFFFFh), it simply clocks over to 0 and continues.

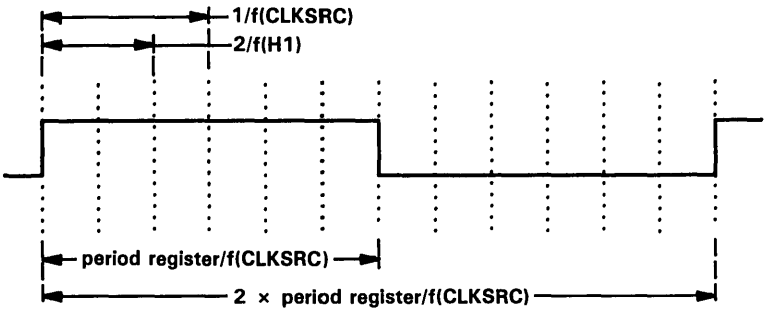
Writes from the peripheral bus override register updates from the counter or new status updates to the control register.

9.1.3 Timer Pulse Generation

The timer pulse generator (see Figure 9-1) can generate several different external signals. These signals may be inverted with the INV bit. The two basic modes are pulse mode and clock mode, as shown in Figure 9-4. In both modes, an internal clock source has a frequency of f(H1)/2, and an external clock source has a maximum frequency less than f(H1)/2. Refer to timer timing in Appendix A. In pulse mode (C/$\overline{P}$ = 0), the width of the pulse is 1/f(H1).



(a) TSTAT AND TIMER OUTPUT (INV = 0) WHEN C/$\overline{P}$ = 0 (PULSE MODE)



(b) TSTAT AND TIMER OUPUT (INV = 0) WHEN C/$\overline{P}$ = 1 (CLOCK MODE)

Figure 9-4. Timer Timing

The rate of timer signaling is determined by the frequency of the timer input clock and the period register. The following equations are valid with either an internal or an external timer clock:

f(pulse mode) = f(timer clock)/period register

f(clock mode) = f(timer clock)/(2 x period register)

9.1.4 Timer Operation Modes

The timer can receive its input and send its output in several different modes, depending upon the setting of CLKSRC, FUNC, and $\overline{I}$/O. The four timer modes of operation are defined as follows:

- If CLKSRC = 1 and FUNC = 0, the timer input comes from the internal clock. The internal clock is not affected by the INV bit. In this mode, TCLK is connected to the I/O port control and can be used as a general-purpose I/O pin (see Figure 9-5). If $\overline{I}$/O = 0, TCLK is configured as a general-purpose input pin whose state can be read in DATIN. DATOUT has no effect on TCLK or DATIN. If $\overline{I}$/O = 1, TCLK is configured as a general-purpose output pin. DATOUT is placed on TCLK and can be read in DATIN.

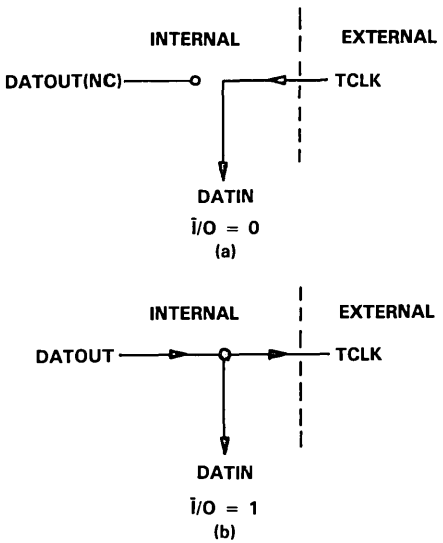


Figure 9-5. Timer I/O Port Configurations

- If CLKSRC = 1 and FUNC = 1, the timer input comes from the internal clock and the timer output goes to TCLK. This value may be inverted using INV, and the value output on TCLK can be read in DATIN.

- If CLKSRC = 0 and FUNC = 0, the timer is driven according to the status of the $\overline{I}$/O bit. If $\overline{I}$/O = 0, the timer input comes from TCLK. This value can be inverted using INV, and the value of TCLK can be read in DATIN.

If $\overline{I}/O$ = 1, TCLK is an output pin. Then TCLK and the timer are both driven by DATOUT. All 0 to 1 transitions of DATOUT increment the counter. INV has no effect on DATOUT. The value of DATOUT can be read in DATIN.

- If CLKSRC = 0 and FUNC = 1, TCLK drives the timer. If INV = 0, all 0 to 1 transitions of TCLK increment the counter. If INV = 1, all 1 to 0 transitions of TCLK increment the counter. The value of TCLK can be read in DATIN.

Figure 9-6 shows the four timer modes of operation.

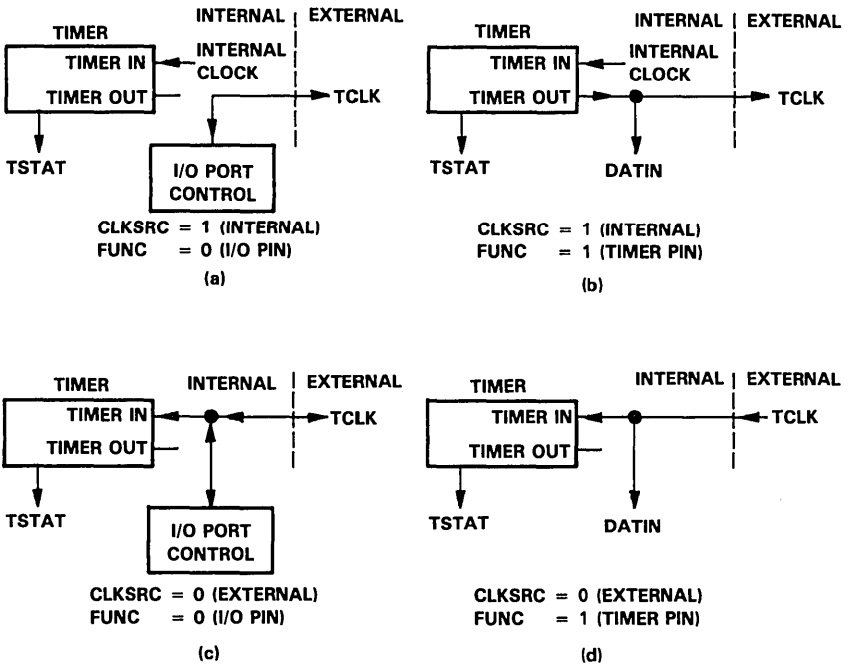


Figure 9-6. Timer Modes as Defined by CLKSRC and FUNC

9.2 Serial Ports

The two TMS320C30 serial ports are totally independent. Both serial ports are identical with a complementary set of control registers in each one. Each serial port can be configured to transfer 8, 16, 24, or 32 bits of data per word. The clock for each serial port can originate either internally or externally. An internally generated clock is a divide-down of the clockout frequency (H1). A continuous transfer mode is available which allows the serial port to transmit and receive any number of words without new synchronization pulses.

Eight memory-mapped registers are provided for each serial port. They are:

- Port global control register
- Two port control registers for the six I/O pins
- Three port receive/transmit timer registers
- Data transmit register
- Data receive register

The global control register controls the global functions of the serial port and determines the serial port operating mode . Two port control registers control the functions of the six serial port pins. The transmit buffer contains the next complete word to be transmitted. The receive buffer contains the last complete word received. Three additional registers are associated with the transmit/receive sections of the serial port timer. A serial port block diagram is shown in Figure 9-7, and the memory map of a serial port is shown in Figure 9-8.

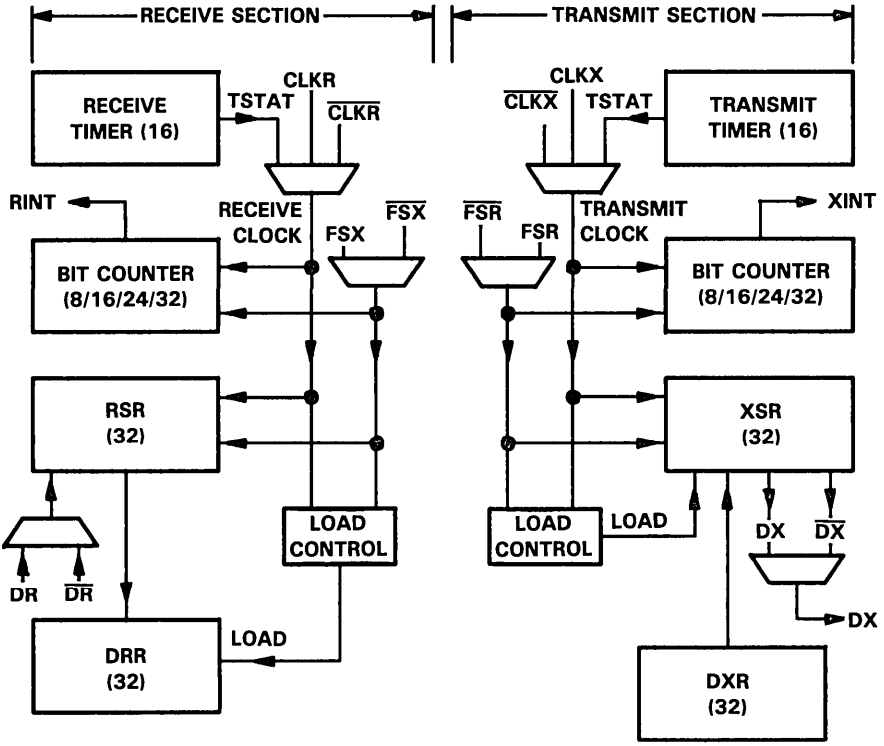


Figure 9-7. Serial Port Block Diagram

Register	Peripheral Address	
	Serial Port 0	Serial Port 1
PORT GLOBAL CONTROL REGISTER	808040h	808050h
RESERVED	808041h	808051h
FSX/DX/CLKX PORT CONTROL REGISTER	808042h	808052h
FSR/DR/CLKR PORT CONTROL REGISTER	808043h	808053h
R/X TIMER CONTROL REGISTER	808044h	808054h
R/X TIMER COUNTER REGISTER	808045h	808055h
R/X TIMER PERIOD REGISTER	808046h	808056h
RESERVED	808047h	808057h
DATA TRANSMIT REGISTER	808048h	808058h
RESERVED	808049h	808059h
RESERVED	80804Ah	80805Ah
RESERVED	80804Bh	80805Bh
DATA RECEIVE REGISTER	80804Ch	80805Ch
RESERVED	80804Dh	80805Dh
RESERVED	80804Eh	80805Eh
RESERVED	80804Fh	80805Fh

Figure 9-8. Memory-Mapped Locations for the Serial Port

9.2.1 Serial Port Global Control Register

The serial port global control register is a 32-bit register that contains the global control bits for the serial port. Table 9-3 defines the register bits, bit names, and bit functions. The register is shown in Figure 9-9.

Table 9-3. Serial Port Global Control Register Bits Summary

BIT	NAME	FUNCTION
0	RRDY	If RRDY = 1, the receive buffer has new data and is ready to be read. A three H1/H3 cycle delay occurs from the reading of DRR to RRDY = 1. The rising edge of this signal sets RINT. If RRDY = 0, the receive buffer does not have new data since the last read. RRDY is set to 0 at reset and after the receive buffer is read.
1	XRDY	If XRDY = 1, the transmit buffer has written the last bit of data to the shifter and is ready for a new word. A three H1/H3 cycle delay occurs from the loading of the transmit shifter to XRDY being set to 1. The rising edge of this signal sets XINT. If XRDY = 0, the transmit buffer has not written the last bit of data to the transmit shifter and is not ready for a new word. XRDY is set to 1 at reset.
2	FSXOUT	This bit configures the FSX pin as an input (FSXOUT = 0) or an output (FSXOUT = 1).
3	XSREMPTY	If XSREMPTY = 0, the transmit shift register is empty. If XSREMPTY = 1, the transmit shift register is not empty. This bit is set to 0 at reset or by an XRESET.
4	RSRFULL	If RSRFULL = 1, an overrun of the receiver has occurred. In continuous mode, RSRFULL is set to 1 when both RSR and DRR are full. In noncontinuous mode, RSRFULL is set to 1 when RSR and DRR are full and a new FSR is received. A read causes this bit to be set to 0. This bit can only be set to 0 by a system reset, a serial port receive reset (RRESET = 1), or a read. When the receiver tries to set RSRFULL to a 1 at the same time that the global register is read, the receiver will dominate and RSRFULL is set to 1. If RSRFULL = 0, no overrun of the receiver has occurred.
5	HS	If HS = 1, the handshake mode is enabled. If HS = 0, the handshake mode is disabled.
6	XCLKSRCE	If XCLKSRCE = 1, the internal transmit clock is used. If XCLKSRCE = 0, the external transmit clock is used.
7	RCLKSRCE	If RCLKSRCE = 1, the internal receive clock is used. If RCLKSRCE = 0, the external receive clock is used.
8	XVAREN	This bit specifies fixed (XVAREN = 0) or variable (XVAREN = 1) data rate signalling when transmitting. With a fixed data rate, FSX is active for at least one XCLK cycle, and then goes inactive before transmission begins. With variable data rate, FSX is active while all bits are being transmitted. When using an external FSX and variable data rate signaling, the DX pin is driven by the transmitter when FSX is held active or when a word is being shifted out.
9	RVAREN	This bit specifies fixed (RVAREN = 0) or variable (RVAREN = 1) data rate signalling when receiving. With a fixed data rate, FSR is active for at least one RCLK cycle, and then goes inactive before the reception begins. With variable data rate, FSR is active while all bits are being received.
10	XFSM	Transmit frame sync mode. Configures the port for continuous mode operation (XFSM = 1) or standard mode (XFSM = 0). In continuous mode, only the first word of a block generates a sync pulse, and the rest are simply transmitted continuously to the end of the block. In standard mode, each word has an associated sync pulse.
11	RFSM	Receive frame sync mode. Configures the port for continuous mode (RFSM = 1) or standard mode (RFSM = 0) operation. In continuous mode, only the first word of a block generates a sync pulse, and the rest are simply received continuously without expectation of another sync pulse. In standard mode, each word received has an associated sync pulse.
12	CLKXP	CLKX polarity. If CLKXP = 0, CLKX is active high. If CLKXP = 1, CLKX is active low.

Table 9-3. Serial Port Global Control Register Bits Summary (Concluded)

BIT	NAME	FUNCTION
13	CLKRP	CLKR polarity. If CLKRP = 0, CLKR is active high. If CLKRP = 1, CLKR is active low.
14	DXP	DX polarity. If DXP = 0, DX is active high. If DXP = 1, DX is active low.
15	DRP	DR polarity. If DRP = 0, DR is active high. If DRP = 1, DR is active low.
16	FSXP	FSX polarity. If FSXP = 0, FSX is active high. If FSXP = 1, FSX is active low.
17	FSRP	FSR polarity. If FSRP = 0, FSR is active high. If FSRP = 1, FSR is active low.
18-19	XLEN	This bit defines the word length of serial data transmitted. All data is assumed to be right-justified in the transmit buffer when fewer than 32 bits are specified. 0 0 --- 8 bits 1 0 --- 24 bits 0 1 --- 16 bits 1 1 --- 32 bits
20-21	RLEN	This bit defines the word length of serial data received. All data is right-justified in the receive buffer. 0 0 --- 8 bits 1 0 --- 24 bits 0 1 --- 16 bits 1 1 --- 32 bits
22	XTINT	Transmit timer interrupt enable. If XTINT = 0, the transmit timer interrupt is disabled. If XTINT = 1, the transmit timer interrupt is enabled.
23	XINT	Transmit interrupt enable. If XINT = 0, the transmit interrupt is disabled. If XINT = 1, the transmit interrupt is enabled. Note that the CPU transmit interrupt flag XINT is the logical OR of the enabled transmit timer interrupt and the enabled transmit interrupt.
24	RTINT	Receive timer interrupt enable. If RTINT = 0, the receive timer interrupt is disabled. If RTINT = 1, the receive timer interrupt is enabled.
25	RINT	Receive interrupt enable. If RINT = 0, the receive interrupt is disabled. If RINT = 1, the receive interrupt is enabled. Note that the CPU receive interrupt flag RINT is the OR of the enabled receive timer interrupt and the enabled receive interrupt
26	XRESET	Transmit reset. If XRESET = 0, the transmit side of the serial port is reset. To take the transmit side of the serial port out of reset, XRESET should be set to 1. However, XRESET should not be set to 1 until at least three cycles after XRESET goes inactive. This applies only to system reset. Setting XRESET to 0 does not change the contents of any of the serial port control registers. It places the transmitter in a state corresponding to the beginning of a frame of data. Resetting the transmitter generates a transmit interrupt. This bit should be set at the same time the mode of the transmitter is set. XFSM can be toggled without resetting the global control register.
27	RRESET	Receive reset. If RRESET = 0, the receive side of the serial port is reset. To take the transmit side of the serial port out of reset, XRESET should be set to 1. Setting RRESET to 0 does not change the contents of any of the serial port control registers. It places the receiver in a state corresponding to the beginning of a frame of data. This bit should be set at the same time the mode of the receiver is set. RFSM can be toggled without resetting the global control register.
28-31	Reserved	Read as 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
xx	xx	xx	xx	RRESET	XRESET	RINT	RTINT	XINT	XTINT	RLEN		XLEN		FSRP	FSXP
				R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DRP	DXP	CLKRP	CLKXP	RFSM	XFSM	RVAREN	XVAREN	RCLK SRCE	XCLK SRCE	HS	RSR FULL	XSR EMPTY	FSXOUT	XRDY	RRDY
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	R	R/W	R	R

NOTE: xx = Reserved bit, read as 0.
R = read, W = write.

Figure 9-9. Serial Port Global Control Register

9.2.2 FSX/DX/CLKX Port Control Register

This 32-bit port control register controls the function of the serial port FSX, DX, and CLKX pins. At reset, all bits are set to 0. Table 9-4 defines the register bits, bit names, and functions. Figure 9-10 shows this port control register.

Table 9-4. FSX/DX/CLKX Port Control Register Bits Summary

BIT	NAME	FUNCTION
0	CLKXFUNC	CLKXFUNC controls the function of CLKX. If CLKXFUNC = 0, CLKX is configured as a general-purpose digital I/O port. If CLKXFUNC = 1, CLKX is a serial port pin.
1	CLKX$\overline{I}$/O	If CLKX$\overline{I}$/O = 0, CLKR is configured as a general-purpose input pin. If CLKX$\overline{I}$/O = 1, CLKX is configured as a general-purpose output pin.
2	CLKXDATOUT	Data output on CLKX.
3	CLKXDATIN	Data input on CLKX. A write has no effect.
4	DXFUNC	DXFUNC controls the function of DX. If DXFUNC = 0, DX is configured as a general-purpose digital I/O port. If DXFUNC = 1, DX is a serial port pin.
5	DX$\overline{I}$/O	If DX$\overline{I}$/O = 0, DX is configured as a general-purpose input pin. If DX$\overline{I}$/O = 1, DX is configured as a general-purpose output pin.
6	DXDATOUT	Data output on DX.
7	DXDATIN	Data input on DX. A write has no effect.
8	FSXFUNC	FSXFUNC controls the function of FSX. If FSXFUNC = 0, FSX is configured as a general-purpose digital I/O port. If FSXFUNC = 1, FSX is a serial port pin.
9	FSX$\overline{I}$/O	If FSX$\overline{I}$/O = 0, FSX is configured as a general-purpose input pin. If FSX$\overline{I}$/O = 1, FSX is configured as a general-purpose output pin.
10	FSXDATOUT	Data output on FSX.
11	FSXDATIN	Data input on FSX. A write has no effect.
12-31	Reserved	Read as 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	xx	xx	FSX DATIN	FSX DATOUT	FSX I/O	FSX FUNC	DX DATIN	DX DATOUT	DX I/O	DX FUNC	CLKX DATIN	CLKX DATOUT	CLKX I/O	CLKX FUNC
				R	R/W	R/W	R/W	R	R/W	R/W	R/W	R	R/W	R/W	R/W

NOTE: xx = Reserved bit, read as 0.
R = read, W = write.

Figure 9-10. FSX/DX/CLKX Port Control Register

9.2.3 FSR/DR/CLKR Port Control Register

This 32-bit port control register is controlled by the function of the serial port FSR, DR, and CLKR pins. At reset, all bits are set to 0. Table 9-5 defines the register bits, the bit names, and functions. Figure 9-11 illustrates this port control register.

Table 9-5. FSR/DR/CLKR Port Control Register Bits Summary

BIT	NAME	FUNCTION
0	CLKRFUNC	CLKRFUNC controls the function of CLKR. If CLKRFUNC = 0, CLKR is configured as a general-purpose digital I/O port. If CLKRFUNC = 1, CLKR is a serial port pin.
1	CLKR$\overline{I}$/O	If CLKR$\overline{I}$/O = 0, CLKR is configured as a general-purpose input pin. If CLKR$\overline{I}$/O = 1, CLKR is configured as a general-purpose output pin.
2	CLKRDATOUT	Data output on CLKR.
3	CLKRDATIN	Data input on CLKR. A write has no effect.
4	DRFUNC	DRFUNC controls the function of DR. If DRFUNC = 0, DR is configured as a general-purpose digital I/O port. If DRFUNC = 1, DR is a serial port pin.
5	DR$\overline{I}$/O	If DR$\overline{I}$/O = 0, DR is configured as a general-purpose input pin. If DR$\overline{I}$/O = 1, TCLK is configured as a general-purpose output pin.
6	DRDATOUT	Data output on DR.
7	DRDATIN	Data input on DR. A write has no effect.
8	FSRFUNC	FSRFUNC controls the function of FSR. If FSRFUNC = 0, FSR is configured as a general-purpose digital I/O port. If FSRFUNC = 1, FSR is a serial port pin.
9	FSR$\overline{I}$/O	If FSR$\overline{I}$/O = 0, FSR is configured as a general-purpose input pin. If FSR$\overline{I}$/O = 1, FSR is configured as a general-purpose output pin.
10	FSRDATOUT	Data output on FSR.
11	FSRDATIN	Data input on FSR. A write has no effect.
12-31	Reserved	Read as 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	xx	xx	FSR DATIN	FSR DATOUT	FSR I/O	FSR FUNC	DR DATIN	DR DATOUT	DR I/O	DR FUNC	CLKR DATIN	CLKR DATOUT	CLKR I/O	CLKR FUNC
				R	R/W	R/W	R/W	R	R/W	R/W	R/W	R	R/W	R/W	R/W

NOTE: xx = Reserved bit, read as 0.
R = read, W = write.

Figure 9-11. FSR/DR/CLKR Port Control Register

9.2.4 Receive/Transmit Timer Control Register

A 32-bit receive/transmit timer control register contains the control bits for the timer module. At reset, all bits are set to 0. Table 9-6 lists the register bits, bit names, and functions. Bits 5-0 control the transmitter timer. Bits 11-6 control the receiver timer. Figure 9-12 shows the register.

Table 9-6. Receive/Transmit Timer Control Register

BIT	NAME	FUNCTION
0	XGO	The XGO bit resets and starts the transmit timer counter. When XGO is set to 1 and the timer is not held, the counter is zeroed and begins incrementing on the next rising edge of the timer input clock. The XGO bit is cleared on the same rising edge. Writing 0 to XGO has no effect on the transmit timer.
1	$\overline{\text{XHLD}}$	Transmit counter hold signal. When this bit is set to 0, the counter is disabled and held in its current state. The internal divide-by-two counter is also held so that the counter will continue where it left off when $\overline{\text{XHLD}}$ is set to 1. The timer registers may be read and modified while the timer is being held. $\overline{\text{RESET}}$ has priority over $\overline{\text{XHLD}}$.
2	XC/$\overline{\text{P}}$	XClock/Pulse mode control. When XC/$\overline{\text{P}}$ = 1, the clock mode is chosen. The signalling of the status flag and external output has a 50-percent duty cycle. When XC/$\overline{\text{P}}$ = 0, the status flag and external output are active for one CLKOUT cycle during each timer period.
3	XCLKSRC	This bit specifies the source of the transmit timer clock. When XCLKSRC = 1, an internal clock with frequency equal to one-half the CLKOUT frequency is used to increment the counter. When XCLKSRC = 0, an external signal from the CLKX pin can be used to increment the counter. The external clock source is synchronized internally, thus allowing for external asynchronous clock sources that do not exceed the specified maximum allowable external clock frequency, i.e., less than f(H1)/2.6.
4	Reserved	Read as zero.
5	XTSTAT	This bit indicates the status of the receive timer. It tracks what would be the output of the uninverted CLKX pin. This flag sets a CPU interrupt on a transition from 0 to 1. A write has no effect.
6	RGO	The RGO bit resets and starts the receive timer counter. When RGO is set to 1 and the timer is not held, the counter is zeroed and begins incrementing on the next rising edge of the timer input clock. The RGO bit is cleared on the same rising edge. Writing 0 to RGO has no effect on the receive timer.
7	$\overline{\text{RHLD}}$	Receive counter hold signal. When this bit is set to 0, the counter is disabled and held in its current state. The internal divide-by-two counter is also held so that the counter will continue where it left off when $\overline{\text{RHLD}}$ is set to 1. The timer registers may be read and modified while the timer is being held. $\overline{\text{RESET}}$ has priority over $\overline{\text{RHLD}}$.

Table 9-6. Receive/Transmit Timer Control Register (Concluded)

BIT	NAME	FUNCTION
8	RC/$\overline{P}$	RClock/Pulse mode control. When RC/$\overline{P}$ = 1, the clock mode is chosen. The signalling of the status flag and external output has a 50-percent duty cycle. When RC/$\overline{P}$ = 0, the status flag and external output are active for one CLKOUT cycle during each timer period.
9	RCLKSRC	This bit specifies the source of the receive timer clock. When RCLKSRC = 1, an internal clock with frequency equal to one-half the CLKOUT frequency is used to increment the counter. When RCLKSRC = 0, an external signal from the CLKR pin can be used to increment the counter. The external clock source is synchronized internally, thus allowing for external asynchronous clock sources that do not exceed the specified maximum allowable external clock frequency, i.e., less than f(H1)/2.6.
10	Reserved	Read as zero.
11	RTSTAT	This bit indicates the status of the receive timer. It tracks what would be the output of the uninverted CLKR pin. This flag sets a CPU interrupt on a transition from 0 to 1. A write has no effect.
12-31	Reserved	Read as 0.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx

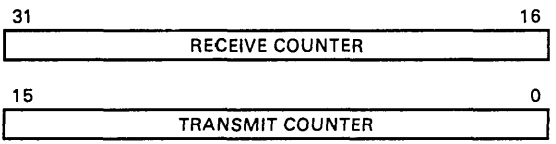
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	xx	xx	RTSTAT	xx	RCLKSRC	RC/$\overline{P}$	$\overline{RHLD}$	RGO	XTSTAT	xx	XCLKSRC	XC/$\overline{P}$	$\overline{XHLD}$	XGO
				R		R/W	R/W	R/W	R/W	R		R/W	R/W	R/W	R/W

NOTE: xx = Reserved bit, read as 0.
R = read, W = write.

Figure 9-12. Receive/Transmit Timer Control Register

9.2.5 Receive/Transmit Timer Counter Register

The timer counter register is a 32-bit register (see Figure 9-13). Bits 15-0 are the transmit timer counter, and bits 31-16 are the receive timer counter. Each counter is set to 0 whenever it increments to the value of the counter. It is also set to 0 at reset.

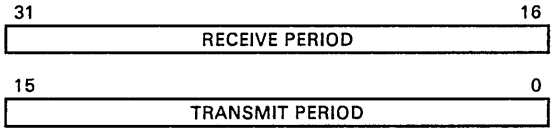


NOTE: All bits are read/write.

Figure 9-13. Receive/Transmit Timer Counter Register

9.2.6 Receive/Transmit Timer Period Register

The timer period register is a 32-bit register (see Figure 9-14) Bits 15-0 are the timer transmit period, and bits 31-16 are the receive period. Each register is used to specify the period of the timer. It is also set to 0 at reset.



NOTE: All bits are read/write.

Figure 9-14. Receive/Transmit Timer Period Register

9.2.7 Data Transmit Register

When the data transmit register (DXR) is loaded, the transmitter loads the word into the transmit shift register (XSR), and the bits are shifted out. The delay from a write to DXR until an FSX occurs (or can be accepted) is two CLKX cycles. The word is not loaded into the shift register until the shifter is empty. When DXR is loaded into XSR, the XRDY bit is set, specifying that the buffer is available to receive the next word. Four tap points within the transmit shift register are used to transmit the word. These tap points correspond to the four data word sizes and are illustrated in Figure 9-15. The shift is a left-shift (LSB to MSB) with the data shifted out of the MSB corresponding to the appropriate tap point.

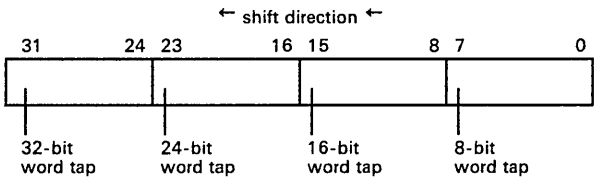


Figure 9-15. Transmit Buffer Shift Operation

9.2.8 Data Receive Register

When serial data is input, the receiver shifts the bits into the receive shift register (RSR). When the specified number of bits are shifted in, the data receive register (DRR) is loaded from RSR and the RRDY status bit is set. The receiver is double-buffered. If the DRR has not been read and the RSR is full, the receiver is frozen. New data coming into the DR pin is ignored. The receive shifter will not write over the DRR. The DRR must be read to allow new data in the RSR to be transferred to the DRR. When a write to DRR occurs at the same time that a RSR to DRR transfer takes place, the RSR to DRR transfer has priority.

Data is shifted to the left (LSB to MSB). Figure 9-16 illustrates what happens when words less than 32 bits are shifted into the serial port. In this figure, it

is assumed that an 8-bit word is being received and that the upper three bytes of the receive buffer are originally undefined. In the first portion of the figure, byte a has been shifted in. When byte b is shifted in, byte a is shifted to the left. When the data receive register is read, both bytes a and b are read.

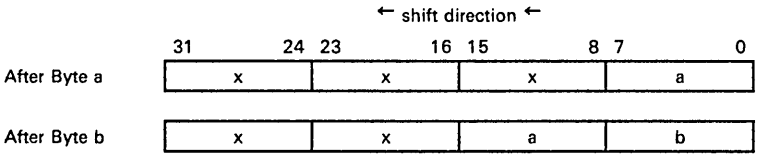


Figure 9-16. Receive Buffer Shift Operation

9.2.9 Serial Port Operation Configurations

Several configurations are provided for the operation of the serial port clocks and timer. The clocks for each serial port can originate either internally or externally. Figure 9-17 shows serial port clocking in the I/O mode (FUNC = 0) when CLKX is either an input or an output. Figure 9-18 shows clocking in the serial port mode (FUNC = 1). Both figures use a transmit section for an example. The same relationship holds for a receive section.

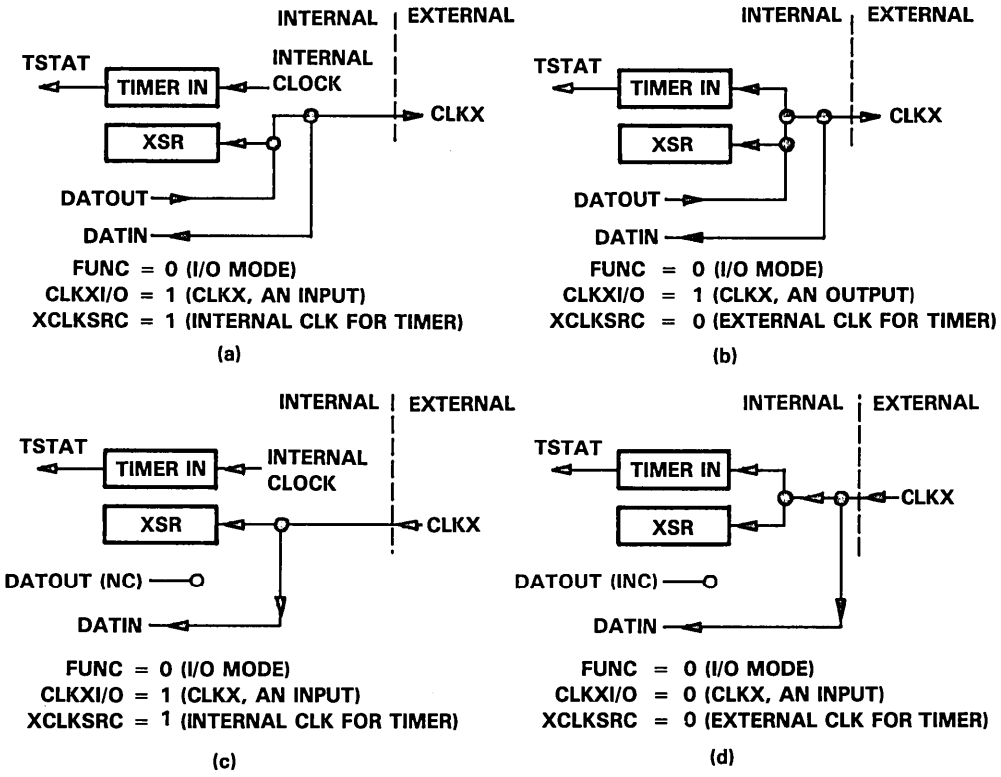


Figure 9-17. Serial Port Clocking in I/O Mode

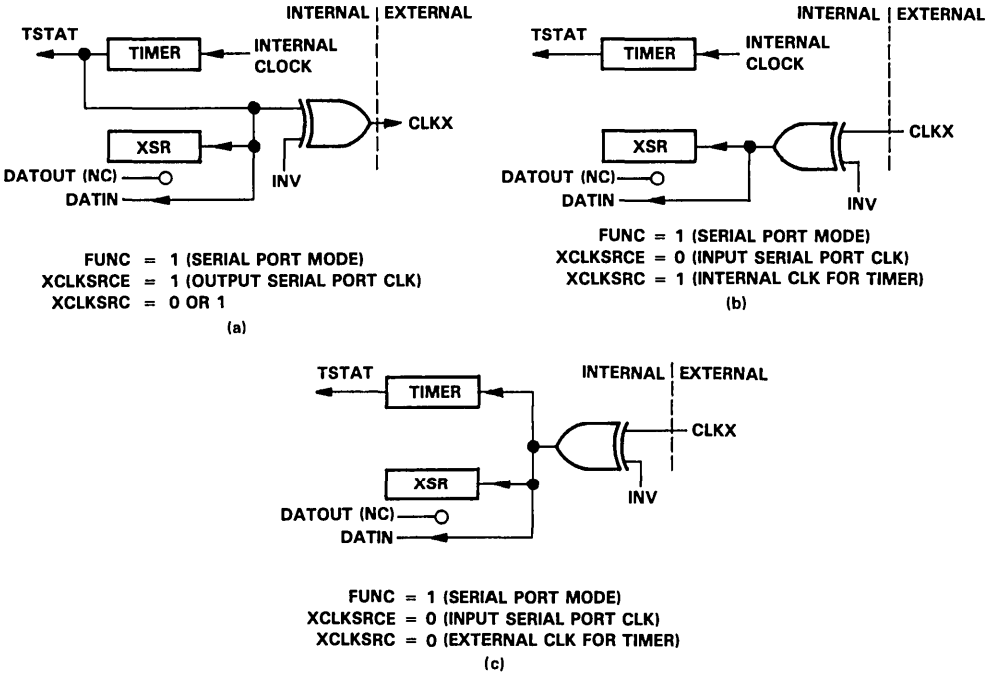


Figure 9-18. Serial Port Clocking in Serial Port Mode

9.2.10 Serial Port Timing

The formula for calculating the frequency of the serial port clock with an internally generated clock is dependent upon the operation mode of the serial port timers, defined as:

$$f \text{ (pulse mode)} = f \text{ (timer clock)}/\text{period register}$$

$$f \text{ (clock mode)} = f \text{ (timer clock)}/(2 \times \text{period register})$$

An externally generated serial port clock (CLKX or CLKR) has a maximum frequency less than f(H1)/2.6. See serial port timing in Appendix A.

Transmit data is clocked out on the rising edge of the selected serial port clock. Receive data is latched into the receive shift register on the falling edge of the serial port clock. All data is transmitted and loaded MSB first and right-justified. If less than 32 bits are transferred, the data is right-justified in the 32-bit transmit and receive buffers. Therefore, the LSBs of the transmit buffer are the bits that are transmitted.

The transmit ready (XRDY) signal specifies that the data transmit register (DXR) is available to be loaded with new data. XRDY goes active as soon as the data is loaded into the transmit shift register (XSR). The last word may still be shifting out when XRDY goes active. If DXR is loaded before the last word has completed transmission, the data bits transmitted will be consecutive, i.e., the LSB of the first word immediately precedes the MSB of the second, with all signalling valid as in two separate transmits. XRDY goes inactive when DXR is loaded, and remains inactive until the data is loaded into the shifter.

The receive ready (RRDY) signal is active as long as a new word of data is loaded into the data receive register and has not been read. As soon as the data is read, the RRDY bit is turned off.

When FSX is specified as an output, the activity of the signal is determined solely by the internal state of the serial port. When a fixed data rate is specified, FSX goes active when DXR is loaded into XSR to be transmitted out. One serial clock cycle later, FSX turns inactive and data transmission begins. When a variable data rate is specified, the FSX pin is activated when the data transmission begins, and remains active during the entire transmission of the word. Again, the data is transmitted one clock cycle after it is loaded into the data transmit register.

An input FSX in the fixed data rate mode should go active for at least one serial clock cycle and then inactive to initiate the data transfer. The transmitter then transmits the number of bits specified by the LEN bits. In the variable data rate mode, the transmitter begins transmitting as soon as FSX goes active until the number of specified bits has been shifted out. In the variable data rate mode, when the FSX status changes prior to all the data bits being shifted out, the transmission completes and the DX pin is placed in a high impedance state. An FSR input is exactly complementary to the FSX.

When using an external FSX, if DXR and XSR are empty, a write to DXR results in a DXR to XSR transfer. This data is held in the XSR until an FSX occurs. When the external FSX is received, the XSR begins shifting the data. If XSR is waiting for the external FSX, a write to DXR will change DXR, but a

DXR to XSR transfer will not occur. XSR begins shifting when the external FSX is received, or when reset using XRESET.

Continuous Transmit and Receive Modes

When continuous mode is chosen, consecutive writes do not generate or expect new sync pulse signalling. Only the first word of a block begins with an active synchronization. Thereafter, data continues to be transmitted as long as new data is loaded into DXR before the last word has been transmitted. As soon as TXRDY is active and all of the data has been transmitted out of the shift register, the DX pin is placed in a high impedance state, and a subsequent write to DXR initiates a new block and a new FSX.

Similarly with FSR, the receiver continues shifting in new data and loading DRR. If the data receive buffer is not read before the next word is shifted in, subsequent incoming data will be lost. The RFSM bit can be used to terminate the receive continuous mode.

Handshake Mode

The handshake mode (HS = 1) allows for direct connection between processors. In this mode, all data words are transmitted with a leading 1 (see Figure 9-19). For example, if an 8-bit word is to be transmitted, the first bit sent is a 1, followed by the 8-bit data word.

In this mode, once the serial port transmits a word, it will not transmit another word until it receives a separately transmitted zero bit. Therefore, the 1 bit that precedes every data word is, in effect, a request bit.

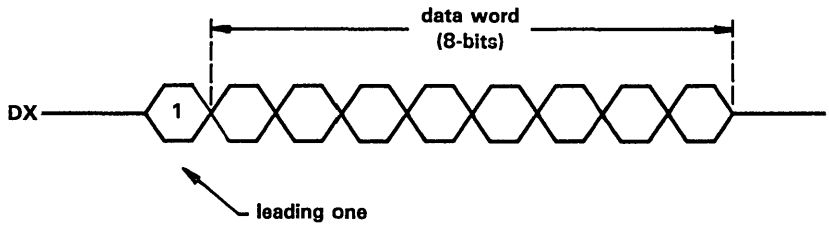


Figure 9-19. Data Word Format in Handshake Mode

After a serial port receives a word (with the leading 1), and it has been read from the DRR, it sends a single 0 to the transmitting serial port. Thus, the single 0 bit acts as an acknowledge bit (see Figure 9-20). This single acknowledge bit is sent every time the DRR is read, even if the DRR does not contain new data.

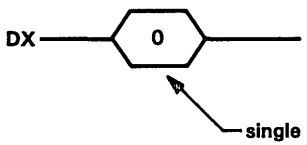


Figure 9-20. Single Zero Sent as an Acknowledge

When the serial port is placed in the handshake mode, the insertion and deletion of a leading 1 for transmitted data, the sending of a 0 for acknowledgement of received data, and the waiting for this acknowledge bit are all performed automatically. Using this scheme, it is simple to connect processors with no external hardware and guarantee secure communication. A typical configuration is shown in Figure 9-21.

In the handshake mode, FSX is automatically configured as an output. Continuous mode is automatically disabled. After a system reset or XRESET, the transmitter is always permitted to transmit. The transmitter and receiver must be reset when entering the handshake mode.

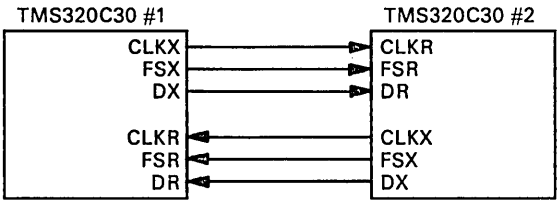


Figure 9-21. Direct Connection Using Handshake Mode

9.2.11 Serial Port Interrupt Sources

A serial port has four interrupt sources:

1) The transmit timer interrupt: The rising edge of XTSTAT causes a single cycle interrupt pulse to occur. When XTINT is 0, this interrupt pulse is disabled.

2) The receive timer interrupt: The rising edge of RTSTAT causes a single cycle interrupt pulse to occur. When RTINT is 0, this interrupt pulse is disabled.

3) The transmitter interrupt: Occurs immediately following a DXR to XSR transfer. The transmitter interrupt is a single cycle pulse. When the global serial-port control register XINT is 0, this interrupt pulse is disabled.

4) The receiver interrupt: Occurs immediately following a RSR to DRR transfer. The receiver interrupt is a single cycle pulse. When the global serial-port control register RINT is 0, this interrupt pulse is disabled.

The transmit timer interrupt pulse is ORed with the transmitter interrupt pulse to create the CPU transmit interrupt flag XINT. The receive timer interrupt pulse is ORed with the receiver interrupt pulse to create the CPU receive interrupt flag RINT.

9.2.12 Serial Port Functional Operation

The following paragraphs and figures illustrate the functional timing of the various serial port modes of operation. The timing descriptions are presented assuming that all signal polarities are configured to be positive, i.e. CLKXP=CLKRP= DXP=DRP=FSXP=FSRP=0. Logical timing, in situations where one or more of these polarities are inverted, is the same but with respect to the opposite polarity reference points, i.e. rising vs. falling edges, etc.

These discussions pertain to the numerous operating modes and configurations of the serial port logic. When it is necessary to switch operating modes or change configurations of the serial port, this should be done only when $\overline{\text{XRESET}}$ or or $\overline{\text{RRESET}}$ are asserted (low) as appropriate. Therefore, when transmit configurations are modified, $\overline{\text{XRESET}}$ should be low, and when receive configurations are modified, $\overline{\text{RRESET}}$ should be low. When in handshake

mode, however, since the transmitter and receiver are interrelated, any configuration changes should be made with $\overline{\text{XRESET}}$ and $\overline{\text{RRESET}}$ both low.

All of the various serial port operating configurations can be broadly classified in two categories: fixed data rate timing and variable data rate timing. The following paragraphs discuss fixed and variable data rate operation and all of their variations.

Fixed Data Rate Timing Operation

Fixed data rate serial port transfers can occur in two varieties: burst mode and continuous mode. In burst mode operation, transfers of single words are separated by periods of inactivity on the serial port. In continuous mode, there are no gaps between successive word tranfers, i.e., the first bit of a new word is transferred on the next CLKX/R pulse following the last bit of the previous word. This occurs continuously until the process is terminated.

In burst mode with fixed data rate timing, FSX/FSR pulses initiate transfers, and each transfer involves a single word. With an internally generated FSX (see Figure 9-22), transmission is initiated by loading DXR. In this mode, there is an approximately 2.5 CLKX cycle delay (depending on CLKX and H1 frequencies) from DXR being loaded until FSX occurs. With an external FSX, the FSX pulse initiates the transfer and the 2.5 cycle delay effectively becomes a setup requirement for loading DXR with respect to FSX. Therefore, in this case, DXR must be loaded no later than 3 CLKX cycles before FSX occurs. Once the XSR is loaded from the DXR, an XINT is generated.

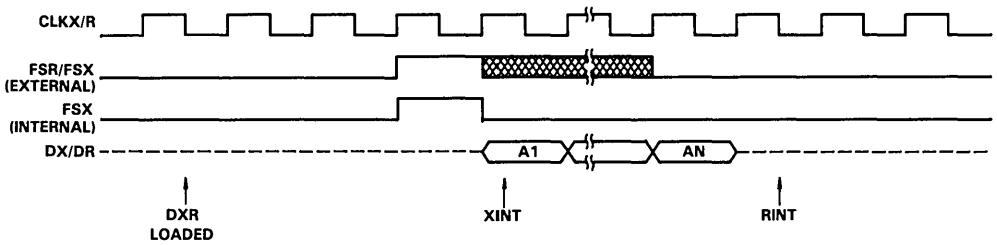


Figure 9-22. Fixed Burst Mode

In receive operations, once a transfer is initiated, FSR is ignored until the last bit. For burst mode transfers, FSR must be low during the last bit, or another transfer will be initiated. After a full word has been received and transfered to the DRR, an RINT is generated.

In fixed data rate mode, continuous transfers may be performed even if R/XFSM=0 as long as properly timed frame synchronization is provided, or if DXR is reloaded each cycle (with an internally generated FSX), see Figure 9-23.

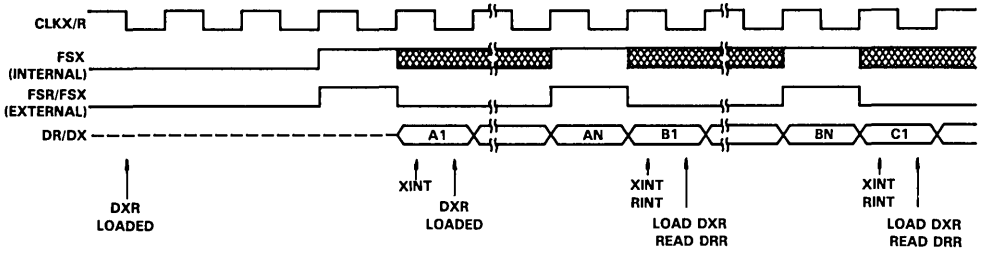


Figure 9-23. Fixed Continuous Mode With Frame Synch

For receive operations and with externally generated FSX, once transfers have begun, frame sync pulses are only required during the last bit transferred to initiate another contiguous transfer. Otherwise, frame sync inputs are ignored. Therefore, continuous transfers will occur if frame sync is held high. With an internally generated FSX, there is an approximately 2.5 CLKX cycle delay following DXR being loaded before FSX occurs. This delay occurs each time DXR is loaded, therefore, during continuous transmission, the instruction which loads DXR must be executed by the N-3 bit, (for an N-bit transmission). Since delays due to pipelining may vary, a conservative margin of safety should be incorporated in accounting for this delay.

Once the process begins, an XINT and an RINT are generated at the beginning of each transfer. The XINT indicates that the XSR has been loaded from DXR, and can be used to cause DXR to be reloaded. To maintain continuous transmission in this mode, especially with an interally generated FSX, DXR must be reloaded early in the ongoing transfer.

The RINT indicates that a full word has been received and transferred into the DRR. RINT is therefore commonly used to indicate an appropriate time to read DRR.

Continuous transfers are terminated by discontinuing frame sync pulses or, in the case of internally generated FSX, not reloading DXR.

Continuous serial port transfers can be accomplished without the use of frame sync pulses if R/XFSM are set to one. In this mode, operation of the serial port is similar to continuous operation with frame sync except that a frame sync pulse is involved only in the first word transferred, and no further frame sync pulses are used. Following the first word transferred (see Figure 9-24), no internal frame sync pulses are generated, and frame sync inputs are ignored. Additionally, R/XFSM should be set prior to or during the first word transferred, and must be set no later than the transfer of the N-1 bit of the first

word, except for transmit operations. For transmit operations in the fixed data rate mode, XFSM must be set no later than the N-2 bit. Clearing R/XFSM must be performed no later than the N-1 bit to be recognized in the current cycle.

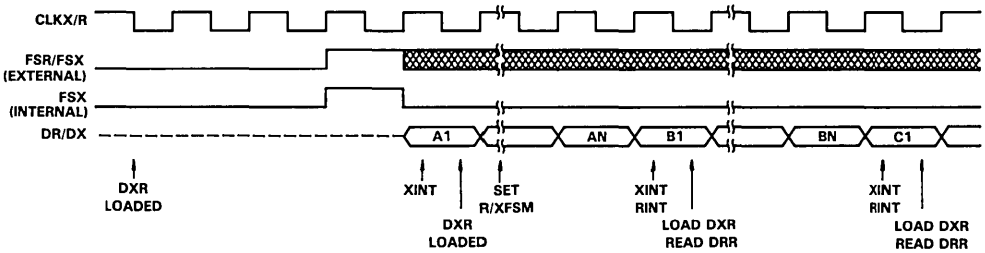


Figure 9-24. Fixed Continuous Mode Without Frame Synch

Timing of RINT and XINT and data transfers to and from DXR and DRR, respectively, are the same as in fixed data rate continuous mode with frame sync. This mode of operation also exhibits the same 2.5 CLKX cycle delay following DXR being loaded before an internal FSX is generated. As in the case of continuous operation in fixed data rate mode with frame sync, DXR must be reloaded no later than transmission of the N-3 bit.

When using continuous operation in fixed data rate mode, R/XFSM may be set and cleared as desired, even during active transfers, to enable or disable the use of frame sync pulses as dictated by system requirements. Under most conditions, the effect of changing the state of R/XFSM occurs during the transfer in which the R/XFSM change was made, provided the change was made early enough in the transfer. For transmit operations with internal FSX in fixed data rate mode, however, a one word delay occurs before frame sync pulse generation resumes when clearing XFSM to zero (see Figure 9-25). Therefore, one additional word is transferred in this case before the next FSX pulse is generated. Also note that, as discussed previously, clearing XFSM will be recognized during the current word being transmitted as long as XFSM is cleared no later than the N-1 bit. Setting XFSM is recognized as long as XFSM is set no later than the N-2 bit.

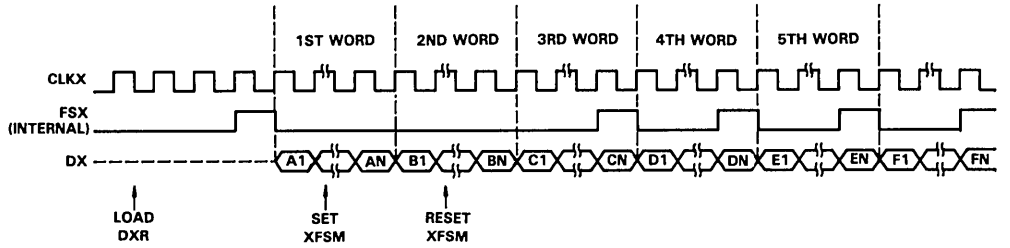


Figure 9-25. Exiting Fixed Continuous Mode Without Frame Synch, FSX Internal

Variable Data Rate Timing Operation

Variable data rate timing also supports operation in either burst or continuous mode. Burst mode operation with variable data rate timing is similar to burst mode operation with fixed data rate timing. With variable data rate timing (see Figure 9-26) however, FSX/R and data timing differs slightly at the beginning and end of transfers. Specifically, there are three major differences between fixed and variable data rate timing.

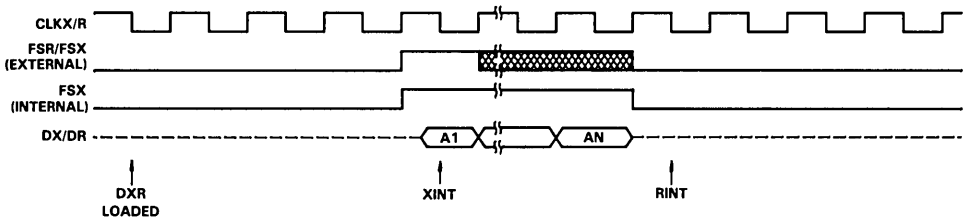


Figure 9-26. Variable Burst Mode

First, FSX/R pulses typically last for the entire transfer interval, although FSR and external FSX are ignored after the first bit transferred. FSX/R pulses in fixed data rate mode typically last only one CLKX/R cycle, but can last longer.

Second, data transfer begins during the CLKX/R cycle in which FSX/R occurs, rather than the CLKX/R cycle following FSX/R, as is the case with fixed data rate timing.

Finally, with variable data rate timing, frame sync inputs are ignored until the end of the last bit transferred, rather than the beginning of the last bit transferred as is the case with fixed data rate timing.

When transmitting continuously in variable data rate mode with frame sync, timing is the same as for fixed data rate mode, besides the differences between these two modes as described under burst mode operation with variable data rate timing. The only exception to this is that when operating continuously in variable data rate mode (see Figure 9-27), DXR must be reloaded no later than the N-4 bit to maintain continuous operation, as opposed to the N-3 bit for fixed data rate mode.

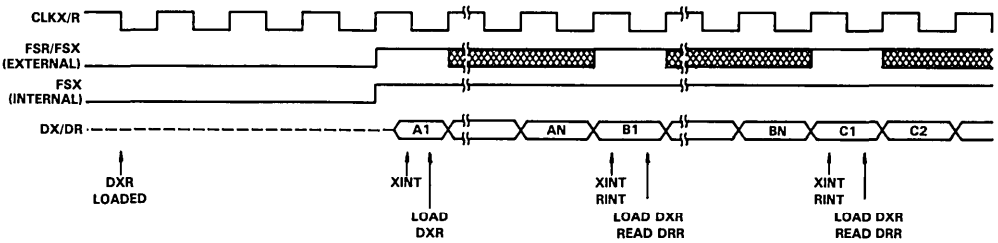


Figure 9-27. Variable Continuous Mode With Frame Synch

9

Continuous operation in variable data rate mode without frame sync is also similar to continuous operation without frame sync in fixed data rate mode. As with variable data rate mode continuous operation with frame sync (see Figure 9-28), DXR must be reloaded no later than the N-4 bit to maintain continuous operation. Additionally, when R/XFSM is set or cleared in the variable data rate mode, the modification must be made no later than the N-1 bit, for the result to be affected in the current transfer.

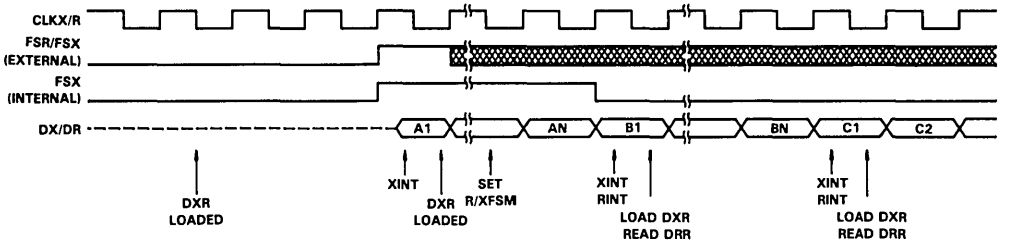


Figure 9-28. Variable Continuous Mode Without Frame Synch

9.3 DMA Controller

The TMS320C30 provides an on-chip Direct Memory Access (DMA) controller. The purpose of the DMA controller is to reduce the need for the CPU to perform input/output functions. The DMA controller can perform input/output operations without interfering with the operation of the CPU. Therefore, it is possible to interface the TMS320C30 to slow external memories and peripherals (A/D's, serial ports, etc.) without reducing the computational throughput of the CPU. The result is improved system performance and decreased system cost.

A DMA transfer consists of two operations: a read from a memory location and a write to a memory location. The DMA controller can read from and write to any location in the TMS320C30 memory map. This includes all memory-mapped peripherals. The operation of the DMA is controlled with the following set of memory-mapped registers:

- DMA global control register
- DMA source address register
- DMA destination address register
- DMA transfer counter register

These registers, their memory-mapped addresses, and functions are shown in Figure 9-29. Each of these DMA registers will be discussed in the succeeding subsections.

Register	Peripheral Address
DMA GLOBAL CONTROL	808000h
RESERVED	808001h
RESERVED	808002h
RESERVED	808003h
DMA SOURCE ADDRESS	808004h
RESERVED	808005h
DMA DESTINATION ADDRESS	808006h
RESERVED	808007h
DMA TRANSFER COUNTER	808008h
RESERVED	808009h
RESERVED	80800Ah
RESERVED	80800Bh
RESERVED	80800Ch
RESERVED	80800Dh
RESERVED	80800Eh
RESERVED	80800Fh

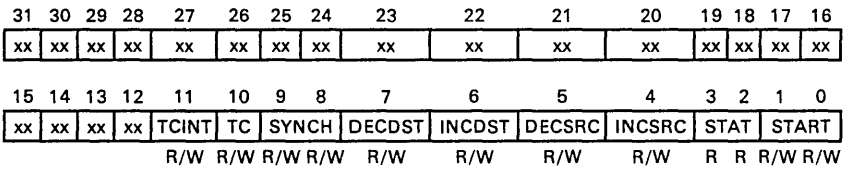
Figure 9-29. Memory-Mapped Locations for a DMA Channel

9.3.1 DMA Global Control Register

The global control register controls the state in which the DMA controller operates. This register also indicates the status of the DMA, which changes every cycle. Source and destination addresses can be incremented, decremented, or synchronized using specified global control register bits. At system reset, all bits in the DMA control register are set to 0. Table 9-7 lists the register bits, names, and functions. Figure 9-30 shows the bit configuration of the global control register.

Table 9-7. Global Control Register Bits

BIT	NAME	FUNCTION
0-1	START	These bits control the state in which the DMA starts and stops. The DMA may be stopped without any loss of data (see Table 9-8).
2-3	STAT	These bits indicate the status of the DMA. These status bits change every cycle (see Table 9-9).
4	INCSRC	If INCSRC = 1, the source address is incremented after every read.
5	DECSRC	If DECSRC = 1, the source address is decremented after every read. If INCSRC = DECSRC, the source address is not modified after a read.
6	INCDST	If INCDST = 1, the destination address is incremented after every write.
7	DECDST	If DECDST = 1, the destination address is decremented after every read. If INCDST = DECDST, the destination address is not modified after a write.
8-9	SYNCH	The SYNCH bits determine the timing synchronization between the events initiating the source and the destination transfers. The interpretation of the SYNCH bits is shown in Table 9-10.
10	TC	The TC bit affects the operation of the transfer counter. If TC = 0, transfers are not terminated when the transfer counter becomes zero. If TC = 1, transfers are terminated when the transfer counter becomes zero.
11	TCINT	If TCINT = 1,the DMA interrupt is set when the transfer counter makes a transition to zero. If TCINT = 0, the DMA interrupt is not set when the transfer counter makes a transition to zero.
12	Reserved	Read as zero.



NOTE: xx = Reserved bit, read as 0.
R = read, W = write.

Figure 9-30. DMA Global Control Register

Table 9-8. START Bits and Operation of the DMA

START	FUNCTION
0 0	DMA read or write cycles in progress will be completed, any data read will be ignored. Any pending read or write will be cancelled. The DMA is reset so that when started, a new transaction is begun; i.e., a read is performed.
0 1	If a read or write has begun, the read or write is completed before stopping, i.e. in the middle or at the end of a DMA transfer. If a read or write has not begun, no read or write is started.
1 0	If a DMA transfer has begun, the entire transfer is completed (including both read and write operations) before stopping. If a transfer has not begun, none is started.
1 1	DMA starts from reset or restarts from the previous state.

Table 9-9. STAT Bits and Status of the DMA

STAT	FUNCTION
0 0	DMA is being held between DMA transfers (between a read and write). This is the value at reset.
0 1	DMA is being held in the middle of a DMA transfer, i.e. between a read and a write.
1 0	Reserved.
1 1	DMA busy; i.e., DMA is performing a read or write.

Table 9-10. SYNCH Bits and Synchronization of the DMA

SYNCH	FUNCTION
0 0	No synchronization. Enabled interrupts are ignored.
0 1	Source synchronization. A read is performed when an enabled interrupt occurs.
1 0	Destination synchronization. A write is performed when an enabled interrupt occurs.
1 1	Source and destination synchronization. A read is performed when an enabled interrupt occurs. A write is then performed when the next enabled interrupt occurs.

9.3.2 Destination and Source Address Registers

The DMA destination and source address registers are 24-bit registers. These registers are used when performing the increment and decrement as specified by control bits DECSRC, INCSRC, DECDST, and INCDST of the DMA global control register. The contents of these registers specify the destination and source addresses. The registers are incremented or decremented at the end of the corresponding memory access, i.e., source register for a read, destination register for a write. On system reset, 0 is written to these registers.

9.3.3 Transfer Counter Register

The transfer counter register is a 24-bit register, controlled by a 24-bit counter that counts down. The counter decrements upon the completion of a DMA memory write. In this way, it can be used to control the size of a block of data transferred. The transfer counter register is set to 0 at system reset.

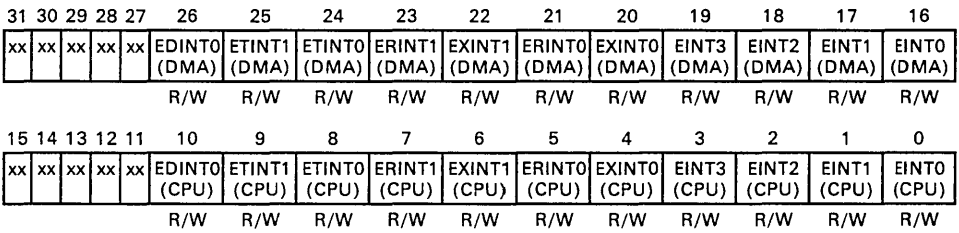
9.3.4 CPU/DMA Interrupt Enable Register

The CPU/DMA interrupt enable register (IE) is a 32-bit register located in the CPU register file. The CPU interrupt enable bits are in locations 10-0. The DMA interrupt enable bits are in locations 26-16. A 1 in a CPU/DMA interrupt enable register bit enables the corresponding interrupt. A 0 disables the corresponding interrupt. At reset, 0 is written to this register.

Table 9-11 list the bits, names, and functions of the CPU/DMA interrupt enable register. Figure 9-31 shows the IE register. The priority and decoding scheme of CPU and DMA interrupts is identical. Note that when the DMA receives an interrupt, this interrupt is acted upon based upon the SYNCH field of the DMA control register. Note that an interrupt may affect the DMA, but not the CPU and vice versa. Refer to Section 7.

Table 9-11. CPU/DMA Interrupt Enable Register Bits

BIT	NAME	FUNCTION
0	EINT0	Enable external interrupt 0 (CPU)
1	EINT1	Enable external interrupt 1 (CPU)
2	EINT2	Enable external interrupt 2 (CPU)
3	EINT3	Enable external interrupt 3 (CPU)
4	EXINT0	Enable serial port 0 transmit interrupt (CPU)
5	ERINT0	Enable serial port 0 receive interrupt (CPU)
6	EXINT1	Enable serial port 1 transmit interrupt (CPU)
7	ERINT1	Enable serial port 1 receive interrupt (CPU)
8	ETINT0	Enable timer 0 interrupt (CPU)
9	ETINT1	Enable timer 1 interrupt (CPU)
10	EDINT	Enable DMA controller interrupt (CPU)
11-15	Reserved	Read as 0
16	EINT0	Enable external interrupt 0 (DMA)
17	EINT1	Enable external interrupt 1 (DMA)
18	EINT2	Enable external interrupt 2 (DMA)
19	EINT3	Enable external interrupt 3 (DMA)
20	EXINT0	Enable serial port 0 transmit interrupt (DMA)
21	ERINT0	Enable serial port 0 receive interrupt (DMA)
22	EXINT1	Enable serial port 1 transmit interrupt (DMA)
23	ERINT1	Enable serial port 1 receive interrupt (DMA)
24	ETINT0	Enable timer 0 interrupt (DMA)
25	ETINT1	Enable timer 1 interrupt (DMA)
26	EDINT	Enable DMA controller interrupt (DMA)
27-32	Reserved	Read as 0



NOTE: xx = Reserved bit, read as 0.
R = read, W = write.

Figure 9-31. CPU/DMA Interrupt Enable Register

9.3.5 DMA Memory Transfer Operation

Each DMA memory transfer consists of two parts:

1) Read data from the address specified by the DMA source register.
2) Write data that has been read to the address specified by the DMA destination register.

A transfer is complete only when the read and write are complete. A transfer may be stopped by setting the START bits to the desired value. When the DMA is restarted (START = 1 1), it completes any pending transfer.

At the end of a DMA read, the source address is modified as specified by the SRCINC and SRCDEC bits of the DMA global control register. At the end of a DMA write, the destination address is modified as specified by the DSTINC and DSTDEC bits of the DMA global control register. At the end of every DMA write, the DMA transfer counter is decremented.

DMA on-chip reads and writes (reads and writes from on-chip memory and peripherals) are single cycle. DMA off-chip reads are two cycles. The first cycle is an internal setup with the external read beginning on the following cycle. The external read cycle is identifical to a CPU read cycle. DMA off-chip writes are identical to CPU off-chip writes.

Through the 24-bit source and destination registers, the DMA is capable of accessing any memory-mapped location in the TMS320C30 memory map. Figure 9-32 through Figure 9-34 show the number of cycles a DMA transfer requires, depending upon whether the source and destination are on-chip memory and peripherals, the external port, or the I/O port. T represents the number of transfers to be performed. C_r represents the number of wait states for the source read. C_w represents the number of wait states for the destination write. Each entry in the table represents the total cycles required to do the T transfers, assuming no pipeline conflicts.

Accompanying each table is a figure illustrating the timing of the DMA transfer. |R| and |W| represent single-cycle reads and writes, respectively. |R.R| and |W.W| represent multicycle reads and writes. |Cr| and |Cw| show the number of wait cycles for a read and write. |–| represents the cycle used as an internal setup for DMA external reads.

CYCLE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Source On-Chip	R		R		R														
Dest On-Chip		W		W		W													
Source Primary Bus	–	R	R	R		–	R	R	R		–	R	R	R					
			C_r					C_r					C_r						
Dest On-Chip					W					W					W				
Source Expansion Bus	–	R	R	R		–	R	R	R		–	R	R	R					
			C_r					C_r					C_r						
Dest On-Chip					W					W					W				

SOURCE	DESTINATION ON-CHIP
On-Chip	$(1+1)T$
Primary Bus	$(2+C_r+1)T$
Expansion Bus	$(2+C_r+1)T$

Figure 9-32. Timing and Number of Cycles for DMA Transfers When Destination is On-Chip

CYCLE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Source On-Chip	R		R				R				R								
Dest Primary bus		W	W	W	W	W	W	W	W	W	W	W	W						
				C_w				C_w				C_w							
Source Primary Bus	–	R	R	R					–	R	R	R							
			C_r								C_r								
Dest Primary Bus					W	W	W	W					W	W	W	W			
							C_w								C_w				
Source Expansion Bus	–	R	R	R		–	R	R	R		–	R	R	R					
			C_r					C_r					C_r						
Dest Primary Bus					W	W	W	W		W	W	W	W		W	W	W	W	
							C_w					C_w					C_w		

SOURCE	DESTINATION PRIMARY BUS
On-Chip	$1+(2+C_w)T$
Primary Bus	$(2+C_r+2+C_w)T$
Expansion Bus	$(2+C_r+2C_w)$ $+(2+C_w+\max(0,C_r-C_w+1)(T-1)$

Figure 9-33. DMA Timing When Destination is a Primary Bus

CYCLE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Source On-Chip	R		R				R												
Dest Expansion Bus		W	W	W	W	W	W	W	W	W	W	W	W						
				C_w				C_w				C_w							
Source Primary Bus	–	R	R	R		–	R	R	R		–	R	R	R					
			C_r					C_r					C_r						
Dest Expansion Bus					W	W	W	W		W	W	W	W		W	W	W	W	
							C_w					C_w					C_w		
Source Expansion Bus	–	R	R	R					–	R	R	R							
			C_r								C_r								
Dest Expansion Bus					W	W	W	W					W	W	W	W			
							C_w								C_w				

SOURCE	DESTINATION EXPANSION BUS
On–Chip	$1+(2+C_w)T$
Primary Bus	$(2+C_r+2+C_w)$ $+(2+C_w+\max(0,C_r-C_w+1))(T-1)$
Expansion Bus	$(2+C_r+2+C_w)T$

Figure 9-34. DMA Timing When Destination is an Expansion Bus

Table 9-12 shows the maximum DMA transfer-rates assuming no wait states (C_r=C_w= 0). Table 9-13 shows the maximum DMA transfer-rates assuming one wait state for the read (C_r=1) and no wait states for the write (C_w=0). Table 9-14 shows the maximum DMA transfer-rates assuming one wait state for the read (C_r =1) and one wait state for the write (C_w).

In each table, the complete transfer is considered (i.e., the time to do the read and the write). Since one bus access is required for the read and another for the write, bus transfer-rates will be twice the transfer-rate. It is also assumed that no conflicts with the CPU exist.

Table 9-12. Maximum DMA Transfer Rates When C_r = C_w = 0

SOURCE	DESTINATION		
	INTERNAL	PRIMARY	EXPANSION
INTERNAL	33.3 Mbytes/sec	33.3 Mbytes/sec	33.3 Mbytes/sec
PRIMARY	22.2 Mbytes/sec	16.7 Mbytes/sec	22.2 Mbytes/sec
EXPANSION	22.2 Mbytes/sec	22.2 Mbytes/sec	16.7 Mbytes/sec

Table 9-13. Maximum DMA Transfer Rates When C_r = 1, C_w = 0

SOURCE	DESTINATION		
	INTERNAL	PRIMARY	EXPANSION
INTERNAL	33.3 Mbytes/sec	33.3 Mbytes/sec	33.3 Mbytes/sec
PRIMARY	16.7 Mbytes/sec	13.3 Mbytes/sec	16.7 Mbytes/sec
EXPANSION	16.7 Mbytes/sec	16.7 Mbytes/sec	13.3 Mbytes/sec

Table 9-14. Maximum DMA Transfer Rates When C_r = 1, C_w = 1

SOURCE	DESTINATION		
	INTERNAL	PRIMARY	EXPANSION
INTERNAL	33.3 Mbytes/sec	22.2 Mbytes/sec	22.2 Mbytes/sec
PRIMARY	16.7 Mbytes/sec	11.1 Mbytes/sec	16.7 Mbytes/sec
EXPANSION	16.7 Mbytes/sec	16.7 Mbytes/sec	11.1 Mbytes/sec

9.3.6 Synchronization of DMA Channels

A DMA channel may be synchronized through the use of interrupts. Refer to Table 9-10 for the relationship between the SYNCH bits of the DMA global control register and the synchronization performed. This section describes the following four synchronization mechanisms:

- No synchronization (SYNCH = 0 0)
- Source synchronization (SYNCH = 0 1)
- Destination synchronization (SYNCH = 1 0)
- Source and destination synchronization (SYNCH = 1 1)

No Synchronization

When SYNCH = 0 0, no synchronization is performed. The DMA will perform reads and writes whenever there are no conflicts. All interrupts are ignored, and therefore can be considered to be globally disabled. However, no bits in the DMA interrupt enable register are changed. Figure 9-35 shows the synchronization mechanism when SYNCH = 0 0.

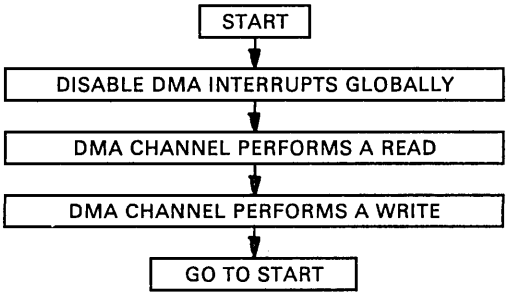


Figure 9-35. No DMA Synchronization

Source Synchronization

When SYNCH = 0 1, the DMA is synchronized to the source (see Figure 9-36). A read will not be performed until an interrupt is received by the DMA. Then, all DMA interrupts are disabled globally. However, no bits in the DMA interrupt enable register are changed.

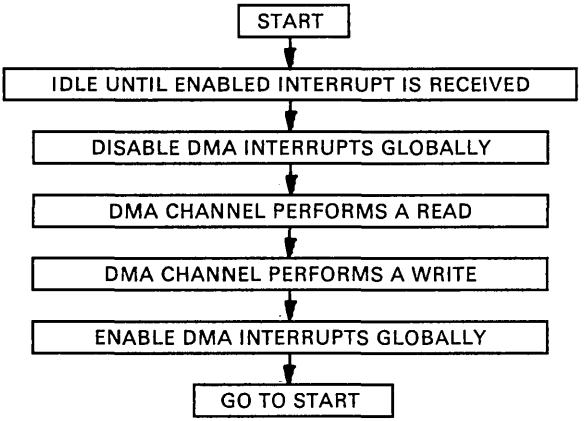


Figure 9-36. DMA Source Synchronization

Destination Synchronization

When SYNCH = 1 0, the DMA is synchronized to the destination. First, all interrupts are ignored until the read is complete. Though the DMA interrupts may be considered to be globally disabled, no bits in the DMA interrupt enable register are changed. A write will not be performed until an interrupt is received by the DMA. Figure 9-37 shows the synchronization mechanism when SYNCH = 1 0.

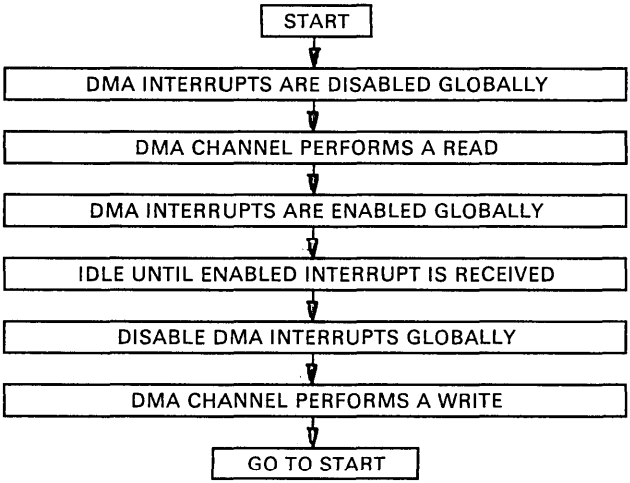


Figure 9-37. DMA Destination Synchronization

Source and Destination Synchronization

When SYNCH = 1 1, all interrupts are ignored, and therefore can be considered to be globally disabled. However, no bits in the DMA interrupt enable register are changed. A read is performed when an interrupt is received. A write is performed on the following interrupt. Source and destination synchronization when SYNCH = 1 1 is shown in Figure 9-38.

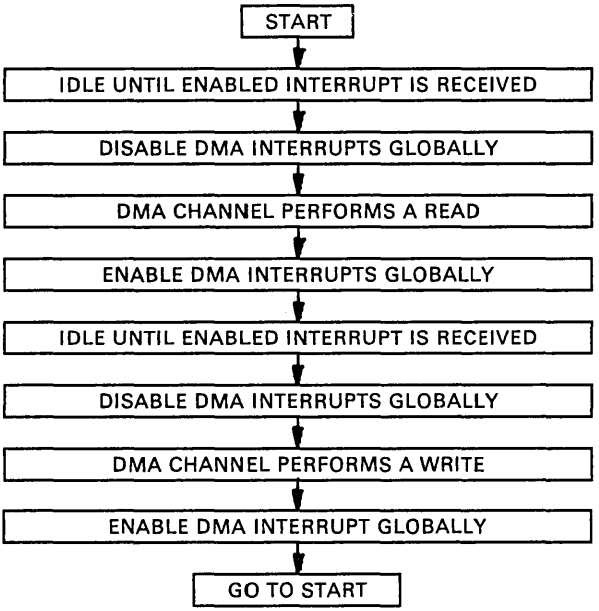


Figure 9-38. DMA Source and Destination Synchronization

Introduction 1

Pinout and Signal Descriptions 2

Architectural Overview 3

CPU Registers, Memory, and Cache 4

Data Formats and Floating-Point Operation 5

Addressing 6

Program Flow Control 7

External Bus Operation 8

Peripherals 9

Pipeline Operation 10

Assembly Language Instructions 11

Software Applications 12

Hardware Applications 13

Appendices A-D

Section 10

Pipeline Operation

TMS320C30 operation is controlled by five major functional units: fetch, decode, read, execute, and DMA. To provide for maximum processor throughput, these units can perform in parallel, with each unit operating on a different instruction. The overlapping of the fetch, decode, read, and execute operations of different instructions is called pipelining. The pipelining of these operations results in the high performance of the TMS320C30. The ability of the DMA to move data within the processor memory space results in an even greater utilization of the CPU with fewer interruptions of the pipeline, thus yielding greater performance.

Major topics discussed in this section are as follows:

- Pipeline Structure (Section 10.1 on page 10-2)
- Pipeline Conflicts (Section 10.2 on page 10-4)
  - Branch conflicts
  - Register conflicts
  - Memory conflicts
- Resolving Memory Conflicts (Section 10.3 on page 10-14)
- Clocking of Memory Accesses (Section 10.4 on page 10-16)
  - Program fetches
  - Data loads and stores
  - DMA accesses

10.1 Pipeline Structure

The five major units of the TMS320C30 pipeline structure and their function are as follows:

Fetch Unit (F) Fetches the instruction words from memory and updates the program counter (PC).

Decode Unit (D) Decodes the instruction word and performs address generation. Any modification of the auxiliary registers and the stack pointer is controlled by this unit.

Read Unit (R) If required, reads the operands from the register file, performs the necessary operation, and if needed writes results to the register file. If required, results of previous operations are written to memory.

Execute Unit (E) If required, reads the operands from the register file, performs the necessary operation, and if needed writes results to the register file. If required, results of previous operations are written to memory.

DMA Channel (DMA) Reads and writes memory.

The basic instruction has four levels: fetch, decode, read, and execute. Figure 10-1 illustrates these four levels of the pipeline structure. The levels are indexed according to instruction and execution cycle. Also indicated is a place in the pipeline where all four units operate in parallel; the perfect overlap occurs at cycle (*m*). Those levels about to be executed are at *m*+1, and those just executed are at *m*-1. The TMS320C30 pipeline control allows for an extremely high-speed execution rate by allowing an effective rate of one execution per cycle. It also manages pipeline conflicts in a way that makes them transparent to the user. The user does not need to take any special precautions to guarantee correct operation.

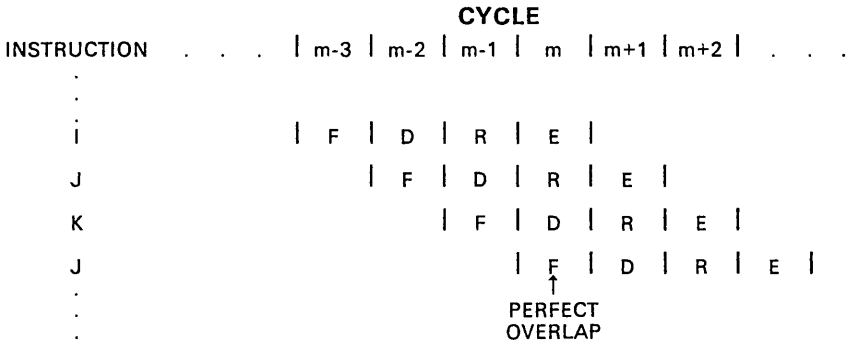


Figure 10-1. TMS320C30 Pipeline Structure

Priorities have been assigned to each of the functional units. The priorities from highest to lowest are:

- Execute (highest)
- Read
- Decode
- Fetch
- DMA (lowest).

When processing of an instruction is ready to pass to the next higher pipeline level, but that level is not ready to accept a new input, a pipeline conflict occurs. In this case, the lower priority unit waits until the higher priority unit completes its currently executing function.

Despite the DMA controllers low priority, conflicts with the CPU can be minimized or even eliminated by suitable data structuring since the DMA controller has its own data and address buses.

10.2 Pipeline Conflicts

The pipeline conflicts of the TMS320C30 can be grouped into the following main categories:

Branch Conflicts Involve most of those instructions or operations which read and/or modify the PC.

Register Conflicts Involve delays that can occur when reading or writing registers used for address generation.

Memory Conflicts Occur when the internal units of the TMS320C30 compete for memory resources.

Each of these three types is discussed in the following sections. Examples are included. Note in these examples, when data is refetched or an operation is repeated, the symbol representing the stage of the pipeline is appended with a number. For example, if a fetch is performed again, the initial fetch is labeled F1 and the refetch is labeled F2. When an access is detained multiple cycles due to a 'not ready,' the symbols $\overline{\text{RDY}}$ and RDY are used to indicate not ready and ready, respectively.

10.2.1 Branch Conflicts

The first class of pipeline conflicts is that which occurs with standard (non-delayed) branches, i.e., BR, B*cond*, DB*cond*, CALL, IDLE, RPTB, RPTS, RETI*cond*, RETS*cond*, interrupts, and reset. Conflicts arise with these instructions and operations since during their execution, the pipeline is used only for the completion of the operation; other information fetched into the pipeline is discarded or refetched, or the pipeline is inactive. This is referred to as flushing the pipeline. Flushing the pipeline is necessary in these cases to guarantee that portions of succeeding instructions do not inadvertantly get partially executed. TRAP*cond* and CALL*cond* are classified somewhat differently from the other types of branches and are considered later.

Example 10-1 shows the code and pipeline operation for a standard branch. Note that one dummy fetch is performed (F1), and then after the branch address is available, a new fetch (F2) is performed. This dummy fetch will affect the cache.

Example 10-1. Standard Branch

```
        BR     THREE      ; Unconditional branch
        MPYF              ; Not executed
        ADDF              ; Not executed
        SUBF              ; Not executed
        AND               ; Not executed
         .
         .
         .
THREE   OR                ; Fetched after BR is fetched
        STI
         .
         .
```

PIPELINE OPERATION

				THREE→PC			
BR THREE	F	D	R	E			
OR		F1	(nop)	(nop)	F2	D	|...
STI						F	|...

RPTS and RPTB both flush the pipeline, thus allowing for the RS, RE, and RC registers to be loaded at the proper time relative to the flow of the pipeline. If these registers are loaded without the use of RPTS or RPTB, no flushing of the pipeline occurs. If none of the repeat modes are being used, RS, RE, and RC may be used as general-purpose 32-bit registers without any pipeline conflicts occurring. In cases such as the nesting of RPTB due to nested interrupts, it may be necessary to load and store these registers directly while using the repeat modes. Since up to four instructions can be fetched before entering the repeat mode, loads should be followed by a branch to flush the pipeline. If the RC is changing when an instruction is loading it, the direct load takes priority over the modification made by the repeat mode logic.

Delayed branches are implemented to guarantee the fetching of the next three instructions. The delayed branches include BRD, B*cond*D, and DB*cond*D. Example 10-2 shows the code and pipeline operation for a delayed branch.

Example 10-2. Delayed Branch

```
        BRD    THREE      ; Unconditional delayed branch
        MPYF              ; Executed
        ADDF              ; Executed
        SUBF              ; Executed
        AND               ; Not executed
         .
         .
         .
THREE   MPYF              ; Fetched after SUBF fetched
         .
         .
         .
```

PIPELINE OPERATION

```
                                        THREE→PC
BRD THREE   | F | D | R | E |
MPYF            | F | D | R | E |
ADDF                | F | D | R | E |
SUBF                    | F | D | R |...
MPYF                        | F | D |...
```

10.2.2 Register Conflicts

Register conflicts involve the reading or writing of registers used for addressing purposes. These conflicts occur when the pertinent register is not ready to be used. The registers comprise the following three functional groups:

Group 1 Auxiliary registers (AR0-AR7), index registers (IR0, IR1), and block size register (BK)

Group 2 Data page pointer (DP)

Group 3 System stack pointer (SP)

If an instruction writes to one of these three groups, the use of any register within that particular group by the decode unit is delayed until the write is complete, i.e. instruction execution is completed. In Example 10-3, an auxiliary register is loaded, and a different auxiliary register is used on the next instruction. Since the decode stage needs the result of the write to the auxiliary register, the decode of this second instruction is delayed two cycles. Every time the decode is delayed, a refetch of the program word is performed; i.e., the first fetch of ADDF is at F1, followed by F2 and F3 (the final fetch). Since these are actual refetches, they can cause conflicts with the DMA controller and cache hits and misses.

Example 10-3. Write to an AR Followed by an AR for Address Generation

```
      LDI     7,AR1       ; 7 → AR1
NEXT  MPYF    *AR2,R0     ; Decode delayed 2 cycles
      ADDF
      FLOAT
```

PIPELINE OPERATION

```
                                          7→AR1
LDI 7,AR1     | F | D | R |  E  |
MPYF *AR2,R0      | F | D1 | D2 | D3 | R |...
ADDF                  | F1 | F2 | F3 | D |...
FLOAT                                  | F |...
```

The case for reads of these groups is similar to the case for writes. If an instruction must read a member of one of these groups, the use of that particular group by the decode for the following instruction is delayed until the read is complete. The registers are read at the start of the execute cycle and therefore require only a one cycle delay of the following decode. For four registers (IR0, IR1, BK, or DP) no delay is incurred. In all other cases, including the SP, the delay occurs. In Example 10-4, two auxiliary registers are added together with the result going to an extended-precision register. The next instruction uses a different auxiliary register as an address register.

Example 10-4. A Read of ARs Followed by ARs for Address Generation

```
      ADDI    AR0,AR1,R1  ; AR0 + AR1 → R1
NEXT  MPYF    *++AR2,R0   ; Decode delayed 1 cycle
      ADDF
      FLOAT
```

PIPELINE OPERATION

```
                                      AR0+AR1→R1
ADDI          | F | D | R |  E  |
MPYF* ++AR2,R0    | F | D1 | D2 | R | E |
ADDF                  | F1 | F2 | D | R |...
FLOAT                              | F | D |...
```

Note that while the DBR (decrement and branch) instruction does not use the auxiliary registers for addressing, its use of them as loop counters is treated as if it did. Therefore, the operation shown in the two previous examples can also occur for this instruction.

10.2.3 Memory Conflicts

Possible memory conflicts occur when the memory bandwidth of a physical memory space is exceeded. For example, RAM blocks 0 and 1 and the ROM block can support only two accesses every cycle. The external interface can support only one access per cycle. Some conditions under which memory conflicts can be easily avoided are discussed in Section 10.3.

Memory pipeline conflicts consist of the following four types:

Program Wait	A program fetch is prevented from beginning.
Program Fetch Incomplete	A program fetch has begun, but is not yet complete.
Execute Only	An instruction sequence requires three CPU-data accesses in a single cycle.
Hold Everything	A primary or expansion bus operation must complete before another one can proceed.

These four types of memory conflicts are discussed in the succeeding paragraphs and examples provided.

Program Wait

Two conditions can prevent the program fetch from beginning:

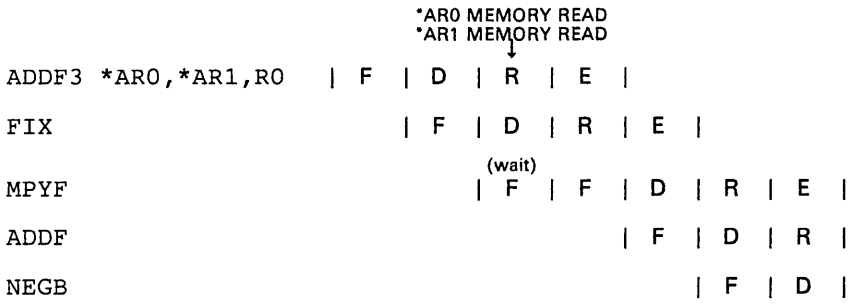
- The start of a CPU-data access when:
  - Two CPU-data accesses are made to an internal RAM or ROM block, and a program fetch from the same block is necessary.
  - One of the external ports is starting a CPU-data access, and a program fetch from the same port is necessary.
- A multicycle CPU-data access or DMA-data access over the external bus is needed.

An example of program wait until a CPU-data access completes is illustrated in Example 10-5. In this case, *AR0 and *AR1 are both pointing to data in RAM block 0, and the MPYF instruction will be fetched from RAM block 0. This results in the conflict shown in Example 10-5. Since no more than two accesses can be made to RAM block 0 in a single cycle, the program fetch cannot begin, and must wait until the CPU-data accesses are complete.

Example 10-5. Program Wait Until CPU-Data Access Completes

```
ADDF3   *AR0,*AR1,R0
FIX
MPYF
ADDF3
NEGB
```

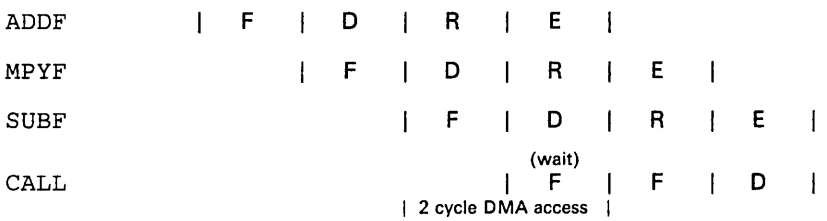
PIPELINE OPERATION



Example 10-6 shows an example of a program wait due to a multicycle data-data access or a multicycle DMA access. The ADDF, MPYF, and SUBF are fetched from some portion in memory other than the external port the DMA requires. The DMA begins a multicycle access. The program fetch corresponding to the CALL is made to the same external port the DMA is using. Even though the DMA has the lowest priority, multi-cycle access cannot be aborted. The program fetch must therefore wait until the DMA access completes.

10

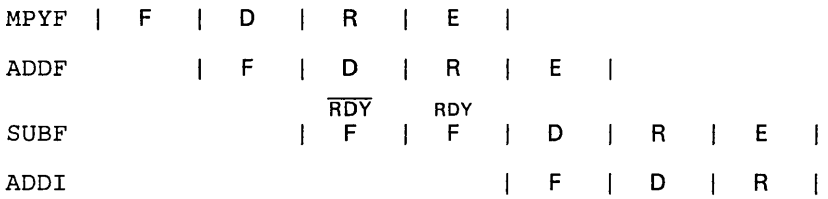
Example 10-6. Program Wait Due to Multicycle Access



Program Fetch Incomplete

A program fetch incomplete occurs when a program fetch takes more than one cycle to complete due to wait states. In Example 10-7, the MPYF and ADDF are fetched from memory that supports single-cycle accesses. The SUBF is fetched from memory requiring one wait state.

Example 10-7. Multicycle Program Memory Fetches



Execute Only

The Execute Only type of memory pipeline conflicts occurs when a sequence of instructions requires three CPU-data accesses in a single cycle or when performing an interlocked load. The three cases where this occurs are:

- An instruction that performs a store, followed by an instruction that does two memory reads.
- An instruction that performs two stores, followed by an instruction that performs at least one memory read.
- An interlocked load (LDII or LDFI) instruction is performed, and XF1 = 1.

The first case is shown in Example 10-8. Since this sequence requires three data memory accesses and only two are available, only the execute phase of the pipeline is allowed to proceed. The dual reads required by the LDF || LDF will be delayed one cycle. Note that a refetch of the next instruction can occur.

Example 10-8. Single Store Followed by Two Reads

```
      STF     R0,*AR1    ; R0 → *AR1
      LDF     *AR2,R1    ; *AR2 → R1 in parallel with
||    LDF     *AR3,R2    ; *AR3 → R2
```

PIPELINE OPERATION

```
                                          R0→*AR1
STF R0,*AR1  |  F  |  D  |  R  |  E  |
                                          *AR2→R1
                                          *AR3→R2
LDF||LDF           |  F  |  D  |  R1 |  R2 |  E  |
                         |  F  |  D1 |  D2 |  R  |...
                               |  F1 |  F2 |  D  |...
```

Example 10-9 shows a parallel store followed by a single load or read. Since the two parallel stores are required, the next CPU-data memory read must wait a cycle before beginning. One program memory refetch may occur.

Example 10-9. Parallel Store Followed by Single Read

```
      STF     R0,*AR0    ; R0 → *AR0 in parallel with
||    STF     R2,*AR1    ; R2 → *AR1
      ADDF    @SUM,R1    ; R1 + @SUM → R1
      IACK
      ASH
```

PIPELINE OPERATION

```
                                          R0→*AR0
                                          R2→*AR1
  STF R0,*AR1
||STF R2,*AR1  |  F  |  D  |  R  |  E  |
                                    (wait)  @SUM
  ADDF @SUM,R1       |  F  |  D  |  R  |  R  |  E  |
  IACK                     |  F  |  D1 |  D2 |  R  |...
  ASH                            |  F1 |  F2 |  D  |...
```

The final case involves an interlocked load (LDII or LDFI) instruction and XF1 = 1. Since the interlocked loads use the XF1 pin as an acknowledge that the read can complete, they may need to extend the read cycle, as shown in Example 10-10. Note that a program refetch may occur.

Example 10-10. Interlocked Load

```
NOT          | F   | D   | R   | E   |
                                  XF1=1   XF1=0
LDII               | F   | D   | R   | R   | E   |
ADDI                     | F   | D1  | D2  | R   |...
CMPI                           | F1  | F2  | D   |...
```

Hold Everything

The three types of Hold Everything memory pipeline conflicts are:

- A CPU-data load or store cannot be performed because an external port is busy.
- An external load that takes more than one cycle.
- Conditional calls and traps.

The first type of Hold Everything conflict occurs when one of the external ports is busy due to an access that has started but is not complete. In Example 10-11, the first store is a two-cycle store. The CPU writes the data to an external port. The port control then takes two cycles to complete the data-data write. The LDF is a read over the same external port. Since the store is not complete, the LDF will continue to be attempted until the port is available. For this case, the first dummy fetch occurs at the same time as D2.

Example 10-11. Busy External Port

```
        STF     R0,@DMA1
        LDF     @DMA2,R0
```

PIPELINE OPERATION

```
                                | 2-cycle DMA access |
STF          | F   | D   | R   | E   |
LDF                | F   | D   | nop | R   | E   |
                         | F   | D1  | D2  | R   |...
                               | F1  | F2  | D   |...
                                           | F   |...
```

The second type of Hold Everything conflict involves multicycle data reads. The read has begun and continues until completed. In Example 10-12, the LDF is performed from an external memory that requires several cycles to complete.

Example 10-12. Multicycle Data Reads

```
                 | F   | D   | R   | E   |
                                   | 2-cycle read |
LDF @DMA,R0          | F   | D   | R   | R   | E   |
                           | F   | D1  | D2  | R   |...
                                   | F   | nop | D   |...
                                         | F1  | F2  |...
```

The final type of Hold Everything conflict deals with conditional calls and traps, which are different from the other branch instructions. Whereas the other branch instructions are conditional loads, the conditional calls and traps are conditional stores, which take one cycle more than a standard branch (see Example 10-13).

Example 10-13. Conditional Calls and Traps

```
                                          (store)
CALLcond | F   | D   | R   | E   | E   |
                 | F1  | (nop) | (nop) | F2  | F3  | D   |
                                                    | F   |
```

10

10.3 Resolving Memory Conflicts

If program fetches and data accesses are performed in such a manner that the resources being used cannot provide the necessary bandwidth, the program fetch is delayed until the data access is complete. Certain configurations of program fetch and data accesses yield conditions under which the TMS320C30 can achieve maximum throughput. Table 10-1 shows how many accesses can be performed from the different memory spaces when it is necessary to do a program fetch and a single data access, and still achieve maximum performance (one cycle). There are four cases that achieve one cycle maximization (see Table 10-1). Table 10-2 shows how many accesses can be performed from the different memory spaces when it is necessary to do a program fetch and two data accesses, still achieving maximum performance (one cycle). There are six cases that achieve this maximization (see Table 10-2).

Table 10-1. One Program Fetch and One-Data Access for Maximum Performance

CASE #	PRIMARY BUS ACCESSES	ACCESSES FROM DUAL-ACCESS INTERNAL MEMORY	EXPANSION BUS OR PERIPHERAL ACCESSES
1	1	1	–
2	1	–	1
3	–	2 from any combination of internal memory	–
4	–	1	1

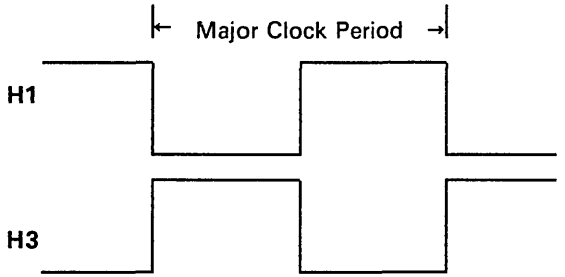
Table 10-2. One Program Fetch and Two Data Accesses for Maximum Performance

CASE #	PRIMARY BUS ACCESSES	ACCESSES FROM DUAL-ACCESS INTERNAL MEMORY	EXPANSION OR PERIPHERAL BUS ACCESSES
1	1	2 from any combination of internal memory	–
2	1 Program	1 Data	1 Data
3	1 Data	1 Data	1 Program
4	–	2 from same internal memory block and 1 from a different internal memory block	–
5	–	3 from different internal memory blocks	–
6	–	2 from any combination of internal memory	1

10.4 Clocking Of Memory Accesses

Internal clock phases (H1 and H3) and their relationship to memory accesses are discussed in this section to show how the TMS320C30 handles multiple memory accesses. Whereas the previous section discussed the interaction between sequences of instructions, this section discusses the flow of data on an individual instruction basis.

Each major clock period of 60 ns is composed of two minor clock periods of 30 ns, labeled as H3 and H1.



The precise operation of memory reads and writes can be defined, based upon these minor clock periods. The types of memory operations which can occur are program fetches, data loads and stores, and DMA accesses.

10.4.1 Program Fetches

Internal program fetches are always performed during H3 unless a single data store must occur at the same time, due to another instruction in the pipeline. In this case, the program fetch occurs during H1 and the data store during H3.

External program fetches always start at the beginning of H3 with the address being presented on the external bus. At the end of H1, they are completed with the latching of the instruction word.

10.4.2 Data Loads and Stores

Four types of instructions perform loads, memory reads, and stores: two-operand instructions, three-operand instructions, multiplier/ALU operation with store instructions, and parallel multiply and add instructions. See Section 6 for detailed information on addressing modes.

Two-Operand Instruction Memory Accesses

Two-operand instructions include all those instructions with bits 31-29 being 000 or 010 (see Figure 10-2). In the case of a data read, bits 15-0 represent the *src* operand. Internal data reads are always performed during H1. External data reads always start at the beginning of H3 with the address being presented on the external bus, and complete with the latching of the instruction word at the end of H1.

In the case of a data store, bits 15-0 represent the *dst* operand. Internal data stores are performed during H3. External data stores always start at the beginning of H3 with the address and data being presented on the external bus.

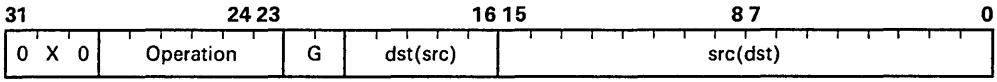


Figure 10-2. Two-Operand Instruction Word

Three-Operand Instruction Memory Reads

Three-operand instructions include all instructions with bits 31-29 being 001 (see Figure 10-3). The source operands, *src1* and *src2*, come from either registers or memory. When one or more of the source operands are from memory, these instructions are always memory reads.

If only one of the source operands is from memory (either *src1* or *src2*) and is located in internal memory, the data is read during H1. If the single memory source operand is in external memory, the read starts at the beginning of H3, with the address being presented on the external bus, and completes with the latching of the data word at the end of H1.

If both source operands are to be fetched from memory, then several cases occur. If both operands are located in internal memory, the *src1* is performed during H3 and *src2* during H1, thus completing two memory reads in a single cycle.

If *src1* is in internal memory and *src2* in external memory, the *src2* access is begun at the start of H3 and latched at the end of H1. At the same time, the *src1* access to internal memory is performed during H3. Again, two memory reads are completed in a single cycle.

If *src1* is in external memory and *src2* in internal memory, two cycles are necessary to complete the two reads. In the first cycle, the internal *src2* access is performed. The *src1* is also performed, but not latched until the next H3.

If *src1* and *src2* are both from external memory, two cycles are required to complete the two reads. In the first cycle, the *src1* access is performed and loaded on the next H3; in the second cycle, the *src2* access is performed and loaded on that cycle's H1.

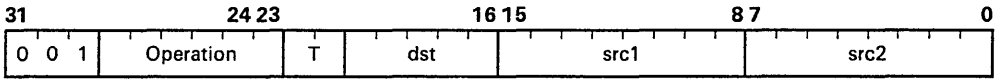


Figure 10-3. Three-Operand Instruction Word

Operations with Parallel Stores

The next class of instructions includes all instructions that have a store, in parallel with another instruction. Bits 31 and 30 for these instruction are equal to 1 1.

For those operations that perform a multiply or ALU operation in parallel with a store, the instruction word format is shown in Figure 10-4. If the store operation to *dst2* is external or internal, it is performed during H3.

If the memory read operation is external, it is started at the beginning of H3 and latched at the end of H1. If the memory read operation is internal, it is performed during H1. Note that memory reads are performed by the CPU during the read (R) phase of the pipeline, and stores during the execute (E) phase.

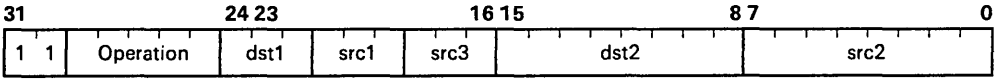


Figure 10-4. A Multiply or CPU Operation with a Parallel Store

The instruction word format for those instructions that have parallel stores to memory is shown in Figure 10-5. If both destination operands, *dst1* and *dst2*, are located in internal memory, *dst1* is stored during H3 and *dst2* during H1, thus completing two memory stores in a single cycle.

If *dst1* is in external memory and *dst2* in internal memory, the *dst1* store is begun at the start of H3. The *dst2* store to internal memory is performed during H1. Again, two memory stores are completed in a single cycle.

If *dst1* is in internal memory and *dst2* in external memory, two cycles are necessary to complete the *dst2* store. In the first cycle, the internal *dst1* store is performed during H3. During the next cycle, the *dst2* store is performed beginning in H3.

If *dst1* and *dst2* are both from external memory, two cycles are necessary to complete the *dst2* store. In the first cycle, the *dst1* access is performed; in the second cycle, the *dst2* access is performed.

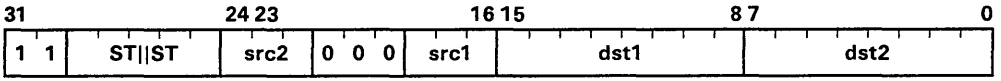


Figure 10-5. Two Parallel Stores

Parallel Multiplies and Adds

The considerations of memory addressing for parallel multiplies and adds is similar to that for three-operand instructions. The parallel multiplies and adds include all instructions with bits 31-30 equal to 10 (see Figure 10-6).

For these operations, *src3* and *src4* are both located in memory. If both operands are located in internal memory, *src3* is performed during H3 and *src4* during H1, thus completing two memory reads in a single cycle.

If *src3* is in internal memory and *src4* in external memory, the *src4* access is begun at the start of H3 and latched at the end of H1. At the same time, the *src3* access to internal memory is performed during H3. Again, two memory reads are completed in a single cycle.

If *src3* is in external memory and *src4* in internal memory, two cycles are necessary to complete the two reads. In the first cycle, the internal *src4* access is performed. During the H3 of the next cycle, the *src3* access is performed.

If *src3* and *src4* are both from external memory, two cycles are necessary to complete the two reads. In the first cycle, the *src3* access is performed; in the second cycle, the *src4* access is performed.

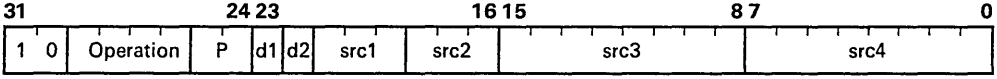


Figure 10-6. Parallel Multiplies and Adds

Introduction 1

Pinout and Signal Descriptions 2

Architectural Overview 3

CPU Registers, Memory, and Cache 4

Data Formats and Floating-Point Operation 5

Addressing 6

Program Flow Control 7

External Bus Operation 8

Peripherals 9

Pipeline Operation 10

Assembly Language Instructions 11

Software Applications 12

Hardware Applications 13

Appendices A-D

Section 11

Assembly Language Instructions

The TMS320C30 assembly language instruction set supports numeric-intensive signal processing and general-purpose applications. The instructions are organized into major groups consisting of load and store, two- or three-operand arithmetic/logical, parallel, program control, and interlocked operations instructions. The addressing modes used with the instructions are described in Section 6.

An additional feature of the TMS320C30 instruction set is the capability of using one of 19 condition codes with any of the 10 conditional instructions, such as LDF*cond*. This section defines the condition codes and flags.

The assembler allows optional syntax forms to simplify the assembly language for special-case instructions. These optional forms are listed and explained.

Each of the individual instructions is described and listed in alphabetical order. An illustration showing an example instruction (see pages 11-15 through 11-17) is provided to show the special format used and explain its content.

Major topics discussed in this section are as follows:

- Instruction Set (Section 11.1 on page 11-2)
  - Load and store instructions
  - Two-operand arithmetic/logical instructions
  - Three-operand arithmetic/logical instructions
  - Program control instructions
  - Interlocked operations instructions
  - Parallel operations instructions
- Condition Codes and Flags (Section 11.2 on page 11-8)
- Individual Instructions (Section 11.3 on page 11-11)
  - Symbols and abbreviations used in instructions
  - Optional assembler syntaxes
  - Individual instruction descriptions alphabetized and including:
    - Syntax
    - Operation
    - Operands
    - Encoding
    - Description
    - Cycles
    - Status bits
    - Mode bit
    - Example(s)

11.1 Instruction Set

The TMS320C30 instruction set is exceptionally well suited to digital signal processing and other numeric-intensive applications. All instructions are a single machine word long, and most instructions take a single cycle to execute. In addition to multiply and accumulate instructions, the TMS320C30 possesses a full complement of general-purpose instructions.

The instruction set contains 114 instructions organized into the following functional groups:

- Load and store
- Two-operand arithmetic/logical
- Three-operand arithmetic/logical
- Program control
- Interlocked operations
- Parallel operations.

Each of these groups is discussed in the succeeding subsections.

11.1.1 Load and Store Instructions

The TMS320C30 supports 12 load and store instructions (see Table 11-1). These instructions can:

- Load a word from memory into a register,
- Store a word from a register into memory, or
- Manipulate data on the system stack.

Two of these instructions can load data conditionally. This is useful for locating the maximum or minimum value in a data set. See Section 11.2 for detailed information on condition codes.

Table 11-1. Load and Store Instructions

INSTRUCTION	DESCRIPTION	INSTRUCTION	DESCRIPTION
LDE	Load floating-point exponent	POP	Pop integer from stack
LDF	Load floating-point value	POPF	Pop floating-point value from stack
LDF*cond*	Load floating-point value conditionally	PUSH	Push integer on stack
LDI	Load integer	PUSHF	Push floating-point value on stack
LDI*cond*	Load integer conditionally	STF	Store floating-point value
LDM	Load floating-point mantissa	STI	Store integer

11.1.2 Two-Operand Instructions

The TMS320C30 supports a complete set of two-operand arithmetic and logical instructions. The two operands are the source and destination. The source operand may be a memory word, a register, or a part of the instruction word. The destination operand is always a register.

These instructions provide integer, floating-point, or logical operations, and multiprecision arithmetic. Table 11-2 lists these instructions.

Table 11-2. Two-Operand Instructions

INSTRUCTION	DESCRIPTION	INSTRUCTION	DESCRIPTION
ABSF	Absolute value of a floating-point number	NORM	Normalize floating-point value
ABSI	Absolute value of an integer	NOT	Bitwise logical-complement
ADDC †	Add integers with carry	OR †	Bitwise logical-OR
ADDF †	Add floating-point values	RND	Round floating-point value
ADDI †	Add integers	ROL	Rotate left
AND †	Bitwise logical-AND	ROLC	Rotate left through carry
ANDN †	Bitwise logical-AND with complement	ROR	Rotate right
ASH †	Arithmetic shift	RORC	Rotate right through carry
CMPF †	Compare floating-point values	SUBB †	Subtract integers with borrow
CMPI †	Compare integers	SUBC	Subtract integers conditionally
FIX	Convert floating-point value to integer	SUBF	Subtract floating-point values
FLOAT	Convert integer to floating-point value	SUBI	Subtract integer
LSH †	Logical shift	SUBRB	Subtract reverse integer with borrow
MPYF †	Multiply floating-point values	SUBRF	Subtract reverse floating-point value
MPYI †	Multiply integers	SUBRI	Subtract reverse integer
NEGB	Negate integer with borrow	TSTB †	Test bit fields
NEGF	Negate floating-point value	XOR †	Bitwise exclusive-OR
NEGI	Negate integer		

† Two- and three-operand versions

11.1.3 Three-Operand Instructions

Most instructions have only two operands; however, some arithmetic and logical instructions have three-operand versions. Three-operand instructions allow the TMS320C30 to read two operands from memory or the CPU register file in a single cycle and store the results in a register. The following differentiates the two- and three-operand instructions:

- Two-operand instructions have a single source operand (or shift count) and a destination operand.
- Three-operand instructions may have two source operands (or one source operand and a count operand) and a destination operand. A source operand may be a memory word or a register. The destination of a three-operand instruction is always a register.

Table 11-3 lists the instructions that have three-operand versions. Note that the '3' in the mnemonic can be omitted from three-operand instructions (see Section 11.3.2).

Table 11-3. Three-Operand Instructions

INSTRUCTION	DESCRIPTION	INSTRUCTION	DESCRIPTION
ADDC3	Add with carry	MPYF3	Multiply floating-point values
ADDF3	Add floating-point values	MPYI3	Multiply integers
ADDI3	Add integers	OR3	Bitwise logical-OR
AND3	Bitwise logical-AND	SUBB3	Subtract integers with borrow
ANDN3	Bitwise logical-AND with complement	SUBF3	Subtract floating-point values
ASH3	Arithmetic shift	SUBI3	Subtract integers
CMPF3	Compare floating-point values	TSTB3	Test bit fields
CMPI3	Compare integers	XOR3	Bitwise exclusive-OR
LSH3	Logical shift		

11

11.1.4 Program Control Instructions

The program-control instruction group consists of all of those instructions which affect program flow. The repeat mode allows repetition of a block of code (RPTB) or of a single line of code (RPTS). Both standard and delayed (single-cycle) branching are supported. Several of the program control instructions are capable of conditional operations (see Section 11.2 for detailed information on condition codes). Table 11-4 lists the program control instructions.

Table 11-4. Program Control Instructions

INSTRUCTION	DESCRIPTION	INSTRUCTION	DESCRIPTION
B*cond*	Branch conditionally (standard)	IDLE	Idle until interrupt
B*cond*D	Branch conditionally (delayed)	NOP	No operation
BR	Branch unconditionally (standard)	RETI *cond*	Return from interrupt conditionally
BRD	Branch unconditionally (delayed)	RETS *cond*	Return from subroutine conditionally
CALL	Call subroutine	RPTB	Repeat block of instructions
CALL*cond*	Call subroutine conditionally	RPTS	Repeat single instruction
DB*cond*	Decrement and branch conditionally (standard)	SWI	Software interrupt
DB*cond*D	Decrement and branch conditionally (delayed)	TRAP *cond*	Trap conditionally

11.1.5 Interlocked Operations Instructions

The interlocked operations instructions support multiprocessor communication. Through the use of external signals, these instructions allow for powerful synchronization mechanisms. They also guarantee the integrity of the communication and result in a high-speed operation. Refer to Section 7 for examples of the use of interlocked instructions. Table 11-5 lists the five interlocked operations instructions.

Table 11-5. Interlocked Operations Instructions

INSTRUCTION	DESCRIPTION	INSTRUCTION	DESCRIPTION
LDFI	Load floating-point value, interlocked	STFI	Store floating-point value, interlocked
LDII	Load integer, interlocked	STII	Store integer, interlocked
SIGI	Signal, interlocked		

11.1.6 Parallel Operations Instructions

The parallel-operations instructions group allows for a high degree of parallelism. Some of the TMS320C30 instructions can occur in pairs that will be executed in parallel. These parallel instructions provide:

- Parallel loading of registers,
- Parallel arithmetic operations, or
- Arithmetic/logical instructions used in parallel with a store instruction.

Each instruction in a pair is entered as a separate source statement. The second instruction in the pair must be preceded by two vertical bars (||). Table 11-6 lists the valid instruction pairs.

Table 11-6. Parallel Instructions

MNEMONIC	DESCRIPTION	OPERATION
PARALLEL ARITHMETIC WITH STORE INSTRUCTIONS		
ABSF || STF	Absolute value of a floating-point	|src2| → dst1 || src3 → dst2
ABSI || STI	Absolute value of an integer	|src2| → dst1 || src3 → dst2
ADDF3 || STF	Add floating-point	src1 + src2 → dst1 || src3 → dst2
ADDI3 || STI	Add integer	src1 + src2 → dst1 || src3 → dst2
AND3 || STI	Bitwise logical-AND	src1 AND src2 → dst1 || src3 → dst2
ASH3 || STI	Arithmetic shift	If count ≥ 0: src2 << count → dst1 || src3 → dst2 Else: src2 >> |count| → dst1 || src3 → dst2
FIX || STI	Convert floating-point to integer	Fix(src2) → dst1 || src3 → dst2
FLOAT || STF	Convert integer to floating-point	Float(src2) → dst1 || src3 → dst2
LDF || STF	Load floating-point	src2 → dst1 || src3 → dst2
LDI || STI	Load integer	src2 → dst1 || src3 → dst2
LSH3 || STI	Logical shift	If count ≥ 0: src2 << count → dst1 || src3 → dst2 Else: src2 >> |count| → dst1 || src3 → dst2
MPYF3 || STF	Multiply floating-point	src1 x src2 → dst1 || src3 → dst2
MPYI3 || STI	Multiply integer	src1 x src2 → dst1 || src3 → dst2
NEGF || STF	Negate floating-point	0- src2 → dst1 || src3 → dst2

LEGEND:

src1 – register addr (R0-R7)
src2 – indirect addr (disp = 0, 1, IR0, IR1)
src3 – register addr (R0-R7)
src4 – indirect addr (disp = 0, 1, IR0, IR1)
dst1 – register addr (R0-R7)
dst2 – indirect addr (disp = 0, 1, IR0, IR1)

Table 11-6. Parallel Instructions (Concluded)

MNEMONIC	DESCRIPTION	OPERATION
PARALLEL ARITHMETIC WITH STORE INSTRUCTIONS (Concluded)		
NEGI || STI	Negate integer	0 - src2 → dst1 || src3 → dst2
NOT3 || STI	Complement	$\overline{\text{src}}$ → dst1 || src3 → dst2
OR3 || STI	Bitwise logical-OR	src1 OR src2 → dst1 || src3 → dst2
STF || STF	Store floating-point	src1 → dst1 || src3 → dst2
STI || STI	Store integer	src1 → dst1 || src3 → dst2
SUBF3 || STF	Subtract floating-point	scr1 - src2 → dst1 || src3 → dst2
SUBI3 || STI	Subtract integer	src1 - src2 → dst1 || src3 → dst2
XOR3 || STI	Bitwise exclusive-OR	src1 XOR src2 → dst1 || src3 → dst2
PARALLEL LOAD INSTRUCTIONS		
LDF || LDF	Load floating-point	src2 → dst1 || src4 → dst2
LDI || LDI	Load integer	src2 → dst1 || src4 → dst2
PARALLEL MULTIPLY AND ADD/SUBTRACT INSTRUCTIONS		
MPYF3 || ADDF3	Multiply and add floating-point	op1 x op2 → op3 || op4 + op5 → op6
MPYF3 || SUBF3	Multiply and subtract floating-point	op1 x op2 → op3 || op4 - op5 → op6
MPYI3 || ADDI3	Multiply and add integer	op1 x op2 → op3 || op4 + op5 → op6
MPYI3 || SUBI3	Multiply and subtract integer	op1 x op2 → op3 || op4 - op5 → op6

LEGEND:

src1 – register addr (R0-R7) **src2** – indirect addr (disp = 0, 1, IR0, IR1)
src3 – register addr (R0-R7) **src4** – indirect addr (disp = 0, 1, IR0, IR1)
dst1 – register addr (R0-R7) **dst2** – indirect addr (disp = 0, 1, IR0, IR1)
op3 – register addr (R0 or R1) **op6** – register addr (R2 or R3)

op1,op2,op4,op5 – Two of these operands must be specified using register addr, and two must be specified using indirect addr.

11.2 Condition Codes and Flags

The TMS320C30 provides 20 condition codes that can be used with any of the conditional instructions, such as RETS*cond* or LDF*cond*. The conditions include signed and unsigned comparisons, comparisons to zero, and comparisons based on the status of individual condition flags. Note that all conditional instructions can accept the suffix 'U' to indicate unconditional operation.

Seven condition flags provide information related to properties of the result of arithmetic and logical instructions. The condition flags are stored in the status register (ST). These flags are modified by the majority of instructions according to whether a result is generated when performing the specified operation to infinite precision or an output is written to the destination register. The formats for output values are shown in Table 11-7.

Table 11-7. Output Value Formats

TYPE OF OPERATION	OUTPUT FORMAT
Floating-point	8-bit exponent, 1 sign bit, 31-bit fraction
Integer	32-bit integer
Logical	32-bit unsigned integer

The condition flags are affected by instructions in only the following cases:

1) The destination register is one of the extended-precision registers (R0 - R7)
2) The instruction is one of the compare instructions (CMPF, CMPF3, CMPI, CMPI3, TSTB, or TSTB3).

Case 1 allows for modification of the registers used for addressing without affecting the condition flags during computation. Case 2 makes it possible to set the condition flags based upon the contents of any of the CPU registers.

The following list defines the condition flags and describes how the flags are set by most instructions. For specific details of the effect of a particular instruction on the condition flags, see the description of that instruction in Section 9.2.

N **Negative Condition Flag.** Logical operations assign N the state of the MSB of the output value. For integer and floating-point operations, N is set if the result is negative, and cleared otherwise. Zero is considered to be positive.

Z **Zero Condition Flag.** For logical, integer, and floating-point operations, Z is set if the output is 0, and cleared otherwise.

V **Overflow Condition Flag.** For integer operations, V is set if the result does not fit into the format specified for the destination (i.e., $-2^{32} \leq$ result $\leq 2^{32} - 1$). Otherwise, V is cleared. For floating-point operations, V is set if the exponent of the result is greater than 127, otherwise,V is cleared. Logical operations always clear V.

C **Carry Flag.** When an integer addition is performed, C is set if a carry occurs out of the bit corresponding to the MSB of the output. When

an integer subtraction is performed, C is set if a borrow occurs into the bit corresponding to the MSB of the output. Otherwise, for integer operations, C is cleared. The carry flag is unaffected by floating-point and logical operations.

UF **Floating-Point Underflow Condition Flag.** A floating-point underflow occurs whenever the exponent of the result is less than or equal to -128. If a floating-point underflow occurs, UF is set, and the output value is set to 0. UF is cleared if a floating-point underflow does not occur.

LV **Latched Overflow Condition Flag.** LV is set whenever V (overflow condition flag) is set. Otherwise, it is unchanged. LV may only be cleared by a processor reset or by modifying it in the status register (ST).

LUF **Latched Underflow Condition Flag.** LUF is set whenever UF (floating-point underflow flag) is set. LUF may only be cleared by a processor reset or by modifying it in the status register (ST).

Table 11-8 lists the condition mnemonic, code, description, and flag for each of the 19 conditions.

Table 11-8. Condition Codes and Flags

CONDITION	CODE	DESCRIPTION	FLAG
		UNCONDITIONAL COMPARES	
U	00000	Unconditional	Don't care
		UNSIGNED COMPARES	
LO	00001	Lower than	C
LS	00010	Lower or same	C OR Z
HI	00011	Higher than	~C AND ~Z
HS	00100	Higher or same	~C
EQ	00101	Equal	Z
NE	00110	Not Equal	~Z
		SIGNED COMPARES	
LT	00111	Less than	N
LE	01000	Less than or equal	N OR Z
GT	01001	Greater than	~N AND ~Z
GE	01010	Greater than or equal	~N
EQ	00101	Equal	Z
NE	00110	Not equal	~Z
		COMPARE TO ZERO	
Z	00101	Zero	Z
NZ	00110	Not zero	~Z
P	01001	Positive	~N AND ~Z
N	00111	Negative	N
NN	01010	Nonnegative	~N
		COMPARE TO CONDITION FLAGS	
NN	01010	Nonnegative	~N
N	00111	Negative	N
NZ	00110	Nonzero	~Z
Z	00101	Zero	Z
NV	01100	No overflow	~V
V	01101	Overflow	V
NUF	01110	No underflow	~UF
UF	01111	Underflow	UF
NC	00100	No carry	~C
C	00001	Carry	C
NLV	10000	No latched overflow	~LV
LV	10001	Latched overflow	LV
NLUF	10010	No latched floating-point underflow	~LUF
LUF	10011	Latched floating-point underflow	LUF
ZUF	10100	Zero or floating-point underflow	Z OR UF

~ Logical complement

11.3 Individual Instructions

This section contains the individual assembly language instructions for the TMS320C30. The instructions are listed in alphabetical order. Information, such as assembler syntax, operation, operands, encoding, description, cycles, status bits, mode bit, and examples, is provided for each instruction. An example instruction precedes the individual instruction listings to show the special format used and explain its content.

Preceding the individual instruction descriptions, the symbols and abbreviations used in the individual instructions are defined. In addition, some optional syntax forms allowed by the assembler are described.

A functional grouping of the instructions is provided in Section 1.6. A complete instruction set summary can be found in Section 1.6.8. Appendix B lists the opcodes for all the instructions. Refer to Section 6 for information on memory addressing. Code examples using many of the instructions are given in Section 12, Software Applications.

11.3.1 Symbols and Abbreviations

Table 11-9 lists the symbols and abbreviations used in the individual instruction descriptions.

Table 11-9. Instruction Symbols

SYMBOL	MEANING
src	Source operand
src1	Source operand 1
src2	Source operand 2
src3	Source operand 3
src4	Source operand 4
dst	Destination operand
dst1	Destination operand 1
dst2	Destination operand 2
disp	Displacement
cond	Condition
count	Shift count
G	General addressing modes
T	Three-operand addressing modes
P	Parallel addressing modes
B	Conditional-branch addressing modes
ARn	Auxiliary register n
IRn	Index register n
Rn	Register address n
RC	Repeat count register
RE	Repeat end address register
RS	Repeat start address register
ST	Status register
C	Carry bit
GIE	Global interrupt enable bit
N	Trap vector
PC	Program counter
RM	Repeat mode flag
SP	System stack pointer
|x|	Absolute value of x
x → y	Assign the value of x to destination y
x(*man*)	Mantissa field (sign + fraction) of x
x(*exp*)	Exponent field of x
op1 || op2	Operation 1 performed in parallel with operation 2
x AND y	Bitwise logical-AND of x and y
x OR y	Bitwise logical-OR of x and y
x XOR y	Bitwise logical-XOR of x and y
~x	Bitwise logical-complement of x
x << y	Shift x to the left y bits
x >> y	Shift x to the right y bits
*++SP	Increment SP and use incremented SP as address
*SP--	Use SP as address and decrement SP

11.3.2 Optional Assembler Syntaxes

The assembler allows a relaxed syntax form for some of the instructions. These optional forms simplify the assembly language so that special-case syntax can be ignored for some of the instructions. The following is a list of these optional syntax forms.

- The destination register can be omitted on unary arithmetic and logical operations when the same register is used as a source. For example,

  `ABSI R0,R0` can be written as `ABSI R0`

  Instructions affected: ABSI, ABSF, FIX, FLOAT, NEGB, NEGF, NEGI, NORM, NOT, RND.

- All 3-operand instructions can be written without the '3'. For example,

  `ADDI3 R0,R1,R2` can be written as `ADDI R0,R1,R2`

  Instructions affected: ADDC3, ADDF3, ADDI3, AND3, ANDN3, ASH3, LSH3, MPYF3, MPYI3, OR3, SUBB3, SUBF3, SUBI3, XOR3.

  This also applies to all the pertinent parallel instructions.

- All 3-operand comparison instructions can be written without the '3'. For example,

  `CMPI3 R0,*AR0` can be written as `CMPI R0,*AR0`

  Instructions affected: CMPI3, CMPF3, TSTB3.

- Indirect operands with an explicit 0 displacement are allowed. In 3-operand or parallel instructions, operands with 0 displacement are automatically converted to "no-displacement" mode. For example:

  ```
  LDI  *+AR0(0),R1
  ```

  is legal

  Also

  `ADDI3 *+AR0(0),R1,R2 is equivalent to ADDI3 *AR0,R1,R2`

- Indirect operands can be written with no displacement, in which case a displacement of one is assumed. For example,

  `LDI *AR0++(1),R0 can be written LDI *AR0++,R0`

- All conditional instructions accept the suffix 'U' to indicate unconditional operation. Also, the U can be omitted from unconditional short branch instructions. For example:

  `BU label` can be written `B label`

- Labels can be written with or without a trailing colon. For example:

  ```
  label0:  NOP
  label1   NOP
  label2:
           NOP
  ```

- Empty expressions are not allowed for the displacement in indirect mode:

```
LDI  *+AR0(),R0          is not legal
```

- Long immediate mode operands (destination of BR and CALL) can be written with an at-sign:

```
BR label          can be written          BR  @label
```

- The LDP pseudo-op can be used to load a register (usually DP) with the 8 MSBs of a relocatable address. The instruction is written:

```
LDP  addr,REG              or              LDP  @addr,REG
```

  The at-sign is optional.

  If the destination REG is the DP, it can be omitted. LDP generates a LDI instruction with an immediate operand, and a special relocation type.

- Parallel instructions can be written in either order. For example:

```
   ADDI              can be written as              STI
|| STI                                          || ADDI
```

- The parallel bars indicating part 2 of a parallel instruction can be written anywhere on the line, from column 0 to the mnemonic. For example:

```
   ADDI              can be written as              ADDI
|| STI                                          || STI
```

- If the second operand of a parallel instruction is the same as the third (destination register) operand, the third operand can be omitted. This allows the writing of 3-operand parallel instructions that 'look like' normal 2-operand instruction. For example,

```
   ADDI *AR0,R2,R2   can be written as      ADDI *AR0,R2
|| MPYI *AR1,R0,R0                       || MPYI *AR1,R0
```

  Instructions (applies to all parallel instructions that have a register second operand) affected: ADDI, ADDF, AND, MPYI, MPYF, OR, SUBI, SUBF, XOR.

- All commutive operations in parallel instructions can be written in either order. For example, the ADDI part of a parallel instruction can be written in either of two ways:

```
ADDI  *AR0,R1,R2           or           ADDI  R1,*AR0,R2
```

  The instructions affected are parallel instructions containing any of the following: ADDI, ADDF, MPYI, MPYF, AND, OR, XOR.

11.3.3 Individual Instruction Descriptions

Each assembly language instruction for the TMS320C30 is described in this section. The instructions are listed in alphabetical order. An example instruction precedes the individual instructions to show the special format used and explain its content. This example instruction describes the assembler syntax, operation, operands, encoding, description, cycles, status bits, mode bit, and examples.

Syntax

INST <*src*>,<*dst*>

or

INST1 <*src2*>,<*dst1*>
|| INST2 <*src3*>,<*dst2*>

Each instruction begins with an assembler syntax expression. Labels may be placed either preceding the command (instruction mnemonic) on the same line or on the preceding line in the first column. The optional comment field that concludes the syntax is not included in the syntax expression. Space(s) are required between each field (label, command, operand, and comment fields).

The syntax examples illustrate the common one-line syntax and the two-line syntax used in parallel addressing. Note that the two vertical bars || that indicate a parallel addressing pair can be placed anywhere before the mnemonic on the second line. The first instruction in the pair can have a label, but the second instruction cannot have a label.

Operation

|*src*| → *dst*

or

|*src2*| → *dst1*
|| *src3* → *dst2*

The instruction operation sequence describes the processing that takes place when the instruction is executed. For parallel instructions, the operation sequence is performed in parallel. Conditional effects of status register specified modes will be listed for conditional instructions such as B*cond*.

Operands

src general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

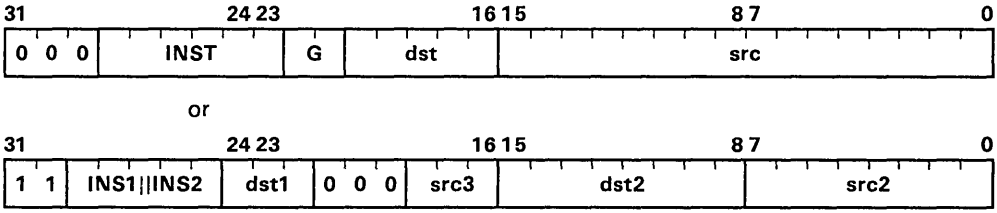
dst register (Rn, 0 ≤ n ≤ 27)

or

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src3 register (Rn2, 0 ≤ n2 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Operands are defined according to the addressing mode and/or the type of addressing used. Note that indirect addressing uses displacements and the index registers. Refer to Section 6 for detailed information on addressing.

Encoding



Encoding examples are shown using general addressing and parallel addressing. The instruction pair for the parallel addressing example consists of INS1 and INS2.

Description Instruction execution and its effect on the rest of the processor or memory contents are described. Any constraints on the operands imposed by the processor or the assembler are discussed. The description parallels and supplements the information given by the operation block.

Cycles 1

The digit specifies the number of cycles required to execute the instruction.

Status Bits

N **Negative Condition Flag.** 1 if a negative result is generated, 0 otherwise. In some instructions, this flag is the MSB of the output. For other instructions, this flag is unaffected.

Z **Zero Condition Flag.** 1 if a zero result is generated, 0 otherwise. For logical and shift instructions, 1 if a zero output is generated, 0 otherwise. This flag may be unaffected.

V **Overflow Condition Flag.** 1 if an integer or floating-point overflow occurs, 0 otherwise. This flag may be unaffected.

C **Carry Flag.** 1 if a carry or borrow occurs, 0 otherwise. For shift instructions, this flag is set to the value of the last bit shifted out; 0 for a shift count of 0. This flag may be unaffected.

UF **Floating-Point Underflow Condition Flag.** If a floating-point underflow occurs, 0 otherwise. This flag may be unaffected.

LV **Latched Overflow Condition Flag.** 1 if an integer or floating-point overflow occurs, unchanged otherwise. This flag may be unaffected.

LUF **Latched Floating-Point Underflow Condition Flag.** 1 if a floating-point underflow occurs, unchanged otherwise. This flag may be unaffected.

The seven condition flags, stored in the status register (ST), are modified by the majority of instructions. They provide information as to the properties of the result or output of arithmetic or logical operations.

Mode Bit **OVM Overflow Mode Flag.** In general, integer operations are affected by the OVM flag.

Example `INST @98AEh,R5`

Before Instruction:

DP = 80h
R5 = 0766900000h = 2.30562500e+02
Memory at 8098AEh = 5CDFh = 1.00001107e+00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R5 = 0066900000h = 1.80126953e+00
Memory at 8098AEh = 5CDFh = 1.00001107e+00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

The sample code presented in the above format shows the effect of the code on system pointers (e.g., DP or SP), registers (e.g., R1 or R5), memory at specific locations, and the seven status bits. The values given for the registers include the leading zeros to show the exponent in floating-point operations. Decimal conversions are provided for all register and memory locations. The seven status bits are listed in the order in which they appear in the assembler and simulator (see Table 11-9 and Section 11.2 for further information on these seven status bits).

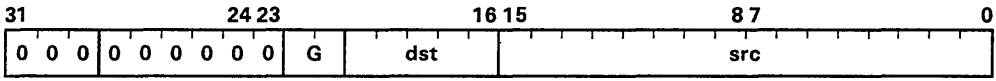
Syntax ABSF *<src>,<dst>*

Operation |*src*| → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The absolute value of the *src* operand is loaded into the *dst* register. The *src* and *dst* operands are assumed to be floating-point numbers.

An overflow occurs if *src* (man) = 80000000h and *src* (exp) = 7Fh. The result is *dst* (man) = 7FFFFFFFh and *dst* (exp) = 7Fh.

Cycles 1

Status Bits
N 0
Z 1 if a zero result is generated, 0 otherwise.
V 1 if a floating-point overflow occurs, 0 otherwise.
C Unaffected.
UF 0
LV 1 if a floating-point overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
ABSF  R4,R7
```

Before Instruction:

```
R4 = 05C8000F971h = -9.90337307e+27
R7 = 07D251100AEh =  5.48527255e+37
LUF LV UF N Z V C = 0 0 0 0 0 0 0
```

After Instruction:

```
R4 = 05C8000F971h = -9.90337307e+27
R7 = 05C7FFF068Fh =  9.90337307e+27
LUF LV UF N Z V C = 0 0 0 0 0 0 0
```

Syntax

ABSF <*src2*>,<*dst1*>
|| STF <*src3*>,<*dst2*>

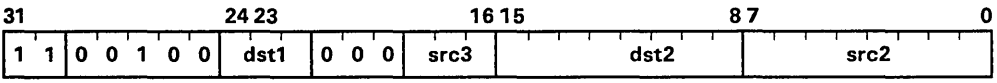
Operation

|*src2*| → *dst1*
|| *src3* → *dst2*

Operands

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src3 register (Rn2, 0 ≤ n2 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

A floating-point absolute value and a floating-point store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STF) reads from a register and the operation being performed in parallel (ABSF) writes to the same register, then STF accepts as input the contents of the register before it is modified by the ABSF.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

An overflow occurs if *src* (man) = 80000000h and *src* (exp) = 7Fh. The result is *dst* (man) = 7FFFFFFFh and *dst* (exp) = 7Fh.

Cycles

1

Status Bits

N 0
Z 1 if a zero result is generated, 0 otherwise.
V 1 if a floating-point overflow occurs, 0 otherwise.
C Unaffected.
UF 0
LV 1 if a floating-point overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit

OVM Operation not affected by OVM.

Example

```
   ABSF  *++AR3(IR1),R4
|| STF   R4,*-AR7(1)
```

Before Instruction:

AR3 = 809800h
IR1 = 0AFh
R4 = 733C00000h = 1.79750e+02
AR7 = 8098C5h
Data at 8098AFh = 58B4000h = -6.118750e+01
Data at 8098C4h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR3 = 8098AFh
IR1 = 0AFh
R4 = 574C00000h = 6.118750e+01
AR7 = 8098C5h
Data at 8098AFh = 58B4000h = -6.118750e+01
Data at 8098C4h = 733C000h = 1.79750e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

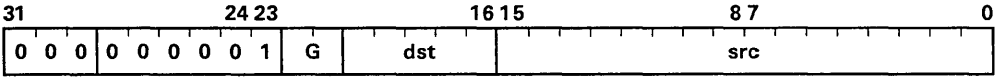
Syntax ABSI <*src*>,<*dst*>

Operation |*src*| → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The absolute value of the *src* operand is loaded into the *dst* register. The *src* and *dst* operands are assumed to be signed integers.

An overflow occurs if *src* = 80000000h. If ST(OVM) = 1, the result is *dst* = 7FFFFFFFh. If ST(OVM) = 0, the result is *dst* = 00000000h.

Cycles 1

Status Bits
N 0
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C Unaffected.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit **OVM** Operation affected by OVM.

Example

```
ABSI  R0,R0
or ABSI  R0
```

Before Instruction:

R0 = 0FFFFFFCBh = -53

After Instruction:

R0 = 035h = 53

Example

```
ABSI  *AR1,R3
```

Before Instruction:

AR1 = 20h
R3 = 0h
Data at 20h = 0FFFFFFCBh = -53

After Instruction:

AR1 = 20h
R3 = 35h = 53
Data at 20h = 0FFFFFFCBh = -53

Syntax

ABSI <*src2*>,<*dst1*>
|| STI <*src3*>,<*dst2*>

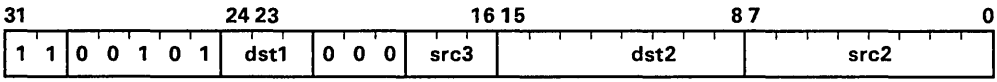
Operation

|*src2*| → *dst1*
|| *src3* → *dst2*

Operands

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src3 register (Rn2, 0 ≤ n2 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

An integer absolute value and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (ABSI) writes to the same register, then STI accepts as input the contents of the register before it is modified by the ABSI.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

An overflow occurs if *src* = 80000000h. If ST(OVM) = 1, the result is *dst* = 7FFFFFFFh. If ST(OVM) = 0, the result is *dst* = 00000000h.

Cycles

1

Status Bits

N 0
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C Unaffected.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit

OVM Operation affected by OVM.

Example

```
   ABSI *-AR5(1),R5
|| STI  R1,*AR2--(IR1)
```

Before Instruction:

AR5 = 8099E2h
R5 = 0h
R1 = 42h = 66
AR2 = 8098FFh
IR1 = 0Fh
Data at 8099E1h = 0FFFFFFCBh = -53
Data at 8098FFh = 2h = 2
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR5 = 8099E2h
R5 = 35h = 53
R1 = 42h = 66
AR2 = 8098F0h
IR1 = 0Fh
Data at 8099E1h = 0FFFFFFCBh = -53
Data at 8098FFh = 42h = 66
LUF LV UF N Z V C = 0 0 0 0 0 0 0

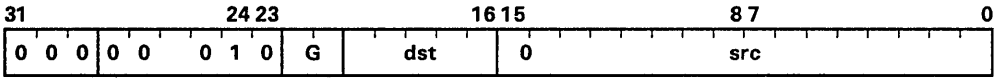
Syntax ADDC <*src*>,<*dst*>

Operation *dst* + *src* + C → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The sum of the *dst* and *src* operands and the C (carry) flag is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C 1 if a carry occurs, 0 otherwise.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit **OVM** Operation affected by OVM.

Example

```
ADDC  R1,R5
```

Before Instruction:

R1 = 00FFFF5C25h = -41,947
R5 = 00FFFF019Eh = -65,122
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

R1 = 00FFFF5C25h = -41,947
R5 = 00FFFE5DC4h = -107,068
LUF LV UF N Z V C = 0 0 0 1 0 0 1

Syntax ADDC3 <*src2*>,<*src1*>,<*dst*>

Operation *src1* + *src2* + C → *dst*

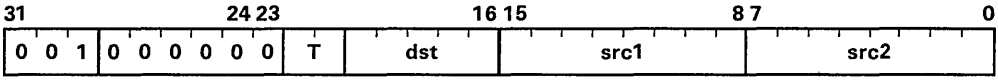
Operands

src1 three-operand addressing modes (T):
- 0 0 register (Rn1, 0 ≤ n1 ≤ 27)
- 0 1 indirect (disp = 0, 1, IR0, IR1)
- 1 0 register (Rn1, 0 ≤ n1 ≤ 27)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):
- 0 0 register (Rn2, 0 ≤ n2 ≤ 27)
- 0 1 register (Rn2, 0 ≤ n2 ≤ 27)
- 1 0 indirect (disp = 0, 1, IR0, IR1)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The sum of the *src1* and *src2* operands and the C (carry) flag is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be signed integers.

Cycles 1

Status Bits
- **N** 1 if a negative result is generated, 0 otherwise.
- **Z** 1 if a zero result is generated, 0 otherwise.
- **V** 1 if an integer overflow occurs, 0 otherwise.
- **C** 1 if a carry occurs, 0 otherwise.
- **UF** 0
- **LV** 1 if an integer overflow occurs, unchanged otherwise.
- **LUF** Unaffected.

Mode Bit **OVM** Operation affected by OVM.

Example

```
ADDC3   *AR5++(IR0),R5,R2
or
ADDC3   R5,*AR5++(IR0),R2
```

Before Instruction:

AR5 = 809908h
IR0 = 10h
R5 = 066h = 102
R2 = 0h
Data at 809908h = 0FFFFFFCBh = -53
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

AR5 = 809918h
IR0 = 10h
R5 = 066h = 102
R2 = 032h = 50
Data at 809908h = 0FFFFFFCBh = -53
LUF LV UF N Z V C = 0 0 0 0 0 0 1

Example

```
ADDC3   R2, R7, R0
```

Before Instruction:

R2 = 02BCh = 700
R7 = 0F82h = 3970
R0 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

R2 = 02BCh = 700
R7 = 0F82h = 3970
R0 = 0123Fh = 4671
LUF LV UF N Z V C = 0 0 0 0 0 0 0

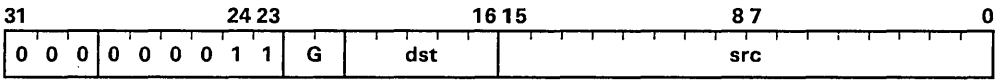
Syntax ADDF *<src>,<dst>*

Operation *dst* + *src* → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The sum of the *dst* and *src* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if a floating-point overflow occurs, 0 otherwise.
C Unaffected.
UF 1 if a floating-point underflow occurs, 0 otherwise.
LV 1 if a floating-point overflow occurs, unchanged otherwise.
LUF 1 if a floating-point underflow occurs, unchanged otherwise.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
ADDF   *AR4++(IR1),R5
```

Before Instruction:

AR4 = 809800h
IR1 = 12Bh
R5 = 0579800000h = 6.23750e+01
Data at 80992Bh = 86B2800h = 4.7031250e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR4 = 80992Bh
IR1 = 12Bh
R5 = 09052C0000h = 5.3268750e+02
Data at 80992Bh = 86B2800h = 4.7031250e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax ADDF3 *<src2>,<src1>,<dst>*

Operation *src1* + *src2* → *dst*

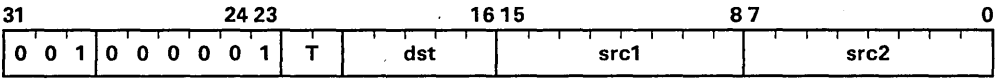
Operands

src1 three-operand addressing modes (T):
- 0 0 register (Rn1, 0 ≤ n1 ≤ 7)
- 0 1 indirect (disp = 0, 1, IR0, IR1)
- 1 0 register (Rn1, 0 ≤ n1 ≤ 7)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):
- 0 0 register (Rn2, 0 ≤ n2 ≤ 7)
- 0 1 register (Rn2, 0 ≤ n2 ≤ 7)
- 1 0 indirect (disp = 0, 1, IR0, IR1)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The sum of the *src1* and *src2* operands is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits
- **N** 1 if a negative result is generated, 0 otherwise.
- **Z** 1 if a zero result is generated, 0 otherwise.
- **V** 1 if a floating-point overflow occurs, 0 otherwise.
- **C** Unaffected.
- **UF** 1 if a floating-point underflow occurs, 0 otherwise.
- **LV** 1 if a floating-point overflow occurs, unchanged otherwise.
- **LUF** 1 if a floating-point underflow occurs, unchanged otherwise.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
ADDF3  R6,R5,R1
or
ADDF3  R5,R6,R1
```

Before Instruction:

R6 = 086B280000h = 4.7031250e+02
R5 = 0579800000h = 6.23750e+01
R1 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R6 = 086B280000h = 4.7031250e+02
R5 = 0579800000h = 6.23750e+01
R1 = 09052C0000h = 5.3268750e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example

```
ADDF3  *+AR1(1),*AR7++(IR0),R4
```

Before Instruction:

AR1 = 809820h
AR7 = 8099F0h
IR0 = 8h
R4 = 0h
Data at 809821h = 700F000h = 1.28940e+02
Data at 8099F0h = 34C2000h = 1.27590e+01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 809820h
AR7 = 8099F8h
IR0 = 8h
R4 = 070DB20000h = 1.41695313e+02
Data at 809821h = 700F000h = 1.28940e+02
Data at 8099F0h = 34C2000h = 1.27590e+01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

ADDF3 *<src2>,<src1>,<dst1>*
|| STF *<src3>,<dst2>*

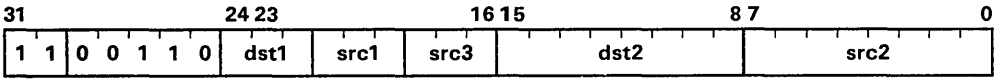
Operation

src1 + *src2* → *dst1*
|| *src3* → *dst2*

Operands

src1 register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn2, 0 ≤ n2 ≤ 7)
src3 register (Rn3, 0 ≤ n3 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

A floating-point addition and a floating-point store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STF) reads from a register and the operation being performed in parallel (ADDF3) writes to the same register, then STF accepts as input the contents of the register before it is modified by the ADDF3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Cycles

1

Status Bits

N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if a floating-point overflow occurs, 0 otherwise.
C Unaffected.
UF 1 if a floating-point underflow occurs, 0 otherwise.
LV 1 if a floating-point overflow occurs, unchanged otherwise.
LUF 1 if a floating-point underflow occurs, unchanged otherwise.

Mode Bit

OVM Operation not affected by OVM.

Example

```
   ADDF3  *+AR3(IR1),R2,R5
|| STF   R4,*AR2
```

Before Instruction:

AR3 = 809800h
IR1 = 0A5h
R2 = 070C800000h = 1.4050e+02
R5 = 0h
R4 = 057B400000h = 6.281250e+01
AR2 = 8098F3h
Data at 8098A5h = 733C000h = 1.79750e+02
Data at 8098F3h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR3 = 809800h
IR1 = 0A5h
R2 = 070C800000h = 1.4050e+02
R5 = 0820200000h = 3.20250e+02
R4 = 057B400000h = 6.281250e+01
AR2 = 8098F3h
Data at 8098A5h = 733C000h = 1.79750e+02
Data at 8098F3h = 57B4000h = 6.28125e+01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

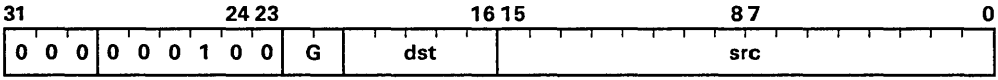
Syntax ADDI *<src>,<dst>*

Operation *dst* + *src* → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The sum of the *dst* and *src* operands is loaded into the the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

Cycles 1

Status Bits

N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C 1 if a carry occurs, 0 otherwise.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit **OVM** Operation affected by OVM.

Example

```
ADDI  R3,R7
```

Before Instruction:

R3 = 0FFFFFFCBh = -53
R7 = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R3 = 0FFFFFFCBh = -53
R7 = 0h
LUF LV UF N Z V C = 0 0 0 0 1 0 0

11

Syntax ADDI3 *<src2>,<src1>,<dst>*

Operation *src1* + *src2* → *dst*

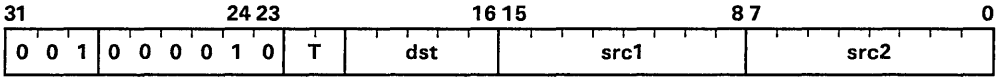
Operands

src1 three-operand addressing modes (T):
- 0 0 register (Rn1, 0 ≤ n1 ≤ 27)
- 0 1 indirect (disp = 0, 1, IR0, IR1)
- 1 0 register (Rn1, 0 ≤ n1 ≤ 27)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):
- 0 0 register (Rn2, 0 ≤ n2 ≤ 27)
- 0 1 register (Rn2, 0 ≤ n2 ≤ 27)
- 1 0 indirect (disp = 0, 1, IR0, IR1)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The sum of the *src1* and *src2* operands is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be signed integers.

Cycles 1

Status Bits
- **N** 1 if a negative result is generated, 0 otherwise.
- **Z** 1 if a zero result is generated, 0 otherwise.
- **V** 1 if an integer overflow occurs, 0 otherwise.
- **C** 1 if a carry occurs, 0 otherwise.
- **UF** 0
- **LV** 1 if an integer overflow occurs, unchanged otherwise.
- **LUF** Unaffected.

Mode Bit **OVM** Operation affected by OVM.

Example `ADDI3 R4,R7,R5`

Before Instruction:

R4 = 0DCh = 220
R7 = 0A0h = 160
R5 = 10h = 16
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R4 = 0DCh = 220
R7 = 0A0h = 160
R5 = 017Ch = 380
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example `ADDI3 *-AR3+(1),*AR6--(IR0),R2`

Before Instruction:

AR3 = 809802h
AR6 = 809930h
IR0 = 18h
R2 = 10h = 16
Data at 809801h = 2AF8h = 11,000
Data at 809930h = 3A98h = 15,000
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR3 = 809852h
AR6 = 809918h
IR0 = 18h
R2 = 06598h = 26,000
Data at 809801h = 2AF8h = 11,000
Data at 809930h = 3A98h = 15,000
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

ADDI3 <*src2*>,<*src1*>,<*dst1*>
|| STI <*src3*>,<*dst2*>

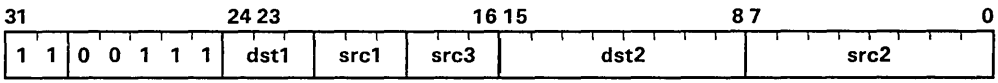
Operation

src1 + *src2* → *dst1*
|| *src3* → *dst2*

Operands

src1 register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn2, 0 ≤ n2 ≤ 7)
src3 register (Rn3, 0 ≤ n3 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

An integer addition and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (ADDI3) writes to the same register, then STI accepts as input the contents of the register before it is modified by the ADDI3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Cycles

1

Status Bits

N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C 1 if a carry occurs, 0 otherwise.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit

OVM Operation affected by OVM.

Example

```
   ADDI3  *AR0--(IR0),R5,R0
|| STI   R3,*AR7
```

Before Instruction:

AR0 = 80992Ch
IR0 = 0Ch
R5 = 0DCh = 220
R0 = 0h
R3 = 35h = 53
AR7 = 80983Bh
Data at 80992Ch = 12Ch = 300
Data at 80983Bh = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR0 = 809920h
IR0 = 0Ch
R5 = 0DCh = 220
R0 = 208h = 520
R3 = 35h = 53
AR7 = 80983Bh
Data at 80992Ch = 12Ch = 300
Data at 80983Bh = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0

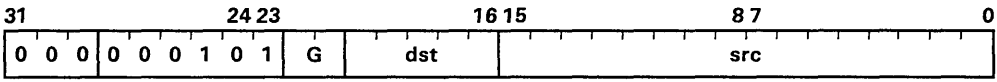
Syntax AND *<src>,<dst>*

Operation *dst* AND *src* → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate (not sign-extended)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The bitwise logical-AND between the *dst* and *src* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be unsigned integers.

Cycles 1

Status Bits
N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
AND  R1,R2
```

Before Instruction:

R1 = 80h
R2 = 0AFFh
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

R1 = 80h
R2 = 80h
LUF LV UF N Z V C = 0 0 0 0 0 0 1

Syntax AND <*src2*>,<*src1*>,<*dst*>

Operation *src1* AND *src2* → *dst*

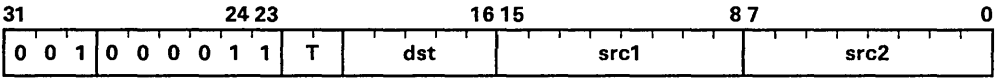
Operands

src1 three-operand addressing modes (T):
- 0 0 register (Rn1, 0 ≤ n1 ≤ 27)
- 0 1 indirect (disp = 0, 1, IR0, IR1)
- 1 0 register (Rn1, 0 ≤ n1 ≤ 27)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):
- 0 0 register (Rn2, 0 ≤ n2 ≤ 27)
- 0 1 register (Rn2, 0 ≤ n2 ≤ 27)
- 1 0 indirect (disp = 0, 1, IR0, IR1)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The bitwise logical-AND between the *src1* and *src2* operands is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be unsigned integers.

Cycles 1

Status Bits
- **N** MSB of the output.
- **Z** 1 if a zero output is generated, 0 otherwise.
- **V** 0
- **C** Unaffected.
- **UF** 0
- **LV** Unaffected.
- **LUF** Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example `AND3 *AR0--(IR0),*+AR1,R4`

Before Instruction:

AR0 = 8098F4h
IR0 = 50h
AR1 = 809951h
R4 = 0h
Data at 8098A4h = 30h
Data at 809952h = 123h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR0 = 8098A4h
IR0 = 50h
AR1 = 809951h
R4 = 020h
Data at 8098A4h = 30h
Data at 809952h = 123h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example `AND3 *-AR5,R7,R4`

Before Instruction:

AR5 = 80985Ch
R7 = 2h
R4 = 0h
Data at 80985Bh = 0AFFh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR5 = 80985Ch
R7 = 2h
R4 = 2h
Data at 80985Bh = 0AFFh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

AND <*src2*>,<*src1*>,<*dst1*>
|| STI <*src3*>,<*dst2*>

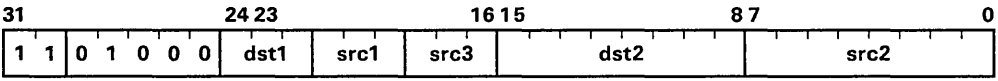
Operation

src1 AND *src2* → *dst1*
|| *src3* → *dst2*

Operands

src1 register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn2, 0 ≤ n2 ≤ 7)
src3 register (Rn3, 0 ≤ n3 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

A bitwise logical-AND and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (AND3) writes to the same register, then STI accepts as input the contents of the register before it is modified by the AND3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Cycles

1

Status Bits

N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit

OVM Operation not affected by OVM.

Example

```
    AND3  *+AR1(IR0),R4,R7
||  STI   R3,*AR2
```

Before Instruction:

AR1 = 8099F1h
IR0 = 8h
R4 = 0A323h
R7 = 0h
R3 = 35h = 53
AR2 = 80983Fh
Data at 8099F9h = 5C53h
Data at 80983Fh = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 8099F1h
IR0 = 8h
R4 = 0A323h
R7 = 03h
R3 = 35h = 53
AR2 = 80983Fh
Data at 8099F9h = 5C53h
Data at 80983Fh = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0

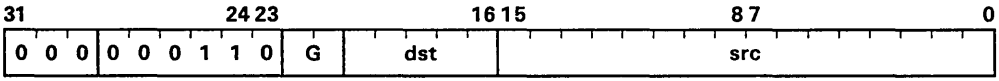
Syntax ANDN *<src>,<dst>*

Operation *dst* AND ~*src* → *dst*

Operands *src* general addressing modes (G):
- 0 0 register (Rn, 0 ≤ n ≤ 27)
- 0 1 direct
- 1 0 indirect
- 1 1 immediate (not sign-extended)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The bitwise logical-AND between the *dst* operand and the bitwise logical complement (~) of the *src* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be unsigned integers.

Cycles 1

Status Bits
- **N** MSB of the output.
- **Z** 1 if a zero output is generated, 0 otherwise.
- **V** 0
- **C** Unaffected.
- **UF** 0
- **LV** Unaffected.
- **LUF** Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
ANDN  @980Ch,R2
```

Before Instruction:

DP = 80h
R2 = 0C2Fh
Data at 80980Ch = 0A02h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R2 = 042Dh
Data at 80980Ch = 0A02h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax ANDN3 <*src2*>,<*src1*>,<*dst*>

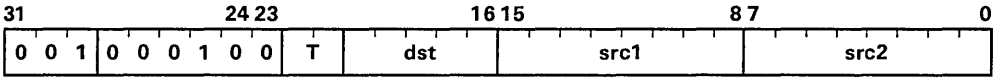
Operation *src1* AND ~*src2* → *dst*

Operands *src1* three-operand addressing modes (T):
0 0 register (Rn1, 0 ≤ n1 ≤ 27)
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 register (Rn1, 0 ≤ n1 ≤ 27)
1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):
0 0 register (Rn2, 0 ≤ n2 ≤ 27)
0 1 register (Rn2, 0 ≤ n2 ≤ 27)
1 0 indirect (disp = 0, 1, IR0, IR1)
1 1 indirect (disp = 0, 1, IO0, IR1)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The bitwise logical-AND between the *src1* operand and the bitwise logical complement (~) of the *src2* operand is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be unsigned integers.

Cycles 1

Status Bits
N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
ANDN3  R5,R3,R7
```

Before Instruction:

R5 = 0A02h
R3 = 0C2Fh
R7 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R5 = 0A02h
R3 = 0C2Fh
R7 = 042Dh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example

```
ANDN3  R1,*AR5++(IR0),R0
```

Before Instruction:

R1 = 0CFh
AR5 = 809825h
IR0 = 5h
R0 = 0h
Data at 809825h = 0FFFh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 0CFh
AR5 = 80982Ah
IR0 = 5h
R0 = 0F30h
Data at 809825h = 0FFFh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

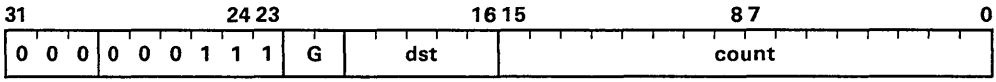
Syntax ASH <*count*>,<*dst*>

Operation
If (*count* ≥ 0):
  dst << *count* → *dst*
Else:
  dst >> |*count*| → *dst*

Operands
count general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (RN, 0 ≤ n ≤ 27)

Encoding



Description The seven least-significant bits of the *count* operand are used to generate the two's-complement shift count of up to 32 bits.

If the *count* operand is greater than zero, the *dst* operand is left-shifted by the value of the *count* operand. Low-order bits shifted in are zero-filled, and high-order bits are shifted out through the C (carry) bit.

Arithmetic left-shift:

C ← *dst* ← 0

If the *count* operand is less than zero, the *dst* operand is right-shifted by the absolute value of the *count* operand. The high-order bits of the *dst* operand are sign-extended as it is right-shifted. Low-order bits are shifted out through the C (carry) bit.

Arithmetic right-shift:

→ sign of *dst* → C

If the *count* operand is zero, no shift is performed, and the C (carry) bit is set to 0. The *count* and *dst* operands are assumed to be signed integers.

Cycles 1

Status Bits
N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C Set to the value of the last bit shifted out. 0 for a shift *count* of 0. Unaffected if *dst* is not R0 - R7.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example `ASH R1,R3`

Before Instruction:

R1 = 10h = 16
R3 = 0AE000h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 10h
R3 = 0E00000000h
LUF LV UF N Z V C = 0 1 0 1 0 1 0

Example `ASH @98C3h,R5`

Before Instruction:

DP = 80h
R5 = 0AEC00001h
Data at 8098C3h = 0FFE8 = -24
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R5 = 0FFFFFFAEh
Data at 8098C3h = 0FFE8 = -24
LUF LV UF N Z V C = 0 0 0 1 0 0 1

Syntax ASH3 <*count*>,<*src*>,<*dst*>

Operation

If (*count* ≥ 0):
  src << *count* → *dst*
Else:
  src >> |*count*| → *dst*

Operands

count three-operand addressing modes (T):

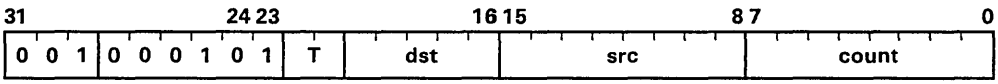
0 0 register (Rn1, 0 ≤ n1 ≤ 27)
0 1 direct (disp = 0, 1, IR0, IR1)
1 0 register (Rn1, 0 ≤ n1 ≤ 27)
1 1 indirect (disp = 0, 1, IR0, IR1)

src three-operand addressing modes (T):

0 0 register (Rn2, 0 ≤ n2 ≤ 27)
0 1 register (Rn2, 0 ≤ n2 ≤ 27)
1 0 indirect (disp = 0, 1, IR0, IR1)
1 1 indirect (disp = 0, 1, IO0, IR1)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The seven least-significant bits of the *count* operand are used to generate the two's-complement shift count of up to 32 bits.

If the *count* operand is greater than zero, the *src* operand is left-shifted by the value of the *count* operand. Low-order bits shifted in are zero-filled, and high-order bits are shifted out through the C (carry) bit.

Arithmetic left-shift:

C ← *src* ← 0

If the *count* operand is less than zero, the *src* operand is right-shifted by the absolute value of the *count* operand. The high-order bits of the *src* operand are sign-extended as it is right-shifted. Low-order bits are shifted out through the C (carry) bit.

Arithmetic right-shift:

→ sign of *dst* → C

If the *count* operand is zero, no shift is performed, and the C (carry) bit is set to 0. The *count*, *src*, and *dst* operands are assumed to be signed integers.

Cycles 1

Status Bits

N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C Set to the value of the last bit shifted out. 0 for a shift *count* of 0. Unaffected if *dst* is not R0 - R7.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit

OVM Operation not affected by OVM.

Example

```
ASH3  *AR3--(1),R5,R0
```

Before Instruction:

AR3 = 809921h
R5 = 02B0h
R0 = 0h
Data at 809921h = 10h = 16
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR3 = 809920h
R5 = 000002B0h
R0 = 02B00000h
Data at 809921h = 10h = 16
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example

```
ASH3  R1,R3,R5
```

Before Instruction:

R1 = 0FFFFFFF8h = -8
R3 = 0FFFFCB00h
R5 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 0FFFFFFF8h = -8
R3 = 0FFFFCB00h
R5 = 0FFFFFFCBh
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Syntax

ASH3 <*count*>,<*src2*>,<*dst1*>
|| STI <*src3*>,<*dst2*>

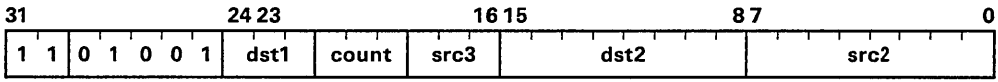
Operation

If (*count* ≥ 0):
  src2 << *count* → *dst1*
Else:
  src2 >> |*count*| → *dst1*
|| *src3* → *dst2*

Operands

count register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn2, 0 ≤ n2 ≤ 7)
src3 register (Rn3, 0 ≤ n3 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

The seven least-significant bits of the *count* operand register are used to generate the two's-complement shift count of up to 32 bits.

If the *count* operand is greater than zero, the *dst* operand is left-shifted by the value of the *count* operand. Low-order bits shifted in are zero-filled, and high-order bits are shifted out through the C (carry) bit.

Arithmetic left-shift:

C ← *src2* ← 0

If the *count* operand is less than zero, the *dst* operand is right-shifted by the absolute value of the *count* operand. The high-order bits of the *dst* operand are sign-extended as it is right-shifted. Low-order bits are shifted out through the C (carry) bit.

Arithmetic right-shift:

→ sign of *src2* → C

If the *count* operand is zero, no shift is performed, and the C (carry) bit is set to 0. The *count* and *dst* operands are assumed to be signed integers.

All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (ASH3) writes to the same register, then STI accepts as input the contents of the register before it is modified by the ASH3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Cycles

1

Status Bits

N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C Set to the value of the last bit shifted out. 0 for a shift *count* of 0.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit

OVM Operation not affected by OVM.

Example

```
   ASH3  R1,*AR6++(IR1),R0
|| STI   R5,*AR2
```

Before Instruction:

AR6 = 809900h
IR1 = 8Ch
R1 = 0FFE8h = -24
R0 = 0h
R5 = 35h = 53
AR2 = 8098A2h
Data at 809900h = 0AE000000h
Data at 8098A2h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

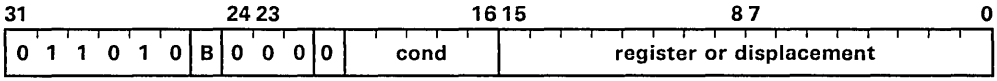
AR6 = 80998Ch
IR1 = 8Ch
R1 = 0FFE8h = -24
R0 = 0FFFFFFAEh
R5 = 35h = 53
AR2 = 8098A2h
Data at 809900h = 0AE000000h
Data at 8098A2h = 35h = 53
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Syntax B*cond* <*src*>

Operation If *cond* is true:
If *src* is in register addressing mode (Rn $0 \leq n \leq 27$),
src → PC.
If *src* is in PC-relative mode (label or address),
displacement + PC + 1 → PC.
Else, continue.

Operands *src* conditional-branch addressing modes (B):
0 register
1 PC-relative

Encoding



Description B*cond* signifies a standard branch that executes in four cycles. A branch is performed if the condition is true. If the *src* operand is expressed in register addressing mode, the contents of the specified register are loaded into the PC. If the *src* operand is expressed in PC-relative mode, the assembler generates a displacement: displacement = label - (PC of branch instruction + 1). This displacement is stored as a 16 bit signed integer in the 16 least significant bits of the branch instruction word. This displacement is added to the PC of the branch instruction plus 1 to generate the new PC.

The TMS320C30 provides 20 condition codes that can be used with this instruction (see Section 11.2 for a list of condition mnemonics, encoding, and flags).

Cycles 4

Status Bits
N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
BZ   R0
```

Before Instruction:

PC = 2B00h
R0 = 0003FF00h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

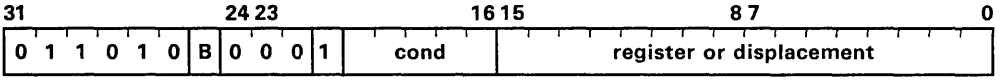
PC = 3FF00h
R0 = 0003FF00h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax B*cond*D *<src>*

Operation
If *cond* is true:
  If *src* is in register addressing mode (Rn $0 \leq n \leq 27$),
    src → PC.
  If *src* is in PC-relative mode (label or address),
    displacement + PC + 3 → PC.
Else, continue.

Operands *src* conditional-branch addressing modes (B):
0 register
1 PC-relative

Encoding



Description B*cond*D signifies a delayed branch that allows the three instructions after the delayed branch to be fetched before the PC is modified. The effect is a single-cycle branch.

A branch is performed if the condition is true. If the *src* operand is expressed in register addressing mode, the contents of the specified register are loaded into the PC. If the *src* operand is expressed in PC-relative mode, the assembler generates a displacement: displacement = label - (PC of branch instruction + 3). This displacement is stored as a 16 bit signed integer in the 16 least significant bits of the branch instruction. This displacement is added to the PC of the branch instruction plus 3 to generate the new PC. The TMS320C30 provides 20 condition codes that can be used with this instruction (see Section 11.2 for a list of condition mnemonics, encoding, and flags).

Cycles 1

Status Bits
N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example
```
BNZD  36  (36 = 24h)
```

Before Instruction:
PC = 50h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

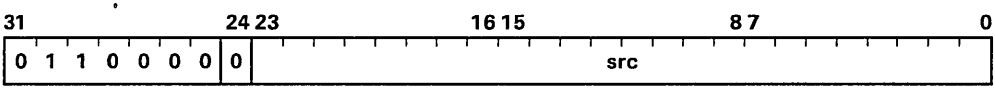
After Instruction:
PC = 77h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax BR *<src>*

Operation *src* → PC

Operands *src* long-immediate addressing mode

Encoding



Description BR signifies a standard branch that executes in four cycles. An unconditional branch is performed. The *src* operand is assumed to be a 24-bit unsigned integer. Note that bit 24 = 0 for a standard branch.

Cycles 4

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
BR   805Ch
```

Before Instruction:

PC = 80h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

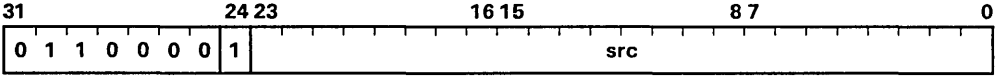
PC = 805Ch
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax BRD <*src*>

Operation *src* → PC

Operands *src* long-immediate addressing mode

Encoding



Description BRD signifies a delayed branch that allows the three instructions after the delayed branch to be fetched before the PC is modified. The effect is a single-cycle branch.

An unconditional branch is performed. The *src* operand is assumed to be a 24-bit unsigned integer. Note that bit 24 = 1 for a delayed branch.

Cycles 1

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
BRD  2Ch
```

Before Instruction:

PC = 1Bh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

PC = 2Ch
LUF LV UF N Z V C = 0 0 0 0 0 0 0

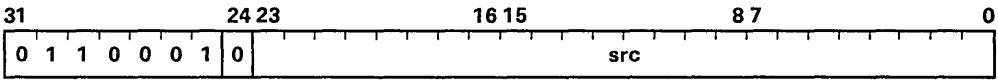
11

Syntax CALL *<src>*

Operation Next PC → *++SP
src → PC

Operands *src* long-immediate addressing mode

Encoding



Description A call is performed. The next PC value is pushed onto the system stack. The *src* operand is loaded into the PC. The *src* operand is assumed to be a 24-bit unsigned immediate operand.

Cycles 4

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
CALL  123456h
```

Before Instruction:

PC = 5h
SP = 809801h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

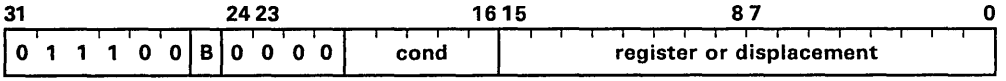
PC = 123456h
SP = 809802h
Data at 809802h = 6h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax CALL*cond* *<src>*

Operation If *cond* is true:
Next PC → *++SP
If *src* is in register addressing mode (Rn 0≤n≤27),
src → PC.
If *src* is in PC-relative mode (label or address),
displacement + PC + 1 → PC.
Else, continue.

Operands *src* conditional-branch addressing modes (B):
0 register
1 PC-relative

Encoding



Description A call is performed if the condition is true. If the condition is true, the next PC value is pushed onto the system stack. If the *src* operand is expressed in register addressing mode, the contents of the specified register are loaded into the PC. If the *src* operand is expressed in PC-relative mode, the assembler generates a displacement: displacement = label - (PC of call instruction + 1). This displacement is stored as a 16-bit signed integer in the 16 least significant bit of the call instruction word. This displacement is added to the PC of the call instruction plus 1 to generate the new PC.

The TMS320C30 provides 20 condition codes that can be used with this instruction (see Section 11.2 for a list of condition mnemonics, encoding, and flags).

Cycles 5

Status Bits
N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
CALLNZ  R5
```

Before Instruction:

PC = 123h
SP = 809835h
R5 = 789h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

PC = 789h
SP = 809836h
R5 = 789h
Data at 809836h = 124h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

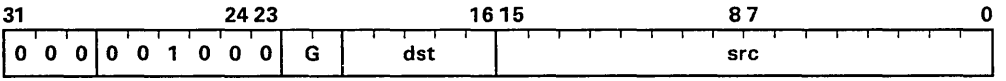
Syntax CMPF *<src>,<dst>*

Operation *dst - src*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The *src* operand is subtracted from the *dst* operand. The result is not loaded into any register, thus allowing for nondestructive compares. The *dst* and *src* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if a floating-point overflow occurs, 0 otherwise.
C Unaffected.
UF 1 if a floating-point underflow occurs, 0 otherwise.
LV 1 if a floating-point overflow occurs, unchanged otherwise.
LUF 1 if a floating-point underflow occurs, unchanged otherwise.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
CMPF  *+AR4,R6
```

Before Instruction:

AR4 = 8098F2h
R6 = 070C800000h = 1.4050e+02
Data at 8098F3h = 070C8000h = 1.4050e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR4 = 8098F2h
R6 = 070C800000h = 1.4050e+02
Data at 8098F3h = 070C8000h = 1.4050e+02
LUF LV UF N Z V C = 0 0 0 0 1 0 0

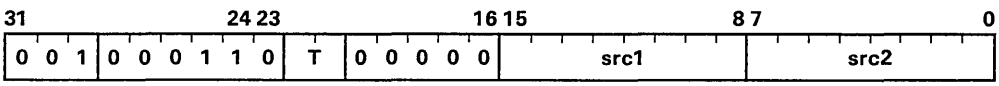
Syntax CMPF3 *<src2>,<src1>*

Operation *src1* - *src2*

Operands *src1* three-operand addressing modes (T):
0 0 register (Rn1, 0 ≤ n1 ≤ 7)
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 register (Rn1, 0 ≤ n1 ≤ 7)
1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):
0 0 register (Rn2, 0 ≤ n2 ≤ 7)
0 1 register (Rn2, 0 ≤ n2 ≤ 7)
1 0 indirect (disp = 0, 1, IR0, IR1)
1 1 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description The *src2* operand is subtracted from the *src1* operand. The result is not loaded into any register, thus allowing for nondestructive compares. The *src1* and *src2* operands are assumed to be floating-point numbers. Although this instruction has only two operands, it is designated as a three operand instruction since operands are specified in the three operand format.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if a floating-point overflow occurs, 0 otherwise.
C Unaffected.
UF 1 if a floating-point underflow occurs, 0 otherwise.
LV 1 if a flaoting-point overflow occurs, unchanged otherwise.
LUF 1 if a floating-point underflow occurs, unchanged otherwise.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
CMPF3   *AR2,*AR3--(1)
```

Before Instruction:

AR2 = 809831h
AR3 = 809852h
Data at 809831h = 77A7000h = 2.5044e+02
Data at 809852h = 57A2000h = 6.253125e+01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 809831h
AR3 = 809851h
Data at 809831h = 77A7000h = 2.5044e+02
Data at 809852h = 57A2000h = 6.253125e+01
LUF LV UF N Z V C = 0 0 0 1 0 0 0

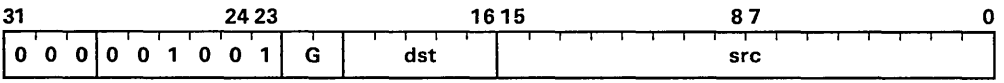
Syntax CMPI *<src>,<dst>*

Operation *dst - src*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The *src* operand is subtracted from the *dst* operand. The result is not loaded into any register, thus allowing for nondestructive compares. The *dst* and *src* operands are assumed to be signed integers.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C 1 if a borrow occurs, 0 otherwise.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
CMPI  R3,R7
```

Before Instruction:

R3 = 898h = 2200
R7 = 3E8h = 1000
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R3 = 898h = 2200
R7 = 3E8h = 1000
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Syntax CMPI3 *<src2>,<src1>*

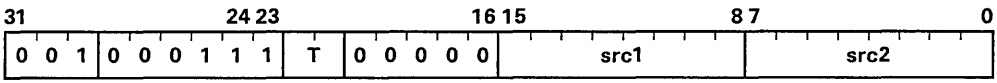
Operation *src1* - *src2*

Operands

src1 three-operand addressing modes (T):
- 0 0 register (Rn1, 0 ≤ n1 ≤ 27)
- 0 1 indirect (disp = 0, 1, IR0, IR1)
- 1 0 register (Rn1, 0 ≤ n1 ≤ 27)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):
- 0 0 register (Rn2, 0 ≤ n2 ≤ 27)
- 0 1 register (Rn2, 0 ≤ n2 ≤ 27)
- 1 0 indirect (disp = 0, 1, IR0, IR1)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description The *src2* operand is subtracted from the *src1* operand. The result is not loaded into any register, thus allowing for nondestructive compares. The *src1* and *src2* operands are assumed to be signed integers. Although this instruction has only two operands, it is designated as a three operand instruction since operands are specified in the three operand format.

Cycles 1

Status Bits
- **N** 1 if a negative result is generated, 0 otherwise.
- **Z** 1 if a zero result is generated, 0 otherwise.
- **V** 1 if an integer overflow occurs, 0 otherwise.
- **C** 1 if a borrow occurs, 0 otherwise.
- **UF** 0
- **LV** 1 if an integer overflow occurs, unchanged otherwise.
- **LUF** Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
CMPI3  R7,R4
```

Before Instruction:

R7 = 03E8h = 1000
R4 = 0898h = 2200
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R7 = 03E8h = 1000
R4 = 0898h = 2200
LUF LV UF N Z V C = 0 0 0 0 0 0 0

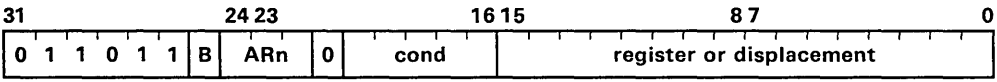
Syntax DB*cond* <ARn>,<*src*>

Operation
ARn - 1 → ARn
If *cond* is true and ARn ≥ 0 :
  If *src* is in register addressing mode (Rn 0≤n≤27)
    src → PC.
  If *src* is in PC-relative mode (label or address)
    displacement + PC + 1 → PC.
Else, continue.

Operands *src* conditional-branch addressing modes (B):
0 register
1 PC-relative

ARn register (0 ≤ n ≤ 7)

Encoding



Description DB*cond* signifies a standard branch that executes in four cycles. The specified auxiliary register is decremented and a branch is performed if the condition is true and the specified auxiliary register is greater than or equal to zero.

The auxiliary register is treated as a 24-bit signed integer. The most-significant eight bits are unmodified by the decrement operation. The comparison of the auxiliary register uses only the 24 least-significant bits of the auxiliary register. Note that the branch condition does not depend on the auxiliary register decrement.

If the *src* operand is expressed in register addressing mode, the contents of the specified register are loaded into the PC. If the *src* operand is expressed in PC-relative addressing mode, the assembler generates a displacement: displacement = label - (PC of branch instruction + 1). This integer is stored as a 16 bit signed integer in the 16 least significant bits of the branch instruction word. This displacement is added to the PC of the branch instruction plus 1 to generate the new PC.

The TMS320C30 provides 20 condition codes that can be used with this instruction (see Section 11.2 for a list of condition mnemonics, encoding, and flags).

Cycles 4

Status Bits
N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
DBLT  AR3,R2
```

Before Instruction:

PC = 5Fh
AR3 = 12h
R2 = 9Fh
LUF LV UF N Z V C = 0 0 0 1 0 0 0

After Instruction:

PC = 9Fh
AR3 = 11h
R2 = 9Fh
LUF LV UF N Z V C = 0 0 0 1 0 0 0

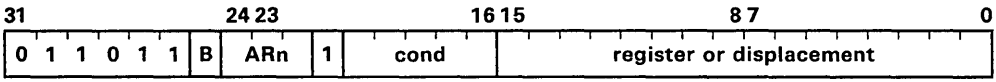
Syntax DB*cond*D <ARn>,<*src*>

Operation
ARn - 1 → ARn
If *cond* is true:
  If *src* is in register addressing mode (Rn 0≤n≤27)
    src → PC
  If *src* is in PC-relative mode (label or address)
    displacement + PC + 3 → PC.
Else, continue.

Operands *src* conditional-branch addressing modes (B):
0 register
1 PC-relative

ARn register (0 ≤ n ≤ 7)

Encoding



Description DB*cond*D signifies a delayed branch that allows the three instructions after the delayed branch to be fetched before the PC is modified. The effect is a single-cycle branch. The specified auxiliary register is decremented and a branch is performed if the condition is true and the specified auxiliary register greater than or equal to zero.

The auxiliary register is treated as a 24-bit signed integer. The most-significant eight bits are unmodified by the decrement operation. The comparison of the auxiliary register uses only the 24 least-significant bits of the auxiliary register. Note that the branch condition does not depend on the auxiliary register decrement.

If the *src* operand is expressed in register addressing mode, the contents of the specified register are loaded into the PC. If the *src* is expressed in PC-relative addressing, the assembler generates a displacement: displacement = label - (PC of branch instruction + 3). This displacement is added to the PC of the branch instruction plus 3 to generate the new PC. Note that bit 21 = 1 for a delayed branch.

The TMS320C30 provides 20 condition codes that can be used with this instruction (see Section 11.2 for a list of condition mnemonics, encoding, and flags).

Cycles 1

Status Bits
N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
DBZD  AR5,$+110h
```

Before Instruction:

PC = 0h
AR5 = 67h
LUF LV UF N Z V C = 0 0 0 0 1 0 0

After Instruction:

PC = 110h
AR5 = 66h
LUF LV UF N Z V C = 0 0 0 0 1 0 0

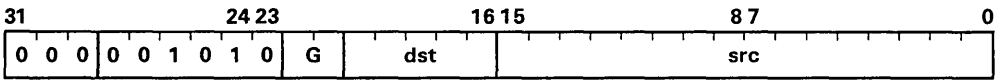
Syntax FIX <*src*>,<*dst*>

Operation fix(*src*) → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The floating-point operand *src* is converted to the nearest integer less than or equal to it in absolute value, and the result is loaded into the *dst* register. The *src* operand is assumed to be a floating-point number and the *dst* operand a signed integer.

The exponent field of the result register (if it has one) is not modified.

Integer overflow occurs when the floating-point number is too large to be represented as a 32-bit two's-complement integer. In the case of integer overflow, the result will be saturated in the direction of overflow.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C Unaffected.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
FIX  R1,R2
```

Before Instruction:

R1 = 0A282CCCCCh = -1.3454e+3
R2 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 0A282CCCCCh = -13454e+3
R2 = 541h = 1345
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

FIX <*src2*>,<*dst1*>
|| STI <*src3*>,<*dst2*>

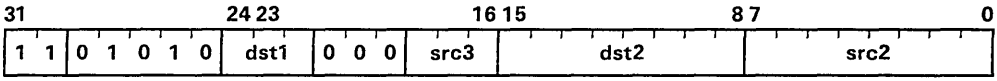
Operation

fix(*src2*) → *dst1*
|| *src3* → *dst2*

Operands

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src3 register (Rn2, 0 ≤ n2 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

A floating-point to integer conversion is performed. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register, and the operation being performed in parallel (FIX) writes to the same register, then STI accepts as input the contents of the register before it is modified by FIX.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Integer overflow occurs when the floating-point number is too large to be represented as a 32-bit two's-complement integer. In the case of integer overflow, the result will be saturated in the direction of overflow.

Cycles

1

Status Bits

N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C Unaffected.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit

OVM Operation affected by OVM.

Example

```
   FIX  *++AR4(1),R1
|| STI  R0,*AR2
```

Before Instruction:

AR4 = 8098A2h
R1 = 0h
R0 = 0DCh = 220
AR2 = 80983Ch
Data at 8098A3h = 733C000h = 1.7950e+02
Data at 80983Ch = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR4 = 8098A3h
R1 = 0B3h = 179
R0 = 0DCh = 220
AR2 = 80983Ch
Data at 8098A3h = 733C000h = 1.79750e+02
Data at 80983Ch = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

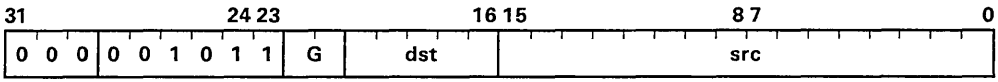
Syntax FLOAT <*src*>,<*dst*>

Operation float(*src*) → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The integer operand *src* is converted to the floating-point value equal to it, and the result loaded into the *dst* register. The *src* operand is assumed to be a signed integer, and the *dst* operand a floating-point number.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
FLOAT  *++AR2(2),R5
```

Before Instruction:

AR2 = 809800h
R5 = 034C2000h = 1.27578125e+01
Data at 809802h = 0AEh = 174
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 809802h
R5 = 072E00000h = 1.74e+02
Data at 809802h = 0AEh = 174
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

```
  FLOAT <src2>,<dst1>
|| STF  <src3>,<dst2>
```

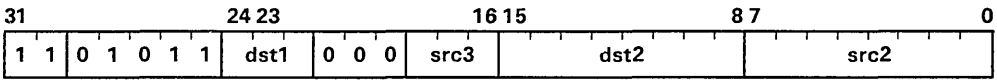
Operation

float(*src2*) → *dst1*
|| *src3* → *dst2*

Operands

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src3 register (Rn2, 0 ≤ n2 ≤ 7)
dst2 register (disp = 0, 1, IR0, IR1)

Encoding



Description

An integer to floating-point conversion is performed. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STF) reads from a register and the operation being performed in parallel (FLOAT) writes to the same register, then STF accepts as input the contents of the register before it is modified by FLOAT.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Cycles

1

Status Bits

N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit

OVM Operation not affected by OVM.

Example

```
   FLOAT  *+AR2(IR0),R6
|| STF  R7,*AR1
```

Before Instruction:

AR2 = 8098C5h
IR0 = 8h
R6 = 0h
R7 = 034C200000h = 1.27578125e+01
AR1 = 809933h
Data at 8098CDh = 0AEh = 174
Data at 809933h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

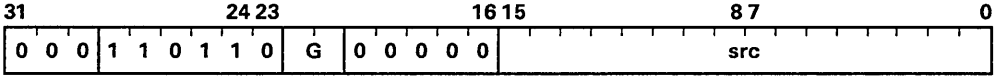
AR2 = 8098C5h
IR0 = 8h
R6 = 072E000000h = 1.740e+02
R7 = 034C200000h = 1.27578125e+01
AR1 = 809933h
Data at 8098CDh = 0AEh = 174
Data at 809933h = 034C2000h = 1.27578125e+01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax IACK *<src>*

Operation Perform a dummy read operation with $\overline{\text{IACK}}$ = 0.
At end of dummy read, set $\overline{\text{IACK}}$ to 1.

Operands *src* general addressing modes (G):
0 1 direct
1 0 indirect

Encoding



Description A dummy read operation is performed with $\overline{\text{IACK}}$ = 0. At the end of the dummy read, $\overline{\text{IACK}}$ is set to 1. This instruction can be used to generate an external interrupt acknowledge. If the address specified is off-chip, a read operation from that address is performed. The $\overline{\text{IACK}}$ signal and the address can then be used to signal interrupt acknowledge to external devices. The data read by the processor is unused.

Cycles 1

Status Bits

N	Unaffected.
Z	Unaffected.
V	Unaffected.
C	Unaffected.
UF	Unaffected.
LV	Unaffected.
LUF	Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
IACK   *AR5
```

Before Instruction:

$\overline{\text{IACK}}$ = 1
PC = 300h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

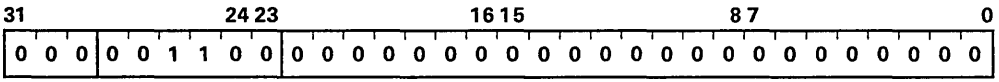
$\overline{\text{IACK}}$ = 1
PC = 301h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax IDLE

Operation 1 → ST(GIE)
Next PC → PC
Idle until interrupt.

Operands None

Encoding



Description The global interrupt enable bit is set, the next PC value is loaded into the PC, and the CPU idles until an interrupt is received. When the interrupt is received, the contents of the PC are pushed on the active system stack.

Cycles 1

Status Bits

N	Unaffected.
Z	Unaffected.
V	Unaffected.
C	Unaffected.
UF	Unaffected.
LV	Unaffected.
LUF	Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

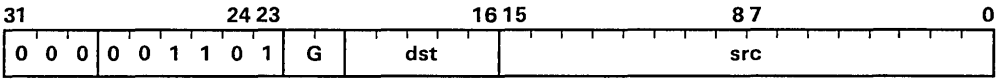
Syntax LDE <*src*>,<*dst*>

Operation *src*(exp) → *dst*(exp)

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The exponent field of the *src* operand is loaded into the exponent field of the *dst* register. No modification of the *dst* register mantissa field is made unless the value of the exponent loaded is the reserved value of the exponent for zero in the precision of the *src* operand. Then the mantissa field of the *dst* register is set to zero. The *src* and *dst* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits
N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
LDE  R0,R5
```

Before Instruction:

R0 = 0200056F30h = 4.00066337e+00
R5 = 0A056FE332h = 1.06749648e+03
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R0 = 0200056F30h = 4.00066337e+00
R5 = 02056FE332h = 4.16990814e+00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

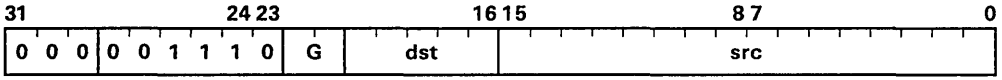
Syntax LDF *<src>,<dst>*

Operation *src* → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The *src* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
LDF  @9800h,R2
```

Before Instruction:

DP = 80h
R2 = 0h
Data at 809800h = 10C52A00h = 2.19254303e+00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R2 = 010C52A000h = 2.19254303e+00
Data at 809800h = 10C52A00h = 2.19254303e+00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

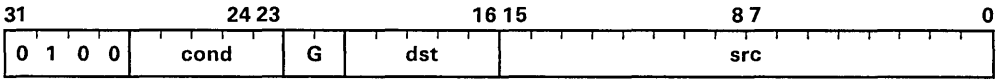
Syntax LDF*cond* <*src*>,<*dst*>

Operation If *cond* is true:
src → *dst*.
Else:
dst is unchanged.

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description If the condition is true, the *src* operand is loaded into the *dst* register. Otherwise, the *dst* register is unchanged. The *dst* and *src* operands are assumed to be floating-point numbers.

The TMS320C30 provides 20 condition codes that can be used with this instruction (see Section 11.2 for a list of condition mnemonics, encoding, and flags). Note that an LDFU (load floating-point unconditionally) instruction is useful for loading R0-R7 without affecting condition flags.

Cycles 1

Status Bits

N	Unaffected.
Z	Unaffected.
V	Unaffected.
C	Unaffected.
UF	Unaffected.
LV	Unaffected.
LUF	Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
LDFZ  R3,R5
```

Before Instruction:

R3 = 2CFF2CD500h = 1.77055560e+13
R5 = 5F0000003Eh = 3.96140824e+28
LUF LV UF N Z V C = 0 0 0 0 1 0 0

After Instruction:

R3 = 2CFF2CD500h = 1.77055560e+13
R5 = 2CFF2CD500h = 1.77055560e+13
LUF LV UF N Z V C = 0 0 0 0 1 0 0

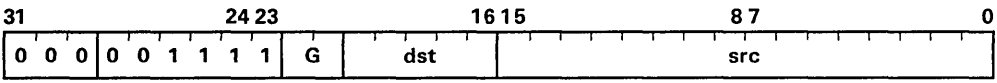
Syntax LDFI *<src>,<dst>*

Operation Signal interlocked operation.
src → *dst*

Operands *src* general addressing modes (G):
0 1 direct
1 0 indirect

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The *src* operand is loaded into the *dst* register. An interlocked operation is signaled over XF0 and XF1. The *src* and *dst* operands are assumed to be floating-point numbers. Note that only direct and indirect modes are allowed. Refer to Section 7.3 for detailed description.

Cycles 1 if XF1 = 0 (see Section 7.3)

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
LDFI   *+AR2,R7
```

Before Instruction:

AR2 = 8098F1h
R7 = 0h
Data at 8098F2h = 584C000h = -6.28125e+01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 8098F1h
R7 = 0584C00000h = -6.28125e+01
Data at 8098F2h = 584C000h = -6.28125e+01
LUF LV UF N Z V C = 0 0 0 0 0 0 1

Syntax

LDF <*src2*>,<*dst2*>
|| LDF <*src1*>,<*dst1*>

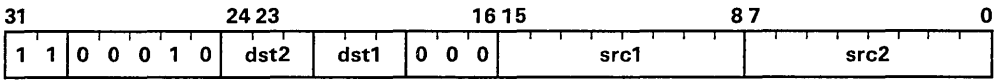
Operation

src2 → *dst2*
|| *src1* → *dst1*

Operands

src1 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst2 register (Rn2, 0 ≤ n2 ≤ 7)

Encoding



Description

Two floating-point loads are performed in parallel. If the LDFs load the same register, the assembler issues a warning. The result is that of LDF <*src2*>,<*dst2*>.

Cycles

1

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit

OVM Operation not affected by OVM.

Example

```
   LDF   *--AR1(IR0),R7
|| LDF   *AR7++(1),R3
```

Before Instruction:

AR1 = 80985Fh
IR0 = 8h
R7 = 0h
AR7 = 80988Ah
R3 = 0h
Data at 809857h = 70C8000h = 1.4050e+02
Data at 80988Ah = 57B4000h = 6.281250e+01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 809857h
IR0 = 8h
R7 = 070C800000h = 1.4050e+02
AR7 = 80988Bh
R3 = 057B400000h = 6.281250e+01
Data at 809857h = 70C8000h = 1.4050e+02
Data at 80988Ah = 57B4000h = 6.281250e+01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

```
  LDF <src2>,<dst1>
|| STF <src3>,<dst2>
```

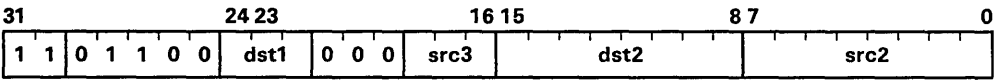
Operation

```
  src2 → dst1
|| src3 → dst2
```

Operands

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src3 register (Rn2, 0 ≤ n2 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

A floating-point load and a floating-point store are performed in parallel.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Cycles

1

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit

OVM Operation not affected by OVM.

Example

```
   LDF  *AR2--(1),R1
|| STF  R3,*AR4++(IR1)
```

Before Instruction:

AR2 = 8098E7h
R1 = 0h
R3 = 057B400000h = 6.28125e+01
AR4 = 809900h
IR1 = 10h
Data at 8098E7h = 70C8000h = 1.4050e+02
Data at 809900h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 8098E6h
R1 = 070C800000h = 1.4050e+02
R3 = 057B400000h = 6.28125e+01
AR4 = 809910h
IR1 = 10h
Data at 8098E7h = 70C8000h = 1.4050e+02
Data at 809900h = 57B4000h = 6.28125e+01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

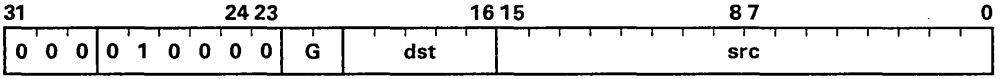
Syntax LDI *<src>,<dst>*

Operation *src* → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The *src* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers. An alternate form of LDI, LDP, is used to load the data page pointer register (DP), or any other register with the eight MSBs of a relocatable address. See Section 11.3.2.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
LDI  *-AR1(IR0),R5
```

Before Instruction:

AR1 = 2Ch
IR0 = 5h
R5 = 3C5h = 965
Data at 27h = 26h = 38
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 2Ch
IR0 = 5h
R5 = 26h = 38
Data at 27h = 26h = 38
LUF LV UF N Z V C = 0 0 0 0 0 0 0

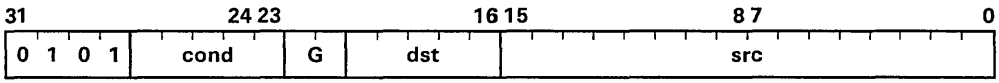
Syntax LDI*cond* <*src*>,<*dst*>

Operation If *cond* is true:
src → *dst*,
Else:
dst is unchanged.

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description If the condition is true, the *src* operand is loaded into the *dst* register. Otherwise, the *dst* register is unchanged. The *dst* and *src* operands are assumed to be signed integers.

The TMS320C30 provides 20 condition codes that can be used with this instruction (see Section 11.2 for a list of condition mnemonics, encoding, and flags). Note that an LDIU (load integer unconditionally) instruction is useful for loading R0-R7 without affecting the condition flags.

Cycles 1

Status Bits

N	Unaffected.
Z	Unaffected.
V	Unaffected.
C	Unaffected.
UF	Unaffected.
LV	Unaffected.
LUF	Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
LDIZ  R4,R6
```

Before Instruction:

R4 = 027Ch = 636
R6 = 0FE2h = 4,066
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R4 = 027Ch = 636
R6 = 0FE2h = 4,066
LUF LV UF N Z V C = 0 0 0 0 0 0 0

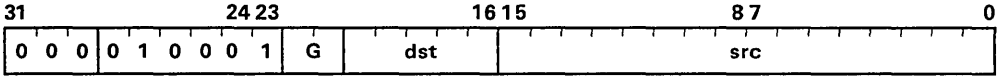
Syntax LDII *<src>,<dst>*

Operation Signal interlocked operation.
src → *dst*

Operands *src* general addressing modes (G):
0 1 direct
1 0 indirect

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The *src* operand is loaded into the *dst* register. An interlocked operation is signaled over XF0 and XF1. The *src* and *dst* operands are assumed to be signed integers. Note that only the direct and indirect modes are allowed. Refer to Section 7.3 for detailed description.

Cycles 1 if XF = 0 (see Section 7.3)

Status Bits

N 1 if a negative result is generated,0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
LDII  @985Fh,R3
```

Before Instruction:

DP = 80
R3 = 0h
Data at 80985Fh = 0DCh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80
R3 = 0DCh
Data at 80985Fh = 0DCh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

LDI <*src2*>,<*dst2*>
|| LDI <*src1*>,<*dst1*>

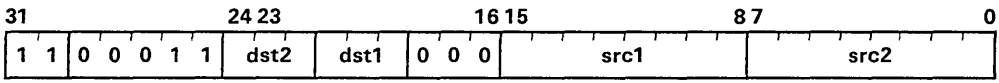
Operation

src2 → *dst2*
|| *src1* → *dst1*

Operands

src1 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst2 register (Rn2, 0 ≤ n2 ≤ 7)

Encoding



Description

Two integer loads are performed in parallel. A warning is issued by the assembler if the LDIs load the same register. The result is that of LDI <*src2*>,<*dst2*>.

Cycles

1

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit

OVM Operation not affected by OVM.

Example

```
   LDI   *-AR1(1),R7
|| LDI   *AR7++(IR0),R1
```

Before Instruction:

AR1 = 809826h
R7 = 0h
AR7 = 8098C8h
IR0 = 10h
R1 = 0h
Data at 809825h = 0FAh = 250
Data at 8098C8h = 2EEh = 750
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 809826h
R7 = 0FAh = 250
AR7 = 8098D8h
IR0 = 10h
R1 = 02EEh = 750
Data at 809825h = 0FAh = 250
Data at 8098C8h = 2EEh = 750
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

LDI <*src2*>,<*dst1*>
|| STI <*src3*>,<*dst2*>

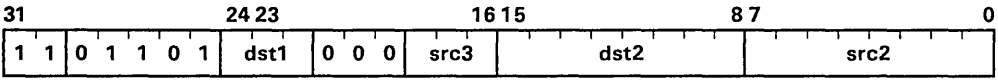
Operation

src2 → *dst1*
|| *src3* → *dst2*

Operands

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src3 register (Rn2, 0 ≤ n2 ≤7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

An integer load and an integer store are performed in parallel.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Cycles

1

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit

OVM Operation not affected by OVM.

Example

```
   LDI   *-AR1(1),R2
|| STI   R7,*AR5++(IR0)
```

Before Instruction:

AR1 = 8098E7h
R2 = 0h
R7 = 35h = 53
AR5 = 80982Ch
IR0 = 8h
Data at 8098E6h = 0DCh = 220
Data at 80982Ch = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 8098E7h
R2 = 0DCh = 220
R7 = 35h = 53
AR5 = 809834h
IR0 = 8h
Data at 8098E6h = 0DCh = 220
Data at 80982Ch = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0

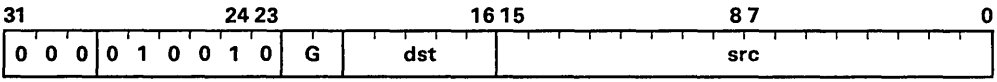
Syntax LDM *<src>,<dst>*

Operation *src*(man) → *dst*(man)

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The mantissa field of the *src* operand is loaded into the mantissa field of the *dst* register. The *dst* exponent field is not modified. The *src* and *dst* operands are assumed to be floating-point numbers. If immediate addressing mode is used, bits 15 - 12 of the instruction word are forced to 0 by the assembler.

Cycles 1

Status Bits

N	Unaffected.
Z	Unaffected.
V	Unaffected.
C	Unaffected.
UF	Unaffected.
LV	Unaffected.
LUF	Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example `LDM 156.75,R2` (156.75 = 071CC00000h)

Before Instruction:

R2 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R2 = 001CC00000h = 1.22460938e+00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

11

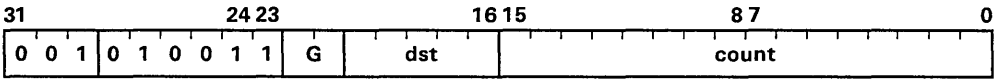
Syntax LSH <*count*>,<*dst*>

Operation
If *count* ≥ 0:
  dst << *count* → *dst*
Else:
  dst >> |*count*| → *dst*

Operands
src general addressing modes (G):
  0 0 register (Rn, 0 ≤ n ≤ 27)
  0 1 direct
  1 0 indirect
  1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The seven least-significant bits of the *count* operand are used to generate the two's-complement shift count. If the *count* operand is greater than zero, the *dst* operand is left- shifted by the value of the *count* operand. Low-order bits shifted in are zero-filled, and high-order bits are shifted out through the C (carry) bit.

Logical left-shift:

C ← *dst* ← 0

If the *count* operand is less than zero, the *dst* is right-shifted by the absolute value of the *count* operand. The high-order bits of the *dst* operand are zero-filled as shifted to the right. Low-order bits are shifted out through the C (carry) bit.

Logical right-shift:

0 → *dst* → C

If the *count* operand is 0, no shift is performed and the C (carry) bit is set to 0. The *count* operand is assumed to be a signed integer and the *dst* operand is assumed to be an unsigned integer.

Cycles 1

Status Bits
N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Set to the value of the last bit shifted out. 0 for a shift *count* of 0. Unaffected if *dst* is not R0-R7.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example `LSH R4,R7`

Before Instruction:

R4 = 018h = 24
R7 = 02ACh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R4 = 018h = 24
R7 = 0AC000000h
LUF LV UF N Z V C = 0 0 0 1 0 1 0

Example `LSH *-AR5(IR1),R5`

Before Instruction:

AR5 = 809908h
IR0 = 4h
R5 = 0012C00000h
Data at 809904h = 0FFFFFFF4h = -12
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR5 = 809908h
IR0 = 4h
R5 = 0000012C00h
Data at 809904h = 0FFFFFFF4h = -12
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax LSH3 <*count*>,<*src*>,<*dst*>

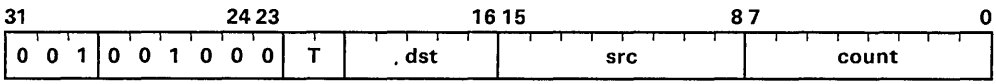
Operation
If *count* ≥ 0:
  src << *count* → *dst*
Else:
  src >> |*count*| → *dst*

Operands
src three-operand addressing modes (T):
0 0 register (Rn1, 0 ≤ n ≤ 27)
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 register (Rn1, 0 ≤ n1 ≤ 26)
1 1 indirect (disp = 0, 1, IR0, IR1)

count three-operand addressing modes (T):
0 0 register (Rn2, 0 ≤ n2 ≤ 27)
0 1 register (Rn2, 0 ≤ n2 ≤ 27)
1 0 indirect (disp = 0, 1, IR0, IR1)
1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The seven least-significant bits of the *count* operand are used to generate the two's-complement shift count.

If the *count* operand is greater than zero, the *dst* operand is left-shifted by the value of the *count* operand. Low-order bits shifted in are zero-filled, and high-order bits are shifted out through the C (carry) bit.

Logical left-shift:

C ← *src* ← 0

If the *count* operand is less than zero, the *src* operand is right-shifted by the absolute value of the *count* operand. The high-order bits of the *dst* operand are zero-filled as shifted to the right. Low-order bits are shifted out through the C (carry) bit.

Logical right-shift:

0 → *src* → C

If the *count* operand is 0, no shift is performed and the C (carry) bit is set to 0. The *count* operand is assumed to be a signed integer. The *src* and *dst* operands are assumed to be unsigned integers.

Cycles 1

Status Bits

N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Set to the value of the last bit shifted out. 0 for a shift *count* of 0. Unaffected if *dst* is not R0-R7.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit

OVM Operation not affected by OVM.

Example

```
LSH3  R4,R7,R2
```

Before Instruction:

R4 = 018h = 24
R7 = 02ACh
R2 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R4 = 018h = 24
R7 = 02ACh
R2 = 0AC000000h
LUF LV UF N Z V C = 0 0 0 1 0 1 0

Example

```
LSH3  *-AR4(IR1)R5,R3
```

Before Instruction:

AR4 = 809908h
IR1 = 4h
R5 = 012C00000h
R3 = 0h
Data at 809904h = 0FFFFFFF4h = -12
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR4 = 809908h
IR1 = 4h
R5 = 012C00000h
R3 = 0000012C00h
Data at 809904h = 0FFFFFFF4h = -12
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

LSH3 <*count*>,<*src2*>,<*dst1*>
|| STI <*src3*>,<*dst2*>

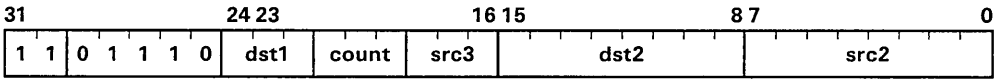
Operation

If *count* ≥ 0:
  src2 << *count* → *dst1*
Else:
  src2 >> |*count*| → *dst1*
|| *src3* → *dst2*

Operands

count register (Rn1, 0 ≤ n1 ≤ 7)
src1 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn3, 0 ≤ n3 ≤ 7)
src2 register (Rn4, 0 ≤ n4 ≤7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

The seven least-significant bits of the *count* operand are used to generate the two's-complement shift count.

If the *count* operand is greater than zero, the *dst* operand is left-shifted by the value of the *count* operand. Low- order bits shifted in are zero-filled and high-order bits are shifted out through the C (carry) bit.

Logical left-shift:

C ← *dst2* ← 0

If the *count* operand is less than zero, the *dst* operand is right-shifted by the absolute value of the *count* operand. The high-order bits of the *dst* operand are zero filled as shifted to the right. Low-order bits are shifted out through the C (carry bit).

Logical right-shift:

0 → *dst2* → C

If the *count* operand is 0, no shift is performed and the carry bit is set to 0.

The *count* operand is assumed to be a 7-bit signed integer and the *src2* and *dst1* operands are assumed to be unsigned integers. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (LSH3) writes to the same register, then STI accepts as input the contents of the register before it is modified by the LSH3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Cycles

1

Status Bits

N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Set to the value of the last bit shifted out. 0 for a shift *count* of 0.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit

OVM Operation not affected by OVM.

Example

```
   LSH3  R2,*++AR3(1),R0
|| STI   R4,*-AR3
```

Before Instruction:

R2 = 18h = 24
AR3 = 8098C2h
R0 = 0h
R4 = 0DCh = 220
AR3 = 8098A3h
Data at 8098C3h = 0ACh
Data at 8098A2h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R2 = 18h = 24
AR3 = 8098C3h
R0 = 0AC000000h
R4 = 0DCh = 220
AR3 = 8098A3h
Data at 8098C3h = 0ACh
Data at 8098A2h = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Example

```
   LSH3  R7,*AR2--(1),R2
|| STI   R0,*+AR0(1)
```

Before Instruction:

R7 = 0FFFFFFF4h = -12
AR2 = 809863h
R2 = 0h
R0 = 12Ch = 300
AR0 = 8098B7h
Data at 809863h = 2C000000h
Data at 8098B8h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R7 = 0FFFFFFF4h = -12
AR2 = 809862h
R2 = 2C000h
R0 = 12Ch = 300
AR0 = 8098B7h
Data at 809863h = 2C000000h
Data at 8098B8h = 12Ch = 300
LUF LV UF N Z V C = 0 0 0 0 0 0 0

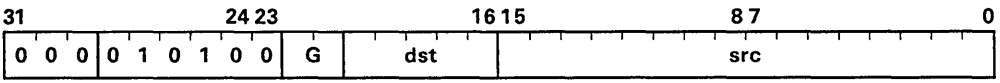
Syntax MPYF <*src*>,<*dst*>

Operation *dst* × *src* → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The product of the *dst* and *src* operands is loaded into the *dst* register. The *src* operand is assumed to be a single-precision floating-point number, and the *dst* operand is an extended-precision floating-point number.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if a floating-point is overflow occurs, 0 otherwise.
C Unaffected.
UF 1 if a floating-point underflow occurs, 0 otherwise.
LV 1 if a floating-point overflow occurs, unchanged otherwise.
LUF 1 if a floating-point underflow occurs, unchanged otherwise.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
MPYF  R0,R2
```

Before Instruction:

R0 = 070C800000h = 1.4050e+02
R2 = 034C200000h = 1.27578125e+01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R0 = 070C800000h = 1.4050e+02
R2 = 0A600F2000h = 1.79247266e+03
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax MPYF3 <*src2*>,<*src1*>,<*dst*>

Operation *src1* × *src2* → *dst*

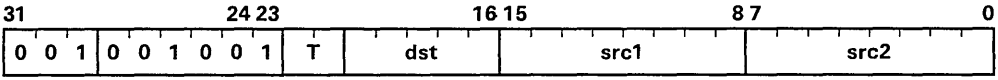
Operands

src1 three-operand addressing modes (T):
- 0 0 register (Rn1, 0 ≤ n1 ≤ 7)
- 0 1 indirect (disp = 0, 1, IR0, IR1)
- 1 0 register (Rn1, 0 ≤ n1 ≤ 7)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):
- 0 0 register (Rn2, 0 ≤ n2 ≤ 7)
- 0 1 register (Rn2, 0 ≤ n2 ≤ 7)
- 1 0 indirect (disp = 0, 1, IR0, IR1)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The product of the *dst1* and *src2* operands is loaded into the *dst* register. The *src1* and *src2* operands are assumed to be single-precision floating-point numbers, and the *dst* operand is an extended-precision floating-point number.

Cycles 1

Status Bits
- **N** 1 if a negative result is generated, 0 otherwise.
- **Z** 1 if a zero result is generated, 0 otherwise.
- **V** 1 if a floating-point overflow occurs, 0 otherwise.
- **C** Unaffected.
- **UF** 1 if a floating-point underflow occurs, 0 otherwise.
- **LV** 1 if a floating-point overflow occurs, unchanged otherwise.
- **LUF** 1 if a floating-point underflow occurs, unchanged otherwise.

Mode Bit **OVM** Operation not affected by OVM.

Example `MPYF3 R0,R7,R1`

Before Instruction:

R0 = 057B400000h = 6.281250e+01
R7 = 0733C00000h = 1.79750e+02
R1 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R0 = 057B400000h = 6.281250e+01
R7 = 0733C00000h = 1.79750e+02
R1 = 0D306A3000h = 1.12905469e+04
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example

```
MPYF3  *+AR2(IR0),R7,R2
or
MPYF3  R7,*+AR2(IR0),R2
```

Before Instruction:

AR2 = 809800h
IR0 = 12Ah
R7 = 057B400000h = 6.281250e+01
R2 = 0h
Data at 80992Ah = 70C8000h = 1.4050e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 809800h
IR0 = 12Ah
R7 = 057B400000h = 6.281250e+01
R2 = 0D09E4A000h = 8.82515625e+03
Data at 80992Ah = 70C8000h = 1.4050e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

MPYF3 <*srcA*>,<*srcB*>,<*dst1*>
|| ADDF3 <*srcC*>,<*srcD*>,<*dst2*>

Operation

srcA × *srcB* → *dst1*
|| *srcC* + *srcD* → *dst2*

Operands

srcA, *srcB*, *srcC*, *srcD*: Any two indirect (disp = 0,1,IR0,IR1); Any two register (0 ≤ ARn ≤ 7)

dst1 register (*d1*):
0 = R0
1 = R1

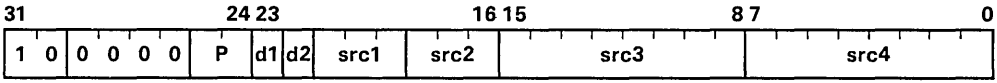
dst2 register (*d2*):
0 = R2
1 = R3

src1 register (Rn, 0 ≤ n ≤ 7)
src2 register (Rn, 0 ≤ n ≤ 7)
src3 indirect (disp = 0, 1, IR0, IR1)
src4 indirect (disp = 0, 1, IR0, IR1)

P parallel addressing modes (0 ≤ P ≤ 3)

	OPERATION
00	*src3* x *src4*, *src1* + *src2*
01	*src3* x *src1*, *src4* + *src2*
10	*src1* x *src2*, *src3* + *src4*
11	*src3* x *src1*, *src2* + *src4*

Encoding



Description

A floating-point multiplication and a floating-point addition are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (MPYF3) reads from a register and the operation being performed in parallel (ADDF3) writes to the same register, then MPYF3 accepts as input the contents of the register before it is modified by the ADDF3.

Any combination of addressing modes may be coded for the four possible source operands as long as the two are coded as indirect and two are register. The assignment of the source operands *srcA-srcD* to the *src1-src4* fields varies depending on the combination of addressing modes used, and the P field is encoded accordingly. The assembler may, when not significant, change the order of operands in commutative operations, in order to simplify processing.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Cycles 1

Status Bits

N 0
Z 0
V 1 if a floating-point overflow occurs, 0 otherwise.
C Unaffected.
UF 1 if a floating-point underflow occurs, 0 otherwise.
LV 1 if a floating-point overflow occurs, 0 unchanged otherwise.
LUF 1 if a floating-point underflow occurs, 0 unchanged otherwise.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
  MPYF3   *AR5++(1),*--AR1(IR0),R0
|| ADDF3   R5,R7,R3
```

Before Instruction:

AR5 = 8098C5h
AR1 = 8098A8h
IR0 = 4h
R0 = 0h
R5 = 0733C00000h = 1.79750e+02
R7 = 070C800000h = 1.4050e+02
R3 = 0h
Data at 8098C5h = 34C0000h = 1.2750e+01
Data at 8098A4h = 1110000h = 2.2500e+00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR5 = 8098C6h
AR1 = 8098A4h
IR0 = 4h
R0 = 0467180000h = 2.88867188e+01
R5 = 0733C00000h = 1.79750e+02
R7 = 070C800000h = 1.4050e+02
R3 = 0820200000h = 3.20250e+02
Data at 8098C5h = 34C0000h = 1.2750e+01
Data at 8098A4h = 1110000h = 2.2500e+00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

MPYF3 <*src2*>,<*src1*>,<*dst1*>
|| STF <*src3*>,<*dst2*>

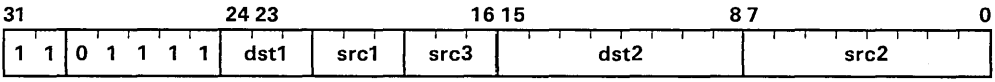
Operation

src1 × *src2*→ *dst1*
|| *src3* → *dst2*

Operands

src1 register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn3, 0 ≤ n3 ≤ 7)
src3 register (Rn4, 0 ≤ n4 ≤7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

A floating-point multiplication and a floating-point store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (MPYF3) reads from a register and the operation being performed in parallel (STF) writes to the same register, then MPYF3 accepts as input the contents of the register before it is modified by the STF.

If *src2* and *dst2* point to the same location, then *src2* is read before the write to *dst2*.

Cycles

1

Status Bits

N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if a floating-point overflow occurs, 0 otherwise.
C Unaffected.
UF 1 if a floating-point underflow occurs, 0 otherwise.
LV 1 if a floating-point overflow occurs, 0 unchanged otherwise.
LUF 1 if a floating-point underflow occurs, 0 unchanged otherwise.

Mode Bit

OVM Operation not affected by OVM.

Example

```
   MPYF3  *-AR2(1),R7,R0
|| STF   R3,*AR0--(IR0)
```

Before Instruction:

AR2 = 80982Bh
R7 = 057B400000h = 6.281250e+01
R0 = 0h
R3 = 086B280000h = 4.7031250e+02
AR0 = 809860h
IR0 = 8h
Data at 80982Ah = 70C8000h = 1.4050e+02
Data at 809860h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 80982Bh
R7 = 057B400000h = 6.281250e+01
R0 = 0D09E4A000h = 8.82515625e+03
R3 = 086B280000h = 4.7031250e+02
AR0 = 809858h
IR0 = 8h
Data at 80982Ah = 70C8000h = 1.4050e+02
Data at 809860h = 86B280000h = 4.7031250e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

MPYF3 *<srcA>,<srcB>,<dst1>*
|| SUBF3 *<srcC>,<srcD>,<dst2>*

Operation

srcA × *srcB* → *dst1*
|| *srcD* − *srcC* → *dst2*

Operands

srcA *srcB* *srcC* *srcD*	Any two indirect (disp = 0,1,IR0,IR1) Any two register (0 ≤ ARn ≤ 7)

dst1 register (*d1*):
0 = R0
1 = R1

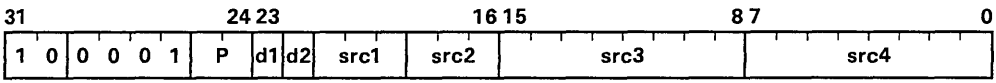
dst2 register (*d2*):
0 = R2
1 = R3

src1	register	(Rn, 0 ≤ n ≤ 7)
src2	register	(Rn, 0 ≤ n ≤ 7)
src3	indirect	(disp = 0, 1, IR0, IR1)
src4	indirect	(disp = 0, 1, IR0, IR1)

P parallel addressing modes (0 ≤ P ≤ 3)

	OPERATION
00	*src3* x *src4*, *src1* − *src2*
01	*src3* x *src1*, *src4* − *src2*
10	*src1* x *src2*, *src3* − *src4*
11	*src3* x *src1*, *src2* − *src4*

Encoding



11

Description A floating-point multiplication and a floating-point subtraction are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (MPYF3) reads from a register, and the operation being performed in parallel (SUBF3) writes to the same register, then MPYF3 accepts as input the contents of the register before it is modified by the SUBF3.

Any combination of addressing modes may be coded for the four possible source operands as long as the two are coded as indirect and two are register. The assignment of the source operands *srcA-srcD* to the *src1-src4* fields varies depending on the combination of addressing modes used, and the P field is encoded accordingly. The assembler may, when not significant, change the order of operands in commutative operations, in order to simplify processing.

Cycles 1

Status Bits

N 0
Z 0
V 1 if a floating-point overflow occurs, 0 otherwise.
C Unaffected.
UF 1 if a floating-point underflow occurs, 0 otherwise.
LV 1 if a floating-point overflow occurs, unchanged otherwise.
LUF 1 if a floating-point underflow occurs, unchanged otherwise.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
   MPYF3  R5,*++AR7(IR1),R0
|| SUBF3  R7,*AR3--(1),R2
```

or

```
   MPYF3  R5,*++AR7(IR1),R5,R0
|| SUBF3  R7,*AR3--(1),R2
```

Before Instruction:

R5 = 034C000000h = 1.2750e+01
AR7 = 809904h
IR1 = 8h
R0 = 0h
R7 = 0733C00000h = 1.79750e+02
AR3 = 8098B2h
R2 = 0h
Data at 80990Ch = 1110000h = 2.250e+00
Data at 8098B2h = 70C8000h = 1.4050e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R5 = 034C000000h = 1.2750e+01
AR7 = 80990Ch
IR1 = 8h
R0 = 0467180000h = 2.88867188e+01
R7 = 0733C00000h = 1.79750e+02
AR3 = 8098B1h
R2 = 05E3000000h = -3.9250e+01
Data at 80990Ch = 1110000h = 2.250e+00
Data at 8098B2h = 70C8000h = 1.4050e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

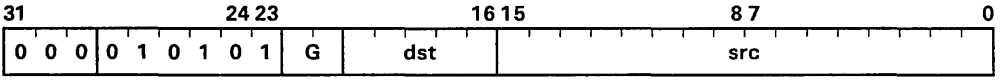
Syntax MPYI *<src>,<dst>*

Operation *dst* × *src* → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The product of the *dst* and *src* operands is loaded into the *dst* register. The *src* and *dst* operands when read are assumed to be 24-bit signed integers. The result is assumed to be a 48-bit signed integer. The output to the *dst* register is the 32 least-significant bits of the result.

Integer overflow occurs when any of the most-significant 16 bits of the 48-bit result differs from the most-significant bit of the 32-bit output value.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C Unaffected.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unchanged.

Mode Bit **OVM** Operation affected by OVM.

Example

```
MPYI  R1,R5
```

Before Instruction:

R1 = 000033C251h = 3,392,081
R5 = 000078B600h = 7,910,912
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 000033C251h = 3,392,081
R5 = 00E21D9600h = -501,377,536
LUF LV UF N Z V C = 0 1 0 1 0 1 0

Syntax MPYI3 *<src2>,<src1>,<dst>*

Operation *src1* × *src2* → *dst*

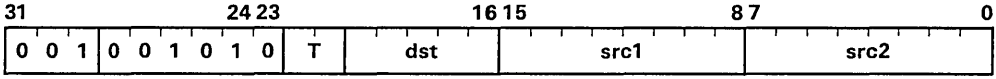
Operands

src1 three-operand addressing modes (T):
- 0 0 register (Rn1, ≤ n1 ≤ 27)
- 0 1 indirect (disp = 0, 1, IR0, IR1)
- 1 0 register (Rn1, ≤ n1 ≤ 27)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):
- 0 0 register (Rn2, ≤ n2 ≤ 27)
- 0 1 register (Rn2, ≤ n2 ≤ 27)
- 1 0 indirect (disp = 0, 1, IR0, IR1)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The product of the *src1* and *src2* operands is loaded into the *dst* register. The *src1* and *src2* operands are assumed to be 24-bit signed integers. The result is assumed to be a signed 48-bit integer. The output to the *dst* register is the 32 least-significant bits of the result.

Integer overflow occurs when any of the most-significant 16 bits of the 48-bit result differs from the most-significant bit of the 32-bit output value.

Cycles 1

Status Bits
- **N** 1 if a negative result is generated, 0 otherwise.
- **Z** 1 if a zero result is generated, 0 otherwise.
- **V** 1 if an integer overflow occurs, 0 otherwise.
- **C** Unaffected.
- **UF** 0
- **LV** 1 if an integer overflow occurs, unchanged otherwise.
- **LUF** Unchanged.

Mode Bit **OVM** Operation affected by OVM.

Example `MPYI3 *AR4,*-AR1(1),R2`

Before Instruction:

AR4 = 809850h
AR1 = 8098F3h
R2 = 0h
Data at 809850h = 0ADh = 173
Data at 8098F2h = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR4 = 809850h
AR1 = 8098F3h
R2 = 094ACh = 38,060
Data at 809850h = 0ADh = 173
Data at 8098F2h = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example `MPYI3 *--AR4(IR0),R2,R7`

Before Instruction:

AR4 = 8099F8h
IR0 = 8h
R2 = 0C8h = 200
R7 = 0h
Data at 8099F0h = 32h = 50
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR4 = 8099F0h
IR0 = 8h
R2 = 0C8h = 200
R7 = 02710h = 10,000
Data at 8099F0h = 32h = 50
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

MPYI3 *<srcA>,<srcB>,<dst1>*
|| ADDI3 *<srcC>,<srcD>,<dst2>*

Operation

srcA × *srcB* → *dst1*
|| *srcD* + *srcC* → *dst2*

Operands

srcA, *srcB*, *srcC*, *srcD* — Any two indirect (disp = 0,1,IR0,IR1) / Any two register (0 ≤ ARn ≤ 7)

dst1 — register (*d1*):
0 = R0
1 = R1

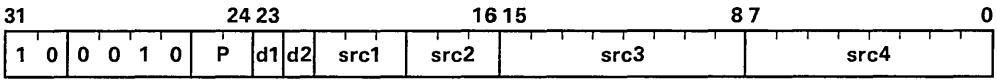
dst2 — register (*d2*):
0 = R2
1 = R3

src1	register	(Rn, 0 ≤ n ≤ 7)
src2	register	(Rn, 0 ≤ n ≤ 7)
src3	indirect	(disp = 0, 1, IR0, IR1)
src4	indirect	(disp = 0, 1, IR0, IR1)

P — parallel addressing modes (0 ≤ P ≤ 3)

	OPERATION
00	*src3* x *src4*, *src1* + *src2*
01	*src3* x *src1*, *src4* + *src2*
10	*src1* x *src2*, *src3* + *src4*
11	*src3* x *src1*, *src2* + *src4*

Encoding



Description An integer multiplication and an integer addition are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (MPYI3) reads from a register and the operation being performed in parallel (ADDI3) writes to the same register, then MPYI3 accepts as input the contents of the register before it is modified by the ADDI3.

Any combination of addressing modes may be coded for the four possible source operands as long as the two are coded as indirect and two are register. The assignment of the source operands *srcA-srcD* to the *src1-src4* fields varies depending on the combination of addressing modes used, and the P field is encoded accordingly. The assembler may, when not significant, change the order of operands in commutative operations, in order to simplify processing.

Cycles 1

Status Bits

N 0
Z 0
V 1 if an integer overflow occurs, 0 otherwise.
C Unaffected.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unchanged.

Mode Bit **OVM** Operation affected by OVM.

Example

```
   MPYI3  R7,R4,R0
|| ADDI3  *-AR3,*AR5--(1),R3
```

Before Instruction:

R7 = 14h = 20
R4 = 64h = 100
R0 = 0h
AR3 = 80981Fh
AR5 = 80996Eh
R3 = 0h
Data at 80981Eh = 0FFFFFFCBh = -53
Data at 80996Eh = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R7 = 14h = 20
R4 = 64h = 100
R0 = 07D0h = 2000
AR3 = 80981Fh
AR5 = 80996Dh
R3 = 0h
Data at 80981Eh = 0FFFFFFCBh = -53
Data at 80996Eh = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

MPYI3 <*src2*>,<*src1*>,<*dst1*>
|| STI <*src3*>,<*dst2*>

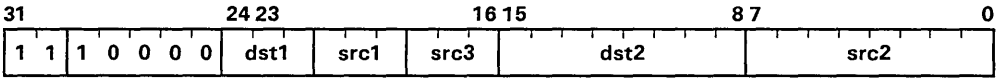
Operation

src1 × *src2* → *dst1*
|| *src3* → *dst2*

Operands

src1 register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn3, 0 ≤ n3 ≤ 7)
src3 register (Rn4, 0 ≤ n4 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

An integer multiplication and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (MPYI3) writes to the same register, then STI accepts as input the contents of the register before it is modified by the MPYI3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Integer overflow occurs when any of the most-significant 16 bits of the 48-bit result differs from the most-significant bit of the 32-bit output value.

Cycles

1

Status Bits

N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C Unaffected.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit

OVM Operation affected by OVM.

Example

```
   MPYI3  *++AR0(1),R5,R7
|| STI   R2,*-AR3(1)
```

Before Instruction:

AR0 = 80995Ah
R5 = 32h = 50
R7 = 0h
R2 = 0DCh = 220
AR3 = 80982Fh
Data at 80995Bh = 0C8h = 200
Data at 80982Eh = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR0 = 80995Bh
R5 = 32h = 50
R7 = 2710h = 10000
R2 = 0DCh = 220
AR3 = 80982Fh
Data at 80995Bh = 0C8h = 200
Data at 80982Eh = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

MPYI3 <*srcA*>,<*srcB*>,<*dst1*>
|| SUBI3 <*srcC*>,<*srcD*>,<*dst2*>

Operation

srcA × *srcB* → *dst1*
|| *srcD* − *srcC* → *dst2*

Operands

srcA *srcB* *srcC* *srcD*	Any two indirect (disp = 0,1,IR0,IR1) Any two register (0 ≤ ARn ≤ 7)

dst1 register (*d1*):
0 = R0
1 = R1

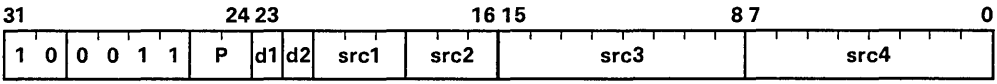
dst2 register (*d2*):
0 = R2
1 = R3

src1	register	(Rn, 0 ≤ n ≤ 7)
src2	register	(Rn, 0 ≤ n ≤ 7)
src3	indirect	(disp = 0, 1, IR0, IR1)
src4	indirect	(disp = 0, 1, IR0, IR1)

P parallel addressing modes (0 ≤ P ≤ 3)

	OPERATION
00	*src3* x *src4*, *src1* − *src2*
01	*src3* x *src1*, *src4* − *src2*
10	*src1* x *src2*, *src3* − *src4*
11	*src3* x *src1*, *src2* − *src4*

Encoding



Description

An integer multiplication and an integer subtraction are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (MPYI3) reads from a register and the operation being performed in parallel (SUBI3) writes to the same register, then MPYI3 accepts as input the contents of the register before it is modified by the SUBI3.

Any combination of addressing modes may be coded for the four possible source operands as long as the two are coded as indirect and two are register. The assignment of the source operands *srcA-srcD* to the *src1-src4* fields varies depending on the combination of addressing modes used, and the P field is encoded accordingly. The assembler may, when not significant, change the order of operands in commutative operations, in order to simplify processing.

Integer overflow occurs when any of the most-significant 16 bits of the 48-bit result differs from the most-significant bit of the 32-bit output value.

Cycles 1

Status Bits

N 0
Z 0
V 1 if an integer overflow occurs, 0 otherwise.
C Unaffected.
UF 1 if an integer underflow occurs, 0 otherwise.
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unchanged.

Mode Bit **OVM** Operation affected by OVM.

Example

```
   MPYI3  R2,*++AR0(1),R0
|| SUBI3  *AR5--(IR1),R4,R2
```

or

```
   MPYI3  *++AR0(1),R2,R0
|| SUBI3  *AR5--(IR1),R4,R2
```

Before Instruction:

R2 = 32h = 50
AR0 = 8098E3h
R0 = 0h
AR5 = 8099FCh
IR1 = 0Ch
R4 = 07D0h = 2000
Data at 8098E4h = 62h = 98
Data at 8099FCh = 4B0h = 1200
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R2 = 320h = 800
AR0 = 8098E4h
R0 = 01324h = 4900
AR5 = 8099F0h
IR1 = 0Ch
R4 = 07D0h = 2000
Data at 8098E4h = 62h = 98
Data at 8099FCh = 4B0h = 1200
LUF LV UF N Z V C = 0 0 0 0 0 0 0

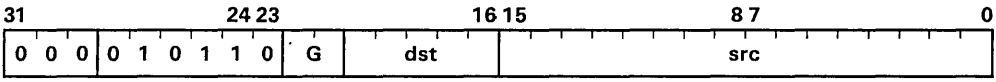
Syntax NEGB *<src>,<dst>*

Operation 0 - *src* - C → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The difference of the 0, *src*, and C operands is loaded into the *dst* register. The *dst* and *src* are assumed to be signed integers.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C 1 if a borrow occurs, 0 otherwise.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit **OVM** Operation affected by OVM.

Example

```
NEGB  R5,R7
```

Before Instruction:

R5 = 0FFFFFFCBh = -53
R7 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

R5 = 0FFFFFFCBh = -53
R7 = 34h = 52
LUF LV UF N Z V C = 0 0 0 0 0 0 1

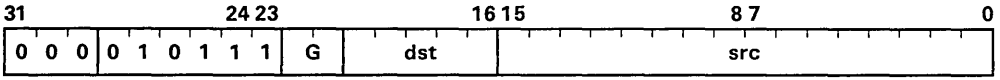
Syntax NEGF *<src>,<dst>*

Operation 0 - *src* → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The difference of the 0 and *src* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if a floating-point overflow occurs, 0 otherwise.
C Unaffected.
UF 1 if a floating-point underflow occurs, 0 otherwise.
LV 1 if a floating-point overflow occurs, unchanged otherwise.
LUF 1 if a floating-point underflow occurs, unchanged otherwise.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
NEGF  *++AR3(2),R1
```

Before Instruction:

AR3 = 809800h
R1 = 057B400025h = 6.28125006e+01
Data at 809802h = 70C8000h = 1.4050e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR3 = 809802h
R1 = 07F3800000h = -1.4050e+02
Data at 809802h = 70C8000h = 1.4050e+02
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Syntax

NEGF *<src2>,<dst1>*
|| STF *<src3>,<dst2>*

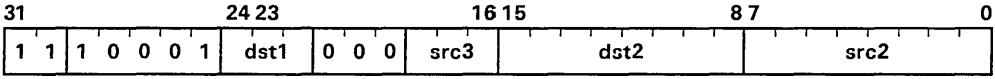
Operation

0 - *src2* → *dst1*
|| *src3* → *dst2*

Operands

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src3 register (Rn2, 0 ≤ n2 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

A floating-point negation and a floating-point store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STF) reads from a register and the operation being performed in parallel (NEGF) writes to the same register, then STF accepts as input the contents of the register before it is modified by the NEGF.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Cycles

1

Status Bits

N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if a floating-point overflow occurs, 0 otherwise.
C Unaffected.
UF 1 if a floating-point underflow occurs, 0 otherwise.
LV 1 if a floating-point overflow occurs, unchanged otherwise.
LUF 1 if a floating-point underflow occurs, unchanged otherwise.

Mode Bit

OVM Operation not affected by OVM.

Example

```
    NEGF   *AR4--(1),R7
||  STF    R2,*++AR5(1)
```

Before Instruction:

AR4 = 8098E1h
R7 = 0h
R2 = 0733C00000h = 1.79750e+02
AR5 = 809803h
Data at 8098E1h = 57B400000h = 6.281250e+01
Data at 809804h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR4 = 8098E0h
R7 = 0584C00000h = -6.281250e+01
R2 = 0733C00000h = 1.79750e+02
AR5 = 809804h
Data at 8098E1h = 57B4000h = 6.281250e+01
Data at 809804h = 733C000h = 1.79750e+02
LUF LV UF N Z V C = 0 0 0 1 0 0 0

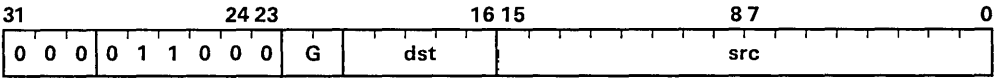
Syntax NEGI *<src>,<dst>*

Operation 0 - *src* → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The difference of the 0 and *src* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C 1 if a borrow occurs, 0 otherwise.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit **OVM** Operation affected by OVM.

Example

```
NEGI  174,R5   (174 = 0AEh)
```

Before Instruction:

R5 = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R5 = 0FFFFFF52 = -174
LUF LV UF N Z V C = 0 0 0 1 0 0 1

Syntax

NEGI *<src2>,<dst1>*
|| STI *<src3>,<dst2>*

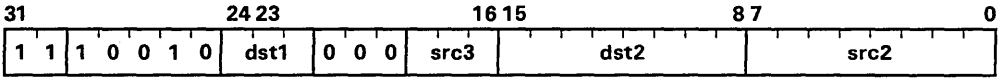
Operation

0 - *src2* → *dst1*
|| *src3* → *dst2*

Operands

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src3 register (Rn2, 0 ≤ n2 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

An integer negation and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (NEGI) writes to the same register, then STI accepts as input the contents of the register before it is modified by the NEGI.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Cycles

1

Status Bits

N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C 1 if a borrow occurs, 0 otherwise.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit

OVM Operation affected by OVM.

Example

```
   NEGI  *-AR3,R2
|| STI   R2,*AR1++
```

Before Instruction:

AR3 = 80982Fh
R2 = 19h = 25
AR1 = 8098A5h
Data at 80982Eh = 0DCh = 220
Data at 8098A5h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

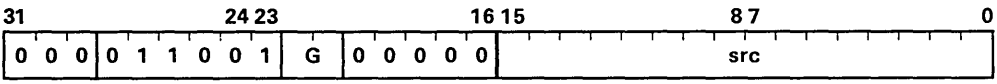
AR3 = 80982Fh
R2 = 0FFFFFF24h = -220
AR1 = 8098A6h
Data at 80982Eh = 0DCh = 220
Data at 8098A5h = 19h = 25
LUF LV UF N Z V C = 0 0 0 1 0 0 1

Syntax NOP *<src>*

Operation No ALU or multiplier operations.
ARn is modified if *src* is specified in indirect mode.

Operands *src* general addressing modes (G):
0 0 register (no operation)
1 0 indirect (modify ARn, 0 ≤ n ≤7)

Encoding



Description If the *src* operand is specified in the indirect mode, the specified addressing operation is performed and a dummy memory read occurs. If the *src* operand is omitted, no operation is performed.

Cycles 1

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
NOP
```

Before Instruction:

PC = 3Ah

After Instruction:

PC = 3Bh

Example

```
NOP  *AR3--(1)
```

Before Instruction:

PC = 5h
AR3 = 809900h

After Instruction:

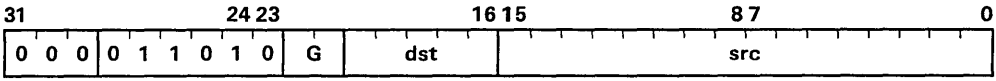
PC = 6h
AR3 = 8098FFh

Syntax NORM <*src*>,<*dst*>

Operation norm (*src*) → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

Encoding



Description The *src* operand is assumed to be an unnormalized floating-point number, i.e., the implied bit is set equal to the sign bit. The *dst* is set equal to the normalized *src* operand with the implied bit removed. The *dst* operand exponent is set to the *src* operand exponent minus the size of the left-shift necessary to normalize the *src*. The *dst* operand is assumed to be a normalized floating-point number.

If *src*(exp) = -128 and *src*(man) = 0, then *dst* = 0, Z = 1, and UF = 0. If *src*(exp) = -128 and *src*(man) ≠ 0, then *dst* = 0, Z = 0, and UF = 1. For all other cases of the *src*, if a floating-point underflow occurs, then *dst*(man) is forced to 0 and *dst*(exp) = -128. If *src*(man) = 0, then *dst*(man) = 0 and *dst*(exp) = -128. Refer to Section 5.6.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 0
C Unaffected.
UF 1 if a floating-point underflow occurs, 0 otherwise.
LV Unaffected.
LUF 1 if a floating-point underflow occurs, unchanged otherwise.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
NORM  R1,R2
```

Before Instruction:

R1 = 0400003AF5h
R2 = 070C800000h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 0400003AF5h
R2 = F26BD40000h = 1.12451613e-04
LUF LV UF N Z V C = 0 0 0 0 0 0 0

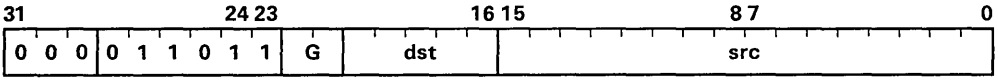
Syntax NOT <*src*>,<*dst*>

Operation ~*src* → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The bitwise logical-complement of the *src* operand is loaded into the *dst* register. The complement is formed by a logical-NOT of each bit of the *src* operand. The *dst* and *src* operands are assumed to be unsigned integers.

Cycles 1

Status Bits
N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
NOT  @982Ch,R4
```

Before Instruction:

DP = 80h
R4 = 0h
Data at 80982Ch = 5E2Fh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R4 = 0FFFFA1D0h
Data at 80982Ch = 5E2Fh
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Syntax

NOT <*src2*>,<*dst1*>
|| STI <*src3*>,<*dst2*>

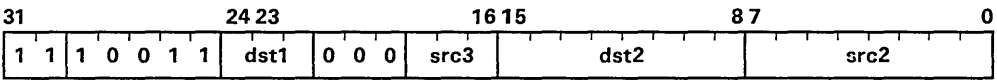
Operation

~*src2* → *dst1*
|| *src3* → *dst2*

Operands

src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn1, 0 ≤ n1 ≤ 7)
src3 register (Rn2, 0 ≤ n2 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

A bitwise logical-NOT and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (NOT) writes to the same register, then STI accepts as input the contents of the register before it is modified by the NOT.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Cycles

1

Status Bits

N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit

OVM Operation not affected by OVM.

Example

```
   NOT  *+AR2,R3
|| STI  R7,*--AR4(IR1)
```

Before Instruction:

AR2 = 8099CBh
R3 = 0h
R7 = 0DCh = 220
AR4 = 809850h
IR1 = 10h
Data at 8099CCh = 0C2Fh
Data at 809840h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 8099CBh
R3 = 0FFFFF3D0h
R7 = 0DCh = 220
AR4 = 809840h
IR1 = 10h
Data at 8099CCh = 0C2Fh
Data at 809840h = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Syntax OR *<src>,<dst>*

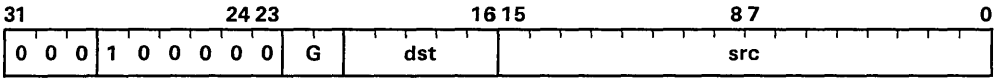
Operation *dst* OR *src* → *dst*

Operands *src* general addressing modes (G):

0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate (not sign-extended)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The bitwise logical-OR between the *src* and *dst* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be unsigned integers.

Cycles 1

Status Bits

N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example `OR *++AR1(IR1),R2`

Before Instruction:

AR1 = 809800h
IR1 = 4h
R2 = 012560000h
Data at 809804h = 2BCDh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 809804h
IR1 = 4h
R2 = 012562BCDh
Data at 809804h = 2BCDh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax OR3 *<src2>,<src1>,<dst>*

Operation *src1* OR *src2* → *dst*

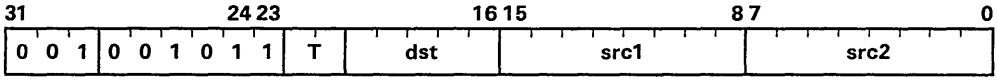
Operands

src1 three-operand addressing modes (T):
- 0 0 register (Rn1, 0 ≤ n1 ≤ 27)
- 0 1 indirect (disp = 0, 1, IR0, IR1)
- 1 0 register (Rn1, 0 ≤ n1 ≤ 27)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):
- 0 0 register (Rn2, 0 ≤ n2 ≤ 27)
- 0 1 register (Rn2, 0 ≤ n2 ≤ 27)
- 1 0 indirect (disp = 0, 1, IR0, IR1)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The bitwise logical-OR between the *src1* and *src2* operands is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be unsigned integers.

Cycles 1

Status Bits
- **N** MSB of the output.
- **Z** 1 if a zero output is generated, 0 otherwise.
- **V** 0
- **C** Unaffected.
- **UF** 0
- **LV** Unaffected.
- **LUF** Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
OR3   *++AR1(IR1),R2,R7
```

Before Instruction:

AR1 = 809800h
IR1 = 4h
R2 = 012560000h
R7 = 0h
Data at 809804h = 2BCDh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 809804h
IR1 = 4h
R2 = 012560000h
R7 = 012562BCDh
Data at 809804h = 2BCDh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

OR3 <*src2*>,<*src1*>,<*dst1*>
|| STI <*src3*>,<*dst2*>

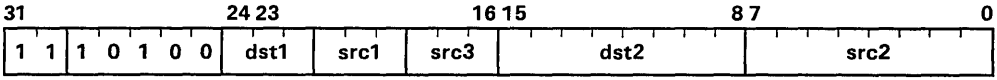
Operation

src1 OR *src2* → *dst1*
|| *src3* → *dst2*

Operands

src1 register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn2, 0 ≤ n2 ≤ 7)
src3 register (Rn3, 0 ≤ n3 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

A bitwise logical-OR and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (OR3) writes to the same register, then STI accepts as input the contents of the register before it is modified by the OR3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Cycles

1

Status Bits

N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit

OVM Operation not affected by OVM.

Example

```
    OR3   *++AR2,R5,R2
||  STI   R6,*AR1--
```

Before Instruction:

AR2 = 809830h
R5 = 800000h
R2 = 0h
R6 = 0DCh = 220
AR1 = 809883h
Data at 809831h = 9800h
Data at 809883h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

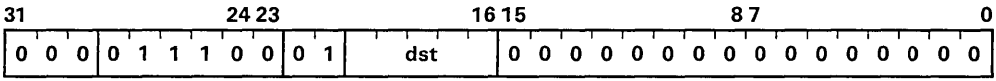
AR2 = 809831h
R5 = 800000h
R2 = 809800h
R6 = 0DCh = 220
AR1 = 809882h
Data at 809831h = 9800h
Data at 809883h = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax POP <*dst*>

Operation *SP-- → *dst*

Operands *dst* register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The top of the current system stack is popped and loaded into the *dst* register. The top of the stack is assumed to be a signed integer. The POP is performed with a post decrement of the stack pointer.

Cycles 1

Status Bits

N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
POP   R3
```

Before Instruction:

SP = 809856h
R3 = 012DAh = 4,826
Data at 809856h = 0FFFF0DA4h = -62,044
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

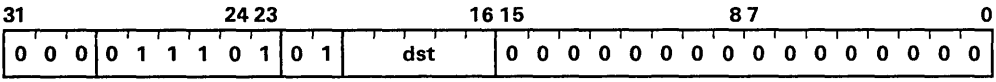
SP = 809855h
R3 = 0FFFF0DA4h = -62,044
Data at 809856h = 0FFFF0DA4h = -62,044
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Syntax POPF *<dst>*

Operation *SP-- → *dst1*

Operands *dst* register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The top of the current system stack is popped and loaded into the *dst* register. The top of the stack is assumed to be a floating-point number. The POP is performed with a post decrement of the stack pointer.

Cycles 1

Status Bits

N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
POPF  R4
```

Before Instruction:

SP = 80984Ah
R4 = 025D2E0123h = 6.91186578e+00
Data at 80984Ah = 5F2C1302h = 5.32544007e+28
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

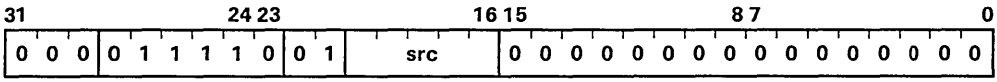
SP = 809849h
R4 = 5F2C130200h = 5.32544007e+28
Data at 80984Ah = 5F2C1302h = 5.32544007e+28
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax PUSH <*src*>

Operation *src* → *++SP

Operands *src* register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The contents of the *src* register are pushed on the current system stack . The *src* is assumed to be a signed integer. The PUSH is performed with a pre-increment of the stack pointer.

Cycles 1

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
PUSH  R6
```

Before Instruction:

SP = 8098AEh
R6 = 815Bh = 33,115
Data at 8098AFh = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

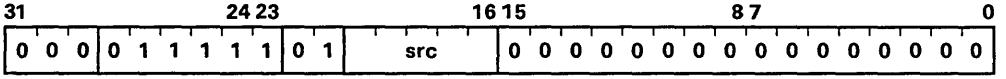
SP = 8098AFh
R6 = 815Bh = 33,115
Data at 8098AFh = 815Bh = 33,115
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax PUSHF <*src*>

Operation *src* → *++SP

Operands *src* register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The contents of the *src* register are pushed on the current system stack . The *src* is assumed to be a floating-point number. The PUSH is performed with a preincrement of the stack pointer.

Cycles 1

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
PUSHF   R2
```

Before Instruction:

SP = 809801h
R2 = 025C128081h = 6.87725854e+00
Data at 809802h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

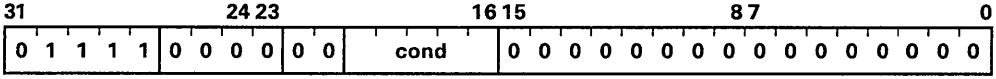
SP = 809802h
R2 = 025C128081h = 6.87725854e+00
Data at 809802h = 025C1280h = 6.87725830e+00
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax RETI*cond*

Operation If *cond* is true:
*SP-- → PC
1 → ST(GIE).
Else, continue.

Operands None

Encoding



Description A conditional return is performed. If the condition is true, the top of the stack is popped to the PC, and 1 is written to the global interrupt enable (GIE) bit of the status register. This has the effect of enabling all interrupts for which the corresponding interrupt enable bit is a 1.

The TMS320C30 provides 20 condition codes that can be used with this instruction (see Section 9.1 for a list of condition mnemonics, encoding, and flags).

Cycles 4

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example RETINZ

Before Instruction:

PC = 456h
SP = 809830h
ST = 0h
Data at 809830h = 123h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

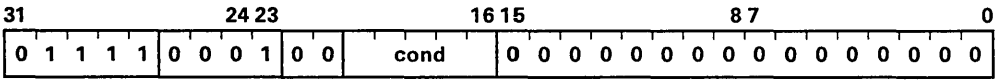
PC = 123h
SP = 80982Fh
ST = 2000h
Data at 809830h = 123h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax RETS*cond*

Operation If *cond* is true:
*SP-- → PC.
Else, continue.

Operands None

Encoding



Description A conditional return is performed. If the condition is true, the top of the stack is popped to the PC, and 1 is written to the global interrupt enable (GIE) bit of the status register. This has the effect of enabling all interrupts for which the corresponding interrupt enable bit is a 1.

The TMS320C30 provides 20 condition codes that can be used with this instruction (see Section 9.1 for a list of condition mnemonics, encoding, and flags).

Cycles 4

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example RETSGE

Before Instruction:

PC = 123h
SP = 80983Ch
Data at 80983Ch = 456h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

PC = 456h
SP = 80983Bh
Data at 80983Ch = 456h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

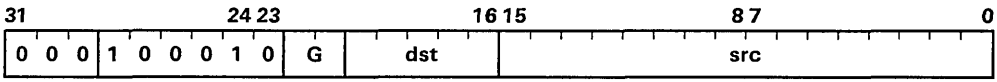
Syntax RND <*src*>,<*dst*>

Operation rnd(*src*) → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The result of rounding the *src* operand is loaded into the *dst* register. The *src* operand is rounded to the nearest single-precision floating-point value. If the *src* operand is exactly half-way between two single-precision values, it is rounded to the most positive of those values.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise,
Z 1 if a zero result is generated, 0 otherwise.
V 1 if a floating-point overflow occurs, 0 otherwise.
C Unaffected.
UF 1 if a floating-point underflow occurs, 0 otherwise.
LV 1 if a floating-point overflow occurs, unchanged otherwise.
LUF 1 if a floating-point underflow occurs, unchanged otherwise.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
RND  R5,R2
```

Before Instruction:

R5 = 0733C16EEFh = 1.79755599e+02
R2 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

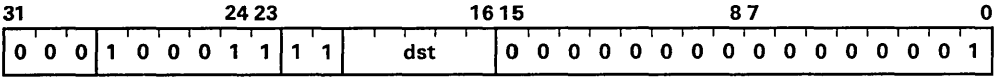
R5 = 0733C16EEFh = 1.79755599e+02
R2 = 0733C16F00h = 1.79755600e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax ROL <*dst*>

Operation *dst* left-rotated 1 bit → *dst*

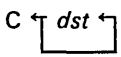
Operands *dst* register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The contents of the *dst* operand are left-rotated one bit and loaded into the *dst* register. This rotate is a circular rotate with the MSB transferred into the LSB.

Rotate left:



Cycles 1

Status Bits

N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Set to the value of the bit rotated out of the high-order bit. Unaffected if *dst* is not R0-R7.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
ROL  R3
```

Before Instruction:

R3 = 80025CD4h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R3 = 0004B9A9h
LUF LV UF N Z V C = 0 0 0 0 0 0 1

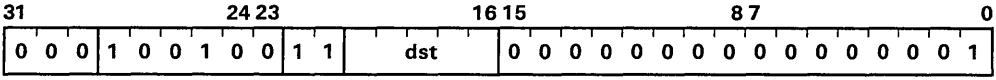
11

Syntax ROLC <*dst*>

Operation *dst* left-rotated 1 bit through carry bit → *dst*

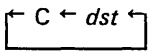
Operands *dst* register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The contents of the *dst* operand are left-rotated one bit through the carry bit and loaded into the *dst* register. The MSB is rotated to the carry bit, at the same time the carry bit is transferred to the LSB.

Rotate left through carry bit:



Cycles 1

Status Bits

N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Set to the value of the bit rotated out of the high-order bit. If *dst* is not R0-R7, then C is shifted into the *dst* but not changed.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
ROLC  R3
```

Before Instruction:

R3 = 00000420h
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

R3 = 000000841h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example

```
ROLC  R3
```

Before Instruction:

R3 = 80004281h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

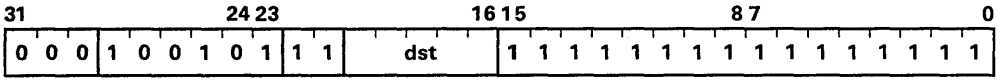
R3 = 00008502h
LUF LV UF N Z V C = 0 0 0 0 0 0 1

Syntax ROR <*dst*>

Operation *dst* right-rotated 1 bit through carry bit → *dst*

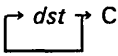
Operands *dst* register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The contents of the *dst* operand are right-rotated one bit and loaded into the *dst* register. The LSB is rotated into the carry bit and also transferred into the MSB.

Rotate right:



Cycles 1

Status Bits

N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Set to the value of the bit rotated out of the low-order bit. Unaffected if *dst* is not R0-R7.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
ROR  R7
```

Before Instruction:

R7 = 00000421h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R7 = 80000210h
LUF LV UF N Z V C = 0 0 0 1 0 0 1

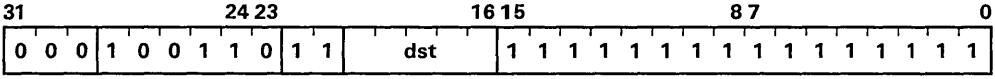
11

Syntax RORC <*dst*>

Operation *dst* right-rotated 1 bit through carry bit → *dst*

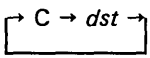
Operands *dst* register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The contents of the *dst* operand are right-rotated one bit through the carry bit and loaded into the *dst* register. The LSB is rotated into the carry bit. At the same time, the carry bit is transferred into the MSB.

Rotate right through carry bit:



Cycles 1

Status Bits

N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Set to the value of the bit rotated out of the low-order bit. If *dst* is not R0-R7, then C is shifted in but not changed.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
RORC  R4
```

Before Instruction:

R4 = 00000081h
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

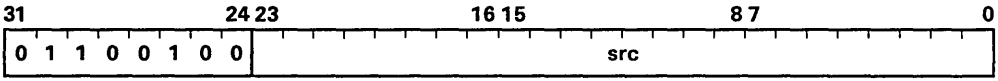
R4 = 80000040h
LUF LV UF N Z V C = 0 0 0 1 0 0 1

Syntax RPTB <*src*>

Operation *src* → RE
1 → ST(RM)
Next PC → RS

Operands *src* long-immediate addressing mode

Encoding



Description RPTB allows a block of instructions to be repeated a number of times without any penalty for looping. This instruction activates the block repeat mode of updating the PC. The *src* operand is a 24-bit unsigned immediate value that is loaded into the repeat end address (RE) register. A 1 is written into the repeat mode bit of status register ST(RM) to indicate that the PC is being updated in the repeat mode. The address of the next instruction is loaded into the repeat start address (RS) register.

Cycles 4

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
RPTB  127h
```

Before Instruction:

PC = 123h
ST = 0h
RE = 0h
RS = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

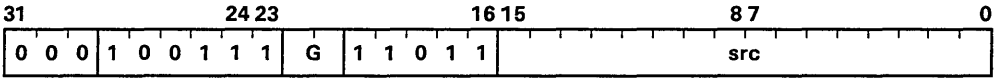
PC = 124h
ST = 100h
RE = 127h
RS = 124h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax RPTS <*src*>

Operation
src → RC
1 → ST(RM)
1 → S
Next PC → RS
Next PC → RE

Operands *src* general addressing modes (G):
0 0 register
0 1 direct
1 0 indirect
1 1 immediate

Encoding



Description The RPTS instruction allows a single instruction to be repeated a number of times without any penalty for looping. Fetches can also be made from the instruction register (IR), thus avoiding repeated memory access.

The *src* operand is loaded into the repeat counter (RC). A 1 is written into the repeat mode bit of the status register ST(RM). A 1 is also written into the repeat single bit (S). This indicates that the program fetches are to be performed only from the instruction register. the repeat single mode. The next PC is loaded into the repeat end address (RE) register and the repeat start address (RS) register.

The *src* operand is assumed to be an unsigned integer and is not sign-extended for immediate mode.

Cycles 4

Status Bits
N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
RPTS  AR5
```

Before Instruction:

PC = 123h
ST = 0h
RS = 0h
RE = 0h
RC = 0h
AR5 = 0FFh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

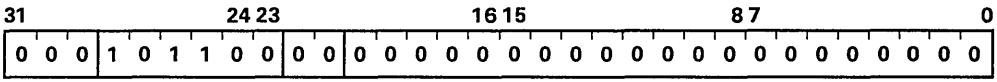
PC = 124h
ST = 100h
RS = 124h
RE = 124h
RC = 0FFh
AR5 = 0FFh
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax SIGI

Operation Signal interlocked operation.
Wait for interlock acknowledge.
Clear interlock.

Operands None

Encoding



Description An interlocked operation is signaled over XF0 and XF1. After the interlocked operation is acknowledged, the interlocked operation ends. SIGI ignores the external ready signals. Refer to Section 7.3 for detailed information.

Cycles 1

Status Bits

N	Unaffected.
Z	Unaffected.
V	Unaffected.
C	Unaffected.
UF	Unaffected.
LV	Unaffected.
LUF	Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

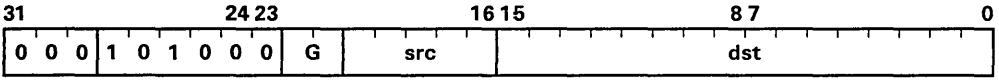
Syntax STF <*src*>,<*dst*>

Operation *src* → *dst*

Operands *src* register (Rn, 0 ≤ n ≤ 7)

dst general addressing modes (G):
0 1 direct
1 0 indirect

Encoding



Description The *src* register is loaded into the *dst* memory location. The *src* and *dst* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
STF  R2,@98A1h
```

Before Instruction:

DP = 80h
R2 = 052C501900h = 4.30782204e+01
Data at 8098A1h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R2 = 052C501900h = 4.30782204e+01
Data at 8098A1h = 52C5019h = 4.30782204e+01
LUF LV UF N Z V C = 0 0 0 0 0 0 0

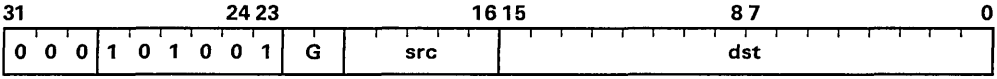
Syntax STFI *<src>,<dst>*

Operation *src* → *dst*
Signal end of interlocked operation.

Operands *src* register (Rn, 0 ≤ n ≤ 7)

dst general addressing modes (G):
0 1 direct
1 0 indirect

Encoding



Description The *src* register is loaded into the *dst* memory location. An interlocked operation is signaled over pins XF0 and XF1. The *src* and *dst* operands are assumed to be floating-point numbers. Refer to Section 7.3 for detailed information.

Cycles 1

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
STFI  R3,*-AR4
```

Before Instruction:

R3 = 0733C00000h = 1.79750e+02
AR4 = 80993Ch
Data at 80993Bh = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R3 = 0733C00000h = 1.79750e+02
AR4 = 80993Ch
Data at 80993Bh = 733C000h = 1.79750e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

STF <*src2*>,<*dst2*>
|| STF <*src1*>,<*dst1*>

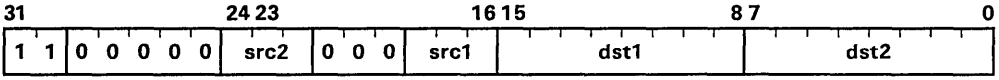
Operation

src2 → *dst2*
|| *src1* → *dst1*

Operands

src1 register (Rn1, 0 ≤ n1 ≤ 7)
dst1 indirect (disp = 0, 1, IR0, IR1)
src2 register (Rn2, 0 ≤ n2 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description Two floating-point stores are performed in parallel. If both stores are executed to the same address, the value written is that of STF <*src2*>,<*dst2*>.

Cycles 1

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit

OVM Operation not affected by OVM.

Example

```
    STF   R4,*AR3--
||  STF   R3,*++AR5
```

Before Instruction:

R4 = 070C800000h = 1.4050e+02
AR3 = 809835h
R3 = 0733C00000h = 1.79750e+02
AR5 = 8099D2h
Data at 809835h = 0h
Data at 8099D3h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R4 = 070C800000h = 1.4050e+02
AR3 = 809834h
R3 = 0733C00000h = 1.79750e+02
AR5 = 8099D3h
Data at 809835h = 070C8000h = 1.4050e+02
Data at 8099D3h = 0733C000h = 1.79750e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

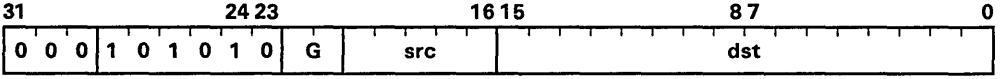
Syntax STI <*src*>,<*dst*>

Operation *src* → *dst*

Operands *src* register (Rn, 0 ≤ n ≤ 27)

dst general addressing modes (G):
0 1 direct
1 0 indirect

Encoding



Description The *src* register is loaded into the *dst* memory location. The *src* and *dst* operands are assumed to be signed integers.

Cycles 1

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
STI  R4,@982Bh
```

Before Instruction:

DP = 80h
R4 = 42BD7h = 273,367
Data at 80982Bh = 0E5FCh = 58,876
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R4 = 42BD7h = 273,367
Data at 80982Bh = 42BD7h = 273,367
LUF LV UF N Z V C = 0 0 0 0 0 0 0

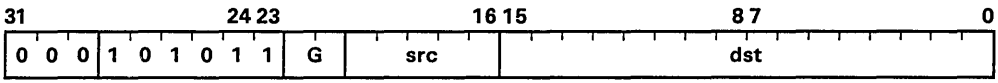
Syntax STII *<src>,<dst>*

Operation *src* → *dst*
Signal end of interlocked operation.

Operands *src* register (Rn, 0 ≤ n ≤ 27)

dst general addressing modes (G):
0 1 direct
1 0 indirect

Encoding



Description The *src* register is loaded into the *dst* memory location. An interlocked operation is signaled over pins XF0 and XF1. The *src* and *dst* operands are assumed to be signed integers. Refer to Section 7.3 for detailed information.

Cycles 1

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
STII  R1,@98AEh
```

Before Instruction:

DP = 80h
R1 = 78Dh
Data at 8098AEh = 25Ch

After Instruction:

DP = 80h
R1 = 78Dh
Data at 8098AEh = 7BDh

Syntax

STI <*src2*>,<*dst2*>
|| STI <*src1*>,<*dst1*>

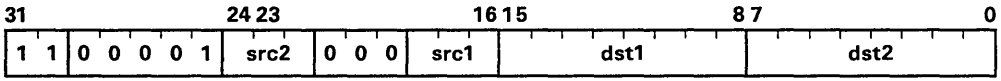
Operation

src2 → *dst2*
|| *src1* → *dst1*

Operands

src1 register (Rn1, 0 ≤ n1 ≤ 7)
dst1 indirect (disp = 0, 1, IR0, IR1)
src2 register (Rn2, 0 ≤ n2 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

Two integer stores are performed in parallel. If both stores are executed to the same address, the value written is that of STI <*src2*>,<*dst2*>.

Cycles

1

Status Bits

N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit

OVM Operation not affected by OVM.

Example

```
   STI  R0,*++AR2(IR0)
|| STI  R5,*AR0
```

Before Instruction:

R0 = 0DCh = 220
AR2 = 809830h
IR0 = 8h
R5 = 35h = 53
AR0 = 8098D3h
Data at 809838h = 0h
Data at 8098D3h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R0 = 0DCh = 220
AR2 = 809838h
IR0 = 8h
R5 = 35h = 53
AR0 = 8098D3h
Data at 809838h = 0DCh = 220
Data at 8098D3h = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0

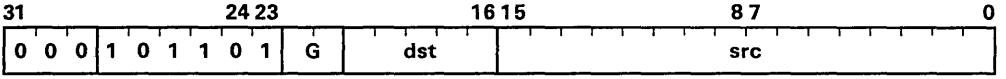
Syntax SUBB *<src>,<dst>*

Operation *dst* - *src* - C → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The difference of the *dst*, *src*, and C operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C 1 if a borrow occurs, 0 otherwise.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit **OVM** Operation affected by OVM.

Example

```
SUBB  *AR5++(4),R5
```

Before Instruction:

AR5 = 809800h
R5 = 0FAh = 250
Data at 809800h = 0C7h = 199
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

AR5 = 809804h
R5 = 032h = 50
Data at 809800h = 0C7h = 199
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax SUBB3 *<src2>,<src1>,<dst>*

Operation *src1* - *src2* - C → *dst*

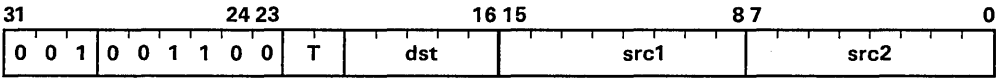
Operands

src1 three-operand addressing modes (T):
- 0 0 register (Rn1, 0 ≤ n1 ≤ 27)
- 0 1 indirect (disp = 0, 1, IR0, IR1)
- 1 0 register (Rn1, 0 ≤ n1 ≤ 27)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):
- 0 0 register (Rn2, 0 ≤ n2 ≤ 27)
- 0 1 register (Rn2, 0 ≤ n2 ≤ 27)
- 1 0 indirect (disp = 0, 1, IR0, IR1)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The difference of the *src1* and *src2* operands and the C (carry) flag is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be signed integers.

Cycles 1

Status Bits
- **N** 1 if a negative result is generated, 0 otherwise.
- **Z** 1 if a zero result is generated, 0 otherwise.
- **V** 1 if an integer overflow occurs, 0 otherwise.
- **C** 1 if a borrow occurs, 0 otherwise.
- **UF** 0
- **LV** 1 if an integer overflow occurs, unchanged otherwise.
- **LUF** Unaffected.

Mode Bit **OVM** Operation affected by OVM.

Example

```
SUBB3  R5,*AR5++(IR0),R0
```

Before Instruction:

AR5 = 809800h
IR0 = 4h
R5 = 0C7h = 199
R0 = 0h
Data at 809800h = 0FAh = 250
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

AR5 = 809804h
IR0 = 4h
R5 = 0C7h = 199
R0 = 32h = 50
Data at 809800h = 0FAh = 250
LUF LV UF N Z V C = 0 0 0 0 0 0 0

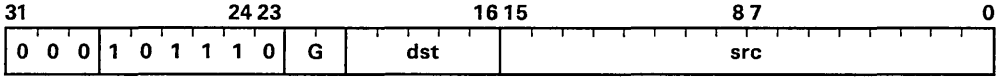
Syntax SUBC <*src*>,<*dst*>

Operation
If (*dst* - *src* ≥ 0):
  (*dst* - *src* << 1) OR 1 → *dst*
Else:
  dst << 1 → *dst*

Operands
src general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description A subtraction of the *src* operand from the *dst* operand is performed. The *dst* operand is loaded with a value dependent upon the result of the subtraction. If (*dst* - *src*) is greater than or equal to zero, then (*dst* - *src*) is left-shifted one bit, the least-significant bit is set to 1, and the result is loaded into the *dst* register. If (*dst* - *src*) is less than zero, *dst* is left-shifted one bit and loaded into the *dst* register. The *dst* and *src* operands are assumed to be unsigned integers.

SUBC may be used to perform a single step of a multi-bit integer division. See Section 12.3.3 for a detailed description.

Cycles 1

Status Bits
N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example SUBC @98C5h,R1

Before Instruction:

DP = 80h
R1 = 04F6h = 1270
Data at 8098C5h = 492h = 1170
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R1 = 0C9h = 201
Data at 8098C5h = 492h = 1170
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example SUBC 3000,R0 (3000 = 0BB8h)

Before Instruction:

R0 = 07D0h = 2000
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R0 = 0FA0h = 4000
LUF LV UF N Z V C = 0 0 0 0 0 0 0

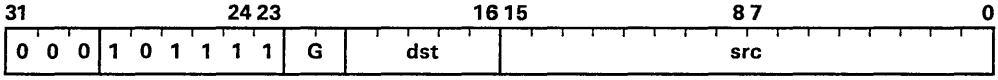
Syntax SUBF *<src>,<dst>*

Operation *dst - src → dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The *dst* operand minus the *src* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an floating-point overflow occurs, 0 otherwise.
C Unaffected
UF 1 if a floating-point underflow occurs, 0 otherwise.
LV 1 if an floating-point overflow occurs, unchanged otherwise.
LUF 1 if a floating-point underflow occurs, unchanged otherwise.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
SUBF  *AR0--(IR0),R5
```

Before Instruction:

AR0 = 809888h
IR0 = 80h
R5 = 0733C00000h = 1.79750000e+02
Data at 809888h = 70C8000h = 1.4050e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR0 = 809808h
IR0 = 80h
R5 = 051D000000h = 3.9250e+01
Data at 809888h = 70C8000h = 1.4050e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax SUBF3 <*src2*>,<*src1*>,<*dst*>

Operation *src1* - *src2* → *dst*

Operands *src1* three-operand addressing modes (T):

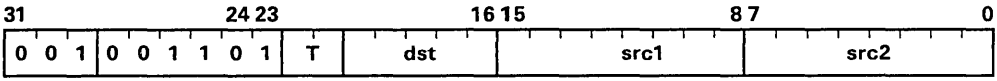
0 0 register (Rn1, ≤ n1 ≤ 7)
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 register (Rn1, ≤ n1 ≤ 7)
1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):

0 0 register (Rn2, ≤ n2 ≤ 7)
0 1 register (Rn2, ≤ n2 ≤ 7)
1 0 indirect (disp = 0, 1, IR0, IR1)
1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The difference of the *src1* and *src2* operands is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits

N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if a floating-point overflow occurs, 0 otherwise.
C Unaffected.
UF 1 if a floating-point underflow occurs, 0 otherwise.
LV 1 if a floating-point overflow occurs, unchanged otherwise.
LUF 1 if a floating-point underflow occurs, unchanged otherwise.

Mode Bit **OVM** Operation not affected by OVM.

Example `SUBF3 *AR0--(IR0),*AR1,R4`

Before Instruction:

AR0 = 809888h
IR0 = 80h
AR1 = 809851h
R4 = 0h
Data at 809888h = 70C8000h = 1.4050e+02
Data at 809851h = 733C000h = 1.79750e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR0 = 809808h
IR0 = 80h
AR1 = 809851h
R4 = 51D000000h = 3.9250e+01
Data at 809888h = 70C8000h = 1.4050e+02
Data at 809851h = 733C000h = 1.79750e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example `SUBF3 R7,R0,R6`

Before Instruction:

R7 = 57B400000h = 6.281250e+01
R0 = 34C200000h = 1.27578125e+01
R6 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R7 = 57B400000h = 6.281250e+01
R0 = 34C200000h = 1.27578125e+01
R6 = 5B7C80000h = -5.00546875e+01
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Syntax

SUBF3 <*src1*>,<*src2*>,<*dst1*>
|| STF <*src3*>,<*dst2*>

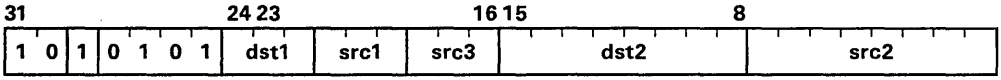
Operation

src2 - *src1* → *dst1*
|| *src3* → *dst2*

Operands

src1 register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn2, 0 ≤ n2 ≤ 7)
src3 register (Rn3, 0 ≤ n3 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

A floating-point subtraction and a floating-point store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STF) reads from a register and the operation being performed in parallel (SUBF3) writes to the same register, then STF accepts as input the contents of the register before it is modified by the SUBF3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Cycles

1

Status Bits

N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if a floating-point overflow occurs, 0 otherwise.
C Unaffected.
UF 1 if a floating-point underflow occurs, 0 otherwise.
LV 1 if a floating-point overflow occurs, unchanged otherwise.
LUF 1 if a floating-point underflow occurs, unchanged otherwise.

Mode Bit

OVM Operation not affected by OVM.

Example

```
   SUBF3  R1,*-AR4(IR1),R0
|| STF   R7,*+AR5(IR0)
```

Before Instruction:

R1 = 057B400000h = 6.28125e+01
AR4 = 8098B8h
IR1 = 8h
R0 = 0h
R7 = 0733C00000h = 1.79750e+02
AR5 = 809850h
IR0 = 10h
Data at 8098B0h = 70C8000h = 1.4050e+02
Data at 809860h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 057B400000h = 6.28125e+01
AR4 = 8098B8h
IR1 = 8h
R0 = 061B600000h = 7.768750e+01
R7 = 0733C00000h = 1.79750e+02
AR5 = 809850h
IR0 = 10h
Data at 8098B0h = 70C8000h = 1.4050e+02
Data at 809860h = 733C000h = 1.79750e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

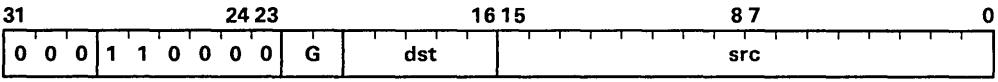
Syntax SUBI <*src*>,<*dst*>

Operation *dst* - *src* → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The *dst* operand minus the *src* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C 1 if a borrow occurs, 0 otherwise.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit **OVM** Operation affected by OVM.

Example

```
SUBI  220,R7
```

Before Instruction:

R7 = 226h = 550
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R7 = 14Ah = 330
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax SUBI3 *<src2>,<src1>,<dst>*

Operation *src1* - *src2* → *dst*

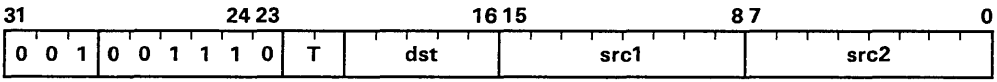
Operands

src1 three-operand addressing modes (T):
- 0 0 register (Rn1, 0 ≤ n1 ≤ 27)
- 0 1 indirect (disp = 0, 1, IR0, IR1)
- 1 0 register (Rn1, 0 ≤ n1 ≤ 27)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):
- 0 0 register (Rn2, 0 ≤ n2 ≤ 27)
- 0 1 register (Rn2, 0 ≤ n2 ≤ 27)
- 1 0 indirect (disp = 0, 1, IR0, IR1)
- 1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The *src1* operand minus the *src2* operand is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be signed integers.

Cycles 1

Status Bits
- **N** 1 if a negative result is generated, 0 otherwise.
- **Z** 1 if a zero result is generated, 0 otherwise.
- **V** 1 if an integer overflow occurs, 0 otherwise.
- **C** 1 if a borrow occurs, 0 otherwise.
- **UF** 0
- **LV** 1 if an integer overflow occurs, unchanged otherwise.
- **LUF** Unaffected.

Mode Bit **OVM** Operation affected by OVM.

Example

```
SUBI3  R7,R2,R0
```

Before Instruction:

R2 = 0866h = 2150
R7 = 0834h = 2100
R0 = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R2 = 0866h = 2150
R7 = 0834h = 2100
R0 = 032h = 50
LUF LV UF N Z V C = 0 0 0 1 0 0 0

Example SUBI3 *-AR2(1),R4,R3

Before Instruction:

AR2 = 80985Eh
R4 = 0226h = 550
R3 = 0h
Data at 80985Dh = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR2 = 80985Eh
R4 = 0226h = 550
R3 = 014Ah = 330
Data at 80985Dh = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

SUBI3 <*src1*>,<*src2*>,<*dst1*>
|| STI <*src3*>,<*dst2*>

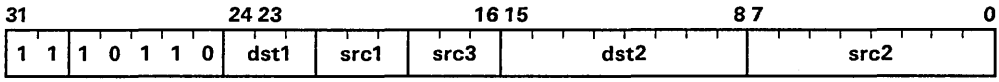
Operation

src2 - *src1* → *dst1*
|| *src3* → *dst2*

Operands

src1 register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn2, 0 ≤ n2 ≤ 7)
src3 register (Rn3, 0 ≤ n3 ≤ 7)
dst2 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description

An integer subtraction and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (SUBI3) writes to the same register, then STI accepts as input the contents of the register before it is modified by the SUBI3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Cycles

1

Status Bits

N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C 1 if a borrow occurs, 0 otherwise.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit

OVM Operation affected by OVM.

Example

```
   SUBI3  R7,*+AR2(IR0),R1
|| STI   R3,*++AR7
```

Before Instruction:

R7 = 14h = 20
AR2 = 80982Fh
IR0 = 10h
R1 = 0h
R3 = 35h = 53
AR7 = 80983Bh
Data at 80983Fh = 0DCh = 220
Data at 80983Ch = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R7 = 14h = 20
AR2 = 80982Fh
IR0 = 10h
R1 = 0C8h = 200
R3 = 35h = 53
AR7 = 80983Ch
Data at 80983Fh = 0DCh = 220
Data at 80983Ch = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0

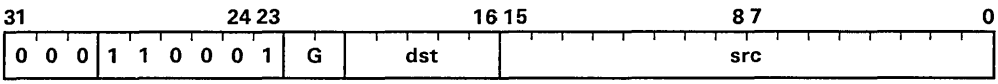
Syntax SUBRB <*src*>,<*dst*>

Operation *src* - *dst* - C → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The difference of the *src*, *dst*, and C operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C 1 if a borrow occurs, 0 otherwise.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit **OVM** Operation affected by OVM.

Example

```
SUBRB  R4,R6
```

Before Instruction:

R4 = 03CBh = 971
R6 = 0258h = 600
LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

R4 = 03CBh = 971
R6 = 0172h = 370
LUF LV UF N Z V C = 0 0 0 0 0 0 0

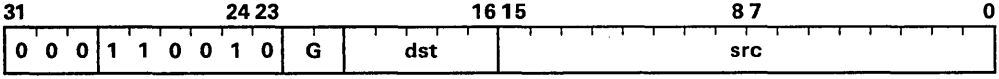
Syntax SUBRF *<src>,<dst>*

Operation *src - dst → dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 7)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 7)

Encoding



Description The *src* operand minus the *dst* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be floating-point numbers.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if a floating-point overflow occurs, 0 otherwise.
C Unaffected.
UF 1 if a floating-point underflow occurs, 0 otherwise.
LV 1 if a floating-point overflow occurs, unchanged otherwise.
LUF 1 if a floating-point underflow occurs, unchanged otherwise.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
SUBRF  @9905h,R5
```

Before Instruction:

DP = 80h
R5 = 057B400000h = 6.281250e+01
Data at 809905h = 733C000h = 1.79750e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

DP = 80h
R5 = 0669E00000h = 1.16937500e+02
Data at 809905h = 733C000h = 1.79750e+02
LUF LV UF N Z V C = 0 0 0 0 0 0 0

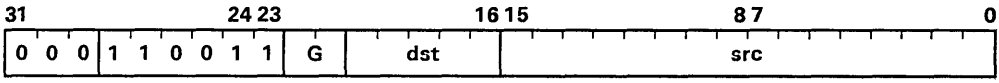
Syntax SUBRI *<src>,<dst>*

Operation *src* - *dst* → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The *src* operand minus the *dst* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

Cycles 1

Status Bits
N 1 if a negative result is generated, 0 otherwise.
Z 1 if a zero result is generated, 0 otherwise.
V 1 if an integer overflow occurs, 0 otherwise.
C 1 if a borrow occurs, 0 otherwise.
UF 0
LV 1 if an integer overflow occurs, unchanged otherwise.
LUF Unaffected.

Mode Bit **OVM** Operation affected by OVM.

Example

```
SUBRI  *AR5++(IR0),R3
```

Before Instruction:

AR5 = 809900h
IR0 = 8h
R3 = 0DCh = 220
Data at 809900h = 226h = 550
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

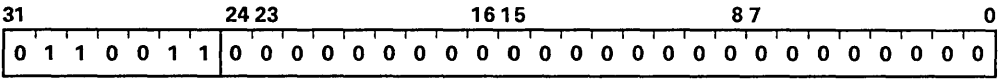
AR5 = 809908h
IR0 = 8h
R3 = 014Ah = 330
Data at 809900h = 226h = 550
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax SWI

Operation Performs an emulation interrupt.

Operands None

Encoding



Description The SWI instruction performs an emulator interrupt. This is a reserved instruction and should not be used in normal programming.

Cycles 4

Status Bits

N	Unaffected.
Z	Unaffected.
V	Unaffected.
C	Unaffected.
UF	Unaffected.
LV	Unaffected.
LUF	Unaffected.

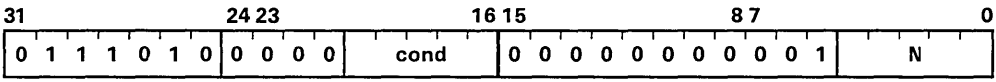
Mode Bit **OVM** Operation not affected by OVM.

Syntax TRAP*cond* N

Operation
0 → ST(GIE)
If *cond* is true:
  Next PC → *++SP
  Trap vector N → PC
Else:
  Set ST(GIE) to original state
  Continue.

Operands N (0 ≤ N ≤ 31)

Encoding



Description Interrupts are disabled globally when 0 is written to ST(GIE). If the condition is true, the contents of the PC are pushed on the system stack and the PC is loaded with the contents of the specified trap vector (N). If the condition is not true, ST(GIE) is set to its value before the TRAP*cond* instruction changed it.

The TMS320C30 provides 20 condition codes that can be used with this instruction (see Section 9.1 for a list of condition mnemonics, encoding, and flags).

Cycles 5

Status Bits
N Unaffected.
Z Unaffected.
V Unaffected.
C Unaffected.
UF Unaffected.
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example TRAPZ 16

Before Instruction:

PC = 123h
SP = 809870h
ST = 0h
Trap Vector 16 = 10h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

PC = 10h
SP = 809871h
Data at 809871h = 124h
ST = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

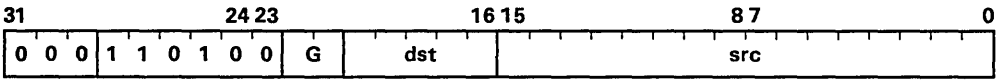
Syntax TSTB *<src>,<dst>*

Operation *dst* AND *src*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The bitwise logical-AND of the *dst* and *src* operands is formed, but the result is not loaded in any register. This allows for nondestructive compares. The *dst* and *src* operands are assumed to be unsigned integers.

Cycles 1

Status Bits
N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
TSTB  *-AR4(1),R5
```

Before Instruction:

AR4 = 8099C5h
R5 = 898h = 2200
Data at 8099C4h = 767h = 1895
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR4 = 8099C5h
R5 = 898h = 2200
Data at 8099C4h = 767h = 1895
LUF LV UF N Z V C = 0 0 0 0 1 0 0

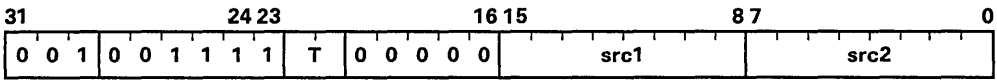
Syntax TSTB3 <*src2*>,<*src1*>

Operation *src1* AND *src2*

Operands *src1* three-operand addressing modes (T):
0 0 register (Rn1, 0 ≤ n1 ≤ 27)
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 register (Rn1, 0 ≤ n1 ≤ 27)
1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):
0 0 register (Rn2, 0 ≤ n2 ≤ 27)
0 1 register (Rn2, 0 ≤ n2 ≤ 27)
1 0 indirect (disp = 0, 1, IR0, IR1)
1 1 indirect (disp = 0, 1, IR0, IR1)

Encoding



Description The bitwise logical-AND between the *src1* and *src2* operands is formed, but is not loaded into any register. This allows for nondestructive compares. The *src1* and *src2* operands are assumed to be unsigned integers. Although this instruction has only two operands, it is designated as a three operand instruction since operands are specified in the three operand format.

Cycles 1

Status Bits
N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example `TSTB3 *AR5--(IR0),*+AR0(1)`

Before Instruction:

AR5 = 809885h
IR0 = 80h
AR0 = 80992Ch
Data at 809885h = 898h = 2200
Data at 80992Dh = 767h = 1895
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR5 = 809805h
IR0 = 80h
AR0 = 80992Ch
Data at 809885h = 898h = 2200
Data at 80992Dh = 767h = 1895
LUF LV UF N Z V C = 0 0 0 0 1 0 0

Example `TSTB3 R4,*AR6--(IR0)`

Before Instruction:

R4 = 0FBC4h
AR6 = 8099F8h
IR0 = 8h
Data at 8099F8h = 1568h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R4 = 0FBC4h
AR6 = 8099F0h
IR0 = 8h
Data at 8099F8h = 1568h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

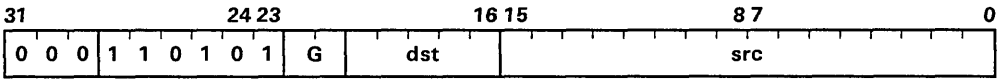
Syntax XOR <*src*>,<*dst*>

Operation *dst* XOR *src* → *dst*

Operands *src* general addressing modes (G):
0 0 register (Rn, 0 ≤ n ≤ 27)
0 1 direct
1 0 indirect
1 1 immediate

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The bitwise exclusive-OR of the *src* and *dst* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be unsigned integers.

Cycles 1

Status Bits
N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example

```
XOR  R1,R2
```

Before Instruction:

R1 = 0FFA32h
R2 = 0FF5C1h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R1 = 0FF3A2h
R2 = 000FF3h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax XOR3 <*src2*>,<*src1*>,<*dst*>

Operation *src1* XOR *src2* → *dst*

Operands *src1* three-operand addressing modes (T):

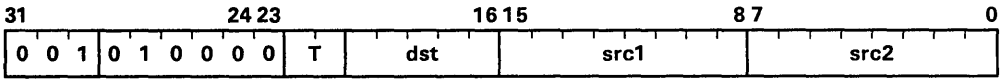
0 0 register (Rn1, 0 ≤ n1 ≤ 27)
0 1 indirect (disp = 0, 1, IR0, IR1)
1 0 register (Rn1, 0 ≤ n1 ≤ 27)
1 1 indirect (disp = 0, 1, IR0, IR1)

src2 three-operand addressing modes (T):

0 0 register (Rn2, 0 ≤ n2 ≤ 27)
0 1 register (Rn2, 0 ≤ n2 ≤ 27)
1 0 indirect (disp = 0, 1, IR0, IR1)
1 1 indirect (disp = 0, 1, IR0, IR1)

dst register (Rn, 0 ≤ n ≤ 27)

Encoding



Description The bitwise exclusive-OR between the *src1* and *src2* operands is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be unsigned integers.

Cycles 1

Status Bits

N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit **OVM** Operation not affected by OVM.

Example `XOR3 *AR3++(IR0),R7,R4`

Before Instruction:

AR3 = 809800h
IR0 = 10h
R7 = 0FFFFh
R4 = 0h
Data at 809800h = 5AC3h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR3 = 80980Fh
IR0 = 10h
R7 = 0FFFFh
R4 = 0A53Ch
Data at 809800h = 5AC3h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Example `XOR3 R5,*-AR1(1),R1`

Before Instruction:

R5 = 0FFA32h
AR1 = 809826h
R1 = 0h
Data at 809825h = 0FF5C1h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R5 = 0FFA32h
AR1 = 809826h
R1 = 000F33h
Data at 809825h = 0FF5C1h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Syntax

XOR3 <*src2*>,<*src1*>,<*dst1*>
|| STI <*src3*>,<*dst2*>

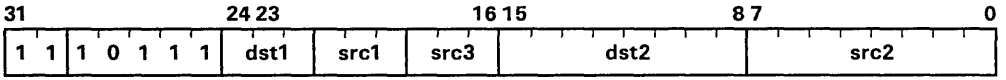
Operation

src1 XOR *src2* → *dst1*
|| *src3* → *dst2*

Operands

src1 register (Rn1, 0 ≤ n1 ≤ 7)
src2 indirect (disp = 0, 1, IR0, IR1)
dst1 register (Rn2, ≤ n2 ≤ 7)
src3 register (Rn3, ≤ n3 ≤ 7)
dst2 indirect0(disp = 0, 1, IR0, IR1)

Encoding



Description

A bitwise exclusive-XOR and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (XOR3) writes to the same register, then STI accepts as input the contents of the register before it is modified by the XOR3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Cycles

1

Status Bits

N MSB of the output.
Z 1 if a zero output is generated, 0 otherwise.
V 0
C Unaffected.
UF 0
LV Unaffected.
LUF Unaffected.

Mode Bit

OVM Operation not affected by OVM.

Example

```
   XOR3  *AR1++,R3,R3
|| STI   R6,*-AR2(IR0)
```

Before Instruction:

AR1 = 80987Eh
R3 = 85h
R6 = 0DCh = 220
AR2 = 8098B4h
IR0 = 8h
Data at 80987Eh = 85h
Data at 8098ACh = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR1 = 80987Fh
R3 = 0h
R6 = 0DCh = 220
AR2 = 8098B4h
IR0 = 8h
Data at 80987Eh = 85h
Data at 8098ACh = 0DCh = 220
LUF LV UF N Z V C = 0 0 0 0 0 0 0

Introduction 1
Pinout and Signal Descriptions 2
Architectural Overview 3
CPU Registers, Memory, and Cache 4
Data Formats and Floating-Point Operation 5
Addressing 6
Program Flow Control 7
External Bus Operation 8
Peripherals 9
Pipeline Operation 10
Assembly Language Instructions 11
Software Applications 12
Hardware Applications 13
Appendices A-D

Section 12

Software Applications

The TMS320C30 is a very powerful digital signal processor with architecture and instruction set designed to make easy system solutions to DSP problems. There are instructions specifically designed for efficient implementations of DSP algorithms, but also there are general-purpose instructions that make the device suitable for more general tasks, like any microprocessor. The floating point and integer arithmetic supported by the device permits the designer to concentrate on the algorithm with minimal concerns about scaling, dynamic range, and overflows.

The purpose of this section is to explain how to use the instruction set, the architecture, and the interface of the TMS320C30 processor. This is done by presenting coding examples for very frequently used applications, and by discussing more involved examples and applications. In all cases, besides explaining the principles involved in the application, the corresponding assembly-language code is given for instructional purposes and for immediate use. Whenever the detailed explanation of the underlying theory is too extensive to be included in this manual, appropriate references are given for further information.

Major topics discussed in this section are listed below.

- Processor Initialization (Section 12.1 on page 12-3)
- Program Control (Section 12.2 on page 12-7)
  - Subroutine calls
  - Software stack
  - Interrupt handling
  - Delayed branches
  - Repeat modes
  - Computed GOTO's
- Logical and Arithmetic Operations (Section 12.3 on page 12-20)
  - Bit manipulation
  - Block moves
  - Bit-reversed addressing
  - Division
  - Square root
  - Extended-precision arithmetic
  - IEEE <==> C30 floating point conversions
- Application-oriented Operations (Section 12.4 on page 12-45)
  - Companding (A-law, μ-law)
  - FIR / IIR filters (fixed and adaptive)
  - Matrix math
  - FFT

  - Lattice filters

- Programming Tips (Section 12.5 on page 12-87)
  - C-callable Routines
  - Code Optimization Check-list

12.1 Processor Initialization

Prior to the execution of a digital signal processing algorithm, it is necessary to initialize the processor. Generally, initialization takes place any time the processor is reset.

When reset is activated by applying a low level to the $\overline{\text{RESET}}$ input for several cycles, the TMS320C30 terminates execution and puts the reset vector (i.e., the contents of memory location 0) in the program counter. The reset vector normally contains the address of the system initialization routine. The hardware reset also initializes various registers and status bits.

After reset, the processor should be initialized to meet the requirements of the system. Instructions should be executed that set up operational modes, memory pointers, interrupts, and the remaining functions needed to meet system requirements.

To configure the processor at reset, the following internal functions should be initialized:

- Memory-mapped registers
- Interrupt structure

Example 12-1 shows coding for initializing the TMS320C30 to the following machine state, in addition to the initialization performed during the hardware reset (for conditions after hardware reset, see section 13):

- All interrupts are enabled.
- The overflow mode is disabled.
- The data memory page pointer is set to zero.
- The internal memory is filled with zeros.

Note that all constants larger than 16 bits should be placed in memory and accessed through direct or indirect addressing.

Example 12-1. TMS320C30 Processor Initialization

```
*
*  TITL  'PROCESSOR INITIALIZATION EXAMPLE'
*
          .global RESET,INIT,BEGIN
          .global INT0,INT1,INT2,INT3
          .global ISR0,ISR1,ISR2,ISR3
          .global DINT,DMA
          .global TINT0,TINT1,XINT0,RINT0,XINT1,RINT1
          .global TIME0,TIME1,XMT0,RCV0,XMT1,RCV1
          .global TRAP0,TRAP1,TRAP2,TRP0,TRP1,TRP2
*
*  PROCESSOR INITIALIZATION FOR THE TMS320C30.
*
*  RESET AND INTERRUPT VECTOR SPECIFICATION.  THIS
*  ARRANGEMENT ASSUMES THAT DURING LINKING, THE FOLLOWING
*  TEXT SEGMENT WILL BE PLACED TO START AT MEMORY
*  LOCATION 0.
*
          .sect   "init"      ; Named section
RESET     .word   INIT        ; RS- loads address INIT to PC
*
INT0      .word   ISR0        ; INT0- loads address INT0 to PC
INT1      .word   ISR1        ; INT1- loads address INT1 to PC
INT2      .word   ISR2        ; INT2- loads address INT2 to PC
INT3      .word   ISR3        ; INT3- loads address INT3 to PC
*
XINT0     .word   XMT0        ; Serial port 0 transmit processing
RINT0     .word   RCV0        ; Serial port 0 receive processing
XINT0     .word   XMT1        ; Serial port 1 transmit processing
RINT0     .word   RCV1        ; Serial port 1 receive processing
TINT0     .word   TIME0       ; Timer 0 interrupt processing
TINT1     .word   TIME1       ; Timer 1 interrupt processing
DINT      .word   DMA         ; DMA interrupt
          .space  20          ; Reserved space
TRAP0     .word   TRP0        ; Trap 0 vector processing begins
TRAP1     .word   TRP1        ; Trap 1 vector processing begins
TRAP2     .word   TRP         ; Trap 2 vector processing begins
          .space  29          ; Leave space for the other 29 traps
*
*  IN THIS SECTION, CONSTANTS THAT CANNOT BE REPRESENTED
*  IN THE SHORT FORMAT ARE INITIALIZED.
          .data
MASK      .word   0FFFFFFFFH
BLK0      .word   0809800H    ; Beginning address of RAM block 0
BLK0      .word   0809C00H    ; Beginning address of RAM block 1
STCK      .word   0809F00H    ; Beginning of stack
CTRL      .word   0808000H    ; Pointer for peripheral-bus memory map
DMACTL    .word   0000000H    ; Initialization for DMA control (0)
TIM0CTL   .word   0000000H    ; Initialization of timer 0 control (32)
TIM1CTL   .word   0000000H    ; Initialization of timer 1 control (48)
SERGLOB0  .word   0000000H    ; Init of serial 0 glbl control (64)
SERPRTX0  .word   0000000H    ; Init of serial 0 xmt port control (66)
SERPRTR0  .word   0000000H    ; Init of serial 0 rcv port control (67)
SERTIM0   .word   0000000H    ; Init of serial 0 timer control (68)
SERGLOB1  .word   0000000H    ; Init of serial 1 glbl control (80)
SERPRTX1  .word   0000000H    ; Init of serial 1 xmt port control (82)
SERPRTR1  .word   0000000H    ; Init of serial 1 rcv port control (83)
SERTIM1   .word   0000000H    ; Init of serial 1 timer control (84)
PARINT    .word   0000000H    ; Init parallel interface control (96)
IOINT     .word   0000000H    ; Init I/O interface control (100)
```

```
        .text
*
*   THE ADDRESS AT MEMORY LOCATION 0 DIRECTS EXECUTION TO BEGIN HERE
*   FOR RESET PROCESSING THAT INITIALIZES THE PROCESSOR.  WHEN RESET
*   IS APPLIED, THE FOLLOWING REGISTERS ARE INITIALIZED TO ZERO:
*
*   ST  - - CPU STATUS REGISTER
*   IE  - - CPU/DMA INTERRUPT ENABLE FLAGS
*   IF  - - CPU INTERRUPT FLAGS
*   IOF - - I/O FLAGS
*
*   THE STATUS REGISTER HAS THE FOLLOWING ARRANGEMENT:
*   BITS:      31-14   13  12  11 10  9   8  7   6   5  4 3 2 1 0
*   FUNCTION:  RESRV  GIE  CC  CE CF RES RM OVM LUF LV UF N Z V C
*
INIT    LDP   0,DP              ; Point the DP register to page 0
        LDI   1800H,ST          ; Clear and enable cache, and disable OVM
        LDI   @MASK,IE          ; Unmask all interrupts
*
  INTERNAL DATA MEMORY INITIALIZATION TO FLOATING POINT ZERO
*
        LDI   @BLK0,AR0         ; AR0 points to block 0
        LDI   @BLK1,AR1         ; AR1 points to block 1
        LDF   0.0,R0            ; Zero register R0
        RPTS  1023              ; Repeat 1024 times ...
        STF   R0,*AR0++(1)      ; Zero out location in RAM block 0 and ...
||      STF   R0,*AR1++(1)      ; zero out location in RAM block 1.
*
*   THE PROCESSOR IS INITIALIZED.  THE REMAINING APPLICATION-
*   DEPENDENT PART OF THE SYSTEM (BOTH ON- AND OFF-CHIP SHOULD
*   NOW BE INITIALIZED.
*
*   FIRST, INITIALIZE THE CONTROL REGISTERS.  IN THIS EXAMPLE,
*   EVERYTHING IS INITIALIZED TO ZERO SINCE THE ACTUAL INITIAL-
*   IZATION IS APPLICATION DEPENDENT.
*
        LDI   @CTRL,AR0         ; LOAD in AR0 the pointer to control
*                               ;   registers
        LDI   @DMACTL,R0
        STI   R0,*+AR0(0)       ; Init DMA control
        LDI   @TIM0CTL,R0
        STI   R0,*+AR0(32)      ; Init timer 0 control
        LDI   @TIM1CTL,R0
        STI   R0,*+AR0(48)      ; Init timer 1 control
        LDI   @SERGLOB0,R0
        STI   R0,*+AR0(64)      ; Init serial 0 global control
        LDI   @SERPRTX0,R0
        STI   R0,*+AR0(66)      ; Init serial 0 xmt control
        LDI   @SERPRTR0,R0
        STI   R0,*+AR0(67)      ; Init serial 0 rcv control
        LDI   @SERTIM0,R0
        STI   R0,*+AR0(68)      ; Init serial 0 timer control
        LDI   @SERGLOBL,R0
        STI   R0,*+AR0(80)      ; Init serial 1 global control
        LDI   @SERPRTX1,R0
        STI   R0,*+AR0(82)      ; Init serial 1 xmt control
        LDI   @SERPRTR1,R0
        STI   R0,*+AR0(83)      ; Init serial 1 rcv control
```

```
        LDI     @SERTIM1,R0
        STI     R0,*+AR0(84)  ;  Init serial timer control
        LDI     @PARINT,R0
        STI     R0,*+AR0(96)  ;  Init parallel interface control
        LDI     @IOINT,R0
        STI     R0,*+AR0(100) ;  Init I/O interface control
*
        LDI     @STCK,SP      ;  Initialize the stack pointer
        OR      2000H,ST      ;  Global interrupt enable
*
        BR      BEGIN         ;  Branch to the beginning of application.

        .end
```

12.2 Program Control

To facilitate the TMS320C30's use in general-purpose, high-speed processing, a variety of instructions are provided to handle the:

- subroutine calls
- software stack
- interrupts
- zero-overhead branches
- single- and multiple-instruction loops without any overhead.

This section describes how to use these features of the TMS320C30.

12.2.1 Subroutines

The TMS320C30 has a 24-bit program counter (PC) and a practically unlimited software stack. The CALL and CALLcond subroutine calls cause the stack pointer to increment, and store the contents of the next value of the PC counter on the stack. At the end of the subroutine, RETScond performs a conditional return.

Example 12-2 illustrates the use of a subroutine to determine the dot product between two vectors. Given two vectors of length N, represented by the arrays a[0], a[1],..., a[N-1] and b[0], b[1],..., b[N-1], the dot product is computed from the expression

$$d = a[0]\ b[0] + a[1]\ b[1] + \ldots + a[N-1]\ b[N-1]$$

Processing proceeds in the main routine to the point where the dot product is to be computed. It is assumed that the arguments of the subroutine have been appropriately initialized. At this point, a CALL is made to the subroutine, transferring control to that section of the program memory for execution, then returning to the calling routine via the RETS instruction when execution has completed. Note for this particular example, it would suffice to save the register R2. However, a larger number of registers are saved for demonstration purposes. The saved registers are stored on the system stack. This stack should be large enough to accommodate the maximum anticipated storage requirements. Besides this way of saving registers, any other method could be used equally well.

Example 12-2. Subroutine Call (Dot Product)

```
*
*  TITL  SUBROUTINE CALL (DOT PRODUCT)
*
*
*  MAIN ROUTINE THAT CALLS THE SUBROUTINE 'DOT' TO COMPUTE THE
*  DOT PRODUCT OF TWO VECTORS.
        .
        .
        .
        LDI     @blk0,AR0       ; AR0 points to vector a
        LDI     @blk1,AR1       ; AR1 points to vector b
        LDI     N,RC            ; RC contains the number of elements

        CALL    DOT
        .
        .
        .

*
*  SUBROUTINE D O T
*
*
*  EQUATION: d = a(0) * b(0) + a(1) * b(1) + ... + a(N-1) * b(N-1)
*
*  THE DOT PRODUCT OF a AND b IS PLACED IN REGISTER R0.  N MUST
*  BE GREATER THAN OR EQUAL TO 2.
*
*  ARGUMENT ASSIGNMENTS:
*    ARGUMENT | FUNCTION
*    ---------+----------------------
*    AR0      | ADDRESS OF a(0)
*    AR1      | ADDRESS OF b(0)
*    RC       | LENGTH OF VECTORS (N)
*
*  REGISTERS USED AS INPUT: AR0, AR1, RC
*  REGISTER MODIFIED: R0
*  REGISTER CONTAINING RESULT: R0
*
*
*
        .global DOT
*
DOT     PUSHF   R2              ; Use the stack to save R2'S
        PUSH    R2              ;   bottom 32 and top 32 bits
        PUSH    ST              ; Save status register
        PUSH    AR0             ; Save AR0
        PUSH    AR1             ; Save AR1
        PUSH    RC              ; Save RC
*                               ; Initialize R0:
        MPYF3   *AR0,*AR1,R0    ; a(0) * b(0) -> R0
        LDF     0.0,R2          ; Initialize R2.
        SUBI    2,RC            ; Set RC = N-2
```

```
*
*  DOT PRODUCT (1 <= i < N)
*
        RPTS    RC              ; Setup the repeat single.
        MPYF3   *++AR0(1),*++AR1(1),R0 ;  a(i) * b(i) -> R0
||      ADDF3   R0,R2,R2        ; a(i-1)*b(i-1) + R2 -> R2
*
        ADDF3   R0,R2,R0        ; a(N-1)*b(N-1) + R2 -> R0
*
*  RETURN SEQUENCE
*
        POP     RC              ; Restore RC
        POP     AR1             ; Restore AR1
        POP     AR0             ; Restore AR0
        POP     ST              ; Restore ST
        POPF    R2              ; Restore top 32 bits of R2
        POP     R2              ; Restore bottom 32 bits of R2
        RETS                    ; Return
*
*  end
*
  .end
```

12.2.2 Software Stack

The TMS320C30 has a software stack whose location is determined by the contents of the stack pointer register SP. The stack pointer increments from low to high values, and provisions should be made to accommodate the anticipated storage requirements. The stack can be used not only during the subroutine CALL and RETS, but also inside the subroutine as a place of temporary storage of the registers as shown in Example 12-2. SP always points to the last value pushed on the stack.

The CALL and CALLcond instructions push the value of the program counter on the stack, as do the interrupt routines. Then, RETScond and RETIcond pop the stack and place the value in the program counter. The integer value of any register can be pushed on and popped off the stack using the PUSH and POP instructions. There are two additional instructions, PUSHF and POPF, for floating point numbers. These instructions can be used to pop and push floating point numbers to registers R0-R7. This feature is very useful if it is desired to save the extended precision registers (see Example 12-2). By using PUSH and PUSHF on the same register, the lower 32 and the upper 32 bits are saved. PUSH saves the lower 32; PUSHF, the upper 32. To recover this extended precision number, a POPF can be done followed by POP. It is important to do the integer and floating-point PUSH and POP in the above order. POPF forces the last eight bits of the extended-precision registers to zero.

The stack pointer (SP) can be both read from, and written to. Multiple stacks for different program segments may be easily created. SP is not initialized by the hardware during reset. It is therefore important to remember to initialize its value so that SP points to a predetermined memory location. This avoids the problem of SP attempting to write into ROM or over other useful data.

12.2.3 Interrupt Service Routines

Interrupts on the TMS320C30 are prioritized and vectored. When an interrupt occurs, the corresponding flag is set in the Interrupt Flag Register IF. If the corresponding bit in the Interrupt Enable Register IE is set, and interrupts are enabled by having the GIE bit in the status register set to 1, interrupt processing begins. The interrupt flag register can also be written to. This enables the user to force an interrupt by software, or to clear interrupts without processing them.

The Interrupt Flag Register IF can be read, and action taken based on whether the interrupt has occurred. This is true even when the interrupt is disabled. This can be useful when an interrupt-driven interface is not implemented. Example 12-3 shows the case where a subroutine is called when interrupt 1 has not occurred.

Example 12-3. Use of Interrupts for Software Polling

```
*  TITL  INTERRUPT POLLING
         .
         .
         .
         TSTB    2,IF          ; Test if interrupt 1 has occurred
         CALLZ   SUBROUTINE    ; If not, call subroutine
         .
         .
         .
```

When interrupt processing begins, the program counter is pushed on the stack, and the interrupt vector is loaded in the program counter. Interrupts are then disabled by setting the GIE=0, and the program continues from the address loaded in the program counter. Since all interrupts are disabled, interrupt processing may proceed without further interruption, unless the interrupt service routine re-enables interrupts.

Except for very simple interrupt service routines, it is important to assure that the processor context is saved during execution of this routine. The context must be saved before executing the routine itself, and restored after the routine is finished. The procedure is called context switching. Context switching is also useful for subroutine calls, especially when extensive use is made of the auxiliary and the extended precision registers. Code examples of context switching and an interrupt service routine are provided in this section.

12.2.3.1 Context Switching

Context switching is commonly required when processing a subroutine call or interrupt. It may be quite extensive or simple, depending on system requirements. On the TMS320C30, the program counter is automatically pushed on the stack. If there is any important information in the other TMS320C30 registers, such as the status, auxiliary or extended-precision registers, these must be saved by special commands.

Examples 12-4 and 12-5 show saving and restoring of the TMS320C30 state. In both examples, the stack is used for saving the registers, and it expands towards higher addresses. If it is not desirable to use the stack pointed at by SP, a separate stack can be created using an auxiliary register as the stack pointer. The registers saved are:

- Extended-precision registers R0 through R7
- Auxiliary registers AR0 through AR7
- Data page pointer DP
- Index registers IR0 and IR1
- Block size register BK
- Status register ST
- Interrupt-related registers IE and IF
- I/O flag IOF
- Repeat-related registers RS, RE, and RC

Example 12-4. Context Save For The TMS320C30

```
*  TITL  CONTEXT-SAVE FOR THE TMS320C30
*
*
        .global SAVE
*
*  CONTEXT SAVE ON SUBROUTINE CALL OR INTERRUPT.
*
SAVE:
*
*  SAVE THE EXTENDED PRECISION REGISTERS
*
        PUSH    R0              ; Save the lower 32 bits of R0
        PUSHF   R0              ;      and the upper 32 bits
        PUSH    R1              ; Save the lower 32 bits of R1
        PUSHF   R1              ;      and the upper 32 bits
        PUSH    R2              ; Save the lower 32 bits of R2
        PUSHF   R2              ;      and the upper 32 bits
        PUSH    R3              ; Save the lower 32 bits of R3
        PUSHF   R3              ;      and the upper 32 bits
        PUSH    R4              ; Save the lower 32 bits of R4
        PUSHF   R4              ;      and the upper 32 bits
        PUSH    R5              ; Save the lower 32 bits of R5
        PUSHF   R5              ;      and the upper 32 bits
        PUSH    R6              ; Save the lower 32 bits of R6
        PUSHF   R6              ;      and the upper 32 bits
        PUSH    R7              ; Save the lower 32 bits of R7
        PUSHF   R7              ;      and the upper 32 bits
*
*  SAVE THE AUXILIARY REGISTERS
*
        PUSH    AR0             ; Save AR0
        PUSH    AR1             ; Save AR1
        PUSH    AR2             ; Save AR2
        PUSH    AR3             ; Save AR3
        PUSH    AR4             ; Save AR4
        PUSH    AR5             ; Save AR5
        PUSH    AR6             ; Save AR6
        PUSH    AR7             ; Save AR7
*
*  SAVE THE REST REGISTERS FROM THE REGISTER FILE
*
        PUSH    DP              ; Save data page pointer
        PUSH    IR0             ; Save index register IR0
        PUSH    IR1             ; Save index register IR1
        PUSH    BK              ; Save block-size register
        PUSH    ST              ; Save status register
        PUSH    IE              ; Save interrupt enable register
        PUSH    IF              ; Save interrupt flag register
        PUSH    IOF             ; Save I/O flag register
        PUSH    RS              ; Save repeat start address
        PUSH    RE              ; Save repeat end address
        PUSH    RC              ; Save repeat counter
*
*  SAVE IS COMPLETE
*
```

Example 12-5. Context-Restore For The TMS320C30

```
*
*  TITL  CONTEXT-RESTORE FOR THE TMS320C30
*
*
       .GLOBAL RESTR
*
*  CONTEXT RESTORE AT THE END OF A SUBROUTINE CALL OR INTERRUPT.
*
RESTR:
*
*  RESTORE THE REST REGISTERS FROM THE REGISTER FILE
*
       POP     RC          ; Restore repeat counter
       POP     RE          ; Restore repeat end address
       POP     RS          ; Restore repeat start address
       POP     IOF         ; Restore I/O flag register
       POP     IF          ; Restore interrupt flag register
       POP     IE          ; Restore interrupt enable register
       POP     ST          ; Restore status register
       POP     BK          ; Restore block-size register
       POP     IR1         ; Restore index register IR1
       POP     IR0         ; Restore index register IR0
       POP     DP          ; Restore data page pointer
*
*  RESTORE THE AUXILIARY REGISTERS
*
       POP     AR7         ; Restore AR7
       POP     AR6         ; Restore AR6
       POP     AR5         ; Restore AR5
       POP     AR4         ; Restore AR4
       POP     AR3         ; Restore AR3
       POP     AR2         ; Restore AR2
       POP     AR1         ; Restore AR1
       POP     AR0         ; Restore AR0
*
*  RESTORE THE EXTENDED PRECISION REGISTERS
*
       POPF    R7          ; Restore the upper 32 bits and
       POP     R7          ;     the lower 32 bits of R7
       POPF    R6          ; Restore the upper 32 bits and
       POP     R6          ;     the lower 32 bits of R6
       POPF    R5          ; Restore the upper 32 bits and
       POP     R5          ;     the lower 32 bits of R5
       POPF    R4          ; Restore the upper 32 bits and
       POP     R4          ;     the lower 32 bits of R4
       POPF    R3          ; Restore the upper 32 bits and
       POP     R3          ;     the lower 32 bits of R3
       POPF    R2          ; Restore the upper 32 bits and
       POP     R2          ;      the lower 32 bits of R2
       POPF    R1          ; Restore the upper 32 bits and
       POP     R1          ;     the lower 32 bits of R1
       POPF    R0          ; Restore the upper 32 bits and
       POP     R0          ;     The lower 32 bits of R0
*
*  RESTORE IS COMPLETE
*
```

12.2.3.2 Interrupt Priority

Interrupts on the TMS320C30 are automatically prioritized. This allows interrupts that occur simultaneously to be serviced in a predefined order. Infrequent, but lengthy, interrupt service routines may need to be interrupted by more frequently occurring interrupts. In Example 12-6, the interrupt service routine for INT2 temporarily modifies the interrupt enable register IE, to permit interrupt processing when an interrupt to INT0 (but no other interrupt) occurs. When the routine has finished processing, the register IE is restored to its original state. Notice that the RETI instruction not only pops the next program counter address from the stack, but also sets the GIE bit of the status register. This enables all interrupts which have their interrupt-enable bit set.

Example 12-6. Interrupt Service Routine

```
*  TITL  INTERRUPT SERVICE ROUTINE
*       .global ISR2
ENABLE  .set    2000h
MASK    .set    1
*
*  INTERRUPT PROCESSING FOR EXTERNAL INTERRUPT INT2-
*
ISR2:
        PUSH    ST              ; Save status register
        PUSH    DP              ; Save data page pointer
        PUSH    IE              ; Save interrupt enable register
        PUSH    R0              ; Save lower 32 bits and
        PUSHF   R0              ;    upper 32 bits of R0
        PUSH    R1              ; Save lower 32 bits and
        PUSHF   R1              ;   upper 32 bits of R1
        LDI     MASK,IE         ; Unmask only INT0
        OR      ENABLE,ST       ; Enable all interrupts
*
*  MAIN PROCESSING SECTION FOR ISR2
        .
        .
        .
        XOR     ENABLE,ST       ; Disable all interrupts
        POPF    R1              ; Restore upper 32 bits and
        POP     R1              ;    lower 32 bits of R1
        POPF    R0              ; Restore upper 32 bits and
        POP     R0              ;    lower 32 bits of R0
        POP     IE              ; Restore interrupt enable register
        POP     DP              ; Restore data page register
        POP     ST              ; Restore status register
*
        RETI                    ; Return and enable interrupts
```

12.2.4 Delayed branches

The TMS320C30 offers the capability of single-cycle branching through the use of the delayed branches. The delay branches operate like regular branches but do not flush the pipeline. Instead, the three instructions following a delayed branch are also executed. As discussed in the section on program-flow control, the only limitation is that the three instructions following a delayed branch cannot be a:

- Branch (standard or delayed)
- Call to a subroutine
- Return from a subroutine
- Return from an interrupt
- Repeat instructions
- A TRAP instruction
- An IDLE instruction

Conditional delayed branches use the conditions that exist at the end of the instruction immediately preceding the delayed branch. Sometimes, a branch is necessary in the flow of a program, but less than three instructions can be placed after a delayed branch. For faster execution, it is still advantageous to use a delayed branch. This is shown in Example 12-7, with NOP's taking the place of the unused instructions. The trade-off is more instruction words for less execution time.

Example 12-7. Delayed Branch Execution

```
*  TITL  DELAYED BRANCH EXECUTION
          .
          .
          .
          .
          LDF     *+AR1(5),R2   ;  Load contents of memory to R2
          BGED    SKIP          ;  If loaded number >=0, branch (delayed)
          LDFN    R2,R1         ;  If loaded number <0, load it to R1
          SUBF    3.0,R1        ;  Subtract 3 from R1
          NOP                   ;  Dummy operation to complete delayed
*                               ;  branch
          MPYF    1.5,R1        ;  Continue here if loaded number <0
          .
          .
          .
SKIP      LDF     R1,R3         ;  Continue here if loaded number >=0
```

12.2.5 Repeat Modes

The TMS320C30 supports looping without any overhead. For that purpose, there are two instructions: RPTB repeats a block of code, and RPTS repeats a single instruction. There are three control registers RS (repeat start address), RE (repeat end address), and RC (repeat counter). These contain the parameters that specify loop execution (refer to Section 7.1 for a complete description of RPTB and RPTS). RS and RE are automatically set from the code, while RC has to be set by the user, as shown in the examples below.

12.2.5.1 Block Repeat

Example 12-8 shows an application of the block repeat construct. In this example, an array of 64 elements is "flipped over" by exchanging the elements that are equidistant from the end of the array. In other words, if the original array is:

a(1), a(2),..., a(31), a(32),..., a(64);

the final array after the rearrangement will be:

a(64), a(63),..., a(32), a(31),..., a(1).

Note that since the exchange operation is done on two elements at the same time, there is a need of 32 operations. The repeat counter RC is initialized to 31. In general, if RC contains the number N, the loop will be executed N+1 times. The loop is defined by the RPTB instruction and the EXCH label.

Example 12-8. Loop Using Block Repeat

```
*  TITL  LOOP USING BLOCK REPEAT
*
*  THIS CODE SEGMENT EXCHANGES THE VALUES OF ARRAY ELEMENTS THAT ARE
*  SYMMETRIC AROUND THE MIDDLE OF THE ARRAY.
*
        .
        .
        .
        LDI     @ADDR,AR0       ; AR0 points to the beginning of the array
        LDI     AR0,AR1
        ADDI    63,AR1          ; AR1 points to the end of the
*                               ;    64-element array
        LDI     31,RC           ; Initialize repeat counter
*
        RPTB    EXCH            ; Repeat RC+1 times between here and
*                                  EXCH
        LDI     *AR0,R0         ; Load one memory element in R0,
||      LDI     *AR1,R1         ;    and the other in R1
EXCH    STI     R1,*AR0++(1)    ; Then, exchange their locations
||      STI     R0,*AR1--(1)
        .
        .
        .
```

Section 7.1.2 specifies restrictions in the block-repeat construct. The basic rule is, since the program counter is modified at the end of the loop according to the contents of the registers RS, RE, and RC, no operation should attempt

to modify the repeat counter or the program counter at the end of the loop in a different way.

In principle, it is possible to nest repeat blocks. However, there is only one set of control registers RS, RE, and RC. It is therefore necessary to save these registers before entering an inside loop. It may be more economical to implement a nested loop by the more traditional method of a register serving as a counter and then using a delayed branch rather than applying the above approach.

Example 12-9 shows another example of using the block repeat to find a maximum of 147 numbers.

Example 12-9. Use of Block Repeat to Find a Maximum

```
*
*
*   TITL  USE OF BLOCK REPEAT TO FIND A MAXIMUM
*
*   THIS ROUTINE FINDS THE MAXIMUM OF N=147 NUMBERS
*
          .
          .
          .
          LDI     146,RC          ;  Initialize repeat counter to 147-1
          LDI     @ADDR,AR0       ;  AR0 points to the beginning of the array
          LDF     *AR0++(1),R0    ;  Initialize MAX to the first value
*
          RPTB    LOOP
          CMPF    *AR0++(1),R0    ;  Compare number to the maximum
LOOP      LDFLT   *-AR0(1),R0     ;  If greater, this is a new maximum
          .
          .
          .
```

12.2.5.2 Single-Instruction Repeat

The single instruction repeat operates using the control registers RS, RE, and RC, in the same way as the block repeat. The advantage over the block repeat is that the instruction is fetched only once, and then the buses are available for moving operands. One difference to note is that the single-instruction repeat construct is not interruptible, while block repeat is interruptible.

Example 12-10 shows an application of the repeat-single construct. In this example, the sum of the products of two arrays is computed. The arrays are not necessarily different. If the arrays are a(i) and b(i), each of length N=512, register R0 will contain, after computation, the quantity:

a(1)b(1)+a(2)b(2)+...+a(N)b(N).

The value of the repeat counter (RC) is specified to be 511 in the instruction. If RC contains the number N, the loop will be executed N+1 times.

Example 12-10.

```
*  TITL  LOOP USING SINGLE REPEAT
*
*                                        N
*  THIS CODE SEGMENT COMPUTES          Σ   a(i)b(i)
*                                      i=1
*
        .
        .
        .
        LDI     @ADDR1,AR0    ; AR0 points to array a(i)
        LDI     @ADDR2,AR1    ; AR1 points to array b(i)
*
        LDF     0.0,R0        ; Initialize R0
*
        MPYF3   *AR0++(1),*AR1++(1),R1
*                             ; Compute first product
        RPTS    511           ; Repeat 512 times
*
        MPYF3   *AR0++(1),*AR1++(1),R1,R0  ;  Compute next product
||      ADDF3   R1,R0,R0      ; and accumulate the previous one
*
        ADDF    R1,R0         ; One final addition
        .
        .
        .
```

12.2.6 Computed GOTO's

Occasionally, it is convenient to select during runtime, and not during assembly, what subroutine needs to be executed. The TMS320C30 offers the capability of a computed GOTO that can satisfy such a need. The computed GOTO is implemented using the CALLcond instruction in the register addressing mode. This instruction uses the contents of the register as the address of the call. Example 12-11 shows the case of a task controller.

Example 12-11. Computed GOTO

```
*  TITL  COMPUTED GOTO
*
*  TASK CONTROLLER
*
*  THIS MAIN ROUTINE CONTROLS THE ORDER OF TASK EXECUTION(6 TASKS
*  IN THE PRESENT EXAMPLE).  TASK0 THROUGH TASK5 ARE THE NAMES OF
*  SUBROUTINES TO BE CALLED. THEY ARE EXECUTED IN ORDER, TASK0,
*  TASK1, . . .TASKS.  WHEN AN INTERRUPT OCCURS, THE INTERRUPT
*  SERVICE ROUTINE IS EXECUTED, AND THE PROCESSOR CONTIMUES
*  WITH THE INSTRUCTION FOLLOWING THE IDLE INSTRUCTION. THIS
*  ROUTINE SELECTS THE TASK APPROPRIATE FOR THE CURRENT CYCLE,
*  CALLS THE TASK AS A SUBROUTINE, AND BRANCHES BACK TO THE IDLE
*  TO WAIT FOR THE NEXT SAMPLE INTERRUPT WHEN THE SCHEDULED TASK
*  HAS COMPLETED EXECUTION. R0 HOLDS THE OFFSET FROM THE BASE
*  ADDRESS OF THE TASK TO BE EXECUTED.
*
*
        LDI     5,R0            ;  Initialize R0
        LDI     @ADDR,AR1       ;  AR1 holds the base address of the table
WAIT    IDLE                    ;  Wait for the next interrupt
        ADDI    *AR1,R0,R1      ;  Add the base address to the table
*                               ;    Entry number
        SUBI    1,R0            ;  Decrement R0
        LDILT   5,R0            ;  If R0<0, reinitialize it to 5
        CALLU   R1              ;  Execute appropriate task
        BR      WAIT
*
TSKSEQ  .word   TASK5           ;  Address of TASK5
        .word   TASK4           ;  Address of TASK4
        .word   TASK3           ;  Address of TASK3
        .word   TASK2           ;  Address of TASK2
        .word   TASK1           ;  Address of TASK1
        .word   TASK0           ;  Address of TASK0
ADDR    .WORD   TSKSEQ
```

12.3 Logical and Arithmetic Operations

The TMS320C30 instruction set supports both integer and floating point arithmetic and logical operations. The basic functions of such instructions can be combined to form more complex operations. This section examines examples of such operations, such as:

- Bit manipulation
- Block moves
- Bit-reversed addressing
- Integer and floating-point division
- Square root
- Extended precision arithmetic
- Floating point format conversion between IEEE and TMS320C30 formats.

12.3.1 Bit Manipulation

The instructions of the TMS320C30 for the usual logical operations, such as AND, OR, NOT, ANDN, and XOR, can be used together with the shift instructions for bit manipulation. In addition to these instructions, there is a special instruction, TSTB, for testing bits. TSTB does the same operation as AND, but the result of the logical AND is not written anywhere and is only used to set the condition flags. Examples 12-12 and 12-13 demonstrate the use of the several instructions for bit manipulation and testing.

Example 12-12. Use of TSTB for Software-Controlled Interrupt

```
*   TITL  USE OF TSTB FOR SOFTWARE-CONTROLLED INTERRUPT
*
*   IN THIS EXAMPLE, ALL INTERRUPTS HAVE BEEN DISABLED BY
*   RESETTING THE GIE BIT OF THE STATUS REGISTER.  WHEN AN
*   INTERRUPT ARRIVES, IT IS STORED IN THE IF REGISTER.  THE
*   PRESENT EXAMPLE ACTIVATES THE INTERRUPT SERVICE ROUTINE INTR
*   WHEN IT DETECTS THAT INT2- HAS OCCURRED.
        .
        .
        .
        TSTB    4,IF            ; Check if bit 2 of IF is set,
        CALLNZ  INTR            ;    and, if so, call subroutine INTR
        .
        .
        .
```

Example 12-13. Copy a Bit From One Location to Another

```
*   TITL  COPY A BIT FROM ONE LOCATION TO ANOTHER
*
*   BIT I OF R1 NEEDS TO BE COPIED TO BIT J OF R2.
*   AR0 POINTS TO A LOCATION HOLDING I, AND IT IS ASSUMED THAT THE
*      NEXT MEMORY LOCATION HOLDS THE VALUE J.
*
*                                  I
*                                  ↓
*                        [          | |          ]      R1
*
*
*                                        J
*                                        ↓
*                        [                | |  ]      R2
*
*
*
*                        [          I          ]      *AR0
*
*
*
*                        [          J          ]      *(AR0+1)
*
*
*
        .
        .
        .
        LDI     1,R0
        LSH     *AR0,R0       ;  Shift 1 to align it with bit I
        TSTB    R1,R0         ;  Test the I-th bit of R1
        BZD     CONT          ;  If bit = 0, branch delayed
        LDI     1,R0
        LSH     *+AR0(1),R0   ;  Align 1 with J-th location
        ANDN    R0,R2         ;  If bit = 0, reset J-th bit of R2
        OR      R0,R2         ;  If bit = 1, set J-th bit of R2
CONT    .
        .
        .
        .
```

12

12.3.2 Block Moves

Since the TMS320C30 directly addresses a large amount of memory, blocks of data or program code can be stored off-chip in slow memories and then loaded on-chip for faster execution. Data can also be moved from on-chip to off-chip for storage or for multiprocessor data trasfers.

Such data transfers can be accomplished very efficiently in parallel with CPU operations, using the DMA. The DMA operation is explained in detail in an earlier section of this manual. An alternative to DMA is to perform data transfers under program control using load and store instructions in a repeat mode. Example 12-14 shows the case where a block of 512 floating-point numbers are transferred from external memory to block 1 of the on-chip RAM.

Example 12-14. Block Move Under Program Control

```
*  TITL  BLOCK MOVE UNDER PROGRAM CONTROL
*
extern  .word   01000H
block1  .word   0809C00H
        .
        .
        .
        LDI     @extern,AR0   ;  Source address
        LDI     @block1,AR1   ;  Destination address

        LDF     *AR0++,R0     ;  Load the first number

        RPTS    510           ;  Repeat following instruction 511 times
        LDF     *AR0++,R0     ;  Load the next number, and...
||      STF     R0,*AR1++     ;  store the previous one

        STF     R0,*AR1       ;  Store the last number
        .
        .
        .
```

12.3.3 Bit-Reversed Addressing

For an efficient implementation of Fast Fourier Transforms (FFT), the TMS320C30 offers the capability of bit-reversed addressing. If the data to be transformed is in the correct order, the final result of the FFT is scrambled (in bit-reversed order). To recover the frequency-domain data in the correct order, certain memory locations have to be swapped. The bit-reversed addressing mode offers the alternative of not doing this swapping. The next time data needs to be accessed, the access is done in a bit-reversed manner rather than sequentially.

In bit-reversed addressing, IR0 holds a value equal to one-half the size of the FFT, if real and imaginary data are stored in separate arrays. During accessing, the auxiliary register is indexed by IR0, but with reverse carry propagation. Example 12-15 illustrates a 512-point complex FFT being moved from the place of computation (pointed at by AR0) to a location pointed at by AR1. In this example, real and imaginary parts XR(i) and XI(i) of the data are not stored in separate arrays, but they are interleaved XR(0),XI(0),XR(1),XI(1),...,

XR(N-1),XI(N-1). Because of this arrangement, the length of the array is 2N instead of N, and IR0 is set to 512 instead of 256.

Example 12-15. Bit-Reversed Addressing

```
*
*  TITL  BIT-REVERSED ADDRESSING
*
*  THIS EXAMPLE MOVES THE RESULT OF THE 512-POINT FFT
*  COMPUTATION, POINTED AT BY AR0, TO A LOCATION POINTED AT
*  BY AR1.  REAL AND IMAGINARY POINTS ARE ALTERNATING.
        .
        .
        .
        LDI     512,IR0
        LDI     2,IR1
        LDI     511,RC          ;  Repeat 511+1 times
        LDF     *+AR0(1),R1     ;  Load first imaginary point
        RPTB    LOOP
*
        LDF     *AR0++(IR0)B,R0 ;  Load real value (and point
||      STF     R1,*+AR1(1)     ;  to next location) and store
*                               ;  the imaginary value
LOOP    LDF     *+AR0(1),R1     ;  Load next imaginary point and store
||      STF     R0,*AR1++(IR1)  ;  previous real value
        .
        .
        .
```

12.3.4 Integer and Floating-point Division

Although division is not implemented as a single instruction in the TMS320C30, the instruction set provides the necessary capabilities for an efficient division routine. Integer and floating-point division will be examined separately because different algorithms are used.

12.3.4.1 Integer Division

Division is implemented on the TMS320C30 by repeated subtractions using SUBC, a special conditional subtract instruction. Consider the case of a 32-bit positive dividend with i significant bits (and 32-i sign bits), and a 32-bit positive divisor with j significant bits (and 32-j sign bits). The repetition of the SUBC command i-j+1 times produces a 32 bit result where the lower i-j+1 bits are the quotient, and the upper 31-i+j bits, the remainder of the division.

SUBC implements binary division in the same manner as in long division. The divisor (assumed to be smaller than the dividend) is shifted left i-j times to be aligned with the dividend. Then, using SUBC, the shifted divisor is subtracted from the dividend. For each subtract that does not produce a negative answer, the dividend is replaced by the difference. It is then shifted to the left, and a one is put in the LSB. If the difference is negative, the dividend is simply shifted left by one. This operation is repeated i-j+1 times.

As an example, consider the division of 33 by 5 using both long division and the SUBC method. In this case, i=6, j=3, and the SUBC operation is repeated 6-3+1=4 times.

LONG DIVISION:

```
                                  | 00000000000000000000000000000110      Quotient
00000000000000000000000000000101  | 00000000000000000000000000100001
                                                                -101
                                                                 1101
                                                                -101
                                                                   11      Remainder
```

SUBC METHOD:

```
00000000000000000000000000100001      Dividend
00000000000000000000000000101000      Divisor (aligned)
        Negative difference                (1st SUBC command)
          ↓
00000000000000000000000001000010      New Dividend + Quotient
00000000000000000000000000101000      Divisor
00000000000000000000000000011010      Difference (>0) (2nd SUBC command)
          ↓
00000000000000000000000000110101      New Dividend + Quotient
00000000000000000000000000101000      Divisor
00000000000000000000000000001101      Difference (>0) (3rd SUBC command)
          ↓
00000000000000000000000000011011      New Dividend + Quotient
00000000000000000000000000101000      Divisor
        Negative difference                (4th SUBC command)
          ↓
|00000000000000000000000000110110|     Final Result
|          ↓                  |  ↓  |
|        Remainder            | Quot.|
```

When using the SUBC command, both the dividend and the divisor must be positive. Example 12-16 shows a realization of the integer division, where the sign of the quotient is properly handled. The last instruction before returning modifies the condition flag, in case subsequent operations depend on the sign of the result.

12

Example 12-16. Integer Division

```
*
*  TITL  INTEGER DIVISION
*
      SUBROUTINE  DIVI
*
*
*     INPUTS:          SIGNED INTEGER DIVIDEND IN R0,
*                      SIGNED INTEGER DIVISOR IN R1.
*
*     OUTPUT:          R0/R1 into R0.
*
*     REGISTERS USED: R0-R3, IR0, IR1
*
*     OPERATION:       1. NORMALIZE DIVISOR WITH DIVIDEND
*                      2. REPEAT SUBC
*                      3. QUOTIENT IS IN LSBs OF RESULT
*
*     CYCLES:          31-62 (DEPENDS ON AMOUNT OF NORMALIZATION)
*
        .globl  DIVI

SIGN    .set    R2
TEMPF   .set    R3
TEMP    .set    IR0
COUNT   .set    IR1


*   DIVI - SIGNED DIVISION


DIVI:
*
*  DETERMINE SIGN OF RESULT.  GET ABSOLUTE VALUE OF OPERANDS.
*
        XOR     R0,R1,SIGN    ; Get the sign
        ABSI    R0
        ABSI    R1

        CMPI    R0,R1         ; Divisor > dividend ?
        BHID    ZERO          ;  If so, return 0
*
*  NORMALIZE OPERANDS.  USE DIFFERENCE IN EXPONENTS AS SHIFT COUNT
*  FOR DIVISOR, AND AS REPEAT COUNT FOR 'SUBC'.
*
        FLOAT   R0,TEMP       ; Normalize dividend
        PUSHF   TEMPF         ; PUSH as float
        POP     COUNT         ; POP as int
        LSH     -24,COUNT     ; Get dividend exponent

        FLOAT   R1,TEMPF      ; Normalize divisor
        PUSHF   TEMPF         ; PUSH as float
        POP     TEMP          ; POP as int
        LSH     -24,TEMP      ; Get divisor exponent
        SUBI    TEMP,COUNT    ; Get difference in exponents
        LSH     COUNT,R1      ; Align divisor with dividend
```

```
*
*  DO COUNT+1 SUBTRACT & SHIFTS.
*
        RPTS    COUNT
        SUBC    R1,R0
*
*  MASK OFF THE LOWER COUNT+1 BITS OF R0
*
        SUBRI   31,COUNT        ; Shift count is (32 - (COUNT+1))
        LSH     COUNT,R0        ; Shift left
        NEGI    COUNT
        LSH     COUNT,R0        ; Shift right to get result
*
*   CHECK SIGN AND NEGATE RESULT IF NECESSARY.
*
        NEGI    R0,R1           ; Negate result
        ASH     -31,SIGN        ; Check sign
        LDINZ   R1,R0           ; If set, use negative result
        CMPI    0,R0            ; Set status from result
        RETS
*
*  RETURN ZERO.
*
ZERO:
        LDI     0,R0
        RETS
        .end
```

If the dividend is less than the divisor and fractional division is desired, a division can be performed after determining the desired accuracy of the quotient in bits. If the desired accuracy is k bits, start by shifting the dividend left by k positions. Then apply the algorithm described above, where i should now be replaced by i+k. It is assumed that i+k is less than 32.

12.3.4.2 Computation of Floating-point Inverse and Division

This section presents a method of implementing floating-point division on the TMS320C30. Since the algorithm outlined here computes the inverse of a number v, to divide y/v, multiply y by the inverse of v.

The computation of 1/v is based on the following iterative algorihm. At the i-th iteration, the estimate x[i] of 1/v is computed from v, and the previous estimate x[i-1] according to the formula:

$$x[i] = x[i-1] * (2.0 - v * x[i-1])$$

To start the operation, an initial estimate x[0] is needed. If $v=a*2^e$, a good initial estimate is:

$$x[0] = 1.0 * 2^{-e-1}$$

Example 12-17 shows the implementation of this algorithm on the TMS320C30, where the iteration has been applied 5 times. The choice of the number of iterations was based on the desire to have maximum accuracy. The accuracy offered by the single-precision floating-point format is 2^{-23}=1.192E-7. If more accuracy is desired, more iterations can be used. If

less accuracy is acceptable, the execution speed of this implementation can be increased by reducing the number of iterations.

This algorithm properly treats the boundary conditions, when the input number is either zero or it has a very large value. When the input is zero, the exponent e=-128. Then the calculation of x[0] yields an exponent equal to -(-128)-1=127 and the algorithm will overflow and saturate. On the other hand, in the case of a very large number, e=127, the exponent of x[0] will be -127-1=-128. This will cause the algorithm to yield zero, which is a reasonable handling of that boundary condition.

Example 12-17. Inverse of a Floating-Point Number

```
*
*  TITL  INVERSE OF A FLOATING-POINT NUMBER
*
*
*  SUBROUTINE INVF
*
*
*  THE FLOATING-POINT NUMBER v IS STORED IN R0.  AFTER THE
*  COMPUTATION IS COMPLETED, 1/v IS ALSO STORED IN R0.
*
*  TYPICAL CALLING SEQUENCE:
*       LDF     v, R0
*       CALL    INVF
*
*    ARGUMENT ASSIGNMENTS:
*    ARGUMENT | FUNCTION
*    ---------+------------------
*    R0       | v = NUMBER TO FIND THE RECIPROCAL OF (UPON THE CALL)
*    R0       | 1/v (UPON THE RETURN)
*
*  REGISTER USED AS INPUT: R0
*  REGISTERS MODIFIED: R0, R1, R2, R3
*  REGISTER CONTAINING RESULT: R0
*
*       CYCLES: 35          WORDS: 32
*
*
       .global INVF
*
INVF:  LDF     R0,R3        ; v is saved for later.
       ABSF    R0           ; The algorithm uses v = |v|.
*
*  EXTRACT THE EXPONENT OF v.
*
       PUSHF   R0
       POP     R1
       ASH     -24,R        ; The 8 LSBs of R1 contain the exponent
*                           ;   of v.
*
*  x[0] FORMATION GIVEN THE EXPONENT OF v.
*
       NEGI    R1
       SUBI    1,R1         ; Now we have -e-1, the exponent of x[0].
       ASH     24,R1
       PUSH    R1
       POPF    R1           ; Now R1 = x[0] = 1.0 * 2**(-e-1).
*
```

```
*   NOW THE ITERATIONS BEGIN.
*
        MPYF    R1,R0,R2        ;  R2 = v * x[0]
        SUBRF   2.0,R2          ;   R2 = 2.0 - v * x[0]
        MPYF    R2,R1           ;  R1 = x[1] = x[0] * (2.0 - v * x[0])
*
        MPYF    R1,R0,R2        ;  R2 = v * x[1]
        SUBRF   2.0,R2          ;  R2 = 2.0 - v * x[1]
        MPYF    R2,R1           ;  R1 = x[2] = x[1] * (2.0 - v * x[1])
*
        MPYF    R1,R0,R2        ;  R2 = v * x[2]
        SUBRF   2.0,R2          ;  R2 = 2.0 - v * x[2]
        MPYF    R2,R1           ;  R1 = x[3] = x[2] * (2.0 - v * x[2])
*
        MPYF    R1,R0,R2        ;  R2 = v * x[3]
        SUBRF   2.0,R2          ;  R2 = 2.0 - v * x[3]
        MPYF    R2,R1           ;  R1 = x[4] = x[3] * (2.0 - v * x[3])
*
        RND     R1              ;  This minimizes error in the LSBs.
*
*   FOR THE LAST ITERATION WE USE THE FORMULATION:
*   x[5] = (x[4] * (1.0 - (v * x[4]))) + x[4]
*
        MPYF    R1,R0,R2        ;  R2 = v * x[4] = 1.0..01.. => 1
        SUBRF   1.0,R2          ;  R2 = 1.0 - v * x[4] = 0.0..01... => 0
        MPYF    R1,R2           ;  R2 = x[4] * (1.0 - v * x[4])
        ADDF    R2,R1           ;  R2 = x[5] = (x[4]*(1.0-(v*x[4])))+x[4]
*
        RND     R1,R0           ;  Round since this is follow by a MPYF.
*
*   NOW THE CASE OF v < 0 IS HANDLED.
*
        NEGF    R0,R2
        LDF     R3,R3           ;  This sets condition flags.
        LDFN    R2,R0           ;  If v < 0, then R0 = -R0
*
        RETS
*
*   END
*
        .end
```

12.3.5 Square Root

The implementation of the square root on the TMS320C30 is done by an iterative algorithm very similar to the one used for the computation of the inverse. This algorithm computes the inverse of the square root of a number v, 1/SQRT(v). To derive SQRT(v), multiply this result by v. Since in many applications, division by the square root of a number is desirable, the output of the algorithm saves the effort to compute the inverse of the square root.

At the i-th iteration, the estimate x[i] of 1/SQRT(v) is computed from v and the previous estimate x[i-1] according to the formula:

$$x[i] = x[i-1] * (1.5 - (v/2) * x[i-1] * x[i-1])$$

To start the operation, an initial estimate x[0] is needed. If $v=a*2^e$, a good initial estimate is:

$$x[0] = 1.0 * 2^{-e/2}$$

Example 12-18 shows the implementation of this algorithm on the TMS320C30, where the iteration has been applied 5 times. The choice of the number of iterations was based on the desire to have maximum accuracy. If more accuracy is desired, more iterations can be used. If less accuracy is acceptable, the execution speed of this implementation may be increased by reducing the number of iterations.

Example 12-18. Square Root of a Floating-Point Number

```
*
*  TITL  SQUARE ROOT OF A FLOATING-POINT NUMBER
*
*
*  SUBROUTINE SQRT
*
*  THE FLOATING POINT NUMBER v IS STORED IN R0.  AFTER THE
*  COMPUTATION IS COMPLETED, SQRT(v) IS ALSO STORED IN R0.  NOTE
*  THAT THE ALGORITHM ACTUALLY COMPUTES 1/SQRT(v).
*
*
*  TYPICAL CALLING SEQUENCE:
*
*       LDF v,  R0
*       CALL    SQRT
*  ARGUMENT ASSIGNMENTS:
*    ARGUMENT | FUNCTION
*    ---------+-----------------
*    R0       | v = NUMBER TO FIND THE SQUARE ROOT OF
*             |     (UPON THE CALL)
*    R0       | SQRT(v) (UPON THE RETURN)
*
*  REGISTER USED AS INPUT: R0
*  REGISTERS MODIFIED: R0, R1, R2, R3
*  REGISTER CONTAINING RESULT: R0
*
*  CYCLES: 39   WORDS: 33
*
               .global SQRT
*
*  EXTRACT THE EXPONENT OF V.
*
SQRT:   LDF     R0,R3          ; Save v
        RETSLE                 ; Return if number non-positive
        PUSHF   R0
        POP     R1
        ASH     -25,R1         ; The 8 LSBs of R1 contain 1/2 the exponent
*                              ;   of v.
*
*  X[0] FORMATION GIVEN THE EXPONENT OF V.
*
        NEGI    R1
        ASH     24,R1
        PUSH    R1
        POPF    R1             ; Now R1 = x[0] = 1.0 * 2**(-e/2).
*
*  GENERATE V/2.
*
        MPYF    0.5,R0
*
*  NOW THE ITERATIONS BEGIN.
*
        MPYF    R1,R1,R2       ; R2 = x[0] * x[0]
        MPYF    R0,R2          ; R2 = (v/2) * x[0] * x[0]
        SUBRF   1.5,R2         ; R2 = 1.5 - (v/2) * x[0] * x[0]
        MPYF    R2,R1          ; R1 = x[1] = x[0] *
*                              ;   (1.5 - (v/2)*x[0]*x[0])
```

```
        MPYF    R1,R1,R2        ; R2 = x[1] * x[1]
        MPYF    R0,R2           ; R2 = (v/2) * x[1] * x[1]
        SUBRF   1.5,R2          ; R2 = 1.5 - (v/2) * x[1] * x[1]
        MPYF    R2,R1           ; R1 = x[2] = x[1] *
*                               ;    (1.5 - (v/2)*x[1]*x[1])
        MPYF    R1,R1,R2        ; R2 = x[2] * x[2]
        MPYF    R0,R2           ; R2 = (v/2) * x[2] * x[2]
        SUBRF   1.5,R2          ; R2 = 1.5 - (v/2) * x[2] * x[2]
        MPYF    R2,R1           ; R1 = x[3] = x[2]
*                               ;    * (1.5 - (v/2)*x[2]*x[2])
*
        MPYF    R1,R1,R2        ; R2 = x[3] * x[3]
        MPYF    R0,R2           ; R2 = (v/2) * x[3] * x[3]
        SUBRF   1.5,R2          ; R2 = 1.5 - (v/2) * x[3] * x[3]
        MPYF    R2,R1           ; R1 = x[4] = x[3]
*                               ;    * (1.5 - (v/2)*x[3]*x[3])
*
        MPYF    R1,R1,R2        ; R2 = x[4] * x[4]
        MPYF    R0,R2           ; R2 = (v/2) * x[4] * x[4]
        SUBRF   1.5,R2          ; R2 = 1.5 - (v/2) * x[4] * x[4]
        MPYF    R2,R1           ; R1 = x[5] = x[4]
*                               ;    * (1.5 - (v/2)*x[4]*x[4])
*
*
        RND     R1,R0           ; Round
*
        MPYF    R3,R0           ; Sqrt(v) from sqrt(v**(-1))
*
        RETS
*
*  end
*
        .end
```

12.3.6 Extended-Precision Arithmetic

The TMS320C30 offers 32 bits of precision for integer arithmetic, and 24 bits of precision in the mantissa for floating point arithmetic. For higher precision in floating-point operations, the eight extended-precision registers R0 to R7 contain eight more bits of accuracy. Since no comparable extension is available for fixed-point arithmetic, this section discusses how fixed-point double precision can be achieved using the capabilities of the processor. The technique consists of performing the arithmetic by parts, similar to the way in which longhand arithmetic is done.

The instruction set has operations ADDC (Add with Carry) and SUBB (Subtract with Borrow) which use the status carry bit for extended-precision arithmetic. The carry bit is affected by the arithmetic operations of the ALU, and the rotate and shift instructions. It can also be manipulated directly by setting the status register to certain values. For proper operation, the overflow mode bit should be reset (OVM = 0) so the accumulator results will not be loaded with the saturation values. Example 12-19 and Example 12-20 show 64-bit addition and 64-bit subtraction. The first operand is stored in the registers R0 (low word) and R1 (high word). The second operand is stored in R2 and R3 respectively. The result is stored in R0 and R1.

Example 12-19. 64-Bit Addition

```
*   TITL  64-BIT ADDITION
*
*   TWO 64-BIT NUMBERS ARE ADDED TO EACH OTHER PRODUCING A 64-BIT
*   RESULT.  THE NUMBERS X (R1,R0) AND Y (R3,R2) ARE ADDED,
*   RESULTING IN W (R1,R0).
*
*         R1  R0
*       + R3  R2
*       --------
*         R1  R0
*
        ADDI    R2,R0
        ADDC    R3,R1
```

Example 12-20. 64-Bit Subtraction

```
*   TITL  64-BIT SUBTRACTION
*
*   TWO 64-BIT NUMBERS ARE SUBTRACTED FROM EACH OTHER PRODUCING
*   A 64-BIT RESULT.  THE NUMBERS X (R1,R0) AND Y (R3,R2) ARE
*   SUBTRACTED, RESULTING IN W (R1,R0).
*
*         R1  R0
*       - R3  R2
*       ---------
*         R1  R0
*
        SUBI    R2,R0
        SUBB    R3,R1
```

When two 32-bit numbers are multiplied, a 64-bit product results. The procedure for multiplication is to split the 32-bit magnitude values of the multiplicand X and the multiplier Y into two parts (X1,X0) and (X3,X2) respectively with 16 bits each. The operation is done on unsigned numbers, and the product is adjusted for the sign bit. Example 12-21 shows the implementation of a 32 bit X 32 bit multiplication.

Example 12-21. 32 by 32 Bit Multiplication

```
*
*  TITL  32 X 32 BIT MULTIPLICATION
*
*
*  SUBROUTINE  EXTMPY
*
*  FUNCTION: TWO 32-BIT NUMBERS ARE MULTIPLIED, PRODUCING A 64-BIT
*  RESULT.  THE TWO NUMBERS (X and Y) ARE EACH SEPARATED INTO TWO
*  PARTS (X1 X0) AND (Y1 Y0), WHERE X0, X1, Y0, AND Y1 ARE 16 BITS.
*  THE TOP BIT IN X1 AND Y1 IS THE SIGN BIT.  THE PRODUCT IS
*  IN TWO WORDS (W0 AND W1).  THE MULTIPLICATION IS PERFORMED ON
*  POSITIVE NUMBERS, AND THE SIGN IS DETERMINED AT THE END.
*
*
*              X1 X0        BITS OF PRODUCTS
*       X      Y1 Y0        (NOT COUNTING SIGN)       PRODUCT
*      -------------
*              X0*Y0                   16+16          P1
*           X0*Y1                    16+16            P2
*           X1*Y0                    16+16            P3
*        X1*Y1                    16+16               P4
*      -------------
*        W1       W0
*
*
*  ARGUMENT ASSIGNMENTS
*    ARGUMENT | FUNCTION
*    ---------+--------------------
*    R0       | MULTIPLIER AND LOW WORD OF THE PRODUCT
*    R1       | MULTIPLICAND AND UPPER WORD OF THE PRODUCT
*
*
*  REGISTERS USED AS INPUT: R0, R1
*  REGISTERS MODIFIED: R0, R1, R2, R3, R4, AR0, AR1,
*  REGISTER CONTAINING RESULT: R0,R1
*
*
*       CYCLES: 28 (WORST CASE)       WORDS: 25
*
        .GLOBAL EXTMPY
*
EXTMPY  XOR3    R0,R1,AR0     ; Store sign
        ABSI    R0            ; Absolute values of X
        ABSI    R1            ;     and Y

*
*  SEPARATE MULTIPLIER AND MULTIPLICAND INTO TWO PARTS
*
        LDI     -16,AR1
        LSH3    AR1,R0,R2     ; R2 = X1 = Upper 16 bits of X
        AND     0FFFFH,R0     ; R0 = X0 = Lower 16 bits of X
        LSH3    AR1,R1,R3     ; R3 = Y1 = Upper 16 bits of Y
        AND     0FFFFH,R1     ; R1 = Y0 = Lower 16 bits of Y
```

```
*
*   CARRY OUT THE MULTIPLICATION
*
        MPYI3   R0,R1,R4        ; X0*Y0 = P1
        MPYI    R3,R0           ; X0*Y1 = P2
        MPYI    R2,R1           ; X1*Y0 = P3
        ADDI    R0,R1           ; P2+P3
        MPYI    R2,R3           ; X1*Y1 = P4
*
        LDI     R1,R2
        LSH     16,R2           ; Lower 16 bits of P2+P3
        CMPI    0,AR0           ; Check the sign of the product
        BGED    DONE            ; If >0, multiplication complete (delayed)
        AND     0FFFFH,R1       ; Upper 16 bits of P2+P3
        ADDI3   R4,R2,R0        ; W0 = R0 = Lower word of the product
        ADDC3   R1,R3,R1        ; W1 = R1 = Upper word of the product
*
*   NEGATE THE PRODUCT IF THE NUMBERS ARE OF OPPOSITE SIGN
*
        NOT     R0
        ADDI    1,R0
        NOT     R1
        ADDC    0,R1
*
DONE    RETS
        .end
```

12.3.7 Floating-point Format Conversion: IEEE to/from TMS320C30

In fixed-point arithmetic, the binary point that separates the integer from the fractional part of the number is fixed at a certain location. For example, if it is chosen that a 32-bit number has the binary point after the most significant bit (which is also the sign bit), only fractional numbers (numbers with absolute values less that 1), can be represented. In this case, it is said that we have a Q31 number, where 31 is the number of fractional bits. All operations assume that the binary point is fixed at this location.

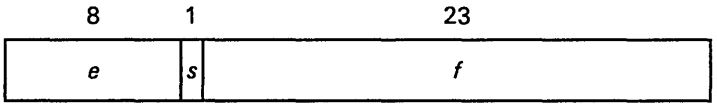
The fixed-point system, although simple to implement in hardware, imposes limitations in the dynamic range of the represented number, which causes scaling problems in many applications. The difficulty is avoided by using floating-point numbers. A floating-point number consists of a mantissa *m* multiplied by base *b* raised to an exponent *e*:

$$m * b^e$$

In current hardware implementations, the mantissa is typically a normalized number with absolute value between 1 and 2, and the base is b=2. Although the mantissa is represented as a fixed-point number, the actual value of the overall number floats the binary point because of the multiplication by b^e The exponent *e* is an integer whose value determines the position of the binary point in the number. IEEE has established a standard format for the representation of floating-point numbers.

In order to achieve higher efficiency in the hardware implementation, the TMS320C30 uses a floating-point format that differs from the IEEE standard. This section describes briefly the two formats and presents software routines to convert between them.

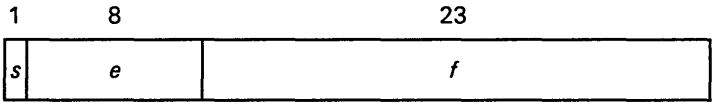
TMS320C30 floating-point format:



In a 32-bit word representing a floating-point number, the first 8 bits correspond to the exponent, expressed in two's-complement format. There is one bit for sign, and 23 bits for the mantissa. The mantissa is expressed in two's-complement form with the binary point after the most significant non-sign bit. Since this bit is the complement of the sign bit s, it is suppressed. In other words, the mantissa actually has 24 bits. One special case occurs when e=-128. In this case, the number is interpreted as zero independent of the values of s and f (which are by default set to zero). To summarize, the values of the represented numbers in the TMS320C30 floating-point format are:

2^e * (01.f)	if s=0
2^e * (10.f)	if s=1
0	if e=-128

IEEE floating-point format:



The IEEE floating-point format uses sign-magnitude notation for the mantissa, and offset by 127 for the exponent. In a 32-bit word representing a floating-point number, the first bit is the sign bit. The next 8 bits correspond to the exponent, expressed in an offset-by-127 format (the actual exponent is e-127). The following 23 bits represent the absolute value of the mantissa with the most significant 1 implied. The binary point is after this most significant 1. In other words, the mantissa actually has 24 bits. There are several special cases, summarized below.

The values of the represented numbers in the IEEE floating-point format are:

$(-1)^s * 2^{e-127} * (01.f)$	if 0<e<255

Special cases:

$(-1)^s * 0.0$	if e=0 and f=0 (zero)
$(-1)^s * 2^{-126} * (0.f)$	if e=0 and f<>0 (denormalized)
$(-1)^s *$ infinity	if e=255 and f=0 (infinity)
NaN	if e=255 and f<>0 (Not a Number)

Based on these definitions of the formats, two versions of the conversion routines were developed. One version handles the complete definition of the formats. The other ignores some of the special cases (typically the ones that are very rarely used), but it has the benefit that it executes faster than the complete conversion. For this discussion, they are referred to as the complete version and the the fast version.

12.3.7.1 IEEE to TMS320C30 Floating-Point Format Conversion

The fast version of the IEEE-to-TMS320C30 conversion routine was originally developed by Keith Henry of Apollo Computer, Inc. The other routines were based on this initial input. Example 12-22 shows the fast conversion from IEEE to TMS320C30 floating-point format. It properly handles the general case when 0<e<255, and zeros (i.e., e=0 and f=0). The other special cases (denormalized, infinity, and NaN) are not treated and, if present, will give erroneous results.

Example 12-22. TMS320C30 To IEEE Conversion (Fast Version)

```
*  TITL  IEEE TO TMS320C30 CONVERSION (FAST VERSION)
*
*
*  SUBROUTINE   FMIEEE
*
*  FUNCTION: CONVERSION BETWEEN THE IEEE FORMAT AND THE
*            320C30 FLOATING POINT NUMBERS.  THE NUMBER TO
*            BE CONVERTED IS IN THE LOWER 32 BITS OF R0.
*            THE RESULT IS STORED IN THE UPPER 32 BITS OF R0.
*            UPON ENTERING THE ROUTINE, AR1 POINTS TO THE
*            FOLLOWING TABLE:
*
*       (0)     0xFF800000 <-- AR1
*       (1)     0xFF000000
*       (2)     0x7F000000
*       (3)     0x80000000
*       (4)     0x81000000
*
*  ARGUMENT ASSIGNMENTS:
*    ARGUMENT | FUNCTION
*    ---------+-----------------------
*    R0       | NUMBER TO BE CONVERTED
*    AR1      | POINTER TO TABLE WITH CONSTANTS
*
*  REGISTERS USED AS INPUT: R0, AR1
*  REGISTERS MODIFIED: R0, R1
*  REGISTER CONTAINING RESULT: R0
*
*  NOTE: SINCE THE STACK POINTER SP IS USED, MAKE SURE TO
*        INITIALIZE IT IN THE CALLING PROGRAM.
*
*
*  CYCLES: 12 (WORST CASE) WORDS: 12
*
       .global FMIEEE
*
FMIEEE AND3    R0,*AR1,R1    ; Replace fraction with 0
       BND     NEG           ; Test sign
       ADDI    R0,R1         ; Shift sign and exponent inserting 0
       LDIZ    *+AR1(1),R1   ; If all zero, generate C30 zero
       SUBI    *+AR1(2),R1   ; Unbias exponent
       PUSH    R1
       POPF    R0            ; Load this as a flt. pt. number
       RETS
*
NEG    PUSH    R1
       POPF    R0            ; Load this as a flt. pt. number
       NEGF    R0,R0         ; Negate if original sign negative
       RETS
```


Example 12-23 is the complete conversion between IEEE and TMS320C30 formats. In addition to the general case and the zeros, it handles the special cases as follows:

- If NaN (e=255, f<>0), the number is returned intact.
- If infinity (e=255, f=0); the output is saturated to the most positive or negative number respectively.

- If denormalized (e=0, f<>0), two cases are considered. If the MSB of f is 1, the number is converted to TMS320C30 format. Otherwise, an underflow occurs and the number is set to zero.

Example 12-23. IEEE to TMS320C30 conversion (complete version)

```
*
*  TITL  IEEE TO TMS320C30 CONVERSION (COMPLETE VERSION)
*
*
*  SUBROUTINE   FMIEEE1
*
*  FUNCTION: CONVERSION BETWEEN THE IEEE FORMAT AND THE 320C30
*            FLOATING POINT NUMBERS.  THE NUMBER TO BE CONVERTED
*            IS IN THE LOWER 32 BITS OF R0.  THE RESULT IS STORED
*            IN THE UPPER 32 BITS OF R0.
*
*
*  UPON ENTERING THE ROUTINE, AR1 POINTS TO THE FOLLOWING TABLE:
*
*       (0)     0xFF800000 <-- AR1
*       (1)     0xFF000000
*       (2)     0x7F000000
*       (3)     0x80000000
*       (4)     0x81000000
*       (5)     0x7F800000
*       (6)     0x00400000
*       (7)     0x007FFFFF
*       (8)     0x7F7FFFFF
*
*  ARGUMENT ASSIGNMENTS:
*    ARGUMENT | FUNCTION
*    ---------+-----------------------
*    R0       | NUMBER TO BE CONVERTED
*    AR1      | POINTER TO TABLE WITH CONSTANTS
*
*  REGISTERS USED AS INPUT: R0, AR1
*  REGISTERS MODIFIED: R0, R1
*  REGISTER CONTAINING RESULT: R0
*
*  NOTE: SINCE THE STACK POINTER SP IS USED, MAKE SURE TO INITIALIZE
*        IT IN THE CALLING PROGRAM.
*
*
*  CYCLES: 23 (WORST CASE)   WORDS: 34
*
       .global FMIEEE1
*
FMIEEE1 LDI     R0,R1
        AND     *+AR1(5),R1
        BZ      UNNORM          ; If e=0, number is either 0 or
*                               ;   unnormalized
        XOR     *+AR1(5),R1
        BNZ     NORMAL          ; If e<255, use regular routine

*  HANDLE NaN AND INFINITY

        TSTB    *+AR1(7),R0
        RETSNZ                  ; Return if NaN
        LDI     R0,R0
        LDFGT   *+AR1(8),R0     ; If positive, infinity=
        LDFN                    ;   most positive number
        *+AR1(5),R0             ; If negative, infinity=
        RETS                    ;   most negative number RETS
```

```
*  HANDLE ZEROS AND UNNORMALIZED NUMBERS

UNNORM  TSTB    *+AR1(6),R0   ;  Is the msb of f equal to 1?
        LDFZ    *+AR1(3),R0   ;  If not, force the number to zero
        RETSZ                 ;        and return

        XOR     *+AR1(6),R0   ;  If (msb of f)=1, make it 0
        BND     NEG1
        LSH     1,R0          ;  Eliminate sign bit and line up mantissa
        SUBI    *+AR1(2),R0   ;  Make e=-127
        PUSH    R0
        POPF    R0            ;  Put number in floating point format
        RETS
NEG1    POPF    R0
        NEGF    R0,R0         ;  If negative, negate R0
        RETS

*  HANDLE THE REGULAR CASES
*
NORMAL  AND3    R0,*AR1,R1    ;  Replace fraction with 0
        BND     NEG           ;  Test sign
        ADDI    R0,R1         ;  Shift sign and exponent inserting 0
        SUBI    *+AR1(2),R1   ;  Unbias exponent
        PUSH    R1
        POPF    R0            ;  Load this as a flt. pt. number
        RETS

NEG     POPF    R0            ;  Load this as a flt. pt. number
        NEGF    R0,R0         ;  Negate if original sign negative
        RETS
```

12.3.7.2 TMS320C30 to IEEE Floating-Point Format conversion

The vast majority of the numbers represented by the TMS320C30 format are covered by the general IEEE format and the representation of zeros. The only special case to consider is when e=-127 in the TMS320C30 format. This corresponds to an denormalized number in IEEE format, and it is ignored in the fast version, while it is treated properly in the complete version. Example 12-24 shows the fast, and Example 12-25, the complete version of the TMS320C30-to-IEEE conversion.

Example 12-24. TMS320C30 to IEEE Conversion (Fast Version)

```
*
*  TITL  TMS320C30 TO IEEE CONVERSION (FAST VERSION)
*
*
*  SUBROUTINE TOIEEE
*
*  FUNCTION: CONVERSION BETWEEN THE 320C30 FORMAT AND THE IEEE
*            FLOATING POINT NUMBERS.  THE NUMBER TO BE CONVERTED
*            IS IN THE UPPER 32 BITS OF R0.  THE RESULT WILL BE IN
*            THE LOWER 32 BITS OF R0.
*
*       UPON ENTERING THE ROUTINE, AR1 POINTS TO THE FOLLOWING TABLE:
*
*       (0)     0xFF800000 <-- AR1
*       (1)     0xFF000000
*       (2)     0x7F000000
*       (3)     0x80000000
*       (4)     0x81000000
*
*  ARGUMENT ASSIGNMENTS:
*    ARGUMENT | FUNCTION
*    ---------+----------------------
*    R0       | NUMBER TO BE CONVERTED
*    AR1      | POINTER TO TABLE WITH CONSTANTS
*
*  REGISTERS USED AS INPUT: R0, AR1
*  REGISTERS MODIFIED: R0
*  REGISTER CONTAINING RESULT: R0
*
*  NOTE: SINCE THE STACK POINTER 'SP' IS USED, MAKE SURE TO
*        INITIALIZE IT IN THE CALLING PROGRAM.
*
*
*   CYCLES: 14 (WORST CASE)  WORDS: 15
*
        .global TOIEEE
*
TOIEEE  LDF     R0,R0           ; Determine the sign of the number
        LDFZ    *+AR1(4),R0     ; If zero, load appropriate number
        BND     NEG             ; Branch to NEG if negative (delayed)
        ABSF    R0              ; Take the absolute value of the number
        LSH     1,R0            ; Eliminate the sign bit in R0
        PUSHF   R0
        POP     R0              ; Place number in lower 32 bits of R0
        ADDI    *+AR1(2),R0     ; Add exponent bias (127)
        LSH     -1,R0           ; Add the positive sign
        RETS

NEG     POP     R0              ; Place number in lower 32 bits of R0
        ADDI    *+AR1(2),R0     ; Add exponent bias (127)
        LSH     -1,R0           ; Make space for the sign
        ADDI    *+AR1(3),R0     ; Add the negative sign
        RETS
```

Example 12-25. TMS320C30 to IEEE Conversion (Complete Version)

```
*
*  TITL  TMS320C30 TO IEEE CONVERSION (COMPLETE VERSION)
*
*
*  SUBROUTINE   TOIEEE1
*
*
*  FUNCTION:  CONVERSION BETWEEN THE 320C30 FORMAT AND THE IEEE
*             FLOATING POINT NUMBERS.  THE NUMBER TO BE CONVERTED
*             IS IN THE UPPER 32 BITS OF R0.  THE RESULT WILL BE
*             IN THE LOWER 32 BITS OF R0.
*
*
*       UPON ENTERING THE ROUTINE, AR1 POINTS TO THE FOLLOWING TABLE:
*       (0)     0xFF800000 <-- AR1
*       (1)     0xFF000000
*       (2)     0x7F000000
*       (3)     0x80000000
*       (4)     0x81000000
*       (5)     0x7F800000
*       (6)     0x00400000
*       (7)     0x007FFFFF
*       (8)     0x7F7FFFFF
*
*  ARGUMENT ASSIGNMENTS:
*    ARGUMENT | FUNCTION
*    ---------+-----------------------
*    R0       | NUMBER TO BE CONVERTED
*    AR1      | POINTER TO TABLE WITH CONSTANTS
*
*  REGISTERS USED AS INPUT: R0, AR1
*  REGISTERS MODIFIED: R0
*  REGISTER CONTAINING RESULT: R0
*
*  NOTE: SINCE THE STACK POINTER 'SP' IS USED, MAKE SURE TO
*        INITIALIZE IT IN THE CALLING PROGRAM.
*
*
*       CYCLES: 31 (WORST CASE)       WORDS: 25
*
        .global   TOIEEE1
*
TOIEEE1 LDF     R0,R0          ; Determine the sign of the number
        LDFZ    *+AR1(4),R0    ; If zero, load appropriate number
        BND     NEG            ; Branch to NEG if negative (delayed)
        ABSF    R0             ; Take the absolute value of the number
        LSH     1,R0           ; Eliminate the sign bit in R0
        PUSHF   R0
        POP     R0             ; Place number in lower 32 bits of R0
        ADDI    *+AR1(2),R0    ; Add exponent bias (127)
        LSH     -1,R0          ; Add the positive sign
```

```
CONT    TSTB    *+AR1(5),R0
        RETSNZ                  ; If E>0, return
        TSTB    *+AR1(7),R0
        RETSZ                   ; If E=0 & F=0, return
        PUSH    R0
        POPF    R0
        LSH     -1,R0           ; Move F right by one bit
        PUSHF   R0
        POP     R0
        ADDI    *+AR1(6),R0     ; Add to F a msb of 1
        RETS

NEG     POP     R0              ; Place number in lower 32 bits of R0
        BRD     CONT
        ADDI    *+ARI(2),R0     ; Add exponent bias (127)
        LSH     -1,R0           ; Make space for the sign
        ADDI    *+AR1(3),R0     ; Add the negative sign
```

12.4 Application-Oriented Operations

The TMS320C30 has been designed to provide efficient implementations of digital signal processing algorithms. The architecture and the instruction set of the device include features that facilitate the solution of numerically intensive problems. This section presents examples of applications using these features, such as companding, filtering, Fast Fourier Transforms (FFT), and matrix arithmetic.

12.4.1 Companding

In the area of telecommunications, one of the primary concerns is the conservation of the channel bandwidth, while at the same time preserving high speech quality. This is achieved by quantizing the speech samples logarithmically. It has been demonstrated that an 8-bit logarithmic quantizer produces speech quality equivalent to a 13-bit uniform quanitizer. The logarithmic quantization is achieved by companding (COMpress/exPANDing). Two international standards have been established for companding: the μ-law (used in the United States and Japan), and the A-law (used in Europe). Detailed descriptions of μ-law and A-law companding are presented in an application report on companding routines included in the book "Digital Signal Processing Applications with the TMS320 Family".

During transmission, logarithmically compressed data in sign-magnitude form are transmitted along the communications channel. If any processing is necessary, these data should be expanded to a 14-bit (for μ-law) or 13-bit (for A-law) linear format. This operation is done upon receiving the data at the digital signal processor. After processing, and in order to continue transmission, the result is compressed back to 8-bit format and transmitted through the channel.

Examples 12-26 and 12-27 show μ-law compression and expansion (i.e., linear to μ-law and μ-law to linear conversion), while examples 12-28 and 12-29 show A-law compression and expansion. For expansion, using a look-up table offers an alternative approach. It trades memory space for speed of execution. Since the compressed data is 8-bits long, a table with 256 entries can be constructed containing the expanded data. If the compressed data is stored in the register AR0, the following two instructions will put the expanded data in register R0:

```
ADDI    @TABL,AR0     ; @TABL = BASE ADDRESS OF TABLE
LDI     *AR0,R0       ; PUT EXPANDED NUMBER IN R0
```

The same look-up table approach could be used for compression, but the required table length would then be 16,384 words for μ-law or 8,192 words for A-law. If this memory size is not acceptable, the subroutines presented in Examples 12-26 or 12-28. should be used.

Example 12-26. U-Law compression

```
*
*  TITL  U-LAW COMPRESSION
*
*
*  SUBROUTINE  MUCMPR
*
*
*  ARGUMENT ASSIGNMENTS:
*    ARGUMENT | FUNCTION
*    ---------+-----------------------
*    R0       | NUMBER TO BE CONVERTED
*
*  REGISTERS USED AS INPUT: R0
*  REGISTERS MODIFIED: R0, R1, R2, SP
*  REGISTER CONTAINING RESULT: R0
*
*  NOTE: SINCE THE STACK POINTER 'SP' IS USED IN THE COMPRESSION
*        ROUTINE 'MUCMPR', MAKE SURE TO INITIALIZE IT IN THE
*        THE CALLING PROGRAM.
*
*
*  CYCLES: 20   WORDS: 17
*
*
       .global MUCMPR
*
MUCMPR  LDI     R0,R1          ; Save sign of number
        ABSI    R0,R0
        CMPI    1FDEH,R0       ; If R0>0x1FDE,
        LDIGT   1FDEH,R0       ; saturate the result
        ADDI    33,R0          ; Add bias

        FLOAT   R0             ; Normalize: (seg+5)0WXYZx...x
        MPYF    0.03125,R0     ; Adjust segment number by 2**(-5)
        LSH     1,R0           ; (seg)WXYZx...x
        PUSHF   R0
        POP     R0             ; Treat number as integer
        LSH     -20,R0         ; Right-justify

        LDI     0,R2
        LDI     R1,R1          ; If number is negative,
        LDILT   80H,R2         ;   set sign bit
        ADDI    R2,R0          ; R0 = compressed number
        NOT     R0             ; Reverse all bits for transmission
        RETS
```

Example 12-27. U-Law Expansion

```
*
*  TITL  'U-LAW EXPANSION'
*
*
*  SUBROUTINE   MUXPND
*
*
*  ARGUMENT ASSIGNMENTS:
*
*  ARGUMENT | FUNCTION
*  ---------+-----------------------
*  R0       | NUMBER TO BE CONVERTED
*
*  REGISTERS USED AS INPUT: R0
*  REGISTERS MODIFIED: R0, R1, R2, SP
*  REGISTER CONTAINING RESULT: R0
*
*
*  CYCLES: 20 (WORST CASE)      WORDS: 14
*
*
        .global MUXPND
MUXPND  NOT     R0,R0          ;  Complement bits
        LDI     R0,R0
        AND     0FH,R1         ;  Isolate quantization bin
        LSH     1,R1
        ADDI    33,R1          ;  Add bias to introduce 1xxxx1
        LDI     R0,R2          ;  Store for sign bit
        LSH     -4,R0
        AND     7,R0           ;  Isolate segment code
        LSH3    R0,R1,R0       ;  Shift and put result in R0
        SUBI    33,R0          ;  Subtract bias
        TSTB    80H,R2         ;  Test sign bit
        RETSZ
        NEGI    R0             ;  Negate if a negative number
        RETS
```

Example 12-28. A-Law Compression

```
*
*  TITL  A-LAW COMPRESSION
*
*
*  SUBROUTINE   ACMPR
*
*
*  ARGUMENT ASSIGNMENTS:
*    ARGUMENT | FUNCTION
*    ---------+-----------------------
*    R0       | NUMBER TO BE CONVERTED
*
*  REGISTERS USED AS INPUT: R0
*  REGISTERS MODIFIED: R0, R1, R2, SP
*  REGISTER CONTAINING RESULT: R0
*
*  NOTE: SINCE THE STACK POINTER 'SP' IS USED IN THE COMPRESSION
*        ROUTINE 'ACMPR', MAKE SURE TO INITIALIZE IT IN THE
*        CALLING PROGRAM.
*
*
*  CYCLES:22    WORDS: 19
*
        .globl  ACMPR
*
ACMPR   LDI     R0,R1           ; Save sign of number
        ABSI    R0,R0
        CMPI    1FH,R0          ; If R0<0x20,
        BLED    END             ;   Do linear coding
        CMPI    0FFFH,R0        ; If R0>0xFFF,
        LDIGT   0FFFH,R0        ;   saturate the result
        LSH     -1,R0           ; Eliminate rightmost bit

        FLOAT   R0              ; Normalize: (seg+3)0WXYZx...x
        MPYF    0.125,R0        ; Adjust segment number by 2**(-3)
        LSH     1,R0            ; (seg)WXYZx...x
        PUSHF   R0
        POP     R0              ; Treat number as integer
        LSH     -20,R0          ; Right-justify

END     LDI     0,R2
        LDI     R1,R1           ; If number is negative,
        LDILT   80H,R2          ;   set sign bit
        ADDI    R2,R0           ; R0 = compressed number
        XOR     0D5H,R0         ; Invert even bits for transmission
        RETS
*
```

Example 12-29. A-Law Expansion

```
*
*  TITL  A-LAW EXPANSION
*
*
*
*  SUBROUTINE   AXPND
*
*
*  ARGUMENT ASSIGNMENTS:
*    ARGUMENT | FUNCTION
*    ---------+-----------------------
*    R0       | NUMBER TO BE CONVERTED
*
*  REGISTERS USED AS INPUT: R0
*  REGISTERS MODIFIED: R0, R1, R2, SP
*  REGISTER CONTAINING RESULT: R0
*
*
*      CYCLES: 25 (WORST CASE)         WORDS: 16
*
*
       .global AXPND
*
AXPND   XOR     D5H,R0        ; Invert even bits
        LDI     R0,R1
        AND     0FH,R1        ; Isolate quantization bin
        LSH     1,R1
        LDI     R0,R2         ; Store for bit sign
        LSH     -4,R0
        AND     7,R0          ; Isolate segment code
        BZ      SKIP1
        SUBI    1,R0
        ADDI    32,R1         ; Create 1xxxx1
SKIP1   ADDI    1,R1          ;   OR 0xxxx1
        LSH3    R0,R1,R0      ; Shift and put result in R0
        TSTB    80H,R2        ; Test sign bit
        RETSZ
        NEGI    R0            ; Negate if a negative number
        RETS
```

12.4.2 FIR, IIR, and Adaptive Filters

Digital filters are a common requirement for digital signal processing systems. There are two types of digital filters, Finite Impulse Response (FIR) and Infinite Impulse Response (IIR). Each of these types can have either fixed or adaptable coefficients. In this section, the fixed-coefficient filters are presented first, and then the adaptive filters are discussed.

12.4.2.1 FIR Filters

If the FIR filter has an impulse response $h[0]$, $h[1]$,..., $h[N-1]$, and $x[n]$ represents the input of the filter at time n, the output $y[n]$ at time n is given by the equation:

$$y[n] = h[0]\ x[n] + h[1]\ x[n-1] + ... + h[N-1]\ x[n-(N-1)]$$

Two features of the TMS320C30 that facilitate the implementation of the FIR filters, are parallel multiply/add operations and the circular addressing. The first one permits the performance of a multiplication and an addition in a single machine cycle, while the second one makes a finite buffer of length N sufficient for the data x.

Figure 12-1 shows the arrangement of the memory locations in order to implement the circular addressing, while Example 12-30 presents the TMS320C30 assembly code for an FIR filter.

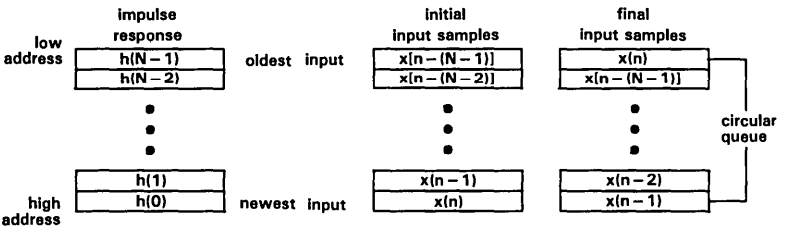


Figure 12-1. Data Memory Organization For a FIR Filter

In order to set up the circular addressing, the block-size register BK should be initialized to block length N. Also, the locations for signal x should start from a memory location whose address is a multiple of the smallest power of 2 that is greater than or equal to N. For instance, if N=24, the first address for x should be a multiple of 32 (the lower 5 bits of the beginning address should be zero). To understand this requirement, look at the section describing circular addressing.

In Example 12-30, the pointer to the input sequence x, is incremented and assumed to be moving from an older input to a newer input. At the end of the subroutine, AR1 will be pointing to the position for the next input sample.

Example 12-30. FIR Filter

```
*
*  TITL  FIR FILTER
*
*
*  SUBROUTINE  F I R
*
*  EQUATION: y(n) = h(0) * x(n) + h(1) * x(n-1) +
*                   ... + h(N-1) * x(n-(N-1))
*
*  TYPICAL CALLING SEQUENCE:
*
*       LOAD    AR0
*       LOAD    AR1
*       LOAD    RC
*       LOAD    BK
*       CALL    FIR
*
*
*  ARGUMENT ASSIGNMENTS:
*    ARGUMENT | FUNCTION
*    ---------+-----------------------
*    AR0      | ADDRESS OF h(N-1)
*    AR1      | ADDRESS OF x(N-1)
*    RC       | LENGTH OF FILTER - 2 (N-2)
*    BK       | LENGTH OF FILTER (N)
*
*  REGISTERS USED AS INPUT: AR0, AR1, RC, BK
*  REGISTERS MODIFIED: R0, R2, AR0, AR1, RC
*  REGISTER CONTAINING RESULT: R0
*
*
*  CYCLES: 11 + (N-1)          WORDS:  6
*
*
        .global FIR
*                           ;  Initialize R0:
FIR     MPYF3   *AR0++(1),*AR1++(1)%,R0
*                           ;  h(N-1) * x(n-(N-1)) -> R0
        LDF     0.0,R2      ;  Initialize R2.
*
*  FILTER ( 1 <= i < N)
*
        RPTS    RC          ;  Setup the repeat cycleE.
        MPYF3   *AR0++(1),*AR1++(1)%,R0  ;  h(N-1-i)*x(n-(N-1-i))->R0
||      ADDF3   R0,R2,R2    ;  Multiply and add operation
*
        ADDF    R0,R2,R0    ;  Add last product
*
*  RETURN SEQUENCE
*
        RETS                ;  Return
*
*  end
*
  .end
```

12.4.2.2 IIR Filters

The transfer function of the IIR filters has both poles and zeros. Its output depends on both the input and the past output. As a rule, they need less computation than an FIR with similar frequency response, but they have the drawback of being sensitive to coefficient quantization. Most often, the IIR filters are implemented as a cascade of second-order sections, called biquads. Examples 12-31 and 12-32 show the implementation for one biquad and for any number of biquads respectively.

The equation for a single biquad is given by:

$$y[n] = a1\ y[n-1] + a2\ y[n-2] + b0\ x[n] + b1\ x[n-1] + b2\ x[n-2]$$

This equation can be implemented more conveniently by the following two equations which have less storage requirements:

$$d[n] = a2\ d[n-2] + a1\ d[n-1] + x[n]$$
$$y[n] = b2\ d[n-2] + b1\ d[n-1] + b0\ d[n]$$

Figure 12-2 shows the memory organization for this approach, and Example 12-31 is an implementation of a single biquad on the TMS320C30.

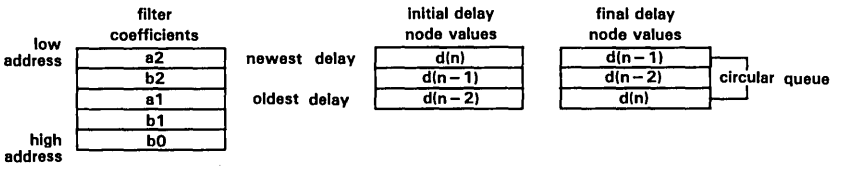


Figure 12-2. Data Memory Organization For a Single Biquad

As in the case of FIR filters, the address for the start of the values d must be a multiple of 4, i.e., the last two bits of the beginning address must be zero. The block-size register BK must be initialized to 3.

12

Example 12-31. IIR Filter (One Biquad)

```
*
*  TITL  IIR filter
*
*
*  SUBROUTINE  I I R 1
*
*  IIR1 == IIR FILTER (ONE BIQUAD)
*
*
*  EQUATIONS: d(n) = a2 * d(n-2) + a1 * d(n-1) + x(n)
*             y(n) = b2 * d(n-2) + b1 * d(n-1) + b0 * d(n)
*
*  OR         y(n) = a1*y(n-1) + a2*y(n-2) + b0*x(n)
*                      + b1*x(n-1) + b2*x(n-2)
*
*  TYPICAL CALLING SEQUENCE:
*
*       load    R2
*       load    AR0
*       load    AR1
*       load    BK
*       CALL    IIR1
*
*
*  ARGUMENT ASSIGNMENTS:
*    ARGUMENT | FUNCTION
*    ---------+-----------------------
*    R2       | INPUT SAMPLE X(N)
*    AR0      | ADDRESS OF FILTER COEFFICIENTS (A2)
*    AR1      | ADDRESS OF DELAY MODE VALUES (D(N-2))
*    BK       | BK = 3
*
*  REGISTERS USED AS INPUT: R2, AR0, AR1, BK
*  REGISTERS MODIFIED: R0, R1, R2, AR0, AR1
*  REGISTER CONTAINING RESULT: R0
*
*  CYCLES:  11  WORDS:  8
*
*
*  FILTER
*
       .global IIR1
*
IIR1   MPYF3   *AR0,*AR1,R0
*                          ; a2 * d(n-2) -> R0
       MPYF3   *++AR0(1),*AR1--(1)%,R1
*                          ; b2 * d(n-2) -> R1
*
       MPYF3   *++AR0(1),*AR1,R0   ;  a1 * d(n-1) -> R0
||     ADDF3   R0,R2,R2       ; a2*d(n-2)+x(n) -> R2
*
       MPYF3   *++AR0(1),*AR1--(1)%,R0 ;  b1 * d(n-1) -> R0
||     ADDF3   R0,R2,R2       ; a1*d(n-1)+a2*d(n-2)+x(n) -> R2
```

```
*
        MPYF3   *++AR0(1),R2,R2 ;  b0 * d(n) -> R2
||      STF     R2,*AR1++(1)%
*                               ;  Store d(n) and point to d(n-1).
*
        ADDF    R0,R2           ;  b1*d(n-1)+b0*d(n) -> R2
        ADDF    R1,R2,R0        ;  b2*d(n-2)+b1*d(n-1)+b0*d(n) -> R0
*
*  RETURN SEQUENCE
*
        RETS                    ;  Return
*
*  end
*
   .end
```

In the more general case, the IIR filter will contain N>1 biquads. The equations for its implementation are given by the following pseudo-C language code:

```
y[0,n] = x[n]
for (i=0; i<N; i++){
     d[i,n] = a2[i] d[i, n-2] + a1[i] d[ i,n-1] + y[ i-1,n]
     y[i,n] = b2[i] d[i,n-2]+ b1[i] d[i,n-1]+ b0[i] d[i,n]
}
y[n] = y[N-1, n]
```

The corresponding memory organization is shown in Figure 12-3, while Example 12-32 is the TMS320C30 assembly-language code.

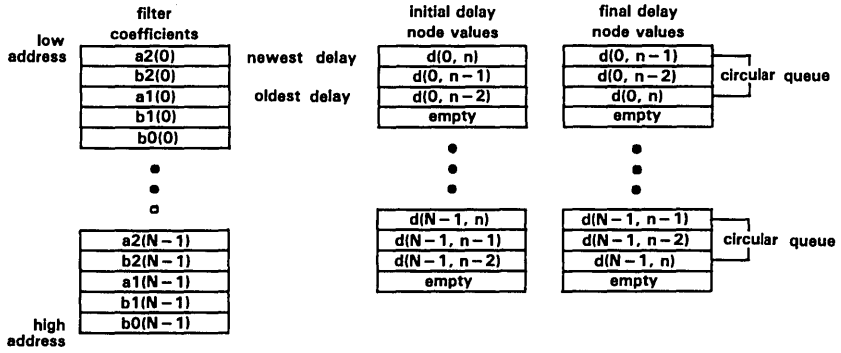


Figure 12-3. Data Memory Organization For N Biquads

The block register BK should be initialized to 3, and the beginning of each set of *d* values (i.e., *d*[*i*,*n*], *i*=0...N-1) should be at an address that is a multiple of 4 (the last two bits zero), as stated in the case of a single biquad.

Example 12-32. IIR Filters (N > 1 Biquads)

```
*
*  TITL  IIR FILTERS (N > 1 BIQUADS)
*
*
*  SUBROUTINE IIR2
*
*
*
*  EQUATIONS:   y(0,n) = x(n)
*
*  FOR (i = 0; i < N; i++)
*      {
*      d(i,n) = a2(i) * d(i,n-2) + a1(i) * d(i,n-1) + y(i-1,n)
*      y(i,n) = b2(i) * d(i,n-2) + b1(i) * d(i,n-1) + b0(i) * d(i,n)
*      }
*  y(n) = y(N-1,n)
*
*  TYPICAL CALLING SEQUENCE:
*
*        load    R2
*        load    AR0
*        load    AR1
*        load    IR0
*        load    IR1
*        load    BK
*        load    RC
*        CALL    IIR2
*
*
*  ARGUMENT ASSIGNMENTS:
*    ARGUMENT | FUNCTION
*    ---------+-----------------------
*    R2       | INPUT SAMPLE x(n)
*    AR0      | ADDRESS OF FILTER COEFFICIENTS (a2(0))
*    AR1      | ADDRESS OF DELAY NODE VALUES (d(0,n-2))
*    BK       | BK = 3
*    IR0      | IR0 = 4
*    IR1      | IR1 = 4*N-4
*    RC       | NUMBER OF BIQUADS (N) - 2
*
*  REGISTERS USED AS INPUT: R2, AR0, AR1, IR0, IR1, BK, RC
*  REGISTERS MODIFIED: R0, R1, R2, AR0, AR1, RC
*  REGISTER CONTAINING RESULT: R0
*
*  CYCLES:  23 + 6(N-1)      WORDS:  17
*
*
*
*
        .global IIR2
*
IIR2    MPYF3   *AR0, *AR1, R0
*                           ; a2(0) * d(0,n-2) -> R0
        MPYF3   *++AR0(1), *AR1--(1)%, R1
*                           ; b2(0) * d(0,n-2) -> R1
```

```
*
        MPYF3   *++AR0(1),*AR1,R0 ;  a1(0) * D(0,n-1) -> R0
||       ADDF3  R0, R2, R2    ;  First sum term of d(0,n).
*
        MPYF3   *++AR0(1),*AR1--(1)%,R0 ;b1(0) * d(0,n-1) -> R0
||      ADDF3   R0, R2, R2    ;  Second sum term of d(0,n).
        MPYF3   *++AR0(1),R2,R2 ;b0(0) * d(0,n) -> R2
||      STF     R2, *AR1--(1)%
*                             ;  Store d(0,n); Point to d(0,n-2)
*
        RPTB    LOOP             ; Loop for 1 <= i < n
*
        MPYF3   *++AR0(1),*++AR1(IR0),R0 ;a2(i) * d(i,n-2) -> R0
||      ADDF3   R0,R2,R2      ;  First sum term of y(i-1,n).
*
        MPYF3   *++AR0(1),*AR1--(1)%,R1 ;b2(i) * D(i,n-2) -> R1
||      ADDF3   R1,R2,R2      ;  Second sum term of y(i-1,n).
*
        MPYF3   *++AR0(1),*AR1,R0 ;a1(i) * d(i,n-1) -> R0
||      ADDF3   R0, R2, R2    ;  First sum of d(i,n).
*
        MPYF3   *++AR0(1),*AR1--(1)%,R0 ;b1(i) * d(i,n-1) -> R0
||      ADDF3   R0, R2, R2    ;  Second sum term of d(i,n).
*
        STF     R2, *AR1--(1)%
*                             ;  Store d(i,n); point to d(i,n-2)
LOOP    MPYF3   *++AR0(1), R2, R2
*                             ;  b0(i) * d(i,n) -> R2
*
*  FINAL SUMMATION
*
        ADDF    R0,R2         ;  First sum term of y(n-1,n)
        ADDF3   R1,R2,R0      ;  Second sum term of y(n-1,n)
*
        NOP     *AR1--(IR1)   ;  Return to first biquad
        NOP     *AR1--(1)%    ;  Point to d(0,n-1)
*
*  RETURN SEQUENCE
*
        RETS                  ;  Return
*
*  end
*
        .end
```

12.4.2.3 Adaptive Filters (LMS Algorithm)

There are applications in digital signal processing where a filter must be adapted over time to keep track of changing conditions. The book "Theory and Design of Adaptive Filters" by Treichler, Johnson, and Larimore (Wiley-Interscience, 1987) presents the theory of adaptive filters. Although in theory both FIR and IIR structures can be used as adaptive filters, the stability problems and the local optimum points that the IIR filters exhibit, make them less attractive for such an application. Hence, until further research makes IIR filters a better choice, only the FIR filter are used in adaptive algorithms of practical applications.

In an adaptive FIR filter, the filtering equation takes the form:

$$y[n] = h[n,0]\ x[n] + h[n,1]\ x[n\text{-}1] + \ldots + h[n,N\text{-}1]\ \ x[n\text{-}(N\text{-}1)]$$

The filter coefficients are time-dependent. In a least-mean-squares (LMS) algorithm, the coefficients are updated by a formula of the form:

$$h[n+1,i] = h[n,i] + \beta \quad x[n-i], \quad i=0,1,...,N-1$$

β is a constant for the computation. The updating of the filter coefficients can be interleaved with the computation of the filter output so that it takes 3 cycles per filter tap to do both. The updated coefficients are written over the old filter coefficients. Example 12-33 shows the implementation of an adaptive FIR filter on the TMS320C30. The memory organization and the positioning of the data in memory should follow the same rules as the above FIR filter with fixed coefficients.

Example 12-33. Adaptive FIR Filter (LMS Algorithm)

```
*  TITL  ADAPTIVE FIR FILTER (LMS ALGORITHM)


*
*  SUBROUTINE L M S


*  LMS == LMS ADAPTIVE FILTER
*
*
*
*  EQUATIONS: y(n) = h(n,0)*x(n) + h(n,1)*x(n-1) + ...
*                  + h(n,N-1)*x(n-(N-1))
*            FOR (i = 0; i < N; i++)
*                  h(n+1,i) = h(n,i) + tmuerr * x(n-i)
*
*  TYPICAL CALLING SEQUENCE:
*
*       load    R4
*       load    AR0
*       load    AR1
*       load    RC
*       load    BK
*       CALL    FIR
*
*
*   ARGUMENT ASSIGNMENTS:
*     ARGUMENT | FUNCTION
*     ---------+----------------------
*    R4        | SCALE FACTOR (2 * mu * err)
*    AR0       | ADDRESS OF h(n,N-1)
*    AR1       | ADDRESS OF x(n-(N-1))
&    RC        | LENGTH OF FILTER - 2 (N-2)
&    BK        | LENGTH OF FILTER  (N)
*
*   REGISTERS USED AS INPUT: R4, AR0, AR1, RC, BK
*   REGISTERS MODIFIED: R0, R1, R2, AR0, AR1, RC
*   REGISTER CONTAINING RESULT: R0
*
*   PROGRAM SIZE: 10 words
*
*   EXECUTION CYCLES: 12 + 3(N-1)
*
*
*   SETUP (i = 0)
*
        .global LMS
*                           ;  Initialize R0:
LMS     MPYF3   *AR0, *AR1, R0
*                           ;  h(n,N-1) * x(n-(N-1)) -> R0
        LDF     0.0,R2      ;  Initialize R2.
*
*                           ;  Initialize R1:
        MPYF3   *AR1++(1)%, R4, R1
*                           ;  x(n-(N-1)) * tmuerr -> R1
        ADDF3    *AR0++(1), R1, R1
*                           ;  h(n,N-1) + x(n-(N-1)) *
*                                tmuerr -> R1
*
```

```
*   FILTER AND UPDATE ( 1 <= I < N)
*
        RPTB    LOOP            ; Setup the repeat block.
*
*                               ; Filter:
        MPYF3   *AR0--(1),AR1,R0  ;  h(n,N-1-i) * x(n-(N-1-i)) -> R0
||      ADDF3    R0,R2,R2       ; Multiply and add operation.
*
*                               ; UPDATE:
        MPYF3    *AR1++(1)%,R4,R1 ;x(n,N-(N-1-i)) * tmuerr -> R1
||      STF     R1,*AR0++(1)    ; R1 -> h(n+1,N-1-(i-1))
*
LOOP    ADDF3   *AR0++(1), R1, R1
*                               ; h(n,N-1-i) + x(n-(N-1-i))*tmuerr -> R1
*
        ADDF3   R0,R2,R0        ; Add last product.
        STF     R1,*-AR0(1)     ; h(n,0) + x(n) * tmuerr -> h(n+1,0)
*
*   RETURN SEQUENCE
*
        RETS                    ; Return
*
*   end
*
    .end
```

12.4.3 Matrix-Vector Multiplication

In matrix-vector multiplication, a K x N matrix of elements m(i,j) having K rows and N columns is multiplied by an N x 1 vector to produce a K x 1 result. The multiplier vector has elements v(j), and the product vector has elements p(i). Each one of the product-vector elements is computed by the expression:

$$p(i) = m(i,0)\ v(0) + m(i,1)\ v(1) + \ldots + m(i,N\text{-}1)\ v(N\text{-}1) \quad i = 0,1,\ldots,K\text{-}1$$

This is essentially a dot product, and the matrix-vector multiplication contains as a special case the dot product presented in Example 12-2. In pseudo-C format, the computation of the matrix multiplication is expressed by:

```
for (i=0; i<K; i++) {
     p(i) = 0
     for (j=0; j<N;j++)
          p(i) = p(i) + m(i,j)* v(j)
}
```

Figure 12-4 shows the data memory organization for matrix-vector multiplication, and Example 12-34 is the TMS320C30 assembly code to implement it. Note that in Example 12-34, K (number of rows) should be greater than 0 and N (number of columns) should be greater than 1.

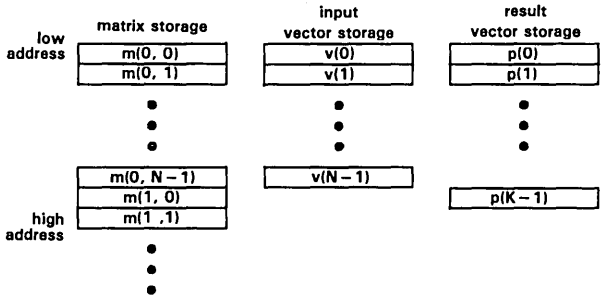


Figure 12-4. Data Memory Organization for Matrix-Vector Multiplication

Example 12-34. Matrix times a vector multiplication

```
*
*   TITL  MATRIX TIMES A VECTOR MULTIPLICATION
*
*
*   SUBROUTINE  M A T
*
*   MAT == MATRIX TIMES A VECTOR OPERATION
*
*
*   TYPICAL CALLING SEQUENCE:
*
*        load    AR0
*        load    AR1
*        load    AR2
*        load    AR3
*        load    R1
*        CALL    MAT
*
*
*   ARGUMENT ASSIGNMENTS:
*     argument | FUNCTION
*     ---------+-----------------------
*     AR0      | ADDRESS OF M(0,0)
*     AR1      | ADDRESS OF V(0)
*     AR2      | ADDRESS OF P(0)
*     AR3      | NUMBER OF ROWS - 1 (K-1)
*     R1       | NUMBER OF COLUMNS - 2 (N-2)
*
*   REGISTERS USED AS INPUT: AR0, AR1, AR2, AR3, R1
*   REGISTERS MODIFIED: R0, R2, AR0, AR1, AR2, AR3, IR0,
*                       RC, RSA, REA
*
*
*   PROGRAM SIZE: 11
*
*   EXECUTION CYCLES: 6 + 10 * K + K * (N - 1)
*
*
*
        .global MAT
*
*   SETUP
*
MAT     LDI     R1,IR0          ;   number of columns-2 -> IR0
        ADDI    2,IR0           ;  IR0 = N
*
*   FOR (i = 0; i < K; i++) LOOP OVER THE ROWS.
*
ROWS    LDF     0.0,R2          ;  initialize R2
        MPYF3   *AR0++(1),*AR1++(1),R0
*                               ;  m(i,0) * v(0) -> R0
*
*   FOR (j = 1; j < N; j++) DO DOT PRODUCT OVER COLUMNS
*
        RPTS    R1              ;  multiply a row by a column.
*
        MPYF3   *AR0++(1),*AR1++(1),R0 ;m(i,j) * v(j) -> R0
||      ADDF3   R0,R2,R2        ;  m(i,j-1) * v(j-1) + R2 -> R2.
*
```

```
        DBD     AR3,ROWS        ; counts the number of rows left.
*
*
        ADDF    R0,R2           ; last accumulate.
        STF     R2,*AR2++(1)    ; result -> p(i)
        NOP     *--AR1(IR0)     ; set AR1 to point to v(0).
*   !!!   DELAYED BRANCH HAPPENS HERE !!!
*
*   RETURN SEQUENCE
*
        RETS                    ; return
*
*   end
*
        .end
```

12.4.4 Fast Fourier Transforms (FFT)

Fourier transforms are an important tool often used in digital signal processing systems. The purpose of the transform is to convert information from the time domain to the frequency domain. The inverse Fourier transform converts information back to the time domain from the frequency domain. Implementation of Fourier transforms that are computationally efficient are known as Fast Fourier Transforms (FFTs). The theory of FFTs can by found in books such as "DFT/FFT and Convolution Algorithms" by C.S. Burrus and T.W. Parks (John Wiley, 1985), and in the book "Digital Signal Processing Applications with the TMS320 Family".

The TMS320C30 has many features that permit very efficient implementation of numerically intensive algorithms. Some of these features are particularly well suited for FFTs. The high speed of the device (60 ns cycle-time) makes easier the implementation of real-time algorithms, while the floating-point capability eliminates the problems associated with dynamic range. The powerful indexing scheme in indirect addressing facilitates the access of FFT butterfly legs that have different spans. A construct that reduces the looping overhead in algorithms heavily dependent on loops (such as the FFTs), is the repeat block implemented by the RPTB instruction. This construct gives the efficiency of in-line coding, but has the form of a loop. Since the output of the FFT is in scrambled (bit-reversed) order when the input is in regular order, there is a need to restore it in the proper order. In the TMS320C30, there is no need to spend extra cycles for this rearrangement. The device has a special form of indirect addressing (bit-reversed addressing mode), that can be used when the FFT output is needed. This mode permits accessing the FFT output in the proper order.

Fast Fourier Transform is a label for a collection of algorithms implementing efficient conversion from time to frequency domain. Different types of FFT are:

- Radix-2 and radix-4 algorithms(depending on the size of the FFT butterfly)
- Decimation in time or frequency (DIT or DIF)
- Complex or real FFTs
- FFTs of different lengths, etc.

The present implementation of the FFT was based on programs contained in the book "DFT/FFT and Convolution Algorithms" by C.S. Burrus and T.W. Parks, and in the paper "Real-Valued Fast Fourier Transform Algorithms" by H.V. Sorensen et al. (IEEE Trans. on ASSP, June 1987).

Examples 12-35 and 12-36 show the implementation of a complex radix-2, DIF, FFT on the TMS320C30. Example 12-35 contains the generic code of the FFT that can be used with any length number. However, for the complete implementation of an FFT, a table of twiddle factors (sines/cosines) is needed, and this table depends on the size of the transform. To retain the generic form of Example 12-35, the table with the twiddle factors (containing 1 1/4 complete cycles of a sine) is presented separately in Example 12-36 for the case of a 64-point FFT. A full cycle of a sine should have a number of points equal to the FFT size. In Example 12-36, the FFT length N and M, which is equal to the logarithm of N to base equal to the radix, are defined. M is the number of stages of the FFT. For a 64-point FFT, M=6 when using a radix-2 algorithm or M=3 when using a radix-4 algorithm. If the table with the twiddle factors and the FFT code are kept in separate files, they should be connected at link time.

Example 12-35. Complex, Radix-2, DIF FFT

```
*
*  TITL  COMPLEX, RADIX-2, DIF FFT
*
*  GENERIC PROGRAM FOR A LOOPED-CODE RADIX-2 FFT COMPUTATION IN 320C30
*
*  THE PROGRAM IS TAKEN FROM THE BURRUS AND PARKS BOOK, P. 111.
*  THE (COMPLEX) DATA RESIDE IN INTERNAL MEMORY.  THE COMPUTATION
*  IS DONE IN-PLACE, BUT THE RESULT IS MOVED TO ANOTHER MEMORY
*  SECTION TO DEMONSTRATE THE BIT-REVERSED ADDRESSING.
*
*  THE TWIDDLE FACTORS ARE SUPPLIED IN A TABLE PUT IN A .DATA SECTION.
*  THIS DATA IS INCLUDED IN A SEPARATE FILE TO PRESERVE THE GENERIC
*  NATURE OF THE PROGRAM.  FOR THE SAME PURPOSE, THE SIZE OF THE FFT
*  N AND LOG2(N) ARE DEFINED IN A .GLOBL DIRECTIVE AND SPECIFIED
*  DURING LINKING.
*
*
        .globl  FFT             ; Entry point for execution
        .globl  N               ; FFT size
        .globl  M               ; LOG2(N)
        .globl  SINE            ; Address of sine table

INP     .usect  "IN",1024       ; Memory with input data
        .BSS    OUTP,1024       ; Memory with output data

        .text

*  INITIALIZE

FFTSIZ  .word   N
LOGFFT  .word   M
SINTAB  .word   SINE
INPUT   .word   INP
OUTPUT  .word   OUTP

FFT:    LDP     FFTSIZ          ; Command to load data page pointer

        LDI     @FFTSIZ,IR1
        LSH     -2,IR1          ; IR1=N/4, pointer for SIN/COS table
        LDI     0,AR6           ; AR6 holds the current stage number
        LDI     @FFTSIZ,IR0
        LSH     1,IR0           ; IR0=2*N1 (because of real/imag)
        LDI     @FFTSIZ,R7      ; R7=N2
        LDI     1,AR7           ; Initialize repeat counter of first loop
        LDI     1,AR5           ; Initialize IE index (AR5=IE)

*  OUTER LOOP

LOOP:   NOP     *++AR6(1)       ; Current FFT stage
        LDI     @INPUT,AR0      ; AR0 points to X(I)
        ADDI    R7,AR0,AR2      ; AR2 points to X(L)
        LDI     AR7,RC
        SUBI    1,RC            ; RC should be one less than desired #
```

```
*  FIRST LOOP

        RPTB    BLK1
        ADDF    *AR0,*AR2,R0    ;  R0=X(I)+X(L)
        SUBF    *AR2++,*AR0++,R1;  R1=X(I)-X(L)
        ADDF    *AR2,*AR0,R2    ;  R2=Y(I)+Y(L)
        SUBF    *AR2,*AR0,R3    ;  R3=Y(I)-Y(L)
        STF     R2,*AR0--       ;  Y(I)=R2  and...
||      STF     R3,*AR2--       ;  Y(L)=R3
BLK1    STF     R0,*AR0++(IR0)  ;  X(I)=R0  and...
||      STF     R1,*AR2++(IR0)  ;  X(L)=R1 and AR0,2 = AR0,2 + 2*n

*  IF THIS IS THE LAST STAGE, YOU ARE DONE

        CMPI    @LOGFFT,AR6
        BZD     END

*  MAIN INNER LOOP

        LDI     2,AR1           ;  Init loop counter for inner loop
        LDI     @SINTAB,AR4     ;  Initialize IA index (AR4=IA)
INLOP:  ADDI    AR5,AR4         ;  IA=IA+IE; AR4 points to cosine
        LDI     AR1,AR0
        ADDI    2,AR1           ;  Increment inner loop counter
        ADDI    @INPUT,AR0      ;  (X(I),Y(I)) pointer
        ADDI    R7,AR0,AR2      ;  (X(L),Y(L)) pointer
        LDI     AR7,RC
        SUBI    1,RC            ;  RC should be one less than desired #
        LDF     *AR4,R6         ;  R6=SIN

*  SECOND LOOP

        RPTB    BLK2
        SUBF    *AR2,*AR0,R2    ;  R2=X(I)-X(L)
        SUBF    *+AR2,*+AR0,R1
*                               ;  R1=Y(I)-Y(L)
        MPYF    R2,R6,R0        ;R0=R2*SIN and...
||      ADDF    *+AR2,*+AR0,R3
*                               ;  R3=Y(I)+Y(L)
        MPYF    R1,*+AR4(IR1),R3 ;R3 = R1 * COS and ...
||      STF     R3,*+AR0        ;  Y(I)=Y(I)+Y(L)
        SUBF    R0,R3,R4        ;  R4=R1*COS-R2*SIN
        MPYF    R1,R6,R0        ;  R0=R1*SIN and...
||      ADDF    *AR2,*AR0,R3    ;  R3=X(I)+X(L)
        MPYF    R2,*+AR4(IR1),R3 ;  R3 = R2 * COS and...
||      STF     R3,*AR0++(IR0)
*                               ;  X(I)=X(I)+X(L) and AR0=AR0+2*N1
        ADDF    R0,R3,R5        ;  R5=R2*COS+R1*SIN
BLK2    STF     R5,*AR2++(IR0) ; X(L)=R2*COS+R1*SIN, incr AR2 and...
||      STF     R4,*+AR2        ;  Y(L)=R1*COS-R2*SIN

        CMPI    R7,AR1
        BNE     INLOP           ;  Loop back to the inner loop

        LSH     1,AR7           ;  Increment loop counter for next time
```

```
        BRDP    LOOP5           ; Next FFT stage (delayed)
        LSH     1,AR5           ; IE=2*IE
        LDI     R7,IR0          ; N1=N2
        LSH     -1,R7           ; N2=N2/2
*  STORE RESULT OUT USING BIT-REVERSED ADDRESSING

END:    LDI     @FFTSIZ,RC      ; RC=N
        SUBI    1,RC            ; RC should be one less than desired #
        LDI     @FFTSIZ,IR0     ; IR0=size of FFT=N
        LDI     2,IR1
        LDI     @INPUT,AR0
        LDI     @OUTPUT,AR1

        RPTB    BITRV
        LDF     *+AR0(1),R0
||      LDF     *AR0++(IR0)B,R1
BITRV   STF     R0,*+AR1(1)
||      STF     R1,*AR1++(IR1)

SELF    BR      SELF            ; Branch to itself at the end
        .end
```

Example 12-36. Table With Twiddle Factors For A 64-Point FFT

```
*
*TITL  TABLE WITH TWIDDLE FACTORS FOR A 64-POINT FFT
*
*  FILE TO BE LINKED WITH THE SOURCE CODE FOR A 64-POINT, RADIX-2 FFT.
*


        .globl   SINE
        .globl   N
        .globl   M

N       .set     64
M       .set     6

        .data

SINE
        .float   0.000000
        .float   0.098017
        .float   0.195090
        .float   0.290285
        .float   0.382683
        .float   0.471397
        .float   0.555570
        .float   0.634393
        .float   0.707107
        .float   0.773010
        .float   0.831470
        .float   0.881921
        .float   0.923880
        .float   0.956940
        .float   0.980785
        .float   0.995185
COSINE
        .float   1.000000
        .float   0.995185
        .float   0.980785
        .float   0.956940
        .float   0.923880
        .float   0.881921
        .float   0.831470
        .float   0.773010
        .float   0.707107
        .float   0.634393
        .float   0.555570
        .float   0.471397
        .float   0.382683
        .float   0.290285
        .float   0.195090
        .float   0.098017
        .float   0.000000
        .float  -0.098017
        .float  -0.195090
        .float  -0.290285
        .float  -0.382683
        .float  -0.471397
        .float  -0.555570
        .float  -0.634393
        .float  -0.707107
        .float  -0.773010
```

```
.float  -0.831470
.float  -0.881921
.float  -0.923880
.float  -0.956940
.float  -0.980785
.float  -0.995185
.float  -1.000000
.float  -0.995185
.float  -0.980785
.float  -0.956940
.float  -0.923880
.float  -0.881921
.float  -0.831470
.float  -0.773010
.float  -0.707107
.float  -0.634393
.float  -0.555570
.float  -0.471397
.float  -0.382683
.float  -0.290285
.float  -0.195090
.float  -0.098017
.float   0.000000
.float   0.098017
.float   0.195090
.float   0.290285
.float   0.382683
.float   0.471397
.float   0.555570
.float   0.634393
.float   0.707107
.float   0.773010
.float   0.831470
.float   0.881921
.float   0.923880
.float   0.956940
.float   0.980785
.float   0.995185
```

The radix-2 algorithm has a great tutorial value because it is relatively easy to understand how the FFT algorithm functions. However, radix-4 implementations can increase the speed of the execution by reducing the overall arithmetic required. Example 12-37 shows the generic implementation of a complex, DIF FFT in radix-4. A companion table, like the one in Example 12-36, should have a value of M equal to the logN, where the base of the logarithm is four.

Example 12-37. Complex, Radix-4, DIF FFT

```
*
*  TITL  COMPLEX, RADIX-4, DIF FFT
*
*  GENERIC PROGRAM TO DO A LOOPED-CODE RADIX-4 FFT COMPUTATION IN
*  THE TMS320C30.
*
*  THE PROGRAM IS TAKEN FROM THE BURRUS AND PARKS BOOK, P. 117.
*  THE (COMPLEX) DATA RESIDE IN INTERNAL MEMORY, AND THE COMPUTATION
*  IS DONE IN-PLACE.
*
*  THE TWIDDLE FACTORS ARE SUPPLIED IN A TABLE PUT IN A .DATA SECTION.
*  THIS DATA IS INCLUDED IN A SEPARATE FILE TO PRESERVE THE GENERIC
*  NATURE OF THE PROGRAM.  FOR THE SAME PURPOSE, THE SIZE OF THE
*  FFT N AND LOG4(N) ARE DEFINED IN A .GLOBL DIRECTIVE AND SPECIFIED
*  DURING LINKING.
*
*  IN ORDER TO HAVE THE FINAL RESULT IN BIT-REVERSED ORDER, THE TWO
*  MIDDLE BRANCHES OF THE RADIX-4 BUTTERFLY ARE INTERCHANGED DURING
*  STORAGE.  NOTE THIS DIFFERENCE WHEN COMPARING WITH THE PROGRAM IN
*  P. 117 OF THE BURRUS AND PARKS BOOK.
*
*
        .globl  FFT             ; Entry point for execution
        .globl  N               ; FFT size
        .globl  M               ; LOG4(N)
        .globl  SINE            ; Address of sine table

        .usect  "IN",INP,1024   ; Memory with input data

        .text

*  INITIALIZE

TEMP    .word   $+2
STORE   .word   FFTSIZ          ; Beginning of temp storage area
        .word   N
        .word   M
        .word   SINE
        .word   INP

        .BSS    FFTSIZ,1        ; FFT size
        .BSS    LOGFFT,1        ; LOG4(FFTSIZ)
        .BSS    SINTAB,1        ; Sine/cosine table base
        .BSS    INPUT,1         ; Area with input data to process
        .BSS    STAGE,1         ; FFT stage #
        .BSS    RPTCNT,1        ; Repeat counter
        .BSS    IEINDX,1        ; IE index for sine/cosine
        .BSS    LPCNT,1         ; Second-loop count
        .BSS    JT,1            ; JT counter in program, P. 117
        .BSS    IA1,1           ; IA1 index in program, P. 117
```

```
FFT:

*  INITIALIZE DATA LOCATIONS
        LDP     TEMP            ;  Command to load data page counter
        LDI     @TEMP,AR0
        LDI     @STORE,AR1
        LDI     *AR0++,R0       ;  Xfer data from one memory to the other
        STI     R0,*AR1++
        LDI     *AR0++,R0
        STI     R0,*AR1++
        LDI     *AR0++,R0
        STI     R0,*AR1++
        LDI     *AR0,R0
        STI     R0,*AR1

        LDP     FFTSIZ          ;  Command to load data page pointer
        LDI     @FFTSIZ,R0
        LDI     @FFTSIZ,IR0
        LDI     @FFTSIZ,IR1
        LDI     0,AR7
        STI     AR7,@STAGE      ;  @STAGE holds the current stage number
        LSH     1,IR0           ;  IR0=2*N1 (because of real/imag)
        LSH     -2,IR1          ;  IR1=N/4, pointer for SIN/COS table
        LDI     1,AR7
        STI     AR7,@RPTCNT     ;  Initialize repeat counter of first loop
        LSH     -2,R0
        STI     AR7,@IEINDX     ;  Initialize IE index
        ADDI    2,R0
        STI     R0,@JT          ;  JT=R0/2+2
        SUBI    2,R0
        LSH     1,R0            ;  R0=N2

*  OUTER LOOP

LOOP:
        LDI     @INPUT,AR0      ;  AR0 points to X(I)
        ADDI    R0,AR0,AR1      ;  AR1 points to X(I1)
        ADDI    R0,AR1,AR2      ;  AR2 points to X(I2)
        ADDI    R0,AR2,AR3      ;  AR3 points to X(I3)
        LDI     @RPTCNT,RC
        SUBI    1,RC            ;  RC should be one less than desired #

*  FIRST LOOP

        RPTB    BLK1
        ADDF    *+AR0,*+AR2,R1
*                               ;  R1=Y(I)+Y(I2)
        ADDF    *+AR3,*+AR1,R3
*                               ;  R3=Y(I1)+Y(I3)
        ADDF    R3,R1,R6        ;  R6=R1+R3
        SUBF    *+AR2,*+AR0,R4
*                               ;  R4=Y(I)-Y(I2)
        STF     R6,*+AR0        ;  Y(I)=R1+R3
        SUBF    R3,R1           ;  R1=R1-R3
        LDF     *AR2,R5         ;  R5=X(I2)
||      LDF     *+AR1,R7        ;  R7=Y(I1)
        ADDF    *AR3,*AR1,R3    ;  R3=X(I1)+X(I3)
        ADDF    R5,*AR0,R1      ;  R1=X(I)+X(I2)
||      STF     R1,*+AR1        ;  Y(I1)=R1-R3
```

```
        ADDF    R3,R1,R6        ; R6=R1+R3
        SUBF    R5,*AR0,R2      ; R2=X(I)-X(I2)
||      STF     R6,*AR0++(IR0)  ;  X(I)=R1+R3
        SUBF    R3,R1           ; R1=R1-R3
        SUBF    *AR3,*AR1,R6    ; R6=X(I1)-X(I3)
        SUBF    R7,*+AR3,R3     ; -R3=Y(I1)-Y(I3)
||      STF     R1,*AR1++(IR0)  ;X(I1)=R1-R3
        SUBF    R6,R4,R5        ; R5=R4-R6
        ADDF    R6,R4           ; R4=R4+R6
        STF     R5,*+AR2        ; Y(I2)=R4-R6
||      STF     R4,*+AR3        ; Y(I3)=R4+R6
        SUBF    R3,R2,R5        ; R5=R2-R3
        ADDF    R3,R2           ; R2=R2+R3
BLK1    STF     R5,*AR2++(IR0)  ;  X(I2)=R2-R3
||      STF     R2,*AR3++(IR0)  ;  X(I3)=R2+R3

*  IF THIS IS THE LAST STAGE, YOU ARE DONE

        LDI     @STAGE,AR7
        ADDI    1,AR7
        CMPI    @LOGFFT,AR7
        BZD     END
        STI     AR7,@STAGE      ; Current FFT stage

*  MAIN INNER LOOP

        LDI     1,AR7
        STI     AR7,@IA1        ; Init IA1 index
        LDI     2,AR7
        STI     AR7,@LPCNT      ; Init loop counter for inner loop
INLOP:
        LDI     2,AR6           ; Increment inner loop counter
        ADDI    @LPCNT,AR6
        LDI     @LPCNT,AR0
        LDI     @IA1,AR7
        ADDI    @IEINDX,AR7     ; IA1=IA1+IE
        ADDI    @INPUT,AR0      ; (X(I),Y(I)) pointer
        STI     AR7,@IA1
        ADDI    R0,AR0,AR1      ; (X(I1),Y(I1)) pointer
        STI     AR6,@LPCNT
        ADDI    R0,AR1,AR2      ; (X(I2),Y(I2)) pointer
        ADDI    R0,AR2,AR3      ; (X(I3),Y(I3)) pointer
        LDI     @RPTCNT,RC
        SUBI    1,RC            ; RC should be one less than desired #
        CMPI    @JT,AR6         ; If LPCNT=JT, go to
        BZD     SPCL            ;    special butterfly
        LDI     @IA1,AR7
        LDI     @IA1,AR4
        ADDI    @SINTAB,AR4     ; Create cosine index AR4
        ADDI    AR4,AR7,AR5
        SUBI    1,AR5           ; IA2=IA1+IA1-1
        ADDI    AR7,AR5,AR6
        SUBI    1,AR6           ; IA3=IA2+IA1-1
```

```
            ; SECOND LOOP

        RPTB    BLK2
        ADDF    *+AR2,*+AR0,R3
*                           ;  R3=Y(I)+Y(I2)
        ADDF    *+AR3,*+AR1,R5
*                           ;  R5=Y(I1)+Y(I3)
        ADDF    R5,R3,R6    ;  R6=R3+R5
        SUBF    *+AR2,*+AR0,R4
*                           ;  R4=Y(I)-Y(I2)
        SUBF    R5,R3       ;  R3=R3-R5
        ADDF    *AR2,*AR0,R1 ;  R1=X(I)+X(I2)
        ADDF    *AR3,*AR1,R5 ;  R5=X(I1)+X(I3)
        MPYF    R3,*+AR5(IR1),R6 R6=R3*CO2
||      STF     R6,*+AR0    ;  Y(I)=R3+R5
        ADDF    R5,R1,R7    ;  R7=R1+R5
        SUBF    *AR2,*AR0,R2 ;  R2=X(I)-X(I2)
        SUBF    R5,R1       ;  R1=R1-R5
        MPYF    R1,*AR5,R7  ;  R7=R1*SI2
||      STF     R7,*AR0++(IR0) ;  X(I)=R1+R5
        SUBF    R7,R6       ;  R6=R3*CO2-R1*SI2
        SUBF    *+AR3,*+AR1,R5
*                           ;  R5=Y(I1)-Y(I3)
        MPYF    R1,*+AR5(IR1),R7 ;R7=R1*CO2
||      STF     R6,*+AR1    ;  Y(I1)=R3*CO2-R1*SI2
        MPYF    R3,*AR5,R6  ;  R6=R3*SI2
        ADDF    R7,R6       ;  R6=R1*CO2+R3*SI2
        ADDF    R5,R2,R1    ;  R1=R2+R5
        SUBF    R5,R2       ;  R2=R2-R5
        SUBF    *AR3,*AR1,R5 ;  R5=X(I1)-X(I3)
        SUBF    R5,R4,R3    ;  R3=R4-R5
        ADDF    R5,R4       ;  R4=R4+R5
        MPYF    R3,*+AR4(IR1),R6 ;  R6=R3*CO1
||      STF     R6,*AR1++(IR0) ;  X(I1)=R1*CO2+R3*SI2
        MPYF    R1,*AR4,R7  ;  R7=R1*SI1
        SUBF    R7,R6       ;  R6=R3*CO1-R1*SI1
        MPYF    R1,*+AR4(IR1),R6 ;  R6=R1*CO1
||      STF     R6,*+AR2    ;  Y(i2)=R3*CO1-R1*SI1
        MPYF    R3,*AR4,R7  ;  R7=R3*SI1
        ADDF    R7,R6       ;  R6=R1*CO1+R3*SI1
        MPYF    R4,*+AR6(IR1),R6  ;  R6=R4*CO3
||      STF     R6,*AR2++(IR0) ;  X(I2)=R1*CO1+R3*SI1
        MPYF    R2,*AR6,R7  ;  R7=R2*SI3
        SUBF    R7,R6       ;  R6=R4*CO3-R2*SI3
        MPYF    R2,*+AR6(IR1),R6 ;  R6=R2*CO3
||      STF     R6,*+AR3    ;  y(i3)=R4*CO3-R2*SI3
        MPYF    R4,*AR6,R7  ;  R7=R4*SI3
        ADDF    R7,R6       ;  R6=R2*CO3+R4*SI3
```

```
BLK2    STF     R6,*AR3++(IR0)
*                               ;  x(i3)=R2*CO3+R4*SI3

        CMPI    @LPCNT,R0
        BP      INLOP           ;  LOOP BACK TO THE INNER LOOP
        BR      CONT

*  SPECIAL BUTTERFLY FOR W=J

SPCL    LDI     IR1,AR4
        LSH     -1,AR4          ;  Point to SIN(45)
        ADDI    @SINTAB,AR4     ;  Create cosine index AR4=CO21

        RPTB    BLK3
        ADDF    *AR2,*AR0,R1    ;  R1=X(I)+X(I2)
        SUBF    *AR2,*AR0,R2    ;  R2=X(I)-X(I2)
        ADDF    *+AR2,*+AR0,R3
*                               ;  R3=Y(I)+Y(I2)
        SUBF    *+AR2,*+AR0,R4
*                               ;  R4=Y(I)-Y(I2)
        ADDF    *AR3,*AR1,R5    ;  R5=X(I1)+X(I3)
        SUBF    R1,R5,R6        ;  R6=R5-R1
        ADDF    R5,R1           ;  R1=R1+R5
        ADDF    *+AR3,*+AR1,R5
*                               ;  R5=Y(I1)+Y(I3)
        SUBF    R5,R3,R7        ;  R7=R3-R5
        ADDF    R5,R3           ;  R3=R3+R5
        STF     R3,*+AR0        ;  Y(I)=R3+R5
||      STF     R1,*AR0++(IR0)  ;  X(I)=R1+R5
        SUBF    *AR3,*AR1,R1    ;  R1=X(I1)-X(I3)
        SUBF    *+AR3,*+AR1,R3
*                               ;  R3=Y(I1)-Y(I3)
        STF     R6,*+AR1        ;  Y(I1)=R5-R1
||      STF     R7,*AR1++(IR0)  ;  X(I1)=R3-R5
        ADDF    R3,R2,R5        ;  R5=R2+R3
        SUBF    R2,R3,R2        ;  R2=-R2+R3
        SUBF    R1,R4,R3        ;  R3=R4-R1
        ADDF    R1,R4           ;  R4=R4+R1
        SUBF    R5,R3,R1        ;  R1=R3-R5
        MPYF    *AR4,R1         ;  R1=R1*CO21
        ADDF    R5,R3           ;  R3=R3+R5
        MPYF    *R4,R3          ;  R3=R3*CO21
||      STF     R1,*+AR2        ;  Y(I2)=(R3-R5)*CO21
        SUBF    R4,R2,R1        ;  R1=R2-R4
        MPYF    *AR4,R1         ;  R1=R1*CO21
||      STF     R3,*AR2++(IR0)  ;  X(I2)=(R3+R5)*CO21
        ADDF    R4,R2           ;  R2=R2+R4
        MPYF    *AR4,R2         ;  R2=R2*CO21
BLK3    STF     R1,*+AR3        ;  Y(I3)=-(R4-R2)*CO21
||      STF     R2,*AR3++(IR0)  ;  X(I3)=(R4+R2)*CO21

        CMPI    @LPCNT,R0
        BPD     INLOP           ;  Loop back to the inner loop
```

```
CONT    LDI     @RPTCNT,AR7
        LDI     @IEINDX,AR6
        LSH     2,AR7           ;  Increment repeat counter for
*                               ;    next time
        STI     AR7,@RPTCNT
        LSH     2,AR6           ;  IE=4*IE
        STI     AR6,@IEINDX
        LDI     R0,IR0          ;  N1=N2
        LSH     -3,R0
        ADDI    2,R0
        STI     R0,@JT          ;  JT=N2/2+2
        SUBI    2,R0
        LSH     1,R0            ;  N2=N2/4
        BR      LOOP            ;  Next FFT stage

*  STORE RESULT OUT USING BIT-REVERSED ADDRESSING

END:    LDI     @FFTSIZ,RC      ;  RC=N
        SUBI    1,RC            ;  RC should be one less than desired #
        LDI     @FFTSIZ,IR0     ;  IR0=size of FFT=N
        LDI     2,IR1
        LDI     @INPUT,AR0
        LDP     STORE
        LDI     @STORE,AR1

        RPTB    BITRV
        LDF     *+AR0(1),R0
||      LDF     *AR0++(IR0)<,R1
BITRV   STF     R0,*+AR1(1)
||      STF     R1,*AR1++(IR1)

SELF    BR      SELF            ;  Branch to itself at the end.
        .end
```

Most often, the data to be transformed is a sequence of real numbers. In this case, the FFT demonstrates certain symmetries that permit the reduction of the computational load even further. Example 12-38 shows the generic implementation of a real-valued, radix-2 FFT. For such an FFT, the total number of storage required for a length-N transform is only N locations instead of 2N that are necessary in a complex FFT. The rest of the points can be recovered based on the symmetry conditions.

Example 12-38. Real, Radix-2 FFT

```
*
*  TITL  REAL, RADIX-2 FFT
*
*  GENERIC PROGRAM TO DO A RADIX-2 REAL FFT COMPUTATION IN 320C30.
*
*  THE PROGRAM IS TAKEN FROM THE PAPER BY SORENSEN ET AL., JUNE 1987
*  ISSUE OF THE TRANSACTIONS ON ASSP.
*
*  THE REAL DATA RESIDE IN INTERNAL MEMORY.  THE COMPUTATION IS
*  DONE IN-PLACE.  THE BIT-REVERSAL IS DONE AT THE BEGINNING OF
*  THE PROGRAM.
*
*  THE TWIDDLE FACTORS ARE SUPPLIED IN A TABLE PUT IN A .DATA
*  SECTION.  THIS DATA IS INCLUDED IN A SEPARATE FILE TO PRESERVE
*  THE GENERIC NATURE OF THE PROGRAM.  FOR THE SAME PURPOSE, THE
*  SIZE OF THE FFT N AND LOG2(N) ARE DEFINED IN A .GLOBL DIRECTIVE
*  AND SPECIFIED DURING LINKING.  THE LENGTH OF THE TABLE IS
*  N/4 + N/4 = N/2.
*
*
        .globl  FFT             ;  Entry point for execution
        .globl  N               ;  FFT size
        .globl  M               ;  LOG2(N)
        .globl  SINE            ;  Address of sine table

        .bss    INP,1024        ;  Memory with input data

        .text

*  INITIALIZE

FFTSIZ  .word   N
LOGFFT  .word   M
SINTAB  .word   SINE
INPUT   .word   INP

FFT:    LDP     FFTSIZ          ;  Command to load data page printer

*  DO THE BIT-REVERSING AT THE BEGINNING

        LDI     @FFTSIZ,RC      ;  RC=N
        SUBI    1,RC            ;  RC should be one less than desired #
        LDI     @FFTSIZ,IR0
        LSH     -1,IR0          ;  IR0=half the size of FFT=N/2
        LDI     @INPUT,AR0
        LDI     @INPUT,AR1

        RPTB    BITRV
        CMPI    AR1,AR0         ;  Exchange locations only
        BGE     CONT            ;    if AR0<AR1
        LDF     *AR0,R0
||      LDF     *AR1,R1
        STF     R0,*AR1
||      STF     R1,*AR0
```

```
CONT    NOP     *AR0++
BITRV   NOP     *AR1++(IR0)B

*  LENGTH-TWO BUTTERFLIES

        LDI     @INPUT,AR0      ;  AR0 points to X(I)
        LDI     IR0,RC          ;  Repeat N/2 times
        SUBI    1,RC            ;  RC should be one less than desired #

        RPTB    BLK1
        ADDF    *+AR0,*AR0++,R0
*                               ;  R0=X(I)+X(I+1)
        SUBF    *AR0,*-AR0,R1
*                               ;  R1=X(I)-X(I+1)
BLK1    STF     R0,*-AR0        ;  X(I)=X(I)+X(I+1)
||      STF     R1,*AR0++       ;  X(I+1)=X(I)-X(I+1)

*  FIRST PASS OF THE DO-20 LOOP (STAGE K=2 IN DO-10 LOOP)

        LDI     @INPUT,AR0      ;  AR0 points to X(I)
        LDI     2,IR0           ;  IR0=2=N2
        LDI     @FFTSIZ,RC
        LSH     -2,RC           ;  Repeat N/4 times
        SUBI    1,RC            ;  RC should be one less than desired #

        RPTB    BLK2
        ADDF    *+AR0(IR0),*AR0++(IR0),R0
*                               ;  R0=X(I)+X(I+2)
        SUBF    *AR0,*-AR0(IR0),R1
*                               ;  R1=X(I)-X(I+2)
        NEGF    *+AR0,R0        ;  R0=-X(I+3)
||      STF     R0,*-AR0(IR0);  X(I)=X(I)+X(I+2)
BLK2    STF     R1,*AR0++(IR0)
*                               ;  X(I+2)=X(I)-X(I+2)
||      STF     R0,*+AR0        ;  X(I+3)=-X(I+3)

*  MAIN LOOP (FFT STAGES)

        LDI     @FFTSIZ,IR0
        LSH     -2,IR0          ;  IR0=index for E
        LDI     3,R5            ;  R5 holds the current stage number
        LDI     1,R4            ;  R4=N4
        LDI     2,R3            ;  R3=N2
LOOP    LSH     -1,IR0          ;  E=E/2
        LSH     1,R4            ;  N4=2*N4
        LSH     1,R3            ;  N2=2*N2

*  INNER LOOP (DO-20 LOOP IN THE PROGRAM)

        LDI     @INPUT,AR5      ;  AR5 points to X(I)
INLOP   LDI     IR0,AR0
        ADDI    @SINTAB,AR0     ;  AR0 points to SIN/COS table
        LDI     R4,IR1          ;  IR1=N4

        LDI     AR5,AR1
        ADDI    1,AR1           ;  AR1 points to X(I1)=X(I+J)
        LDI     AR1,AR3
```

```
        ADDI    R3,AR3          ; AR3 points to X(I3)=X(I+J+N2)
        LDI     AR3,AR2
        SUBI    2,AR2           ; AR2 points to X(I2)=X(I-J+N2)
        ADDI    R3,AR2,AR4      ; AR4 points to X(I4)=X(I-J+N1)

        LDF     *AR5++(IR1),R0
*                               ; R0=X(I)
        ADDF    *+AR5(IR1),R0,R1
*                               ; R1=X(I)+X(I+N2)
        SUBF    R0,*++AR5(IR1),R0
*                               ; R0=-X(I)+X(I+N2)
||      STF     R1,*-AR5(IR1)   ; X(I)=X(I)+X(I+N2)
        NEGF    R0              ; R0=X(I)-X(I+N2)
        NEGF    *++AR5(IR1),R1
*                               ; R1=-X(I+N4+N2)
||      STF     R0,*AR5         ; X(I+N2)=X(I)-X(I+N2)
        STF     R1,*AR5         ; X(I+N4+N2)=-X(I+N4+N2)


           *INNERMOST LOOP

        LDI     @FFTSIZ,IR1
        LSH     -2,IR1          ; IR1=separation between SIN/COS tbls
        LDI     R4,RC
        SUBI    2,RC            ; Repeat N4-1 times

        RPTB    BLK3
        MPYF    *AR3,*+AR0(IR1),R0
*                               ; R0=X(I3)*COS
        MPYF    *AR4,*AR0,R1    ; R1=X(I4)*SIN
        MPYF    *AR4,*+AR0(IR1),R1 ;  R1=X(I4)*COS
||      ADDF    R0,R1,R2        ; R2=X(I3)*COS+X(I4)*SIN
        MPYF    *AR3,*AR0++(IR0),R0
*                               ; R0=X(I3)*SIN
        SUBF    R0,R1,R0        ; R0=-X(I3)*SIN+X(I4)*COS
        SUBF    *AR2,R0,R1      ; R1=-X(I2)+R0
        ADDF    *AR2,R0,R1      ; R1=X(I2)+R0
||      STF     R1,*AR3++       ; X(I3)=-X(I2)+R0
        ADDF    *AR1,R2,R1      ; R1=X(I1)+R2
||      STF     R1,*AR4--       ; X(I4)=X(I2)+R0
        SUBF    R2,*AR1,R1      ; R1=X(I1)-R2
||      STF     R1,*AR1++       ; X(I1)=X(I1)+R2
BLK3    STF     R1,*AR2--       ; X(I2)=X(I1)-R2

        SUBI    @INPUT,AR5
        ADDI    R3,AR5          ; AR5=i+N1
        CMPI    @FFTSIZ,AR5
        BLED    INLOP           ; Loop back to the inner loop
        ADDI    @INPUT,AR5
        NOP
        NOP

        ADDI    1,R5
        CMPI    @LOGFFT,R5
        BLE     LOOP

END     BR      END             ; Branch to itself at the end.
        .end
```

Table 12-1 summarizes the execution time required for FFT lengths between 64 and 1024 points for the three algorithms in Examples 12-35, 12-37, and 12-38. As can be seen, the TMS320C30 permits very fast execution of such transforms. FFT lengths up to 1024 points (complex) or 2048 points (real), covering the majority of applications, can be executed almost entirely in the on-chip memory.

Table 12-1. TMS320C30 FFT Timing Benchmarks

NUMBER OF POINTS	FFT TIMING (in milliseconds) RADIX-2 (complex)	RADIX-4 (complex)	RADIX-2 (real)
64	0.167	0.123	0.075
128	0.367	–	0.162
256	0.801	0.624	0.354
512	1.740	–	0.771
1024	3.750	3.040	1.670

12.4.5 Lattice Filters

The lattice form is an alternative way of implementing digital filters, and it has found applications in speech processing, spectral estimation, and other areas. In the present discussion, the notation and the terminology from speech processing applications will be used.

If H(z) is the transfer function of a digital filter that has only poles, A(z) = 1/H(z) will be a filter having only zeros and it will be called the inverse filter. The inverse lattice filter is shown in Figure 12-5. In mathematical terms, it is described by the equations:

$$f(i,n) = f(i\text{-}1,n) + k(i)\ \ b(i\text{-}1,n\text{-}1)$$
$$b(i,n) = b(i\text{-}1,n\text{-}1) + k(i)\ f(i\text{-}1,n)$$

Initial conditions:

$$f(0,n) = b(0,n) = x(n)$$

Final conditions:

$$y(n) = f(p,n).$$

$f(i,n)$ is called the forward error, $b(i,n)$ backward error, $k(i)$ is the i-th reflection coefficient, $x(n)$ is the input, and $y(n)$ the output signal. The order of the filter (i.e., the number of stages) is p. In the linear predictive coding (LPC) method of speech processing, the inverse lattice filter is used during analysis, and the (forward) lattice filter during speech synthesis.

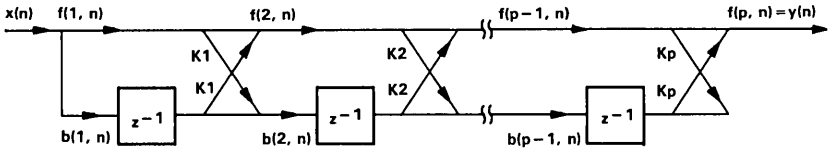


Figure 12-5. Structure of the Inverse Lattice Filter

Figure 12-6 shows the data memory organization of the inverse lattice-filter on the TMS320C30.

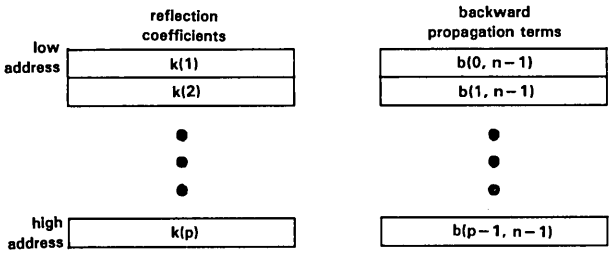


Figure 12-6. Data Memory Organization for Lattice Filters

Example 12-39. Inverse Lattice Filter

```
*  TITL  INVERSE LATTICE FILTER
*
*
*  SUBROUTINE L A T I N V
*
*  LATINV == LATTICE FILTER (LPC INVERSE FILTER - ANALYSIS)
*
*
*  TYPICAL CALLING SEQUENCE:
*
*        load    R2
*        load    AR0
*        load    AR1
*        load    RC
*        CALL    LATINV
*
*
*  ARGUMENT ASSIGNMENTS:
*    ARGUMENT | FUNCTION
*    ---------+-----------------------
*    R2       | f(0,n) = x(n)
*    AR0      | ADDRESS OF FILTER COEFFICIENTS  k(1))
*    AR1      | ADDRESS OF BACKWARD PROPAGATION
*             |   VALUES (b(0,n-1))
*    RC       | RC = p - 2
*
*  REGISTERS USED AS INPUT: R2, AR0, AR1, RC
*  REGISTERS MODIFIED: R0, R1, R2, R3, RS, RE, RC, AR0, AR1
*  REGISTER CONTAINING RESULT: R2 (f(p,n))
*
*
*  PROGRAM SIZE: 10 WORDS
*
*  EXECUTION CYCLES: 13 + 3 * (p-1)
*
*
        .global LATINV
*
*  i = 1
*
LATINV   MPYF3   *AR0, *AR1, R0
*                              ; k(1) * b(0,n-1) -> R0
*                              ; Assume f(0,n) -> R2.
         LDF     R2,R3         ; Put b(0,n) = f(0,n) -> R3.
         MPYF3   *AR0++(1),R2,R1
*                              ; k(1) * f(0,n) -> R1
*
```

```
*   2 <= i <= p
*
        RPTB    LOOP
        MPYF3   *AR0,*++AR1(1),R0   ;  k(i) * b(i-1,n-1) -> R0
||      ADDF3   R2,R0,R2            ;  f(i-1-1,n)+k(i-1)
*                                   ;     *b(i-1-1,n-1)
*                                   ;     = f(i-1,n) -> R2
*
*                                   ;  b(i-1-1,b-1)+k(i-1)*f(i-1-1,n)
        ADDF3   *-AR1(1), R1, R3    ;  = b(i-1,n) -> R3
||      STF     R3, *-AR1(1)        ;  b(i-1-1,n) -> b(i-1-1,n-1)
*
LOOP    MPYF3   *AR0++(1),R2,R1
*                                   ;  k(i) * f(i-1,n) -> R1
*
*   I = P+1 (CLEANUP)

        ADDF3   R2,R0,R2            ;  f(p-1,n)+k(p)*b(p-1,n-1)
*                                   ;   = f(p,n) -> R2
*
*                                   ;  b(p-1,n-1)+k(p)*f(p-1,n)
        ADDF35  AR1, R1, R3         ;   = b(p,n) -> R3
||      STF     R3, *AR1            ;  b(p-1,n) -> b(p-1,n-1)
*
*   RETURN SEQUENCE
*
        RETS                        ;  RETURN
*
*   end
*
    .end
```

The (forward) lattice filter has a structure very similar to the inverse filter, as shown in Figure 12-7. The corresponding equations describing the lattice filter are:

$$f(i\text{-}1,n) = f(i,n) - k(i)\ \ b(i\text{-}1,n\text{-}1)$$
$$b(i,n) = b(i\text{-}1,n\text{-}1) + k(i)\ f(\ i\text{-}1,n)$$

Initial conditions:

$$f(p,n) = x(\ n),\ b(i,n\text{-}1) = 0 \qquad \text{for } i=1,\ldots,p$$

Final conditions:

$$y(n) = f(0,n).$$

The data memory organization is identical to the one of the inverse filter, as shown in Figure 12-6. Example 12-40 is the implementation of the lattice filter on the TMS320C30.

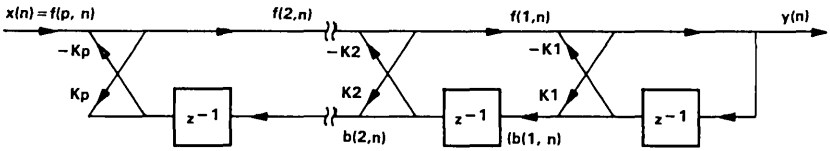


Figure 12-7. Structure of the (Forward) Lattice Filter

Example 12-40. Lattice Filter

```
*  TITL  LATTICE FILTER
*
*
*  SUBROUTINE L A T I C E
*
*       LOAD    AR0
*       LOAD    AR1
*       LOAD    RC
*       CALL    LATICE
*
*
*  ARGUMENT ASSIGNMENTS:
*    ARGUMENT | FUNCTIONfunction
*    ---------+----------------------
*    R2       | F(P,N) = E(N) = EXCITATION
*    AR0      | ADDRESS OF FILTER COEFFICIENTS (K(P))
*    AR1      | ADDRESS OF BACKWARD PROPAGATION VALUES (B(P-1,N-1))
*    RC       | RC = P - 2
*
*  REGISTERS USED AS INPUT: R2, AR0, AR1, RC
*  REGISTERS MODIFIED: R0, R1, R2, R3, RS, RE, RC, AR0, AR1
*  REGISTER CONTAINING RESULT: R2 (f(0,n))
*
*  STACK USAGE: NONE
*
*  PROGRAM SIZE: 12 WORDS
*
*  EXECUTION CYCLES: 13 + 5 * (P-1)
*
*
       .global LATICE
*
*
LATICE  MPYF3   *AR0, *AR1, R0
*                           ;  K(P) * B(P-1,N-1) -> R0
       SUBF3   R0,R2,R2     ;  ASSUME F(P,N) -> R2.
*                           ;  F(P,N)-K(P)*B(P-1,N-1)
*                           ;   =F(P-1,N) -> R2
*
*  2 <= I <= P
*
       RPTB    LOOP
       MPYF3   *AR0,R2,R1   ;  K(I) * F(I-1,N) -> R1
       MPYF3   *--AR0(1), *-AR1(1), R0
*                           ;  K(I-1) * B(I-1-1,N-1) -> R
       ADDF3   *AR1--(1), R1, R3
*                           ;  B(I-1,N-1) + K(I) * F(I-1,N)
*                           ;  = B(I,N) -> R3
       STF     R3, *+AR1(2) ;  B(I,N) -> B(I,N-1)
LOOP   SUBF3   R0,R2,R2     ;  F(I-1,N)-K(I-1)*B(I-1-1,N-1)
*                           ;   = F(I-1-1,N) -> R2
*
*
```

```
*   I = 1 (CLEANUP)
*
        MPYF3    *AR0, R2, R1 ;  K(1) * F(0,N) -> R1
        ADDF3    *AR1, R1, R3 ;  B(0,N-1) + K(1) * F(0,N)
*                             ;     = B(1,N) -> R3
        STF      R3, *+AR1(1) ;  B(1,N) -> B(1,N-1)
||      STF      R2, *AR1     ;  F(0,N) -> B(0,N-1)
*
*   RETURN SEQUENCE
*
        RETS                  ;  RETURN
*
*   end
*
        .end
```

12.5 Programming Tips

Programming style is highly personal, and reflects each individual's preferences and experiences. The purpose of this section is not to impose any particular style. Instead, it intends to emphasize some of the features of the TMS320C30 that can help in producing faster and/or shorter programs. The following covers both C compiler and assembly language programming.

12.5.1 C-Callable Routines

The TMS320C30 was designed with a high-level language (HLL) in mind. The large register file, the software stack and the large memory space makes implementation of a HLL compiler an easy task. The first such implementation supplied is a C compiler. Use of the C compiler increases the transportability of applications that have been tested on large, general-purpose computers, and decreases their porting time.

For best usage of the compiler:

1) Write the application in the high-level language.
2) Debug the program.
3) Estimate if it runs in real-time.
4) If not, identify places where most of the execution time is spent.
5) Optimize these areas by writing assembly language routines implementing the functions.
6) Call the routines from the C program as C functions.

When writing a C program, a simple way to increase the execution speed is to maximize the use of register variables. For more information, refer to the TMS320C30 C Compiler Reference Guide.

There are certain conventions that have to be observed in writing a C-callable routine. These conventions are outlined in the Runtime Environment chapter of the "TMS320C30 C Compiler Reference Guide". Certain registers are saved by the calling function and others need to be saved by the called function. The C compiler manual will help achieve a very clean interface. The end result is the readability and natural flow of a high-level language combined with the efficiency and special-feature use of assembly language.

12.5.2 Hints for Assembly Coding

Each program will have its particular requirements. Not all possible optimizations will make sense in every case. The suggestions presented in this section can be used as a checklist of available software tools.

- **Use delayed branches.** Delayed branches take a single cycle to execute, regular branches take four. The following three instructions are also executed no matter if the branch is taken or not. If there are less than three instructions that could be used, use the delayed branch and append NOPs. Machine cycles (time) are still being saved.

- **Apply the repeat single/block construct.** In this way, loops are achieved with no overhead. Nesting such constructs normally will not increase efficiency, so try to use the repeat feature on the most often performed loop. Note that RPTS is not interruptible, and the executed

instruction is not refetched for execution. This frees the buses for operands.

- **Use parallel instructions.** It is possible to have a multiply in parallel with an add (or subtract), and stores in parallel with any multiply or ALU operation. This increases the number of operations executed in a single cycle. For maximum efficiency, observe the addressing modes used in parallel instructions and arrange the data appropriately.

- **Maximize the use of registers.** The registers are a very efficient, easy way to access scratch-pad memory. Extensive use of the register file will also help when using parallel instructions and in avoiding the pipeline conflicts encountered using the registers in addressing modes.

- **Use the cache.** Especially in conjunction with external slow memory. The cache is transparent to the user, so make sure that it is enabled.

- **Use internal memory instead of external memory.** The internal memory (2K x 32 bits RAM and 4K x 32 bits ROM) is considerably faster to access. In a single cycle, two operands can be brought from internal memory. A way of maximizing performance is to use the DMA in parallel with the CPU to transfer data to internal memory before operating on them.

- **Avoid pipeline conflicts.** If there is no problem with program speed, ignore this suggestion. For time-critical operations, make sure that cycles are not missed because of conflicts. The way to identify such conflicts is to run the trace function on the development tools (simulator, emulators) with the program tracing option enabled. The tracing will identify immediately the pipeline conflicts. Consulting the appropriate section of this User's Guide will explain the reason for the conflict. Steps can then be taken to correct the problem.

The above checklist is not exhaustive, and it does not address the more detailed features outlined in the different sections of this manual. To exploit the full power of the TMS320C30 it is recommended that the architecture, hardware configuration, and instruction set of the device, described in earlier chapters, be carefully studied.

Introduction 1

Pinout and Signal Descriptions 2

Architectural Overview 3

CPU Registers, Memory, and Cache 4

Data Formats and Floating-Point Operation 5

Addressing 6

Program Flow Control 7

External Bus Operation 8

Peripherals 9

Pipeline Operation 10

Assembly Language Instructions 11

Software Applications 12

Hardware Applications 13

Appendices A-D

Section 13

Hardware Applications

The TMS320C30's advanced interface design allows this device to be used to implement a wide variety of system configurations. Its two external buses and DMA capability provide a parallel 32-bit interface to byte- or word-wide devices, while the interrupt interface, dual serial ports, and general purpose digital I/O provide communication with a multitude of peripherals.

This section describes how to use the TMS320C30's interfaces to connect to various external devices. Specific discussions include implementation of parallel interface to devices with and without wait states, use of DMA and general purpose I/O, and multiprocessing considerations.

Major topics discussed in this section are as follows:

- System Configuration Options Overview (Section 13.1 on page 13-2)
- Primary Bus Interface (Section 13.2 on page 13-4)
  - Zero Wait State Interface to RAMs
  - Ready Generation
  - Bank Switching Techniques
- Expansion Bus Interface (Section 13.3 on page 13-14)
- System Control Functions (Section 13.4 on page 13-18)
  - Clock Oscillator Circuitry
  - Reset Signal Generator
- User Target Design Considerations When Using the XDS1000 (Section 13.5 on page 13-22)

13.1 System Configuration Options Overview

The various TMS320C30 interfaces allow connections to a wide variety of different device types. Each of these interfaces is tailored to a particular family of devices.

13.1.1 Categories of Interfaces on the TMS320C30

The interface types on the TMS320C30 fall into several different categories depending on the devices to which they were intended to be connected. Each interface comprises one or more signal lines which transfer information and control its operation. Shown in Figure 13-1 are the signal line groupings for each of these various interfaces.

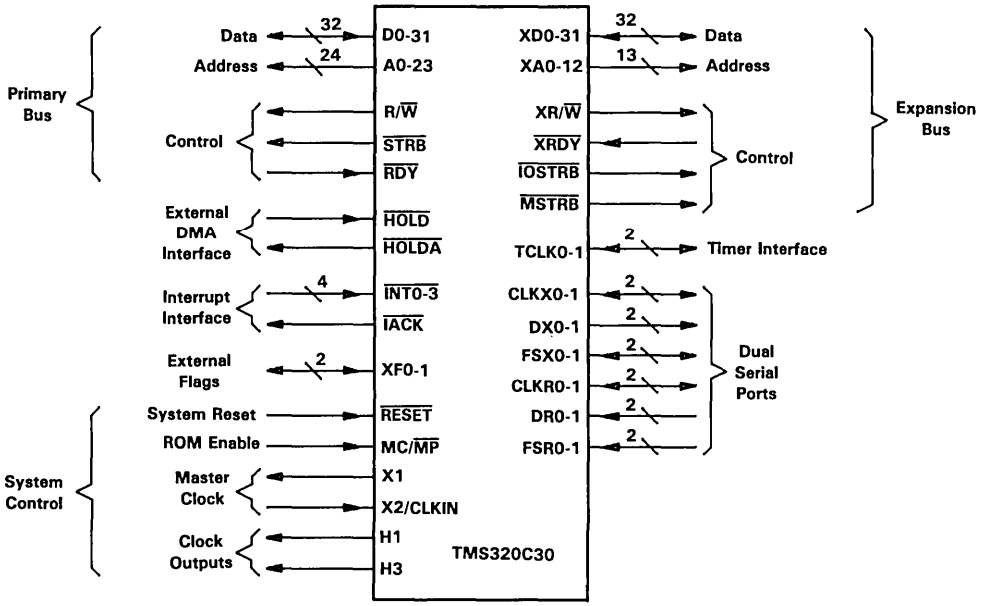


Figure 13-1. External Interfaces on the TMS320C30

13

All of the interfaces are independent of one another and different operations may be performed simultaneously on each interface.

The Primary and Expansion buses implement the memory mapped interface to the device. The external DMA interface allows external devices to cause the processor to relinquish the Primary bus and allow direct memory access.

13.1.2 Typical System Block Diagram

The devices which can be interfaced to the TMS320C30 include memory, DMA devices, and numerous parallel and serial peripherals and I/O devices. Figure 13-2 illustrates a typical configuration of a TMS320C30 system showing different types of external devices and the interfaces to which they are connected.

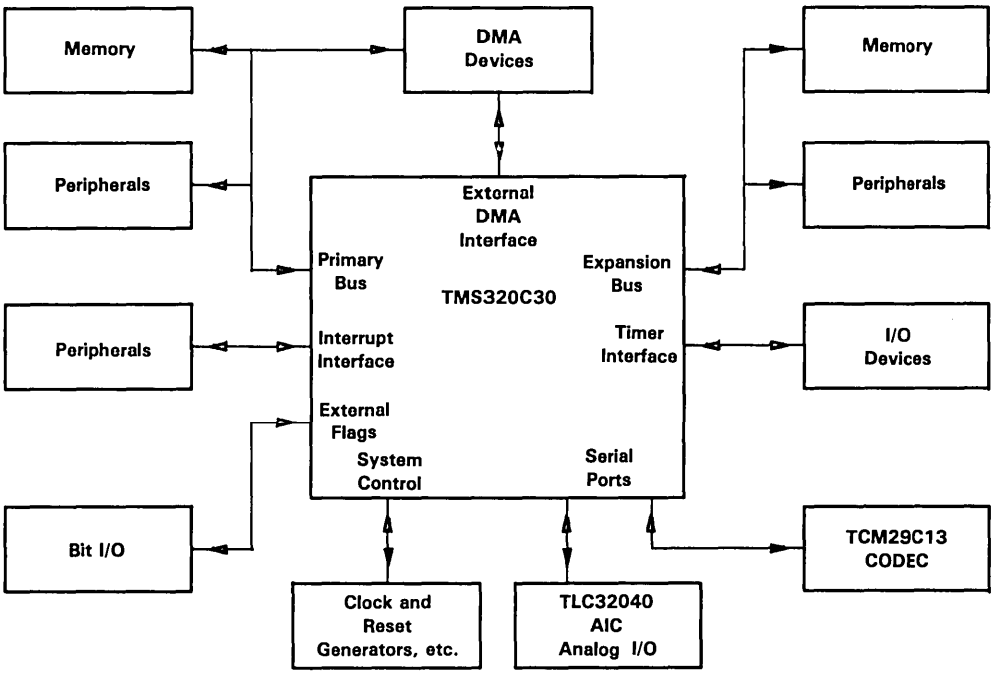


Figure 13-2. Possible System Configurations

This block diagram constitutes more or less a fully expanded system. In an actual design any subset of the illustrated configuration may be used.

13.2 Primary Bus Interface

The primary bus is used by the TMS320C30 to access the majority of its memory mapped locations. Therefore, typically when a large amount of external memory is required in a system, it is interfaced to the primary bus. The expansion bus (discussed in the next subsection) actually comprises two mutually exclusive interfaces, controlled by the $\overline{\text{MSTRB}}$ and $\overline{\text{IOSTRB}}$ signals respectively. Cycles on the expansion bus controlled by the $\overline{\text{MSTRB}}$ signal are identical in timing to cycles on the primary bus, with the exception that bank switching is not implemented on the expansion bus. Accordingly, the discussion of primary bus cycles in this section applies equally to $\overline{\text{MSTRB}}$ cycles on the expansion bus.

Although both the primary bus and the expansion bus may be used to interface to a wide variety of devices, the devices most commonly interfaced to these buses are memories. Therefore, detailed examples of memory interface will presented in this subsection.

13.2.1 Zero Wait-State Interface To RAMs

For full speed, zero wait-state interface to any devices, the TMS320C30 requires a read access time of 35 ns from address stable to data valid. Since, for most memories, access time from chip select is the same as access time from address, it is theoretically possible to use 35 ns memories at full speed with the TMS320C30. This, however, dictates that there be no delays present between the processor and the memories. This is usually not the case in practice, due to interconnection delays and the fact that typically some gating is required for chip select generation. Therefore, slightly faster memories are generally required in most systems. If one level of reasonably high-speed (below 10 ns in propagation delay) gating is used to generate chip select for the memories, 25 ns devices may be used.

Among currently available RAMs, there are two distinct categories of devices with different interface characteristics. These two categories are RAMs without output enable control lines ($\overline{\text{OE}}$), which include the 1-bit wide organized RAMs and most of the 4-bit wide RAMs, and those with $\overline{\text{OE}}$ controls, which include the byte wide and a few of the 4-bit wide RAMs. Many of the fastest RAMs do not provide $\overline{\text{OE}}$ control, and use chip select ($\overline{\text{CS}}$) controlled write cycles to insure that data outputs do not turn on for write operations. In $\overline{\text{CS}}$ controlled write cycles, the write control line ($\overline{\text{WE}}$) goes low prior to $\overline{\text{CS}}$ going low, and internal logic holds the outputs disabled until the cycle is completed. Using $\overline{\text{CS}}$ controlled write cycles is an efficient way to interface fast RAMs without $\overline{\text{OE}}$ controls to the TMS320C30 at full speed.

Figure 13-3 shows the TMS320C30 interfaced to Cypress Semiconductor's CY7C164 25 ns 16k x 4-bit CMOS static RAMs with zero wait states using $\overline{\text{CS}}$ controlled write cycles. These RAMs are arranged to implement 16k 32-bit words located at addresses 00000H thru 03FFFH, which are the first 16k words in external memory. Note that in Figure 13-3 the $\overline{\text{RDY}}$ input is tied low, selecting zero wait states for all accesses on the bus.

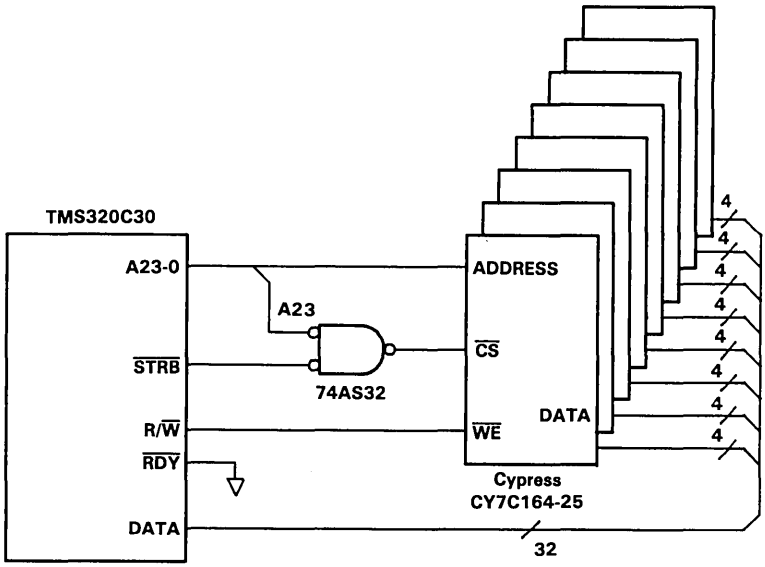


Figure 13-3. Ram Interface - No $\overline{OE}$

In this circuit, chip select is generated from $\overline{STRB}$ and A23 using a 74AS32, whose propagation delay is only 5.8 ns. Thus, the chip select delay added to the RAM's 25 ns chip select access time satisfies the TMS320C30's 35 ns read access time from address. This approach works well if only a single bank of external memory is implemented where the chip select decode can be accomplished in only one level of gating. If more than one bank is required to implement very large memory spaces, bank switching can be used to provide for multiple bank select generation while still maintaining full speed accesses within each bank. Bank switching is discussed in detail in a later subsection.

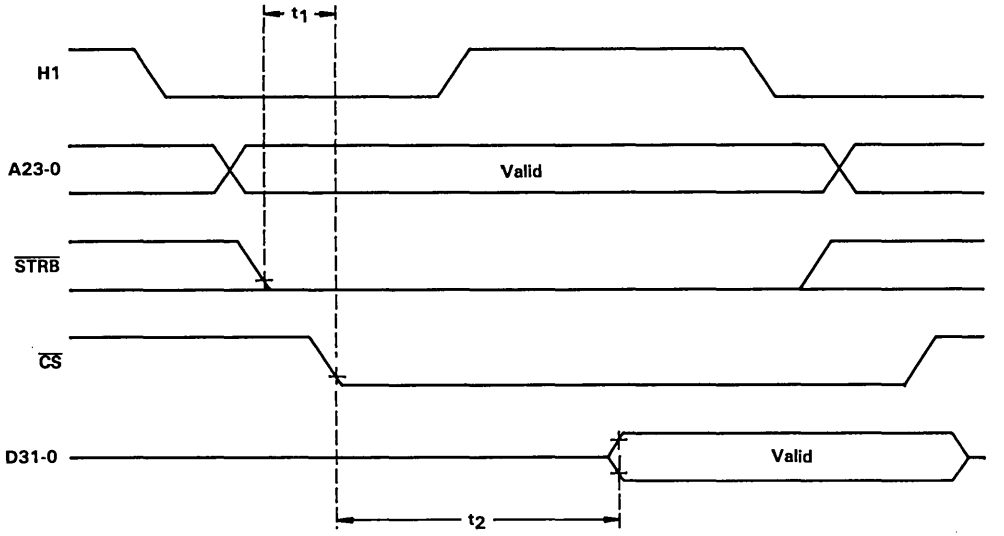


Figure 13-4. Interface Read Timing

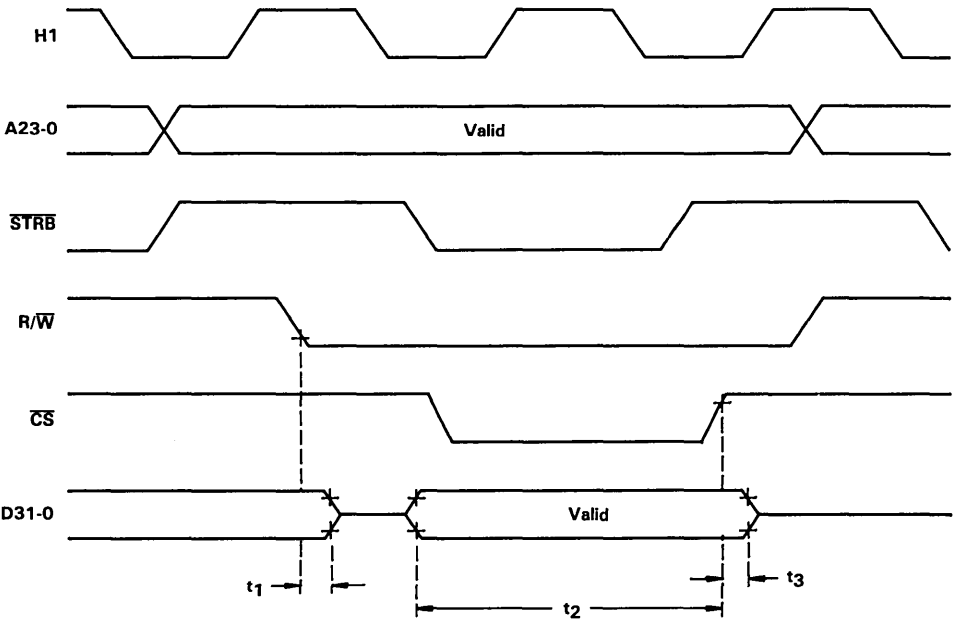


Figure 13-5. Interface Write Timing

Figures 13-4 and 13-5 show the read and write timings of this interface, respectively. For read operations, $\overline{WE}$ (R/$\overline{W}$) is inactive (high), and the device is selected whenever both $\overline{STRB}$ and A23 are low. The total time from address to data from the RAM is therefore:

$$tacc = t1 + t2 = 5.8 + 25 = 30.8 \text{ ns}$$

This easily meets the TMS320C30's 35 ns access time requirement. For write operations, address and R/$\overline{W}$ change state far enough away in time from the low $\overline{STRB}$ pulse to allow this interface to easily meet specifications for most RAMs' $\overline{CS}$ controlled write cycles. In this case, the CY7C164s outputs disable at the beginning of the cycle well early enough (t1 =7 ns) to avoid bus contention with the TMS320C30. Data is then driven into the RAMs as $\overline{STRB}$ goes low. The RAMs require 13 ns of write data setup prior to $\overline{CS}$ going high, and this design provides around 65 ns (t2). A data hold time of 0 ns (t3) is required by the RAMs, and this design provides greater than 10 ns. Finally, the RAMs setup and hold times for address with respect to $\overline{CS}$ of 0 ns are also met with a clear margin.

Some RAMs with $\overline{OE}$ controls can also use $\overline{CS}$ controlled write cycles and this interface may be used with some of these devices with $\overline{OE}$ tied low. There are, however, two requirements for the use of $\overline{OE}$ RAMs with this interface. First, the RAM's $\overline{OE}$ input must be gated with chip select and $\overline{WE}$ internally so that the device's outputs do not turn on unless a read is being performed. Secondly, the RAM must allow address inputs to change while $\overline{WE}$ is low, which some RAMs specifically prohibit.

Many RAMs with $\overline{OE}$ controls that do not meet the design criteria for the circuit shown in Figure 13-3 may be interfaced to the TMS320C30 using the approach shown in Figure 13-6

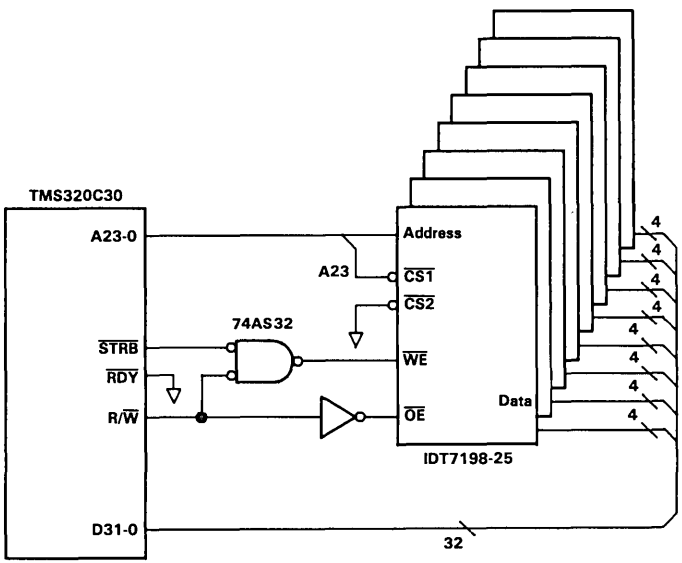


Figure 13-6. RAM Interface - $\overline{OE}$

This design shows an interface to Integrated Device Technology's IDT7198 25 ns 16k x 4-bit CMOS static RAMs using $\overline{OE}$ to enable and disable the data outputs.

In this circuit, chip select is driven directly by a single address line, which locates the RAM at addresses 00000H through 03FFFH in external memory. The RAM's $\overline{WE}$ input is generated by ANDing R/$\overline{W}$ and $\overline{STRB}$, and therefore $\overline{WE}$ goes low after $\overline{CS}$ only during write cycles. This satisfies the RAM's requirement that address never changes when $\overline{WE}$ is low.

The timing of read operations, shown in Figure 13-7, is very straightforward since $\overline{CS}$ is driven directly. The read access time of the circuit, t1, is therefore simply the RAM's chip select/address access time, which is 25 ns. This provides 10 ns of margin over the TMS320C30's 35 ns requirement.

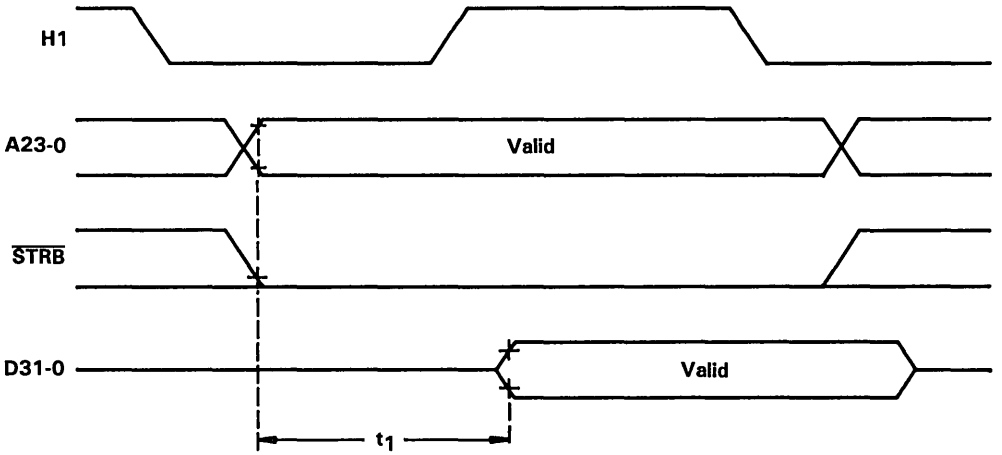


Figure 13-7. Read Operations Timing

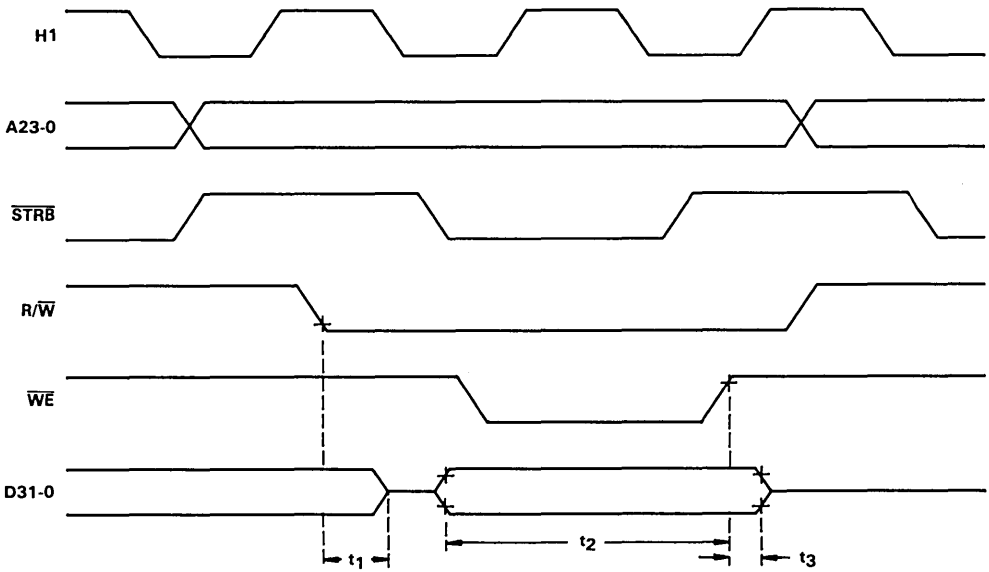


Figure 13-8. Write Operations Timing

During write operations, as shown in Figure 13-8, the RAM's outputs are disabled after a delay of t1 following R/$\overline{W}$ going low. This delay comprises the inverter propagation delay and the RAM's turn-off delay, therefore t1 is given by:

$$t1 = 5 + 15 = 20 \text{ ns}$$

which results in the outputs being disabled no later than the falling edge of H1, thereby avoiding bus contention with the TMS320C30. The circuit's data setup and hold times of approximately 65 and 10 ns, respectively also easily meet the RAM's timing requirements.

As with the circuit of Figure 13-3, if more complex chip select decode is required than can be accomplished in time to meet zero wait state timing, wait states or bank switching techniques (discussed in a later subsection) should be used.

It should be noted that the IDT7198's $\overline{OE}$ control is gated with $\overline{CS}$ internally, therefore the RAM's outputs are not enabled unless the device is selected. This is critical if there are any other devices connected to the same bus; if there are no other devices connected to the bus, then $\overline{OE}$ need not be gated internally with chip select.

13.2.2 Ready Generation

The use of wait states can greatly increase system flexibility and reduce hardware requirements over systems without wait state capability. The TMS320C30 has the capability of generating wait states on either the primary bus or the expansion bus; both buses have independent sets of ready control logic. Ready generation is discussed in this subsection from the perspective of the primary bus interface, however, wait state operation on the expansion bus is identical to that of the primary bus, therefore these discussions pertain equally well to expansion bus operation. Ready generation will not be included in the specific discussions of the expansion bus interface.

Wait states are generated on the basis of the internal wait state generator, the external ready input ($\overline{RDY}$), or the logical AND or OR of the two (see Section 8.3). When enabled, internally generated wait states effect all external cycles, regardless of the address accessed. If different numbers of wait states are required for various external devices, the external $\overline{RDY}$ input may be used to tailor wait state generation to specific system requirements.

If the logical OR (or electrical AND since the signals are low true) of the external and wait count ready signals is selected, the earlier of either of the two signals will generate a ready condition and allow the cycle to be completed. It is not required that both signals be present.

The OR of the two ready signals can be used to implement wait states for devices which require a greater number of wait states than are implemented with external logic (up to eight). This feature is useful, for example, if a system contains some fast and some slow devices. In this case, fast devices can generate ready externally with a minimum of logic, and slow devices can use the internal wait counter for larger numbers of wait states. Thus, when fast devices are accessed, the external hardware responds promptly with ready which terminates the cycle. When slow devices are accessed, the external hardware does not respond, and the cycle is appropriately terminated after the internal wait count.

The OR of the two ready signals may also be used if conditions occur which require termination of bus cycles prior to the number of wait states implemented with external logic. In this case, a shorter wait count is specified internally than the number of wait states implemented with the external ready logic, and the bus cycle is terminated after the wait count. This feature may also be used as a safeguard against inadvertent accesses to nonexistent memory which would never respond with ready and therefore lock up the TMS320C30.

If the OR of the two ready signals is used, however, and the internal wait state count is less than the number of wait states implemented externally, the external ready generation logic must have the ability to reset its sequencing to allow a new cycle to begin immediately following the end of the internal wait count. This requires that, under these conditions, consecutive cycles must be from independently decoded areas of memory and that the external ready generation logic be capable of restarting its sequence as soon as a new cycle begins. Otherwise, the external ready generation logic may loose synchronization with bus cycles and therefore generate improperly timed wait states.

If the logical AND (electrical OR) of the wait count and external ready signals is selected, the later of the two signals will control the internal ready signal, but both signals must occur. Accordingly, external ready control must be implemented for each wait state device in addition to the wait count ready signal being enabled.

This feature is useful if there are devices in a system which are equipped to provide a ready signal but cannot respond quickly enough to meet the TMS320C30's timing requirements. In particular, if these devices normally indicate a ready condition and, when accessed, respond with a wait until they become ready, the logical AND of the two ready signals can be used to save hardware in the system. In this case, the internal wait counter can be used to provide wait states initially, and become ready after the external device has had time to send a not ready indication. The internal wait counter then remains ready until the external device also becomes ready, which terminates the cycle.

Additionally, the AND of the two ready signals may be used for extending the number of wait states for devices which already have external ready logic implemented but require additional wait states under certain unique circumstances.

In the implementation of external ready generation hardware, the particular technique employed depends heavily on the specific characteristics of the system. The optimum approach to ready generation varies depending on the relative number of wait state and non-wait state devices in the system and the maximum number of wait states required for any one device. The approaches discussed here are intended to be general enough for most applications, and are easily modifiable to comprehend many different system configurations.

In general, ready generation involves the following three functions:

1) Segmentation of the address space in some fashion to distinguish fast and slow devices.

2) Generate properly timed ready indications.

3) Logically ORing all of the separate ready timing signals together to connect to the physical ready input.

Segmentation of the address space is required so that a unique indication of each of the particular areas within the address space that require wait states can be obtained. This segmentation is commonly implemented in a system in the form of chip select generation. Chip select signals may be used to initiate wait states in many cases, however, occasionally chip select decoding considerations may provide signals which will not allow ready input timing requirements to be met. In this case, coarse address space segmentation may be made on the basis of a small number of address lines, where simpler gating allows signals to be generated more quickly. In either case, the signal indicating that a particular area of memory is being addressed is normally used to initiate a ready or wait state indication.

Once the region of address space being accessed has been established, a timing circuit of some sort is normally used to provide a ready indication to the processor at the appropriate point in the cycle to satisfy each device's unique requirements.

Finally, since indications of ready status from multiple devices are typically present, an OR gate is commonly used to combine the signals to drive the $\overline{RDY}$ input.

One of two basic approaches may be taken in the implementation of ready control logic depending upon the state in which the ready input is to be between accesses. If $\overline{RDY}$ is low between accesses, the processor is always ready unless a wait state is required; if $\overline{RDY}$ is high between accesses, the processor will always enter a wait state unless a ready indication is generated.

If $\overline{RDY}$ is low between accesses, control of full speed devices is straightforward; no action is necessary since ready is always active unless otherwise required. Devices requiring wait states, however, must drive ready high fast enough to meet the input timing requirements. Then, after an appropriate delay, a ready indication must be generated. This can be quite difficult in many circumstances since wait state devices are inherently slow and often require complex select decoding.

If $\overline{RDY}$ is high between accesses, zero wait state devices, which tend to be inherently fast, can usually respond immediately with a ready indication. Wait state devices may simply delay their select signals appropriately to generate a ready. Typically, this approach results in the most efficient implementation of ready control logic. Figure 13-9 shows a circuit of this type which can be used to generate 0, 1, or 2 wait states for multiple devices in a system.

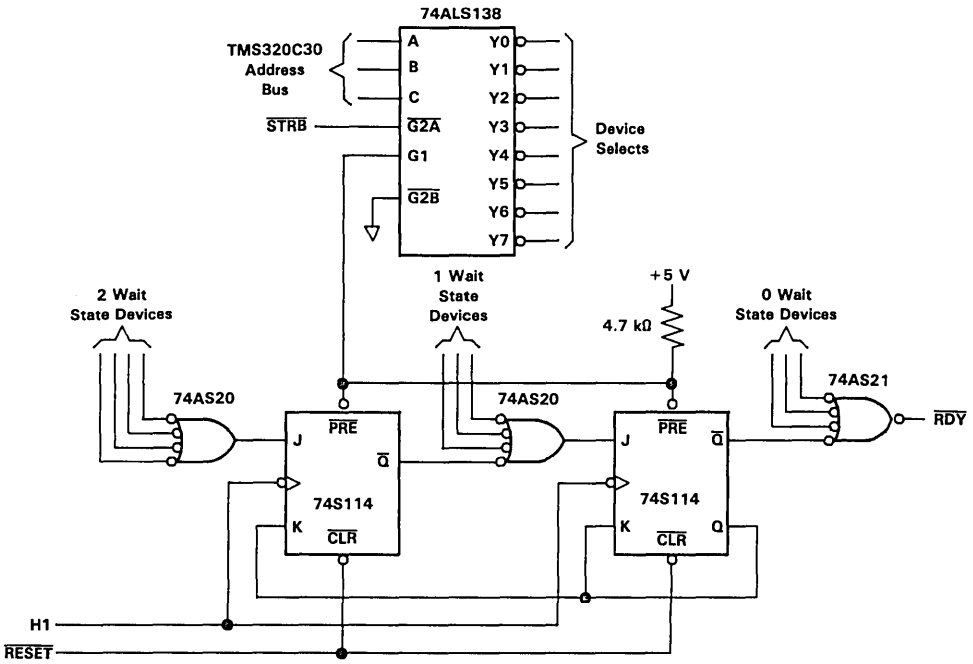


Figure 13-9. Circuit For Generation of 0, 1, or 2 Wait States For Multiple Devices

In this circuit, full speed devices drive ready directly through the 74AS21, and the two flip-flops delay wait state devices' select signals one or two H1 cycles to provide 1 or 2 wait states.

Considering the TMS320C30's ready delay time of 8 ns following address, zero wait state devices must use ungated address lines directly to drive the input of the 74AS21, since this gate contributes a maximum propagation delay of 6 ns to the $\overline{RDY}$ signal. Thus, zero wait state devices should be grouped together within a coarse segmentation of address space if other devices in the system require wait states.

With this circuit, devices requiring wait states may take up to 42 ns from a valid address on the TMS320C30 to provide inputs to the 74AS20's inputs. Typically, this allows sufficient time for any decoding required in generating select signals for slower devices in the system. For example, the 74ALS138 driven by address and $\overline{STRB}$, can generate select decodes in 22 ns, which easily meets the TMS320C30's timing requirements.

With this circuit, unused inputs to either the 74AS20s or the 74AS21 should be tied to a logic high level to prevent noise from generating spurious wait states.

If more than 2 wait states are required by devices within a system, other approaches may be employed for ready generation. If between three and eight wait states are required, additional flip-flops may be included, in the same manner as shown in Figure 13-9, or internally generated wait states may be used in conjunction with external hardware. If greater than eight wait states are required, an external circuit using a counter may be used to supplement the internal wait state generators capabilities.

13.2.3 Bank Switching Techniques

The TMS320C30's programmable bank switching feature can greatly ease system design when large amounts of memory are required. This feature is used to provide a period of time during which all device selects are disabled that would not normally be present otherwise (refer to Section 8.4 for further information regarding bank switching). During this interval, slow devices are allowed time to turn off before other devices have the opportunity to drive the data bus, thus avoiding bus contention.

When bank switching is enabled, any time a portion of the high order address lines change, as defined by the contents of the BNKCMPR register, $\overline{\text{STRB}}$ goes high for one full H1 cycle. Provided $\overline{\text{STRB}}$ is included in chip select decodes, this causes all devices to be disabled during this period. The next bank of devices is not enabled until $\overline{\text{STRB}}$ goes low again.

Bank switching is not required during writes since these cycles always exhibit an inherent one-half H1 cycle setup of address information before $\overline{\text{STRB}}$ goes low. Thus, when using bank switching for read/write devices, a minimum of half of one H1 cycle of address setup is provided for all accesses. Therefore, large amounts of memory can be implemented without wait states or extra hardware required for isolation between banks. Also, note that access time for cycles during bank switching is the same as that of cycles without bank switching, and accordingly, full speed accesses may still be accomplished within each bank.

The circuit shown in Figure 13-10 illustrates the use of bank switching with Cypress Semiconductor's CY7C185 25 ns 8kx8 CMOS static RAM. This circuit implements 32k 32-bit words of memory with full speed zero wait state accesses within each bank.

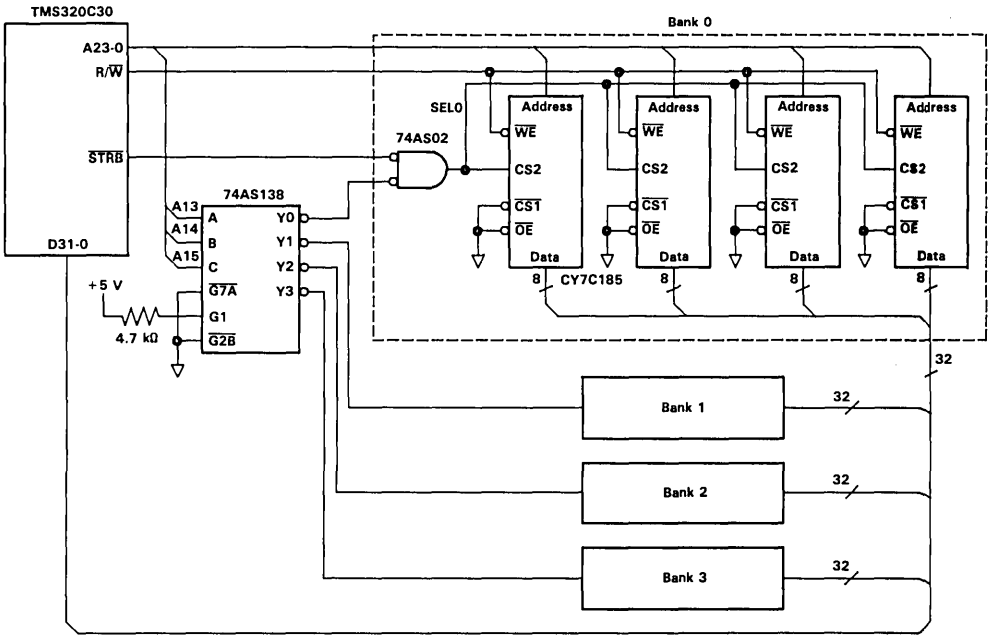


Figure 13-10. Bank Switching For Cyprus Semiconductors CY7C185

Each of the four banks in this circuit is selected using a decode of A15-A13 generated by the 74AS138. With the BNKCMPR register set to >0Bh, the banks will be selected on even 8k word boundaries starting at location zero in external memory space.

This circuit could not have been implemented without bank switching, since data output's turn-on and turn-off delays would have caused bus conflicts, and full speed accesses do not allow enough time for chip select decoding for the four banks. Here, the propagation delay of the 74AS138 is only involved during bank switches, where there is sufficient time between cycles to allow new chip selects to be decoded.

The timing of this circuit for read operations using bank switching is shown in Figure 13-11 With the BNKCMPR register set to >0Bh, when a bank

switch occurs, the bank address on address lines A23-A13, is updated during the extra H1 cycle while $\overline{STRB}$ is high. Then, after chip select decodes have stabilized, and the previously selected bank has disabled its outputs, $\overline{STRB}$ goes low for the next read cycle. Further accesses occur at full speed with the normal bus timings, as long as another bank switch is not necessary. Write cycles do not require bank switching due to the inherent address setup provided in their timings.

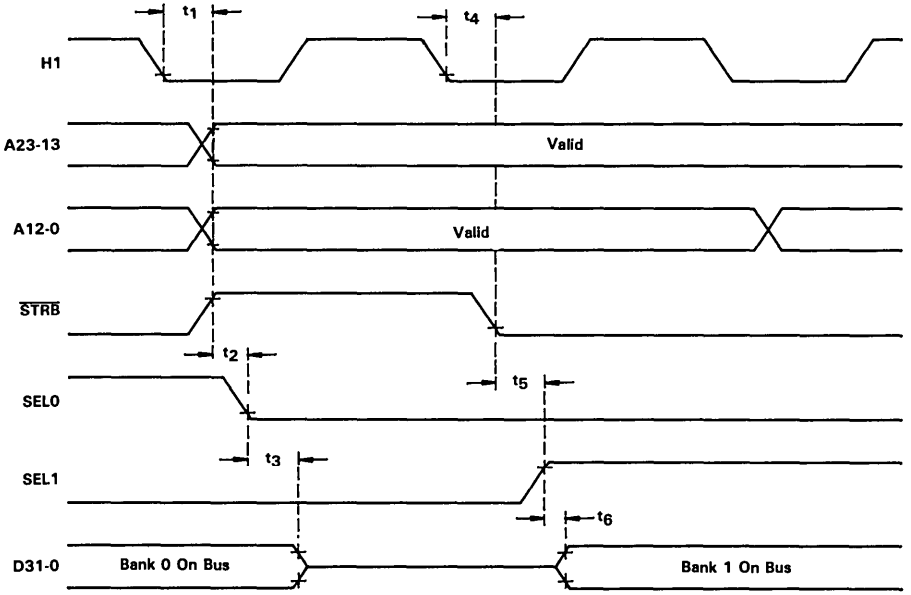


Figure 13-11. Timing For Read Operations Using Bank Switching

This timing is summarized in Table 13-1.

Table 13-1. Bank Switching Interface Timing

Time Interval	Event	Time Period
t1	H1 falling to address/$\overline{STRB}$ valid	10 ns
t2	$\overline{STRB}$ to select delay	4.5 ns
t3	Memory disable from select	15 ns
t4	H1 falling to $\overline{STRB}$	10 ns
t5	$\overline{STRB}$ to select delay	4.5 ns
t6	Memory output enable delay	3 ns

13.3 Expansion Bus Interface

The TMS320C30s expansion bus interface provides a second complete parallel bus which can be used to implement data transfers concurrently with, and independent of, operations on the primary bus. The expansion bus comprises two mutually exclusive interfaces controlled by the $\overline{\text{MSTRB}}$ and $\overline{\text{IOSTRB}}$ signals, respectively. These two signals are activated depending on what section of the memory space is accessed. This subsection discusses interface to the expansion bus using $\overline{\text{IOSTRB}}$; $\overline{\text{MSTRB}}$ cycles are identical in timing to primary bus cycles, and are discussed in Section 13.2.

Unlike thre primary bus, both read and write cycles on the I/O portion of the expansion bus are two H1 cycles in duration and exhibit the same timing. Thr XR/$\overline{\text{W}}$ signal is high for reads and low for writes. Since I/O accesses take two cycles, many peripherals that require wait states if interfaced either to the primary bus or using $\overline{\text{MSTRB}}$ may be used in a system without the need for wait states. Specifically, any devices with address access times greater than the 35 ns required by the primary bus but not less than 46 ns can be interfaced to the I/O bus without wait states.

A/D converters are one common DSP system component which often falls into this category. These devices are available in many speed ranges and with a variety of features, and while some may require one or more wait states on the I/O bus, others may be used at full speed.

One A/D converter that interfaces to the I/O bus without wait states and requires minimal additional logic is the ad 1332 from Analog Devices. Figure 13-12 illustrates an interface to this device.

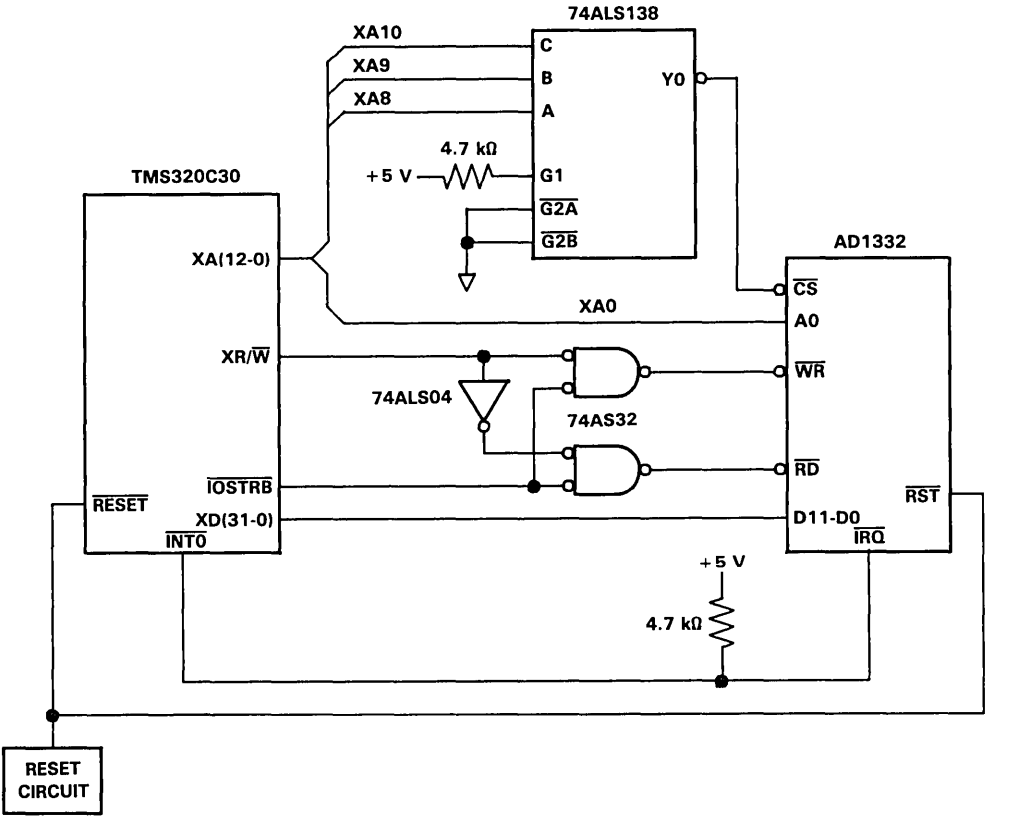


Figure 13-12. Expansion Bus Interface to A/D Converter

The interface uses a 74ALS138 to decode chip select for the converter. This configuration is shown assuming that other peripheral devices in the system also require chip select decodes. XA(8-10) are decoded to locate the converter at address 0804000h, which is the beginning of the I/O address space. Other peripherals may also use the outputs of the decoder, which generates chip selects in the I/O address space on 256 word boundaries.

XA0 is used to drive the single address line required in interfacing to the converter. This input selects between an internal 32-word FIFO buffer and the A/D's control/status register. Thus, the FIFO is located at address 0804000h and the control/status register is located at address 0804001h.

Since the converter requires $\overline{RD}$ and $\overline{WR}$ control signals rather than $\overline{WE}$ and $\overline{OE}$, random logic is used to generate these signals from $\overline{IOSTRB}$ and XR/$\overline{W}$. The converter's $\overline{IRQ}$ (Interrupt Request) output is used to alert the TMS320C30 to various conditions of converter status.

Figure 13-13 shows the timing for read and write operations between the TMS320C30 and the AD1332. Both operations are shown on the same timing diagram since, unlike the primary bus, only data bus timing and the state of XR/$\overline{W}$ differ between the two different types of cycles.

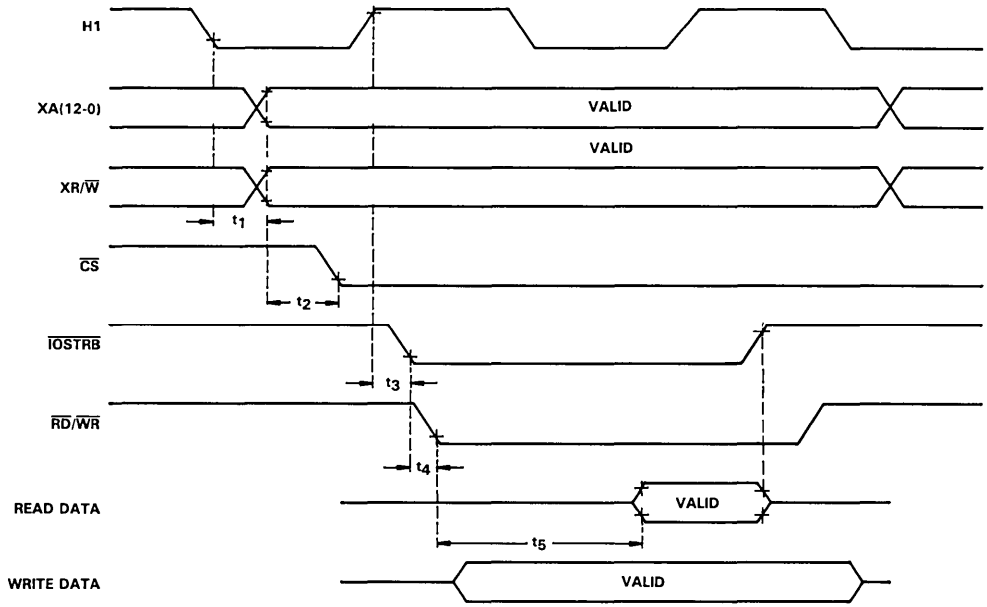


Figure 13-13. Timing of Expansion Bus Interface

In both cases, address and R/$\overline{W}$ are valid t1 = 10 ns after the falling edge of H1. After t2 = 17 ns, the propagation delay of the 74ALS138, the A/D converter's chip select goes low, selecting the device. Then, t3 = 10 ns after the rising edge of H1, $\overline{IOSTRB}$ goes low, and t4 = 5.8 ns following this, the $\overline{RD}$ or $\overline{WR}$ signal to the converter goes low, initiating either a read or write cycle, respectively.

For a read operation, the A/D converter provides data back to the TMS320C30 t4 + t5 = 30.8 ns after $\overline{RD}$ goes low. This satisfies the TMS320C30's requirement of having data valid 35 ns after $\overline{IOSTRB}$. For write operations, the A/D converter requires less than 5 ns of data setup and hold time with respect

to the rising edge of $\overline{WR}$. This is met with a high degree of margin by the TMS320C30.

It should be noted that for the AD1332's FIFO to be clocked properly, the $\overline{RD}$ signal must go high between accesses to the device. Therefore, although the AD1332 may be fast enough in some cases to be used at speeds approaching those of the primary bus, the $\overline{STRB}$ signal on the primary bus stays low for multiple consecutive read cycles. The I/O bus, therefore, is the preferable choice for interface to this device.

13.4 System Control Functions

There are several aspects of TMS320C30 system hardware design which are critical to overall system operation. These include such functions as clock and reset signal generation and interrupt control.

13.4.1 Clock Oscillator Circuitry

An input clock may be provided to the TMS320C30 either from an external clock input or by using the on-board oscillator. Unless special clock requirements exist, using the on-board oscillator is generally a convenient method of clock generation. This method requires few external components and can provide stable, reliable clock generation for the device.

Figure 13-14 shows a clock generator circuit using the internal oscillator. This circuit is designed to operate at 33.33 MHz and since crystals with fundamental oscillation frequencies of 30 MHz and above are not readily available, a parallel-resonant third-overtone circuit is used.

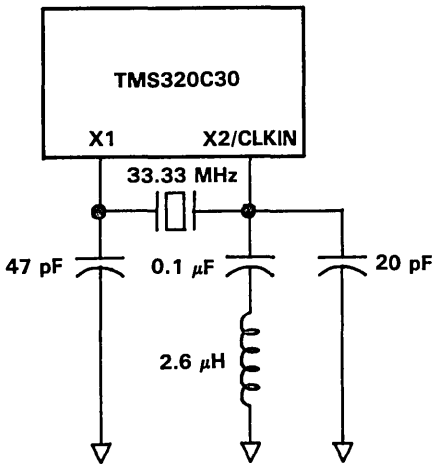


Figure 13-14. Crystal Oscillator Circuit

In a third-overtone oscillator, the crystal fundamental frequency must be attenuated so that oscillation is at the third harmonic. This is achieved with an LC circuit that filters out the fundamental, thus allowing oscillation at the third harmonic. The impedance of the LC network must be inductive at the crystal fundamental and capacitive at the third harmonic. The impedance of the LC circuit is given by:

$$z(\omega) = \frac{\frac{L}{C}}{J\left[\omega L - \frac{1}{\omega C}\right]} \qquad (3)$$

13

Therefore, the LC circuit has a pole at:

$$\omega_p = \frac{1}{\sqrt{LC}} \qquad (4)$$

At frequencies significantly lower than ω_p, the $1/(\omega C)$ term in (3) becomes the dominating term, while ωL can be neglected. This gives:

$$z(\omega) = j\omega L \qquad \text{for } \omega << \omega_p \qquad (5)$$

In (5), the LC circuit appears inductive at frequencies lower than ω_p. On the other hand, at frequencies much higher than ω_p, the ωL term is the dominant term in (3), and $1/(\omega C)$ can be neglected. This gives:

$$z(\omega) = \frac{1}{j\omega C} \qquad \text{for } \omega >> \omega_p \qquad (6)$$

The LC circuit in (6) appears increasingly capacitive as frequency increases above ω_p. This is shown in Figure 13-15, which is a plot of the magnitude of the impedance of the LC circuit of Figure 13-14 versus frequency.

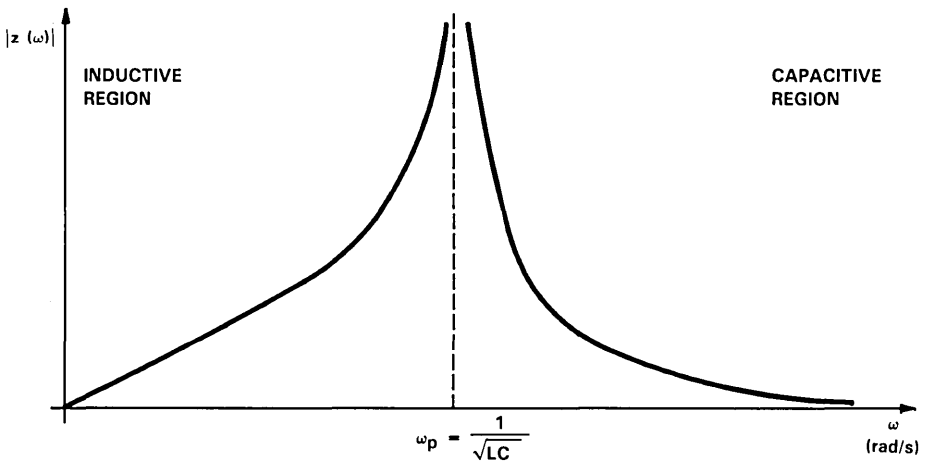


Figure 13-15. Magnitude of the Impedance of the Oscillator LC Network

13

Based on the discussion above, the design of the LC circuit proceeds as follows:

1) Choose the pole frequency ω_p approximately halfway between the crystal fundamental and the third harmonic.

2) The circuit now appears inductive at the fundamental frequency and capacitive at the third harmonic.

In the oscillator of Figure 13-14, choose ω_p = 22.2 MHz, which is approximately halfway between the fundamental and the third harmonic. Choose C = 20 pF. Then, using (4), L = 2.6 μH.

13.4.2 Reset Signal Generation

The reset input controls initialization of internal TMS320C30 logic and also causes execution of the system initialization software. For proper system initialization, the reset signal must be applied at least ten H1 cycles, i.e., 600 ns for a TMS320C30 operating at 33.33 MHz. Upon powerup, however, it can take 20 ms or more before the system oscillator reaches a stable operating state. Therefore, the powerup reset circuit should generate a low pulse on the reset line for 100 to 200 ms. Once a proper reset pulse has been applied, the processor fetches the reset vector from location zero which contains the address of the system initialization routine. Figure 13-16 shows a circuit which will generate an appropriate powerup reset signal.

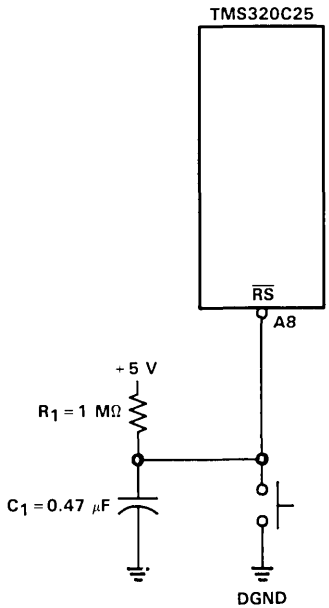


Figure 13-16. Reset Circuit

The voltage on the reset pin ($\overline{RESET}$) is controlled by the R_1C_1 network. After a reset, this voltage rises exponentially according to the time constant R_1C_1, as shown in Figure 13-17.

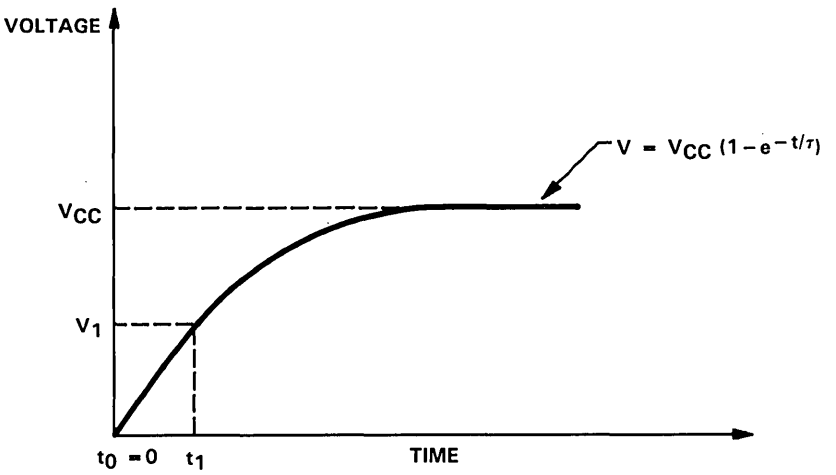


Figure 13-17. Voltage on the TMS320C30 Reset Pin

The duration of the low pulse on the reset pin is approximately t_1, which is the time it takes for the capacitor C_1 to be charged to 1.5 V. This is approximately the voltage at which the reset input switches from a logic 0 to a logic 1. The capacitor voltage is given by:

$$V = V_{CC}\left[1 - e^{-\frac{t}{\tau}}\right] \quad (7)$$

where $\tau = R_1C_1$ is the reset circuit time constant. Solving (7) for t gives:

$$t = -R_1C_1 \ln\left[1 - \frac{V}{V_{CC}}\right] \quad (8)$$

Setting the following:

$R_1 = 1\ M\Omega$

$C_1 = 0.47\ \mu F$

$V_{CC} = 5\ V$

$V = V_1 = 1.5\ V$

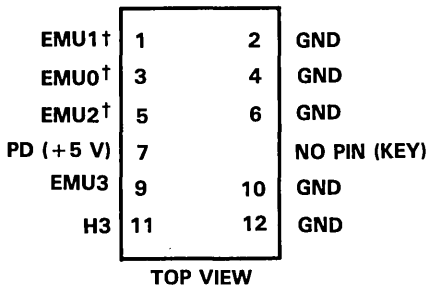
gives t = 167 ms. Therefore, the reset circuit of Figure 13-16 provides a low pulse of long enough duration to ensure the stabilization of the system oscillator upon powerup.

Note that if synchronization of multiple TMS320C30's is required, all processors should be provided with the same input clock and the same reset signal. After powerup, when the clock has stabilized, all processors may then be synchonized by generating a falling edge on the common reset signal. Since it is in fact the falling edge of reset that establishes synchronization, reset must be high for a period of time (at least ten H1 cycles) initially. Following the falling edge, reset should remain low for at least ten H1 cycles and then be driven high. This sequencing of reset may be accomplished using additional circuitry, based on either RC time delays or counters.

13.5 XDS1000 Target Design Considerations

The TMS320C30 Emulator is an Extended Development System (XDS1000), which has all the features necessary for full-speed emulation. The TMS320C30 uses a revolutionary technology to allow complete emulation via a serial scan path. If the user provides a 12 pin header on their target system, realtime emulation can be performed using the TMS320C30 device in their target system. Refer to Appendix B, Section B.1.4 for a complete description of the XDS1000.

To use the emulation connector of the XDS1000, the signals shown in Figure 13-18 should be provided to a 12 pin header (two rows of six pins) with pin 8 cut out to provide keying.



†These signals should always be pulled up with separate 20 kΩ resistors to +5 volts on the TMS320C30.

Figure 13-18. 12 Pin Header Signals

Signal Description:

EMU0	Emulation pin 0.	
EMU1	Emulation pin 1.	
EMU2	Emulation pin 2.	
EMU3	Emulation pin 3.	
H3	TMS320C30 H3	
PD	Presence detect.	It indicates that the cable is connected and target system is powered up. It should be tied to +5 volts in the target system.

Figure 13-19 is a diagram of the typical setup when using the emulation connection of the XDS1000.

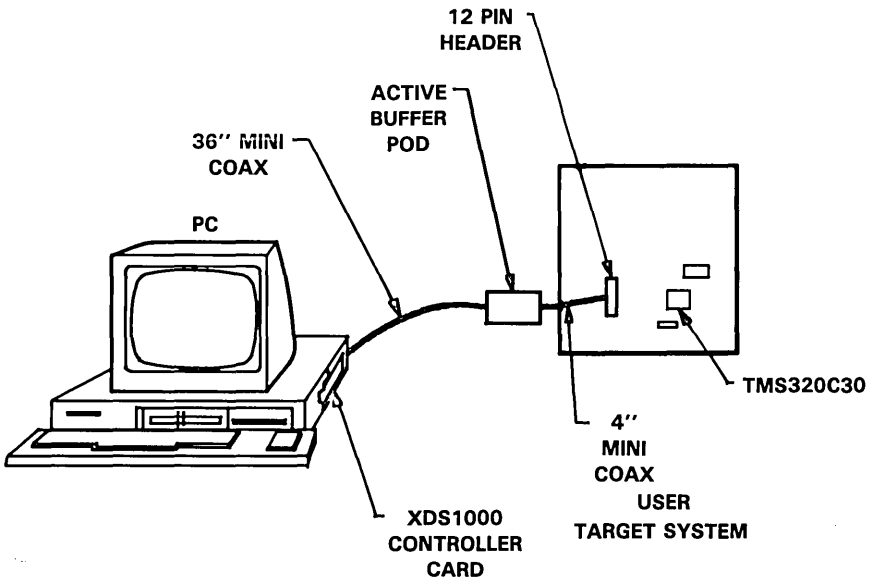


Figure 13-19. Typical Setup For Using the Emulation Connection of the XDS1000

For unbuffered signals, the distance between the TMS320C30 emulation pins (EMU0, EMU1, EMU2, EMU3, and H3) and the 12 pin header should be less than two inches. If the distance between the header and the TMS320C30 emulation pins is more than two inches but less than six inches, the EMU3 and H3 signals should be buffered. The buffer should be noninverting with a worst case propagation delay of 6.0 ns. For TMS320C30 emulation pins to 12 pin header distances greater than six inches, all emulation signals should be buffered. Recall that EMU0, EMU1, and EMU2 are inputs and EMU3 and H3 are outputs. The buffer should have the same characteristics as given above.

Introduction 1
Pinout and Signal Descriptions 2
Architectural Overview 3
CPU Registers, Memory, and Cache 4
Data Formats and Floating-Point Operation 5
Addressing 6
Program Flow Control 7
External Bus Operation 8
Peripherals 9
Pipeline Operation 10
Assembly Language Instructions 11
Software Applications 12
Hardware Applications 13
Appendices A-D

Appendix A

TMS320C30 Timing Specifications & Dimensions

This section provides timing specifications and dimensions for the TMS320C30 (third-generation TMS320) processor. In order to provide information in advance of the complete data sheet, this section is included. Characterization data on the TMS320C30 is still being collected. A complete data sheet with additional information will be available in the future. Please contact the local TI field sales office to obtain these data sheets.

Table A-1. Absolute Maximum Ratings Over Specified Temperature Range

Condition/Characteristic	Range
Supply voltage range, V_{DD}	-0.3 V to 7 V
Input voltage range	-0.3 V to 7 V
Output voltage range	-0.3 V to 7 V
Continuous power dissipation	2.0 W
Operating free-air temperature range	0°C to 70°C
Storage temperature range	-55°C to 150°C

Notes:

1) Stresses beyond those listed under 'Absolute Maximum Ratings' may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions beyond those indicated in the 'Recommended Operating Conditions' section of this specification is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

2) All voltage values are with respect to V_{SS}.

Table A-2. Recommended Operating Conditions

Operating Condition	Min	Nom	Max	Unit
V_{DD} Supply voltages (DVDD, etc.)	4.75	5	5.25	V
V_{SS} Supply voltages (CVSS, etc.)		0		V
V_{IH} High-level input voltage	2		V_{DD} + 0.3	V
V_{IL} Low-level input voltage	-0.3		0.8	V
I_{OH} High-level output current			300	µA
I_{OL} Low-level output current			2	mA
T Operating free-air temperature	0		70	°C

Table A-3. Electrical Characteristics Over Specified Free-Air Temperature Range

Electrical Characteristic	Min	Nom	Max	Unit
V_{OH} High-level output voltage (V_{DD} = Min,I_{OH} = Max)	2.4	3		V
V_{OL} Low-level output voltage (V_{DD} = Min,I_{OL} = Max)		0.3	0.6	V
I_Z Three-state current (V_{DD} = Max)	-20		20	µA.
I_I Input current (V_I = V_{SS} toV_{DD})	-10		10	µA
I_{CC} Supply current (T_A = 25°C,V_{DD}= Max, f_x = Max)		300		mA
C_I Input capacitance			15	pF
C_O Output capacitance			15	pF

Notes:

1) All typical values are at V_{DD} = 5 V, T_A = 25°C.

2) f_x is the input clock frequency. The maximum value is 33.3 MHz.

3) All input and output voltage levels are TTL compatible.

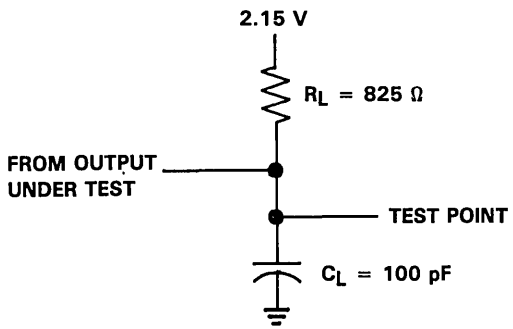


Figure A-1. Test Load Circuit

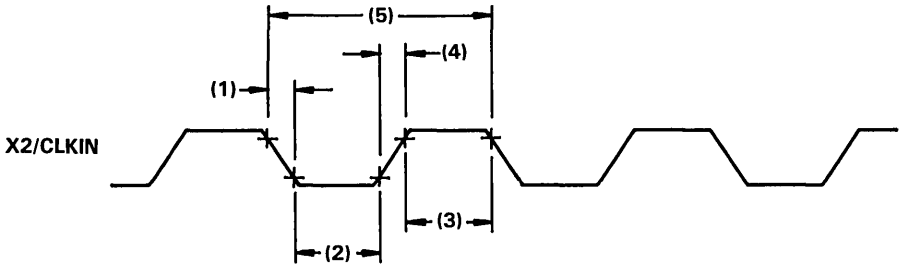


Figure A-2. X2/CLKIN Timing

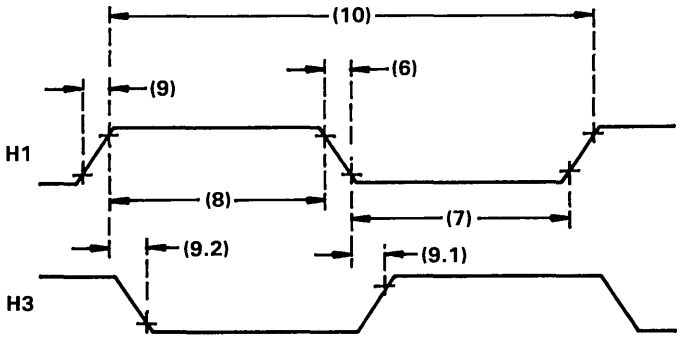


Figure A-3. H1/H3 Timing

Table A-4. Switching Characteristics for CLKIN, H1, and H3

No.	Name	Description	Min	Typ	Max	Unit
(1)	t_f(CI)	CLKIN fall time			5	ns
(2)	t_w(CIL)	CLKIN low pulse duration t_c(CI) = 30 ns	10			ns
(3)	t_w(CIH)	CLKIN high pulse duration t_c(CI) = 30 ns	10			ns
(4)	t_r(CI)	CLKIN rise time			5	ns
(5)	t_c(CI)	CLKIN cycle time	30			ns
(6)	t_f(H)	H1/H3 fall time	1		3	ns
(7)	t_w(HL)	H1/H3 low pulse duration	P - 6			ns
(8)	t_w(HH)	H1/H3 high pulse duration	P - 7			ns
(9)	t_r(H)	H1/H3 rise time	2		4	ns
(9.1)	t_d(HL - HH)	Delay from H1(H3) low to H3(H1) high			5	ns
(9.2)	t_d(HH - HL)	Delay from H1(H3) high to H3(H1) low			5	ns
(10)	t_c(H)	H1/H3 cycle time	60			ns

Note: P = t_c(CI)

Table A-5. Switching Characteristics for a memory ($\overline{(M)STRB}$ = 0) read

No.	Name	Description	Min	Typ	Max	Unit
(11)	t_d(H1L - (M)SL)	H1 low to $\overline{(M)STRB}$ low	0		10	ns
(12)	t_d(H1L - (M)SH)	H1 low to $\overline{(M)STRB}$ high	0		10	ns
(13)	t_d(H1H - (IO)RWL)	H1 high to (IO)R/$\overline{W}$ low	0		10	ns
(14)	t_d(H1L - (IO)A)	H1 low to (IO)A valid	0		10	ns
(15)	t_{su}((IO)D)R	(IO)D valid before H1 low (read)	15			ns
(16)	t_h((IO)D)R	(IO)D hold time after H1	0			ns
(17)	t_{su}((IO)RDY)	(IO)RDY valid before H1 high	8			ns
(18)	t_h((IO(RDY)	(IO)RDY hold time after H1 high	0			ns

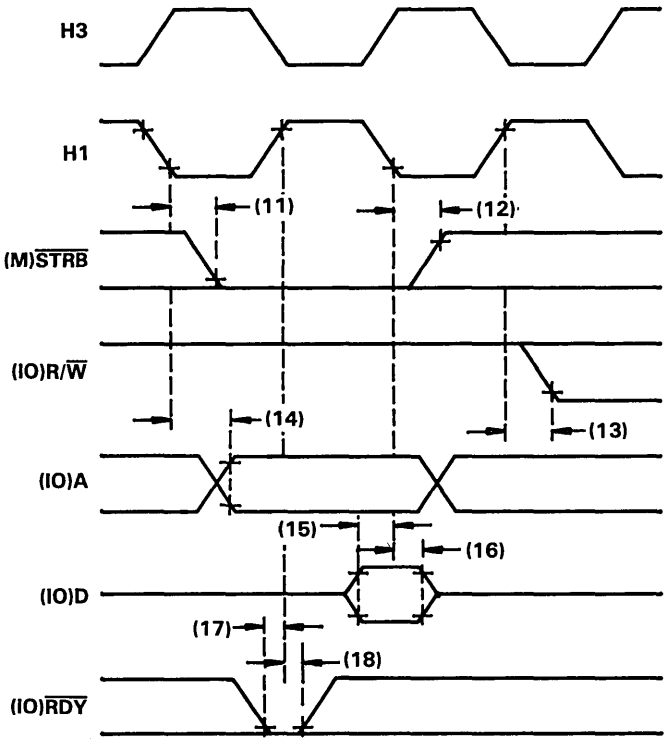


Figure A-4. Memory ($\overline{(M)STRB}$ = 0) Read

Table A-6. Switching Characteristics for a memory ($\overline{(M)STRB}$ = 0) Write

No.	Name	Description	Min	Typ	Max	Unit
(19)	t_d(H1H - (IO)RWH)	H1 high to (IO)R/$\overline{W}$ high			10	ns
(20)	t_v((IO)D)W	(IO)D valid after H1 low (write)			20	ns
(21)	t_h((IO)D)W	(IO)D hold time after H1 high (write)	0			ns

A-D

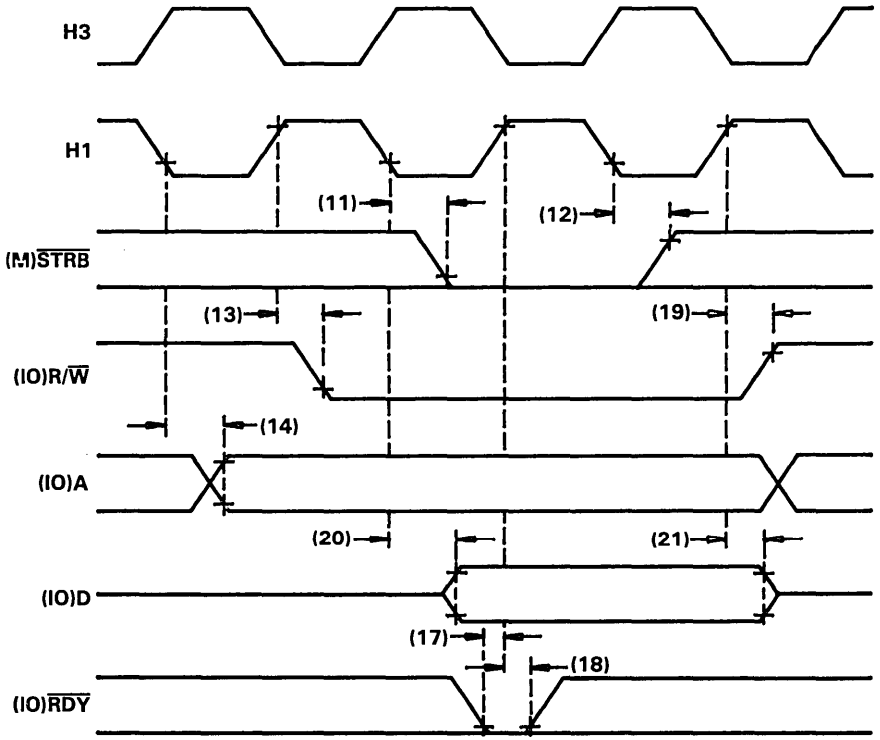


Figure A-5. Memory ($\overline{(M)STRB}$ = 0) Write

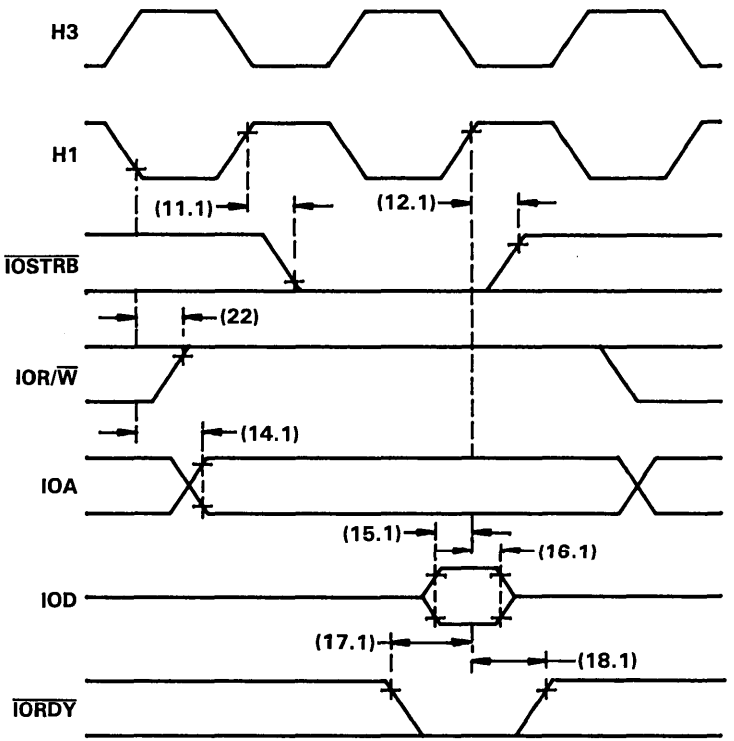


Figure A-6. Memory ($\overline{(M)STRB}$ = 0) Read

Table A-7. Switching Characteristics for a Memory ($\overline{IOSTRB}$ = 0) Read

No.	Name	Description	Min	Typ	Max	Unit
(11.1)	t_d(H1H - IOSL)	H1 high to $\overline{IOSTRB}$ low	0		10	ns
(12.1)	t_d(H1H - IOSH)	H1 high to $\overline{IOSTRB}$ high	0		10	ns
(22)	t_d(H1L - IORWH)	H1 low to IOR/$\overline{W}$ high	0		10	ns
(14.1)	t_d(H1L - IOA)	H1 low to IOA valid	0		10	ns
(15.1)	t_{su}(IOD)R	IOD valid before H1 high (read)	15			ns
(16.1)	t_h(IOD)R	IOD hold time after H1 high (read)	0			ns
(17.1)	t_{su}(IORDY)	IORDY valid before H1 high	8			ns
(18.1)	t_h(IORDY)	IORDY hold time after H1 high	0			ns

Table A-8. Switching Characteristics for a Memory ($\overline{IOSTRB}$ = 0) Write

No.	Name	Description	Min	Typ	Max	Unit
(23)	t_d(H1L - IORWL)	H1 low to (IO)R/$\overline{W}$ low	0		10	ns
(20.1)	t_v(IOD)W	IOD valid before H1 low (write)	15			ns
(21.1)	t_v(IOD)W	IOD hold time after H1 low (write)	0			ns

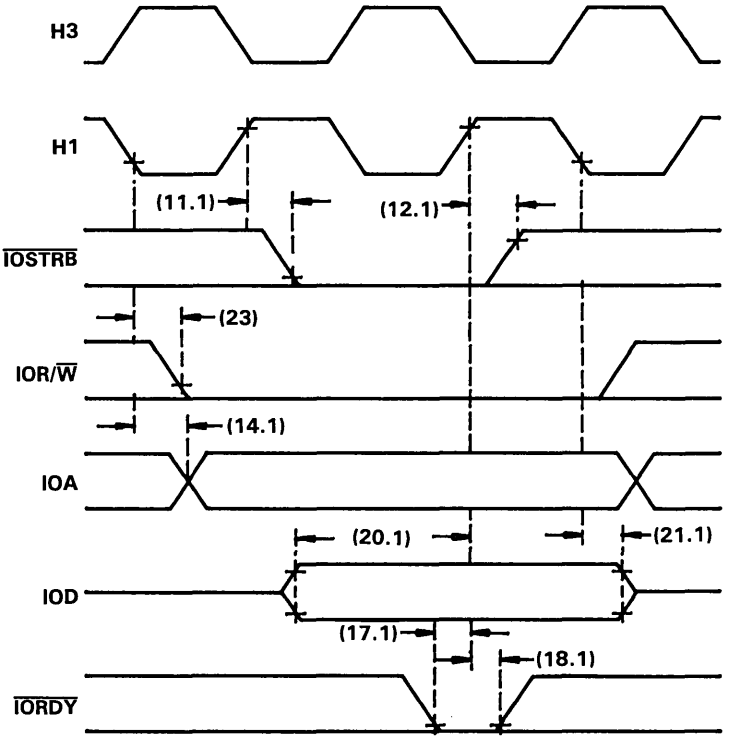


Figure A-7. Memory ($\overline{\text{IOSTRB}}$ = 0) Write

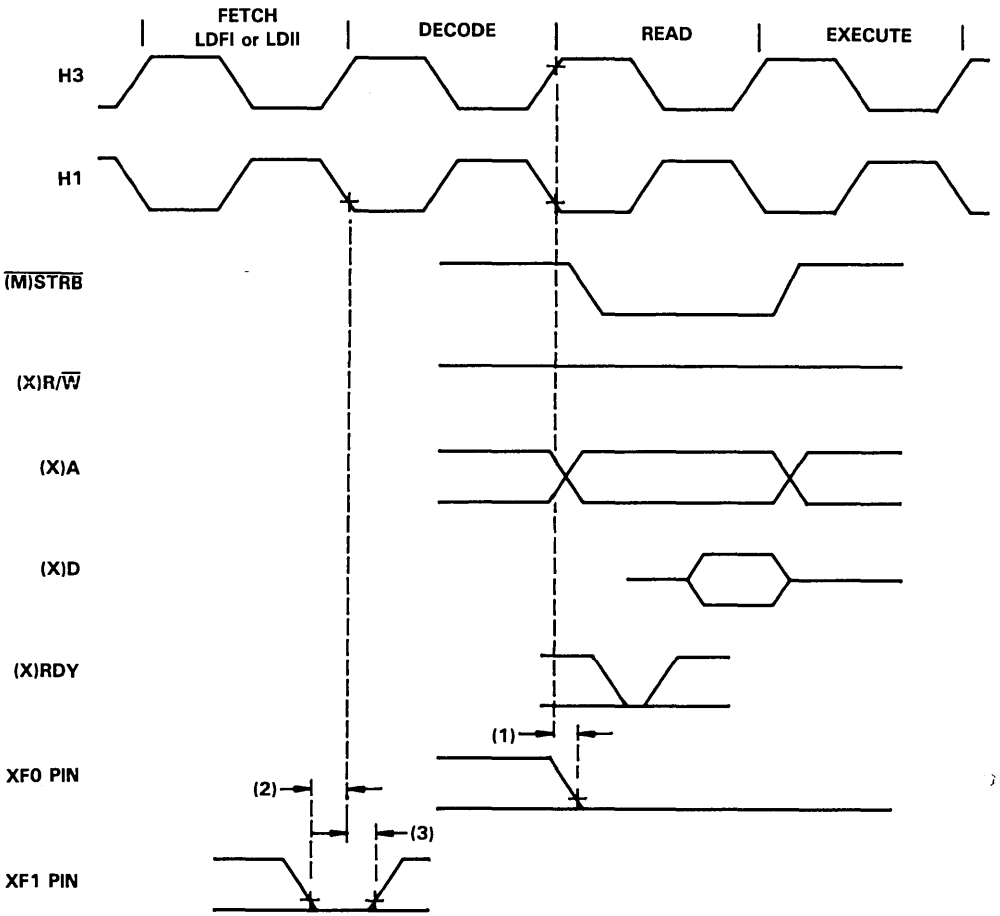


Figure A-8. Timing for XF0 and XF1 When Executing a LDFI or LDII

Table A-9. Information for Figure A-8

No.	Name	Description	Min	Typ	Max	Unit
(1)	t_d(H3H-XF0L)	H3 high to XF0 low			10	ns
(2)	t_{su}(XF1)	XF1 valid before H1 low	8			ns
(3)	t_h(XF1)	XF1 hold time after H1 low	0			ns

A-D

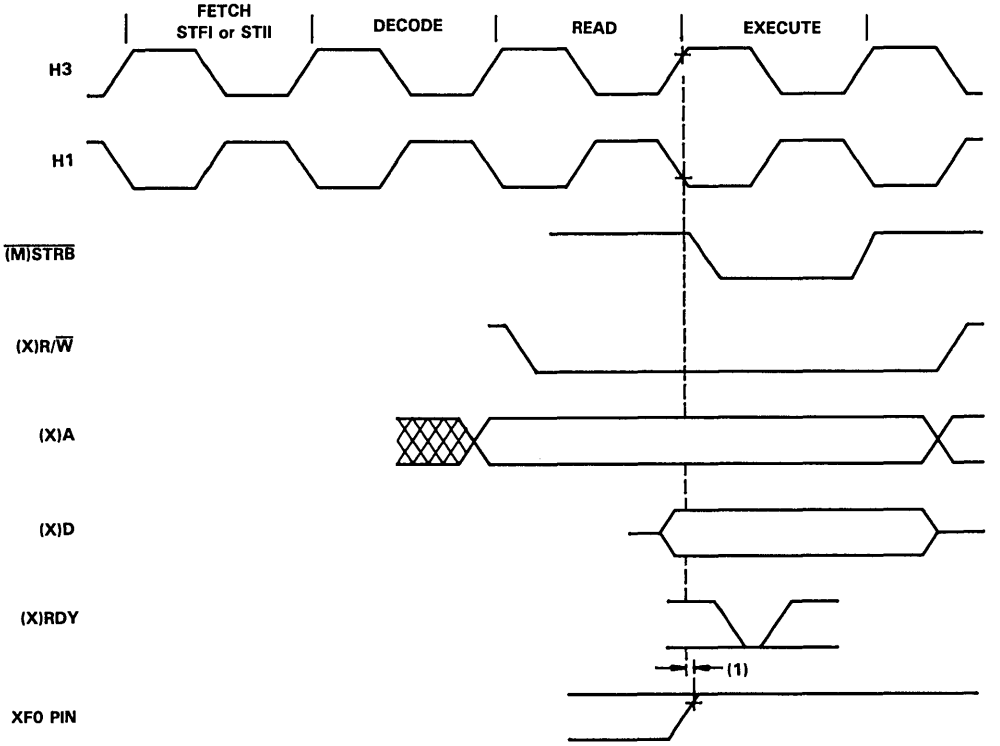


Figure A-9. Timing for XF0 When Executing a STFI or STII

Table A-10. Information for Figure A-9

No.	Name	Description	Min	Typ	Max	Unit
(1)	t_d(H3H-XF0H)	H3 high to XF0 high			10	ns

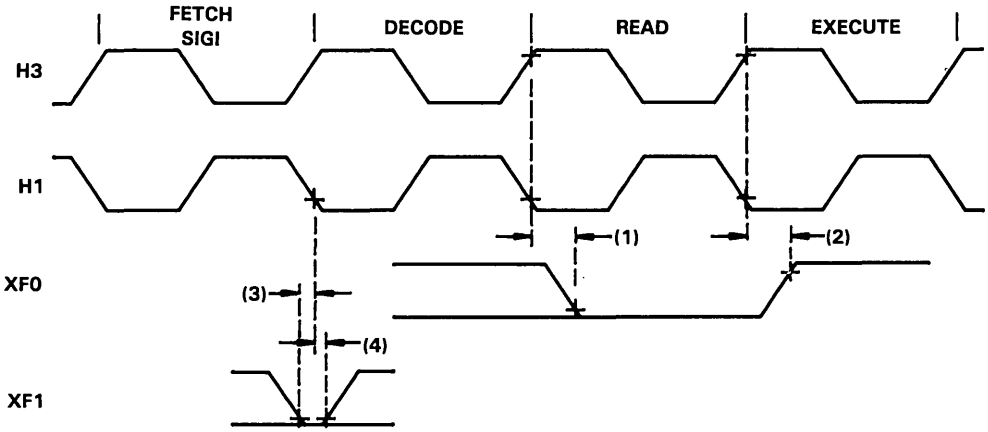


Figure A-10. Timing for XF0 and XF1 When Executing SIGI

Table A-11. Information for Figure A-10

No.	Name	Description	Min	Typ	Max	Unit
(1)	t_d(H3H-XF0L)	H3 high to XF0 low			10	ns
(2)	t_d(H3H-XF0H)	H3 high to XF0 high			10	ns
(3)	t_{su}(XF1)	XF1 valid before H1 low	8			ns
(4)	t_h(XF1)	XF1 hold time after H1 low	0			ns

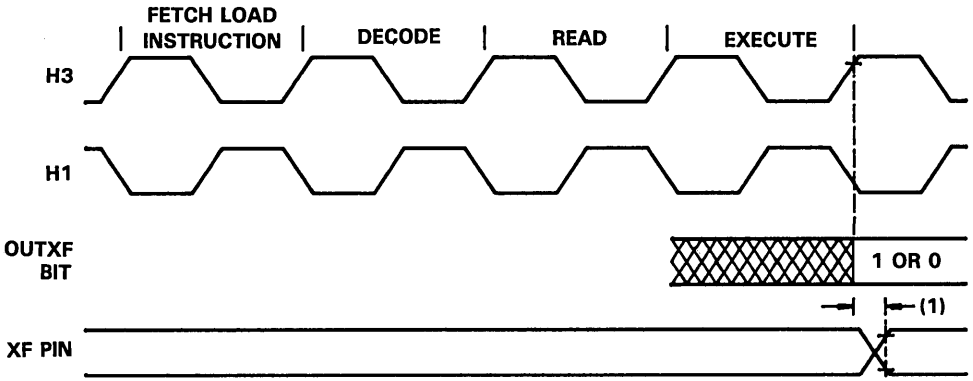


Figure A-11. Timing for Loading XF Register When Configured as an Output Pin

Table A-12. Information for Figure A-11

No.	Name	Description	Min	Typ	Max	Unit
(1)	t_v(H3H-XF)	H3 high to XF valid			10	ns

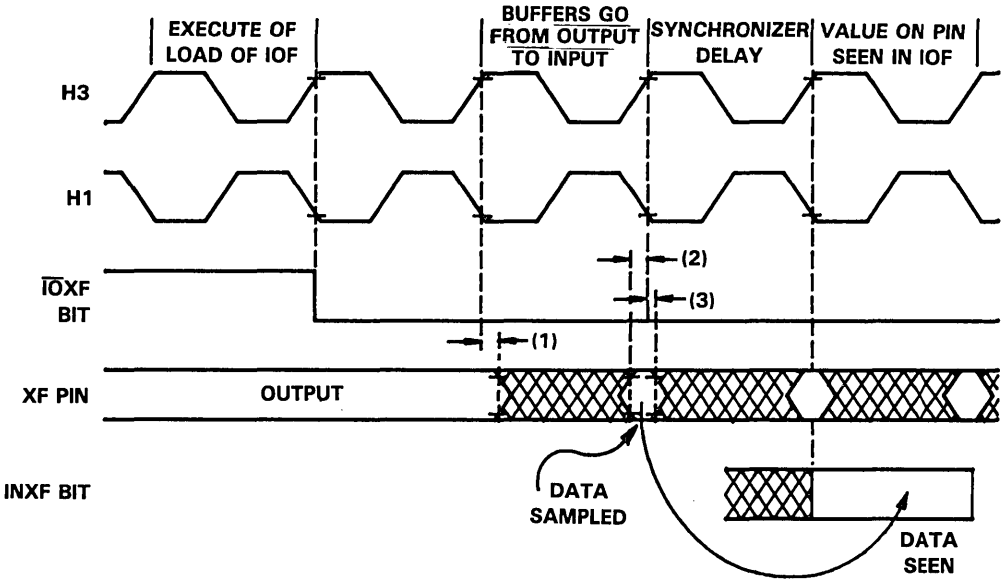


Figure A-12. Change of XF From Output to Input Mode

Table A-13. Information for Figure A-12

No.	Name	Description	Min	Typ	Max	Unit
(1)	t_d(H3H-XF0I)	H3 high to XF switching from output to input			15	ns
(2)	t_{su}(XF)	XF setup before H1 low	10			ns
(3)	t_h(XF)	XF hold before H1 low	0			ns

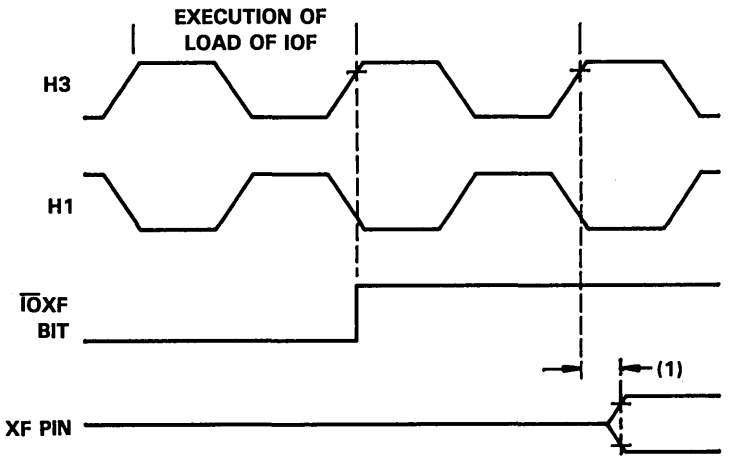
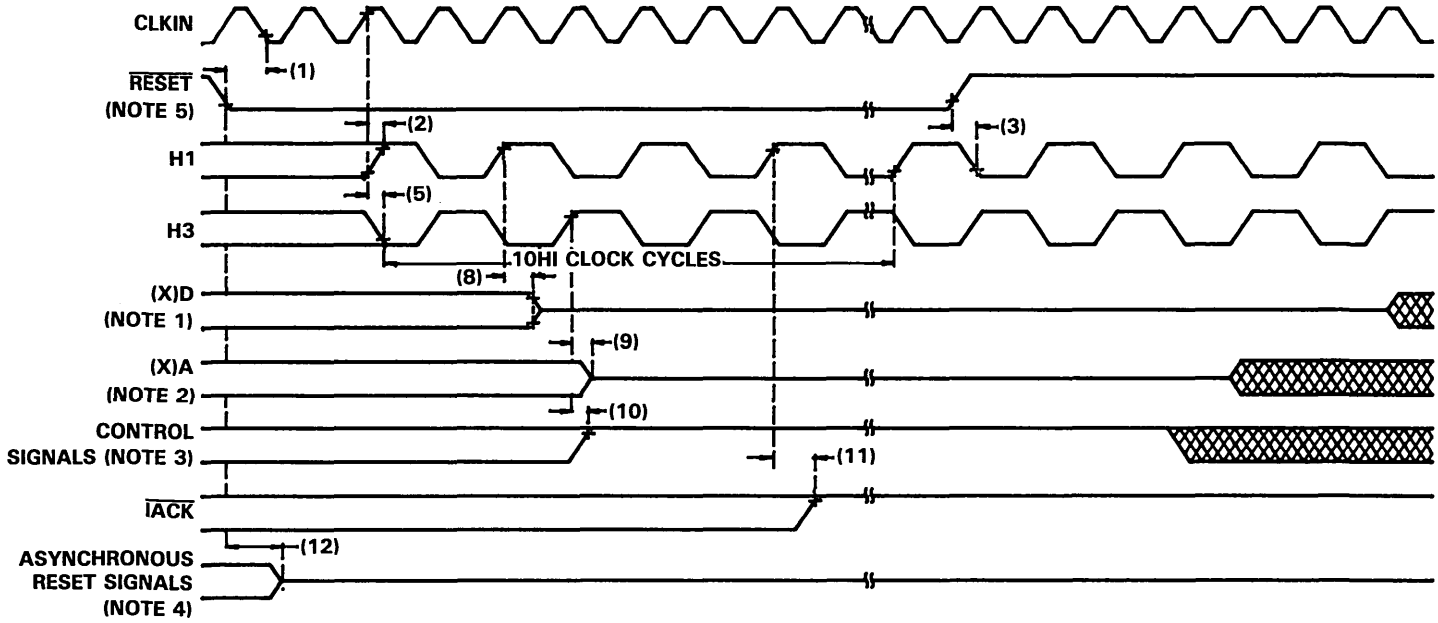


Figure A-13. Change of XF From Input to Output Mode

Table A-14. Information for Figure A-13

No.	Name	Description	Min	Typ	Max	Unit
(1)	t_d(H3H-XFIO)	H3 high to XF switching from input to output			10	ns



NOTES: 1. (X)D includes D(31-0) and XD(31-0).
2. X(A) includes A(23-0) and XA(12-0).
3. Control signals include R/$\overline{W}$, $\overline{STRB}$, XR/$\overline{W}$, $\overline{MSTRB}$, and $\overline{IOSTRB}$.
4. Asynchronously reset signals include XF1, XF0, CLKX0, DX0, FSX0, CLKR0, DR0, FSR0, CLKX1, DX1, FSX1, CLKR1, DR1, FSR1, TCLK0, and TCLK1.
5. $\overline{RESET}$ is an asynchronous input.

Figure A-14. $\overline{RESET}$ Timing

Table A-15. Information for Figure A-14

No.	Name	Description	Min	Typ	Max	Unit
(1)	$t_{su}(\overline{RESET})$	Setup for $\overline{RESET}$ before CLKIN low	10			ns
(2)	t_d(CLKINH-HIH)	CLKIN high to H1 high	0		15	ns
(3)	t_{su}(RESETH-HIL)	Setup for $\overline{RESET}$ high before H1 low and after 10 H1 clock cycles	10			ns
(5)	t_d(CLKINH-H3L)	CLKIN high to H3 low	0		15	ns
(8)	t_{dis}(H1H-XD)	H1 high to (X)D three state			15	ns
(9)	t_{dis}(H3H-XA)	H3 high to (X)A three state			10	ns
(10)	t_d(H3H-CONTROLH)	H3 high to control signals high			10	ns
(11)	t_d(H1H-IACKH)	H1 high to $\overline{IACK}$ high			10	ns
(12)	t_{dis}(RESETL-ASYNCH)	$\overline{RESET}$ low to asynchronously reset signals three state			15	ns

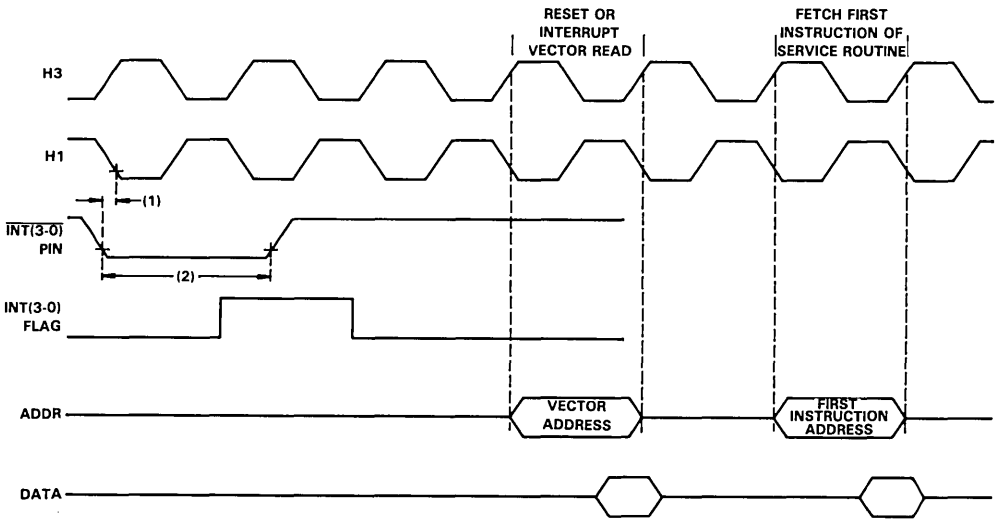


Figure A-15. $\overline{\text{RESET}}$ and $\overline{\text{INT}}$(3-0) Response Timing

Table A-16. Information for Figure A-15

No.	Name	Description	Min	Typ	Max	Unit
(1)	t_{su}(INT)	$\overline{\text{INT}}$(3-0) setup before H1 low	10			ns
(2)	t_w(INT)*Note 1*	Interrupt pulse width to guarantee one interrupt seen	P	1.5P	<2P	ns

Note 1: Interrupt pulse width must be at least 1 P wide to guarantee it will be seen. It must be less than 2 P wide to guarantee it will be responded to only once. The recommended pulse width is 1.5 P.

A-D

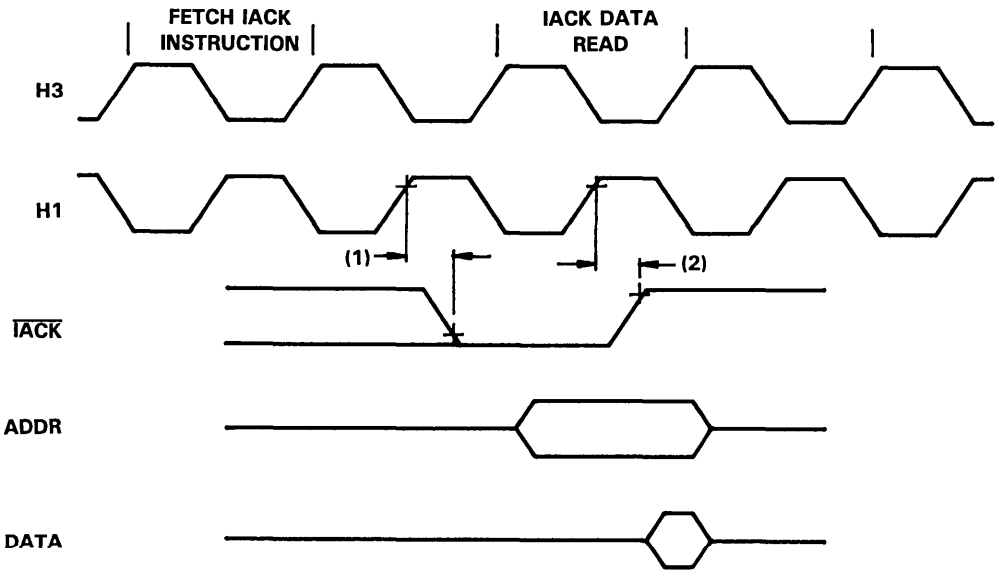


Figure A-16. $\overline{\text{IACK}}$ Timing

Table A-17. Information for Figure A-16

No.	Name	Description	Min	Typ	Max	Unit
(1)	t_d(H1H-IACKL)	H1 high to $\overline{\text{IACK}}$ low			10	ns
(2)	t_d(H1H-IACKH)	H1 high to $\overline{\text{IACK}}$ high during first cycle of IACK instruction data read			10	ns

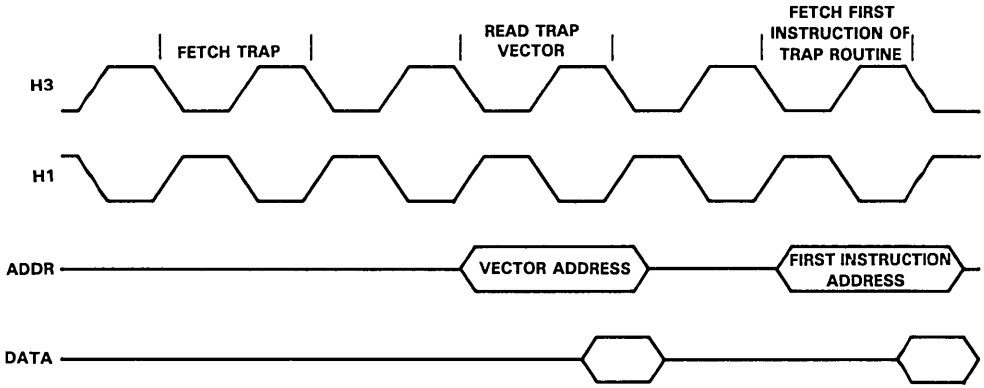
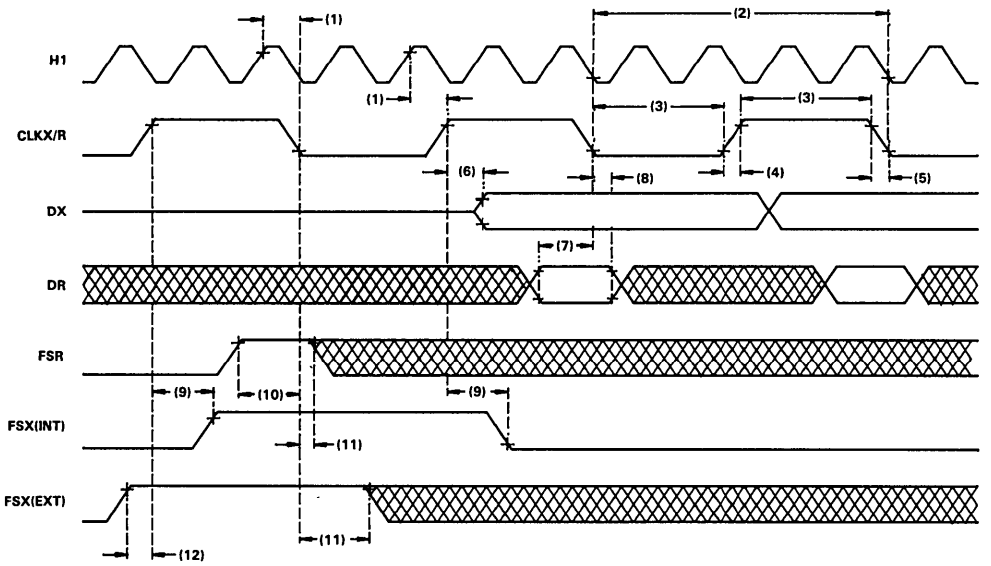
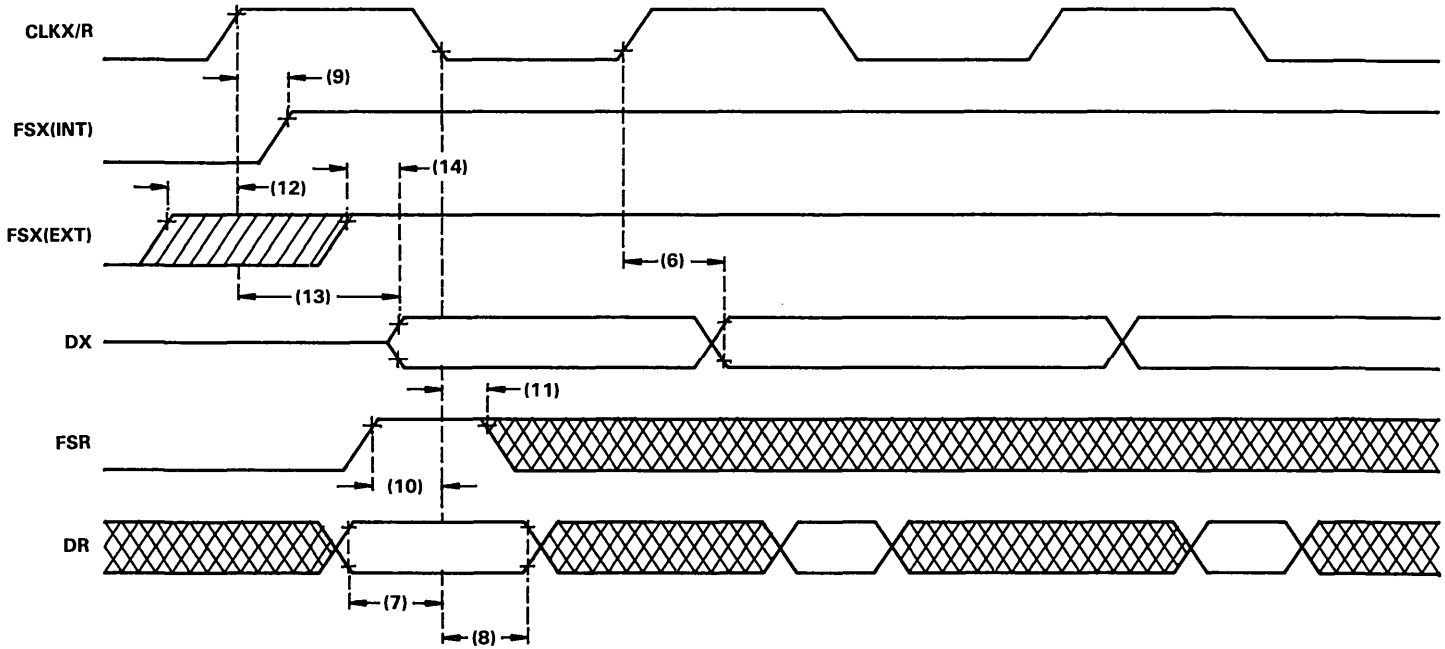


Figure A-17. TRAP Response Timing



NOTE: Timing diagrams show operation with CLKXP = CLKRP = FSXP = FSRP = 0.

Figure A-18. Fixed Data Rate Mode



NOTES: 1. Timing diagrams show operation with CLKXP=CLKRP=FSXP=FSRP=0.
2. Timings not expressly specified for variable data rate mode are the same as those for fixed data rate mode.

Figure A-19. Variable Data Rate Mode

Table A-18. Serial Port Timing as Shown in Figures A-18 and A-19

No.	Name	Description		Max	Min	Unit
(1)	t_d(H1-SCK)	H1 high to internal CLKX/R.		15		ns
(2)	t_c(SCK)	CLKX/R cycle time.	CLKX/R external		t_c(H)x2.6	ns
			CLKX/R internal	t_c(H)x2[34]	t_c(H)x2	
(3)	t_w(SCK)	CLKX/R high/ low pulsewidth	CLKX/R external		t_c(H)+5	ns
			CLKX/R internal	t_c(SCK)/2	[t_c(SCK)/2]-15	
(4)	t_r(SCK)	CLKX/R rise time.		8		ns
(5)	t_f(SCK)	CLKX/R fall time		8		ns
(6)	t_d(DX)	CLKX to DX valid.	CLKX external	35		ns
			CLKX internal	20		
(7)	t_{su}(DR)	DR setup before CLKR.	CLKR external		10	ns
			CLKR internal		25	
(8)	t_h(DR)	DR hold from CLKR.	CLKR external		10	ns
			CLKR internal		-5	
(9)	t_d(FSX)	CLKX to internal FSX.	CLKX external	32		ns
			CLKX internal	17		
(10)	t_{su}(FSR)	FSR setup before CLKR.	CLKR external		10	ns
			CLKR internal		10	
(11)	t_h(FS)	FSX/R input hold from CLKX/R.	CLKX/R external		10	ns
			CLKX/R internal		-5	
(12)	t_{su}(FSX)	External FSX setup before CLKX.	CLKX external	[t_c(CLKX)/2]-10	-[t_c(H)-8]	ns
			CLKX internal	t_c(CLKX)/2	-[t_c(H)-21]	
(13)	t_d(CH-DX)V	CLKX to first DX bit, FSX precedes CLKX.	CLKX external	36		ns
			CLKX internal	21		
(14)	t_d(FSX-DX)V	FSX to first DX bit, CLKX precedes FSX.		36		ns

A-D

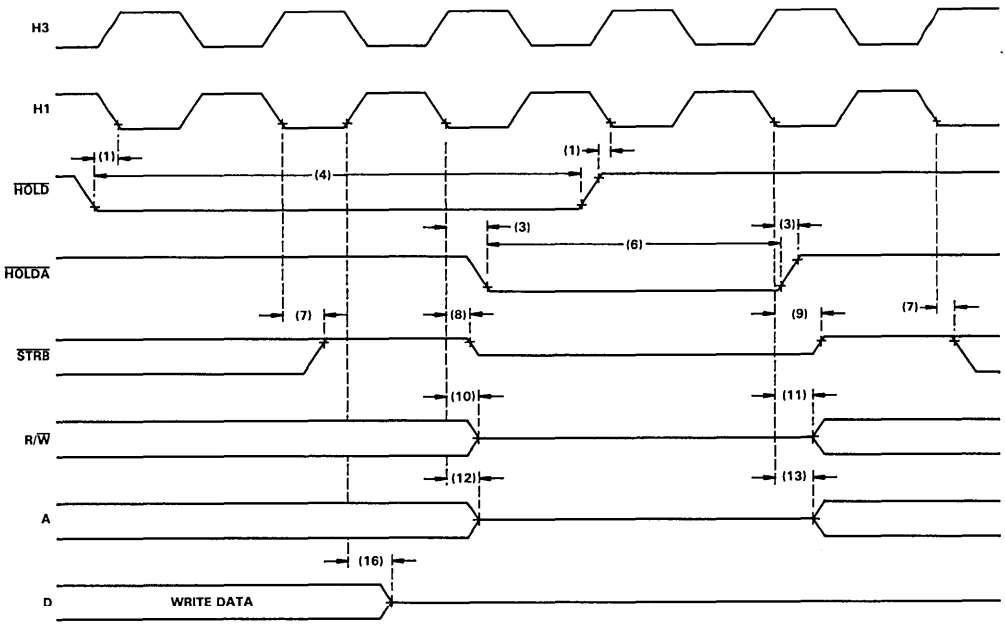


Figure A-20. $\overline{\text{HOLD}}/\overline{\text{HOLDA}}$ Timing

Table A-19. Information for Figure A-20

No.	Name	Description	Min	Max	Unit
(1)	$t_{su}(\overline{\text{HOLD}})$	$\overline{\text{HOLD}}$ valid before H1 low	15		ns
(3)	$t_v(\overline{\text{HOLD}})$	$\overline{\text{HOLD}}$ valid after H1 low	0	10	ns
(4)	$t_w(\overline{\text{HOLD}})$	$\overline{\text{HOLD}}$ low width	2		H1 cycles
(6)	$t_w(\overline{\text{HOLDA}})$	$\overline{\text{HOLDA}}$ low width	1		H1 cycle
(7)	t_d(H1L-SH)H	H1 low to $\overline{\text{STRB}}$ high for a $\overline{\text{HOLD}}$	0	10	n s
(8)	t_{dis}(H1L-S)	H1 low to $\overline{\text{STRB}}$ three state	0	10	ns
(9)	t_{en}(H1L-S)	H1 low to $\overline{\text{STRB}}$ active	0	10	ns
(10)	t_{dis}(H1L-RW)	H1 low to R/$\overline{\text{W}}$ three state	0	10	ns
(11)	t_{en}(H1L-RW)	H1 low to R/$\overline{\text{W}}$ active	0	10	ns
(12)	t_{dis}(H1L-A)	H1 low to address three state	0	10	ns
(13)	t_{en}(H1L-A)	H1 low to address valid	0	10	ns
(16)	t_{dis}(H1H-D)	H1 high to data three state	0	10	ns

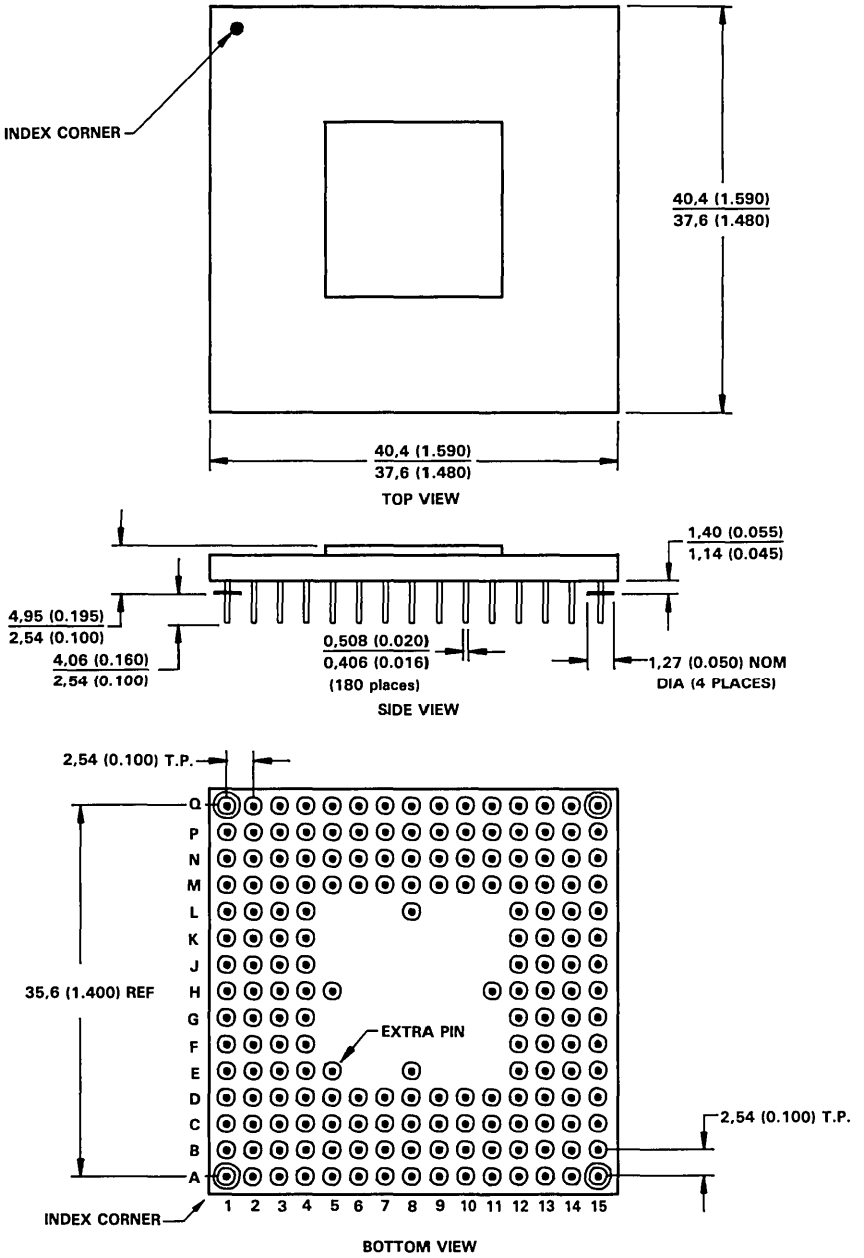


Figure A-21. TMS320C30 180 Pin PGA Dimensions

Appendix B

Development Support/Part Order Information

This section provides development support information, device part numbers, and support tool ordering information for the TMS320C30 (third-generation TMS320) processor. Figure B-1 shows the software and hardware development tools available and the development environment for the TMS320C30.

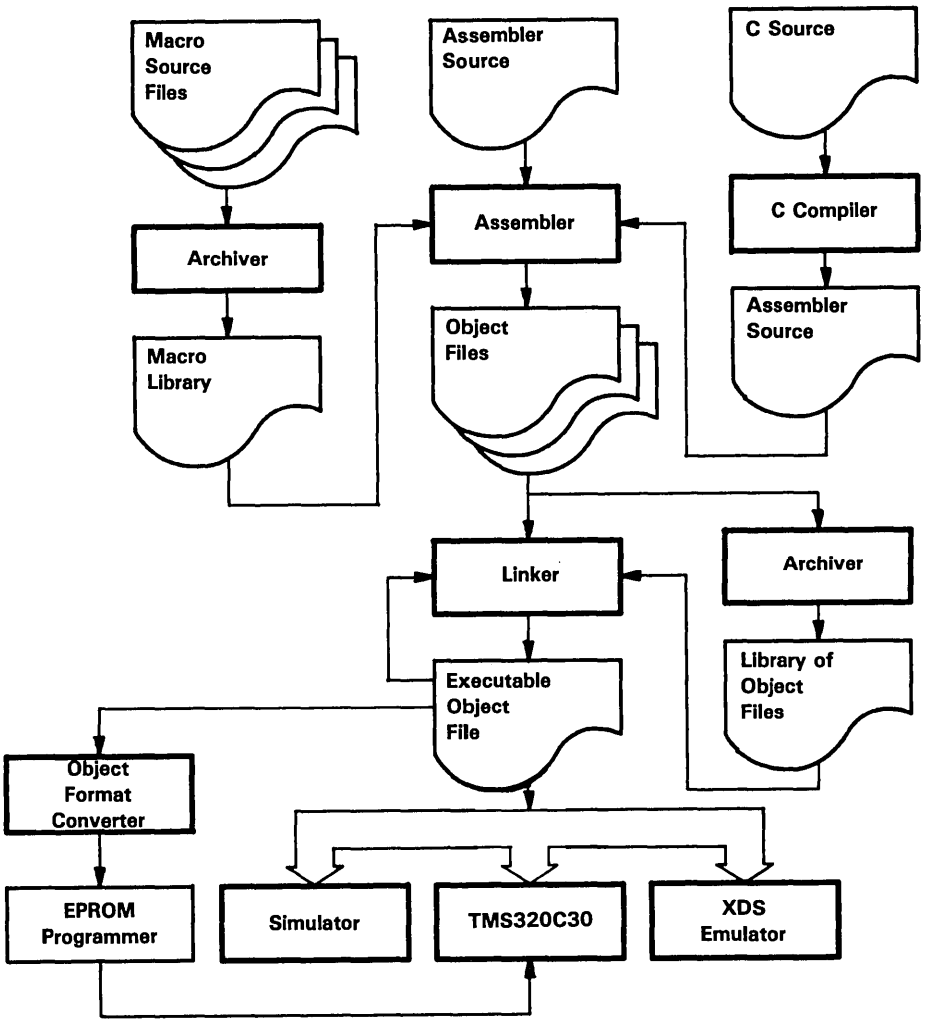


Figure B-1. TMS320C30 Development Environment

Extensive documentation, including data sheets, user's guides, and application reports, is available to support DSP design. A series of DSP textbooks has been published both by Prentice-Hall and John Wiley and Sons to support research and education. Other support includes a technical support hotline (713-274-2320) and a bulletin board service (713-274-2323). TI's Regional Technology Centers (RTCs) provide hands-on workshops and design services.

Many third-parties and consultants with DSP expertise can assist in various application areas. TMS320C30 Algorithm Development Packages will be available from multiple third-parties and consultants in the near future. Subscribe to the DSP newsletter "Details on Signal Processing" for up to date information on new products and services from third-parties and consultants. Call TI's Customer Response Center at (800) 232-3200 to subscribe to the newsletter. Contact the nearest TI field sales office for support tool availability or further details (see list of sales offices and distributors at end of book).

The major topics discussed in this section are listed below.

- TMS320C30 Development Support (Section B.1 on page B-4)
  - Macro Assembler/Linker
  - C Compiler
  - Simulator
  - Extended Development System (XDS1000)
  - TMS320 DSP Hotline/Bulletin Board Service
- TMS320C30 Part Order Information (Section B.2 on page B-12)
  - Device part numbers
  - Software and hardware support tools part numbers
  - Device and support tool prefix designators
  - Device nomenclature

B.1 TMS320C30 Development Support

Texas Instruments offers extensive development support and complete documentation with the TMS320C30 (third-generation) digital signal processor. Tools are provided for the TMS320C30 to evaluate the performance of the processor, develop algorithm implementations, and fully integrate the design's software and hardware modules. Development operations are performed with the TMS320C30 Macro Assembler/Linker, C Compiler, Simulator, and Emulator (Extended Development System - XDS1000).

A description and key features for each TMS320C30 development support tool is provided in the following subsections. For ordering information, see Section B.2.

B.1.1 Macro Assembler/Linker

The TMS320C30 Macro Assembler/Linker is a software tool that converts source mnemonics to executable object code.

The following key features distinguish the TMS320C30 Macro Assembler/Linker:

- Macro capabilities and library functions
- Conditional assembly
- Relocatable modules
- Complete error diagnostics
- Symbol table and cross reference

The TMS320C30 Macro Assembler/Linker is shipped with four programs to address specific needs. They are:

1) The assembler
2) The archiver
3) The linker
4) The object format converter

These programs and their functionality are described in the following paragraphs.

- The **assembler** translates assembly language source files into machine language object files. Source files can contain instructions, assembler directives, and macro directives. Assembler directives can be used to control various aspects of the assembly process, such as the source listing format, data alignment, and section content.

- The **archiver** allows collection of a group of files into a single archive file. For example, several macros can be collected together into a macro library. The assembler will search through the library and use the members that are called as macros by the source file. It is also possible to use the archiver to collect a group of object files into an object library. The linker will include the members in the library that resolve external references during the link.

- The **linker** combines object files into a single executable object module. As it creates the executable module, it performs relocation and resolves external references. The linker accepts relocatable object files (created by the assembler) as input. It also accepts archive library members and output modules created by a previous linker run. Linker directives allow combining of file sections, binding of sections or symbols to addresses, and defining of global symbols.

- The main purpose of this development process is to produce a module that can be executed in a system that contains a **TMS320C30 device or the software or hardware development tools.** (Note that only *linked* files can be executed).

- Most EPROM programmers do not accept assembler/linker files as input. The **object format converter** converts the object file into Intel, Tektronix, or TI-tagged object format. The converted file can be downloaded to an EPROM programmer. This EPROM code can then be executed on the TMS320C30 device.

Refer to Figure B-1 for a diagram of the development environment when using the Assembler/Linker.

The macro assembler/linker is currently available for PC/MS-DOS, VAX VMS, SUN-3 UNIX , and VAX ULTRIX operating systems.

B.1.2 C Compiler

The optimizing C compiler is a full implementation of the standard Kernighan and Ritchie C. The compiler accepts a digital signal processing program written in C language. It outputs TMS320C30 assembly language source code which is then processed by the assembler where the TMS320C30 mnemonics are converted to object code.

This high-level language compiler allows time-critical routines written in assembly language to be called from within the C program. The converse is also available; assembly routines may call C functions. The output of the compiler can be edited prior to assembly/link to further optimize the performance of the code. The compiler supports the insertion of assembly language code into C source code. The result is a compiler that allows the relative amounts of high-level programming and assembly language code to be tailored according to the application. Refer back to Figure B-1 for a diagram of the development environment when using the C compiler.

The compiler is currently available for PC/MS-DOS, VAX VMS, SUN-3 UNIX, and VAX ULTRIX operating systems. The assembler/linker is included with the shipment of the TMS320C30 C compiler. The output of this assembler/linker can be downloaded and used with the simulator, XDS, or PROM programmer.

B.1.3 Simulator

The TMS320C30 Simulator is a software program that simulates operation of the TMS320C30.

The following features highlight simulator capability for effective TMS320C30 software development:

- Simulates the entire TMS320C30 digital signal processor instruction set
- Simulates the key TMS320C30 peripheral features (DMA, timers, and serial port)
- Command entry from either menu-driven keystrokes (menu mode) or from a batch file (line mode)
- Help menus for all screen modes
- Standard interface can be user customized
- Simulation parameters quickly stored/retrieved from files to facilitate preparation for individual sessions
- Reverse assembly allows editing and re-assembly of source statements
- Memory can be displayed (at same time) as:
  - hexadecimal 32-bit values
  - assembled source
- Execution modes include:
  - single/multiple instruction count
  - single/multiple cycle count
  - *until* condition is met
  - *while* condition exists
  - *for* set loop count
  - unrestricted run with halt by key input
- Easy to define trace expressions
- Trace execution with display choices of:
  - designated expression values
  - cache registers
  - instruction pipeline for easy optimization of code
- Breakpoint conditions include: :
  - address read
  - address write
  - address read or write
  - address execute
  - expression valid
- Simulates cache utilization
- Cycle counting
  - display the number of clock cycles in single step or run mode
  - external memory can be configured with wait states for accurate cycle counting

The simulator allows verification and monitoring of the state of the processor. Simulation speed is on the order of thousands of instructions per second(VAX/VMS, VAX/ULTRIX, and SUN-3 UNIX) or hundreds of instructions per second (PC/MS-DOS).

The simulators use TMS320C30 object code, produced by the Macro Assembler/Linker. Input and output files may be associated with the port addresses of the I/O instructions in order to simulate I/O devices connected to the processor. Before initiating program execution, breakpoints may be set, and the trace format defined.

During program execution, the internal registers and memory of the simulated TMS320C30 are modified as each instruction is interpreted by the host computer. Execution is suspended when one of the following conditions exists:

1) A breakpoint or error is encountered.

2) Execution is halted.

Once program execution is suspended, the internal registers and both program and data memories can be inspected and/or modified. The trace memory can also be displayed. A record of the simulation session can be maintained in a journal file, so that it can be re-executed to regain the same machine state during another simulation session.

The user interface in the simulator is identical to that in the XDS. See Figure B-2 for an example of the user interface.

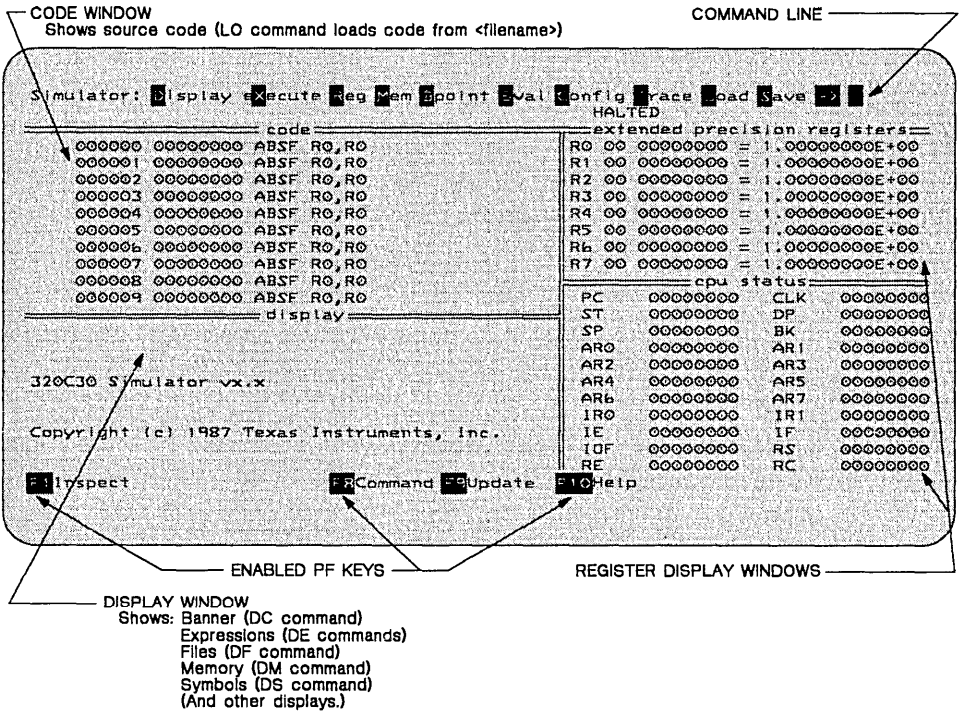


Figure B-2. TMS320C30 Simulator User Interface

The simulator is currently available from TI for PC/MS-DOS, VAX VMS, and VAX ULTRIX operating systems. A SUN-3 UNIX version of the simulator can be purchased from a third party: Spectron Microsystems Inc. This version is the same as TI's simulator for the PC/MS-DOS, VAX VMS, and VAX ULTRIX. Contact Spectron at (805 967-0503) for more information.

B.1.4 TMS320C30 Emulator - Extended Development System (XDS1000)

The TMS320C30 Emulator (XDS1000) is a user-friendly system that has all the features necessary for full-speed emulation to debug hardware, software, or integrate the software with the hardware. Some of the XDS1000's features include:

- Full-speed execution and monitoring out of the customers target system via a 12 pin target connector
- Software breakpoint
- Software trace
- Software timing capabilities
- Single-step execution
- Inspect/modify registers and program/data memory
- Upload/download capabilities to/from data/program memory
- Windowed user interface similar to the TMS320C30 simulator

Full-speed execution and monitoring of the customers target system via a 12 pin target connector has the advantage of using a serial scan path to give access to the internal registers as well as internal and external memory of the device. Since execution is out of the TMS320C30 located in the target system, there is no timing difference during emulation.

Software breakpoints means the program can be stopped on a specific address. When the program counter reaches the designated breakpoint address, the emulator will halt execution and allow the user to observe the status of the TMS320C30 (i.e., inspect memory or registers). Software trace allows viewing of the TMS320C30's state when a breakpoint is reached. This information can be saved to a file for future analysis. Software timing permits keeping track of clock ticks between breakpoints or while program single stepping.

The XDS1000 consists of two full-size PC-XT/AT cards. One card is the TMS320C30 XDS1000 Controller Card, the other is the TMS320C30 XDS1000 Development Board.

The TMS320C30 XDS1000 Controller Card is responsible for interpreting commands sent from the PC and converting those commands into appropriate signal sequences to control the TMS320C30 in the user's target system.

The TMS320C30 XDS1000 Development Board is a predefined target system that contains:

- A TMS320C30 device
- 16K x 32-bits full-speed (zero wait state) SRAM on the primary bus
- Two selectable banks of 8K x 32-bits full-speed (zero wait state) SRAM on the expansion bus

See figure Figure B-3 for a visual representation of the TMS320C30 XDS1000's development environment.

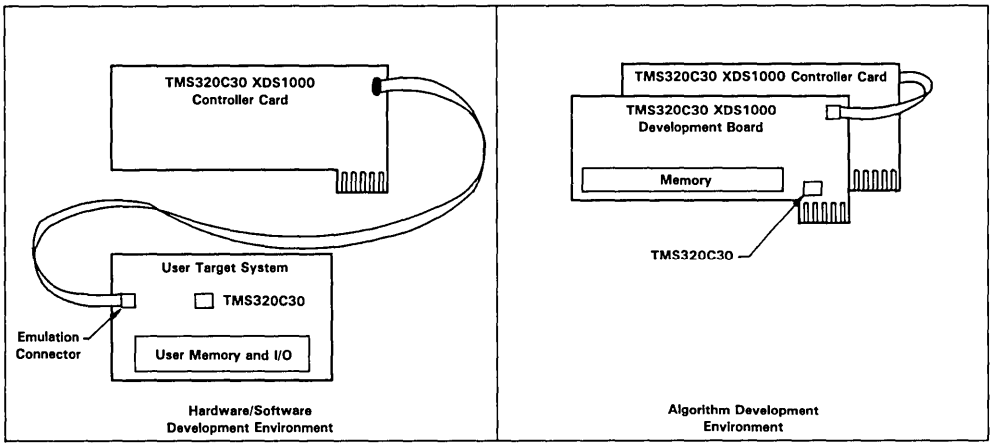


Figure B-3. TMS320C30 XDS1000 Development Environment

This figure shows the two environments in which the TMS320C30 XDS1000 can operate:

1) The hardware/software development configures the TMS320C30 XDS1000 and the user's target system in the emulator mode. Section 13.5 of this document shows the 12-pin header or emulator connector necessary for the user's target system to work with the TMS320C30 XDS1000.

2) The algorithm development environment allows the user to debug his software before the user's target system is built. In this configuration, the TMS320C30 XDS1000 Development Board can be used in place of the user's target system. In this mode, code can be downloaded into the

A-D

memory on the TMS320C30 XDS1000 Development Board and execute at full speed.

To use the TMS320C30 XDS1000, the following equipment is required:

- IBM PC-XT/AT compatible
- Two and one-half eight-bit slots for the PC-AT, three full-size eight-bit slots for the PC-XT
- A minimum of 640K bytes of memory in the PC
- PC/MS DOS rev 2.0 or later

In summary, the TMS320C30 XDS1000 is a full-speed emulator that comes with a pre built target system for early design development. The TMS320C30 XDS1000 can help debug hardware in realtime, debug software in realtime, and integrate the hardware and software together.

B.1.5 TMS320 DSP Hotline/Bulletin Board Service

The TMS320 group at Texas Instruments provides a DSP Hotline to answer TMS320 technical questions such as device problems, development tools, documentation, upgrades, and new TMS320 products. The hotline is open five days a week from 8:00 AM to 6:00 PM Central Time. The phone number is (713) 274-2320. For pricing and availability of TMS320 devices and development tools, contact the nearest TI sales office. To order literature, call the Customer Response Center (CRC) at (800) 232-3200.

The TMS320 DSP Bulletin Board Service is a telephone-line computer bulletin board that provides access to information pertaining to TMS320 devices. Specification updates for current or new TMS320 devices and development tools are communicated via the bulletin board as the information becomes available. The Bulletin Board Service can be accessed by dialing (713) 274-2323 with a 300, 1200, or 2400-bps modem.

The bulletin board contains TMS320C30 source code from Section 12 of the TMS320C30 Users Guide as well as development tool and silicon revisions and enhancements. The bulletin board also provides new DSP application software as it becomes available. See the *TMS320 Family Development Support Reference Guide* for further information on how to access the bulletin board.

B.2 TMS320C30 Part Order Information

This section provides the device and support tool part numbers. Table B-1 lists the part numbers for the TMS320C30, and Table B-2 gives ordering information for TMS320C30 hardware and software support tools. A discussion of the TMS320 family device and development support tool prefix designators is included to assist in understanding the TMS320 product numbering system.

Table B-1. TMS320C30 Digital Signal Processor Part Numbers

DEVICE	TECHNOLOGY	OPERATING FREQUENCY	PACKAGE TYPE	TYPICAL DISSIPATION
†TMX320C30GBH	1.0-µm CMOS	33 MHz	Ceramic 180-pin PGA	1.5 W

†Military version planned; contact nearest sales office for availability.

Table B-2. TMS320C30 Support Tool Part Numbers

TOOL DESCRIPTION	OPERATING SYSTEM	PART NUMBER
SOFTWARE		
Macro Assembler/Linker	VAX VMS PC/MS-DOS SUN-3 UNIX * VAX ULTRIX	TMDX3243250-08 TMDX3243850-02 TMDX3243550-08 TMDX3243260-08
C Compiler & Macro Assembler/ Linker	VAX VMS PC/MS DOS SUN-3 UNIX * VAX ULTRIX	TMDX3243255-08 TMDX3243855-02 TMDX3243555-08 TMDX3243265-08
Simulator	VAX VMS PC/MS-DOS SUN-3 UNIX * VAX ULTRIX	TMDX3243251-08 TMDX3243851-02 Offered by Spectron Inc. (805) 967-0503 TMDX3243261-08
HARDWARE		
XDS1000	PC/MS-DOS	TMDX3261030

* Please note SUN UNIX support for TMS320C30 software tools is for the 68000 family based SUN-3 series workstations. These tools are **NOT SUPPORTED** on the SUN-4 series machines that use the SPARC processor, or the SUN-386i series of workstations.

A-D

B.2.1 Device and Development Support Tool Prefix Designators

To assist the user in understanding the stages in the product development cycle, Texas Instruments assigns prefix designators in the part number nomenclature. A device prefix designator has three options: TMX, TMP, and TMS, and a development support tool prefix designator has two options: TMDX and TMDS. These prefixes are representative of the evolutionary stages of product development from engineering prototypes (TMX/TMDX) through fully qualified production devices (TMS/TMDS). This development flow is defined below.

Device Development Evolutionary Flow:

TMX Experimental device that is not necessarily representative of the final device's electrical specifications.

TMP Final silicon die that conforms to the device's electrical specifications but has not completed quality and reliability verification.

TMS Fully qualified production device.

Support Tool Development Evolutionary Flow:

TMDX Development support product that has not yet completed Texas Instruments internal qualification testing.

TMDS Fully qualified development support product.

TMX and TMP devices and TMDX development support tools are shipped with the following disclaimer:

"Developmental product is intended for internal evaluation purposes."

> **Note:**
>
> Texas Instruments recommends that prototype devices (TMX or TMP) not be used in production systems since their expected end-use failure rate is undefined but predicted to be greater than standard qualified production devices.

TMS devices and TMDS development support tools have been fully characterized and the quality and reliability of the device has been fully demonstrated. Texas Instruments standard warranty applies.

B.2.2 Device Nomenclature

In addition to the prefix, the device nomenclature includes a suffix that follows the device family name. This suffix indicates the package type (e.g., N, FN, or GB) and temperature range (e.g., L). Figure B-4 provides a legend for reading the complete device name for any TMS320 family member.

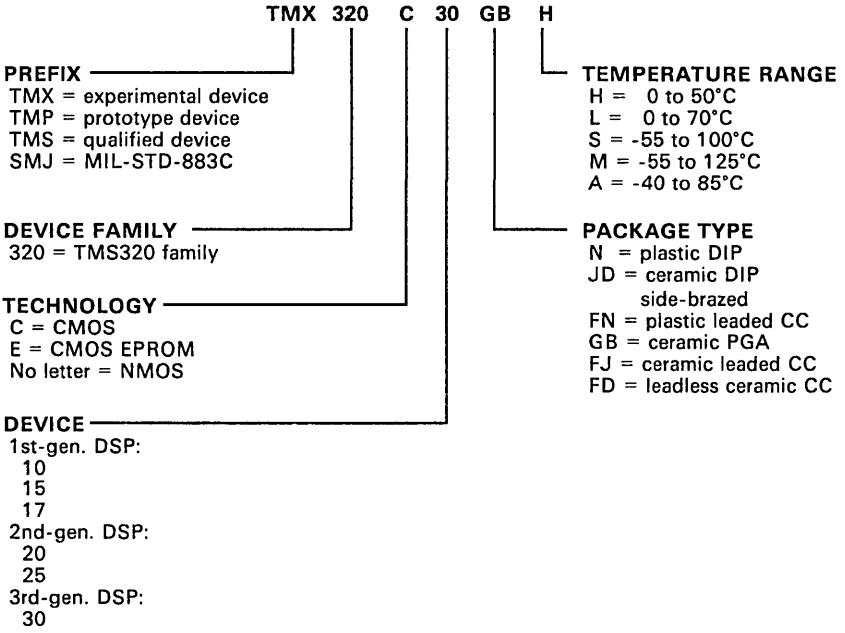


Figure B-4. TMS320 Device Nomenclature

Appendix C

Instruction Opcodes

The opcode fields for all the TMS320C30 instructions are shown in Table C-1. Bits in the table marked with a hyphen are defined in the individual instruction description (see Section 11). Table C-1 along with the instruction descriptions fully define the instruction words. The opcodes are listed in numerical order.

Table C-1. TMS320C30 Instruction Opcodes

INSTRUCTION	31	30	29	28	27	26	25	24	23
ABSF	0	0	0	0	0	0	0	0	0
ABSI	0	0	0	0	0	0	0	0	1
ADDC	0	0	0	0	0	0	0	1	0
ADDF	0	0	0	0	0	0	0	1	1
ADDI	0	0	0	0	0	0	1	0	0
AND	0	0	0	0	0	0	1	0	1
ANDN	0	0	0	0	0	0	1	1	0
ASH	0	0	0	0	0	0	1	1	1
CMPF	0	0	0	0	0	1	0	0	0
CMPI	0	0	0	0	0	1	0	0	1
FIX	0	0	0	0	0	1	0	1	0
FLOAT	0	0	0	0	0	1	0	1	1
IDLE	0	0	0	0	0	1	1	0	0
LDE	0	0	0	0	0	1	1	0	1
LDF	0	0	0	0	0	1	1	1	0
LDFI	0	0	0	0	0	1	1	1	1
LDI	0	0	0	0	1	0	0	0	0
LDII	0	0	0	0	1	0	0	0	1
LDM	0	0	0	0	1	0	0	1	0
LSH	0	0	0	0	1	0	0	1	1
MPYF	0	0	0	0	1	0	1	0	0
MPYI	0	0	0	0	1	0	1	0	1
NEGB	0	0	0	0	1	0	1	1	0
NEGF	0	0	0	0	1	0	1	1	1
NEGI	0	0	0	0	1	1	0	0	0
NOP	0	0	0	0	1	1	0	0	1

Table C-1. TMS320C30 Instruction Opcodes (Continued)

INSTRUCTION	31	30	29	28	27	26	25	24	23
NORM	0	0	0	0	1	1	0	1	0
NOT	0	0	0	0	1	1	0	1	1
POP	0	0	0	0	1	1	1	0	0
POPF	0	0	0	0	1	1	1	0	1
PUSH	0	0	0	0	1	1	1	1	0
PUSHF	0	0	0	0	1	1	1	1	1
OR	0	0	0	1	0	0	0	0	0
RND	0	0	0	1	0	0	0	1	0
ROL	0	0	0	1	0	0	0	1	1
ROLC	0	0	0	1	0	0	1	0	0
ROR	0	0	0	1	0	0	1	0	1
RORC	0	0	0	1	0	0	1	1	0
RPTS	0	0	0	1	0	0	1	1	1
STF	0	0	0	1	0	1	0	0	0
STFI	0	0	0	1	0	1	0	0	1
STI	0	0	0	1	0	1	0	1	0
STII	0	0	0	1	0	1	0	1	1
SIGI	0	0	0	1	0	1	1	0	0
SUBB	0	0	0	1	0	1	1	0	1
SUBC	0	0	0	1	0	1	1	1	0
SUBF	0	0	0	1	0	1	1	1	1
SUBI	0	0	0	1	1	0	0	0	0
SUBRB	0	0	0	1	1	0	0	0	0
SUBRF	0	0	0	1	1	0	0	1	0
SUBRI	0	0	0	1	1	0	0	1	1
TSTB	0	0	0	1	1	0	1	0	0
XOR	0	0	0	1	1	0	1	0	1
IACK	0	0	0	1	1	0	1	1	0
ADDC3	0	0	1	0	0	0	0	0	0
ADDF3	0	0	1	0	0	0	0	0	1
ADDI3	0	0	1	0	0	0	0	1	0
AND3	0	0	1	0	0	0	0	1	1
ANDN3	0	0	1	0	0	0	1	0	0
ASH3	0	0	1	0	0	0	1	0	1
CMPF3	0	0	1	0	0	0	1	1	0
CMPI3	0	0	1	0	0	0	1	1	1
LSH3	0	0	1	0	0	1	0	0	0

Table C-1. TMS320C30 Instruction Opcodes (Continued)

INSTRUCTION	31	30	29	28	27	26	25	24	23
MPYF3	0	0	1	0	0	1	0	0	1
MPYI3	0	0	1	0	0	1	0	1	0
OR3	0	0	1	0	0	1	0	1	1
SUBB3	0	0	1	0	0	1	1	0	0
SUBF3	0	0	1	0	0	1	1	0	1
SUBI3	0	0	1	0	0	1	1	1	0
TSTB3	0	0	1	0	0	1	1	1	1
XOR3	0	0	1	0	1	0	0	0	0
LDF*cond*	0	1	0	0	-	-	-	-	-
LDI*cond*	0	1	0	1	-	-	-	-	-
BR(D)†	0	1	1	0	0	0	0	-	-
CALL	0	1	1	0	0	0	1	-	-
RPTB	0	1	1	0	0	1	0	-	-
SWI	0	1	1	0	0	1	1	-	-
B*cond*(D)†	0	1	1	0	1	0	-	-	-
DB*cond*(D)†	0	1	1	0	1	1	-	-	-
CALL*cond*	0	1	1	1	0	0	-	-	-
TRAP*cond*	0	1	1	1	0	1	0	-	-
RETI*cond*	0	1	1	1	1	0	0	0	0
RETS*cond*	0	1	1	1	1	0	0	0	1
MPYF3||ADDF3	1 1 1 1	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 1 1	0 1 0 1	- - - -
MPYF3||SUBF3	1 1 1 1	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	1 1 1 1	0 0 1 1	0 1 0 1	- - - -
MPYI3||ADDI3	1 1 1 1	0 0 0 0	0 0 0 0	0 0 0 0	1 1 1 1	0 0 0 0	0 0 1 1	0 1 0 1	- - - -
MPYI3||SUBI3	1 1 1 1	0 0 0 0	0 0 0 0	0 0 0 0	1 1 1 1	1 1 1 1	0 0 1 1	0 1 0 1	- - - -
STF||STF	1	1	0	0	0	0	0	-	-
STI||STI	1	1	0	0	0	0	1	-	-
LDF||LDF	1	1	0	0	0	1	0	-	-

† Opcode same for standard and delayed instructions.

A-D

Table C-1. TMS320C30 Instruction Opcodes (Concluded)

INSTRUCTION	31	30	29	28	27	26	25	24	23
LDI||LDI	1	1	0	0	0	1	1	-	-
ABSF||STF	1	1	0	0	1	0	0	-	-
ABSI||STI	1	1	0	0	1	0	1	-	-
ADDF3||STF	1	1	0	0	1	1	0	-	-
ADDI3||STI	1	1	0	0	1	1	1	-	-
AND3||STI	1	1	0	1	0	0	0	-	-
ASH3||STI	1	1	0	1	0	0	1	-	-
FIX||STI	1	1	0	1	0	1	0	-	-
FLOAT||STF	1	1	0	1	0	1	1	-	-
LDF||STF	1	1	0	1	1	0	0	-	-
LDI||STI	1	1	0	1	1	0	1	-	-
LSH3||STI	1	1	0	1	1	1	0	-	-
MPYF3||STF	1	1	0	1	1	1	1	-	-
MPYI3||STI	1	1	1	0	0	0	0	-	-
NEGF||STF	1	1	1	0	0	0	1	-	-
NEGI||STI	1	1	1	0	0	1	0	-	-
NOT||STI	1	1	1	0	0	1	1	-	-
OR3||STI	1	1	1	0	1	0	0	-	-
SUBF3||STF	1	1	1	0	1	0	1	-	-
SUBI3||STI	1	1	1	0	1	1	0	-	-
XOR3||STI	1	1	1	0	1	1	1	-	-
Reserved for reset, traps, and interrupts	0	1	1	1	1	1	1	1	1

Appendix D

Quality and Reliability

The quality and reliability performance of Texas Instruments Microprocessor and Microcontroller Products, which includes the three generations of TMS320 digital signal processors, relies on feedback from:

- Our customers
- Our total manufacturing operation from front-end wafer fabrication to final shipping inspection
- Product quality and reliability monitoring.

Our customer's perception of quality must be the governing criterion for judging performance. This concept is the basis for Texas Instruments Corporate Quality Policy, which is as follows:

> "For every product or service we offer, we shall define the requirements that solve the customer's problems, and we shall conform to those requirements without exception."

Texas Instruments offers a leadership reliability qualification system, based on years of experience with leading-edge memory technology as well as years of research into customer requirements. Quality and reliability programs at TI are therefore based on customer input and internal information to achieve constant improvement in quality and reliability.

D.1 Reliability Stress Tests

Accelerated stress tests are performed on new semiconductor products and process changes to ensure product reliability excellence. The typical test environments used to qualify new products or major changes in processing are:

- High-temperature operating life
- Storage life
- Temperature cycling
- Biased humidity
- Autoclave
- Electrostatic discharge
- Package integrity
- Electromigration
- Channel-hot electrons (performed on geometries less than 2.0 μm).

Typical events or changes that require internal requalification of product include:

- New die design, shrink, or layout
- Wafer process (baseline/control systems, flow, mask, chemicals, gases, dopants, passivation, or metal systems)
- Packaging assembly (baseline control systems or critical assembly equipment)
- Piece parts (such as lead frame, mold compound, mount material, bond wire, or lead finish)
- Manufacturing site.

TI reliability control systems extend beyond qualification. Total reliability controls and management include product reliabily monitor as well as final product release controls. MOS memories, utilizing high-density active elements, serve as the leading indicator in wafer-process integrity at TI MOS fabrication sites, enhancing all MOS logic device yields and reliability. TI places more than several thousand MOS devices per month on reliability test to ensure and sustain built-in product excellence.

Table D-1 lists the microprocessor and microcontroller reliability tests, the duration of the test, and sample size. The following defines and describes those tests in the table.

AOQ (Average Outgoing Quality) Amount of defective product in a population, usually expressed in terms of parts per million (PPM).

FIT (Failure In Time) Estimated field failure rate in number of failures per billion power-on device hours; 1000 FITS equals 0.1 percent fail per 1000 device hours.

Operating lifetest Device dynamically exercised at a high ambient temperature (usually 125°C) to simulate field usuage that would

	expose the device to a much lower ambient temperature (such as 55°C). Using a derived high temperature, a 55°C ambient failure rate can be calculated.
High-temperature storage	Device exposed to 150°C unbiased condition. Bond integrity is stressed in this environment.
Biased humidity	Moisture and bias used to accelerate corrosion-type failures in plastic packages. Conditions include 85°C ambient temperature with 85-percent relative humidity (RH). Typical bias voltage is +5 V and ground on alternating pins.
Autoclave (pressure cooker)	Plastic-packaged devices exposed to moisture at 121°C using a pressure of one atmosphere above normal pressure. The pressure forces moisture permeation of the package and accelerates corrosion mechanisms (if present) on the device. External package contaminates can also be activated and caused to generate inter-pin current leakage paths.
Temperature cycle	Device exposed to severe temperature extremes in an alternating fashion (-65°C for 15 minutes and 150°C for 15 minutes per cycle) for at least 1000 cycles. Package strength, bond quality, and consistency of assembly process are stressed in this environment.
Thermal shock	Test similar to the temperature cycle test, but involving a liquid-to-liquid transfer, per MIL-STD-883C, Method 1011.
PIND	Particle Impact Noise Detection test. A non-destructive test to detect loose particles inside a device cavity.
Mechanical Sequence:	
Fine and gross leak	Per MIL-STD-883C, Method 1014.5
Mechanical shock	Per MIL-STD-883C, Method 2002.3, 1500 g, 0.5 ms, Condition B
PIND (optional)	Per MIL-STD-883C, Method 2020.4
Vibration, variable frequency	Per MIL-STD-883C, Method 2007.1, 20 g, Condition A
Constant acceleration	Per MIL-STD-883C, Method 2001.2, 20 kg, Condition D, Y1 Plane min
Fine and gross leak	Per MIL-STD-883C, Method 1014.5

Electrical test	To data sheet limits
Thermal Sequence:	
Fine and gross leak	Per MIL-STD-883C, Method 1014.5
Solder heat (optional)	Per MIL-STD-750C, Method 1014.5
Temperature cycle (10 cycles minimum)	Per MIL-STD-883C, Method 1010.5, -65 to +150°C, Condition C
Thermal shock (10 cycles minimum)	Per MIL-STD-883C, Method 1011.4, -55 to +125°C, Condition B
Moisture resistance	Per MIL-STD-883C, Method 1004.4
Fine and gross leak	Per MIL-STD-883C, Method 1014.5
Electrical test	To data sheet limits
Thermal/Mechanical Sequence:	
Fine and gross leak	Per MIL-STD-883C, Method 1014.5
Temperature cycle (10 cycles minimum)	Per MIL-STD-883C, Method 1010.5, -65 to +150°C, Condition C
Constant acceleration	Per MIL-STD-883C, Method 2001.2, 30 kg, Y1 Plane
Fine and gross leak	Per MIL-STD-883C, Method 1014.5
Electrical test	To data sheet limits
Electrostatic discharge	Per MIL-STD-883C, Method 3015
Solderability	Per MIL-STD-883C, Method 2003.3
Solder heat	Per MIL-STD-750C, Method 2031, 10 sec
Salt atmosphere	Per MIL-STD-883C, Method 1009.4, Condition A, 24 hrs min
Lead pull	Per MIL-STD-883C, Method 2004.4, Condition A
Lead integrity	Per MIL-STD-883C, Method 2004.4, Condition B1
Electromigration	Accelerated stress testing of conductor patterns to ensure acceptable lifetime of power-on operation
Resistance to solvents	Per MIL-STD-883C, Method 2015.4

Table D-1. Microprocessor and Microcontroller Tests

TEST	DURATION	SAMPLE SIZE	
		PLASTIC	CERAMIC
Operating life, 125°C, 5.0 V	1000 hrs	129	129
Operating life, 150°C, 5.0 V	1000 hrs	77*	77
Storage life, 150°C	1000 hrs	77	77
Biased 85°C/85 percent RH, 5.0 V	1000 hrs	129	-
Autoclave, 121°C, 1 ATM	240 hrs	77	-
Temperature cycle, -65 to 150°C	1000 cyc†	129	129
Temperature cycle, 0 to 125°C	3000 cyc	129	129
Thermal shock, -65 to 150°C	200 cyc	129	129
Electrostatic discharge, ±2 kV		12	12
Latch-up (CMOS devices only)		5	5
Mechanical sequence		-	38
Thermal sequence		-	38
Thermal/mechanical sequence		-	38
PIND		-	45
Internal water vapor		-	3
Solderability		22	22
Solder heat		22	22
Resistance to solvents		15	15
Lead integrity		15	15
Lead pull		22	-
Lead finish adhesion		15	15
Salt atmosphere		15	15
Flammability (UL94-V0)		3	-
Thermal impedance		5	5

*If junction temperature does not exceed plasticity of package.
†For severe environments; reduced cycles for office environments.

Table D-2 lists the TMS320C30 device, the approximate number of transistors, and the equivalent gates. The numbers have been determined from design verification runs.

Table D-2. TMS320C30 Transistors

DEVICE	# TRANSISTORS	# GATES
CMOS: TMS320C30	600K-700K	200K

TI Qualification test updates are available upon request at no charge. TI will consider performing any additional reliability test(s), if requested. For more information on TI quality and reliability programs, contact the nearest TI field sales office.

Note:

Texas Instruments reserves the right to make changes in MOS Semiconductor test limits, procedures, or processing without notice. Unless prior arrangements for notification have been made, TI advises all customers to reverify current test and manufacturing conditions prior to relying on published data.

Index

A

addition (floating-point) 5-13
addressing 6-1
  groups of addressing modes 6-18
    conditional-branch addressing modes 6-21
    general addressing modes 6-18
    long-immediate addressing modes 6-21
    parallel addressing modes 6-20
    three-operand addressing modes 6-19
  types of addressing 6-2
    direct 6-3
    indirect 6-4
    long-immediate 6-17
    PC-relative 6-17
    register 6-2
    short-immediate 6-16
addressing modes 3-12
ALU 3-5
  ALU 3-5
application-oriented operations 12-45
  companding 12-45
  fast fourier transforms 12-63
  FIR,IIR, and adaptive filters 12-49
  lattice filters 12-79
  matrix-vector multiplication 12-60
applications 1-5
ARAUs 3-5
  ARAUs 3-5
architectural overview 3-1
assembler B-4
auxiliary registers 6-22
auxiliary registers (AR0-AR7) 4-3

B

bank switching (programmable) 8-21
bank switching techniques 13-14
bit-reversed addressing 6-27
block repeat registers (RS, RE) 4-10
block size register (BK) 4-4
blocksize register (BK) 6-22
branch conflicts 10-4
branches 7-7
bulletin board B-11
bus operation (external) 8-1
buses (external) 3-21
buses (internal) 3-20

C

C compiler B-5
Cache 3-8
cache (instruction) 4-15
  algorithm 4-16
  architecture 4-15
  clear bit (CC) 4-17
  control bits 4-17
  enable bit (CE) 4-17
  freeze bit (CF) 4-17
central processing unit (CPU) 3-3
circular addressing 6-22
  algorithm 6-24
  auxiliary registers 6-22
  buffer 6-22
  FIR filters 6-25
  flowchart 6-23
  implementation 6-24
circular buffer 6-22
clock divide (serial port) 9-23
clock oscillator circuitry 13-21
clock phases 10-16
clocking memory accesses 10-16
condition codes 11-8
condition flags 11-8
conditional-branch addressing modes 6-21
conflict resolution (memory) 10-14

conflicts (pipeline) 10-4
continuous modes (serial port transmit/receive) 9-24
conversion 5-22
  floating-point to integer 5-22
  integer to floating-point 5-24
conversions between floating-point formats 5-7
counter register (timer) 9-5
CPU interrupt flag register (IF) 4-8
CPU register file 4-2
  auxiliary registers (AR0-AR7) 4-3
  block repeat registers (RS, RE) 4-10
  block size register (BK) 4-4
  CPU interrupt flag register (IF) 4-8
  CPU/DMA interrupt enable register (IE) 4-7
  data page pointer (DP) 4-3
  extended-precision registers (R0-R7) 4-3
  I/O flags register (IOF) 4-9
  index registers (IR0, IR1) 4-4
  register file 4-2
  repeat counter (RC) 4-10
  status register (ST) 4-4
  system stack pointer (SP) 4-4
CPU/DMA interrupt enable register (IE) 4-7

D

data formats 5-1
  floating-point 5-4
  integer 5-2
  unsigned integer 5-3
data loads and stores 10-16
data page pointer (DP) 4-3
data receive register (serial port) 9-19
data transmit register (serial port) 9-19
delayed branches 7-7
deques 6-28, 6-30
development support B-1
  C compiler B-5
  macro assembler/linker B-4
  simulator B-6
  TMS320 DSP bulletin board service B-11
  TMS320 DSP hotline B-11
  XDS1000 B-9
direct addressing 6-3
DMA 3-24
DMA controller 9-33
  memory transfer operation 9-38
  registers 9-33
    destination/source address 9-36
    global control 9-34
    interrupt enable 9-36
    transfer counter 9-36
  synchronization of DMA channels 9-42
documentation B-3

E

effective base (EB) of buffer 6-22
execute only 10-10
expansion bus control register 8-4
expansion bus I/O cycles 8-10
expansion bus interface 13-17
extended-precision floating-point 5-6
extended-precision registers (R0-R7) 4-3
external bus operation 8-1
external interface control registers 8-2
external interface timing 8-5
  expansion bus I/O cycles 8-10
  primary bus cycles 8-5

F

FIR, IIR, and adaptive filters 12-49
  adaptive filters (LMS algorithm) 12-57
  FIR filters 12-50
  IIR filters 12-52
FIX Instructions 5-22
FLOAT instruction 5-24
floating-point format conversion: IEEE to/from TMS320C30 12-35
  IEEE to TMS320C30 floating-point format conversion 12-37
  TMS320C30 to IEEE floating-point format conversion 12-41
floating-point formats 5-4
  conversions between formats 5-7
  extended-precision 5-6
  short 5-4
  single-precision 5-5
floating-point operations 5-1
  addition/subtraction 5-13
  conversion to floating-point 5-24
  conversion to integer 5-22
  multiplication 5-9
  normalization 5-17

rounding 5-20
FSR 9-24
FSR/DR/CLKR control register (serial port) 9-15
FSX 9-23
FSX/DX/CLKX control register (serial port) 9-14

G

gates D-5
general addressing modes 6-18
global control register (serial port) 9-11
global control register (timer) 9-3
groups of addressing modes 6-18

H

handshake mode (serial port) 9-25
hardware applications 13-1
expansion bus interface 13-17
primary bus interface 13-4
system configuration options overview 13-2
system control functions 13-21
Harvard architecture 1-3
hold everything conflicts 10-12
hotline B-11

I

I/O flags register (IOF) 4-9
id=file.CPU register file 3-5
register file 3-5
index registers (IR0, IR1) 4-4
indirect addressing 6-4
instruction cache 4-15
instruction opcodes C-1
instruction set 11-2
interlocked operations 11-5
load and store 11-2
parallel operations 11-5
program control 11-4
three-operand 11-4
two-operand 11-3
instruction set summary 3-12
integer and floating-point division 12-23
computation of floating-point inverse and division 12-26
integer division 12-23
integer formats 5-2
interlocked loads 10-12
interlocked operations 7-8
interlocked operations instructions 11-5
interrupt service routines 12-10
context switching 12-11
interrupt priority 12-14
interrupt vectors 4-13
interrupts 7-16

K

key features 1-4

L

linker B-4
load instructions 11-2
logical and arithmetic operations 12-20
bit manipulation 12-20
bit-reversed addressing 12-22
block moves 12-22
extended-precision arithmetic 12-32
floating-point format conversion: IEEE to/from TMS320C30 12-35
integer and floating-point division 12-23
square root 12-29
long-immediate addressing 6-17
long-immediate addressing modes 6-21

M

macro assembler B-4
memory 3-8, 4-11
maps 3-10
memory maps 4-11
peripheral bus map 4-13
reset/interrupt/trap map 4-13
memory access timing 10-16
memory conflict resolution 10-14
memory conflicts 10-8
memory stacks 6-28
memory transfer operation (DMA) 9-38
multiplication (floating-point) 5-9
multiplier 3-5
multiplier 3-5
multiprocessing 9-25

N

nomenclature B-15
normalization (floating-point) 5-17

O

opcodes (instruction) C-1
operation configurations (serial port) 9-20
operation modes (timer) 9-7
ordering information B-13

P

parallel addressing modes 6-20
parallel multiplies/adds 10-19
parallel operations 10-18
parallel operations instructions 11-5
part numbers B-13
PC-relative addressing 6-17
period register (timer) 9-5
peripheral bus map 4-13
peripheral bus memory map 4-13
peripherals 3-22, 9-1
  DMA controller 9-33
  serial ports 9-9
  timers 9-2
pipeline operation 10-1
  clocking memory accesses 10-16
  conflicts 10-4
    branch 10-4
    memory 10-8
    register 10-6
  resolving memory conflicts 10-14
  structure 10-2
primary bus control register 8-3
primary bus cycles 8-5
primary bus interface 13-4
  bank switching techniques 13-14
  ready generation 13-10
  zero wait-state interface to RAMs 13-4
product quality/reliability D-1
program control 12-7
  computed GOTO's 12-18
  delayed branches 12-15
  interrupt service routines 12-10
  repeat modes 12-16
  software stack 12-9
  subroutines 12-7
program control instructions 11-4
program counter (PC) 4-10
program fetch incomplete 10-10
program fetches 10-16
program flow control 7-1
program wait 10-8
programmable bank switching 8-21
programmable wait states 8-19
programming tips 12-86
  C-callable routines 12-86
  hints for assembly coding 12-86
pulse generation (timer) 9-6

Q

quality/reliability D-1
queues 6-28, 6-30

R

RAM 3-8
ready generation 13-10
receive/transmit timer control register (serial port) 9-16
receive/transmit timer counter register (serial port) 9-18
receive/transmit timer period register (serial port) 9-19
register addressing 6-2
register conflicts 10-6
registers (DMA) 9-33
reliability tests D-2
repeat counter (RC) 4-10
repeat end address (RE) 4-10
repeat modes 7-2, 12-16
  block repeat 12-16
  initialization 7-2, 7-3
  operation 7-4
  single-instruction repeat 12-17
repeat start address (RS) 4-10
reserved bits and compatibility 4-10
reset operation 7-13
reset signal generation 13-23
reset vectors 4-13
resolving memory conflicts 10-14
RND instruction 5-20
ROM 3-8
rounding (floating-point) 5-20
RPTB 7-2
RPTS 7-2

RRDY 9-23

S

serial port functional operation 9-26
serial port interrupt sources 9-26
serial ports 3-23, 9-9
  data receive register 9-19
  data transmit register 9-19
  FSR/DR/CLKR control register 9-15
  FSX/DX/CLKX control register 9-14
  global control register 9-11
  operation configurations 9-20
  receive/transmit timer control register 9-16
  receive/transmit timer counter register 9-18
  receive/transmit timer period register 9-19
  timing 9-23
short-immediate addressing 6-16
signal descriptions 2-3
simulator B-6
single-precision floating-point 5-5
software applications 12-1
  application-oriented operations 12-45
  logical and arithmetic operations 12-20
  processor initialization 12-3
  program control 12-7
  programming tips 12-86
stacks 6-28
status register (ST) 4-4
store instructions 11-2
subtraction (floating-point) 5-13
synchronization of DMA channels 9-42
system configuration options overview 13-2
  categories of interfaces on the TMS320C30 13-2
  typical system block diagram 13-3
system control functions 13-21
  clock oscillator circuitry 13-21
  reset signal generation 13-23
system stack management 6-28
system stack pointer (SP) 4-4

T

three-operand addressing modes 6-19
three-operand instructions 11-4
timers 9-2
  counter register 9-5
  global control register 9-3
  operation modes 9-7
  period register 9-5
  pulse generation 9-6
timing (serial port) 9-23
timing memory accesses 10-16
TMS320 device nomenclature B-15
TMS320 DSP bulletin board service B-11
TMS320 DSP hotline B-11
TMS320C30 emulator - extended development system (XDS1000) B-9
transistors D-5
trap vectors 4-13
two-operand instructions 11-3
types of addressing 6-2

U

unsigned-integer formats 5-3

W

wait states (programmable) 8-19

X

XDS1000 target design considerations 13-26
XRDY 9-23

Z

zero-glue multiprocessing 9-25

TI Worldwide Sales Offices

ALABAMA: Huntsville: 500 Wynn Drive, Suite 514, Huntsville, AL 35805, (205) 837-7530.

ARIZONA: Phoenix: 8825 N. 23rd Ave., Phoenix, AZ 85021, (602) 995-1007.

CALIFORNIA: Irvine: 17891 Cartwright Rd., Irvine, CA 92714, (714) 660-8187; **Sacramento:** 1900 Point West Way, Suite 171, Sacramento, CA 95815, (916) 929-1521; **San Diego:** 4333 View Ridge Ave., Suite B., San Diego, CA 92123, (619) 278-9601; **Santa Clara:** 5353 Betsy Ross Dr., Santa Clara, CA 95054, (408) 980-9000; **Torrance:** 690 Knox St., Torrance, CA 90502, (213) 217-7010; **Woodland Hills:** 21220 Erwin St., Woodland Hills, CA 91367, (818) 704-7759.

COLORADO: Aurora: 1400 S. Potomac Ave., Suite 101, Aurnra, CO 80012, (303) 368-8000.

CONNECTICUT: Wallingford: 9 Barnes Industrial Park Rd., Barnes Industrial Park, Wallingford, CT 06492, (203) 269-0074.

FLORIDA: Ft. Lauderdale: 2765 N.W. 62nd St., Ft. Lauderdale, FL 33309, (305) 973-8502; **Maitland:** 2601 Maitland Center Parkway, Maitland, FL 32751, (305) 660-4600; **Tampa:** 5010 W. Kennedy Blvd., Suite 101, Tampa, FL 33609, (813) 870-6420.

GEORGIA: Norcross: 5515 Spalding Drive, Norcross, GA 30092, (404) 662-7900

ILLINOIS: Arlington Heights: 515 W. Algonquin, Arlington Heights, IL 60005, (312) 640-2925.

INDIANA: Ft. Wayne: 2020 Inwood Dr., Ft. Wayne, IN 46815, (219) 424-5174; **Indianapolis:** 2346 S. Lynhurst, Suite J-400, Indianapolis, IN 46241, (317) 248-8555.

IOWA: Cedar Rapids: 373 Collins Rd. NE, Suite 200, Cedar Rapids, IA 52402, (319) 395-9550.

MARYLAND: Baltimore: 1 Rutherford Pl., 7133 Rutherford Rd., Baltimore, MD 21207, (301) 944-8600.

MASSACHUSETTS: Waltham: 504 Totten Pond Rd., Waltham, MA 02154, (617) 895-9100.

MICHIGAN: Farmington Hills: 33737 W. 12 Mile Rd., Farmington Hills, MI 48018, (313) 553-1500.

MINNESOTA: Eden Prairie: 11000 W. 78th St., Eden Prairie, MN 55344 (612) 828-9300.

MISSOURI: Kansas City: 8080 Ward Pkwy., Kansas City, MO 64114, (816) 523-2500; **St. Louis:** 11816 Borman Drive, St. Louis, MO 63146, (314) 569-7600.

NEW JERSEY: Iselin: 485E U.S. Route 1 South, Parkway Towers, Iselin, NJ 08830 (201) 750-1050

NEW MEXICO: Albuquerque: 2820-D Broadbent Pkwy NE, Albuquerque, NM 87107, (505) 345-2555.

NEW YORK: East Syracuse: 6365 Collamer Dr., East Syracuse, NY 13057, (315) 463-9291; **Endicott:** 112 Nanticoke Ave., P.O. Box 618, Endicott, NY 13760, (607) 754-3900; **Melville:** 1 Huntington Quadrangle, Suite 3C10, P.O. Box 2936, Melville, NY 11747, (516) 454-6600; **Pittsford:** 2851 Clover St., Pittsford, NY 14534, (716) 385-6770; **Poughkeepsie:** 385 South Rd., Poughkeepsie, NY 12601, (914) 473-2900.

NORTH CAROLINA: Charlotte: 8 Woodlawn Green, Woodlawn Rd., Charlotte, NC 28210, (704) 527-0930; **Raleigh:** 2809 Highwoods Blvd., Suite 100, Raleigh, NC 27625, (919) 876-2725.

OHIO: Beachwood: 23408 Commerce Park Rd., Beachwood, OH 44122, (216) 464-6100; **Dayton:** Kingsley Bldg., 4124 Linden Ave., Dayton, OH 45432, (513) 258-3877.

OREGON: Beaverton: 6700 SW 105th St., Suite 110, Beaverton, OR 97005, (503) 643-6758.

PENNSYLVANIA: Ft. Washington: *260 New York Dr., Ft. Washington, PA 19034, (215) 643-6450;* **Coraopolis:** 420 Rouser Rd., 3 Airport Office Park, Coraopolis, PA 15108, (412) 771-8550.

PUERTO RICO: Hato Rey: Mercantil Plaza Bldg., Suite 505, Hato Rey, PR 00919, (809) 753-8700.

TEXAS: Austin: P.O. Box 2909, Austin, TX 78769, (512) 250-7655; **Richardson:** 1001 E. Campbell Rd., Richardson, TX 75080, (214) 680-5082; **Houston:** 9100 Southwest Frwy., Suite 237, Houston, TX 77036, (713) 778-6592; **San Antonio:** 1000 Central Parkway South, San Antonio, TX 78232, (512) 496-1779.

UTAH: Murray: 5201 South Green SE, Suite 200, Murray, UT 84107, (801) 266-8972.

VIRGINIA: Fairfax: 2750 Prosperity, Fairfax, VA 22031, (703) 849-1400.

WASHINGTON: Redmond: 5010 148th NE, Bldg B, Suite 107, Redmond, WA 98052, (206) 881-3080.

WISCONSIN: Brookfield: 450 N. Sunny Slope, Suite 150, Brookfield, WI 53005, (414) 785-7140.

CANADA: Nepean: 301 Moodie Drive, Mallorn Center, Nepean, Ontario, Canada, K2H9C4, (613) 726-1970. **Richmond Hill:** 280 Centre St. E., Richmond Hill L4C1B1, Ontario, Canada (416) 884-9181; **St. Laurent:** Ville St. Laurent Quebec, 9460 Trans Canada Hwy., St. Laurent, Quebec, Canada H4S1R7, (514) 335-8392.

ARGENTINA: Texas Instruments Argentina S.A.I.C.F.: Esmeralda 130, 15th Floor, 1035 Buenos Aires, Argentina, 1 + 394-3008.

AUSTRALIA (& NEW ZEALAND): Texas Instruments Australia Ltd.: 6-10 Talavera Rd., North Ryde (Sydney), New South Wales, Australia 2113, 2 + 887-1122; 5th Floor, 418 St. Kilda Road, Melbourne, Victoria, Australia 3004, 3 + 267-4677; 171 Philip Highway, Elizabeth, South Australia 5112, 8 + 255-2066.

AUSTRIA: Texas Instruments Ges.m.b.H.: Industriestrabe B/16, A-2345 Brunn/Gebirge, 2236-846210.

BELGIUM: Texas Instruments N.V. Belgium S.A.: Mercure Centre, Raketstraat 100, Rue de la Fusee, 1130 Brussels, Belgium, 2/720.60.00.

BRAZIL: Texas Instruments Electronicos do Brasil Ltda.: Rua Paes Leme, 524-7 Andar Pinheiros, 05424 Sao Paulo, Brazil, 0815-6166.

DENMARK: Texas Instruments A/S, Mairelundvej 46E, DK-2730 Herlev, Denmark, 2 - 91 74 00.

FINLAND: Texas Instruments Finland OY: Teollisuuskatu 19D 00511 Helsinki 51, Finland, (90) 701-3133.

FRANCE: Texas Instruments France: Headquarters and Prod. Plant, BP 05, 06270 Villeneuve-Loubet, (93) 20-01-01; Paris Office, BP 67 8-10 Avenue Morane-Saulnier, 78141 Velizy-Villacoublay, (3) 946-97-12; Lyon Sales Office, L'Oree D'Ecully, Batiment B, Chemin de la Forestiere, 69130 Ecully, (7) 833-04-40; Strasbourg Sales Office, Le Sebastopol 3, Quai Kleber, 67055 Strasbourg Cedex, (88) 22-12-66; Rennes, 23-25 Rue du Pults Mauger, 35100 Rennes, (99) 31-54-86; Toulouse Sales Office, Le Peripole—2, Chemin du Pigeonnier de la Cepiere, 31100 Toulouse, (61) 44-18-19; Marseille Sales Office, Nolily Paradis—146 Rue Paradis, 13006 Marseille, (91) 37-25-30.

TEXAS INSTRUMENTS

GERMANY (Fed. Republic of Germany): Texas Instruments Deutschland GmbH: Haggertystrasse 1, D-8050 Freising, 8161 + 80-4591; Kurfuerstendamm 195/196, D-1000 Berlin 15, 30 + 882-7365; III, Hagen 43/Kibbelstrasse, 19, D-4300 Essen, 201-24250; Frankfurter Allee 6-8, D-6236 Eschborn 1, 06196 + 8070; Hamburgerstrasse 11, D-2000 Hamburg 76, 040 + 220-1154, Kirchhorsterstrasse 2, D-3000 Hannover 51, 511 + 648021; Maybachstrabe 11, D-7302 Ostfildern 2-Nellingen, 711 + 547001; Mixikoring 19, D-2000 Hamburg 60, 40 + 637 + 0061; Postfach 1309, Roonstrasse 16, D-5400 Koblenz, 261 + 35044.

HONG KONG (+ PEOPLES REPUBLIC OF CHINA): Texas Instruments Asia Ltd., 8th Floor, World Shipping Ctr., Harbour City, 7 Canton Rd., Kowloon, Hong Kong, 3 + 722-1223.

IRELAND: Texas Instruments (Ireland) Limited: Brewery Rd., Stillorgan, County Dublin, Eire, 1 831311.

ITALY: Texas Instruments Semiconduttori Italia Spa: Viale Delle Scienze, 1, 02015 Cittaducale (Rieti), Italy, 746 694.1; Via Salaria KM 24 (Palazzo Cosma), Monterotondo Scalo (Rome), Italy, 6 + 9003241; Viale Europa, 38-44, 20093 Cologno Monzese (Milano), 2 2532541; Corso Svizzera, 185, 10100 Torino, Italy, 11 774545; Via J. Barozzi 6, 40100 Bologna, Italy, 51 355851.

JAPAN: Texas Instruments Asia Ltd.: 4F Aoyama Fuji Bldg., 6-12, Kita Aoyama 3-Chome, Minato-ku, Tokyo, Japan 107, 3-498-2111; Osaka Branch, 5F, Nissho Iwai Bldg., 30 Imabashi 3- Chome, Higashi-ku, Osaka, Japan 541, 06-204-1881; Nagoya Branch, 7F Daini Toyota West Bldg., 10-27, Meieki 4-Chome, Nakamura-ku Nagoya, Japan 450, 52-583-8691.

KOREA: Texas Instruments Supply Co.: 3rd Floor, Samon Bldg., Yuksam-Dong, Gangnam-ku, 135 Seoul, Korea, 2 + 462-8001.

MEXICO: Texas Instruments de Mexico S.A.: Mexico City, AV Reforma No. 450 — 10th Floor, Mexico, D.F., 06600, 5 + 514-3003.

MIDDLE EAST: Texas Instruments: No. 13, 1st Floor Mannai Bldg., Diplomatic Area, P.O. Box 26335, Manama Bahrain, Arabian Gulf, 973 + 274681.

NETHERLANDS: Texas Instruments Holland B.V., P.O. Box 12995, (Bullewijk) 1100 CB Amsterdam, Zuid-Oost, Holland 20 + 5602911.

NORWAY: Texas Instruments Norway A/S: PB106, Refstad 131, Oslo 1, Norway, (2) 155090.

PHILIPPINES: Texas Instruments Asia Ltd.: 14th Floor, Ba- Lepanto Bldg., 8747 Paseo de Roxas, Makati, Metro Manila, Philippines, 2 + 8188987.

PORTUGAL: Texas Instruments Equipamento Electronico (Portugal), Lda.: Rua Eng. Frederico Ulrich, 2650 Moreira Da Maia, 4470 Maia, Portugal, 2-948-1003.

SINGAPORE (+ INDIA, INDONESIA, MALAYSIA, THAILAND): Texas Instruments Asia Ltd.: 12 Lorong Bakar Batu, Unit 01-02, Kolam Ayer Industrial Estate, Republic of Singapore, 747-2255.

SPAIN: Texas Instruments Espana, S.A.: C/Jose Lazaro Galdiano No. 6, Madrid 16, 1/458.14.58.

SWEDEN: Texas Instruments International Trade Corporation (Sverigefilialen): Box 39103, 10054 Stockholm, Sweden, 8 - 235480.

SWITZERLAND: Texas Instruments, Inc., Reidstrasse 6, CH-8953 Dietikon (Zuerich) Switzerland, 1-740 2220.

TAIWAN: Texas Instruments Supply Co.: Room 903, 205 Tun Hwan Rd., 71 Sung-Kiang Road, Taipei, Taiwan, Republic of China, 2 + 521-9321.

UNITED KINGDOM: Texas Instruments Limited: Manton Lane, Bedford, MK41 7PA, England, 0234 67466; St. James House, Wellington Road North, Stockport, SK4 2RT, England, 61 + 442-7162.

BM

TI Sales Offices

ALABAMA: Huntsville (205) 837-7530.

ARIZONA: Phoenix (602) 995-1007;
Tucson (602) 624-3276.

CALIFORNIA: Irvine (714) 660-1200;
Sacramento (916) 929-0197;
San Diego (619) 278-9600;
Santa Clara (408) 980-9000;
Torrance (213) 217-7000;
Woodland Hills (818) 704-7759.

COLORADO: Aurora (303) 368-8000.

CONNECTICUT: Wallingford (203) 269-0074.

FLORIDA: Altamonte Springs (305) 260-2116;
Ft. Lauderdale (305) 973-8502;
Tampa (813) 286-0420.

GEORGIA: Norcross (404) 662-7900.

ILLINOIS: Arlington Heights (312) 640-3000.

INDIANA: Carmel (317) 573-6400;
Ft. Wayne (219) 424-5174.

IOWA: Cedar Rapids (319) 395-9550.

KANSAS: Overland Park (913) 451-4511.

MARYLAND: Baltimore (301) 944-8600.

MASSACHUSETTS: Waltham (617) 895-9100.

MICHIGAN: Farmington Hills (313) 553-1500;
Grand Rapids (616) 957-4200.

MINNESOTA: Eden Prairie (612) 828-9300.

MISSOURI: St. Louis (314) 569-7600.

NEW JERSEY: Iselin (201) 750-1050.

NEW MEXICO: Albuquerque (505) 345-2555.

NEW YORK: East Syracuse (315) 463-9291;
Melville (516) 454-6600; Pittsford (716) 385-6770;
Poughkeepsie (914) 473-2900.

NORTH CAROLINA: Charlotte (704) 527-0930;
Raleigh (919) 876-2725.

OHIO: Beachwood (216) 464-6100;
Dayton (513) 258-3877.

OREGON: Beaverton (503) 643-6758.

PENNSYLVANIA: Blue Bell (215) 825-9500.

PUERTO RICO: Hato Rey (809) 753-8700.

TENNESSEE: Johnson City (615) 461-2192.

TEXAS: Austin (512) 250-6769;
Houston (713) 778-6592; Richardson (214) 680-5082;
San Antonio (512) 496-1779.

UTAH: Murray (801) 266-8972.

VIRGINIA: Fairfax (703) 849-1400.

WASHINGTON: Redmond (206) 881-3080.

WISCONSIN: Brookfield (414) 782-2899.

CANADA: Nepean, Ontario (613) 726-1970;
Richmond Hill, Ontario (416) 884-9181;
St. Laurent, Quebec (514) 336-1860.

TI Regional Technology Centers

CALIFORNIA: Irvine (714) 660-8140;
Santa Clara (408) 748-2220;
Torrance (213) 217-7019.

COLORADO: Aurora (303) 368-8000.

GEORGIA: Norcross (404) 662-7945.

ILLINOIS Arlington Heights (313) 640-2909.

MASSACHUSETTS: Waltham (617) 895-9196.

TEXAS: Richardson (214) 680-5066.

CANADA: Nepean, Ontario (613) 726-1970.

TI Distributors

TI AUTHORIZED DISTRIBUTORS
Arrow/Kierulff Electronics Group
Arrow Canada (Canada)
Future Electronics (Canada)
GRS Electronics Co., Inc.
Hall-Mark Electronics
Marshall Industries
Newark Electronics
Schweber Electronics
Time Electronics
Wyle Laboratories
Zeus Components

–OBSOLETE PRODUCT ONLY–
Rochester Electronics, Inc.
Newburyport, Massachusetts
(617) 462-9332

ALABAMA: Arrow/Kierulff (205) 837-6955;
Hall-Mark (205) 837-8700; Marshall (205) 881-9235;
Schweber (205) 895-0480.

ARIZONA: Arrow/Kierulff (602) 437-0750;
Hall-Mark (602) 437-1200; Marshall (602) 496-0290;
Schweber (602) 997-4874; Wyle (602) 866-2888.

CALIFORNIA: Los Angeles/Orange County:
Arrow/Kierulff (818) 701-7500, (714) 838-5422;
Hall-Mark (818) 716-7300, (714) 669-4100,
(213) 217-8400; Marshall (818) 407-0101, (818) 459-5500,
(714) 458-5395; Schweber (818) 999-4702;
(714) 863-0200, (213) 320-8090; Wyle (213) 322-9953,
(818) 880-9000, (714) 863-9953; Zeus (714) 921-9000;
Sacramento: Hall-Mark (916) 722-8600;
Marshall (916) 635-9700; Schweber (916) 929-9732;
Wyle (916) 638-5282;
San Diego: Arrow/Kierulff (619) 565-4800;
Hall-Mark (619) 268-1201; Marshall (619) 578-9600;
Schweber (619) 450-0454; Wyle (619) 565-9171;
San Francisco Bay Area: Arrow/Kierulff (408) 745-6600,
Hall-Mark (408) 432-0900; Marshall (408) 942-4600;
Schweber (408) 432-7171; Wyle (408) 727-2500;
Zeus (408) 998-5121.

COLORADO: Arrow/Kierulff (303) 790-4444;
Hall-Mark (303) 790-1662; Marshall (303) 451-8383;
Schweber (303) 799-0258; Wyle (303) 457-9953.

CONNETICUT: Arrow/Kierulff (203) 265-7741;
Hall-Mark (203) 269-0100; Marshall (203) 265-3822;
Schweber (203) 748-7080.

FLORIDA: Ft. Lauderdale:
Arrow/Kierulff (305) 429-8200; Hall-Mark (305) 971-9280;
Marshall (305) 977-4880; Schweber (305) 977-7511;
Orlando: Arrow/Kierulff (305) 725-1480, (305) 682-6923;
Hall-Mark (305) 855-4020; Marshall (305) 767-8585;
Schweber (305) 331-7555; Zeus (305) 365-3000;
Tampa: Hall-Mark (813) 530-4543;
Marshall (813) 576-1399.

GEORGIA: Arrow/Kierulff (404) 449-8252;
Hall-Mark (404) 447-8000; Marshall (404) 923-5750;
Schweber (404) 449-9170.

ILLINOIS: Arrow/Kierulff (312) 250-0500;
Hall-Mark (312) 860-3800; Marshall (312) 490-0155;
Newark (312) 784-5100; Schweber (312) 364-3750.

INDIANA: Indianapolis: Arrow/Kierulff (317) 243-9353;
Hall-Mark (317) 872-8875; Marshall (317) 297-0483.

IOWA: Arrow/Kierulff (319) 395-7230;
Schweber (319) 373-1417.

KANSAS: Kansas City: Arrow/Kierulff (913) 541-9542;
Hall-Mark (913) 888-4747; Marshall (913) 492-3121;
Schweber (913) 492-2922.

MARYLAND: Arrow/Kierulff (301) 995-6002;
Hall-Mark (301) 988-9800; Marshall (301) 840-9450;
Schweber (301) 840-5900; Zeus (301) 997-1118.

MASSACHUSETTS Arrow/Kierulff (617) 935-5134;
Hall-Mark (617) 667-0902; Marshall (617) 658-0810;
Schweber (617) 275-5100, (617) 657-0760;
Time (617) 532-6200; Zeus (617) 863-8800.

MICHIGAN: Detroit: Arrow/Kierulff (313) 971-8220;
Marshall (313) 525-5850; Newark (313) 967-0600;
Schweber (313) 525-8100;
Grand Rapids: Arrow/Kierulff (616) 243-0912.

MINNESOTA: Arrow/Kierulff (612) 830-1800;
Hall-Mark (612) 941-2600; Marshall (612) 559-2211;
Schweber (612) 941-5280.

MISSOURI: St. Louis: Arrow/Kierulff (314) 567-6888;
Hall-Mark (314) 291-5350; Marshall (314) 291-4650;
Schweber (314) 739-0526.

NEW HAMPSHIRE: Arrow/Kierulff (603) 668-6968;
Schweber (603) 625-2250.

NEW JERSEY: Arrow/Kierulff (201) 538-0900,
(609) 596-8000; GRS Electronics (609) 964-8560;
Hall-Mark (201) 575-4415, (609) 235-1900;
Marshall (201) 882-0320, (609) 234-9100;
Schweber (201) 227-7880.

NEW MEXICO: Arrow/Kierulff (505) 243-4566.

NEW YORK: Long Island:
Arrow/Kierulff (516) 231-1000; Hall-Mark (516) 737-0600;
Marshall (516) 273-2424; Schweber (516) 334-7555;
Zeus (914) 937-7400;
Rochester: Arrow/Kierulff (716) 427-0300;
Hall-Mark (716) 244-9290; Marshall (716) 235-7620;
Schweber (716) 424-2222;
Syracuse: Marshall (607) 798-1611.

NORTH CAROLINA: Arrow/Kierulff (919) 876-3132,
(919) 725-8711; Hall-Mark (919) 872-0712;
Marshall (919) 878-9882; Schweber (919) 876-0000.

OHIO: Cleveland: Arrow/Kierulff (216) 248-3990;
Hall-Mark (216) 349-4632; Marshall (216) 248-1788;
Schweber (216) 464-2970;
Columbus: Arrow/Kierulff (614) 438-0928;
Hall-Mark (614) 888-3313;
Dayton: Arrow/Kierulff (513) 435-5563;
Marshall (513) 898-4480; Schweber (513) 439-1800.

OKLAHOMA: Arrow/Kierulff (918) 252-7537;
Schweber (918) 622-8003.

OREGON: Arrow/Kierulff (503) 645-6456;
Marshall (503) 644-5050; Wyle (503) 640-6000.

PENNSYLVANIA: Arrow/Kierulff (412) 856-7000,
(215) 928-1800; GRS Electronics (215) 922-7037;
Schweber (215) 441-0600, (412) 963-6804.

TEXAS: Austin: Arrow/Kierulff (512) 835-4180;
Hall-Mark (512) 258-8848; Marshall (512) 837-1991;
Schweber (512) 339-0088; Wyle (512) 834-9957;
Dallas: Arrow/Kierulff (214) 380-6464;
Hall-Mark (214) 553-4300; Marshall (214) 233-5200;
Schweber (214) 661-5010; Wyle (214) 235-9953;
Zeus (214) 783-7010;
Houston: Arrow/Kierulff (713) 530-4700;
Hall-Mark (713) 781-6100; Marshall (713) 895-9200;
Schweber (713) 784-3600; Wyle (713) 879-9953.

UTAH: Arrow/Kierulff (801) 973-6913;
Hall-Mark (801) 972-1008; Marshall (801) 485-1551;
Wyle (801) 974-9953.

WASHINGTON: Arrow/Kierulff (206) 575-4420;
Marshall (206) 747-9100; Wyle (206) 453-8300.

WISCONSIN: Arrow/Kierulff (414) 792-0150;
Hall-Mark (414) 797-7844; Marshall (414) 797-8400;
Schweber (414) 784-9020.

CANADA: Calgary: Future (403) 235-5325;
Edmonton: Future (403) 438-2858;
Montreal: Arrow Canada (514) 735-5511;
Future (514) 694-7710;
Ottawa: Arrow Canada (613) 226-6903;
Future (613) 820-8313;
Quebec City: Arrow Canada (418) 687-4231;
Toronto: Arrow Canada (416) 672-7769;
Future (416) 638-4771;
Vancouver: Future (604) 294-1166;
Winnipeg: Future (204) 339-0554.

Customer Response Center

TOLL FREE: (800) 232-3200

OUTSIDE USA: (214) 995-6611
(8:00 a.m. – 5:00 p.m. CST)

TEXAS INSTRUMENTS

BU

Expires December 31, 1989 SPR39IPR800R

TMS320 Literature Request Card

Please send me literature regarding the following TMS320 DSP areas:

PR01 [] DSP Applications
PR02 [] DSP Development Support
PR03 [] DSP University Program
PR04 [] First-Generation TMS320
PR05 [] Second-Generation TMS320

PR30 [] Please add me to your mailing list.

Name _____

Company _____ Title _____

Address _____

City/State/Zip _____ Telephone _____

Reader Response Card September 1988

Third-Generation TMS320 User's Guide

Please use this form to communicate your comments about this document, its organization and subject matter, for the purpose of improving technical documentation. Thank you for your cooperation.

1) What do you feel are the best features of this document? _____

2) How does this document meet your digital signal processing needs? _____

3) Do you find the organization of this document easy to follow? If not, why? _____

4) What additions do you think would enhance the structure and subject matter? _____

5) What deletions could be made without affecting overall usefulness? _____

6) Is there any incorrect or misleading information? _____

7) How would you improve this document? _____

Name _____

Company _____ Title _____

Address _____

City/State/Zip _____ Telephone _____

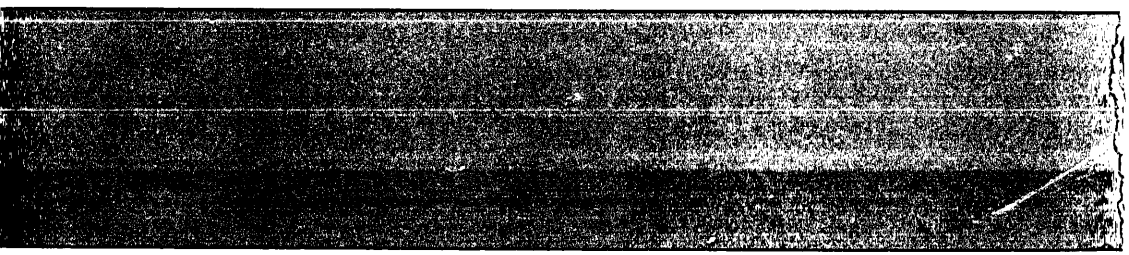
SPRU014

PLACE
STAMP
HERE

Literature Response Center
Texas Instruments Incorporated
P.O. Box 809066
Dallas, TX 75380-9990

PLACE
STAMP
HERE

Technical Publications Manager
Texas Instruments Incorporated
P.O. Box 1443, MS 640
Houston, TX 77001



August 1988
Printed in U.S.A.

TEXAS INSTRUMENTS