Butterfly Microcontroller Handbook

HB4100-2.0

For performance characteristics, refer to the Butterfly Microcontroller Performance Supplement: Publication no. SP4578 (for Commercial grade specification) or SP4708 (for Industrial grade specification)

.

© Mitel Corporation 1998 Publication No.HB4100-2.0 Issue No.2.0 Rev.C March 1998

TECHNICAL DOCUMENTATION - NOT FOR RESALE. PRINTED IN UNITED KINGDOM

This publication is issued to provide information only which (unless agreed by the Company in writing) may not be used, applied or reproduced for any purpose no form part of any order or contract nor to be regarded as a representation relating to the products or services concerned. No warranty or guarantee express or implied is made regarding the capability, performance or suitability of any product or service. The Company reserves the right to alter without prior notice the specification, design or price of any product or service. Information concerning possible methods of use is provided as a guide only and does not constitute any guarantee that such methods of use will be satisfactory in a specific piece of equipment. It is the user's responsibility to fully determine the performance and suitability of any equipment using such information and to ensure that any publication or data used is up to date and has not been superseded. These products are not suitable for use in any medical products whose failure to perform may result in significant injury or death to the user. All products and materials are sold and services provided subject to the Company's conditions of sale, which are available on request. All brand names and product names used in this publication are trademarks, registered trademarks or tradenames of their respective owners.

TABLE OF CONTENTS .... i - viii

LIST OF FIGURES ix

LIST OF TABLES..xiii

Document Conventions..xvii

Chapter 1 - Introduction

1.1 BUTTERFLY microcontroller overview 1-1
1.1.1 Features 1-1
1.2 Functional Block Description... 1-2
1.2.1 BmILD Bus 1-2
1.2.2 ARM7 Processor (ARM7)... 1-2
1.2.3 BmILD Broadcast Module (BBM) 1-3
1.2.4 Phase Locked Loop (PLL).. 1-3
1.2.5 Power Control (POCO).. 1-3
1.2.6 Power-On Reset (POR).. 1-3
1.2.7 Programmable Peripheral Interface (PPI) 1-3
1.2.8 Memory/Peripheral Controller (MPC)................................. 1-4
1.2.9 Universal Asynchronous Receiver Transmitter (UART)........ 1-4
1.2.10 Interrupt Controller (INTC) 1-4
1.2.11 DMA Controller (DMAC)... 1-4
1.2.12 Timer/Counters (TIC) 1-5
1.2.13 Watchdog Timer (WDOG)... 1-5
1.3 BUTTERFLY System Details.. 1-6
1.3.1 System Address Map 1-6
1.3.2 Address Map for internal I/O 1-7
1.3.3 System Clock Control.. 1-8
1.3.4 System Reset.. 1-8
1.3.5 System Bus Arbitration.. 1-8
1.3.6 System Configuration Register 1-9
1.3.7 DMA 1-11
1.3.8 Interrupt Sources... 1-12
1.3.9 Timer Counter (TIC) configuration 1-14

Chapter 2 - ARM7 Microprocessor

2.1 Overview 2-1

2.2 Architecture 2-2

2.3 Programmer's Model 2-3

2.3.1 Operating Mode Selection 2-3
2.3.2 Registers 2-3
2.3.3 Exceptions 2-7
2.3.4 FIQ 2-7
2.3.5 IRQ 2-8
2.3.6 Abort 2-8
2.3.7 Software Interrupt 2-9
2.3.8 Undefined Instruction Trap 2-9
2.3.9 Vector Summary 2-10
2.3.10 Exception Priorities 2-10
2.3.11 Interrupt Latencies 2-11
2.3.12 Reset 2-11

2.4 Instruction Set 2-12

2.4.1 Instruction Set Summary 2-12
2.4.2 The Condition Field 2-13
2.4.3 Branch and Branch with Link (B, BL) 2-14
2.4.4 Data Processing 2-15
2.4.5 PSR Transfer (MRS, MSR) 2-24
2.4.6 Multiply and Multiply-Accumulate (MUL, MLA) 2-29
2.4.7 Single Data Transfer (LDR, STR) 2-31
2.4.8 Block Data Transfer (LDM, STM) 2-38
2.4.9 Single Data Swap (SWP) 2-45
2.4.10 Software Interrupt (SWI) 2-47
2.4.11 Butterfly Coprocessor Support 2-50
2.4.12 Coprocessor Data Operations (CDP) 2-50
2.4.13 Undefined Instruction 2-52

2.5 Instruction Cycle Operations 2-53

2.5.1 Cycle Types 2-53
2.5.2 Branch and Branch with Link 2-54
2.5.3 Data Operations 2-55
2.5.4 Multiply and Multiply Accumulate 2-57
2.5.5 Load Register 2-58
2.5.6 Store Register 2-59
2.5.7 Load Multiple Registers 2-59
2.5.8 Store Multiple Registers 2-61

2.5.9 Data Swap 2-62
2.5.10 Software Interrupt and Exception Entry 2-63
2.5.11 Coprocessor Data Operation 2-64
2.5.12 Undefined Instructions and Coprocessor Absent 2-65
2.5.13 Unexecuted Instructions 2-65
2.5.14 Instruction Speed Summary 2-65

Chapter 3 - Diagnostic Broadcast (BBM)

3.1 Overview 3-1
3.1.1 Diagnostic Broadcast 3-2

Chapter 4 - Phase Locked Loop (PLL)

4.1 Overview 4-1
4.2 Design features 4-1
4.3 Architecture 4-2
4.3.1 Interface Definitions 4-3
4.3.2 Operating Modes. 4-4
4.4 PLL Operational Description 4-5
4.4.1 PLL for User Clock input. 4-5
4.4.2 PLL for Oscillator Clock input. 4-5
4.4.3 PLL Clock bypass 4-6
4.4.4 PLL Operational details. 4-6
4.5 Oscillator Operational Description 4-7
4.5.1 Pin Descriptions 4-7
4.5.2 Selection of External Components 4-8
4.5.3 Electrical Specification: 4-12
4.5.4 Application Notes 4-13

Chapter 5 - Power Control (POCO)

5.1 Overview 5-1
5.2 Architecture 5-2
5.3 Operational Description 5-2
5.3.1 System Reset/Power Up 5-2
5.3.2 Run Mode Operation 5-3

5.3.3 Standby Mode Clock Control Operation 5-3
5.3.4 Low Power Sleep Mode 5-3
5.3.5 Typical Configuration 5-4

5.4 Programmer's Model 5-5

5.4.1 Basic Register Operation 5-6

Chapter 6 - Programmable Peripheral Interface (PPI)

6.1 Overview 6-1

6.2 Architecture 6-1

6.3 Operational Description.. 6-2

6.4 Programmer's Model 6-4

6.4.1 Data Direction Register (DDR) 6-4
6.4.2 Data Input Register (DIR).. 6-5
6.4.3 Data Output Register (DOR) 6-5
6.4.4 The Control Status Register (CSR) 6-5

6.5 Timing relationship diagrams 6-6

6.6 External Interface 6-8

Chapter 7 - Memory/Peripheral Controller (MPC)

7.1 Overview 7-1

7.2 Architecture 7-2

7.3 Operational Description.. 7-2

7.3.1 Memory Areas 7-3
7.3.2 Signal Relationships 7-4
7.3.3 Wait State Insertion 7-5
7.3.4 Instruction Fetches from Memory 7-8
7.3.5 Data Transfers to and from Memory 7-10
7.3.6 Endian configuration 7-12
7.3.7 Access to Non-Aligned Memory Addresses 7-14

7.4 Programmer's Model 7-15

7.4.1 MPC Configuration Registers 7-15

7.5 External Interfaces 7-18

7.6 Application Information: Designing a Memory System .. 7-19
7.6.1 Example system configuration 7-19
7.6.2 MPC Configuration Register Settings 7-22
7.6.3 Calculating Required Memory Timing Parameters............. 7-23

Chapter 8 - Universal Asynchronous Receiver/ Transmitter (UART)

8.1 Overview 8-1
8.2 Operational Description 8-3
8.2.1 Baud Rate Generation 8-3
8.2.2 Transmit Channel 8-4
8.2.3 Receive Channel 8-5
8.2.4 Receive Data Filter... 8-6
8.2.5 Data Transfer Methods 8-7
8.2.6 Manual Flow Control 8-8
8.2.7 Automatic Flow Control 8-8
8.2.8 Modem Flow Control 8-9
8.2.9 Null Modem Flow Control.. 8-10
8.3 Programmer's Model.. 8-11
8.3.1 Register Descriptions 8-12

Chapter 9 - Interrupt Controller (INTC)

9.1 Introduction 9-1
9.1.1 Design Features 9-1
9.2 Architecture 9-2
9.3 Operational Description 9-3
9.3.1 Interrupt Controller Structure.. 9-3
9.3.2 Interrupt Processor... 9-3
9.3.3 FIQ Priority Encoder... 9-4
9.3.4 Bus Interface Control.. 9-4
9.3.5 External Interface 9-5
9.4 Programmer's Model.. 9-5
9.5 Using the Interrupt Controller.. 9-6

Chapter 10 - DMA Controller (DMAC)

10.1 Overview 10-1
10.1.1 DMA Controller Trigger Selection 10-2
10.2 Operational Description 10-3
10.2.1 Single Addressed (Fly-by) Transfer 10-3
10.2.2 Dual Addressed (Buffered) Transfer 10-6
10.2.3 Configuration 10-10
10.3 Programmer's Model 10-13
10.3.1 DMA Registers 10-13

Chapter 11 - Timer/Counter (TIC)

11.1 Overview 11-1
11.1.1 Design Features 11-1
11.2 Architecture 11-2
11.3 Operational Description 11-2
11.3.1 Prescaler Operation 11-3
11.3.2 Halt on Zero (Mode 0) 11-3
11.3.3 Free Running (Mode 1) 11-3
11.3.4 Reload on Trigger (Mode 2) 11-4
11.3.5 Pulse Width Modulation (Mode 3) 11-4
11.4 Programmer's Model 11-5
11.4.1 Register Descriptions 11-5

Chapter 12 - Watchdog Timer (WDOG)

12.1 Overview 12-1
12.1.1 Design Features 12-1
12.2 Architecture 12-1
12.3 Operational Description 12-2
12.3.1 Timer Operation and Watchdog Restart Key 12-2
12.4 Programmer's Model 12-3
12.4.1 Control Register Format 12-3

12.5 External Interface 12-4

Appendix A - BmILD Bus Operation

A.1 Introduction A-1
A.1.1 Bus Masters A-1
A.1.2 Bus Slaves A-2
A.1.3 System Arbitration and Multiple Bus Master Support.......... A-2

Appendix B - Physical and Electrical Specifications

B.1 Device I/O Summary B-1
B.2 Pin Position Details B-3
B.3 Package Options... B-5
B.4 Electrical performance Characteristics. B-6

Appendix C - Further Information

C.1 Related Documents C-1
C.2 Worldwide Offices C-3

Appendix D - PCB Layout Guidelines

D.1 Considerations regarding Power distribution D-1
D.2 Considerations regarding PCB track lengths D-2

LIST OF FIGURES

Chapter 1 - Introduction

Figure 1-1: Functional Block Diagram - BUTTERFLY 1-2
Figure 1-3: Timer Counter connectivity 1-14

Chapter 2 - ARM7 Microprocessor

Figure 2-1: ARM7 Architecture 2-2
Figure 2-2: Register Organisation 2-5
Figure 2-3: Format of the Program Status Registers (PSRs) 2-6
Figure 2-4: Instruction Set Summary 2-12
Figure 2-5: Condition Codes 2-13
Figure 2-6: Branch Instructions 2-14
Figure 2-7: Data Processing Instructions 2-16
Figure 2-8: ARM Shift Operations 2-18
Figure 2-9: Logical Shift Left 2-19
Figure 2-10: Logical Shift Right 2-19
Figure 2-11: Arithmetic Shift Right 2-20
Figure 2-12: Rotate Right 2-21
Figure 2-13: Rotate Right Extended 2-21
Figure 2-14: PSR Transfer 2-28
Figure 2-15: Multiply Instructions 2-29
Figure 2-16: Single Data Transfer Instructions 2-32
Figure 2-17: Little Endian Offset Addressing 2-34
Figure 2-18: Block Data Transfer Instructions 2-38
Figure 2-19: Post-Increment Addressing 2-40
Figure 2-20: Pre-Increment Addressing 2-40
Figure 2-21: Post-Decrement Addressing 2-41
Figure 2-22: Pre-Decrement Addressing 2-41
Figure 2-23: Swap Instruction 2-45
Figure 2-24: Software Interrupt Instruction 2-47
Figure 2-25: Coprocessor Data Operation Instruction 2-50
Figure 2-26: Undefined Instruction 2-52

Chapter 3 - Diagnostic Broadcast (BBM)

Figure 3-1: B ILD cycle and bus master details on 'Bdiag pins 3-2

Chapter 4 - Phase Locked Loop (PLL)

Figure 4-1: Architecture 4-2
Figure 4-2: PLL Operational ranges 4-4
Figure 4-3: PLL circuit for User supplied clock. 4-5
Figure 4-4: PLL circuit for Crystal supplied clock. 4-5
Figure 4-5: PLL circuit for PLL bypass mode. 4-6
Figure 4-6: External Components 4-8

Chapter 5 - Power Control (POCO)

Figure 5-1: Block Diagram of POCO 5-2

Chapter 6 - Programmable Peripheral Interface (PPI)

Figure 6-1: PPI Module Block Diagram 6-1
Figure 6-2: Data Direction Register (Read/Write) 6-4
Figure 6-3: Data Input Register (Read Only) 6-5
Figure 6-4: Data Output Register (Read/Write) 6-5
Figure 6-5: 8-Bit Port Control Status Register (CSR) 6-6
Figure 6-6: Port Data Output (Modes 1 and 2) 6-7
Figure 6-7: Port Data Input (Modes 1 and 2) 6-7

Chapter 7 - Memory/Peripheral Controller (MPC)

Figure 7-1: MPC Functional Blocks 7-2
Figure 7-2: MPC External Interface Signals 7-4
Figure 7-3: Effect of Wait and Stop States on a Read Access 7-6
Figure 7-4: MPC Externally Generated Wait State Insertion 7-8
Figure 7-5: Instruction Fetch from 32-bit Memory (one wait state) 7-9
Figure 7-6: Instruction Fetch from 16-bit Memory (one wait state) 7-9
Figure 7-7: Instruction Fetch from 8-bit Memory (one wait state) 7-10
Figure 7-8: Writing to Individual Bytes of Memory 7-12
Figure 7-9: Little Endian addresses of bytes within words 7-13
Figure 7-10: Big Endian addresses of bytes within words 7-13
Figure 7-11: Single Register Slice 7-15
Figure 7-12: Configuration Register Reset State 7-17
Figure 7-13: Example Little-Endian System Configuration 7-20
Figure 7-14: Example Big-Endian System Configuration 7-21
Figure 7-15: Example MPC Configuration Register Settings 7-23
Figure 7-16: Interfacing the MPC to SRAM 7-24

Chapter 8 - Universal Asynchronous Receiver/Transmitter (UART)

Figure 8-1: Block Diagram 8-2
Figure 8-2: The Clock Chain 8-3
Figure 8-3: Serial Transmission Example 8-4
Figure 8-4: Receive data filter action 8-6
Figure 8-5: Configuration 0 - Modem Flow Control 8-9
Figure 8-6: Configuration 1 - Null modem Flow Control 8-10

Chapter 9 - Interrupt Controller (INTC)

Figure 9-1: Interrupt Controller 9-2
Figure 9-2: FIQ Encoded Priority Register 9-4

Chapter 10 - DMA Controller (DMAC)

Figure 10-1: Interface Diagram 10-2
Figure 10-2: Architecture of Single-Addressed DMA 10-3
Figure 10-3: Edge Triggered Block Transfer 10-4
Figure 10-4: Level Triggered Block Transfer 10-4
Figure 10-5: Edge Triggered Packet Transfer (Size = 2) 10-5
Figure 10-6: Software Triggered Block Transfer 10-5
Figure 10-7: Architecture of Dual Addressed DMA 10-6
Figure 10-8: Edge Triggered Block Transfer 10-7
Figure 10-9: Level Triggered Block Transfer 10-7
Figure 10-10: Edge Triggered Packet Transfer 10-8
Figure 10-11: Software Triggered Block Transfer 10-8
Figure 10-12: Software Triggered Packet Transfer 10-9

Chapter 11 - Timer/Counter (TIC)

Figure 11-1: Timer/Counter Architecture 11-2
Figure 11-2: Control/Status Register 11-6

Chapter 12 - Watchdog Timer (WDOG)

Figure 12-1: Watchdog Structure 12-1
Figure 12-2: Control Register Format 12-3
Figure 12-3: Timing Relationship Between nWden and Sclk 12-4

Appendix B - Physical and Electrical Specifications

Figure B-1: BUTTERFLY 144 Plastic Quad Flat Pack B-5

LIST OF TABLES

Chapter 1 - Introduction

Table 1-1: Address Map 1-6
Table 1-2: Module Mapping 1-7
Table 1-3: Priority Levels 1-9
Table 1-4: Register View 1-9
Table 1-5: System Configuration Register 1-10
Table 1-6: DMAC Trigger Selection 1-11
Table 1-7: Interrupt Source Channels 1-12

Chapter 2 - ARM7 Microprocessor

Table 2-1: The Mode Bits 2-6
Table 2-2: Vector Summary 2-10
Table 2-3: ARM Data Processing Instructions 2-17
Table 2-4: Addressing Mode Names 2-44
Table 2-5: Branch Instruction Cycle Operations 2-54
Table 2-6: Data Operation Instruction Cycle Operations 2-56
Table 2-7: Multiply Instruction Cycle Operations 2-57
Table 2-8: Load Register Instruction Cycle Operations 2-58
Table 2-9: Store Register Instruction Cycle Operations 2-59
Table 2-10: Load Multiple Registers Instruction Cycle Operations 2-60
Table 2-11: Store Multiple Registers Instruction Cycle Operations 2-61
Table 2-12: Data Swap Instruction Cycle Operations 2-62
Table 2-13: Software Interrupt Instruction Cycle Operations 2-63
Table 2-14: Coprocessor Data Operation Instruction Cycles 2-64
Table 2-15: Undefined Instruction Cycle Operations 2-65
Table 2-16: Unexecuted Instruction Cycle Operations 2-65
Table 2-17: ARM Instruction Speed Summary 2-66

Chapter 3 - Diagnostic Broadcast (BBM)

Table 3-1: Encoding of Bus Master ID for Diagnostic Broadcast 3-2
Table 3-2: Encoding of Cycle Type for Diagnostic Broadcast 3-3

Chapter 4 - Phase Locked Loop (PLL)

Table 4-1: Signal definitions and Names 4-3
Table 4-2: Clock Modes 4-3
Table 4-3: Specification (3V and 5V nominal) 4-12

Table 4-4: Amplifier Specification: Vdd=5V (nom) 4-12
Table 4-5: Amplifier Specification: Vdd=3V(nom) 4-12

Chapter 5 - Power Control (POCO)
Table 5-1: Programmers Register View 5-5
Table 5-2: Module Clock Enable Register (MCER) 5-5

Chapter 6 - Programmable Peripheral Interface (PPI)
Table 6-1: Programmers Register View 6-4
Table 6-2: PPI Module External Pin List Description 6-8

Chapter 7 - Memory/Peripheral Controller (MPC)
Table 7-1: Data Read from Memory 7-11
Table 7-2: Data Write to Memory 7-12
Table 7-3: Non-Aligned Address Accesses 7-14
Table 7-4: Programmer's Register View 7-15
Table 7-5: MPC nScs External Decode Map 7-15
Table 7-6: Configuration Register Bit Actions 7-16
Table 7-7: External Pin-Out Description 7-18
Table 7-8: MPC Timing Parameters 7-25
Table 7-9: Typical SRAM Parameters (with formulae) 7-26

Chapter 8 - Universal Asynchronous Receiver/Transmitter (UART)
Table 8-1: Interface Description 8-2
Table 8-2: Program Register View 8-11
Table 8-3: Serial Control Register (CR) - Read / Write, + 0x000 8-12
Table 8-4: Serial Mode Register (MR) - Read / Write, + 0x004 8-13
Table 8-5: Baud Rate Register (BRR) - Read / Write, + 0x008 8-13
Table 8-6: Serial Status Register (SR) - Read, + 0x00C 8-14
Table 8-7: Transmit Register (TR) - Write,+ 0x010 8-14
Table 8-8: Receive Register (RR) - Read, + 0x010 8-15
Table 8-9: Modem Control Register (MCR) - Read / Write,+0x 014 8-15
Table 8-10: Modem Status Register - Read, + 0x01C 8-16

Chapter 9 - Interrupt Controller (INTC)
Table 9-1: Register Map 9-5

Chapter 10 - DMA Controller (DMAC)

Table 10-1: DMA Controller Signals 10-2
Table 10-2: DMA Controller Registers 10-13
Table 10-3: DMA Controller Channel Registers 10-14
Table 10-4: Channel Control Register (CCR) - offset +0x0 10-15
Table 10-5: Channel Status Register (CSR) - offset +0x0 10-16
Table 10-6: Packet Size Register (PSR) - offset +0x4 10-17
Table 10-7: Transfer Count Registers - offset +0x8 10-17
Table 10-8: Address Registers - offset +0xC 10-18
Table 10-9: DMA Controller Status Register (DSR) - offset +0x200 10-18

Chapter 11 - Timer/Counter (TIC)

Table 11-1: Timer/Counter Address Map 11-5
Table 11-2: Control/Status Register bit descriptions 11-6

Chapter 12 - Watchdog Timer (WDOG)

Table 12-1: Register Map 12-3

Appendix B - Physical and Electrical Specifications

Table B-1: Device I/O Summary B-1

Appendix C - Further Information

Table C-1: Related Documents C-1

Document Conventions

a. Numbers prefixed with ’0x’ are hexadecimal

b. Register, flag and signal names are all in **bold** type

c. External signal names begin with a Capital letter (which are prefixed with an "n" if active low), all internal signal names are in lower case.

d. All active low signal names begin with an ’n’

e. Bit positions within a Register Field are represented within square brackets [n]

f. "Reserved": When associated with a register field, the location should not be written to, or read from. When used in a bit field amongst other referenced fields, the default value must be maintained during write operations.

g. "Low","clear": When associated with a signal or Bit field, refers to a logical condition 0.

h. "High","set": When associated with a signal or Bit field, refers to a logical condition 1

Chapter 1 - Introduction

1.1 *BUTTERFLY* microcontroller overview

This book describes the *BUTTERFLY* microcontroller which is one of a range of products designed by Mitel to offer high performance processing whilst consuming very little power. This device is an entry level microcontroller targeted for emerging markets such as digital phones, set-top boxes and games where cost sensitivity is the utmost priority. The *BUTTERFLY* is object-code compatible with earlier ARM products and fully supported with industry standard cross development tools available on both PC and Sun platforms, including the ARM toolkit (from Mitel), a development board (MAP-1), and software routines for on-chip functions.

This is one of a series of integrated controllers designed around the BμILD (Bus for μController Integration in Low-power Designs) modular architecture. This on-chip 32-bit bus architecture has been developed to facilitate fully-testable, reliable and debuggable integrated processor products. The BμILD architecture means that Mitel can produce right-first-time complex Application Specific Standard Products (ASSP's) efficiently, meeting the fast time to market needs of customers in the emerging embedded processing markets.

BUTTERFLY is fabricated in 0.7μm CMOS technology and designed to operate at 25MHz (5 Volts) or 15MHz (3 Volts) over a commercial temperature range (0-70˚C). At 5 Volts/25MHz, up to 22 MIPS (Dhrystone 2.1) of processing power is available and at 3 Volts/15MHz 13 MIPS is available.

1.1.1 Features

BUTTERFLY, available in a 144QFP package, incorporates an ARM7 processor core, the industry's 32 bit processor leader in performance efficiency (MIPS/Watt), to which Mitel has added:

- Serial I/O
- Power management control circuitry
- A programmable memory interface
- support for 8,16 and 32-bit data transfers and memory widths
- Flexible address interface – providing 4 memory areas, each of 4 Mbytes
- Flexible, 15 channel interrupt controller with programmable priority
- Watchdog Timer and four 32-bit timer/counters
- 8-bit wide Programmable Peripheral Interface (PPI)
- A Direct Memory Access (DMA) controller providing 2 fly-by channels or 1 memory channel

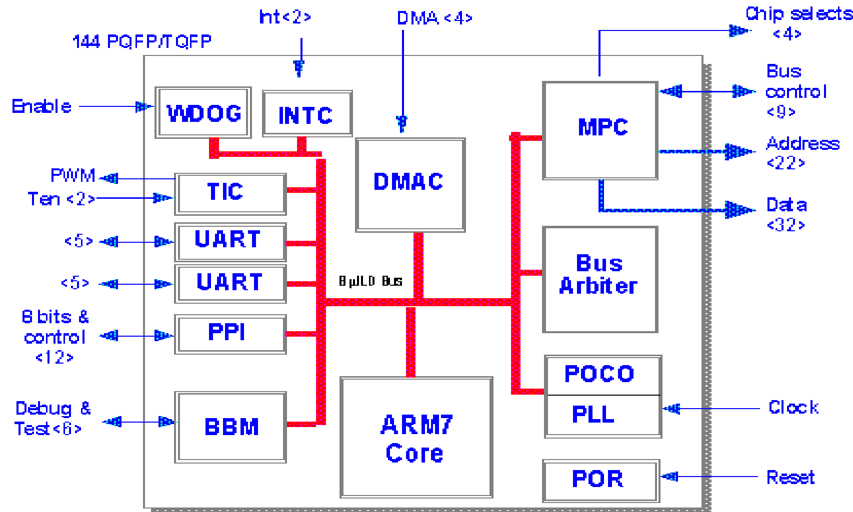


Figure 1-1: Functional Block Diagram - BUTTERFLY

1.2 Functional Block Description

1.2.1 BμILD Bus

This is a modular bus architecture and specification, via which all on-chip modules interface to each other. Such modules can either be bus masters or slaves. A bus master can initiate a bus access, generate addresses and control read or write transfers. A bus slave responds to a bus master request when selected by the system address decoder, and may, if required, assert a wait signal on the bus until the relevant data transfer has been completed. All internal data transfers on the BμILD bus are single cycle.

BUTTERFLY has three modules that are capable of operating as Bus masters. These are the ARM7 Core, DMAC and BBM, described below.

1.2.2 ARM7 Processor (ARM7)

32-bit RISC processor core, object-code compatible with all ARM6 and ARM7 based products. The ARM7 is a fully static design and as such consumes dynamic power only when clocks are active.

1.2.3 BμILD Broadcast Module (BBM)

During application debugging, the BBM provides information about the activity on the internal BμILD bus, thus providing better observability of the device. It also holds the **System Configuration Register** and controls the bus mode (e.g. **RESET, ERROR, RUN** etc.).

1.2.4 Phase Locked Loop (PLL)

This balances the mark-space ratio of the incoming clock and reduces skew between clock and I/O signals, consequently improving set-up and hold margins of Data and Control signals. The PLL is also capable of deriving a highly stable system clock from a low cost off-chip crystal.

1.2.5 Power Control (POCO)

Each on-chip module has its own clock source network. The POCO permits any or all of these clock networks to be disabled under software control to minimise power consumption in any particular module. Other power control features can be found on certain on-chip functions to provide further power-management optimization. For example the processor can be put into “SLEEP” mode so that no CPU activity occurs until reactivated with an interrupt.

1.2.6 Power-On Reset (POR)

This circuitry controls the hardware POR signal to ensure that all internal registers are set to a known state and all bus drivers are initialised to a tri-state condition at power-up. An external reset line is also provided.

1.2.7 Programmable Peripheral Interface (PPI)

8 I/O pins are provided which may be bit or byte addressed and configured in a latched or transparent mode. When in byte mode, buffer full/empty flags are available which can be used to generate an interrupt to the ARM7 processor.

1.2.8 Memory/Peripheral Controller (MPC)

The MPC ensures the correct multiplexing of data is applied for bus transfers between 8,16 or 32- bit on-chip or off-chip peripherals. Four different contiguous memory areas are available, each with an address range of 4Mbytes, with individually programmable wait and stop state generation. A “SWAP” function allows memory area “1”, which is addressed at system reset, to be switched with memory area “4”. This allows, for example, booting from ROM and then switching memory area 1 to address SRAM so that time-critical software and interrupt routines can operate from fast memory.

1.2.9 Universal Asynchronous Receiver Transmitter (UART)

Each full duplex asynchronous channel provides an RS232 type interface, which supports either a XON/XOFF software protocol, or hardware RTS/CTS handshake mechanism. The Receive and Transmit channels are double buffered. Each UART may be polled or use an interrupt scheme for BµILD bus transfers. An internal Baud rate generator can provide selectable data rates derived from on- or off-chip sources for an Rx/Tx pair. Directly triggered DMA transfers with the UART are also possible without the need for CPU intervention.

1.2.10 Interrupt Controller (INTC)

The ARM7 core accepts two types of interrupt: Normal (IRQ) and Fast (FIQ). All Interrupts on Butterfly can be switched between types, depending upon the relative priorities required.

The **INTC** is the central control logic that decodes the priority level and handles interrupt request signals from a total of 15 sources. External Interrupts can be set for edge or level sensitivity with a polarity option. To minimise interrupt latency, there is a hard-wired FIQ priority scheme for each channel, alternatively this can be ignored and the priority assessment handled in software.

1.2.11 DMA Controller (DMAC)

Two DMA engines are available on Butterfly. These may also be configured as a pair to provide a memory to memory DMA capability between any locations in the ARM7 memory space. Alternatively they may be used independently for Fly-by transfers between off-chip requestors and either on-chip or off-chip locations.

Single or multiple byte transfers (Demand or Burst Mode) are supported and may be word, half-word or byte wide.

1.2.12 Timer/Counters (TIC)

Four independent 32-bit timer/counters, each with an 8-bit prescaler capability are provided (Timers 1A, 1B, 2A and 2B). These are synchronous to the system clock and may be polled or set-up to generate interrupts on over-run and auto-reload. Timer 1A is permanently enabled. Timer 1B has an enable control input, Timers 2A and 2B, which both have enable inputs, can be configured as a pair to generate a PWM output that is connected to the PWM output pin.

1.2.13 Watchdog Timer (WDOG)

The function of the Watchdog Timer (WDOG) is to detect hardware lock-ups or run-time software errors. It performs this function by requiring the processor to write to one of its registers periodically. Should this not occur, the Watchdog will timeout and reset the system. This ensures that hardware/software lock-ups are recoverable.

1.3 *BUTTERFLY* System Details

1.3.1 System Address Map

The system address map for the *BUTTERFLY* device is shown below in Table 1-1.

Address Range	Function
0x0000 0000 >> 003F FFFF	External-1:nScs0/nScs3#
0x0040 0000 >> 1FFF FFFF	External-1 reflected*
0x2000 0000 >> 203F FFFF	External-2:nScs1
0x2040 0000 >> 3FFF FFFF	External-2 reflected*
0x4000 0000 >> 403F FFFF	External-3:nScs2
0x4040 0000 >> 5FFF FFFF	External-3 reflected*
0x6000 0000 >> 603F FFFF	External-4:nScs3/nScs0#
0x6040 0000 >> 7FFF FFFF	External-4 reflected*
0x8000 0000 >> 9FFF FFFF	Reserved
0xA000 0000 >> BFFF FFFF	Reserved
0xC000 0000 >> DFFF FFFF	Reserved
0xE000 0000 >> E001 FFFF	Internal I/O
0xE002 0000 >> FFFF FFFF	Reserved

* Reflected address areas will produce images of data in the original memory area addresses e.g. External-1 will repeat at addresses 0040 0000, 0080 0000, 00C0 0000 etc.

See section 1.3.1.2 overleaf

Table 1-1: Address Map

The Internal I/O memory area is subdivided into 32 sub-regions extending from xE000 0000 to xE001 FFFF and assigned to the on-chip modules. Those modules capable of generating addresses are configured as ‘bus masters’ and the remaining modules as ‘bus slaves’. The bus masters are:-

The ARM7 Processor

The BµILD Broadcast Module (BBM)

The DMA Controller

As only one bus master can be in control of the BµILD bus at any one time, they are prioritised and given access to the bus in accordance with the arbitration system.

1.3.1.2 SWAP Function

The **System Configuration Register** (section 1.3.6) includes a control bit that swaps the address memory areas for chip selects **nScs0** and **nScs3**. This allows ROM devices to be used for ARM bootstrap code and then allow other memory (e.g. fast SRAM) to reside in low memory address locations (including those where exception vector tables reside), once the users application code is up and running. Note if **SWAP** is used, the set-up parameters for the relative external devices may also need reprogramming as described in section 7.6.2

1.3.2 Address Map for internal I/O

Table 1-2 shows how registers for the System and individual modules are allocated memory areas within the Internal I/O memory space. Refer to the relevant chapter for further register descriptions.

Nb: Attempts to access an address that has no mapped register, or one that is designated as reserved, are likely to result in a run-time bus error.

Address Range	Function	Address Range	Function
0xE000 0000- > 1FFF	Reserved	0xE000 D000- > DFFF	Reserved
0xE000 2000- > 2FFF	System Configuration (including BBM)	0xE000 E000- > EFFF	Timer 1
0xE000 3000- > 3FFF	Reserved	0xE000 F000- > FFFF	Timer 2
0xE000 4000- > 4FFF	Watchdog Timer	0xE001 0000- > 1FFF	Reserved
0xE000 5000- > 5FFF	Power Control	0xE001 2000- > 2FFF	PPI
0xE000 6000- > 6FFF	Interrupt Controller	0xE001 3000- > 7FFF	Reserved
0xE000 7000- >7FFF	Reserved	0xE001 8000- > 8FFF	UART 1
0xE000 8000- > 8FFF	Memory/Peripheral Controller	0xE001 9000- > 9FFF	UART 2
0xE000 9000- > BFFF	Reserved	0xE001 A000- > AFFF	Reserved (Internal)
0xE000 C000- > CFFF	DMA controller	0xE001 B000> FFFF FFFF	Reserved

Table 1-2: Module Mapping

1.3.3 System Clock Control

Extensive control of the system clock distribution is provided by a Power Control (POCO) module. This provides both sleep and programmable standby modes. See Chapter 5 - Power Control (POCO) for more details.

1.3.4 System Reset

The microcontroller system may be reset by a number of mechanisms:

- At power-up
- via an external reset pin **nSreset**
- via the watchdog timing out

In each case the initialization sequence is the same.

- The microcontroller enters **RESET** mode, resetting all internal modules, including the ARM CPU
- The ARM7 processor is granted the bus and proceeds to fetch its next instruction from address location 0x0000 0000.
- The MPC defaults to accessing external area 1 as slow memory (with 15 wait states and 15 stop states, and with a width defined by pins **Sr_Size<1:0>**.
- Following successful Butterfly initialisation, the user should configure the MPC appropriate to the external devices connected to it.

IMPORTANT NOTE: Immediately after power-up and/or a rising edge of **nSreset**, **Sclk** should be clocked for at least 2 ARM **processor** cycles to bring the core into a stable state. If this is not performed then several mA can be consumed by the ARM core until these clocks have been provided.

1.3.5 System Bus Arbitration

The system contains three separate bus masters, (Modules capable of generating addresses), these being the ARM7 Processor, the DMA controller, and the BµILD Broadcast Module (BBM). The arbitration system uses a “highest priority” scheme. Bus masters request use of the Bus from the Arbiter. The highest priority bus master is granted the bus for as long as it wishes irrespective of the demands of lower priority bus masters. The priority levels implemented are as shown in Table 1-3

.

Priority Level	Function
0- Highest	BBM
1	DMA
2- Lowest	ARM7

Table 1-3: Priority Levels

The BBM block is primarily used for manufacturing test and system debug and hence requires the highest priority level, although it is not active during normal system operation. The ARM7 processor is bandwidth intensive, if not given the lowest priority then any lower priority master would rarely get access to the bus. A further feature allows the ARM to have priority on receipt of a FIQ interrupt. In this instance the ARM may be optionally promoted in priority to be above the DMA for the duration of any FIQ. Bit 1 in the **System Configuration Register** enables this function.

1.3.6 System Configuration Register

After power-up, this register is should be initialised with suitable values for desired operation of the device. It also determines whether the Diagnostic broadcast facility is enabled or not.

Address	Register view	Function
0xE000 2000	Reserved (must remain '0')	RW
0xE000 2004	System Configuration Register	RW

Table 1-4: Register View

The **System Configuration Register** controls the operation of various system functions.

- The SWAP bit controls the operation of **nScs0** and **nScs3** as described in section 1.3.1
- Bit 1 controls the operation of the bus arbitration unit as described in section 1.3.5
- Bit 2 is reserved
- Bits 3-5 control the selection of the DMA request input to the DMA controller described in Chapter 10 - DMA Controller (DMAC)
- Bits 6-7 select the UART's to work in either external clock (1) or DCD (0) mode.
- Bits 8-31 are reserved and should be programmed to 0.

Bit	Function	Reset Value
0	SWAP [1] swaps external nScs0 and nScs3 addresses	0
1	Arbitration ARM (FIQ mode) promote	0
2	Reserved (must be 0)	0
3-5	DMA mux-1 channel select: see below	0
6	UART1 [1] external clock / [0] nDCD enable pin	0
7	UART2 [1] external clock / [0] nDCD enable pin	0
8-31	Reserved (must be 0)	0

Table 1-5: System Configuration Register

1.3.7 DMA

The following table describes the encoding system for allocating DMA channels to sources that have a DMA capability. More details can be found in Chapter 10 - DMA Controller (DMAC)

Channel Select System Configuration Register			Source Channel
Bit 5	**Bit 4**	**Bit 3**	
0	0	0	External Dreq-1
0	0	1	UART 2 Receive
0	1	0	PPI 1 IBF
0	1	1	UART 1 Receive
1	0	0	External Interrupt-1
1	0	1	UART 2 Transmit
1	1	0	PPI 1 OBE
1	1	1	UART 1 transmit

Table 1-6: DMAC Trigger Selection

1.3.8 Interrupt Sources

The following is the channel allocation number for each Bμild module. More information on interrupt control may be found in Chapter 9 - Interrupt Controller (INTC).

Channel	Interrupt Source	Function
0 (highest priority)	WDOG	“Bark”
1	Reserved	
2	Reserved	
3	DMAC	
4	Reserved	
5	Reserved	
6	TIC1	Time out A
7	TIC1	Time out B
8	External Interrupt 1	User defined
9	Reserved	
10	Reserved	
11	PPI	IBF
12	PPI	OBE
13	Reserved	
14	UART 1	Error
15	UART 1	RBF
16	UART 1	TBE
17	Reserved	
18	TIC2	Time out A
19	TIC2	Time out B
20	External Interrupt 2	User defined
21	Reserved	
22	Reserved	
23	Reserved	

Table 1-7: Interrupt Source Channels

Channel	Interrupt Source	Function
24	Reserved	
25	Reserved	
26	UART 2	Error
27	UART 2	RBF
28	UART 2	TBE
29	Reserved	
30	Reserved	
31(lowest priority)	Reserved	

Table 1-7: Interrupt Source Channels (Continued)

The four timer interrupts (channels 6, 7, 18, and 19) are edge triggered. All other internally generated interrupt sources are level sensitive, active *HIGH*.

Directly driven external interrupts are user defined, and can be set up for either Edge or Level sensitivity. For these, the edge/level and polarity registers need to be set up to match the required operation. The values that need to be written are:

edge/level	0x000C00C0	**external interrupts are level sensitive**
	0x001C01C0	**external interrupts are edge sensitive**
polarity	0xFFFFFFFF	**external interrupts are active high or rising edge triggered**
	0xFFEFFEFF	**external interrupts are active low or falling edge triggered**

1.3.9 Timer Counter (TIC) configuration

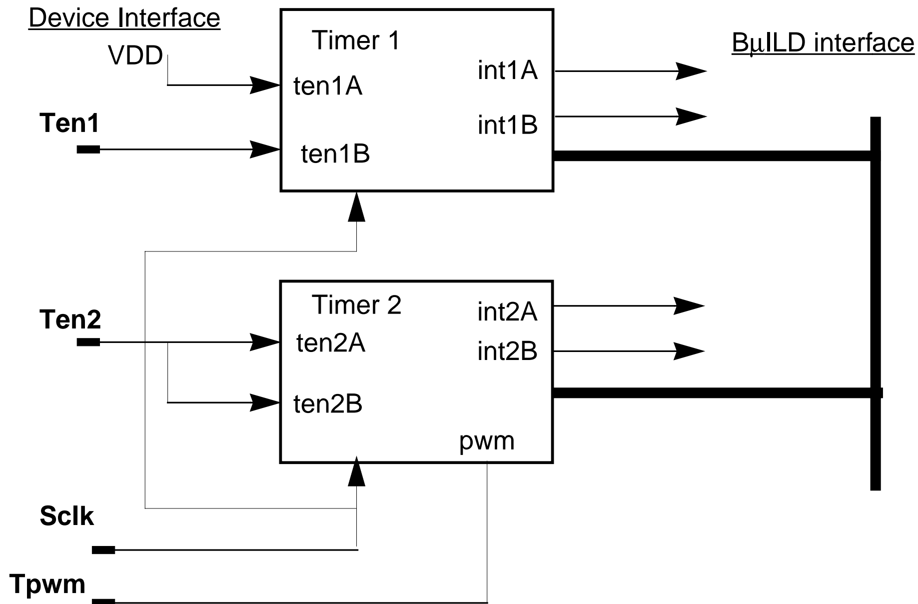


Figure 1-3: Timer Counter connectivity

Timer Counters “1” and “2” are connected as shown in Figure 1-3: Timer Counter connectivity. Please refer to Chapter 11 - Timer/Counter (TIC) for more details

Chapter 2 - ARM7 Microprocessor

2.1 Overview

The ARM7 is part of the Advanced RISC Machines (ARM) family of general purpose 32-bit microprocessors, which offer very low power consumption and low price for high performance devices. The architecture is based on Reduced Instruction Set Computer (RISC) principles, and the instruction set and related decode mechanism are much simpler in comparison with microprogrammed Complex Instruction Set Computers. This results in a high instruction throughput and impressive real-time interrupt response from a small and cost-effective chip.

The instruction set comprises eleven basic instruction types:

- Two of these make use of the on-chip arithmetic logic unit, barrel shifter and multiplier to perform high-speed operations on the data in a bank of 31 registers, each 32 bits wide;
- Three classes of instruction control data transfer between memory and the registers, one optimised for flexibility of addressing, another for rapid context switching and the third for swapping data;
- Three instructions control the flow and privilege level of execution; and
- Three types are dedicated to the control of external coprocessors which allow the functionality of the instruction set to be extended in an open and uniform way.

The ARM instruction set is a good target for compilers of many different high-level languages. Where required for critical code segments, assembly code programming is also straightforward, unlike some RISC processors which depend on sophisticated compiler technology to manage complicated instruction interdependencies.

Pipelining is employed so that all parts of the processing and memory systems can operate continuously. Typically, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory.

ARM7 is a fully static CMOS implementation of the ARM which allows the clock to be stopped in any part of the cycle with extremely low residual power consumption and no loss of state.

2.2 Architecture

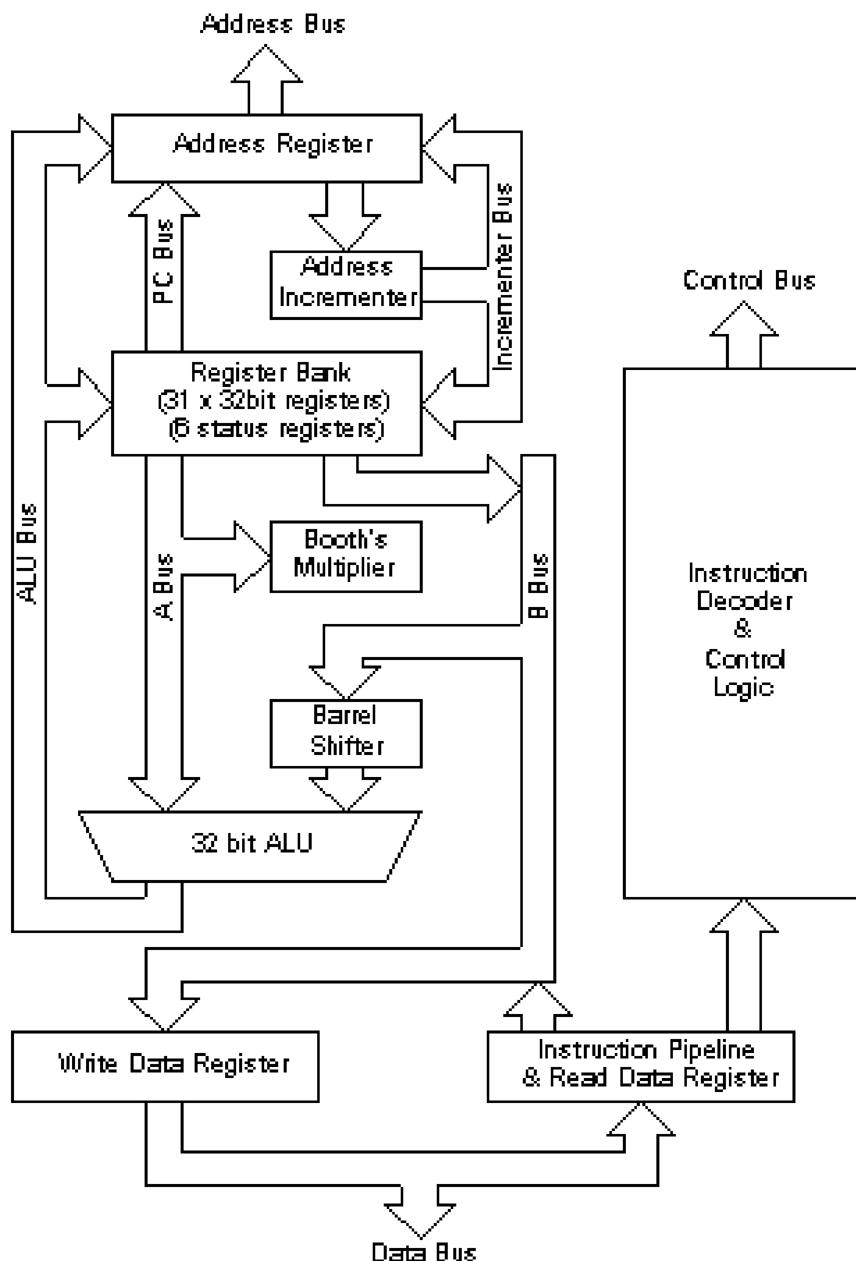


Figure 2-1: ARM7 Architecture

2.3 Programmer's Model

ARM7 supports a variety of operating configurations. One is controlled by an input and is known as the *hardware configuration*. Others may be controlled by software and these are known as *operating modes*. On Butterfly's internal Bµild bus, the ARM7's lowest BIGEND control bit, which determines whether the lowest addressed byte is least of most significant in a multi-byte word, is set (i.e. Little Endian). However for off-chip accesses, this may be programmed through the MPC using the Sbigendian pin. Please see Chapter 6 for further information.

2.3.1 Operating Mode Selection

ARM7 has a 32 bit data bus and a 32 bit address bus. The data types the processor supports are Bytes (8 bits) and Words (32 bits), where words must be aligned to four byte boundaries. Instructions are exactly one word, and data operations (e.g. ADD) are only performed on word quantities. Load and store operations can transfer either bytes or words.

ARM7 supports six modes of operation:

[1] User mode (**usr**): the normal program execution state

[2] FIQ mode (**fiq**): designed to support a data transfer or channel process

[3] IRQ mode (**irq**): used for general purpose interrupt handling

[4] Supervisor mode (**svc**): a protected mode for the operating system

[5] Abort mode (**abt**): entered after a data or instruction prefetch abort

[6] Undefined mode (**und**): entered when an undefined instruction is executed

Mode changes may be made under software control or may be brought about by external interrupts or exception processing. Most application programs will execute in User mode. The other modes, known as *privileged modes*, will be entered to service interrupts or exceptions or to access protected resources.

2.3.2 Registers

The processor has a total of 37 registers made up of 31 general 32 bit registers and 6 status registers. At any one time 16 general registers (R0 to R15) and one or two status registers are visible to the programmer. The visible registers depend on the processor mode and the other registers (the *banked registers*) are switched in to support IRQ, FIQ, Supervisor, Abort and Undefined mode processing. The register bank organisation is shown in *Figure 2-2: Register Organisation*. The banked registers are shaded in the diagram.

In all modes 16 registers, R0 to R15, are directly accessible. All registers except R15 are general purpose and may be used to hold data or address values. Register R15 holds the Program Counter (PC). When R15 is read, bits [1:0] are zero and bits [31:2] contain the PC. A seventeenth register (the CPSR - Current Program Status Register) is also accessible. It contains condition code flags and the current mode bits and may be thought of as an extension to the PC.

R14 is used as the subroutine link register and receives a copy of R15 when a Branch and Link instruction is executed. It may be treated as a general purpose register at all other times. R14_svc, R14_irq, R14_fiq, R14_abt and R14_und are used similarly to hold the return values of R15 when interrupts and exceptions arise, or when Branch and Link instructions are executed within interrupt or exception routines.

FIQ mode has seven banked registers mapped to R8-14 (R8_fiq-R14_fiq). Many FIQ programs will not need to save any registers. User mode, IRQ mode, Supervisor mode, Abort mode and Undefined mode each have two banked registers mapped to R13 and R14. The two banked registers allow these modes to each have a private stack pointer and link register. Supervisor, IRQ, Abort and Undefined mode programs which require more than these two banked registers are expected to save some or all of the caller's registers (R0 to R12) on their respective stacks. They are then free to use these registers which they will restore before returning to the caller. In addition there are also five SPSRs (Saved Program Status Registers) which are loaded with the CPSR when an exception occurs. There is one SPSR for each privileged mode.

General Registers and Program Counter Modes

User32	FIQ32	Supervisor32	Abort32	IRQ32	Undefined32
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

Program Status Registers

User32	FIQ32	Supervisor32	Abort32	IRQ32	Undefined32
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

Figure 2-2: Register Organisation

The format of the Program Status Registers is shown in *Figure 2-3: Format of the Program Status Registers (PSRs)*. The N, Z, C and V bits are the *condition code flags*. The condition code flags in the CPSR may be changed as a result of arithmetic and logical operations in the processor and may be tested by all instructions to determine if the instruction is to be executed.

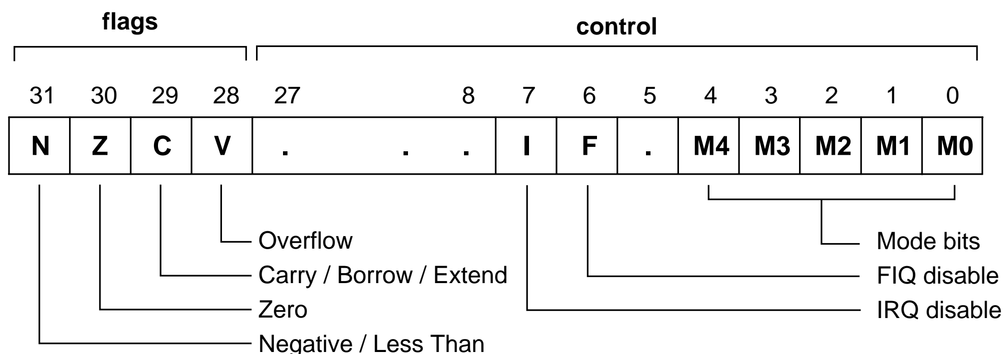


Figure 2-3: Format of the Program Status Registers (PSRs)

The I and F bits are the *interrupt disable bits.* The I bit disables IRQ interrupts when it is set and the F bit disables FIQ interrupts when it is set. The M0, M1, M2, M3 and M4 bits (M[4:0]) are the *mode bits*, and these determine the mode in which the processor operates. The interpretation of the mode bits is shown in *Table 2-1: The Mode Bits*. Not all combinations of the mode bits define a valid processor mode. Only those explicitly described shall be used.

The bottom 28 bits of a PSR (incorporating I, F and M[4:0]) are known collectively as the *control bits*. The control bits will change when an exception arises and in addition can be manipulated by software when the processor is in a privileged mode. Unused bits in the PSRs are reserved and their state shall be preserved when changing the flag or control bits. Programs shall not rely on specific values from the reserved bits when checking the PSR status, since they may read as one or zero in future processors.

M[4:0]	Mode	Accessible register set	
10000	User	PC,R14..R0	CPSR
10001	FIQ	PC,R14_fiq..R8_fiq,R7..R0	CPSR, SPSR_fiq
10010	IRQ	PC,R14_irq..R13_irq,R12..R0	CPSR, SPSR_irq
10011	Supervisor	PC,R14_svc..R13_svc,R12..R0	CPSR, SPSR_svc
10111	Abort	PC,R14_abt..R13_abt,R12..R0	CPSR, SPSR_abt
11011	Undefined	PC,R14_und..R13_und, R12..R0	CPSR, SPSR_und

Table 2-1: The Mode Bits

2.3.3 Exceptions

Exceptions arise whenever there is a need for the normal flow of program execution to be broken, so that (for example) the processor can be diverted to handle an interrupt from a peripheral. The processor state just prior to handling the exception must be preserved so that the original program can be resumed when the exception routine has completed. Many exceptions may arise at the same time.

ARM7 handles exceptions by making use of the banked registers to save state. The old PC and CPSR contents are copied into the appropriate R14 and SPSR and the PC and mode bits in the CPSR bits are forced to a value which depends on the exception. Interrupt disable flags are set where required to prevent otherwise unmanageable nestings of exceptions. In the case of a re-entrant interrupt handler, R14 and the SPSR should be saved onto a stack in main memory before re-enabling the interrupt; when transferring the SPSR register to and from a stack, it is important to transfer the whole 32 bit value, and not just the flag or control fields. When multiple exceptions arise simultaneously, a fixed priority determines the order in which they are handled. The priorities are listed later in this chapter.

In Butterfly, all interrupts are controlled by the Interrupt Controller described in Chapter 8. This provides the capability of multiple interrupt source channels.

2.3.4 FIQ

The FIQ (Fast Interrupt reQuest) exception is externally generated by taking the **nFIQ** input LOW. This input can accept asynchronous transitions, and is delayed by one clock cycle for synchronisation before it can affect the processor execution flow. It is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications (thus minimising the overhead of context switching). The FIQ exception may be disabled by setting the F flag in the CPSR (but note that this is not possible from User mode). If the F flag is reset, ARM7 checks for a LOW level on the output of the FIQ synchroniser at the end of each instruction.

When a FIQ is detected, ARM7 performs the following:

[1] Saves the address of the next instruction to be executed plus 4 in R14_fiq; saves CPSR in SPSR_fiq

[2] Forces M[4:0]=10001 (FIQ mode) and sets the F and I bits in the CPSR

[3] Forces the PC to fetch the next instruction from address 0x1C

To return normally from FIQ, use SUBS PC, R14_fiq,#4 which will restore both the PC (from R14) and the CPSR (from SPSR_fiq) and resume execution of the interrupted code.

2.3.5 IRQ

The IRQ (Interrupt ReQuest) exception is a normal interrupt caused by a LOW level on the **nIRQ** input. It has a lower priority than FIQ, and is masked out when a FIQ sequence is entered. Its effect may be masked out at any time by setting the I bit in the CPSR (but note that this is not possible from User mode). If the I flag is reset, ARM7 checks for a LOW level on the output of the IRQ synchroniser at the end of each instruction. When an IRQ is detected, ARM7 performs the following:

[1] Saves the address of the next instruction to be executed plus 4 in R14_irq; saves CPSR in SPSR_irq

[2] Forces M[4:0]=10010 (IRQ mode) and sets the I bit in the CPSR

[3] Forces the PC to fetch the next instruction from address 0x18

To return normally from IRQ, use SUBS PC,R14_irq,#4 which will restore both the PC and the CPSR and resume execution of the interrupted code.

2.3.6 Abort

An ABORT can be signalled by the external **ABORT** input. ABORT indicates that the current memory access cannot be completed. For instance, in a virtual memory system the data corresponding to the current address may have been moved out of memory onto a disc, and considerable processor activity may be required to recover the data before the access can be performed successfully. ARM7 checks for ABORT during memory access cycles. When successfully aborted ARM7 will respond in one of two ways:

[1] If the abort occurred during an instruction prefetch (a *Prefetch Abort*), the pre-fetched instruction is marked as invalid but the abort exception does not occur immediately. If the instruction is not executed, for example as a result of a branch being taken while it is in the pipeline, no abort will occur. An abort will take place if the instruction reaches the head of the pipeline and is about to be executed.

[2] If the abort occurred during a data access (a *Data Abort*), the action depends on the instruction type.

  [a] Single data transfer instructions (LDR, STR) will write back modified base registers and the Abort handler must be aware of this.

  [b] The swap instruction (SWP) is aborted as though it had not executed, though externally the read access may take place.

  [c] Block data transfer instructions (LDM, STM) complete, and if write-back is set, the base is updated. If the instruction would normally have over-written the base with data (i.e. LDM with the base in the transfer list), this over-writing is

prevented. All register over-writing is prevented after the Abort is indicated, which means in particular that R15 (which is always last to be transferred) is preserved in an aborted LDM instruction.

When either a prefetch or data abort occurs, ARM7 performs the following:

[1] Saves the address of the aborted instruction plus 4 (for prefetch aborts) or 8 (for data aborts) in R14_abt; saves CPSR in SPSR_abt.

[2] Forces M[4:0]=10111 (Abort mode) and sets the I bit in the CPSR.

[3] Forces the PC to fetch the next instruction from either address 0x0C (prefetch abort) or address 0x10 (data abort).

To return after fixing the reason for the abort, use SUBS PC,R14_abt,#4 (for a prefetch abort) or SUBS PC,R14_abt,#8 (for a data abort). This will restore both the PC and the CPSR and retry the aborted instruction.

The abort mechanism allows a *demand paged virtual memory system* to be implemented when suitable memory management software is available. The processor is allowed to generate arbitrary addresses, and when the data at an address is unavailable the MMU signals an abort. The processor traps into system software which must work out the cause of the abort, make the requested data available, and retry the aborted instruction. The application program needs no knowledge of the amount of memory available to it, nor is its state in any way affected by the abort.

2.3.7 Software Interrupt

The software interrupt instruction (SWI) is used for getting into Supervisor mode, usually to request a particular supervisor function. When a SWI is executed, ARM7 performs the following:

[1] Saves the address of the SWI instruction plus 4 in R14_svc; saves CPSR in SPSR_svc

[2] Forces M[4:0]=10011 (Supervisor mode) and sets the I bit in the CPSR

[3] Forces the PC to fetch the next instruction from address 0x08

To return from a SWI, use MOVS PC,R14_svc. This will restore the PC and CPSR and return to the instruction following the SWI.

2.3.8 Undefined Instruction Trap

When the ARM7 comes across an instruction which it cannot handle, it will take the undefined instruction trap.

The trap may be used for software emulation of a coprocessor in a system which does not have the coprocessor hardware, or for general purpose instruction set extension by software emulation.

When ARM7 takes the undefined instruction trap it performs the following:

[1] Saves the address of the Undefined or coprocessor instruction plus 4 in R14_und; saves CPSR in SPSR_und.

[2] Forces M[4:0]=11011 (Undefined mode) and sets the I bit in the CPSR

[3] Forces the PC to fetch the next instruction from address 0x04

To return from this trap after emulating the failed instruction, use MOVS PC,R14_und. This will restore the CPSR and return to the instruction following the undefined instruction.

2.3.9 Vector Summary

Address	Exception	Mode on entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	-- reserved --	--
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

Table 2-2: Vector Summary

These are byte addresses, and will normally contain a branch instruction pointing to the relevant routine.

The FIQ routine might reside at 0x1C onwards, and thereby avoid the need for (and execution time of) a branch instruction.

2.3.10 Exception Priorities

When multiple exceptions arise at the same time, a fixed priority system determines the order in which they will be handled:

[1] Reset (highest priority)

[2] Data abort

[3] FIQ

[4] IRQ

[5] Prefetch abort

[6] Undefined Instruction, Software interrupt (lowest priority)

Note: Not all exceptions can occur at once. Undefined instruction and software interrupt are mutually exclusive since they each correspond to particular (non-overlapping) decodings of the current instruction.

If a data abort occurs at the same time as a FIQ, and FIQs are enabled (i.e. the F flag in the CPSR is reset), ARM7 will enter the data abort handler and then immediately proceed to the FIQ vector. A normal return from FIQ will cause the data abort handler to resume execution. Placing data abort at a higher priority than FIQ is necessary to ensure that the transfer error does not escape detection; the time for this exception entry should be added to worst case FIQ latency calculations.

2.3.11 Interrupt Latencies

The worst case latency for FIQ, assuming that it is enabled, consists of the longest time the request can take to pass through the synchroniser (*Tsyncmax*), plus the time for the longest instruction to complete (*Tldm*, the longest instruction is an LDM which loads all the registers including the PC), plus the time for the data abort entry (*Texc*), plus the time for FIQ entry (*Tfiq*). The ARM7 will then execute the instruction at 0x1C.

Tsyncmax is 3 processor cycles, *Tldm* is 20 cycles, *Texc* is 3 cycles, and *Tfiq* is 2 cycles. The total time is therefore 28 processor cycles. This is just over 1.4 microseconds in a system which uses a continuous 20 MHz processor clock. The maximum IRQ latency calculation is similar, but must allow for the fact that FIQ has higher priority and could delay entry into the IRQ handling routine for an arbitrary length of time. The minimum latency for FIQ or IRQ consists of the shortest time the request can take through the synchroniser (*Tsyncmin*) plus *Tfiq*. This is 4 processor cycles.

2.3.12 Reset

When the **nSreset** signal goes LOW, ARM7 abandons the executing instructions and then continues to fetch instructions from incrementing word addresses.

When **nSreset** goes HIGH again, ARM7 does the following:

[1] Over-writes R14_svc and SPSR_svc by copying the current values of the PC and CPSR into them. The value of the saved PC and CPSR is not defined.

[2] Forces M[4:0]=10011 (Supervisor mode) and sets the I and F bits in the CPSR.

[3] Forces the PC to fetch the next instruction from address 0x0000 0000

2.4 Instruction Set

2.4.1 Instruction Set Summary

A summary of the ARM7 instruction set is shown in *Figure 2-4: Instruction Set Summary*.

Bit positions: 31 28 27 26 25 24 23 22 21 20 19 16 15 12 11 8 7 5 4 3 0

Cond	0 0	I	Opcode	S	Rn	Rd	Operand 2				**Data Processing PSR Transfer**
Cond	0 0 0 0 0 0	A	S	Rd	Rn	Rs	1 0 0 1	Rm			**Multiply**
Cond	0 0 0 1 0	B	0 0	Rn	Rd	0 0 0 0	1 0 0 1	Rm			**Single Data Swap**
Cond	0 1	I	P	U	B	W	L	Rn	Rd	offset	**Single Data Transfer**
Cond	0 1 1	XXXXXXXXXXXXXXXXXXXX	1	XXXX							**Undefined**
Cond	1 0 0	P	U	S	W	L	Rn	Register List			**Block Data Transfer**
Cond	1 0 1	L	offset								**Branch**
Cond	1 1 0	P	U	N	W	L	Rn	CRd	CP#	offset	**Coproc Data Transfer***
Cond	1 1 1 0	CP Opc	CRn	CRd	CP#	CP	0	CRm			**Coproc Data Operation**
Cond	1 1 1 0	CP Opc	L	CRn	Rd	CP#	CP	1	CRm		**Coproc Register Transfer***
Cond	1 1 1 1	ignored by processor									**Software Interrupt**

* Not supported for external devices on Butterfly

Figure 2-4: Instruction Set Summary

Note: Some instruction codes are not defined but do not cause the Undefined instruction trap to be taken, for instance a Multiply instruction with bit 6 changed to a 1. These instructions shall not be used, as their action may change in future ARM implementations

2.4.2 The Condition Field

31 28 27

Condition Field

0000 = EQ - Z set (equal)
0001 = NE - Z (not equal)
0010 = CS - C set (unsigned higher or same)
0011 = CC - C clear (unsigned lower)
0100 = MI - N set (negative)
0101 = PL - N clear (positive or zero)
0110 = VS - V set (overflow)
0111 = VC - V clear (no overflow)
1000 = HI - C set and Z clear (unsigned higher)
1001 = LS - C clear or Z set (unsigned lower or same)
1010 = GE - N set and V set, or N clear and V clear (greater or equal)
1011 = LT - N set and V clear, or N clear and V set (less than)
1100 = GT - Z clear, and either N set and V set, or N clear and V clear (greater than)
1101 = LE Z set, or N set and V clear, or N clear and V set (less than or equal)
1110 = AL - always
1111 = NV - never

Figure 2-5: Condition Codes

All ARM7 instructions are conditionally executed, which means that their execution may or may not take place depending on the values of the N, Z, C and V flags in the CPSR. The condition encoding is shown in *Figure 2-5: Condition Codes*.

If the *always* (AL) condition is specified, the instruction will be executed irrespective of the flags. The *never* (NV) class of condition codes shall not be used as they will be redefined in future variants of the ARM architecture. If a NOP is required it is suggested that MOV R0,R0 be used. The assembler treats the absence of a condition code as though *always* had been specified.

The other condition codes have meanings as detailed in *Figure 2-5: Condition Codes*, for instance code 0000 (EQual) causes the instruction to be executed only if the Z flag is set. This would correspond to the case where a compare (CMP) instruction had found the two operands to be equal. If the two operands were different, the compare instruction would have reset the Z flag and the instruction will not be executed.

2.4.3 Branch and Branch with Link (B, BL)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 2-6: Branch Instructions*.

Branch instructions contain a signed 2's complement 24 bit offset. This is shifted left two bits, sign extended to 32 bits, and added to the PC. The instruction can therefore specify a branch of +/- 32Mbytes. The branch offset must take account of the prefetch operation, which causes the PC to be 2 words (8 bytes) ahead of the current instruction.

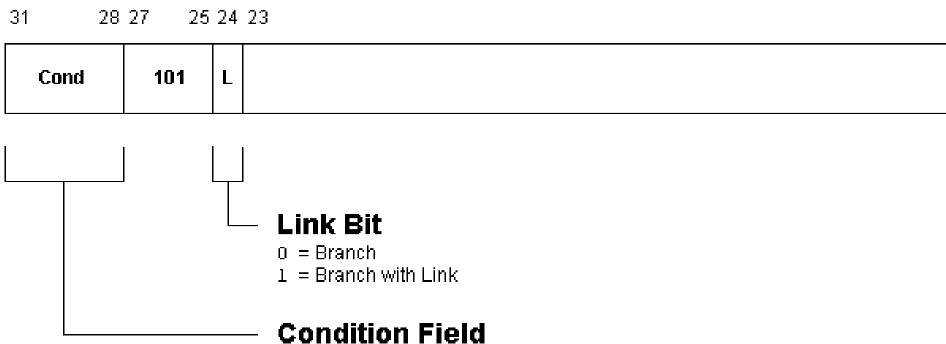


Figure 2-6: Branch Instructions

Branches beyond +/- 32Mbytes must use an offset or absolute destination which has been previously loaded into a register. In this case the PC should be manually saved in R14 if a Branch with Link type operation is required.

The Link Bit

Branch with Link (BL) writes the old PC into the link register (R14) of the current bank. The PC value written into R14 is adjusted to allow for the prefetch, and contains the address of the instruction following the branch and link instruction. Note that the CPSR is not saved with the PC.

To return from a routine called by Branch with Link use MOV PC,R14 if the link register is still valid or LDM Rn!,{..PC} if the link register has been saved onto a stack pointed to by Rn.

Instruction Cycle Times

Branch and Branch with Link instructions take 2S + 1N incremental cycles, where S and N are as defined in *Section 2.5.1, Cycle Types*.

Assembler Syntax

B{L}{cond} <expression>

{L} is used to request the Branch with Link form of the instruction. If absent, R14 will not be affected by the instruction.

{cond} is a two-char mnemonic as shown in *Figure 2-5: Condition Codes* (EQ, NE, VS etc). If absent then AL (ALways) will be used.

<expression> is the destination. The assembler calculates the offset.

Items in {} are optional. Items in <> must be present.

Examples

```
here BAL  here    ; assembles to 0xEAFFFFFE (note effect of
                  ; PC offset)

     B    there   ; Always condition used as default

     CMP  R1,#0   ; compare R1 with zero and branch to fred if R1
     BEQ  fred    ; was zero, otherwise continue to next instruction

     BL   sub+ROM; call subroutine at computed address

     ADDSR1,#1    ; add 1 to register 1, setting CPSR flags on the
     BLCCsub      ; result then call subroutine if the C flag is
                  ; reset, which will be the case unless R1 held
                  ; 0xFFFFFFFF
```

2.4.4 Data Processing

The instruction is only executed if the condition is true, defined at the beginning of this chapter. The instruction encoding is shown in *Figure 2-7: Data Processing Instructions*.

The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands. The first operand is always a register (Rn). The second operand may be a shifted register (Rm) or a rotated 8 bit immediate value (Imm) according to the value of the I bit in the instruction. The condition codes in the CPSR may be preserved or updated as a result of this instruction, according to the value of the S bit in the instruction. Certain operations (TST, TEQ, CMP, CMN) do not write the result to Rd. They are used only to perform tests and to set the condition codes on the result and always have the S bit set. The instructions and their effects are listed in *Table 2-3: ARM Data Processing Instructions*.

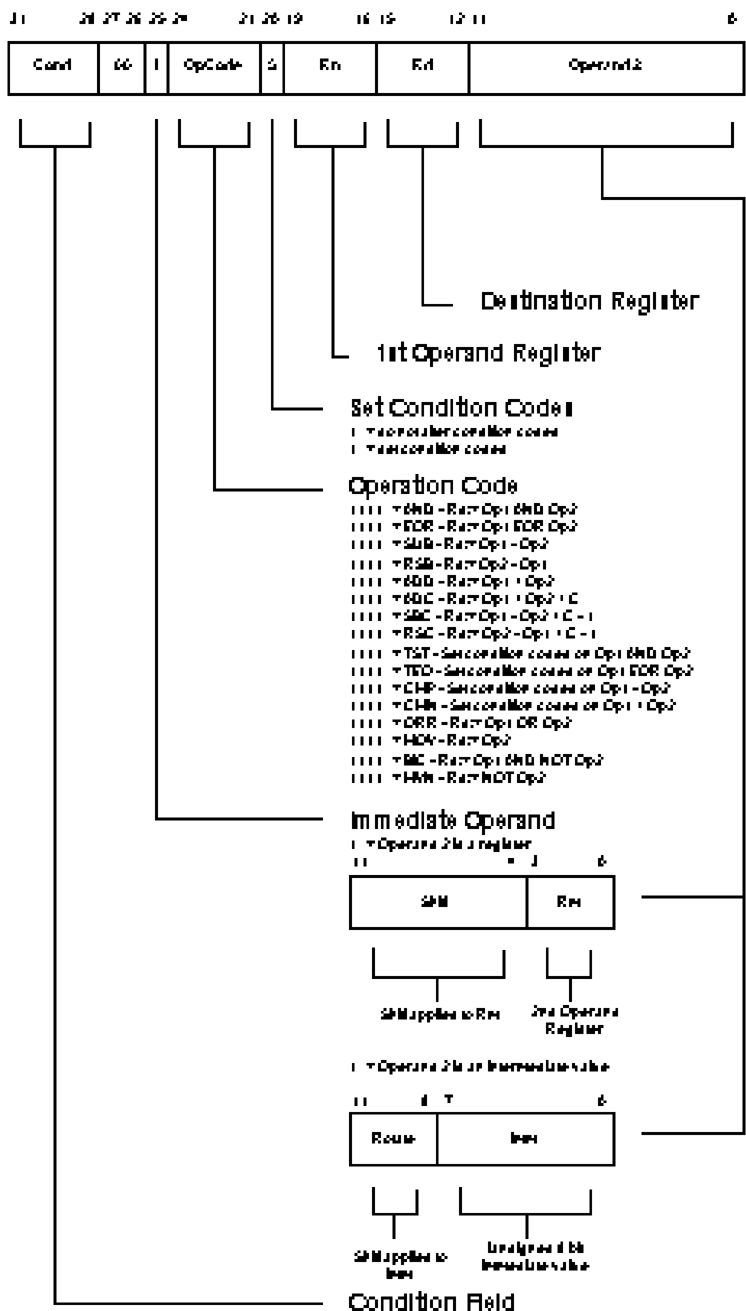


Figure 2-7: Data Processing Instructions

CPSR Flags

The data processing operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result. If the S bit is set (and Rd is not R15, see below) the V flag in the CPSR will be unaffected, the C flag will be set to the carry out from the barrel shifter (or preserved when the shift operation is LSL #0), the Z flag will be set if and only if the result is all zeros, and the N flag will be set to the logical value of bit 31 of the result.

Assembler Mnemonic	OpCode	Action
AND	0000	operand1 AND operand2
EOR	0001	operand1 EOR operand2
SUB	0010	operand1 - operand2
RSB	0011	operand2 - operand1
ADD	0100	operand1 + operand2
ADC	0101	operand1 + operand2 + carry
SBC	0110	operand1 - operand2 + carry - 1
RSC	0111	operand2 - operand1 + carry - 1
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	operand1 OR operand2
MOV	1101	operand2 (operand1 is ignored)
BIC	1110	operand1 AND NOT operand2 (Bit reset)
MVN	1111	NOT operand2 (operand1 is ignored)

Table 2-3: ARM Data Processing Instructions

The arithmetic operations (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) treat each operand as a 32 bit integer (either unsigned or 2's complement signed, the two are equivalent). If the S bit is set (and Rd is not R15) the V flag in the CPSR will be set if an overflow occurs into bit 31 of the result; this may be ignored if the operands were considered unsigned, but warns of a possible error if the operands were 2's complement signed. The C flag will be set to the carry out of bit 31 of the ALU, the Z flag will be set if and only if the result was zero, and the N flag will be set to the value of bit 31 of the result (indicating a negative result if the operands are considered to be 2's complement signed).

Shifts

When the second operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the Shift field in the instruction. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should be shifted may be contained in an immediate field in the instruction, or in the bottom byte of another register (other than R15). The encoding for the different shift types is shown in *Figure 2-8: ARM Shift Operations*.

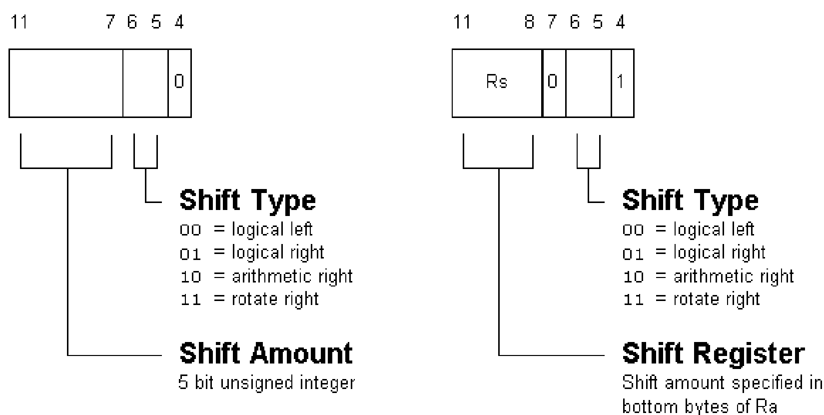


Figure 2-8: ARM Shift Operations

Instruction specified shift amount

When the shift amount is specified in the instruction, it is contained in a 5 bit field which may take any value from 0 to 31. A logical shift left (LSL) takes the contents of Rm and moves each bit by the specified amount to a more significant position. The least significant bits of the result are filled with zeros, and the high bits of Rm which do not map into the result are discarded, except that the least significant discarded bit becomes the shifter carry output which may be latched into the C bit of the CPSR when the ALU operation is in the logical class (see above). For example, the effect of LSL #5 is shown in *Figure 2-9: Logical Shift Left*.

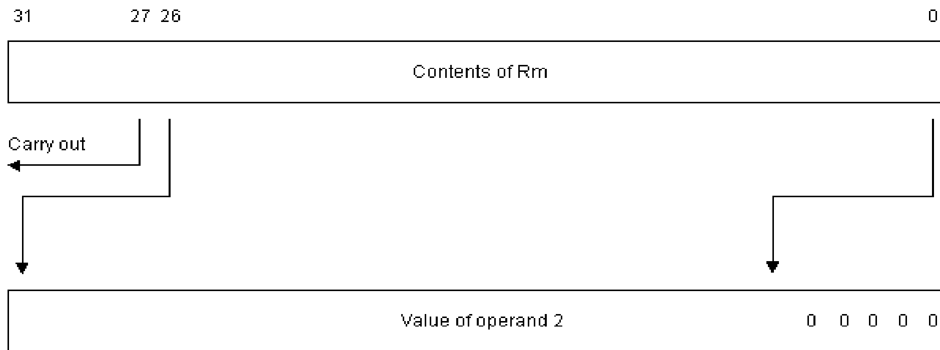


Figure 2-9: Logical Shift Left

Note that LSL #0 is a special case, where the shifter carry out is the old value of the CPSR C flag. The contents of Rm are used directly as the second operand.

A logical shift right (LSR) is similar, but the contents of Rm are moved to less significant positions in the result. LSR #5 has the effect shown in *Figure 2-10: Logical Shift Right*.

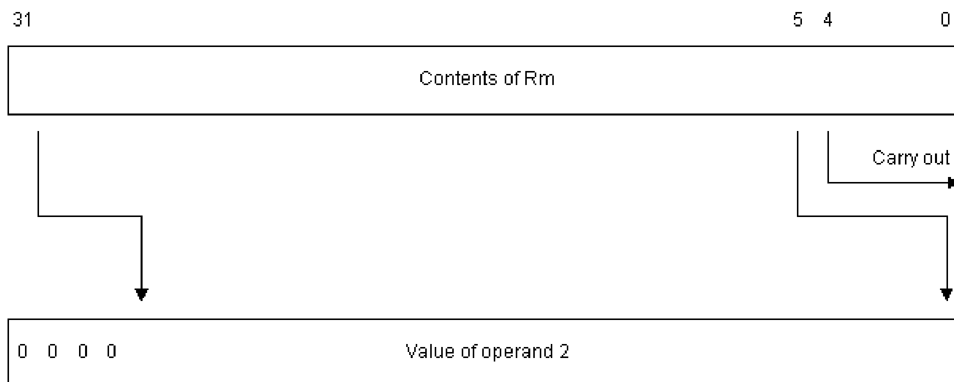


Figure 2-10: Logical Shift Right

The form of the shift field which might be expected to correspond to LSR #0 is used to encode LSR #32, which has a zero result with bit 31 of Rm as the carry output. Logical shift right zero is redundant as it is the same as logical shift left zero, so the assembler will convert LSR #0 (and ASR #0 and ROR #0) into LSL #0, and allow LSR #32 to be specified.

An arithmetic shift right (ASR) is similar to logical shift right, except that the high bits are filled with bit 31 of Rm instead of zeros. This preserves the sign in 2's complement notation. For example, ASR #5 is shown in *Figure 2-11: Arithmetic Shift Right*.

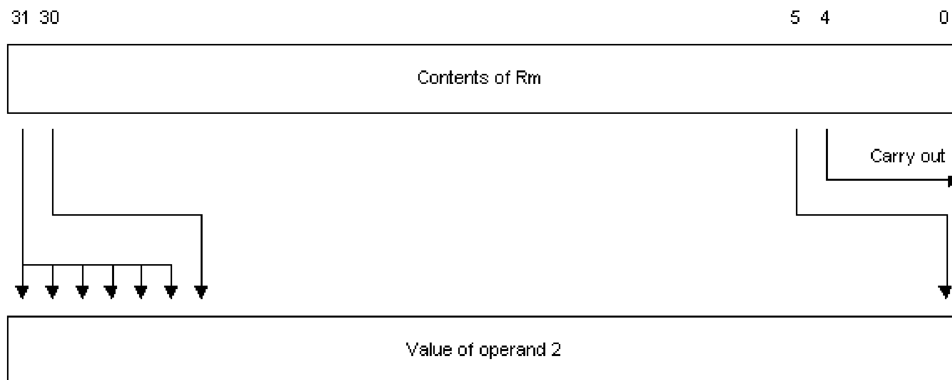


Figure 2-11: Arithmetic Shift Right

The form of the shift field which might be expected to give ASR #0 is used to encode ASR #32. Bit 31 of Rm is again used as the carry output, and each bit of operand 2 is also equal to bit 31 of Rm. The result is therefore all ones or all zeros, according to the value of bit 31 of Rm.

Rotate right (ROR) operations reuse the bits which 'overshoot' in a logical shift right operation by reintroducing them at the high end of the result, in place of the zeros used to fill the high end in logical right operations. For example, ROR #5 is shown in *Figure 2-12: Rotate Right*.

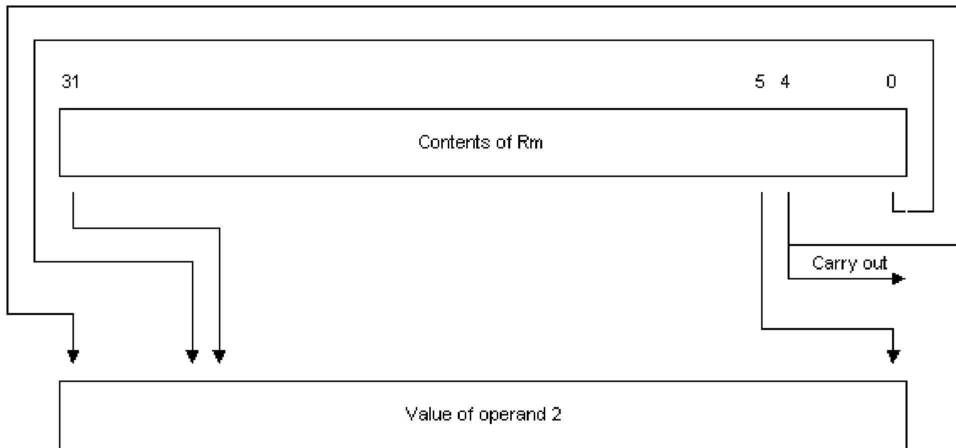


Figure 2-12: Rotate Right

The form of the shift field which might be expected to give ROR #0 is used to encode a special function of the barrel shifter, rotate right extended (RRX). This is a rotate right by one bit position of the 33 bit quantity formed by appending the CPSR C flag to the most significant end of the contents of Rm as shown in *Figure 2-13: Rotate Right Extended*.

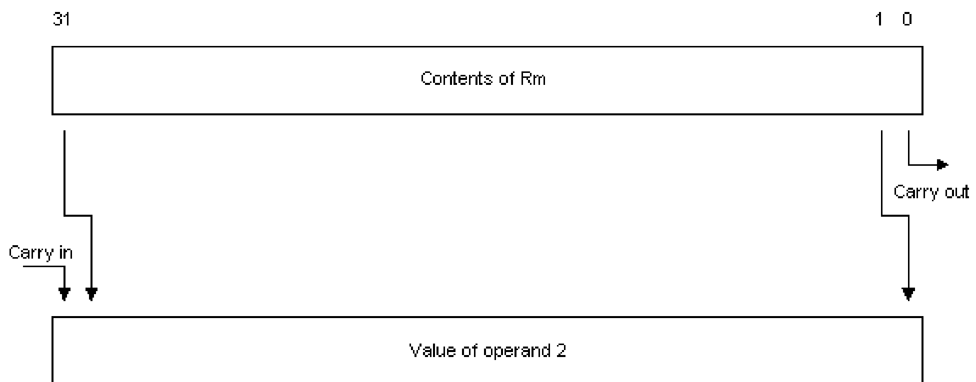


Figure 2-13: Rotate Right Extended

Register Specified Shift Amount

Only the least significant byte of the contents of Rs is used to determine the shift amount. Rs can be any general register other than R15.

If this byte is zero, the unchanged contents of Rm will be used as the second operand, and the old value of the CPSR C flag will be passed on as the shifter carry output.

If the byte has a value between 1 and 31, the shifted result will exactly match that of an instruction specified shift with the same value and shift operation.

If the value in the byte is 32 or more, the result will be a logical extension of the shift described above:

[1] LSL by 32 has result zero, carry out equal to bit 0 of Rm.

[2] LSL by more than 32 has result zero, carry out zero.

[3] LSR by 32 has result zero, carry out equal to bit 31 of Rm.

[4] LSR by more than 32 has result zero, carry out zero.

[5] ASR by 32 or more has result filled with and carry out equal to bit 31 of Rm.

[6] ROR by 32 has result equal to Rm, carry out equal to bit 31 of Rm.

[7] ROR by n where n is greater than 32 will give the same result and carry out as ROR by n-32; therefore repeatedly subtract 32 from n until the amount is in the range 1 to 32 and see above.

Note that the zero in bit 7 of an instruction with a register controlled shift is compulsory; a one in this bit will cause the instruction to be a multiply or undefined instruction.

Immediate Operand Rotates

The immediate operand rotate field is a 4 bit unsigned integer which specifies a shift operation on the 8 bit immediate value. This value is zero extended to 32 bits, and then subject to a rotate right by twice the value in the rotate field. This enables many common constants to be generated, for example all powers of 2.

Writing to R15

When Rd is a register other than R15, the condition code flags in the CPSR may be updated from the ALU flags as described above.

When Rd is R15 and the S flag in the instruction is not set the result of the operation is placed in R15 and the CPSR is unaffected.

When Rd is R15 and the S flag is set the result of the operation is placed in R15 and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of instruction shall not be used in User mode.

Using R15 as an Operand

If R15 (the PC) is used as an operand in a data processing instruction the register is used directly.

The PC value will be the address of the instruction, plus 8 or 12 bytes due to instruction prefetching. If the shift amount is specified in the instruction, the PC will be 8 bytes ahead. If a register is used to specify the shift amount the PC will be 12 bytes ahead.

TEQ, TST, CMP & CMN Opcodes

These instructions do not write the result of their operation but do set flags in the CPSR. An assembler shall always set the S flag for these instructions even if it is not specified in the mnemonic.

The TEQP form of the instruction used in earlier processors shall not be used in the 32 bit modes, the PSR transfer operations should be used instead. If used in these modes, its effect is to move SPSR_<mode> to CPSR if the processor is in a privileged mode and to do nothing if in User mode.

Instruction Cycle Times

Data Processing instructions vary in the number of incremental cycles taken as follows:

[1] Normal Data Processing 1S

[2] Data Processing with register specified shift 1S + 1I

[3] Data Processing with PC written 2S + 1N

[4] Data Processing with register specified shift and PC written2S + 1N + 1I

S, N and I are as defined in *Section 2.5.1, Cycle Types*.

Assembler Syntax

[1] MOV,MVN - single operand instructions
opcode>{cond}{S} Rd,<Op2>

[2] CMP,CMN,TEQ,TST - instructions which do not produce a result.
<opcode>{cond} Rn,<Op2>

[3] AND,EOR,SUB,RSB,ADD,ADC,SBC,RSC,ORR,BIC
<opcode>{cond}{S} Rd,Rn,<Op2>

where <Op2> is **Rm{,<shift>}** or,**<#expression>**

{cond} - two-character condition mnemonic, see *Figure 2-5: Condition Codes*

{S} - set condition codes if S present (implied for CMP, CMN, TEQ, TST).

Rd, Rn and Rm are expressions evaluating to a register number.

If <#expression> is used, the assembler will attempt to generate a shifted immediate 8-bit field to match the expression. If this is impossible, it will give an error.

<shift> is <shiftname> <register> or <shiftname> #expression, or RRX (rotate right one bit with extend).

<shiftname>s are: ASL, LSL, LSR, ASR, ROR. (ASL is a synonym for LSL, they assemble to the same code.

Examples

```
ADDEQ R2,R4,R5  ; if the Z flag is set make R2:=R4+R5

TEQS  R4,#3     ; test R4 for equality with 3
                ; (the S is in fact redundant as the
                ; assembler inserts it automatically)

SUB   R4,R5,R7,LSR R2
                ; logical right shift R7 by the number in
                ; the bottom byte of R2, subtract result
                ; from R5, and put the answer into R4

MOV   PC,R14    ; return from subroutine

MOVS  PC,R14    ; return from exception and restore CPSR
                ; from SPSR_mode
```

2.4.5 PSR Transfer (MRS, MSR)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter.

The MRS and MSR instructions are formed from a subset of the Data Processing operations and are implemented using the TEQ, TST, CMN and CMP instructions without the S flag set. The encoding is shown in *Figure 2-14: PSR Transfer*.

These instructions allow access to the CPSR and SPSR registers. The MRS instruction allows the contents of the CPSR or SPSR_<mode> to be moved to a general register. The MSR instruction allows the contents of a general register to be moved to the CPSR or SPSR_<mode> register.

The MSR instruction also allows an immediate value or register contents to be transferred to the condition code flags (N,Z,C and V) of CPSR or SPSR_<mode> without affecting the control bits. In this case, the top four bits of the specified register contents or 32 bit immediate value are written to the top four bits of the relevant PSR.

Operand Restrictions

In User mode, the control bits of the CPSR are protected from change, so only the condition code flags of the CPSR can be changed. In other (privileged) modes the entire CPSR can be changed.

The SPSR register which is accessed depends on the mode at the time of execution. For example, only SPSR_fiq is accessible when the processor is in FIQ mode.

R15 shall not be specified as the source or destination register.

A further restriction is that no attempt shall be made to access an SPSR in User mode, since no such register exist

Reserved Bits

Only eleven bits of the PSR are defined in ARM7 (N,Z,C,V,I,F & M[4:0]); the remaining bits (= PSR[27:8,5]) are reserved for use in future versions of the processor. To ensure the maximum compatibility between ARM7 programs and future processors, the following rules should be observed:

[1] The reserved bits shall be preserved when changing the value in a PSR.

[2] Programs shall not rely on specific values from the reserved bits when checking the PSR status, since they may read as one or zero in future processors.

A read-modify-write strategy should therefore be used when altering the control bits of any PSR register; this involves transferring the appropriate PSR register to a general register using the MRS instruction, changing only the relevant bits and then transferring the modified value back to the PSR register using the MSR instruction.

e.g. The following sequence performs a mode change:

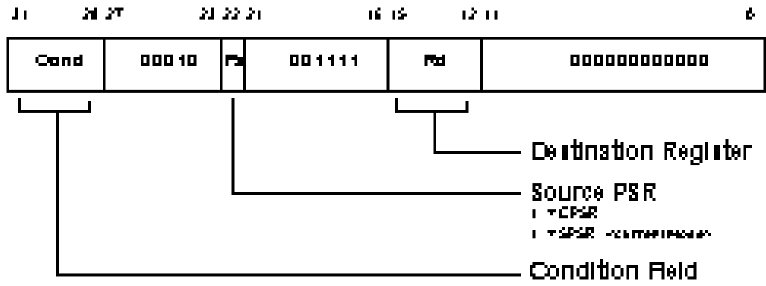
```
MRS  R0,CPSR          ; take a copy of the CPSR
BIC  R0,R0,#0x1F      ; reset the mode bits
ORR  R0,R0,#new_mode; select new mode
MSR  CPSR,R0          ; write back the modified CPSR
```

When the aim is simply to change the condition code flags in a PSR, a value can be written directly to the flag bits without disturbing the control bits. e.g. The following instruction sets the N,Z,C & V flags:

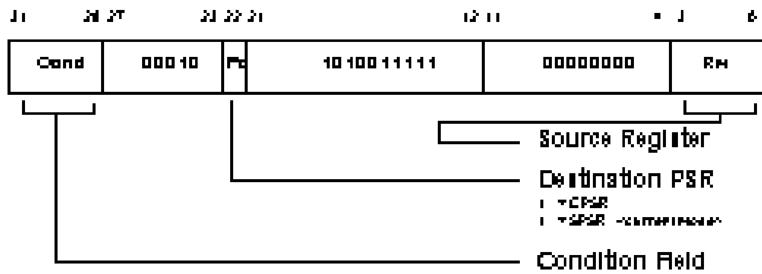
```
MSR  CPSR_flg,#0xF0000000 ; set all the flags regardless of
                          ; their previous state (does not
                          ; affect any control bits)
```

No attempt shall be made to write an 8 bit immediate value into the whole PSR since such an operation cannot preserve the reserved bits.

MSR (transfer PSR contents to a register)



MSR (transfer register contents to a PSR)



MSR (transfer register contents or immediate value to PSR flag bits only)

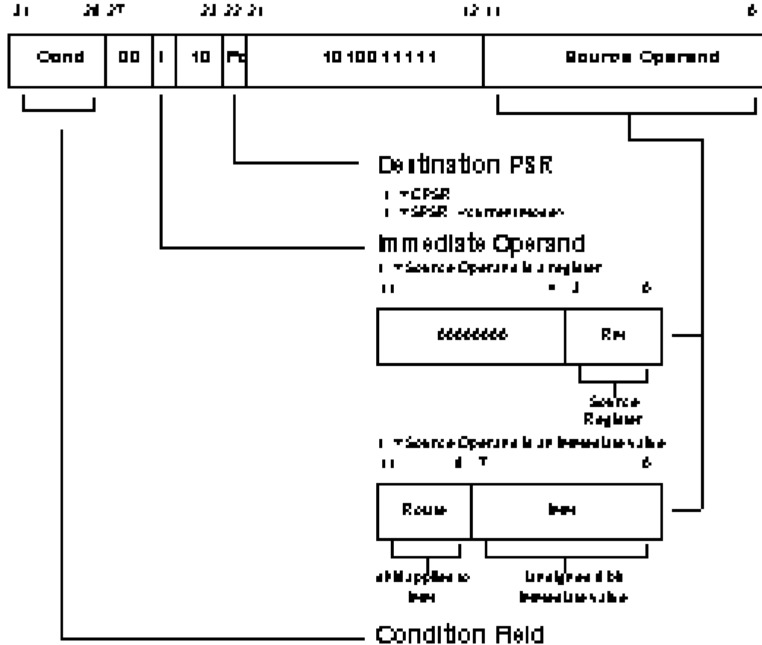


Figure 2-14: PSR Transfer

Instruction Cycle Times

PSR Transfers take 1S incremental cycle, where S as defined in *Section 2.5.1, Cycle Types*.

Assembler Syntax

[1] MRS - transfer PSR contents to a register
MRS{cond} Rd,<psr>

[2] MSR - transfer register contents to PSR
MSR{cond} <psr>,Rm

[3] MSR - transfer register contents to PSR flag bits only
MSR{cond} <psrf>,Rm

The most significant four bits of the register contents are written to the N,Z,C & V flags respectively.

[4] MSR - transfer immediate value to PSR flag bits only
MSR{cond} <psrf>,<#expression>

The expression should symbolise a 32 bit value of which the most significant four bits are written to the N,Z,C & V flags respectively.

{cond} - two-character condition mnemonic, see *Figure 2-5: Condition Codes*

Rd and Rm are expressions evaluating to a register number other than R15

<psr> is CPSR, CPSR_all, SPSR or SPSR_all. (CPSR and CPSR_all are synonyms as are SPSR and SPSR_all)

<psrf> is CPSR_flg or SPSR_flg

Where <#expression> is used, the assembler will attempt to generate a shifted immediate 8-bit field to match the expression. If this is impossible, it will give an error.

Examples

In User mode the instructions behave as follows:

```
MSR  CPSR_all,Rm             ; CPSR[31:28] <- Rm[31:28]
MSR  CPSR_flg,Rm             ; CPSR[31:28] <- Rm[31:28]
MSR  CPSR_flg,#0xA0000000    ; CPSR[31:28] <- 0xA
                             ;   (i.e. set N,C; reset Z,V)

MRS  Rd,CPSR                 ; Rd[31:0] <- CPSR[31:0]
```

In privileged modes the instructions behave as follows:

```
MSR  CPSR_all,Rm              ; CPSR[31:0]  <- Rm[31:0]
MSR  CPSR_flg,Rm              ; CPSR[31:28] <- Rm[31:28]

MSR  CPSR_flg,#0x50000000     ; CPSR[31:28] <- 0x5
                              ;   (i.e. set Z,V; reset N,C)

MRS  Rd,CPSR                  ; Rd[31:0] <- CPSR[31:0]

MSR  SPSR_all,Rm              ; SPSR_<mode>[31:0]  <- Rm[31:0]
MSR  SPSR_flg,Rm              ;       SPSR_<mode>[31:28]       <-
Rm[31:28]

MSR  SPSR_flg,#0xC0000000     ; SPSR_<mode>[31:28] <- 0xC
                              ;   (i.e. set N,Z; reset C,V)

MRS  Rd,SPSR                  ; Rd[31:0] <- SPSR_<mode>[31:0]
```

2.4.6 Multiply and Multiply-Accumulate (MUL, MLA)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 2-15: Multiply Instructions*. The multiply and multiply-accumulate instructions use a 2 bit Booth's algorithm to perform integer multiplication.

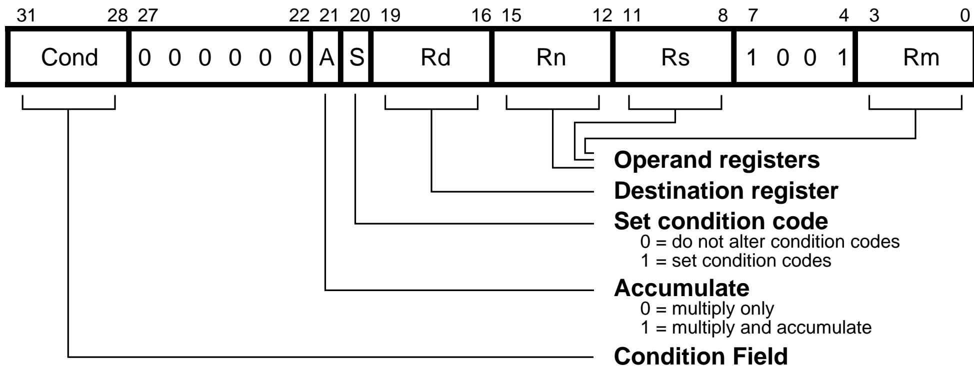


Figure 2-15: Multiply Instructions

The multiply form of the instruction gives Rd:=Rm*Rs. Rn is ignored, and should be set to zero for compatibility with possible future upgrades to the instruction set.

The multiply-accumulate form gives Rd:=Rm*Rs+Rn, which can save an explicit ADD instruction in some circumstances.

The results of a signed multiply and of an unsigned multiply of 32 bit operands differ only in the upper 32 bits - the low 32 bits of the signed and unsigned results are identical. As these instructions only produce the low 32 bits of a multiply, they can be used for both signed and unsigned multiplies.

For example consider the multiplication of the operands:

Operand A	Operand B	Result
0xFFFFFFF6	0x00000014	0xFFFFFF38

If the operands are interpreted as signed, operand A has the value -10, operand B has the value 20, and the result is -200 which is correctly represented as 0xFFFFFF38

If the operands are interpreted as unsigned, operand A has the value 4294967286, operand B has the value 20 and the result is 85899345720, which is represented as 0x13FFFFFF38, so the least significant 32 bits are 0xFFFFFF38.

Operand Restrictions

Due to the way multiplication was implemented in other ARM processors, certain combinations of operand registers should be avoided. The ARM7's advanced multiplier can handle all operand combinations but by observing these restrictions code written for the ARM7 will be more compatible with other ARM processors. (The assembler will issue a warning if these restrictions are overlooked.)

The destination register Rd shall not be the same as the operand register Rm. R15 shall not be used as an operand or as the destination register.

All other register combinations will give correct results, and Rd, Rn and Rs may use the same register when required.

CPSR Flags

Setting the CPSR flags is optional, and is controlled by the S bit in the instruction. The N (Negative) and Z (Zero) flags are set correctly on the result (N is made equal to bit 31 of the result, and Z is set if and only if the result is zero). The C (Carry) flag is set to a meaningless value and the V (oVerflow) flag is unaffected.

Instruction Cycle Times

The Multiply instructions take 1S + mI cycles to execute, where S and I are as defined in *Section 2.5.1, Cycle Types*.

m is the number of cycles required by the multiply algorithm, which is determined by the contents of Rs. Multiplication by any number between *2^(2m-3)* and *2^(2m-1)-1* takes 1S+mI *m* cycles for *1<m>16*. Multiplication by 0 or 1 takes 1S+1I cycles, and multiplication by any number greater than or equal to 2^(29) takes 1S+16I cycles. The maximum time for any multiply is thus 1S+16I cycles.

Assembler Syntax

MUL{cond}{S} Rd,Rm,Rs

MLA{cond}{S} Rd,Rm,Rs,Rn

{cond} - two-character condition mnemonic, see *Figure 2-5: Condition Codes*

{S} - set condition codes if S present

Rd, Rm, Rs and Rn are expressions evaluating to a register number other than R15.

Examples

```
MUL    R1,R2,R3          ; R1 := R2 * R3

MLAEQS R1,R2,R3,R4       ; conditionally R1 := R2 * R3 + R4,
                         ; setting condition codes
```

2.4.7 Single Data Transfer (LDR, STR)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 2-16: Single Data Transfer Instructions*.

The single data transfer instructions are used to load or store single bytes or words of data. The memory address used in the transfer is calculated by adding an offset to or subtracting an offset from a base register.

The result of this calculation may be written back into the base register if `auto-indexing' is required.

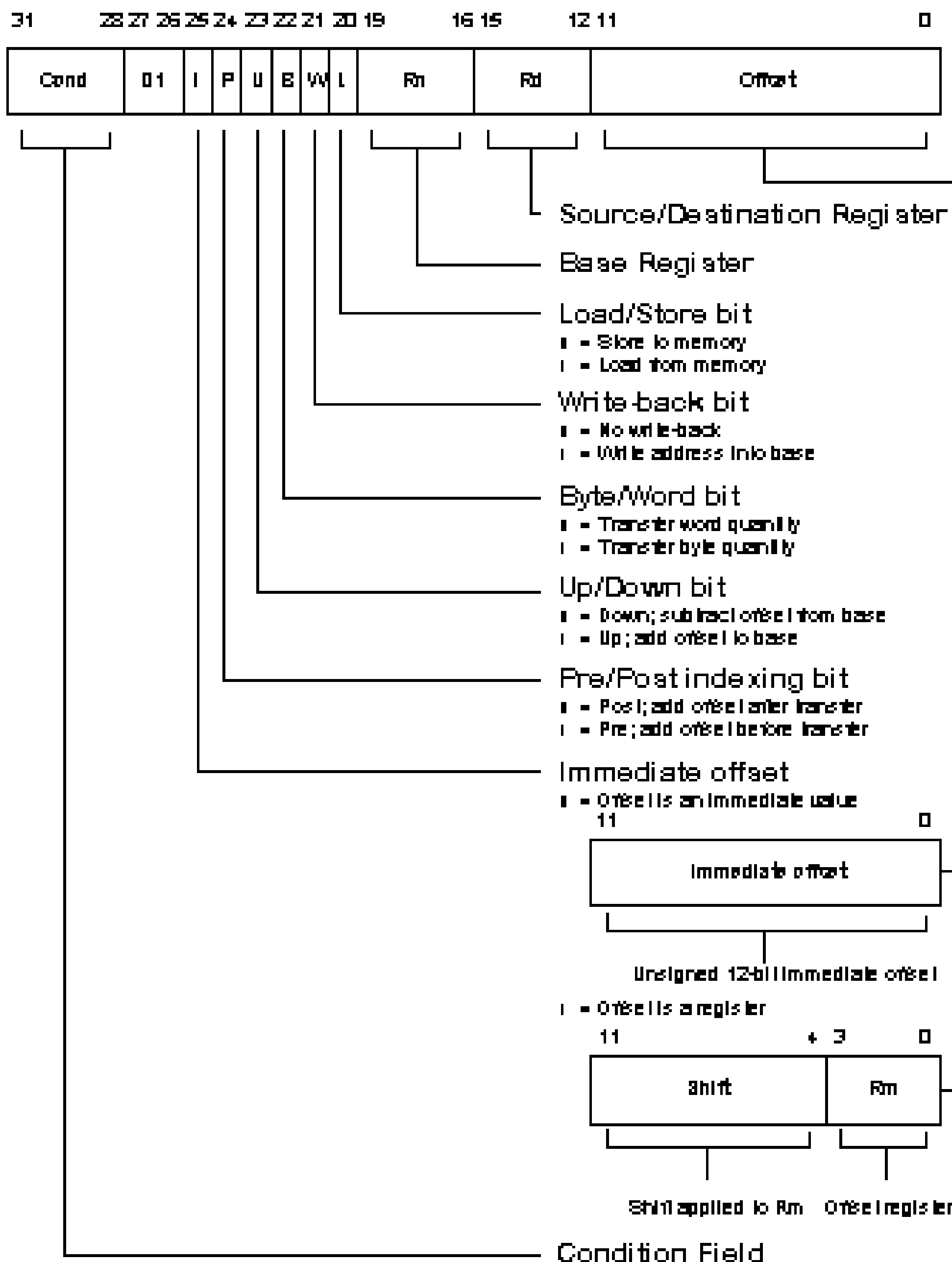


Figure 2-16: Single Data Transfer Instructions

Offsets and Auto-Indexing

The offset from the base may be either a 12 bit unsigned binary immediate value in the instruction, or a second register (possibly shifted in some way). The offset may be added to (U=1) or subtracted from (U=0) the base register Rn. The offset modification may be performed either before (pre-indexed, P=1) or after (post-indexed, P=0) the base is used as the transfer address.

The W bit gives optional auto increment and decrement addressing modes. The modified base value may be written back into the base (W=1), or the old base value may be kept (W=0). In the case of post-indexed addressing, the write back bit is redundant and is always set to zero, since the old base value can be retained by setting the offset to zero. Therefore post-indexed data transfers always write back the modified base. The only use of the W bit in a post-indexed data transfer is in privileged mode code, where setting the W bit forces non-privileged mode for the transfer, allowing the operating system to generate a user address in a system where the memory management hardware makes suitable use of this hardware.

Shifted Register Offset

The 8 shift control bits are described in the data processing instructions section. However, the register specified shift amounts are not available in this instruction class. See *Figure 2-8: ARM Shift Operations*.

Bytes and Words

This instruction class may be used to transfer a byte (B=1) or a word (B=0) between an ARM7 register and memory.

The action of LDR(B) and STR(B) instructions is influenced by the **BIGEND** control signal. The two possible configurations are described below.

Little Endian Configuration

A byte load (LDRB) expects the data on data bus inputs 7 through 0 if the supplied address is on a word boundary, on data bus inputs 15 through 8 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register, and the remaining bits of the register are filled with zeros. This is shown in *Figure 7-9: Little Endian addresses of bytes within words*.

A byte store (STRB) repeats the bottom 8 bits of the source register four times across data bus outputs 31 through 0. The external memory system should activate the appropriate byte subsystem to store the data.

A word load (LDR) will normally use a word aligned address. However, an address offset from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 0 to 7. This means that half-words accessed at offsets 0 and 2 from the word boundary will be correctly loaded into bits 0 through 15 of the register. Two shift operations are then required to reset or to sign extend the upper 16 bits. This is illustrated in *Figure 2-17: Little Endian Offset Addressing*.

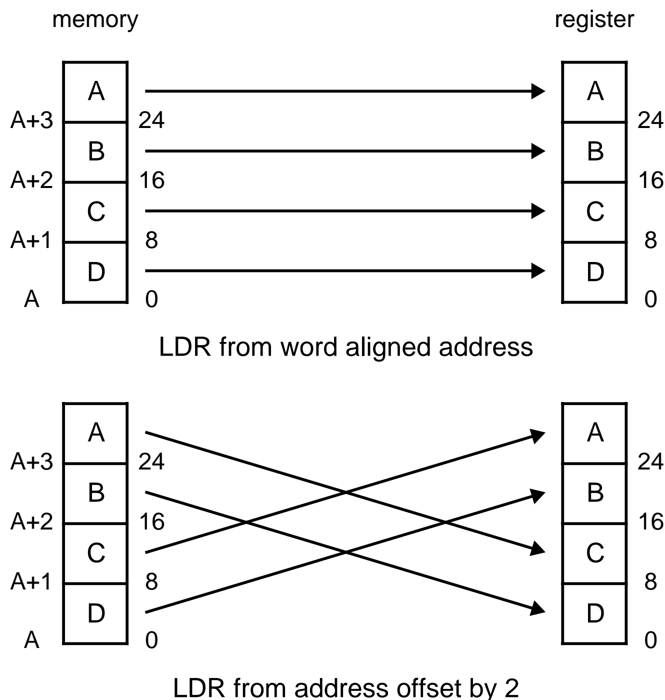


Figure 2-17: Little Endian Offset Addressing

A word store (STR) should generate a word aligned address. The word presented to the data bus is not affected if the address is not word aligned. That is, bit 31 of the register being stored always appears on data bus output 31.

Big Endian Configuration

A byte load (LDRB) expects the data on data bus inputs 31 through 24 if the supplied address is on a word boundary, on data bus inputs 23 through 16 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register and the remaining bits of the register are filled with zeros.

A byte store (STRB) repeats the bottom 8 bits of the source register four times across data bus outputs 31 through 0. The external memory system should activate the appropriate byte subsystem to store the data.

A word load (LDR) should generate a word aligned address. An address offset of 0 or 2 from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 31 through 24. This means that half-words accessed at these offsets will be correctly loaded into bits 16 through 31 of the register. A shift operation is then required to move (and optionally sign extend) the data into the bottom 16 bits. An address offset of 1 or 3 from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 15 through 8.

A word store (STR) should generate a word aligned address. The word presented to the data bus is not affected if the address is not word aligned. That is, bit 31 of the register being stored always appears on data bus output 31.

Use of R15

Write-back shall not be specified if R15 is specified as the base register (Rn). When using R15 as the base register note it contains an address 8 bytes on from the address of the current instruction.

R15 shall not be specified as the register offset (Rm). When R15 is the source register (Rd) of a register store (STR) instruction, the stored value will be address of the instruction plus 12.

Restriction on the use of Base Register

When configured for late aborts, the following example code is difficult to unwind as the base register, Rn, gets updated before the abort handler starts. Sometimes it may be impossible to calculate the initial value.

For example:

```
LDR   R0,[R1],R1
<LDR|STR> Rd,[Rn],{+/-}Rn{,<shift>}
```

Therefore a post-indexed LDR|STR where Rm is the same register as Rn should not be used.

Data Aborts

A transfer to or from a legal address may cause problems for a memory management system. For instance, in a system which uses virtual memory the required data may be absent from main memory. The memory manager can signal a problem by taking the processor ABORT input HIGH whereupon the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

Instruction Cycle Times

Normal LDR instructions take 1S + 1N + 1I and LDR PC take 2S + 2N +1I incremental cycles, where S,N and I are as defined in *Section 2.5.1, Cycle Types*.

STR instructions take 2N incremental cycles to execute.

Assembler Syntax

<LDR|STR>{cond}{B}{T} Rd,<Address>

LDR - load from memory into a register

STR - store from a register into memory

{cond} - two-character condition mnemonic, see *Figure 2-5: Condition Codes*

{B} - if B is present then byte transfer, otherwise word transfer

{T} - if T is present the W bit will be set in a post-indexed instruction, forcing non-privileged mode for the transfer cycle. T is not allowed when a pre-indexed addressing mode is specified or implied.

Rd is an expression evaluating to a valid register number.

<Address> can be:

[1] An expression which generates an address:
<expression>

The assembler will attempt to generate an instruction using the PC as a base and a corrected immediate offset to address the location given by evaluating the expression. This will be a PC relative, pre-indexed address. If the address is out of range, an error will be generated.

[2] A pre-indexed addressing specification:

[Rn] offset of zero

[Rn,<#expression>]{!} offset of <expression> bytes

[Rn,{+/-}Rm{,<shift>}]{!} offset of +/- contents of index register, shifted by <shift>

[3] A post-indexed addressing specification:

[Rn],<#expression> offset of <expression> bytes

[Rn],{+/-}Rm{,<shift>} offset of +/- contents of index register, shifted as by <shift>.

Rn and Rm are expressions evaluating to a register number. If Rn is R15 then the assembler will subtract 8 from the offset value to allow for ARM7 pipelining. In this case base write-back shall not be specified.

<shift> is a general shift operation (see section on data processing instructions) but note that the shift amount may not be specified by a register.

{!} writes back the base register (set the W bit) if ! is present.

Examples

```
        STR   R1,[R2,R4]! ; store R1 at R2+R4 (both of which are
                          ; registers) and write back address to R2

        STR   R1,[R2],R4  ; store R1 at R2 and write back
                          ; R2+R4 to R2

        LDR   R1,[R2,#16] ; load R1 from contents of R2+16
                          ;  Don't write back

        LDR   R1,[R2,R3,LSL#2]; load R1 from contents of R2+R3*4

        LDREQBR1,[R6,#5]; conditionally load byte at R6+5 into
                          ;  R1 bits 0 to 7, filling bits 8 to 31
                          ;  with zeros

        STR   R1,PLACE    ; generate PC relative offset to address
              •           ; PLACE
              •
PLACE...
```

2.4.8 Block Data Transfer (LDM, STM)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 2-18: Block Data Transfer Instructions*.

Block data transfer instructions are used to load (LDM) or store (STM) any subset of the currently visible registers. They support all possible stacking modes, maintaining full or empty stacks which can grow up or down memory, and are very efficient instructions for saving or restoring context, or for moving large blocks of data around main memory.

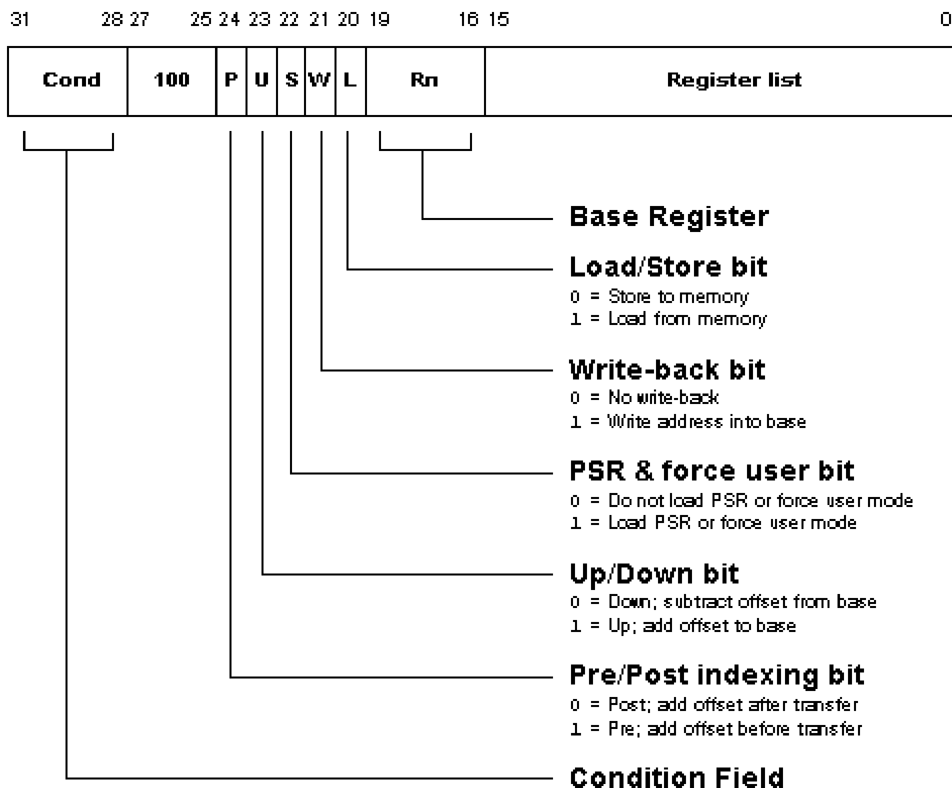


Figure 2-18: Block Data Transfer Instructions

The Register List

The instruction can cause the transfer of any registers in the current bank (and non-user mode programs can also transfer to and from the user bank, see below). The register list is a 16 bit field in the instruction, with each bit corresponding to a register. A 1 in bit 0 of the register field will cause R0 to be transferred, a 0 will cause it not to be transferred; similarly bit 1 controls the transfer of R1, and so on.

Any subset of the registers, or all the registers, may be specified. The only restriction is that the register list should not be empty.

Whenever R15 is stored to memory the stored value is the address of the STM instruction plus 12.

Addressing Modes

The transfer addresses are determined by the contents of the base register (Rn), the pre/post bit (P) and the up/down bit (U). The registers are transferred in the order lowest to highest, so R15 (if in the list) will always be transferred last. The lowest register also gets transferred to/from the lowest memory address. By way of illustration, consider the transfer of R1, R5 and R7 in the case where Rn=0x1000 and write back of the modified base is required (W=1). *Figure 2-19: Post-Increment Addressing*, *Figure 2-20: Pre-Increment Addressing*, *Figure 2-21: Post-Decrement Addressing* and *Figure 2-22: Pre-Decrement Addressing* show the sequence of register transfers, the addresses used, and the value of Rn after the instruction has completed.

In all cases, had write back of the modified base not been required (W=0), Rn would have retained its initial value of 0x1000 unless it was also in the transfer list of a load multiple register instruction, when it would have been over-written with the loaded value.

Address Alignment

The address should normally be a word aligned quantity and non-word aligned addresses do not affect the instruction. However, the bottom 2 bits of the address will appear on **A[1:0]** and might be interpreted by the memory system

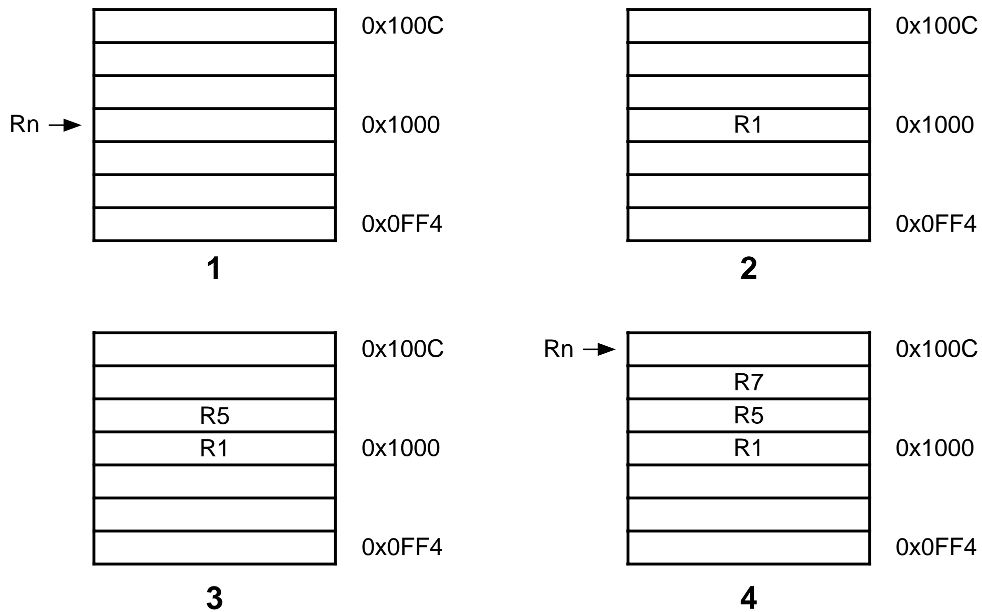


Figure 2-19: Post-Increment Addressing

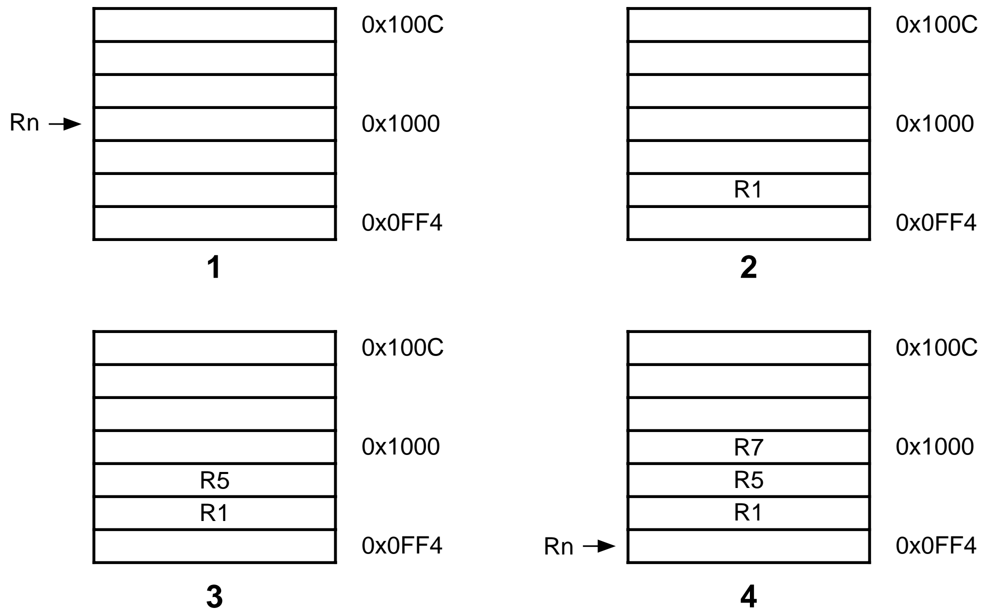


Figure 2-20: Pre-Increment Addressing

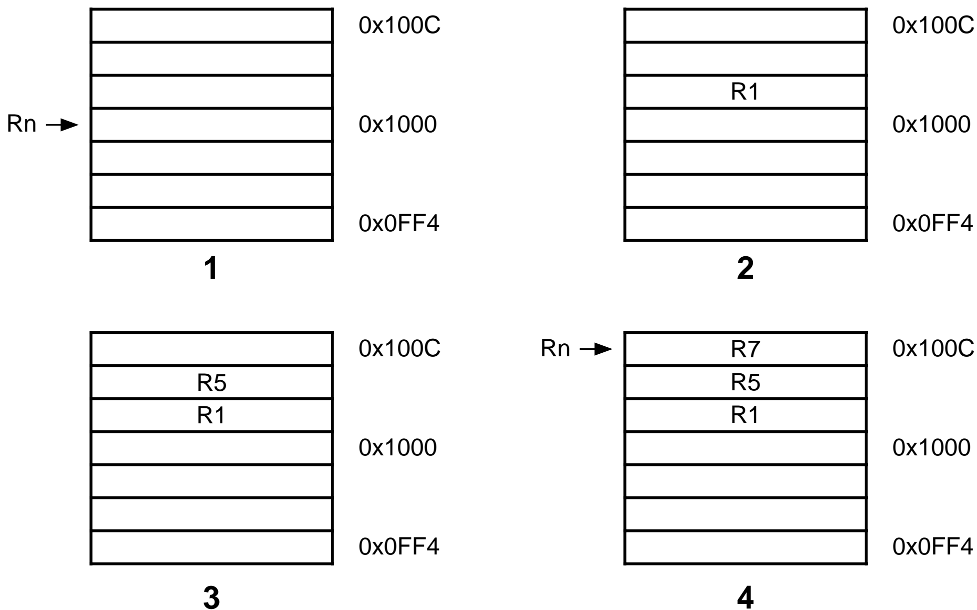


Figure 2-21: Post-Decrement Addressing

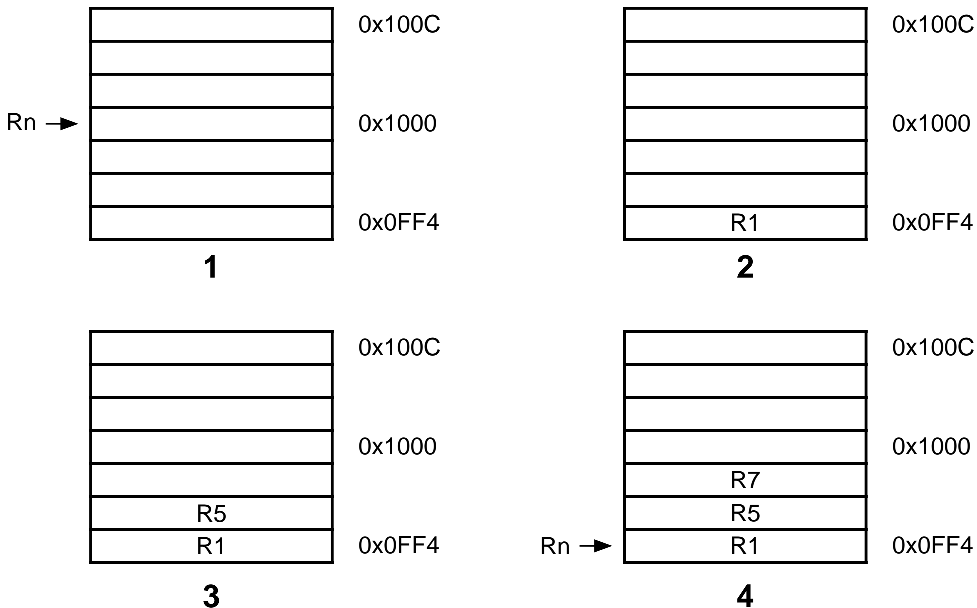


Figure 2-22: Pre-Decrement Addressing

Use of the S Bit

When the S bit is set in a LDM/STM instruction its meaning depends on whether or not R15 is in the transfer list and on the type of instruction. The S bit should only be set if the instruction is to execute in a privileged mode.

LDM with R15 in transfer list and S Bit set (Mode changes)

If the instruction is a LDM then SPSR_<mode> is transferred to CPSR at the same time as R15 is loaded.

STM with R15 in transfer list and S Bit set (User bank transfer)

The registers transferred are taken from the User bank rather than the bank corresponding to the current mode. This is useful for saving the user state on process switches. Base write-back shall not be used when this mechanism is employed.

R15 not in list and S bit set (User bank transfer)

For both LDM and STM instructions, the User bank registers are transferred rather than the register bank corresponding to the current mode. This is useful for saving the user state on process switches. Base write-back shall not be used when this mechanism is employed.

When the instruction is LDM, care must be taken not to read from a banked register during the following cycle (inserting a NOP after the LDM will ensure safety).

Use of R15 as the base

R15 shall not be used as the base register in any LDM or STM instruction.

Inclusion of the base in the register list

When write-back is specified, the base is written back at the end of the second cycle of the instruction. During a STM, the first register is written out at the start of the second cycle. A STM which includes storing the base, with the base as the first register to be stored, will therefore store the unchanged value, whereas with the base second or later in the transfer order, will store the modified value. A LDM will always over-write the updated base if the base is in the list.

Data Aborts

Some legal addresses may be unacceptable to a memory management system, and the memory manager can indicate a problem with an address by taking the **ABORT** signal HIGH. This can happen on any transfer during a multiple register load or store, and must be recoverable if ARM7 is to be used in a virtual memory system.

Aborts During STM instructions

If the abort occurs during a store multiple instruction, ARM7 takes little action until the instruction completes, whereupon it enters the data abort trap. The memory manager is responsible for preventing erroneous writes to the memory. The only change to the internal state of the processor will be the modification of the base register if write-back was specified, and this must be reversed by software (and the cause of the abort resolved) before the instruction may be retried.

Aborts During LDM instructions

When ARM7 detects a data abort during a load multiple instruction, it modifies the operation of the instruction to ensure that recovery is possible.

[1] Overwriting of registers stops when the abort happens. The aborting load will not take place but earlier ones may have over-written registers. The PC is always the last register to be written and so will always be preserved.

[2] The base register is restored, to its modified value if write-back was requested. This ensures recoverability in the case where the base register is also in the transfer list, and may have been over-written before the abort occurred.

The data abort trap is taken when the load multiple has completed, and the system software must undo any base modification (and resolve the cause of the abort) before restarting the instruction.

Instruction Cycle Times

Normal LDM instructions take nS + 1N + 1I and LDM PC takes (n+1)S + 2N + 1I incremental cycles, where S,N and I are as defined in *Section 2.5.1, Cycle Types*.

STM instructions take (n-1)S + 2N incremental cycles to execute, where *n* is the number of words transferred.

Assembler Syntax

<LDM|STM>{cond}<FD|ED|FA|EA|IA|IB|DA|DB> Rn{!},<Rlist>{^}

{cond} - two character condition mnemonic, see *Figure 2-5: Condition Codes*

Rn is an expression evaluating to a valid register number

<Rlist> is a list of registers and register ranges enclosed in {} (eg {R0,R2-R7,R10}).

{!} if present requests write-back (W=1), otherwise W=0

{^} if present sets S bit to load the CPSR along with the PC, or forces transfer of user bank when in privileged mode

Addressing Mode Names

There are different assembler mnemonics for each of the addressing modes, depending on whether the instruction is being used to support stacks or for other purposes. The equivalences between the names and the values of the bits in the instruction are shown in the following table:

name	stack	other	L bit	P bit	U bit
pre-increment load	LDMED	LDMIB	1	1	1
post-increment load	LDMFD	LDMIA	1	0	1
pre-decrement load	LDMEA	LDMDB	1	1	0
post-decrement load	LDMFA	LDMDA	1	0	0
pre-increment store	STMFA	STMIB	0	1	1
post-increment store	STMEA	STMIA	0	0	1
pre-decrement store	STMFD	STMDB	0	1	0
post-decrement store	STMED	STMDA	0	0	0

Table 2-4: Addressing Mode Names

FD, ED, FA, EA define pre/post indexing and the up/down bit by reference to the form of stack required. The F and E refer to a “full” or “empty” stack, i.e. whether a pre-index has to be done (full) before storing to the stack. The A and D refer to whether the stack is ascending or descending. If ascending, a STM will go up and LDM down, if descending, vice-versa.

IA, IB, DA, DB allow control when LDM/STM are not being used for stacks and simply mean Increment After, Increment Before, Decrement After, Decrement Before.

Examples

```
LDMFDSP!,{R0,R1,R2}   ; unstack 3 registers

STMIAR0,{R0-R15}      ; save all registers

LDMFDSP!,{R15}        ; R15 <- (SP),CPSR unchanged
LDMFDSP!,{R15}^       ;R15 <- (SP),CPSR <- SPSR_mode
                      ; (allowed only in privileged modes)
STMFDR13,{R0-R14}^    ; Save user mode regs on stack (allowed
                      ;  only in privileged modes)
```

These instructions may be used to save state on subroutine entry, and restore it efficiently on return to the calling routine:

```
STMEDSP!,{R0-R3,R14}; save R0 to R3 to use as workspace
                    ; and R14 for returning
BL     somewhere    ; this nested call will over-write R14

LDMEDSP!,{R0-R3,R15}; restore workspace and return
```

2.4.9 Single Data Swap (SWP)

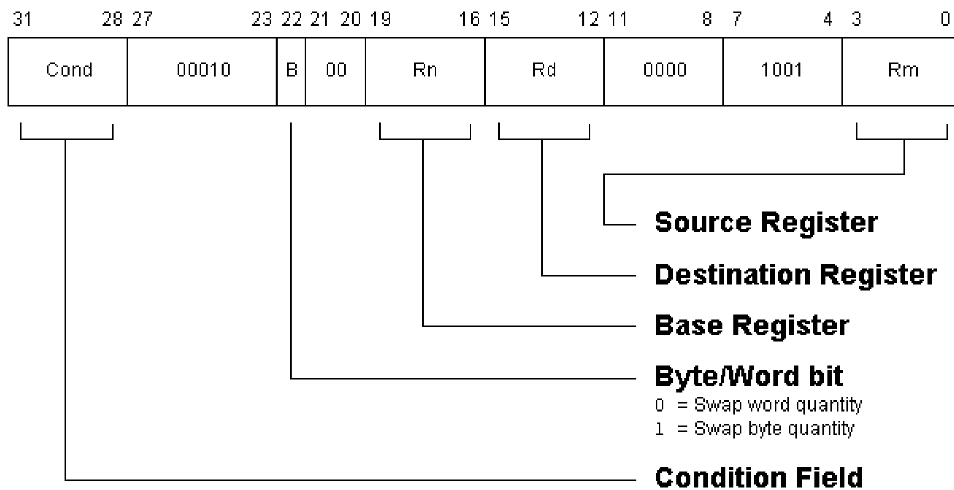


Figure 2-23: Swap Instruction

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 2-23: Swap Instruction*.

The data swap instruction is used to swap a byte or word quantity between a register and external memory. This instruction is implemented as a memory read followed by a memory write which are “locked” together (the processor cannot be interrupted until both operations have completed, and the memory manager is warned to treat them as inseparable). This class of instruction is particularly useful for implementing software semaphores.

The swap address is determined by the contents of the base register (Rn). The processor first reads the contents of the swap address. Then it writes the contents of the source register (Rm) to the swap address, and stores the old memory contents in the destination register (Rd). The same register may be specified as both the source and destination.

The **LOCK** output goes HIGH for the duration of the read and write operations to signal to the external memory manager that they are locked together, and should be allowed to complete without interruption. This is important in multi-processor systems where the swap instruction is the only indivisible instruction which may be used to implement semaphores; control of the memory must not be removed from a processor while it is performing a locked operation.

Bytes and words

This instruction class may be used to swap a byte (B=1) or a word (B=0) between an ARM7 register and memory. The SWP instruction is implemented as a LDR followed by a STR and the action of these is as described in the section on single data transfers. In particular, the description of Big and Little Endian configuration applies to the SWP instruction.

Use of R15

R15 shall not be used as an operand (Rd, Rn or Rs) in a SWP instruction.

Data Aborts

If the address used for the swap is unacceptable to a memory management system, the internal MMU or external memory manager can flag the problem by driving ABORT HIGH. This can happen on either the read or the write cycle (or both), and in either case, the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

Instruction Cycle Times

Swap instructions take 1S + 2N +1I incremental cycles to execute, where S,N and I are as defined in *Section 2.5.1, Cycle Types*.

Assembler Syntax

<SWP>{cond}{B} Rd,Rm,[Rn]

{cond} - two-character condition mnemonic, see *Figure 2-5: Condition Codes*

{B} - if B is present then byte transfer, otherwise word transfer

Rd,Rm,Rn are expressions evaluating to valid register numbers

Examples

```
SWP  R0,R1,[R2]  ; load R0 with the word addressed by R2,
                 ; and store R1 at R2

SWPBR2,R3,[R4]   ; load R2 with the byte addressed by R4,
                 ; and store bits 0 to 7 of R3 at R4

SWPEQR0,R0,[R1]; conditionally swap the contents of R1
                 ; with R0
```

2.4.10 Software Interrupt (SWI)

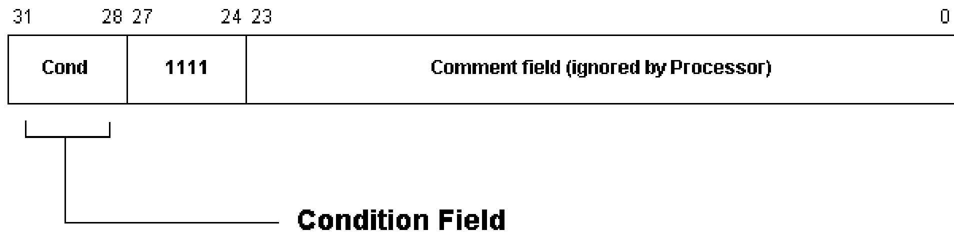


Figure 2-24: Software Interrupt Instruction

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 2-24: Software Interrupt Instruction*.

The software interrupt instruction is used to enter Supervisor mode in a controlled manner. The instruction causes the software interrupt trap to be taken, which effects the mode change. The PC is then forced to a fixed value (0x08) and the CPSR is saved in SPSR_svc. If the SWI vector address is suitably protected (by external memory management hardware) from modification by the user, a fully protected operating system may be constructed.

Return from the Supervisor

The PC is saved in R14_svc upon entering the software interrupt trap, with the PC adjusted to point to the word after the SWI instruction. MOVS PC,R14_svc will return to the calling program and restore the CPSR.

Note that the link mechanism is not re-entrant, so if the supervisor code wishes to use software interrupts within itself it must first save a copy of the return address and SPSR.

Comment Field

The bottom 24 bits of the instruction are ignored by the processor, and may be used to communicate information to the supervisor code. For instance, the supervisor may look at this field and use it to index into an array of entry points for routines which perform the various supervisor functions.

Instruction Cycle Times

Software interrupt instructions take 2S + 1N incremental cycles to execute, where S and N are as defined in *Section 2.5.1, Cycle Types*.

Assembler Syntax

SWI{cond} <expression>

{cond} - two character condition mnemonic, see *Figure 2-5: Condition Codes*

<expression> is evaluated and placed in the comment field (which is ignored by ARM7).

Examples

```
        SWI   ReadC          ; get next character from read stream
        SWI   WriteI+"k"     ; output a "k" to the write stream
        SWINE0               ; conditionally call supervisor
                             ; with 0 in comment field
```

The above examples assume that suitable supervisor code exists, for instance:

```
0x08  B     Supervisor       ; SWI entry point

EntryTable                   ; addresses of supervisor routines
      DCD   ZeroRtn
      DCD   ReadCRtn
      DCD   WriteIRtn
      . . .
Zero  EQU  0
ReadCEQU  256
```

```
WriteIEQU  512

Supervisor

; SWI has routine required in bits 8-23 and data (if any) in bits 0-7.
; Assumes R13_svc points to a suitable stack

        STMFDR13,{R0-R2,R14}; save work registers and return address
        LDR   R0,[R14,#-4]        ; get SWI instruction
        BIC   R0,R0,#0xFF000000; reset top 8 bits
        MOV   R1,R0,LSR#8         ; get routine offset
        ADR   R2,EntryTable       ; get start address of entry table
        LDR   R15,[R2,R1,LSL#2]; branch to appropriate routine

WriteIRtn                         ; enter with character in R0 bits 0-7
        . . . . . .
        LDMFDR13,{R0-R2,R15}^; restore workspace and return
                                  ; restoring processor mode and flags
```

2.4.11 Butterfly Coprocessor Support

The following describes the generic ARM7 instruction set capability for coprocessing and is included for completeness.

On Butterfly however, only coprocessor number 3 is supported and is used for on-chip power control (see *Section 5.3.4, Low Power Sleep Mode*).

All other coprocessor instruction types will result in an undefined instruction exception being taken.

2.4.12 Coprocessor Data Operations (CDP)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 2-25: Coprocessor Data Operation Instruction*.

This class of instruction is used to tell a coprocessor to perform some internal operation. No result is communicated back to ARM7, and it will not wait for the operation to complete. The coprocessor could contain a queue of such instructions awaiting execution, and their execution can overlap other ARM72S + 1N incremental cycles, where S and N are as defined in *Section 2.5.1, Cycle Types*. activity allowing the coprocessor and ARM7 to perform independent tasks in parallel.

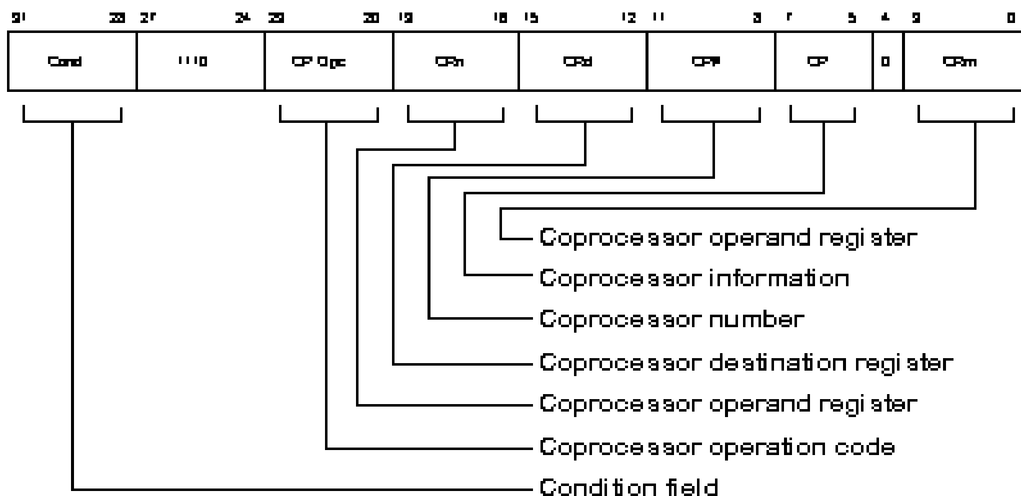


Figure 2-25: Coprocessor Data Operation Instruction

The Coprocessor Fields

Only bit 4 and bits 24 to 31 are significant to ARM7; the remaining bits are used by coprocessors. The above field names are used by convention, and particular coprocessors may redefine the use of all fields except CP# as appropriate. The CP# field is used to contain an identifying number (in the range 0 to 15) for each coprocessor, and a coprocessor will ignore any instruction which does not contain its number in the CP# field.

The conventional interpretation of the instruction is that the coprocessor should perform an operation specified in the CP Opc field (and possibly in the CP field) on the contents of CRn and CRm, and place the result in CRd.

Instruction Cycle Times

1S + bI incremental cycles to execute, where S and I are as defined in *Section 2.5.1, Cycle Types*, and **Assembler Syntax**

CDP{cond} p#,<expression1>,cd,cn,cm{,<expression2>}

{cond} - two character condition mnemonic, see *Figure 2-5: Condition Codes*

p# - the unique number of the required coprocessor

<expression1> - evaluated to a constant and placed in the CP Opc field

cd, cn and cm evaluate to the valid coprocessor register numbers CRd, CRn and CRm respectively

<expression2> - where present is evaluated to a constant and placed in the CP field

Examples

```
CDP   p1,10,c1,c2,c3      ; request coproc 1 to do operation 10
                          ; on CR2 and CR3, and put the result in
CR1

CDPEQp2,5,c1,c2,c3,2      ; if Z flag is set request coproc 2 to do
                          ; operation 5 (type 2) on CR2 and CR3,
                          ; and put the result in CR1
```

2.4.13 Undefined Instruction

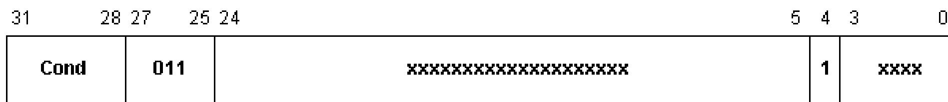


Figure 2-26: Undefined Instruction

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction format is shown in *Figure 2-26: Undefined Instruction*.

If the condition is true, the undefined instruction trap will be taken.

Note that the undefined instruction mechanism involves offering this instruction to any coprocessors which may be present, and all coprocessors must refuse to accept it by driving **CPA** and **CPB** HIGH.

Assembler Syntax

At present the assembler has no mnemonics for generating this instruction. If it is adopted in the future for some specified use, suitable mnemonics will be added to the assembler. Until such time, this instruction shall not be used.

2.5 Instruction Cycle Operations

2.5.1 Cycle Types

All memory transfer cycles can be placed in one of four categories:

[1] Non-sequential cycle. ARM7 requests a transfer to or from an address which is unrelated to the address used in the preceding cycle.

[2] Sequential cycle. ARM7 requests a transfer to or from an address which is either the same as the address in the preceding cycle, or is one word after the preceding address.

[3] Internal cycle. ARM7 does not require a transfer, as it is performing an internal function and no useful prefetching can be performed at the same time.

[4] Coprocessor register transfer. ARM7 wishes to use the data bus to communicate with a coprocessor, but does not require any action by the memory system.

The length of an internal cycle is always one system clock period. Sequential and non-sequential cycles are dependent on the speed of the memory or peripheral they are accessing. Page mode memories can take advantage of sequential accesses since the access times to consecutive locations are less than to unrelated locations. All on chip accesses are accomplished in a single clock period.

2.5.2 Branch and Branch with Link

A branch instruction calculates the branch destination in the first cycle, whilst performing a prefetch from the current PC. This prefetch is done in all cases, since by the time the decision to take the branch has been reached it is already too late to prevent the prefetch.

During the second cycle a fetch is performed from the branch destination, and the return address is stored in register 14 if the link bit is set.

The third cycle performs a fetch from the destination + 4, refilling the instruction pipeline, and if the branch is with link R14 is modified (4 is subtracted from it) to simplify return from SUB PC,R14,#4 to MOV PC,R14. This makes the STM..{R14} LDM..{PC} type of subroutine work correctly. The cycle timings are shown below in *Table 2-5: Branch Instruction Cycle Operations*, where:

pc is the address of the branch instruction

alu is an address calculated by ARM7

(alu) are the contents of that address, etc

Cycle	Address	Data	Cycle Type
1	pc+8	(pc + 8)	S
2	alu	(alu)	N
3	alu+4	(alu + 4)	S
	alu+8		S

Table 2-5: Branch Instruction Cycle Operations

2.5.3 Data Operations

A data operation executes in a single datapath cycle except where the shift is determined by the contents of a register. A register is read onto the A bus, and a second register or the immediate field onto the B bus. The ALU combines the A bus source and the shifted B bus source according to the operation specified in the instruction, and the result (when required) is written to the destination register. (Compares and tests do not produce results, only the ALU status flags are affected.)

An instruction prefetch occurs at the same time as the above operation, and the program counter is incremented.

When the shift length is specified by a register, an additional datapath cycle occurs before the above operation to copy the bottom 8 bits of that register into a holding latch in the barrel shifter. The instruction prefetch will occur during this first cycle, and the operation cycle will be internal (i.e. will not request memory). This internal cycle can be merged with the following sequential access by the memory manager as the address remains stable through both cycles.

The PC may be one or more of the register operands. When it is the destination external bus activity may be affected. If the result is written to the PC, the contents of the instruction pipeline are invalidated, and the address for the next instruction prefetch is taken from the ALU rather than the address incrementer. The instruction pipeline is refilled before any further execution takes place, and during this time exceptions are locked out.

PSR Transfer operations exhibit the same timing characteristics as the data operations except that the PC is never used as a source or destination register. The cycle timings are shown below *Table 2-6: Data Operation Instruction Cycle Operations*.

	Cycle	Address	Data	Cycle Type
normal	1	pc+8	(pc+8)	S
		pc+12		S
dest = pc	1	pc+8	(pc+8)	S
	2	alu	(alu)	N
	3	alu+4	(alu+4)	N
		alu+8		N
shift (Rs)	1	pc+8	(pc+8)	S
	2	pc+12	-	I
		pc+12		S
shift Rs, dest = pc	1	pc+8	(pc+8)	S
	2	pc+12	-	I
	3	alu	(alu)	N
	4	alu+4	(alu+4)	S
		alu+8		S

Table 2-6: Data Operation Instruction Cycle Operations

2.5.4 Multiply and Multiply Accumulate

The multiply instructions make use of special hardware which implements a 2 bit Booth's algorithm with early termination. During the first cycle the accumulate Register is brought to the ALU, which either transmits it or produces zero (depending on the instruction being MLA or MUL) to initialise the destination register. During the same cycle, the multiplier (Rs) is loaded into the Booth's shifter via the A bus.

The datapath then cycles, adding the multiplicand (Rm) to, subtracting it from, or just transmitting, the result register. The multiplicand is shifted in the Nth cycle by 2N or 2N+1 bits, under control of the Booth's logic. The multiplier is shifted right 2 bits per cycle, and when it is zero the instruction terminates (possibly after an additional cycle to reset a pending borrow).

All cycles except the first are internal. The cycle timings are shown below in *Table 2-7: Multiply Instruction Cycle Operations*. *m* is the number of cycles required by the Booth's algorithm; see the section on instruction speeds.

	Cycle	Address	Data	Cycle Type
(Rs)=0,1	1	pc+8	(pc+8)	S
	2	pc+12	-	I
		pc+12	(pc+8)	S
(Rs)>1	1	pc+8	(pc+8)	S
	2	pc+12	-	I
	•	pc+12	-	I
	m	pc+12	-	I
	m+1	pc+12	-	I
		pc+12		S

Table 2-7: Multiply Instruction Cycle Operations

2.5.5 Load Register

The first cycle of a load register instruction performs the address calculation. The data is fetched from memory during the second cycle, and the base register modification is performed during this cycle (if required). During the third cycle the data is transferred to the destination register, and external memory is unused. This third cycle may normally be merged with the following prefetch to form one memory N-cycle. The cycle timings are shown below in *Table 2-8: Load Register Instruction Cycle Operations*.

Either the base or the destination (or both) may be the PC, and the prefetch sequence will be changed if the PC is affected by the instruction.

The data fetch may abort, and in this case the destination modification is prevented.

	Cycle	Address	Data	Cycle Type
normal	1	pc+8	(pc+8)	S
	2	alu	(alu)	N
	3	pc+12	-	I
		pc+12		S
dest=pc	1	pc+8	(pc+8)	S
	2	alu	pc’	N
	3	pc+12	-	I
	4	pc’	(pc’)	N
	5	pc’+4	(pc’+4)	S
		pc’+8		S

Table 2-8: Load Register Instruction Cycle Operations

2.5.6 Store Register

The first cycle of a store register is similar to the first cycle of load register. During the second cycle the base modification is performed, and at the same time the data is written to memory. There is no third cycle. The cycle timings are shown below in *Table 2-9: Store Register Instruction Cycle Operations*.

Cycle	Address	Data	Cycle Type
1	pc+8	(pc+8)	S
2	alu	Rd	N
	pc+12		N

Table 2-9: Store Register Instruction Cycle Operations

2.5.7 Load Multiple Registers

The first cycle of LDM is used to calculate the address of the first word to be transferred, whilst performing a prefetch from memory. The second cycle fetches the first word, and performs the base modification. During the third cycle, the first word is moved to the appropriate destination register while the second word is fetched from memory, and the modified base is latched internally in case it is needed to patch up after an abort. The third cycle is repeated for subsequent fetches until the last data word has been accessed, then the final (internal) cycle moves the last word to its destination register. The cycle timings are shown in *Table 2-10: Load Multiple Registers Instruction Cycle Operations*.

The last cycle may be merged with the next instruction prefetch to form a single memory N-cycle.

If an abort occurs, the instruction continues to completion, but all register writing after the abort is prevented. The final cycle is altered to restore the modified base register (which may have been over-written by the load activity before the abort occurred).

When the PC is in the list of registers to be loaded the current instruction pipeline must be invalidated.

Note that the PC is always the last register to be loaded, so an abort at any point will prevent the PC from being over-written.

	Cycle	Address	Data	Cycle Type
1 register	1	pc+8	(pc+8)	S
	2	alu	(alu)	N
	3	pc+12	-	I
		pc+12		S
1 register dest=pc	1	pc+8	(pc+8)	S
	2	alu	pc’	N
	3	pc+12	-	I
	4	pc’	(pc’)	N
	5	pc’+4	(pc’+4)	S
		pc’+8		S
n registers (n>1)	1	pc+8	(pc+8)	S
	2	alu	(alu)	N
	•	alu+•	(alu+•)	S
	n	alu+•	(alu+•)	S
	n+1	alu+•	(alu+•)	S
	n+2	pc+12	-	I
		pc+12		S
n registers (n>10) incl pc	1	pc+8	(pc+8)	S
	2	alu	(alu)	N
	•	alu+•	(alu+•)	S
	n	alu+•	(alu+•)	S
	n+1	alu+•	pc’	S
	n+2	pc+12	-	I
	n+3	pc’	(pc’)	N
	n+4	pc’+4	(pc’+4)	S
		pc’+8		S

Table 2-10: Load Multiple Registers Instruction Cycle Operations

2.5.8 Store Multiple Registers

Store multiple proceeds very much as load multiple, without the final cycle. The restart problem is much more straightforward here, as there is no wholesale over-writing of registers to contend with. The cycle timings are shown in *Table 2-11: Store Multiple Registers Instruction Cycle Operations*.

	Cycle	Address	Data	Cycle Type
1 register	1	pc+8	(pc+8)	S
	2	alu	Ra	N
		pc+12		N
n registers (n>1)	1	pc+8	(pc+8)	S
	2	alu	Ra	N
	•	alu+•	R•	S
	n	alu+•	R•	S
	n+1	alu+•	R•	S
		pc+12		N

Table 2-11: Store Multiple Registers Instruction Cycle Operations

2.5.9 Data Swap

This is similar to the load and store register instructions, but the actual swap takes place in cycles 2 and 3. In the second cycle, the data is fetched from external memory. In the third cycle, the contents of the source register are written out to the external memory. The data read in cycle 2 is written into the destination register during the fourth cycle. The cycle timings are shown below in *Table 2-12: Data Swap Instruction Cycle Operations*.

The data swapped may be a byte or word quantity (b/w).

The swap operation may be aborted in either the read or write cycle, and in both cases the destination register will not be affected.

Cycle	Address	Data	Cycle Type
1	pc+8	(pc+8)	S
2	Rn	(Rn)	N
3	Rn	Rm	N
4	pc+12	-	I
	pc+12		S

Table 2-12: Data Swap Instruction Cycle Operations

2.5.10 Software Interrupt and Exception Entry

Exceptions (and software interrupts) force the PC to a particular value and refill the instruction pipeline from there. During the first cycle the forced address is constructed, and a mode change may take place. The return address is moved to R14 and the CPSR to SPSR_svc.

During the second cycle the return address is modified to facilitate return, though this modification is less useful than in the case of branch with link.

The third cycle is required only to complete the refilling of the instruction pipeline. The cycle timings are shown below in *Table 2-13: Software Interrupt Instruction Cycle Operations*.

Cycle	Address	Data	Cycle Type	Processor Mode
1	pc+8	(pc+8)	S	old mode
2	Xn	(Xn)	N	exception mode
3	Xn+4	(Xn+4)	S	exception mode
	Xn+8		S	

Table 2-13: Software Interrupt Instruction Cycle Operations

For software interrupts, *pc* is the address of the SWI instruction.

For interrupts and reset, *pc* is the address of the instruction following the last one to be executed before entering the exception.

For prefetch abort, *pc* is the address of the aborting instruction.

For data abort, *pc* is the address of the instruction following the one which attempted the aborted data transfer.

Xn is the appropriate trap address.

2.5.11 Coprocessor Data Operation

Section 5.3.4 on Butterfly sleep mode should be read in conjunction with Section 2.5.11 and Section 2.5.12.

A coprocessor data operation is a request from ARM7 for the coprocessor to initiate some action. The action need not be completed for some time, but the coprocessor must commit to doing it before driving **CPB** LOW.

If the coprocessor can never do the requested task, it should leave **CPA** and **CPB** HIGH. If it can do the task, but can't commit right now, it should drive **CPA** LOW but leave **CPB** HIGH until it can commit. ARM7 will busy-wait until **CPB** goes LOW. The cycle timings are shown in *Table 2-14: Coprocessor Data Operation Instruction Cycles*.

	Cycle	Address	Data	Cycle Type
ready	1	pc+8	(pc+8)	S
		pc+12		N
not ready	1	pc+8	(pc+8)	S
	2	pc+8	-	S
	•	pc+8	-	S
	n	pc+8	-	S
		pc+12		N

Table 2-14: Coprocessor Data Operation Instruction Cycles

2.5.12 Undefined Instructions and Coprocessor Absent

When a coprocessor detects a coprocessor instruction which it cannot perform, and this must include all undefined instructions, it must not drive **CPA** or **CPB** LOW. These will remain HIGH, causing the undefined instruction trap to be taken. Cycle timings are shown in *Table 2-15: Undefined Instruction Cycle Operations*

Cycle	Address	Data	Cycle Type	Processor Mode
1	pc+8	(pc+8)	S	Old
2	pc+8	-	I	Old
3	Xn	(Xn)	N	undefined
4	Xn+4	(Xn+4)	S	undefined
	Xn+8		S	undefined

Table 2-15: Undefined Instruction Cycle Operations

2.5.13 Unexecuted Instructions

Any instruction whose condition code is not met will fail to execute. It will add one cycle to the execution time of the code segment in which it is embedded (see *Table 2-16: Unexecuted Instruction Cycle Operations*).

Cycle	Address	Data	Cycle Type
1	pc+8	(pc+8)	S
	pc+12		S

Table 2-16: Unexecuted Instruction Cycle Operations

2.5.14 Instruction Speed Summary

Due to the pipelined architecture of the CPU, instructions overlap considerably. In a typical cycle one instruction may be using the data path while the next is being decoded and the one after that is being fetched. For this reason the following table presents the incremental number of cycles required by an instruction, rather than the total number of

cycles for which the instruction uses part of the processor. Elapsed time (in cycles) for a routine may be calculated from these figures which are shown in *Table 2-17: ARM Instruction Speed Summary*. These figures assume that the instruction is actually executed. Unexecuted instructions take one cycle.

Instruction	Cycle count	Additional
Data Processing	1S	+ 1I for SHIFT(Rs) + 1S + 1Nif R15 written
MSR, MRS	1S	
LDR	1S + 1N + 1I	+ 1S + 1Nif R15 loaded
STR	2N	
LDM	nS + 1N + 1I	+ 1S + 1Nif R15 loaded
STM	(n-1)S + 2N	
SWP	1S + 2N + 1I	
B,BL	2S + 1N	
SWI, trap	2S + 1N	
CDP	1S + bI	
LDC,STC	(n-1)S + 2N + bI	
MCR	1N + bI +1C	
MRC	1S + (b+1)I +1C	

Table 2-17: ARM Instruction Speed Summary

n is the number of words transferred.

m is the number of cycles required by the multiply algorithm, which is determined by the contents of Rs:

Multiplication by any number between *2^(2m-3)* and *2^(2m-1)-1* takes 1S+mI *m* cycles for *1<m>16*.

Multiplication by 0 or 1 takes 1S+1I cycles

Multiplication by any number greater than or equal to 2^(29) takes 1S+16I cycles. The maximum time for any multiply is thus 1S+16I cycles.

m is 2 if bits[32:16] of the multiplier operand are all zero or one, *m* is 4 otherwise.

b is the number of cycles spent in the coprocessor busy-wait loop.

If the condition is not met all the instructions take one S-cycle.

Chapter 3 - Diagnostic Broadcast (BBM)

3.1 Overview

During systems development, there is conflict between the requirement to maintain maximum processor throughput, and the desire to interact with the system to achieve maximum observability of the processor activity. Traditionally this has been resolved, in part, by the use of expensive bond-out chips and In Circuit Emulation (ICE) support, to monitor and debug application code running on processor. With the advent of low-cost deeply embedded systems, further on-chip integration has increased, resulting in constraints on system observability through a limited number of external package pins. Additionally, the required capital investment to develop high speed analysis and debugging systems based on traditional in-circuit emulation techniques has risen dramatically, whilst the target market requires cheaper and more flexible tools. Furthermore, system integration is increasingly performed at higher levels of the design, through analysis and debugging of high level language source code and real time kernel calls.

The BµILD implementation of diagnostic broadcast, using the Bµild broadcast module (BBM) is intended to satisfy the requirement that maximum observability of the components of the embedded system be provided without compromising the performance of the processor or bus. In addition, the implementation seeks to provide a straight-forward interface that can be used with industry standard development equipment such as Logic Analysers.

3.1.1 Diagnostic Broadcast

In order to further facilitate real-time debugging of application code, Butterfly provides a four bit diagnostic port, '**Bdiag**', which can be used to identify the current bus master together with the type of cycle that is active during each cycle of the clock. In order to save on device pins, this information is time multiplexed as shown below

During the low phase of '**Sclk**', '**Bdiag**' carries a binary coded representation of the bus master which initiated the current Bμμild bus transaction. The encoded value for the cycle type is output on '**Bdiag**' during the high phase of '**Sclk**'.

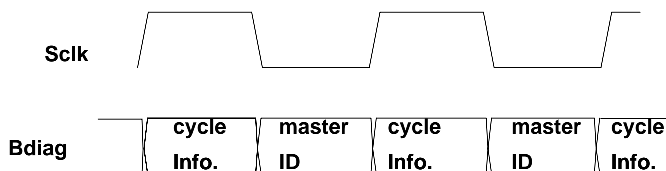


Figure 3-1: BμILD cycle and bus master details on 'Bdiag pins

The encoding for the bus master ID is shown in Table 3-1, while the encoding for the cycle type is shown in Table 3-2

Code				Master
Bdiag<3>	Bdiag<2>	Bdiag<1>	Bdiag<0>	
1	1	0	0	ARM7 Core
1	1	1	0	DMA Controller
1	1	1	1	Bμild Broadcast Module

Table 3-1: Encoding of Bus Master ID for Diagnostic Broadcast

Code				Cycle Type
Bdiag<3>	Bdiag<2>	Bdiag<1>	Bdiag<0>	
1	1	1	1	Wait Cycle
1	1	1	0	Internal Cycle
1	1	0	1	Coprocessor Read Cycle
1	1	0	0	Coprocessor Write Cycle
1	0	1	1	Instruction Fetch - Non-Sequential* - Not-Executed**
1	0	1	0	Instruction Fetch - Sequential* - Not-Executed**
1	0	0	1	Instruction Fetch - Non-Sequential* - Executed**
1	0	0	0	Instruction Fetch - Sequential* - Executed**
0	1	1	1	Word Read - Non-Sequential*
0	1	1	0	Word Read - Sequential*
0	1	0	1	Byte Read
0	1	0	0	Unused
0	0	1	1	Word Write - Non-Sequential*
0	0	1	0	Word Write - Sequential*
0	0	0	1	Byte Write
0	0	0	0	Unused

Table 3-2: Encoding of Cycle Type for Diagnostic Broadcast

* The ARM7 processor distinguishes between sequential memory accesses and non sequential memory accesses. This information can be used by a logic analyser to determine what is happening during internal accesses to the macrocells.

** The Executed/Not-Executed information indicates whether the instruction at the execute stage of the ARM7 pipeline is being executed. This does not refer to the data that is present on the data bus. The reason for an instruction not executing is when that instruction is conditional and the CPSR values do not match the condition. All other cases will result in the Broadcast data indicating that the instruction has been executed.

It should be noted that when the pipeline is flushed due to the program flow being changed, the instructions that are already present within the pipeline are changed to NOP (No OPeration) instructions and are still executed.

Chapter 4 - Phase Locked Loop (PLL)

4.1 Overview

When data is supplied to a chip it is usually synchronised with a clock. Unfortunately the clock signal path characteristics within a chip is typically different from that of the data. If the data at the pins is removed at the same time as the active clock edge, It is therefore not always possible to guarantee that data will be latched on the internal circuit. Also, for correct operation, the mark:space ratio of the clock can be critical, especially at higher clock rates, requiring the use an expensive external oscillator.

In order to help overcome these problems, Butterfly contains an internal Phase Locked Loop (PLL) and a crystal oscillator driver. These can be configured to produce a clock, internal to the device, which is phased-advanced such that its phase matches that of the clock applied externally. They can also be used to produce a stable system clock from a low cost crystal.

4.2 Design features

- Provides a means of generating a phase shifted clock to minimise the requirements for positive hold times on latched data inputs
- Generates a balanced Mark:Space ratio clock from an uneven Mark:Space ratio User-clock input.
- Provides a means of generating a stable high frequency balanced Mark:Space ratio clock from a low frequency commercial crystal. (Note: Under these circumstances a fixed x4 multiplication factor is used and the clock generated is made available to the User).
- Option to bypass the PLL. This allows the use of clocks which are outside the PLL capture range.

4.3 Architecture

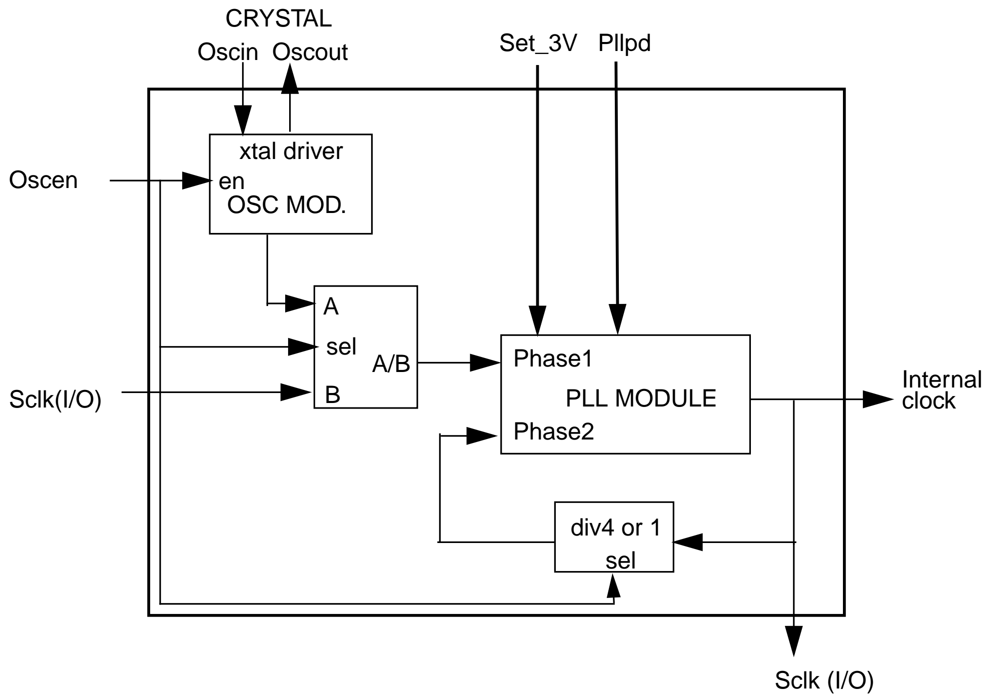


Figure 4-1: Architecture

4.3.1 Interface Definitions

Table 4-1 shows the signal definition and names.

Pad name	Type	Definition
Oscen*	I/P	0 : Oscillator Disabled 1 : Oscillator Enabled
Set_3V	I/P	0 : 5V supply 1 : 3V supply
Pllpd	I/P	0 - PLL circuit operational 1 - PLL circuit powered down
Sclk	I/O*	I/P - User clock in (Oscin =1) O/P - PLL Multiplied clock out (Oscin =0)
Oscin/Bypass	I/P	Crystal connection 1 OR 0 - Clock synchronisation mode 1 - PLL Clock bypass mode
Oscout	I/P	Crystal connection 2

*Note: See Table 4-2

Table 4-1: Signal definitions and Names

Oscen	Oscin/Bypass	Operational mode	Sclk- type
1	CRYSTAL	Multiplication	O/P - 4X Crystal
0	0	Synchronisation	I/P - User Clkin
0	1	Clock bypass*	I/P - User Clkin

Table 4-2: Clock Modes

*Note: With all clocks stopped ("Power Down mode") and with **Oscin/Bypass** held high, the PLL quiescent current is typically 10 - 15 uA

4.3.2 Operating Modes.

In addition to Power Down mode, three modes of operation (see Table 4-2) are available to the user, which can be summarised as follows:

- Clock Synchronisation mode:
  User clock supplied via **Sclk** pin. Internal balanced Mark:Space ratio clock generated, phase locked to **Sclk**.
- Clock Multiplication mode (crystal):
  Crystal connected to **Oscin/Bypass** and **Oscout** pins. x4 Crystal clock generated with balanced Mark:Space ratio and supplied to the user via **Sclk** pin.
- Clock Bypass mode:
  User clock is input via **Sclk** pin and routed directly to the system bus. There are no PLL operating frequency limitations in this mode as the PLL core circuitry is automatically powered down.

The Lock time of the PLL is up to 450us and is made up of the initial Lock which takes approximately 300 µs and a further 512 cycles in Lock before the internal **lock** signal goes active. Hence for 4MHz operation the Lock time is (512 x 250ns) + 300 µs = 428us. From power up the PLL output clock will tend to start at a low frequency and speed up to the Lock condition.

in Clock Multiplication and Clock Bypass mode, the output from the PLL is always enabled whether it is in Lock or not.

In Clock Synchronisation mode the output clock from the PLL will only be enabled when the PLL circuitry is in Lock. The only exception to this is with **nSreset** active (i.e. low), when the PLL output will also be enabled. Hence it is recommended that **nSreset** is held active at least for the period required to achieve Lock.

The PLL operational output ranges are as follows:

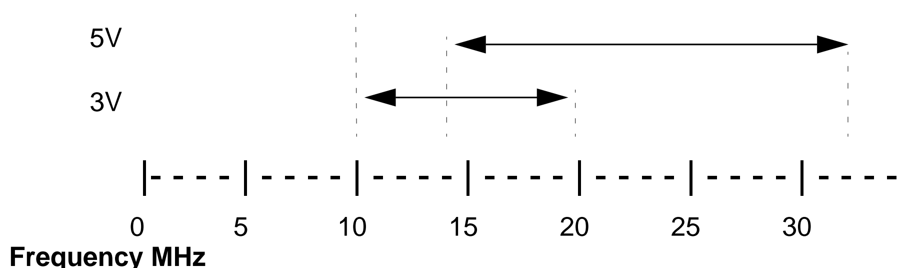


Figure 4-2: PLL Operational ranges

4.4 PLL Operational Description

In order to provide all the operating modes it is necessary to have three different PLL circuit configurations.

4.4.1 PLL for User Clock input.

When the oscillator circuit is disabled it is assumed that the User will supply an input clock. The PLL circuit will be configured as shown in Figure 4-3. Its function is to generate an internal clock which has a balanced Mark:Space ratio and is phase shifted so that the need for hold time on data input is minimised.

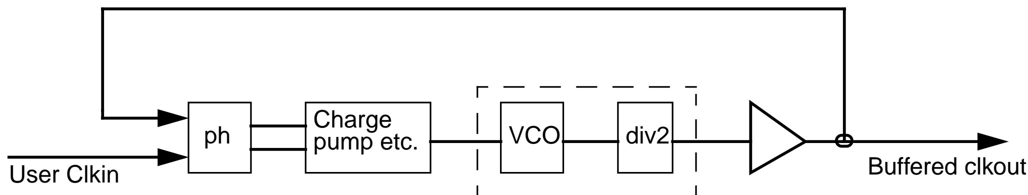


Figure 4-3: PLL circuit for User supplied clock.

4.4.2 PLL for Oscillator Clock input.

When the oscillator circuit is enabled it is assumed that the User requires a clock to be derived from a crystal via OSCMID. The PLL circuit will be configured as shown in Figure 4-4. It will generate an internal clock which has a balanced Mark:Space ratio and is at a frequency of four times that of the crystal connected to the oscillator input. In this way it is possible to generate a stable high frequency clock from a, low cost, low frequency crystal.

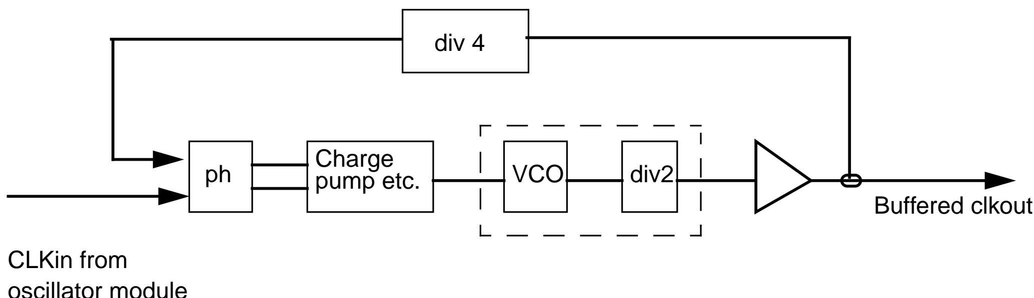


Figure 4-4: PLL circuit for Crystal supplied clock.

4.4.3 PLL Clock bypass

In order that the PLL can be operated outside its frequency limits the ability to bypass the PLL circuit has been included in the design. Figure 4-5 shows how the User clock input can be routed in to the chip via a mux which can be operated from the **Oscin/Bypass** pin. When using this configuration it will be necessary to ensure that data inputs have a +ve hold time with respect to the User clock.

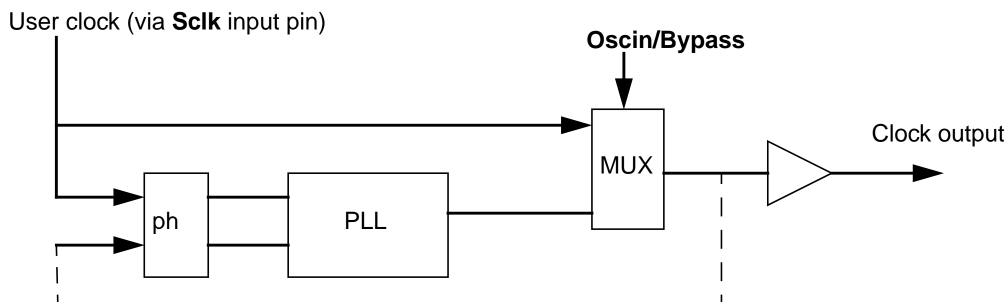


Figure 4-5: PLL circuit for PLL bypass mode.

4.4.4 PLL Operational details.

Upon power up the PLL will operate in one of 3 distinct modes of operation:

- PLL bypass mode; defined as Oscillator output not used, PLL disabled, with the external clock routed directly from the **Sclk** pin to the internal macrocells. **Note**: In this mode, **Pllpd** must be inactive ('0').
- Clock synchronisation; defined as Oscillator output not used and programmable divider set to divide by one mode.
- Clock multiplication; defined as Oscillator output used and programmable divider set to divide by four mode

In the last two cases it is necessary to select 3V or 5V operation via the **Set_3V** pin so that the VCO operates correctly.

Lock detection

PLL Lock detection is indicated via an internal **lock** signal. The PLL should obtain initial Lock within 300 µs, the detector then requires a further 514 cycles before it sets **lock** high. The output remains high until 4 consecutive cycles of out of Lock are detected. This protects against noise causing spurious loss of Lock.

Reset and Power down

One pin can be used for resetting the PLL macrocell:

- **Pllpd** is the PLL power down pin. A low on this pin means the PLL is operational, a high means it is powered down and the **lock** signal is reset.

It is important to consider the effect of the loss of the PLL clock output on the other modules. If the chip is placed in a stand-by mode whereby power is to be saved BUT instant response to an interrupt is required, then the PLL should not be powered down. Also, in order that other modules can be reset correctly a clock must be present. Therefore the **Pllpd** pin is not connected to the chip reset pin but is kept as a separate signal which is specific to the PLL. In this way, if the chip is reset the PLL will remain operational, allowing any synchronous resets to occur. In Clock Synchronisation mode, the PLL **lock** signal is used in conjunction with the chip reset signal (**nSreset**) to gate the PLL output clock. If **nSreset** is active i.e. low, then the PLL clock is always fed to the bus, otherwise the clock is only enabled when **lock** is active. If the chip is to be powered down completely then the PLL can be powered down BUT it must be remembered that the re-Lock time is greater than 300 µs.

4.5 Oscillator Operational Description

OSCMID is an amplifier cell designed to be configured with external components as a crystal oscillator (Pierce oscillator). The cell has been designed to drive crystals in the range 1 to 10MHz and will function as an oscillator with 3V or 5V power supplies. Output from the cell is via a Schmitt triggered gate and buffer stage which provides a square wave output suitable for internal clocks. The feedback resistor R_f has been included internally with the amplifier circuit.

4.5.1 Pin Descriptions

Oscin/Bypass - Crystal oscillator input terminal

Oscout - Crystal oscillator output terminal

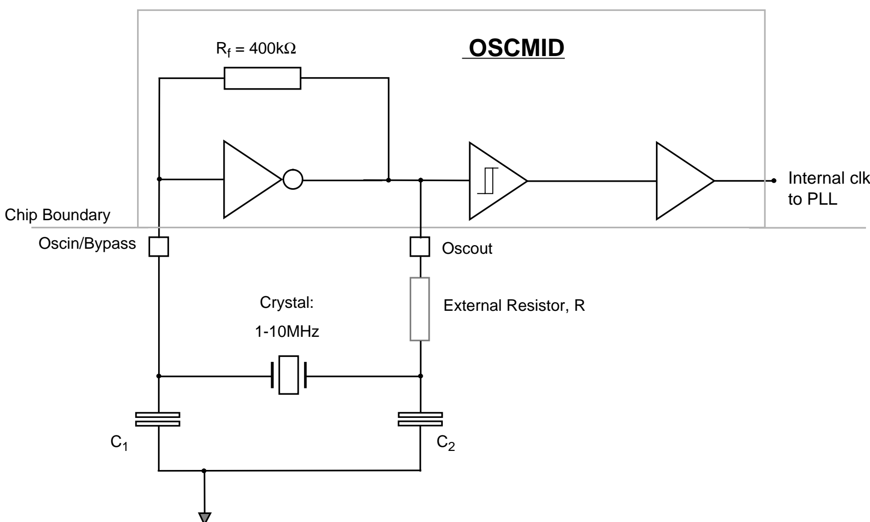


Figure 4-6: External Components

4.5.2 Selection of External Components

Once a crystal has been chosen, the load capacitors C_1 and C_2 can be selected. The capacitor values used ensure correct operation of the Pierce oscillator - such that the total loop gain is greater than unity. Correct selection of the two capacitors is very important and it is strongly recommended that the following method is used to obtain values for C_1 and C_2:

Loop gain equation

Although oscillations may still occur if the loop gain is just above 1, a loop gain of between 4 and 5 is recommended to ensure that oscillations will occur across all variations in temperature, process and supply voltage, and the circuit will exhibit good start-up characteristics.

eqn.1:

$$A = \frac{C_{out} g_m}{C_{in}} \times \left[\frac{(C_{out} + C_{in})}{(R_f C_{in})} + \frac{1}{Z_{in}} + \frac{1}{Z_o} \right]^{-1}$$

with **eqn.2:**

$$Z_{in} = \frac{1}{(2\pi f C_{out})^2 \times ESR}$$

List of equation parameters

A = total loop gain (set between 4 and 5)

$\mathbf{C_{in}}$ = (C_1 + $5x10^{-12}$) Farads

$\mathbf{C_{out}}$ = (C_2 + $5x10^{-12}$) Farads

$\mathbf{Z_o}$ = 18kΩ (output impedance of amplifier at 5V operation - typical*

$\mathbf{g_m}$ = 1.05 mA/V (transconductance of amplifier at 5V operation -typical*

$\mathbf{Z_o}$ = 50kΩ (output impedance of amplifier at 3V operation - typical*

$\mathbf{g_m}$ = 0.625 mA/V (transconductance of amplifier at 3V operation -typical*

$\mathbf{R_f}$ = 400kΩ (internal feedback resistor)

ESR = Equivalent series resistance of crystal - given by crystal manufacturer (Ω)

f = fundamental frequency of crystal (Hz)

*(see Amplifier Specification in section 4.5.3)

Equations for choice of External Resistor, R

The crystal drive level (P_c) may be higher than the manufacturers figure. The drive level can be reduced by adding a resistor in series with the crystal network (see Figure 4-1). To calculate the power dissipated in a crystal the following equation can be used.

eqn.3:

$$P_C = \frac{[V_{pp}]^2}{8 \times Z_{in}}$$

$\mathbf{P_c}$ = power dissipated in crystal at resonant frequency (W)

V_{pp} = Peak to peak output swing of amplifier:- **4.5V** for **Vdd=5V**, and **2.7V** for **Vdd=3V**.

f = frequency of crystal (Hz)

Z_{in} = crystal network impedance (see eqn.2)

If the power dissipation in the crystal is too high, eqn.4 may be used to calculate the series resistor needed to reduce the crystal drive level. Having decided the maximum power dissipation required in the crystal, a value for R can be calculated:

eqn.4:

$$R = \left[\frac{V_{pp}}{2} \times \sqrt{\frac{Z_{in}}{2 \times P_c}}\right] - Z_{in}$$

A check is now made using a modified loop gain equation, eqn.5, to ensure that the addition of R has not caused the loop gain to fall below a value of 4:

eqn.5:

$$A = \frac{C_{out} g_m}{C_{in}} \times \frac{Z_{in}}{Z_{in} + R} \times \left[\frac{(C_{out} + C_{in})}{(R_f C_{in})} + \frac{1}{Z_{in} + R} + \frac{1}{Z_o}\right]^{-1}$$

Worked Examples of Component Selection

Capacitor Values

The table below contains recommended pairs of capacitors which can be substituted into eqn.1 until a loop gain of between 4 and 5 is obtained. If two pairs of capacitor values give a loop gain between 4 and 5, the pair that yield the highest load capacitance C_L should be used. {N.B. $C_L=(C_{in}.C_{out})/(C_{in}+C_{out})$}

Note: Note, C_1 and C_2 are the capacitors used on the final oscillator design, but C_{in} and C_{out} are the physical capacitor values that are substituted into the loop gain equation. The difference between the capacitor values is due to the parasitic capacitance (value 5pF) of the pins and tracks associated with the circuit.

The equations given are used to calculate the components required for a 3.6MHz and a 10MHz crystal oscillator. Firstly, the gain for the two oscillators is found using eqn.1:

choice	C_1 /pF	C_2 /pF	C_{in} /pF	C_{out} /pF	C_L /pF	A - Osc1	A - Osc2
1	82	82	87	87	43.5	1.3	0.6
2	82	56	87	61	35.8	1.8	0.8
3	56	82	61	87	35.8	1.8	0.8
4	56	56	61	61	30.5	2.4	1.2
5	56	33	61	38	23.4	3.2	1.7
6	33	56	38	61	23.4	3.8	1.9
7	33	33	38	38	19	5.0	2.8
8	33	27	38	32	17.4	5.4	3.2
9	27	33	32	38	17.4	6.0	3.3
10	27	27	32	32	16.0	6.4	3.8
11	27	22	32	27	14.6	6.6	4.2
12	22	27	27	32	14.6	7.5	4.4
13	22	22	27	27	13.5	7.8	5

Osc1: 3.6MHz surface mount IQD crystal, ESR 200Ω, Vdd=5V

Osc2: 10MHz Microprocessor Crystal, ESR 35Ω, Vdd=3V

(For **Osc1**: Z_o=18kΩ, g_m=1.05e-3, R_f=400kΩ. For **Osc2**: Z_o=50kΩ, g_m=0.625e-3, R_f=400kΩ - see Amplifier Specification)

For **Osc1**, from the table choose **C_1=33pF** and **C_2=33pF** for a gain between 4 and 5. For **Osc2**, from the table choose **C_1=27pF** and **C_2=22pF** (the highest value of C_L, for a gain between 4 and 5).

Checking power dissipation:

<u>**Osc1:**</u> **C_{out}** is calculated as (33pF + 5pF) = 38pF, and **V_{pp}**=4.5V.

Using equations 2 and 3, the power dissipated is calculated as 374µW.

If a drive level of 200µW maximum is required eqn.4 can be used to calculate R:

Z_{in}=6767Ω, hence R=2487Ω. To reduce the drive level to the crystal to 200µW (max.), a **2700**Ω resistor is placed in series with the crystal network (*See Figure 4-6:*).

Checking the value of gain after adding R:

From eqn.5, the gain after adding R has been reduced to 4.5. If the adjusted loop gain had turned out to be less than 4, the next capacitor combination from the table would have to be chosen, and the loop gain and power values re-calculated.

The final choice of components for **Osc1:** C1=33pF and C2=33pF, R=2700Ω

4.5.3 Electrical Specification:

Parameters	TYP	Units	Conditions
Start up time	20	ms	Across frequency range

Table 4-3: Specification (3V and 5V nominal)

Parameter	minimum	typical	maximum
g_m mA/V	0.5	1.05	
Z_o kΩ		18	36

Table 4-4: Amplifier Specification: Vdd=5V (nom)

Parameter	min.	typical	maximum
g_m mA/V	0.31	0.625	
Z_o kΩ		50	80

Table 4-5: Amplifier Specification: Vdd=3V(nom)

4.5.4 Application Notes

[1] On the printed circuit board, the tracking to the crystal and capacitors must be made as short as possible. Other signal tracks must not be allowed to cross through this area. Ideally the component tracks should be ringed by a ground track connected to the chip ground (0V). It is advisable to provide a ground plane for the circuit to reduce noise.

Chapter 5 - Power Control (POCO)

5.1 Overview

The Power Control module allows flexible clock management to modules in the system.

There are 3 programmable methods for Power control on Butterfly:-

RUN mode: module clocks can be selectively enabled and disabled.

STANDBY mode: All on-chip Bus activity is disabled, however clocks to each module can be selectably enabled/disabled (programmable) until the occurrence of a hardware interrupt which will then put the system back into RUN mode.

‘SLEEP’ mode for the ARM core: Causes the ARM7 to go into an inactive state until a interrupt occurs.

Features of this module include:

- Individual module clock control in RUN mode
- Individual module clock control in STANDBY mode
- ‘SLEEP’ mode, to save power on the ARM core
- Minimal system wake-up latency

5.2 Architecture

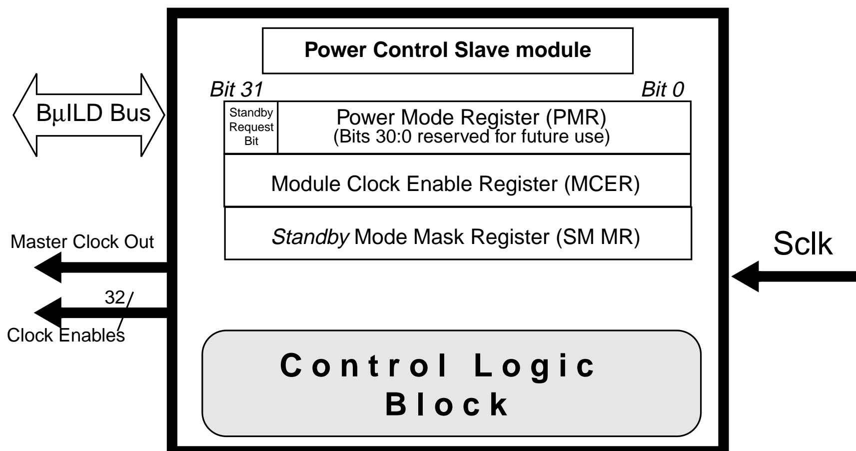


Figure 5-1: Block Diagram of POCO

5.3 Operational Description

5.3.1 System Reset/Power Up

Following an *abnormal exception* occurrence (e.g. a hard or soft system reset), the registers will default to the following states:

a. All bits in the Power Mode Register (**PMR)** reset to ‘0’, i.e. Standby-Request-Bit reset.

b. All bits in the Module Clock Enable Register (**MCER)** set to a ‘1’, i.e. all clock channels ENABLED in RUN mode.

c. All bits in the Standby Mode Mask Register (**SM MR)** set to a ‘1’, i.e. all clock channels ENABLED in STANDBY mode.

Nb: In order to minimise system static power consumption, the Reset input should not be held low for prolonged periods, as this pin has a 100KΩ (nominal) on-chip pull-up resistance.

5.3.2 Run Mode Operation

In normal system operation the BµILD bus µwill be in *RUN* mode. Writing directly to the ***Module Clock Enable Register*** will affect the clock enables of the modules at the beginning of the next clock cycle. A ‘1’ in any 32-bit position will ‘enable’ the respective module and writing a ‘0’ will ‘disable’ it. After Reset PoCo will enter Run Mode operation.

5.3.3 Standby Mode Clock Control Operation

Standby Mode allows the clock to one or more modules to be removed to save system power. The *Standby Mode Mask Register* is used to define which modules will still be clocked after Standby Mode has been entered.

Once the system is in *STANDBY* mode it will remain so until an *interrupt* occurs or the **Standby Request Bit** in the register is reset. In either case the Power Control module will request the system to ‘wake-up’ via the BBM. Note, when an *interrupt* occurs the ***Standby-Request-Bit*** in the **PMR** will be automatically reset to ‘0’.

Before the ***Standby-Request-Bit*** is set in the Power Mode Register, the **SM MR** will have normally been previously setup. A ‘1’ in any bit position will ‘enable’ (a ‘0’ will ‘disable’) the system clock to the associated channel when the module enters *STANDBY* mode.

When the ***Standby-Request-Bit*** is set the **PMR** sends a *Standby Request* signal to the BBM (Bµild Broadcast Module). When the BBM module receives this request it begins an orderly system-wide shutdown procedure where all modules (masters and slaves) remove themselves from the bus. When system shutdown is complete the Power Control module will enable the **SM MR** at the beginning of the next clock cycle.

5.3.4 Low Power Sleep Mode

The ARM 7 core does not inherently contain a low power sleep mode; however, the architecture does contain a mechanism for instruction set extension through the coprocessor interface. Mitel has taken advantage of this interface to define a coprocessor instruction set implementing different levels of low power operation.

This coprocessor is assigned coprocessor number 3, and performs Coprocessor Data operations (CDP). In the implementation contained within the Butterfly microcontroller, one coprocessor instruction is defined:

0 Suspend processor operation and halt processor clock until interrupt is received. Upon receipt of interrupt, execute interrupt service routine, then resume the normal flow of execution subsequent to the CDP instruction.

All other instructions for that co-processor, and all other coprocessor instruction types are reserved.

The assembly code for the SLEEP instruction is:

```
CDP p3,0,c0,c0,c0,#0
```

An interrupt impulse to the ARM will cause it to exit SLEEP mode. In certain circumstances, this may cause the ARM to enter into an UNDEF (Undefined Instruction) trap (to address 0x04). In order to return to normal program control, a MOVS PC,R14_und instruction should be placed at address 0x04 (see *Section 2.3.8, Undefined Instruction Trap*).

If the UNDEF trap is to be used for other purposes also, a test starting at location 0x04, will be necessary, to identify if the trap was as a result of an interrupt whilst in SLEEP mode.

Note: With the ARM processor in SLEEP mode, other Bus Masters (e.g.: DMAC) are still able to utilize the bus.

5.3.5 Typical Configuration

By default, all modules will be enabled in the RUN mode and disabled in the STANDBY mode.

5.4 Programmer's Model

Table 5-1 summarizes the programmers register views.

Address Offset (Hexadecimal)	Function	Read/Write	Reset Value
+ 000	Power Mode Register (PMR)	R/W	0x00
+ 004	Module Clock Enable Register (MCER)	R/W	0xFFFF
+ 008	Standby Mode Mask Register (SMMR)	R/W	0xFFFF
+ 00C-FFF	Reserved		x

Table 5-1: Programmers Register View

Table 5-2 shows the bit allocation used to enable (1) or disable (0) each individual module.

Bit	Module	Bit	Module
0-5	Reserved	16-17	Reserved
6	INTC	18	PPI
7-11	Reserved	19-23	Reserved
12	DMA	24	UART1
13	Reserved	25	UART2
14	TIC1	26-31	Reserved
15	TIC 2		

Table 5-2: Module Clock Enable Register (MCER)

Note: Bit values for all registers will be set to a “1” at power up. Clock control bits for any functions not needed in the application and for the reserved bits should be initialised to zero after power up in order to minimise power consumption.

5.4.1 Basic Register Operation

Power Mode Register (PMR)

This register allows overall control of the Power Control module. The only programmable bit is the **'standby request'** bit (bit 31). Writing a '1' (or '0') into this bit position 'enables' (or 'disables') STANDBY mode. The remaining fields have been reserved for possible future use and return a '0' if read.

Module Clock Enable Register (MCER)

This register contains 32 'channel enables' for the system in normal RUN mode. Writing a '1' (or '0') into any bit position in this register 'enables' (or 'disables') the associated channel at the beginning of the next clock cycle. Details of how this register is mapped to the individual modules are provided in Table 5-2.

STANDBY Mode Mask Register (SM MR)

This register contains 32 'channel enables' for the system when in STANDBY mode. Writing a '1' (or '0') into any bit position in this register will enables (or disable) the associated channel. Details of how bits within this are mapped to the individual modules are provided in Table 5-2.

Note, that writes to the **SM MR** will be disabled when the ***Standby-Request-Bit*** in the **PMR** is set.

Chapter 6 - Programmable Peripheral Interface (PPI)

6.1 Overview

The Programmable Peripheral Interface consists of 8 bidirectional data signals which may be individually set, reset or read. Alternatively, data signals may be grouped together to form one byte-wide bidirectional parallel port. In this mode, additional dedicated strobe signals are provided for hand-shaking with external components.

Features of this module include:-

- Port direction fully programmable
- Static/strobed operation
- Interrupt or polled operation

6.2 Architecture

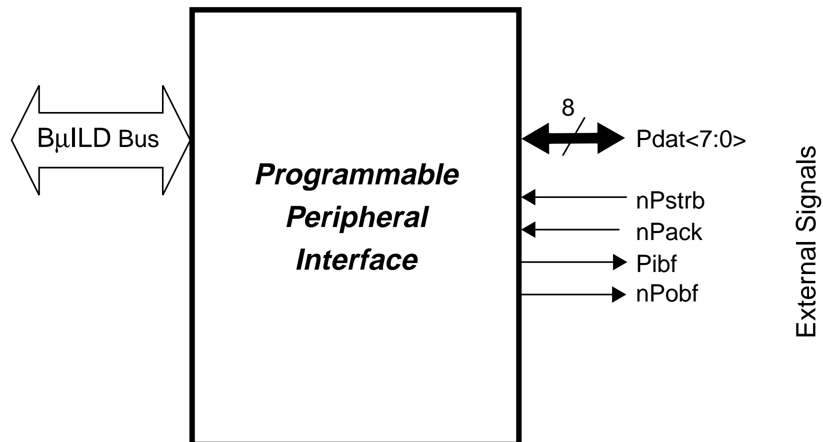


Figure 6-1: PPI Module Block Diagram

6.3 Operational Description

Static I/O (Mode-0)

In this mode, the dedicated input and output control pins are inactive. The direction of flow for each data bit is set in the **Data Direction Register (DDR)**. If a Data-pin is set up as an output, the value in the corresponding bit of the **Data Output Register (DOR)** is set onto the associated port data pin. If the pin is configured as an input, the output driver for that pin will be trisected. Reading the **Data Input Register (DIR)** gives the values being set onto the port pins, that is, the value read reflects the logic levels at the pins for each bit of the port.

The **DIR** is latched by the rising edge of the system clock, **Sclk,** during a **DIR** read operation, and changes in the **DOR** are reflected on the port pins on the falling edge of **Sclk,** which terminates the write access.

In Mode-0, Interrupts from the PPI module are disabled.

Strobed I/O (Mode-1)

The direction of data flow for each bit of the port can be individually set using the **DDR**, but in Mode-1 the latching of inputs and driving of outputs are determined by the port control signals.

Strobed Data Output

When data is written to the **DOR**, the **Output Buffer Full (nPobf)** pin is set low to indicate to the external device that there is data present in the port. Port bits which are set up as outputs, are held tri-stated until the external acknowledge signal (**nPack**) is set. Once **nPack** is set by the external device, the data values held in the corresponding **DOR** bits are presented on the port pins. When **nPack** is again reset, the port pins return to tri-state, the **nPobf** line is reset and the **Output Buffer Empty flag (OBE)** in the status register is set. An **OBE** interrupt is also generated - unless disabled by the **INTEN** bit in the **Control/Status register (CSR)**. Both the flag and the interrupt are reset by writing the **DOR**.

Strobed Data Input

If an external device wishes to clock data into the port, it should set the strobe line (**nPstrb**) once the data on the relevant connected port data pins is valid. This action causes the Input Buffer Full line (**Pibf**) to be set. The data is latched into the **DIR** on the rising edge of **nPstrb**, which also sets the **IBF** flag in the status register and generates an **IBF** interrupt (unless disabled). The interrupt and status bit are both reset by the rising edge of **Sclk** during a **DIR** read. The **Pibf** line is cleared by **Sclk** falling at the end of the read.

Bi-directional (Mode-2)

The data flow is fully bi-directional and controlled by the port control pins **nPstrb**, **nPack**, **Pibf** and **nPobf**, with the **DDR** being ignored.

To transfer to an external device, data is first written to the **DOR**, causing the **nPobf** line to be set and the **OBE** flag/interrupt to be reset. The external device then drives **nPack** low which enables the data drive to the port pins. Once the data has been latched by the external device, **nPack** is reset, setting the port data pins to tri-state, resetting the **nPobf** line and setting the **OBE** flag and interrupt.

For the external device to transfer data into the port, it must first drive the data onto the port pins. It then sets **nPstrb** low, which causes the **Pibf** line to be set high by the port. The **nPstrb** line is then reset, latching the data from the port pins into the **DIR** and setting the **Pibf** flag/interrupt. Reading the **DIR** resets the flag, interrupt and the **Pibf** line

Note: In both modes 1 and 2 there are certain limits on accessing the **DOR** and the **DIR** read. If the output register is being read by the external device (i.e. **nPack** set) then the **DOR** cannot be written to. If the input register is being written to by the external device (i.e. **nPstrb** set) then the **DIR** cannot be read.
Also, if the external device sets **nPack** low to read data from the port, the outputs will only be set if the output register is full, i.e. when **nPobf** is set. This enables ports to be linked for automatic data transfers by linking the data buses together and cross-coupling both the **nPstrb** and **nPobf** pins and the **nPack** and **Pibf** pins.

6.4 Programmer's Model

The PPI module contains 12 registers in total, 4 for the 8-bit port and an additional 8 registers reserved for future use. All registers are accessed as **32-bit words** with unused bits set to logic zero. Table 6-1 presents the programmers view of the register set. Addresses are specified as offsets from the system-defined PPI base address.

Address Offset (Hexadecimal)	Function	Register Size (bits)	R/W	Reset Value
+ 000	Data Direction Register (DDR)	8	RW	0xFF
+ 004	Data Input Register (DIR)	8	R	Undefined
+ 008	Data Output Register (DOR)	8	RW	0
+ 00C	Control/Status Register (CSR)	8	RW	0
+ 010 - +FFC	Reserved	-	-	

Table 6-1: Programmers Register View

6.4.1 Data Direction Register (DDR)

This register controls the direction of data transfer on the port I/O pins. It is used only in operating modes 0 and 1. If a bit is *set* (1) then the corresponding pin of the port is an input, if it is *reset* (0) then the pin is an output. The direction of the most significant I/O line is controlled by the most significant bit in the register.

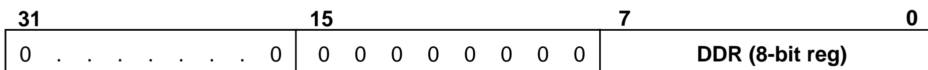


Figure 6-2: Data Direction Register (Read/Write)

6.4.2 Data Input Register (DIR)

This register reflects the current data on the port I/O pins – the value of the most significant I/O line occupying the most significant bit in the register. In static mode, data is latched from the port I/O pins into the **DIR** by a *read* operation; in strobed and bi-directional modes the data is latched by nPstrb rising. For port I/O pins which are currently defined as outputs (by the **DDR**) the corresponding **DIR** bit reflects the data value set from the **DOR**.

The 8-bit port has a corresponding 8-bit **DIR**.

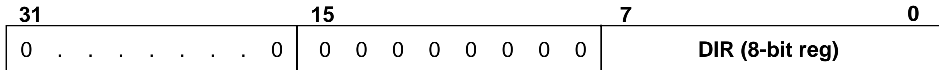


Figure 6-3: Data Input Register (Read Only)

6.4.3 Data Output Register (DOR)

This register holds the value which is to be set out on those port I/O pins which are defined as outputs – the most significant bit in the register driving the most significant port I/O line. In static mode, any change in the **DOR** is immediately reflected on the output pins. In strobed and bi-directional modes, the DOR can only be updated when nPack is clear (i.e.: no external READ being performed). Subsequent assertion of **nPack** causes the updated **DOR** value to be presented to the ports' output pins.

The 8-bit port has an 8-bit **DOR**.

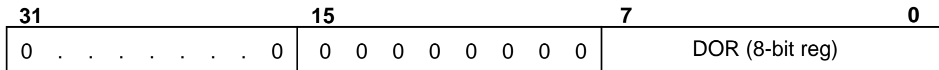


Figure 6-4: Data Output Register (Read/Write)

6.4.4 The Control Status Register (CSR)

This register contains the configuration and status information for the port. There are three such registers, one for the 8-bit port and two reserved for future use.

8-Bit Port Control Register

The 8-bit port control register is defined as follows:

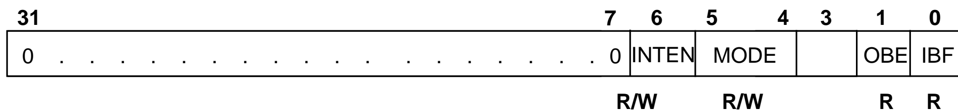


Figure 6-5: 8-Bit Port Control Status Register (CSR)

a. A single bit (**INTEN**) which controls the generation of interrupts from the port. When *set,* interrupts are enabled. When *reset,* interrupt generation is disabled.

b. A 2 bit field (**MODE**) which controls the current port's operating mode.

- **MODE**=00 Static I/O mode (Mode-0)
- **MODE**=01 Strobed I/O mode (Mode-1)
- **MODE**=1*x* Bi-directional I/O (Mode-2, *x*: Do not care)

c. A single read only bit (**OBE**) which indicates that the data output register (**DOR**) is empty. This flag may be reset by placing new data into the **DOR**.

d. A single read only bit (**IBF**) which indicates that the data input register (**DIR**) contains new data. This flag may be reset by reading the data value from the **DIR**.

6.5 Timing relationship diagrams

The diagrams in Figure 6-6 and Figure 6-7 show the relationship between the control signals used/generated when the port is in mode 1 or 2 (**nPstrb, nPack, nPobf** and **Pibf**), the data signals on the Port Data bus, and register access by the processor.

Port Data Output (Modes 1 & 2)

When the processor writes data to the **DOR**, **nPobf** will be set low, and after an external device sets **nPack** low, the data in **DOR** is output on the Port Data bus. When **nPack** is set high by the external device, **nPobf** is set high (inactive) and a (maskable) processor interrupt is generated setting the **OBE** flag in the **Interrupt Source Channel Register** (Table 1-7). This interrupt is reset by writing data to the **DOR**. The Ports Data Bus is returned to a high impedance state when **nPobf** is set high.

Port Data Input (Modes 1 & 2)

After an external device has put data onto the port data bus, it must drive **nPstrb** low and then high again. The rising edge of **nPstrb** latches the port data into the **DIR** and the following falling edge of **nPstrb** causes **Pibf** to be set high. A (maskable) processor interrupt will then result setting the **IBF** flag in the **Interrupt Source Channel Register** (Table 1-7). This interrupt is reset by the reading of **DIR** by the processor. Once the **DIR** has been read **Pibf** is reset.

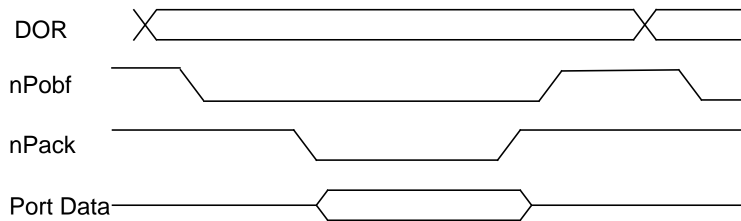


Figure 6-6: Port Data Output (Modes 1 and 2)

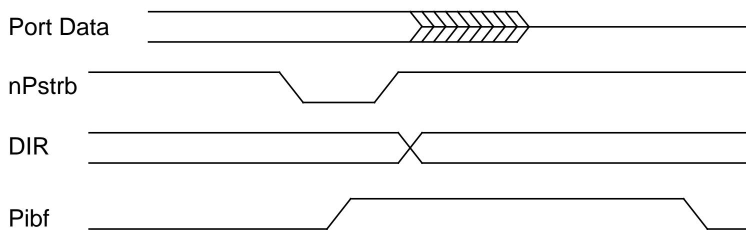


Figure 6-7: Port Data Input (Modes 1 and 2)

6.6 External Interface

Refer to Appendix B for further details.

Table 6-2 provides information on external interface pins.

Pin Name	Function	Dir	Description
Pdat<7:0>	Port Data	B	Port data input/output
nPstrb	Port Strobe [0]	I	Effective in Modes 1 & 2 only. External control input used to latch data into the **Data Input Register (DIR)**. Data is latched on the rising edge.
nPack	Port Acknowledge [0]	I	Effective in Modes 1 & 2 only. External control input used to clock data from the **Data Output Register (DOR)** onto the **Pdat<7:0>** pins. When high, **Pdat<7:0>** are tri-stated.
nPobf	Port OBF [0]	O	Effective in Modes 1 & 2 only. External control output When active, indicates new data is ready in the **Data Output Register (DOR)**
Pibf	Port IBF [1]	O	External control output reflecting the status of the input register. Set indicates valid data in the register.

Table 6-2: PPI Module External Pin List Description

Dir = Direction i.e. input (I), output (O) or bidirectional (B).

Chapter 7 - Memory/Peripheral Controller (MPC)

7.1 Overview

The MPC acts as the main gateway between internal and external bus systems. The external bus has to be capable of interfacing to the multitude of standard parts which could be used in conjunction with an ARM microcontroller within embedded applications.

Features include:

- Performs accesses to memories (ROM, SRAM), and peripherals (ADC's, DAC's, UART's etc.).
- Generates all control signals to access external components (chip-selects, write-enables, output-enable, etc.)
- Dynamic bus sizing, so that only accesses of the correct width are directed at the external component (8/16/32 bits - data width).
- Programmable wait state generation when accessing a specific external device.
- Programmable “stop wait state” generation, allowing for the slow turn-off time of slower external devices (e.g. ROM).
- Can perform an external access with zero wait states at up to the maximum system clock speed.
- Supports “fly-by” DMA mode of operation so that internal->external, external->internal, and external->external modes are supported.
- Supports both big and little endian operation.
- SWAP function allows RAM devices connected to **nCs3** (which is mapped to external area 4 at start up) to be switched to external area 1 (see section 1.3.1.2 on page 7)

7.2 Architecture

The diagram in Figure 7-1 shows the various functional blocks within the MPC. The functions of the various external pins are described more fully in Table 7-7, “External Pin-Out Description” .

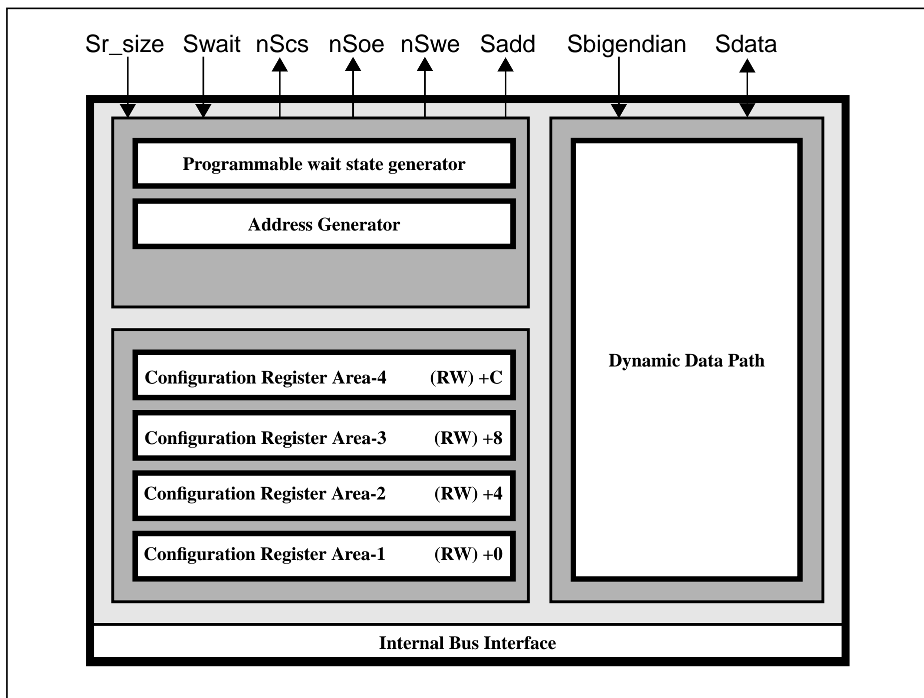


Figure 7-1: MPC Functional Blocks

7.3 Operational Description

In order for the microcontroller to do useful work, it must interface successfully to external memory and peripheral subsystems. The Memory & Peripheral Controller (MPC) is designed to provide a “glueless” interface to common memory sub-systems, such as ROM, EPROM & SRAM. It is also designed to interface to many types of external peripheral interface functions, ranging from analogue interfaces such as A/D and D/A convertors, to networking interfaces such as Serial Communications Controllers and Ethernet ports.

The MPC provides a standard functional set of external interface signals, for which it derives the appropriate timing relationships in order to exchange data between the microcontroller and external devices.

Throughout this section, please refer to the MPC **Configuration Register** format described in *Figure 7-11: Single Register Slice*

7.3.1 Memory Areas

The MPC is able to access up to four memory areas with varying characteristics. This allows a memory subsystem to be constructed using most appropriate types of memory. For example, one memory area could be set-up to contain slow ROM, another for fast SRAM, and a third for an external peripheral.

The characteristics of each memory area are set-up in a **Configuration Register** (see Figure 7-1). The parameters that can be varied include:

- automatic wait state generation, or wait indication supplied by memory subsystem
- number of wait states
- number of stop states (to allow slow memory/peripherals time to turn off)
- size of memory (e.g. 8, 16 or 32 bits wide)
- read only or read/write
- memory or peripheral
- capability to accept writes to less than the whole memory width

These parameters are described more fully in the following sections.

7.3.2 Signal Relationships

Figure 7-2 shows the relationship of external signals of the MPC.

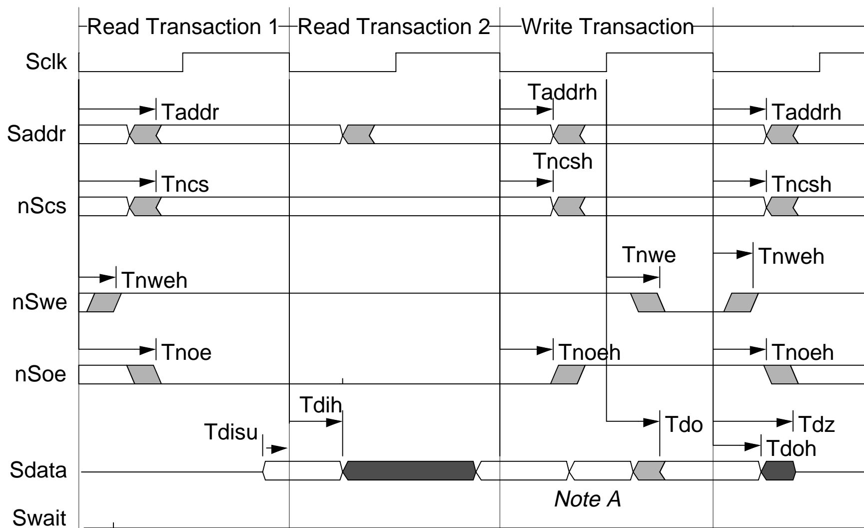


Figure 7-2: MPC External Interface Signals

During a read transaction the address (**Sadd**) and chip selects (**nScs**[3:0]) are generated shortly after the falling edge of the system clock (**Sclk**). The write enables (**nSwe**[3:0]) are *cleared* to a high state and the output enable (**nSoe**) is asserted. The external memory must then put the data on the bus (**Sdata**) within a prescribed set-up time before the next falling edge of the system clock.

The address and chip selects behave similarly in a write transaction. However the output enable is turned off at the start of the transaction. The write enables and the data to be written to the memory appear shortly after the first rising edge of the clock. After the end of the cycle the write enables are *cleared* first to avoid any error in the transaction. The data, address and chip selects are then allowed to change for the next cycle.

During adjacent read accesses to the same memory area, the signals **nScs**[0-3] and **nSoe** will remain permanently enabled. The new access is indicated solely by a change of address.

Note A: During a write cycle data becomes valid during the high phase of Sclk, however in order to ensure minimum propagation delay the Sdata drivers are enabled during the low phase. The data driven during this low phase will be the same as that supplied by any preceding read cycle and any potential clash of Sdata is avoided.

7.3.3 Wait State Insertion

While maximum system operating bandwidth is achieved when each memory or peripheral reference is completed in one clock cycle, it is often the case that wait states must be inserted to match the microcontroller speed to the timing parameters of the memory or peripheral being referenced. This may be the access time of the device, or some other timing parameter, such as data bus turn off time.

Automatic Wait State Generation

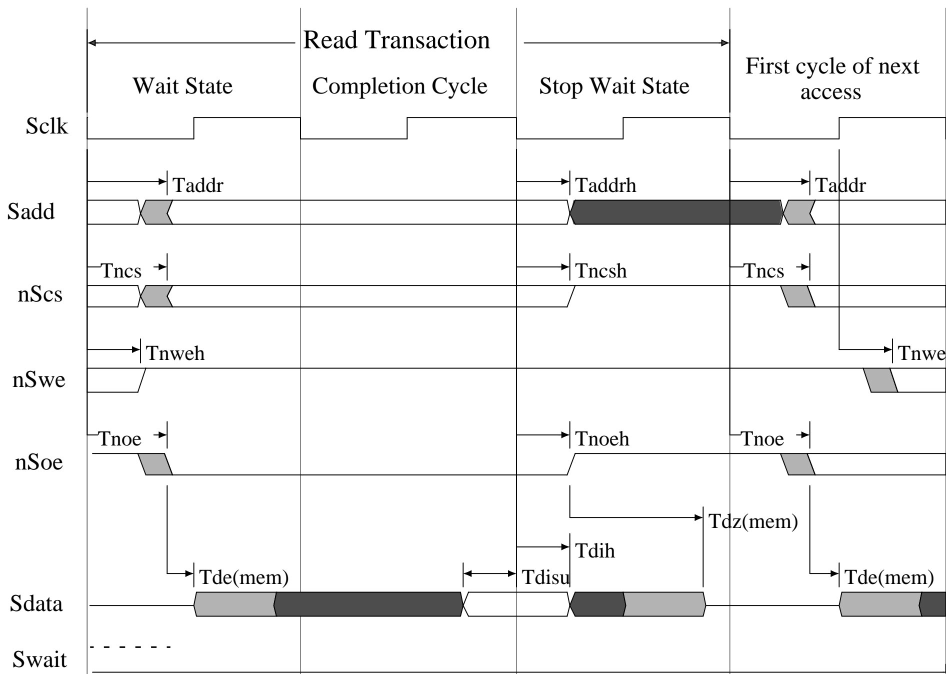
In most cases, the relationship between the clock period and the memory timing parameter is well understood. The MPC has a programmable wait state generator that can be set-up to introduce a number of wait states into each memory access. Two parameters are defined:

- the number of wait states required for the first and subsequent accesses to a memory or peripheral device, or **wait state (CRx[31:28])**.
- the number of wait states for which the data bus must be idle after the last access to a memory or peripheral device, or **stop wait (CRx[27:24])**.

The use of stop wait states is to allow for slow turn-off memories. If the next memory access was initiated before the memory devices have turned off from the previous transaction, there will be contention on the data bus. Therefore stop wait states are used to prevent another external memory access being started until after the previously addressed devices have turned off.

The application programmer must calculate the number of wait states or stop wait states required for the particular memory or peripheral connected to that memory area. An example of the effect of wait and stop states is shown in Figure 7-3 . This shows a single wait state, combined with a single stop wait state. The effect of this is that the memory access in total takes three clock cycles.

However, if in the clock cycle following the read transaction there is no access to external memory, the internal operation will be continue as normal. The operation of Butterfly is only stalled during stop wait states if the next access is also to external memory.



Tde(mem): memory turn on time
Tdz(mem): memory turn off time

Figure 7-3: Effect of Wait and Stop States on a Read Access

During read transactions the control signals (**Sadd**, **nScs**, **nSoe** and **nSwe**) are maintained at the same level throughout any wait states, until the end of the completion cycle. During any subsequent stop states, the chips select (**nScs**) and output enable (**nSoe**) signals are *cleared*, and the address may take on any value.

Write transactions behave in a similar manner with respect to wait states, except that **nSwe** becomes active after the first rising edge of **Sclk** in the transaction, as described in Section 7.3.2. It then remains active until the end of the completion cycle. However, stop wait states have no effect, since the Butterfly drives the data bus during write cycles, Therefore there is no necessity to allow a long turn-off time to prevent bus contention.

External Wait State Generation

In some cases, it cannot be determined at initialisation how many clock cycles are required to access a particular memory-mapped device. This is particularly true of bridging devices, which give access to another bus, where access latency is dependent on the current conditions on the remote bus. In this case, wait states must be applied externally, by the assertion of the Swait signal.

If the external wait state insertion signal is to be used, the **Configuration Register** corresponding to that memory-mapped device must be programmed for external wait states; **auto wait (CRx[5])** must be *clear*. An initial default wait state is inserted into the transaction when external wait states are indicated. **Swait** is used to indicate how many additional wait states should be inserted.

Swait is setup initially to the rising edge of the clock in the first cycle following the initial default wait state. While **Swait** is held high, the system will keep inserting wait states into the transaction. This is shown in Figure 7-4 .

When the transaction is to complete, **Swait** must be *cleared*. It should be *cleared* after the falling edge at the start of the cycle in which the access is to complete. For correct functioning **Swait** should be *clear* before the rising edge of **Sclk** in the middle of that cycle, as shown in Figure 7-4 .

Figure 7-4 shows the timing required for **Swait** during a read transaction. The control signals (**Sadd**, **nScs**, **nSoe** and **nSwe**) are maintained at the same level throughout the transaction until the end of the completion cycle. Write transactions behave in a similar manner, except that **nSwe** becomes active after the first rising edge of **Sclk** in the transaction, as described in Section 7.3.2. It then remains active until the end of the completion cycle.

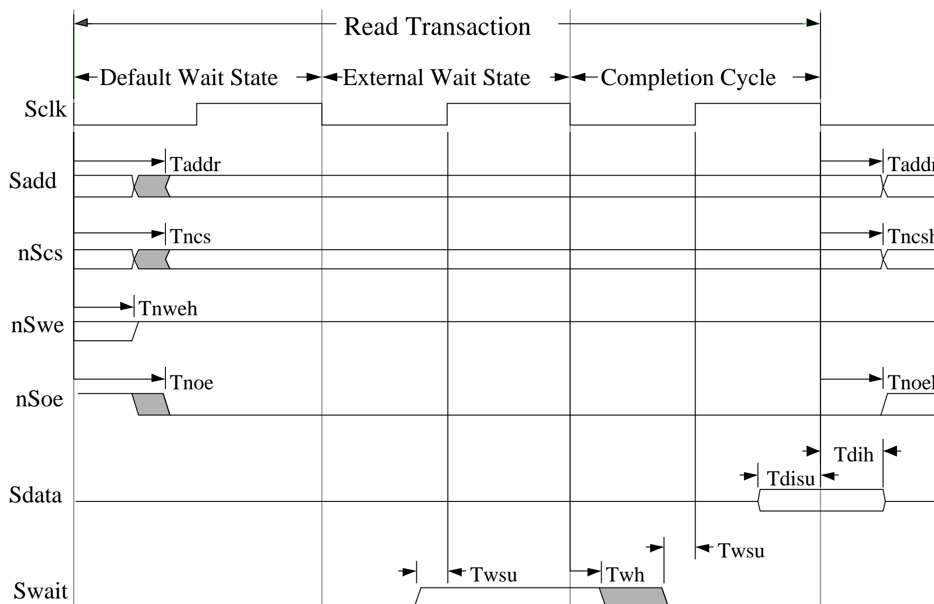


Figure 7-4: MPC Externally Generated Wait State Insertion

7.3.4 Instruction Fetches from Memory

Since the Butterfly microcontroller contains no internal memory, all processor instructions must be fetched from external memory. The ARM processor has a 32-bit instruction set; therefore 32-bit instructions must be fetched from the external memory sub-system.

However, it is not a requirement that the external instruction memory be configured as 32-bit memory, and it may be desirable from cost or size considerations to configure the memory as 16-bit or 8-bit memory. In these cases the MPC must reconstruct the 32-bit instructions from the data fetched from the memory subsystem. This function is performed by the dynamic data path sub-block within the MPC (see Figure 7-1).

It does this by performing multiple read accesses to the memory system, reconstructing the instruction in an internal buffer. When the instruction is complete, the MPC signals that the instruction fetch is over and passes the complete instruction back to the processor. Hence fetching an instruction from 16 bit memory will require two memory accesses, and from 8 bit memory will require 4 memory accesses.

Note that each read access to the memory sub-system may take one or more clock cycles, depending on the characteristics of the memory in use, and the way it has been configured in the MPC **Configuration Register**. The diagrams in Figure 7-5 , Figure 7-6 and Figure 7-7 show instruction fetches from a memory system requiring one wait state for each memory access. In this example **wait states (CRx[31:28])** should be set to **[0001]**.

Instruction Fetches from 32-bit Memory

In this case, each instruction is fetched in its entirety. Only one memory access is required per instruction. **Data Size (CR[1:0])** must be programmed to **[10]** in the appropriate **Configuration Register**.

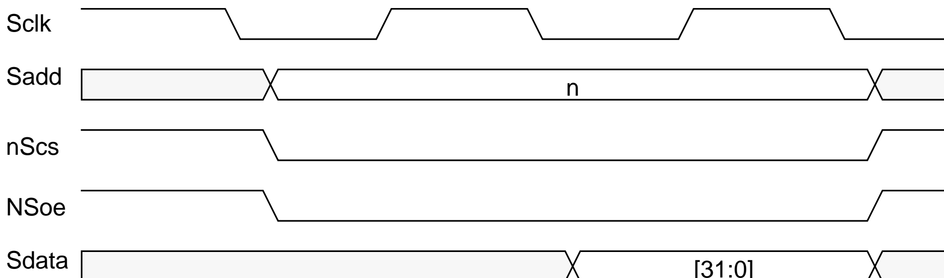


Figure 7-5: Instruction Fetch from 32-bit Memory (one wait state)

Instruction Fetches from 16-bit Memory

In this case, each instruction is fetched in two halves. Each half is fetched in consecutive memory accesses, lowest 16-bits first. **Data Size (CR[1:0])** must be programmed to **[01]** in the appropriate **Configuration Register**

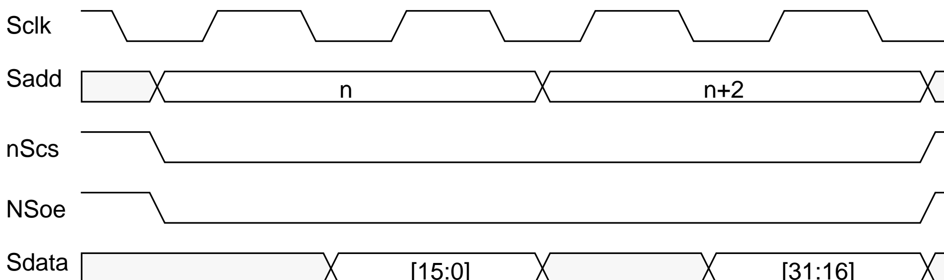


Figure 7-6: Instruction Fetch from 16-bit Memory (one wait state)

Instruction Fetches from 8-bit Memory

Here each instruction is fetched one byte at a time. Each byte is fetched in consecutive memory accesses, lowest order byte first. **Data Size (CR[1:0])** must be programmed to **[00]** in the appropriate **Configuration Register**.

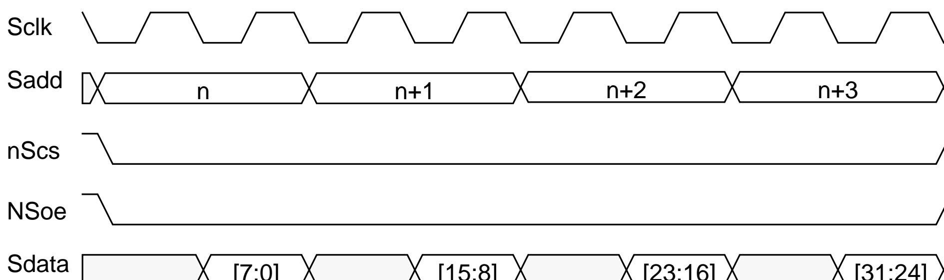


Figure 7-7: Instruction Fetch from 8-bit Memory (one wait state)

Since instructions are only fetched from memory, **Memory or Peripheral (CRx[2])** should be set. If the associated memory area only contains instructions and constants, and the variable data is maintained in a separate memory area, then **Read Only (CRx[4])** may be set. Illegal writes to this memory area will then cause a data abort exception, as described in Chapter 2: The ARM Processor.

7.3.5 Data Transfers to and from Memory

While instructions are defined to only be 32-bit operands, data operands may be either 8-bit, 16-bit or 32-bit in size. Similarly, the external memory or peripheral device may be either 8-bit, 16-bit or 32-bit in size. Some adaptation of the read or write data must therefore be done to translate the operand size into the memory or peripheral size. The mechanism employed depends on whether a read or a write operation is being performed.

Data Read from Memory

If the external memory size is greater or equal to the operand size, then the operand may be read from memory in a single memory access. However, if the external memory size is smaller than the operand size, then operand packing is required.

Operand packing describes the process in which the MPC performs multiple bus accesses to an external memory device to build up the required quantity of data for the requested transaction (using **Data Size (CRx[1:0]**). This is done in a similar manner to the way that 32 bit instructions are built up, as described in Section 7.3.4.

For example the MPC would carry out four 8-bit accesses to an 8-bit external memory in order to read a 32-bit data operand. In a similar manner the MPC would perform 2 accesses for 16-bit external memory.

The multiple read accesses required by operand packing rely on the external device residing in contiguous memory space. The device must be capable of handling the address bus changing (for loading the relevant the bytes or half words, whilst **nSoe** and it's chip select line remains active.

When the operand size requested is smaller than the memory data width, the MPC reads the entire memory word, ignoring the bits that are not required. The way in which read accesses are broken down is shown in Table 7-1.

Operand Size	Memory Size		
	8 bit	**16 bit**	**32 bit**
8 bit	Single 8 bit read	Single 16 bit read, (top 8 bits ignored)	Single 32 bit read (top 24 bits ignored)
16 bit	Two consecutive 8 bit reads, packed to form16 bit value.	Single 16 bit read	Single 32 bit read (top 16 bits ignored)
32 bit	Four consecutive 8 bit reads, packed to form 32 bit value.	Two consecutive 16 bit reads, packed to form 32 bit value	Single 32 bit read

Table 7-1: Data Read from Memory

Data Write to Memory

Data writes are handled in a similar manner to data reads. Hence a 32 bit write to an 8 bit memory will be broken down into four consecutive 8 bit writes. However, when the operand size is smaller than the memory width, it may not be possible to perform the requested transaction. This is because the memory may be incapable of accepting a write to less than the full memory width. In this case, the **Sub Memory Write** bit **(CRx[3])** must be *cleared*. Sub memory width writes to this memory area will then cause a data abort exception.

If the memory is capable of accepting sub memory width writes, the **Sub Memory Write** bit **(CRx[3])** should be *set*. The MPC uses separate write enable signals (nSwe[3:0]) to address each byte of the memory word. This allows it to perform 8 or 16 bit accesses to 32 bit wide external memories, and 8 bit accesses to 16 bit wide external memories. This is illustrated in Figure 7-8 :

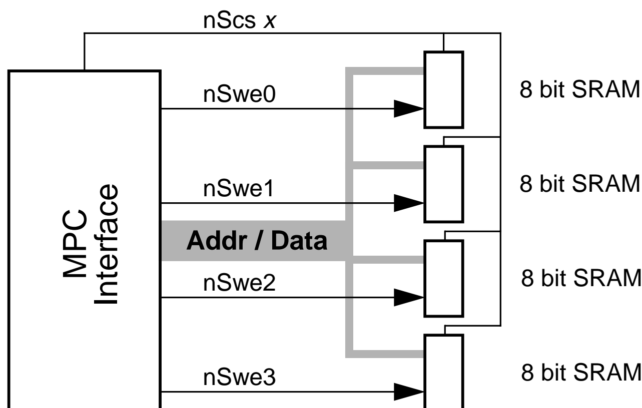


Figure 7-8: Writing to Individual Bytes of Memory

The way the MPC writes to various widths of memory is summarized in Table 7-2:

Operand Size	Memory Size		
	8 bit	**16 bit**	**32 bit**
8 bit	Single 8 bit write	Single 8 bit write (if the memory can accept sub-width writes)	
16 bit	Two consecutive 8 bit writes	Single 16 bit write	Single 16 bit write (if the memory can accept sub-width writes)
32 bit	Four consecutive 8 bit writes	Two consecutive 16 bit writes	Single 16 bit write

Table 7-2: Data Write to Memory

7.3.6 Endian configuration

The **Sbigendian** input determines whether the MPC treats words in memory as being stored in “Big Endian” or “Little Endian” format. This reverses the order in which individual bytes are stored within a larger data value.

In the Little Endian scheme (i.e. when **Sbigendian** is *clear*) the lowest numbered byte in a word is considered to be the least significant byte of the word and the highest numbered byte is the most significant. Byte 0 of the memory system should be connected to data lines 7 through 0 in this scheme.

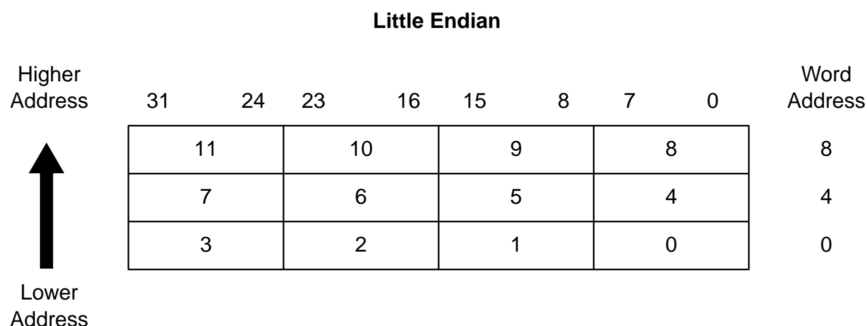


Figure 7-9: Little Endian addresses of bytes within words

In the Big Endian scheme (**Sbigendian** is *set*) the most significant byte of a word is stored at the lowest numbered byte and the least significant byte is stored at the highest numbered byte. Byte 0 of the memory system should therefore be connected to data lines 31 through 24.

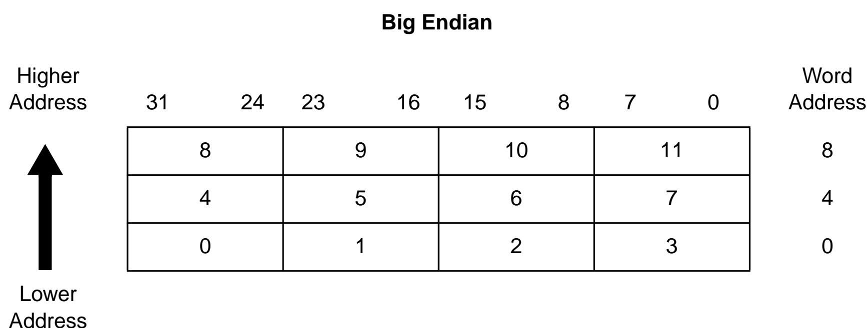


Figure 7-10: Big Endian addresses of bytes within words

Load and store are the only ARM7 instructions affected by the endian type, see *Little Endian Configuration* and *Big Endian Configuration* in *Section 2.4.7, Single Data Transfer (LDR, STR)* for more details.

The Endian type will also affect the use of DMA operations. Care should be taken to ensure that DMA operations do not attempt to transfer data from memory written with one endian scheme to memory written with the other scheme. This will result in a scrambling of the correct byte order.

7.3.7 Access to Non-Aligned Memory Addresses

Non-aligned accesses are defined as accesses where the address is not an integer multiple of the data size in bytes. For example, if x is a multiple of four, a 32 bit access to address x is aligned, whereas a 32 bit access to address x+1 is non-aligned.

The MPC does not handle non-aligned memory accesses. If an access is made to a non-aligned address, the MPC will ignore the non-alignment. The address is rounded down to the next nearest aligned address, and the data returned is the value starting from this aligned address.

Using the previous example, if a 32 bit access is requested from address x+1, the MPC will return the aligned data starting from address x. The same response will be obtained by a 32 bit access to address x+2 and x+3. When the address is x+4, this is aligned again, and the MPC returns the data starting from address x+4.

This is summarised in the Table 7-3 below:

<table>
<tr><th rowspan="2">Address Offset</th><th colspan="3">Data Bytes Returned</th></tr>
<tr><th>8 bit access</th><th>16 bit access</th><th>32 bit access</th></tr>
<tr><td>0</td><td>byte 0</td><td rowspan="2">bytes 0, 1</td><td rowspan="4">bytes 0, 1, 2, 3</td></tr>
<tr><td>+1</td><td>byte1</td></tr>
<tr><td>+2</td><td>byte 2</td><td rowspan="2">bytes 2, 3</td></tr>
<tr><td>+3</td><td>byte 3</td></tr>
<tr><td>+4</td><td>byte 4</td><td rowspan="2">bytes 4, 5</td><td rowspan="4">bytes 4, 5, 6, 7</td></tr>
<tr><td>+5</td><td>byte 5</td></tr>
<tr><td>+6</td><td>byte 6</td><td rowspan="2">bytes 6, 7</td></tr>
<tr><td>+7</td><td>byte 7</td></tr>
</table>

Table 7-3: Non-Aligned Address Accesses

7.4 Programmer's Model

Table 7-4 displays the register views for this module. All registers are accessed as 32-bit words. Addresses are specified as an offset from the system-defined base address.

The external memory areas for which the chip select signals **nScs[3:0]** become active are listed in Table 7-5.

Address Offset (Hexadecimal)	Function	Reset Value	R/W
+ 000	Area 1 Configuration Register	0xFF000034	RW
+ 004	Area 2 Configuration Register	0xFF000034	RW
+ 008	Area 3 Configuration Register	0xFF000034	RW
+ 00C	Area 4 Configuration Register	undefined	RW

Table 7-4: Programmer's Register View

Address (Hexadecimal)	Active Chip Select	External Memory Area
0000 0000 --> 1FFF FFFF	nScs[0]/nScs[3]*	Area 1
2000 0000 --> 3FFF FFFF	nScs[1]	Area 2
4000 0000 --> 5FFF FFFF	nScs[2]	Area 3
6000 0000 --> 7FFF FFFF	nScs[3]/nScs[0]*	Area 4

Table 7-5: MPC nScs External Decode Map

* When **SWAP** = 1 in System Configuration Register (See Section 1.3.1.2)

7.4.1 MPC Configuration Registers

32-bit wide registers are supplied with each chip select area of the external memory map. A single register slice is shown in Figure 7-11 :

Wait States *(4 bits)*	Stop Waits *(4 bits)*	*Reserved* *(18 bits)*	Auto wait	Read Only	Sub Memory Write	Memory or Peripheral	Data Size *(2 bits)*
31 28	27 24	23 6	5	4	3	2	1 0

Figure 7-11: Single Register Slice

The ***Data Size[1:0]*** bits indicate to the MPC control state machine how large the external data bus is which connects to the external component.

The ***Memory Or Peripheral*** bit indicates to the MPC what type of accesses are accepted by the external component. Peripheral components are defined as being unable to accept operand packing or sub memory width accesses. Therefore an area defined with the ***Memory Or Peripheral*** bit *cleared* will return a data abort if the operand size does not equal the memory size.

Memory components are defined as being able to accept operand packing and sub memory width reads (but not necessarily sub memory width writes). Therefore when the bit is *set*, the MPC will construct the requested transaction from one or more memory accesses.

The ***Sub Memory Write*** bit, when *set,* informs the MPC if the external memory is capable of receiving sub-memory width writes.

A sub memory width write is defined as a write access where the operand is smaller than the word width of the memory. For example, an 8 or 16 bit write to a 32 bit wide memory is a sub memory width write.

The following table (Table 7-6) shows the action for each of the combinations of ***Data Size***, ***Memory Or Peripheral***, *and* ***Sub Memory Write***.

Sub Memory Write	Memory Or Peripheral	Data Size bit 1	Data Size bit 0	Comments
X	0	0	0	8 bit peripheral
X	0	0	1	16 bit peripheral
X	0	1	0	32 bit peripheral
X	0	1	1	Illegal (reserved for future use)
X	1	0	0	8 bit memory
0	1	0	1	16 bit memory with no 8 bit writes
1	1	0	1	16 bit memory with 8 bit writes
0	1	1	X	32 bit memory with no 8 or 16 bit writes
1	1	1	0	32 bit memory with 8 and 16 bit writes
1	1	1	1	32 bit memory with 16 bit writes

Table 7-6: Configuration Register Bit Actions

The ***Read Only*** bit, when *set,* is used to inform the MPC that only read accesses can be performed to this external component. If the bus tries to write to this external component, then the MPC will cause a data abort.

The ***Auto Wait*** bit, when *set,* informs the MPC state machine that the current access has its wait states controlled by the internal wait state generator. When this bit is *clear* the wait states are controlled by the external signal **Swait**.

External Wait State generation

If external wait state control is required, the MPC inserts a default single wait state into the system. This is to allow the external memory system sufficient time to generate the **Swait** signal, which has to be returned before start of the next clock cycle. Therefore it is not possible to perform single cycle memory access when the auto wait bit is *cleared*.

The four ***Wait State*** bits define the length of time required to perform a memory access to the selected memory area.

The four ***Stop Wait*** bits are used by the MPC to hold off the ***next*** bus access. This is to allow slow peripherals time to get off the external bus before the start of the next external access. **Stop Waits** are therefore only applied on read accesses. These register bits must be loaded with zero if the **Auto Wait** bit is reset**.**

Wait States and **Stop Waits** are in ‘**n**’ multiples of system clock (**Sclk**), where ‘**n**’ is the value programmed into the register bit field (0-15).

MPC Configuration Register Reset Conditions

At power up, or after the **nSreset** signal has been asserted, the MPC registers reset into the following state:

Wait States *(4 bits)*	Stop Waits *(4 bits)*	*Reserved* *(18 bits)*	Auto wait	Read Only	Sub Memory Write	Memory or Peripheral	Data Size *(2 bits)*
1111	1111	0..0	1	1	0	1	00
31 28	27 24	23 6	5	4	3	2	1 0

Figure 7-12: Configuration Register Reset State

The reset state is designed as a “lowest common denominator”, the slowest, smallest, read only memory that can be configured. One of the first operations for a processor is to set-up the each MPC **Configuration Register** to the correct value.

There is one exception, the **Configuration Register** for memory area one. This resets to the above state except for its ***Data Size*** bits. These reset to the values on the MPC input signals **Sr_size[1:0]**. Memory area one starts at address 0000 0000, which is where the processor starts executing code from after reset. Therefore the **Sr_size** bits are used to indicate the width of the boot code memory to the MPC, allowing it to be set to something other than 8 bits wide if required. Once the processor has started up, it can then reset the speed of the memory as required.

7.5 External Interfaces

The following lists the external pins used for the MPC on Butterfly:

Signal [width]	Direction	Sense	Description
Sadd [21:0]	Output	-	Address Bus - 22 bits
nScs [3:0]	Output	Active low	Memory area chip select
nSwe [3:0]	Output	Active low	Byte Write Enable
nSoe	Output	Active low	Output Enable
Swait	Input	Active high	Wait State Request
Sdata [31:0]	Input/Output	-	Data Bus - 32 bits
Sr_size [1:0]	Input	[00]= 8 bits [01]=16 bits [1X]=32 bits	Boot ROM memory size configuration bits
Sbigendian	Input	Active high	BIG/LITTLE Endian Support BIG = 1, LITTLE = 0

Table 7-7: External Pin-Out Description

7.6 Application Information: Designing a Memory System

7.6.1 Example system configuration

An example memory system configuration is shown in Figure 7-13 . This shows a system that contains 8 bit wide EPROM in memory area 1. This is where the boot code for the microcontroller will reside. Note that the **Sr_size[1:0]** bits have both been tied to ground, to indicate that area 1 will contain 8 bit wide memory. Area 2 contains 32 bit wide fast SRAM, made up of four 8 bit wide devices. This will allow writes to individual bytes. Area 3 contains a 16 bit peripheral, and in this example memory area 4 is unused.

The system shown in Figure 7-13 is little endian. A big endian version of the same system is shown in Figure 7-14 . The only differences between the two are in how the **Sbigendian** pin is tied off, and in how the write enable signals (**nSwe[3:0]**) are connected.

In the big endian version, the write enables to the 32 bit SRAM are connected the opposite way round, i.e. **nSwe[0]** is connected to the byte at **Sdata[31:24]**, **nSwe[1]** is connected to the byte at **Sdata[23:16]**, **nSwe[2]** is connected to the byte at **Sdata[15:8]**, and **nSwe[0**] is connected to the byte at **Sdata[7:0]**.

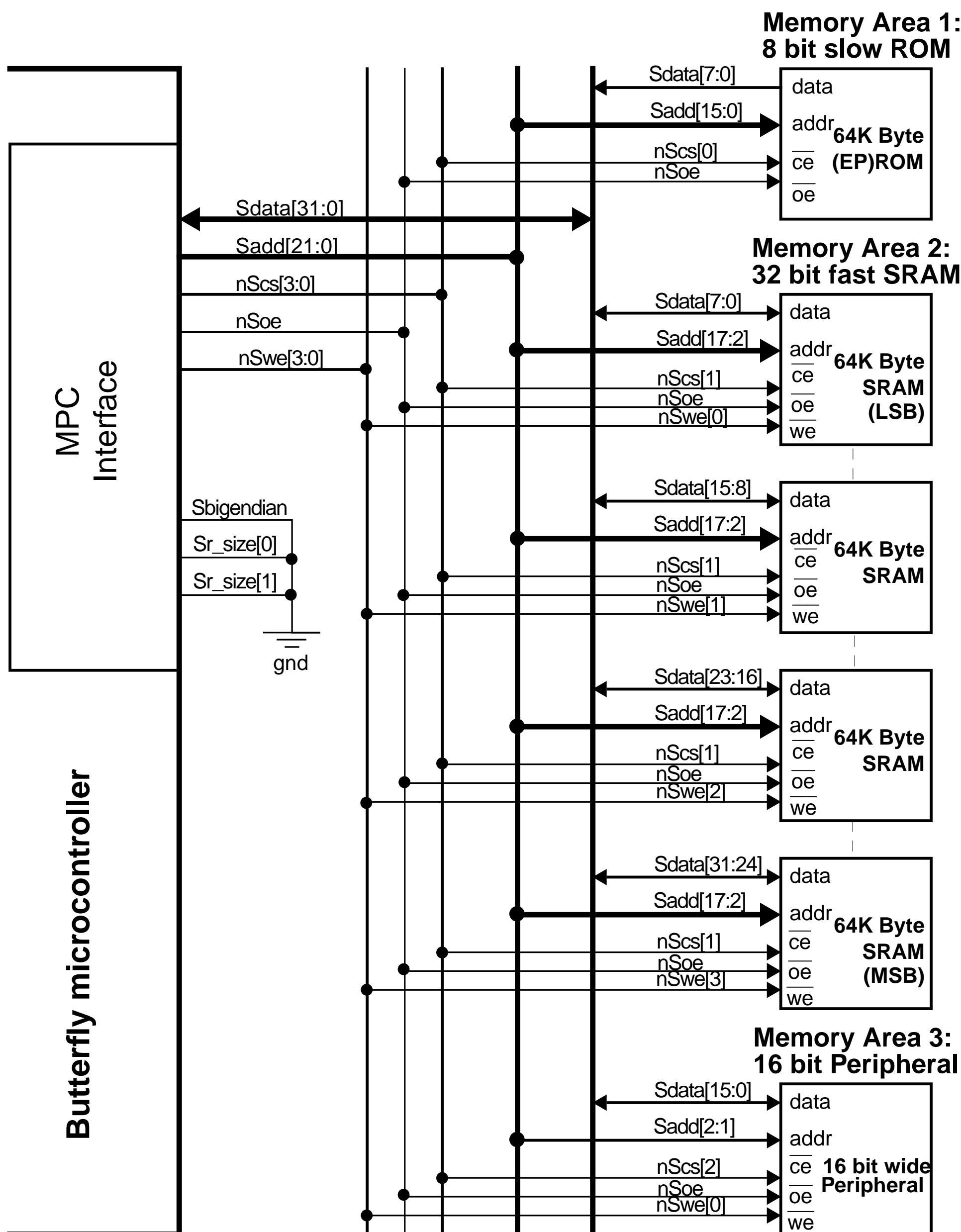


Figure 7-13: Example Little-Endian System Configuration

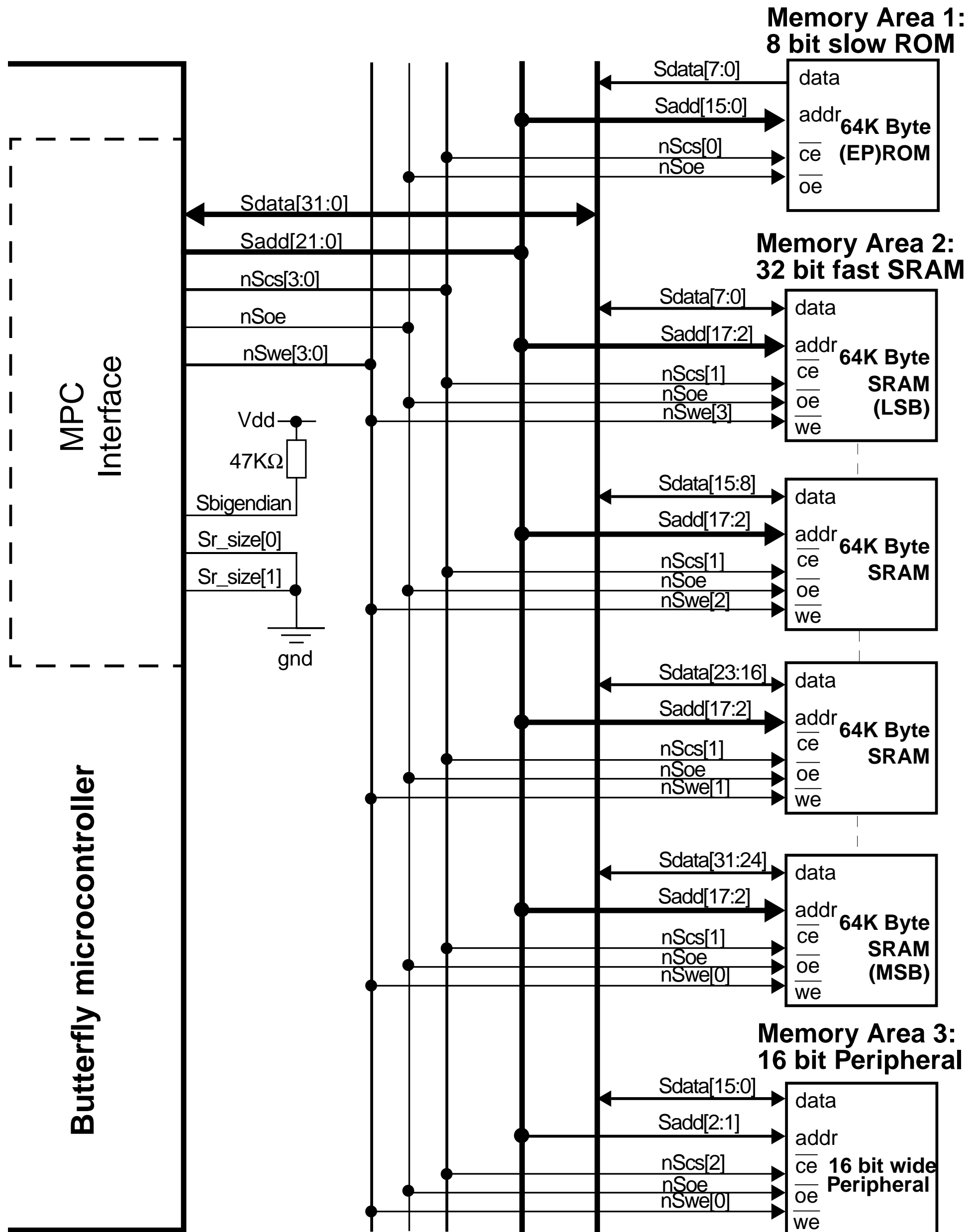


Figure 7-14: Example Big-Endian System Configuration

7.6.2 MPC Configuration Register Settings

A typical EPROM to use in the example might be a 27C512, with 100ns access time. At an operating speed of 25MHz (40 ns period), it would require 2 wait states to access this device. Similarly, the EPROM has a slow turn off time (typically 40 ns), therefore one stop state is required to prevent contention on the external data bus.

The **auto wait** bit must to be *set*, to enable automatic wait state generation. The memory is read only, so the **read only** bit can also be *set*. The **sub memory write** bit is redundant, since it is a ROM, and the **memory or peripheral** bit should be *set* to indicate memory. Finally the data width is 8 bits, so the **data size** bits should both be *cleared*.

In memory area 2, the SRAM to be used is fast enough to be used at zero wait state, zero stop state. Therefore both of these fields should be set to zero. Again the automatic wait state generator is being used, so **auto wait** is *set*. The memory is capable of accepting writes, therefore the **read only** bit should be *cleared*. The SRAM is capable of accepting both 8 and 16 bit writes, so the **sub memory write** bit is *set*, along with the **memory or peripheral** bit. Finally the size is 32 bits, so **data size** is set to 1,0.

Memory area 3 contains a 16 bit peripheral. In this example the peripheral has a 200 ns access time, therefore 5 wait states will be required. SInce the turn off time of its data drivers is 50 ns, two stop states are required to prevent bus contention.

The automatic wait state generator is being used, so **auto wait** is *set*. The peripheral is capable of accepting writes, therefore the **read only** bit should be *cleared*. Since peripherals cannot accept wrong-sized accesses, the **sub memory write** bit should also be *cleared*, along with the **memory or peripheral** bit. Finally the size is 16 bits, so **data size** is set to 0,1.

The MPC **Configuration Register** settings are shown in Figure 7-15 .

Wait States *(4 bits)*	Stop Waits *(4 bits)*	*Reserved* *(18 bits)*	Auto wait	Read Only	Sub Memory Write	Memory or Peripheral	Data Size *(2 bits)*
0010	0001	0..0	1	1	0	1	0 0
31 28	27 24	23 6	5	4	3	2	1 0

Memory Area 1 configuration: 8 bit slow EPROM

Wait States *(4 bits)*	Stop Waits *(4 bits)*	*Reserved* *(18 bits)*	Auto wait	Read Only	Sub Memory Write	Memory or Peripheral	Data Size *(2 bits)*
0000	0000	0..0	1	0	1	1	1 0
31 28	27 24	23 6	5	4	3	2	1 0

Memory Area 2 configuration: 32 bit fast SRAM

Wait States *(4 bits)*	Stop Waits *(4 bits)*	*Reserved* *(18 bits)*	Auto wait	Read Only	Sub Memory Write	Memory or Peripheral	Data Size *(2 bits)*
0101	0010	0..0	1	0	0	0	1 0
31 28	27 24	23 6	5	4	3	2	1 0

Memory Area 3 configuration: 16 bit peripheral

Figure 7-15: Example MPC Configuration Register Settings

7.6.3 Calculating Required Memory Timing Parameters

The timings provided with the MPC describe the characteristics of the Butterfly chip, but do not describe how to calculate what speed of RAM can be used in a Butterfly system. For example, it is not obvious what speed of SRAM is required to run single-cycle accesses. This section explains how to calculate the common timing parameters for a memory sub-system.

The diagram in Figure 7-16 shows a write transaction followed by a read transaction. The normal MPC timing parameters are shown in grey. The timing parameters required by a standard SRAM are shown in black. A description of the timing parameters for the MPC can be found in Table 7-8, whilst Table 7-9 contains a description of some typical SRAM timing parameters/constraints.

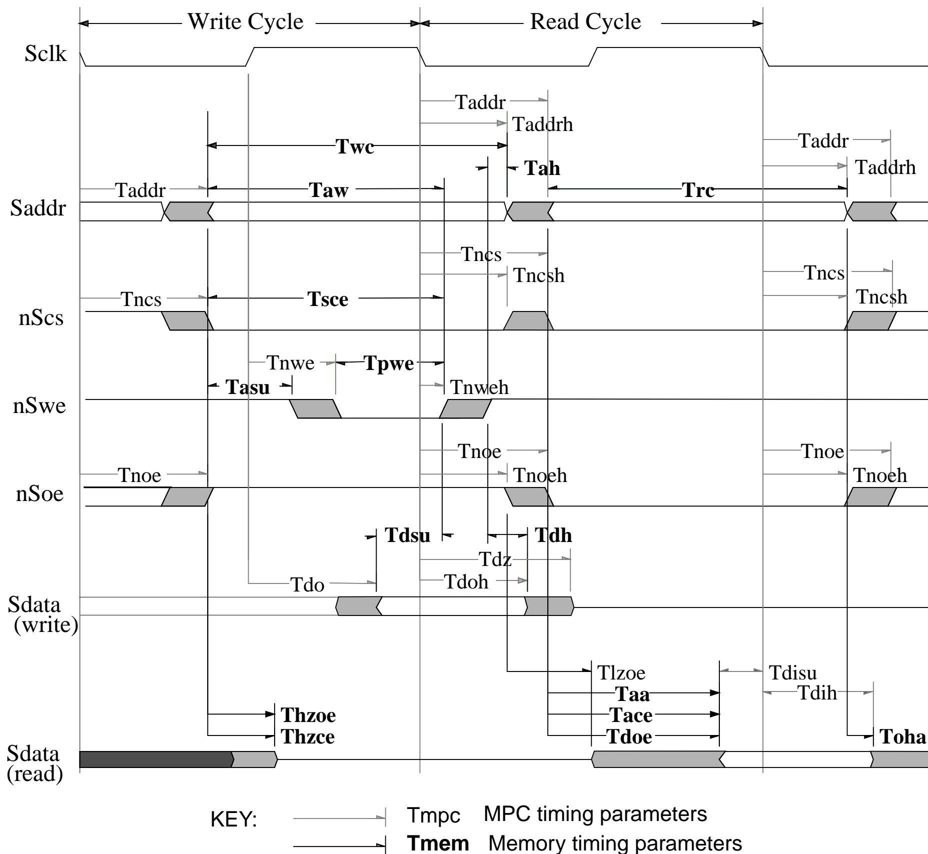


Figure 7-16: Interfacing the MPC to SRAM

In order to calculate what speed of memory is required, there are connecting formulae that can be used to check that the timing requirements are all met. These are given in Table 7-9. For example, to calculate the address access time required on a read cycle the formula is given as **Taa <=** n x (**Tclk**) **-** (**Taddr** + **Tdisu**)**.** This says that the memory access time, **Taa**, must be less than or equal to the number of clock periods, less the MPC address output delay, less the required input setup time on the MPC.

Throughout the equations the parameter **Tclk** represents the clock period to be used in the application, and the parameter n represents the number of clock cycles to be taken for each memory access in the memory area under consideration. This is one more than the number of wait states, as programmed into the **Configuration Register** (i.e. a one wait state access takes two clock cycles).

Name	Type	Description
Tclk	-	System clock period in application (Sclk)
Taddr	delay	Sclk falling to address (Sadd) valid
Taddrh	delay	address hold from Sclk falling
Tncs	delay	Sclk falling to chip select (nScs) valid
Tncsh	delay	chip select hold from Sclk falling
Tnoe	delay	Sclk falling to output enable (nSoe) valid
Tnoeh	delay	output enable hold from Sclk falling
Tnwe	delay	Sclk falling to write enable (nSwe) valid
Tnweh	delay	chip select hold from Sclk falling
Tdz	delay	data output turn off time
Tdo	delay	Sclk rising to data output valid
Tdoh	delay	Data output hold from Sclk falling
Tdisu	constraint	required data input setup time
Tdih	constraint	required data input hold time

Table 7-8: MPC Timing Parameters

Name	Type	Description and Formula
Twc	constraint	Write cycle time FORMULA: **Twc <=** n x (**Tclk**) **-** (**Taddr - Taddrh**)
Tsce	constraint	CE low to write end FORMULA: **Tsce <=** n x (**Tclk**) **-** (**Tncs - Tnweh**)
Taw	constraint	Address set-up to write end FORMULA: **Taw <=** n x (**Tclk**) **-** (**Taddr - Tnweh**)
Tah	constraint	Address hold from write enable FORMULA: **Tah <= Taddrh - Tnweh**
Tasu	constraint	Address setup to write start FORMULA: **Tasu <= Tclk/2 -** (**Taddr - Tnwe**)
Tpwe	constraint	Write enable pulse width FORMULA: **Tpwe <=** n x (**Tclk**) **-** (**Tclk/2 + Tnwe -Tnweh**)
Tdsu	constraint	Data input setup to write enable FORMULA: **Tdsu <=** n x (**Tclk**) **-** (**Tclk/2 + Tdo - Tnweh**)
Tdh	constraint	Data input hold from write enable FORMULA: **Tdh <= Tdoh - Tnweh**
Trc	constraint	Read cycle time FORMULA: **Trc <=** n x (**Tclk**) **-** (**Taddr - Taddrh**)
Taa	delay	Address access time to data valid FORMULA: **Taa <=** n x (**Tclk**) **-** (**Taddr +Tdisu**)
Toha	delay	Data hold after address change FORMULA: **Toha >= Tdih - Taddrh**
Tace	delay	CE low to data valid FORMULA: **Tace <=** n x (**Tclk**) **-** (**Tncs + Tdisu**)
Tdoe	delay	OE low to data valid FORMULA: **Tdoe <=** n x (**Tclk**) **-** (**Tnoe + Tdisu**)
Tlzoe	delay	Output enable to data drive FORMULA: **Tlzoe >= Tdz - Tnoe**
Thzoe	delay	Data output turn off from output enable FORMULA: **Thzoe <= Tclk/2 -** (**Tnoeh - Tdo**)
Thzce	delay	Data output turn off from chip enable FORMULA: **Thzce <= Tclk/2 -** (**Tncsh - Tdo**)

Table 7-9: Typical SRAM Parameters (with formulae)

Chapter 8 -

Chapter 9 -

Chapter 10 -

Chapter 11 -

Chapter 12 -

Chapter 13 -

Chapter 14 -

Chapter 15 -

Chapter 16 -

Chapter 17 -

Chapter 18 -

Chapter 19 -

Chapter 20 -

Chapter 21 -

Chapter 22 -

Chapter 23 -

Chapter 24 -

Chapter 25 -

Chapter 26 -

Chapter 27 -

Chapter 28 -

Chapter 29 -

Chapter 30 -

Chapter 31 -

Chapter 32 -

Chapter 33 -

Chapter 34 -

Chapter 35 -

Chapter 36 -

Chapter 37 -

Chapter 38 -

Chapter 39 -

Chapter 40 -

Chapter 41 -

Chapter 42 -

Chapter 43 -

Chapter 44 -

Chapter 45 -

Chapter 46 -

Chapter 47 -

Chapter 48 -

Chapter 49 -

Chapter 50 -

Chapter 51 -

Chapter 52 -

Chapter 53 -

Chapter 54 -

Chapter 55 -

Chapter 56 -

Chapter 57 -

Chapter 58 -

Chapter 59 -

Chapter 60 -

Chapter 61 -

Chapter 62 -

Chapter 63 -

Chapter 64 -

Chapter 65 -

Chapter 66 -

Chapter 67 -

Chapter 68 -

Chapter 69 -

Chapter 70 -

Chapter 71 -

Chapter 72 -

Chapter 73 -

Chapter 74 -

Chapter 75 -

Chapter 76 -

Chapter 77 -

Chapter 78 -

Chapter 79 -

Chapter 80 -

Chapter 81 -

Chapter 82 -

Chapter 83 -

Chapter 84 -

Chapter 85 -

Chapter 86 -

Chapter 87 -

Chapter 88 -

Chapter 89 -

Chapter 90 -

Chapter 91 -

Chapter 92 -

Chapter 93 -

Chapter 94 -

Chapter 95 -

Chapter 96 -

Chapter 97 -

Chapter 98 -

Chapter 99 -

Chapter 100 -

Chapter 101 -

Chapter 102 -

Chapter 103 -

Chapter 104 -

Chapter 105 -

Chapter 106 -

Chapter 107 -

Chapter 108 -

Chapter 109 -

Chapter 110 -

Chapter 111 -

Chapter 112 -

Chapter 113 -

Chapter 114 -

Chapter 115 -

Chapter 116 -

Chapter 117 -

Chapter 118 -

Chapter 119 -

Chapter 120 -

Chapter 121 -

Chapter 122 -

Chapter 123 -

Chapter 124 -

Chapter 125 -

Chapter 126 -

Chapter 127 -

Chapter 128 -

Chapter 129 -

Chapter 130 -

Chapter 131 -

Chapter 132 -

Chapter 133 -

Chapter 134 -

Chapter 135 - Universal Asynchronous Receiver/

Transmitter (UART)

135.1 Overview

The Universal Asynchronous Receiver Transmitter (UART) is a component that provides industry-standard levels of support for full-duplex asynchronous serial communications, with appropriate mechanisms for both software and hardware flow control. Standard modem handshake signals are incorporated to facilitate communications via intermediate devices in the communications channel.

Design Features

- Full duplex operation, independent transmit and receive channels
- 7 or 8-bit serial data length
- 1 or 2 stop bits
- Even, odd or no parity generation
- Internal selectable baud rate generator, derived from system clock
- Double buffered transmit and receive channels
- Software polling to determine channel status
- Optional interrupt generation on transmit channel becoming empty
- Optional interrupt generation on receive channel becoming full
- Detection of parity, overrun, and framing errors on receive channel, with optional interrupt generation
- Support for modem signals **RTS**, **CTS** and **DCD** with edge detection and optional interrupt generation
- Maximum transmission rates corresponding to 1/16th of the system clock*
- Power saving through automatic clock suspension of the transmit and receive channel circuits when both are disabled
- Clock suspended to modem control section when modem signals are disabled
- Digital input filter to improve noise immunity

Note: * This corresponds to the line rate and NOT the Data bandwidth which will be typically a maximum of 80% of this speed (i.e.: including Start and Stop bits).

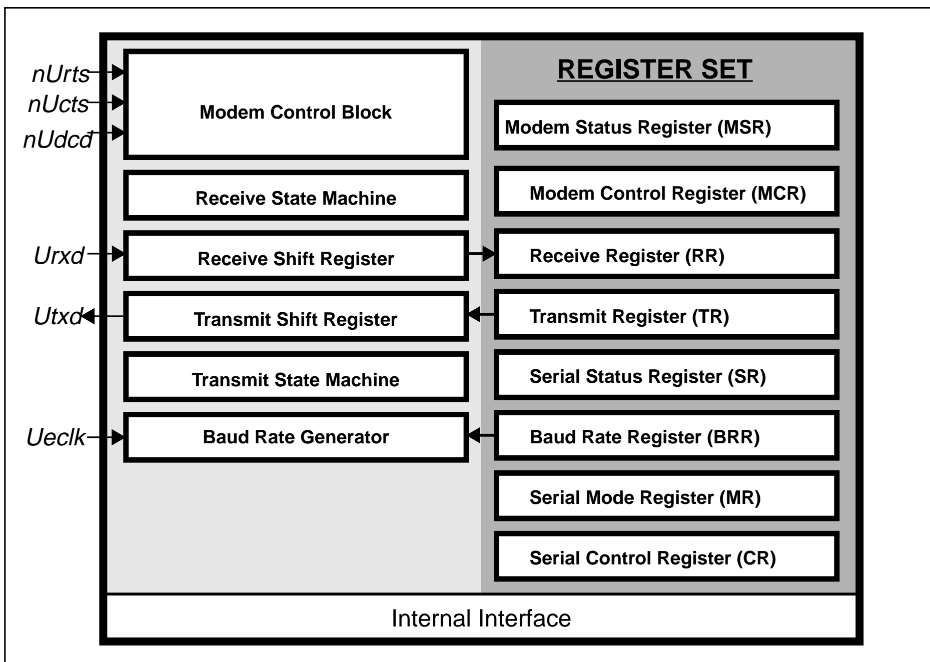


Figure 135-1: Block Diagram

Signal		Description
Utxd	Data out	**Serial transmit data** Idles high when no data is being transmitted
Urxd	Data in	**Serial receive data** Expects input signal to idle high when no data is being transferred
nUcts	Input, active low	**Clear To Send** When asserted, indicates that the modem or data set is ready to exchange data; the Modem Status Register monitors actual level and level changes. Optionally used as hardware handshake signal on either the transmit or receive channel
nUrts	Output, active low	**Ready To Send** When asserted, indicates that the UART is ready to exchange data. Optionally used as hardware handshake signal on either transmit or receive channel; set from Modem Control Register
Ueclk / nUdcd	Input, active low (nUdcd)	**External Clock / Data Carrier Detect Input** Selectable input - programmed in the System Configuration Register (section 1.3.4). When Ueclk, external source for baud rate generator. When nUdcd, indicates that the communications carrier is present.

Table 135-1: Interface Description

135.2 Operational Description

For register details, see Section 135.3 Programmer’s Model

135.2.1 Baud Rate Generation

The UART transmit and receive channels are clocked from a single, internally derived clock (**internal clock**), whose period is determined by the reference clock, and the value programmed into the baud rate register. The reference clock is selected, using **Clock Select (CR[2])**, from one of two sources:

- the external clock input **Ueclk**, or
- an internal prescaler driven by the system clock, **Sclk**

In the latter case, clock prescaler, 16 bits long, is configured to generate a reference clock of period **‘Sclk** divided by 1,2,4,... up to 32768’, as specified by the value programmed into **Division Select (MR[7:4])**.

The **Baud Rate Register** is used to divide the reference clock period within the range 1 (**BRR** = 0) to 256 (**BRR** = 255) to allow approximation to the desired baud rate. The required baud rate register value may be calculated according to the following formula:

BRR = ((Reference Clock) / (16 x Baud Rate)) - 1

The clock is not applied to the transmit or receive channels if neither channel is enabled or currently active. Similarly, the system clock is not applied to the modem control section when the modem enable bit of the control register is reset.

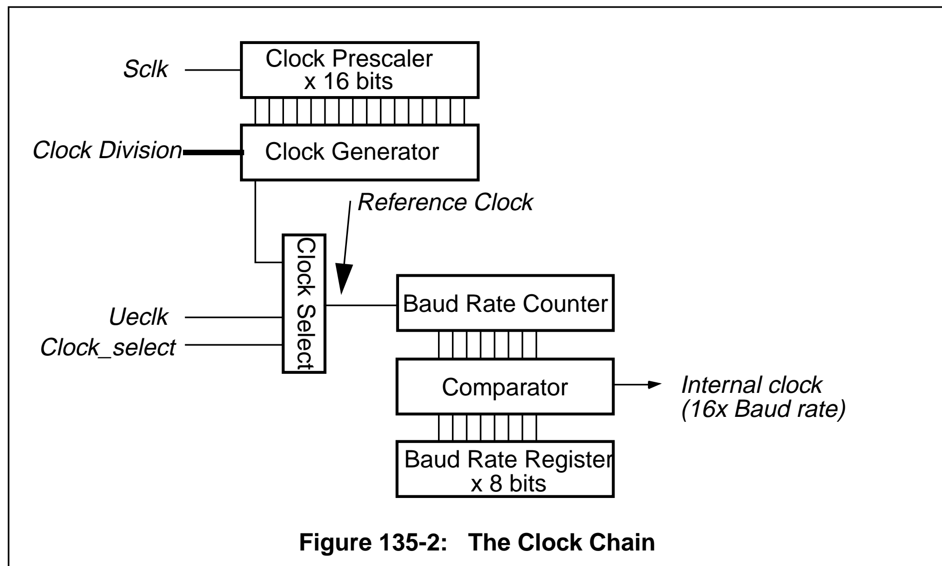


Figure 135-2: The Clock Chain

135.2.2 Transmit Channel

The Transmit channel is clocked by the internal clock referenced in Figure 135-2. At reset, the Transmit Shift Register and the **Transmit Register** are empty, **Transmit Register Empty (SR[1])** is set, and the channel is in Idle state. If **Transmit Interrupt Enable (CR[5])** is subsequently set, a transmit interrupt request is generated.

The following set of conditions must be satisfied for data transmission to take place:

- **Transmit Enable (CR[1])** must be set.
- The **Transmit Register Empty (SR[1])** must be clear. This occurs when a byte of data is written to the **Transmit Register**.

If the above conditions are met, then the following sequence of operations occurs:

- The contents of the **Transmit Register** are transferred into the Transmit Shift Register, a start bit is output from the Transmit Data port, and **Transmit Register Empty (SR[1])** is set, disabling new transfers until fresh data is provided. If **Transmit Interrupt Enable (CR[5])** is set, then an interrupt is generated. **Transmit Active (SR[3])** is also set to indicate that a transfer is taking place.
- The contents of the Transmit Shift Register are then shifted out from the Transmit Data port, low order bit first, at one 16th of the internal clock rate. The value of **Character Length (MR[0])** dictates the number of transmit bits per character; if set, 7 bits are transmitted, whilst if clear, 8 bits are transmitted.
- If **Parity Enable (MR[1])** is set, then a parity bit is inserted into the serial data stream. If **Parity Sense (MR[2])** is set, then Odd Parity is generated; if clear, then Even Parity is generated.
- The number of Stop bits inserted into the data stream is indicated by **Stop Count (MR[3])**. If the bit is clear then 1 Stop bit is inserted; if set then 2 Stop bits are inserted.

Upon completion of the transfer, the transmission condition described above is re-evaluated. If the evaluation is true, then the next character will be transmitted; otherwise the channel returns to the idle state, and **Transmit Active (SR[3])** is cleared.

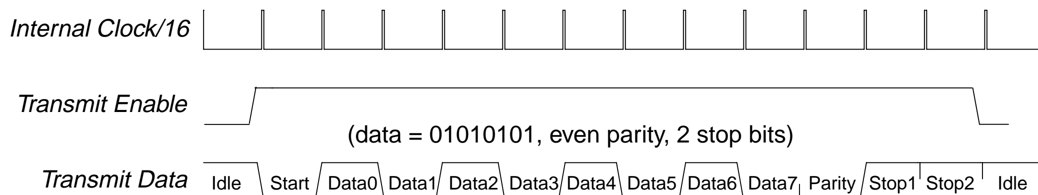


Figure 135-3: Serial Transmission Example

135.2.3 Receive Channel

Data transmission from the Transmit Channel is illustrated in Figure 135-3. The receive channel is clocked at 16x the desired channel baud rate. At reset, the receive shift register and the **Receive Register** are both empty, **Receive Register Full (SR[0])** is clear, and the receive channel is disabled. The channel may be enabled by setting **Receive Enable (CR[0])**. The channel idles until the following conditions are met:

- **Receive Overrun Error (SR[6])** and **Receive Framing Error (SR[5])** are clear, indicating the receive channel is free to receive new data,
- A transition from high to low is detected on the filtered receive data (Urxd) input, which signals the leading edge of a Start bit.

The detection of the leading edge of a Start bit causes the following actions:

- The receive channel translates into the start state, resetting the x16 clock count to zero, and indicating that a Start bit has been detected by setting **Receive Active (SR[2])**.
- On the eighth clock cycle, the channel resamples for the Start bit, in order to reject spurious transitions. If the Start bit is not present, the channel returns to an idle state.
- Each subsequent 16 clocks, the receiver samples the filtered receive data input, and shifts the sampled bit into the receive shift register, lowest significant bit first, until the number of bits indicated by **Character Length (MR[0])** have been assembled. If the character length is 7 bits, then the most significant bit is set to zero.
- The parity of the incoming data is calculated cumulatively with each bit. If **Parity Enable (MR[1])** is set, the result is compared with the sampled incoming Parity bit. If **Parity Sense (MR[2])** is clear, then even parity is expected; if set, then odd parity is expected. If a parity check error is detected, **Parity Error (SR[4])** is set.
- The incoming data stream is sampled for the correct number of stop bits, as indicated by the value of **Stop Count (MR[3])** - if an expected Stop bit is absent, the **Framing Error (SR[5])** flag is set.
- **Receive Register Full (SR[0])** is then checked to determine the status of the receive buffer. If clear, the received data is transferred from the receive shift register to the **Receive Register**, **Receive Register Full (SR[0])** is set, and sampling for a new character begins; If set, **Overrun Error (SR[6])** is set, and the incoming data discarded. The receive channel should then disabled by writing a zero into the **Receive Enable (CR[0])** to avoid further sampling for a new start bit.

135.2.4 Receive Data Filter

To reduce the susceptibility of the UART receive channel to extraneous noise, the input signal is passed through a low-pass digital filter to smooth the received serial data stream. The digital filter is implemented as a 3-bit up/down counter, with no roll-over or roll-under, which is clocked at 16x the desired baud rate. The unfiltered serial input is applied to the up/down control of the counter, whilst the filtered serial output is taken from the most significant bit of the counter. The filter introduces a four clock (pi/4) phase delay between input and output signals, but can reject up to four incorrect samples per received bit. The function of the filter is illustrated below:

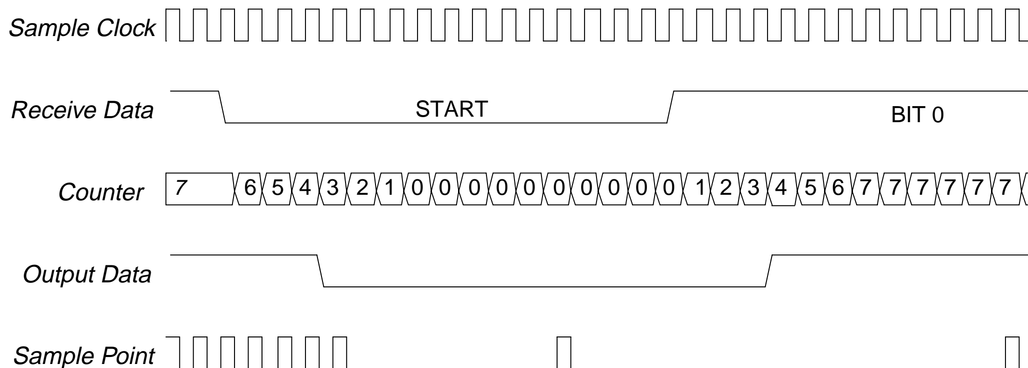


Figure 135-4: Receive data filter action

135.2.5 Data Transfer Methods

A variety of methods are available to move data to and from the UART - the selection of which is dictated by the nature of the transfer in terms of real time constraints, and the software or hardware overhead to support it. When the **Receive Register** is full, or the **Transmit Register** is empty, the condition is indicated by associated status flags - **Receive Register Full (SR[0])** and **Transmit Register Empty (SR[1])** respectively. The implications of each and their derived signals are described below:

Software Polled I/O

The simplest and quickest mechanism available, in which the processor constantly polls the UART **Status Register** to determine whether there is data to be transferred from the UART upon reception (as indicated by **Receive Register Full (SR[0])** = 1) or whether data can be transferred to the UART for transmission (indicated by **Transmit Register Empty (SR[1])** = 1). **Receive Interrupt Enable (CR[4])** and **Transmit Interrupt Enable (CR[5])** should be disabled. This technique is dependent on the data rate not exceeding the worst case polling repetition rate, in which case data may be lost. Alternatively, if the data rate is significantly slower than the polling repetition rate, then unnecessary and hence inefficient polling will result.

Interrupt driven I/O

In this instance, the processor is interrupted from executing its current task to service a transfer request from the UART. **Receive Interrupt Enable (CR[4])** and/or **Transmit Interrupt Enable (CR[5])** should be enabled, as appropriate. When either **Receive Register Full (SR[0])** or **Transmit Register Empty (SR[1])** is set, a corresponding Receive Interrupt or Transmit Interrupt is generated, which is processed by the Interrupt Controller before being applied to the processor. Reading the **Receive Register** or writing to the **Transmit Register** will clear the corresponding status flag, and hence negate the interrupt. Using interrupt driven I/O assures prompt service of the UART channel, but there is some associated latency and software overhead.

Direct Memory Access

Since the interrupt generating condition may be removed by either reading the Receive Register or writing to the Transmit Register as appropriate, the interrupt service routine may be replaced by an operation which mimics the function of the interrupt service routine and transfers data between a memory buffer and the UART, and vice-versa; the internal DMA controller is capable of this function. **Receive Interrupt Enable (CR[4])** or **Transmit Interrupt Enable (CR[5])** should be enabled, as appropriate. When either

Receive Register Full (SR[0]) or **Transmit Register Empty (SR[1])** is set, a corresponding Receive Interrupt or Transmit Interrupt is generated. These interrupts should be disabled in the Interrupt Controller if DMA is to be used, and the appropriate interrupt should be selected as a DMA trigger (see Table 1-6 on page 11).

For reception of data, the DMA source channel must be programmed with the UART Receive register address (as a static address), and the DMA destination channel address should be set to the appropriate static or incrementing-address destination. See Chapter 10 - DMA Controller (DMAC) for detailed information about other set-up requirements.

With the DMA controller suitably programmed and enabled (using the software enable mechanism) data received by the UART will trigger the DMAC automatically. The DMAC will request the Bµild bus, and upon the bus being granted, will transfer the UART data to the required destination. This mechanism incurs no on-going software overhead or CPU activity and is highly efficient. Transmission of data can be handled in a similar manner.

135.2.6 Manual Flow Control

If **Flow Control (CR[3])** is reset, then automatic flow control is disabled, and flow control must be performed in software. Either an in-channel signalling protocol (such as XON/ XOFF) must be used, or the **nUcts** and **nUdcd** signals must be monitored by software polling or interrupts.

The Ready To Send (**nUrts**) signal may be changed under processor control by writing to **Ready To Send (MCR[0])**. If **Modem Status Enable (MCR[2])** is set, then the **Modem Status Register** is updated with the Status and Change indication for each modem input signal. **Clear To Send Status (MSR[0])** and **Clear To Send Change (MSR[4])** are updated from **nUcts**, while **Data Carrier Detect Status (MSR[2])** and **Data Carrier Detect Change (MSR[6])** are updated from **nUdcd**.

Changes in **nUcts** or **nUdcd** will also cause **Modem Signal Change (SR[7])** to be set. If the **Modem Interrupt Enable (CR[7])** is set, then an interrupt will be generated. Reading the **Modem Status Register** automatically clears **Clear To Send Change (MSR[4])** and **Data Carrier Detect Change (MSR[5])**. It also clears **Modem Signal Change (SR[7])**, and consequently the interrupt.

135.2.7 Automatic Flow Control

If **Flow Control (CR[3])** is **set**, then automatic flow control is established using the modem control signals.

The value of **Ready To Send (MCR[0])** is ignored. If **Modem Status Enable (MCR[2])** is set, then the **Modem Status Register** is updated with the Status and Change indication for each modem input signal. **Clear To Send Status (MSR[0])** and **Clear To Send Change (MSR[4])** are updated from **nUcts**, while **Data Carrier Detect Status (MSR[2])** and **Data Carrier Detect Change (MSR[6])** are updated from **nUdcd**, as above.

Changes in **nUcts** or **nUdcd** will NOT cause **Modem Signal Change (SR[7])** to be set. Thus a modem interrupt will not be generated when automatic flow control is in use.

The UART supports two channel configurations - modem configuration and null-modem configuration. The function of **nUrts**, **nUcts** and **nUdcd** differs between configurations as identified below.

135.2.8 Modem Flow Control

If **(MCR[3])** is reset, then the modem configuration is selected. A modem converts digital signals into an appropriate format for transmission over a communications channel. Since the communications channel usually has reduced bandwidth compared to the UART, it is necessary to perform flow control to reduce the UART data rate to that which can be sustained by the channel.

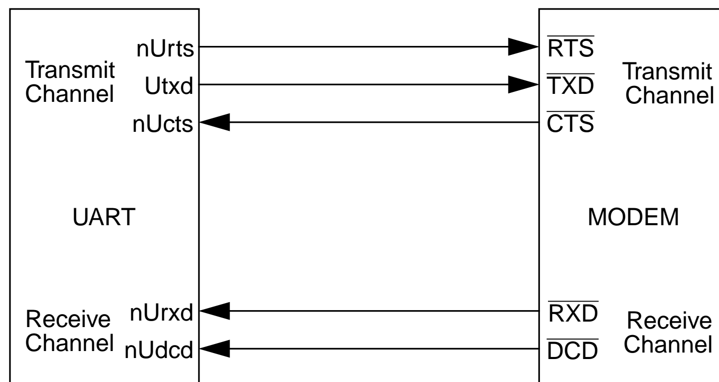


Figure 135-5: Configuration 0 - Modem Flow Control

When data is written to the **Transmit Register, nUrts** is asserted to indicate that there is data for transmission. It reflects the inverted state of **Transmit Register Empty (SR[1])**. If the **nUcts** input is asserted then the Transmit Channel is enabled, and data transmission takes place until the **Transmit Register** is empty, or **nUcts** is negated. **nUdcd** is used to indicate to the Receive Channel that there is a data carrier present, implying valid data is present which may be sampled by the Receive Channel.

135.2.9 Null Modem Flow Control

If **Configuration (MCR[3])** is **set**, then the **null-modem** configuration is selected. In the case of null-modem flow control, the bandwidth of data transfer is limited by the ability of the receiving device to accept new data or by the line characteristics:

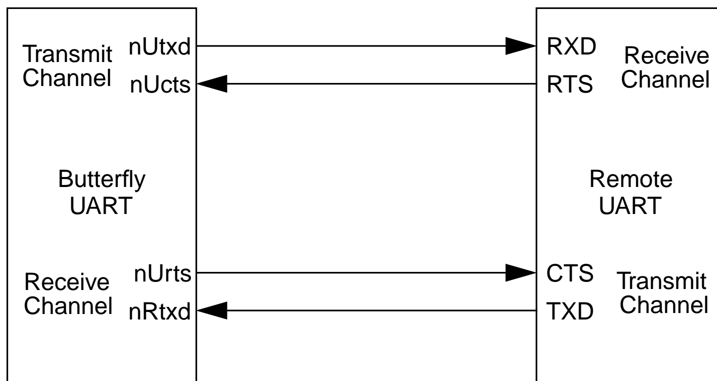


Figure 135-6: Configuration 1 - Null modem Flow Control

The Receive Channel indicates that it is ready to receive new data by asserting the Ready To Send (**nUrts**) output, which is the inverse of **Receive Buffer Full (SR[0])**. If the **nUcts** input is negated, then the Transmit Channel remains Idle, and no data transmission takes place.

135.3 Programmer's Model

Addresses are specified as offsets from the base address as defined for the system. All accesses must be 8-bit. Attempts of any other access width (e.g.: 32 bit wide) will cause a Data Abort exception.

Address Offset	R/W	Name	Abbreviation
+0x000	RW	Serial Control Register	CR
+ 0x004	RW	Serial Mode Register	MR
+ 0x008	RW	Serial Baud Rate Register	BRR
+ 0x00C	R	Serial Status Register	SR
+ 0x010	RW	Transmit/Receive Register	TR / RR
+ 0x014	RW	Modem Control Register	MCR
+ 0x018	-	Reserved	
+ 0x01C	R	Modem Status Register	MSR

Table 135-2: Program Register View

135.3.1 Register Descriptions

Serial Control Register (CR) and Serial Mode Register (MR)

The **CR** defines the current operation of the transmit and receive channels, and the interaction with the processor via the interrupt signals.

Bit	Function	Value	Reset Value
7	Modem Interrupt Enable	[0] disabled, [1] enabled	0
6	Error Interrupt Enable	[0] disabled, [1] enabled	0
5	Transmit Interrupt Enable	[0] disabled, [1] enabled	0
4	Receive Interrupt Enable	[0] disabled, [1] enabled	0
3	Flow Control	[0] software flow control, [1] hardware flow control	0
2	Clock Select	[0] Internal, [1] external	0
1	Transmit Enable	[0] channel disabled, [1] channel enabled	0
0	Receive Enable	[0] channel disabled, [1] channel enabled	0

Table 135-3: Serial Control Register (CR) - Read / Write, + 0x000

When an external clock is selected, the interface should be run at the desired baud rate and not at x16 of the rate. Performance characteristics of the external Clock input can be found in the Butterfly Microcontroller Performance Supplement: SP4578-1.0

The **Serial Mode Register (MR)** Defines the mode of operation of the transmit and receive channels, with the format of the serial bit stream and clock information. **Clock Select** is used to choose between the internal clock or an external clock, whilst **Division Select** determines which of 16 clocks from a division chain should be applied to the internal clock.

Bit	Function	Value	Reset Value
7-4	Division Select	[0000] Divide by 1... [1111] Divide by 32768	0
3	Stop Count	[0] 1 stop bit, [1] 2 stop bits	0
2	Parity Sense	[0] even, [1] odd	0
1	Parity Enable	[0] No parity, [1] Parity enabled	0
0	Character Length	[0] 8 bit data, [1] 7 bit data	0

Table 135-4: Serial Mode Register (MR) - Read / Write, + 0x004

Baud Rate Register (BRR)

Allows generation of baud rates to within a small tolerance (described in Section 135.2.1) of desired rate.

Bit	Function	Reset Value
7-0	Scaling factor for internal serial clock: BRR(0) -> 1, BRR(n) -> n+1, BRR(255) -> 256	0x0

Table 135-5: Baud Rate Register (BRR) - Read / Write, + 0x008

Serial Status Register (SR)

Reflects the current status of the transmit and receive channels, and receive errors. Interacts with interrupt enable bits in the **serial control register** (**CR**) to generate interrupts on various conditions.

Bit	Function	Value	Reset Value
7	Modem signal change	[0] no change, [1] change detected	0
6	Overrun Error	[0] no error, [1] error detected	0
5	Framing Error	[0] no error, [1] error detected	0
4	Parity Error	[0] no error, [1] error detected	0
3	Transmit Active	[0] idle, [1] active	
2	Receive Active	[0] idle, [1] active	0
1	Transmit Register Empty	[0] full, [1] empty	1
0	Receive Register Full	[0] empty, [1] full	0

Table 135-6: Serial Status Register (SR) - Read, + 0x00C

Transmit Register (TR)

This register holds data to be transmitted. Upon the start of a transmit cycle, the contents are transferred to the transmit shift register and **Transmit Register Empty (SR[1])** is set to indicate that new data may be written, upon which the flag is cleared. The L.S.B is transmitted first.

Bit	Function
7-0	Data to be transmitted from serial transmit port

Table 135-7: Transmit Register (TR) - Write,+ 0x010

Receive Register (RR)

A read-only register which holds data received from the serial receive port; when loaded at the end of a receive cycle, the receiver is disabled and **Receive Register Full (SR[0])** is set to indicate that there is new data. This flag is cleared when data is read. The M.S.B will contain the most recently received Bit.

Bit	Function
7-0	Data received from serial receive port

Table 135-8: Receive Register (RR) - Read, + 0x010

Modem Control Register (MCR)

Defines function of modem control and status signals and their interaction with the transmit and receive channels for hardware handshake mechanisms.

Bit	Function	Value	Reset Value
7-4	Reserved	Write as [0000]; returns [0000] when read	0
3	Configuration - controls nUrts if **Flow Control (CR[3])** is set	[0] Modem configuration [1] Null-modem configuration	0
2	Modem Status Enable - when enabled, **Modem Status Register** values are updated	[0] disabled, [1] enabled.	0
1	Reserved	Write as [0]; returns [0] when read	0
0	Ready To Send (nUrts) - ignored if **Flow Control (CR[3])** is set	[0] clear, [1] set	0

Table 135-9: Modem Control Register (MCR) - Read / Write,+0x 014

Modem Status Register (MSR)

If **Modem Status Enable (MCR[2])** is set, then this register reflects the current status and past changes on the modem control signal inputs. If **Modem Interrupt Enable (CR[7])** is set, then interrupts are generated on changes of these signals.

Bit	Function	Value	Reset Value
7	Reserved	Returns [0] when read	0
6	Data Carrier Detect (nUdcd) Change	[0] no change, [1] change detected	0
5	Reserved	Returns [0] when read	0
4	Clear To Send (nUcts) Change	[0] no change, [1] change detected	0
3	Reserved	Returns [0] when read	0
2	Data Carrier Detect (nUdcd) status	[1] No carrier, [0] Carrier	0
1	Reserved	Returns [0] when read	0
0	Clear To Send (nUcts) Status	[1] Not Ready, [0]Ready	0

Table 135-10: Modem Status Register - Read, + 0x01C

Chapter 9 - Interrupt Controller (INTC)

9.1 Introduction

The ARM processor has two interrupt inputs named FIQ (Fast Interrupt Request) and IRQ (Interrupt Request). The FIQ channel has a higher priority and a greater number of dedicated banked registers than the IRQ channel. Both inputs accept asynchronous transitions which are delayed by one cycle for synchronisation before there is any effect on processor execution flow. Refer to ARM7 Microprocessor *Exceptions* for more details on how the processor handles interrupts.

Many applications require a greater number of interrupt channels than the two provided by the ARM processor, therefore an Interrupt Controller is required to interface between multiple interrupt sources and the ARM processor.

9.1.1 Design Features

- Independently controlled interrupt channels
- Hardware priority encoding is provided for FIQ interrupts to indicate the highest priority active FIQ channel

Each channel provides the following functions, all under software control:

- generation of either a FIQ or IRQ request
- independent masking of the channel interrupt source
- status bits to indicate the state of the channel both before and after masking
- responds to edge triggered or level sensitive sources
- Programmable for active low or active high interrupts
- A FIQ interrupt can be downgraded to an IRQ interrupt
- An IRQ interrupt can be upgraded to a FIQ interrupt

9.2 Architecture

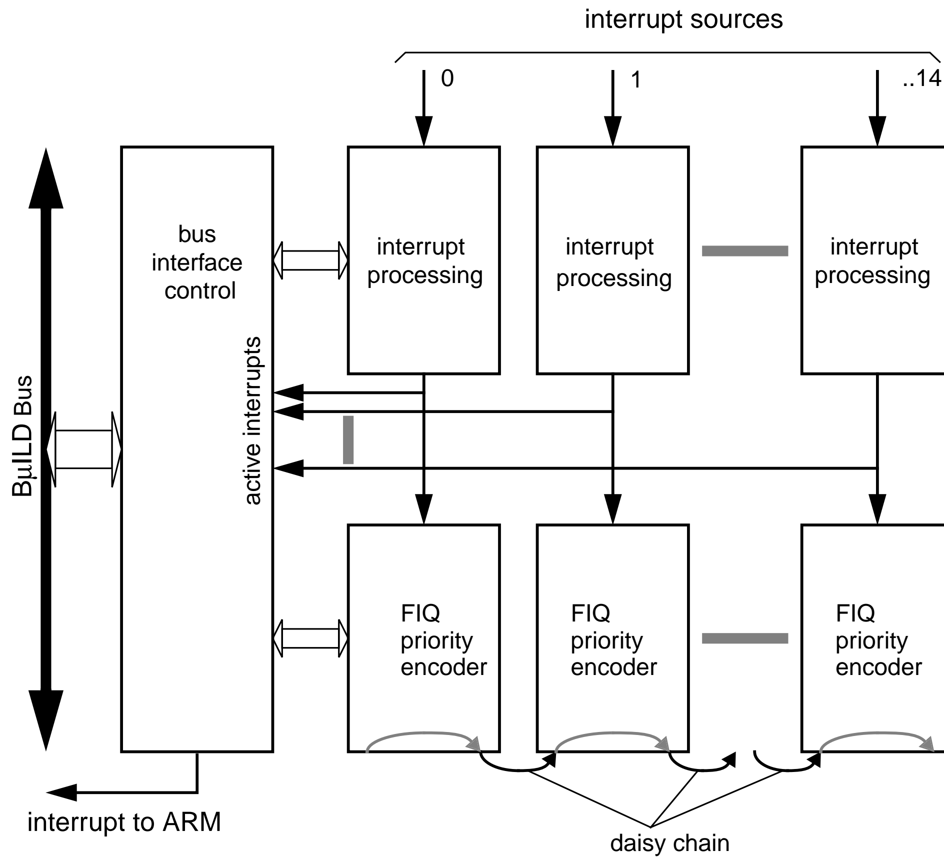


Figure 9-1: Interrupt Controller

Note: Interrupt sources can be from internal modules or from external interrupt pins.

9.3 Operational Description

9.3.1 Interrupt Controller Structure

The interrupt controller consists of identical, independent channels, with each channel capable of handling one interrupt source. The channel is split into two parts, the interrupt processor and the FIQ priority encoder. The channels are supported by the bus interface, which ties all the channels into a common register view. Each channel is represented in the control/status registers by a single bit. Channel 0 is represented by bit 0, channel 1 by bit 1 and so on.

On Butterfly, 15 interrupt channels are provided. Interrupt Channel allocation is detailed in Table 1-7 on page 1-12

9.3.2 Interrupt Processor

This is a block which allows the interrupt source to be processed in a number of ways. Each of these options is controlled by a single bit of a register corresponding to that particular interrupt channel. The options available are:

- programmable polarity of interrupt source [**Polarity Register**]
- edge or level sensitivity [**Edge/Level Triggered Register**]
- enabling/disabling of interrupt [**Enable Register**]
- reset of edge sensitive inputs (level sensitive inputs must be reset at the source device) [**Reset Edge Triggers**]
- programmable type assignment, i.e. FIQ or IRQ interrupt to ARM. [**Type Register**]

Further read-only registers are provided to read the interrupt status at several points in the processing chain. This provides significant flexibility for real-time programming control.

The points at which the signal can be read are:

- at input (the raw interrupt source signal) visible via the **Interrupt Sources** register
- After polarity selection but before the **Enable Register** via the **Pre-enable Sources** register. (n.b after polarity selection, all signals are active high)
- After the **Enable Register** via the **Post-enable Sources** register. n.b. only unmasked interrupts will be visible at this point
- After type selection - via **FIQ Active Sources** and **IRQ Active Sources** registers

All channels are capable of handling asynchronous inputs. The synchronisation is handled within the ARM processor itself, and there is no restriction on the timing of inputs to the interrupt controller.

9.3.3 FIQ Priority Encoder

This module can be used to shorten FIQ interrupt latency times. If not used, all priority sorting must be carried out in software. The cost of shortening the FIQ interrupt latency using the priority encoder is that the relative priorities of all FIQ interrupts are fixed, since the encoder uses a daisy-chain approach. This puts each interrupt channel in a chain with the highest priority sources at the top (see Table 1-7 on page 12).

The control scheme associated with this is as follows: on receipt of an interrupt, the processor vectors to the interrupt vector address (also known as the exception vector address). When the interrupt controller sees this address on the bus, it activates a lock signal. This latches the state of all the FIQ interrupt lines. An enable signal is propagated down the chain from highest to lowest priority, ensuring that if a particular channel is active, all lower priority channels are disabled.

Each channel produces a signal that indicates if the channel is both enabled and active. When the daisy chain is stable, only one channel will be setting this signal. The signals are encoded into a five bit number to indicate the highest priority active FIQ channel.

This number can be read from the **FIQ Encoded Priority Register** (Figure 9-2). It is placed in bit positions 2 to 6 to allow it to be used directly for indexing into a branch table. An example code sequence showing how this can be done is found in Section 9.5.

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	x x x x x	0 0

highest priority active FIQ channel

Figure 9-2: FIQ Encoded Priority Register

The lock is released by the act of reading the priority register. This allows new interrupt requests to propagate through the daisy chain.

9.3.4 Bus Interface Control

The bus interface handles all the register selects and control functions of the interrupt controller. Seen from the Bµild bus, the interrupt controller is a collection of memory-mapped 32-bit registers. The address map is given in Table 9-1. The FIQ encoded priority value can be read from address offset 0x028. The five remaining read-only registers shown describe the status of each channel.

Offsets 0x02C to 0xFFC are blank, with no registers allocated to this area. The interrupt controller will cause a bus error if the system attempts to access these locations. It will also cause a bus error if an attempt is made to write to the read-only registers. The “reset edge triggers” at offset 0x010 is a transient function, i.e. it resets the edge triggering but does not hold it reset, leaving it ready to receive further interrupts. The location does not

store a value. A bus error occurs if a read is attempted to this location. All bus accesses to the Interrupt controller are handled in a single cycle.

9.3.5 External Interface

External interrupt pins (**Iextint1** and **Iextint2**) are provided that can feed directly into the interrupt controller. In addition to the dedicated pins. any change on the external pins of the PPI can be used to initiate an interrupt. See chapter on the Programmable Peripheral Interface (PPI) for more information.

9.4 Programmer's Model

Address Offset	Function	Type	Interpretation	Reset Value
0x000	Interrupt Sources	R		
0x004	Polarity Register	RW	1 = active	Undefined
0x008	Pre-enable Sources	R	1 = active	Undefined
0x00C	Edge/Level triggered	RW	1 = edge, 0 = level	Undefined
0x010	Reset Edge Triggers	W	1 = reset	Undefined
0x014	Enable Register	RW	1 = enable	Undefined
0x018	Post-enable Sources	R	1 = active	Undefined
0x01C	Type Register	RW	1 = FIQ, 0 = IRQ	Undefined
0x020	FIQ Active Sources	R	1 = active	Undefined
0x024	IRQ Active Sources	R	1 = active	Undefined
0x028	FIQ Encoded Priority	R	active bits 2-6	Undefined
0x02C - 0xFFC	Reserved	-		Undefined

Table 9-1: Register Map

A detailed description of the functions and the bit significance is given in Section 9.3

9.5 Using the Interrupt Controller

The ARM processor has a simple mechanism for interrupt priority handling. When an IRQ interrupt is received, it sets a mask bit in the status register (see section 2.3.3 on page 7) to prevent any further IRQ interrupts being serviced. However, the ARM may service a FIQ interrupt during this time. When n FIQ interrupt is received , it sets mask bits in the status register to prevent any further IRQ or FIQ interrupts being serviced. On the Butterfly microcontroller this capability is enhanved further with the addition of an on-chip Interrupt controller.

In a system with multiple interrupt sources, it is usually required that a higher priority interrupt will be able to interrupt a lower priority service routine. If both are either FIQ or IRQ interrupts, then the interrupt service routines must be written as re-entrant code. The writing of re-entrant code is beyond the scope of this manual, but is fully explained in the Mitel ARM software toolkit manuals.

Two simple examples are given below, of using the Butterfly interrupt controller in conjunction with the ARM processor interrupt mechanism. The first is written in assembly code and uses the **FIQ Encoded Priority Register** to determine the channel of an FIQ interrupt, while the second is written in C and uses the **IRQ Active Source Register** to determine the interrupting channel and branch to the appropriate service routine.

Example 1

```
fiq_service   LDR R9,[R8,#0x28]     ; read the priority register
              ADD PC,PC,R9          ; index into branch table
              NOP                   ; take account of pipeline
              B fiq_isr0            ; branch table start
              B fiq_isr1
              B fiq_isr2
                  |
                  |
              B fiq_isrn
```

The assembly code example above illustrates the use of the **FIQ Encoded Priority Register** and a means of achieving extremely low interrupt latency. To carry out any form of processing, it would more common to save some of the register values, and restore them afterwards. This can be implemented within the individual functions that are invoked, thus allowing a high priority, but simple, function to execute in minimum time.

The **fiq_service** function may be located at address 0x1C, the FIQ interrupt vector address, so that a branch instruction to this function (fiq_service) is not required.

In this example, to improve latency. the base address of the interrupt controller has already been loaded into register 8 of the FIQ banked ARM registers during initialisation.

Example 2

```
static void __irq irq_service(void)
{
        unsigned long irq_active;

        irq_active = *((unsigned long*)INTBASE + 0x24);
        if (irq_active & 0x8) irq_priority_0();
        else if (irq_active & 0x80) irq_priority_1();
        else if (irq_active & 0x800) irq_priority_2();
                                     |
                                     |
        else if (irq_active & 0x40) irq_priority_n();
}
```

The algorithm above is a very simple reorganisation of the fixed priorities of the hardware. Further complexity could easily be implemented, however more complexity is likely to affect interrupt latency.

Another common problem is when one interrupt channel may normally need to be low priority, but under certain circumstances, may want to promote itself to complete a specific task. This may be accomplished by the routine writing to the **Type Register** and changing the Type of its channel from an IRQ to an FIQ. As there is still an active interrupt, a FIQ interrupt is immediately generated, thus promoting that task. In the case of Edge Triggered interrupts, this must be carried out before the edge trigger is reset by writing to the **Reset Edge Triggers** location.

It is also possible for a FIQ to downgrade to an IRQ interrupt in a similar way, however, the IRQ will still be locked out until the FIQ has finished, when it will then be processed in the normal way.

Chapter 10 - DMA Controller (DMAC)

10.1 Overview

Data transfer between memory blocks, or between memory and a peripheral, can be extremely cycle intensive for a processor; the ARM processor, for example, requires at least nine clock cycles to move a word of data from one address in memory to another. The Butterfly Microcontroller contains a DMA controller, which assists the processor to move large blocks of data around a system.

The DMA system is initialised by the processor with a source and a destination for the data transfer. On receipt of a DMA request, the DMA Controller acquires control of the system address and data buses and proceeds to transfer data until a stop condition is met. The DMA Controller may then be auto-initialised or manually reprogrammed as desired.

- Maximum transfer rates of 100 MBytes per second (single addressing), 50 MBytes per second (dual addressing) at 25 MHz, zero wait-state transactions
- 32 bit (4 GByte) addressing range, address increment, decrement and hold
- Data transfer sizes of 8, 16, and 32 bits (statically sized)
- 16 bit (65536 item) maximum transfer count
- Single addressed (fly-by) transfer performed by individual channels
- Dual addressed (memory-memory and peripheral-peripheral) transfers supported by linked channel pairs
- Transfers can be triggered by software
- Maskable level or edge sensitive hardware transfer triggers with selectable polarity
- Maskable hardware transfer acknowledge signals with selectable polarity
- Block and Packet mode transfers supported
- Wait state insertion when indicated by slow memory
- Auto-initialisation of channels on completion
- Chained DMA transfers supported for scatter-gather operations
- Selectable interrupt generation on completion
- Fixed channel priority
- Optional bus locking to prevent interruption of DMA services by other bus masters
- Programmable DMA triggers from internal and external sources
- Abort mechanism for illegal access with interrupt generation and transfer halt

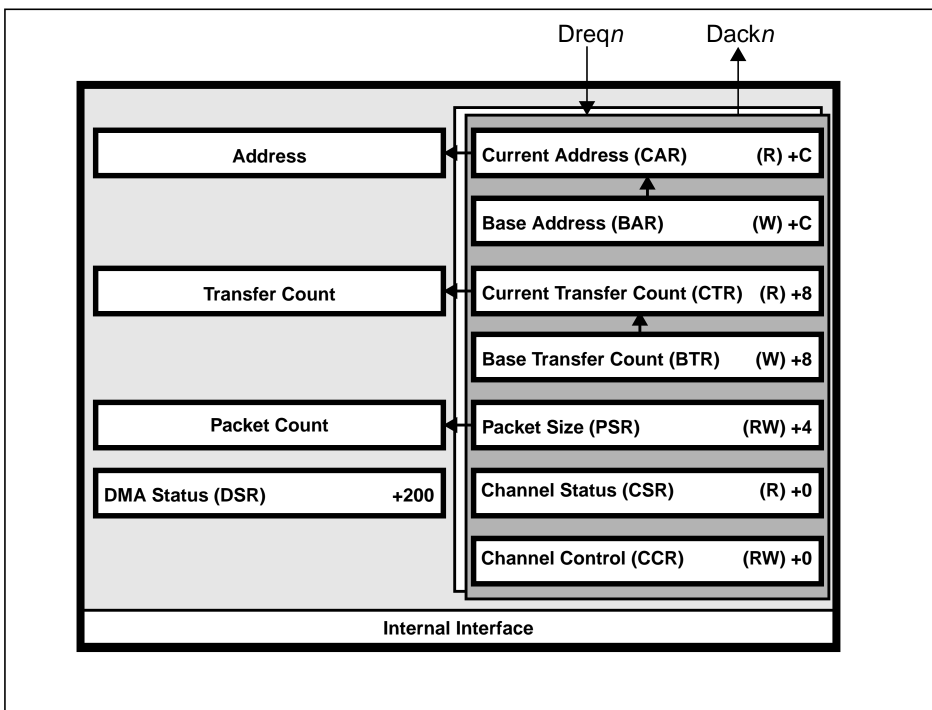


Figure 10-1: Interface Diagram

Signal		Description
Dreq*n*	Input Programmable active high/low	DMA service request for channel *n*.
Dack*n*	Output Programmable active high/low	DMA service acknowledge for channel *n* - implicitly addresses a peripheral

Table 10-1: DMA Controller Signals

10.1.1 DMA Controller Trigger Selection

Each channel DMA request (Dreq) may be connected to one of several sources by programming the appropriate value into the Channel Select bits of the System Configuration Register, which is described in Chapter 1.

10.2 Operational Description

10.2.1 Single Addressed (Fly-by) Transfer

DMA transfers are one-to-one transactions, with a data source and destination. High speed data transfer applications require use of the full bandwidth of the internal data bus, which corresponds to one data transaction per bus cycle. Since only a single address may be explicitly associated with the data (and presented on the address bus as either source or destination), the corresponding address must be implicitly signalled. This is performed using a hardware handshake protocol, which allows the DMA controller to directly access the implicitly addressed device.

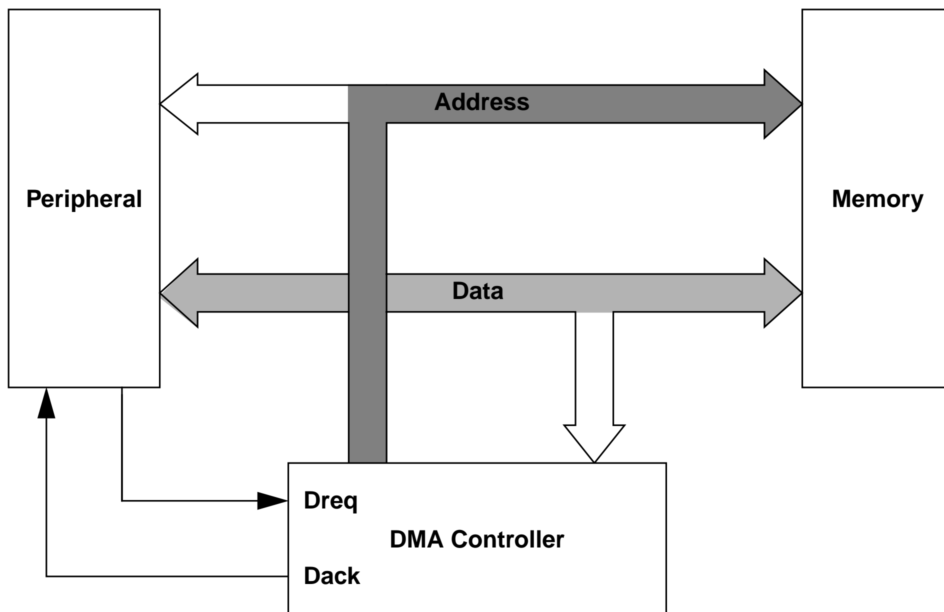


Figure 10-2: Architecture of Single-Addressed DMA

Each DMA channel is capable of generating an address and a hardware acknowledge signal for a particular data transaction. Data is presented to the bus by the source and written to the destination during a single bus transaction; it is **not** buffered by the DMA controller. An address is broadcast onto the bus simultaneously. This transfer mode is referred to as a single addressed or Fly-by transaction, and **Transfer Mode (CCR[9])** must be *clear*. If the address is the source of the transfer, then **Transfer Direction (CCR[8])** should be *clear*; if the address is the destination, then the bit should be *set*.

Transfers are usually triggered by hardware transfer requests:

Block Transfer

If the target device can process data with the same bandwidth as the bus then the data may be transferred in consecutive bus cycles; this is referred to as a Block Transfer, and **Transfer Type (CCR[10])** must be *set*. The function of the hardware request signal is determined by the value of **Request Trigger Type (CCR[11])**.

Edge Triggered Block Transfer

If **Request Trigger Type (CCR[11])** is *clear*, the channel becomes edge sensitive, and monitors for the assertion of the request input. Once triggered, the channel will ignore subsequent transitions on the hardware request input until channel activity is completed. The input must be negated before the channel can be retriggered.

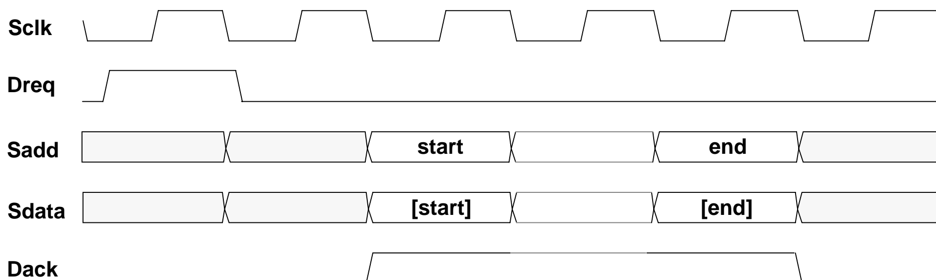


Figure 10-3: Edge Triggered Block Transfer

Level Triggered Block Transfer

If **Request Trigger Type (CCR[11])** is *set*, then the channel continues to test the hardware request input throughout a transaction; if the input is negated, the channel activity is halted on completion of the current bus cycle and the channel returns to the Idle state. If no other channel is requesting DMA activity, then at least one null bus cycle will be inserted.

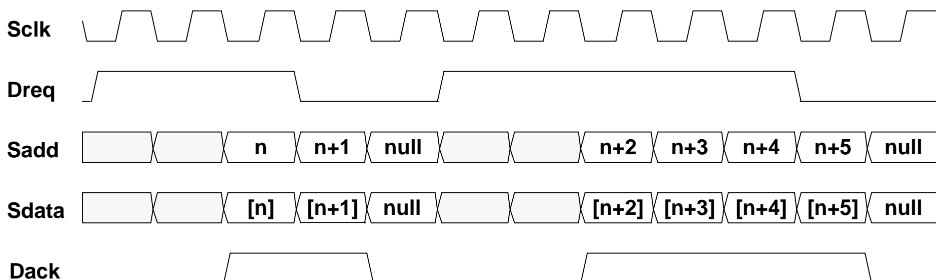


Figure 10-4: Level Triggered Block Transfer

Edge Triggered Packet Transfer

However, some devices of reduced bandwidth and with limited internal buffering require the interruption of DMA service whilst previously transferred data is processed. Transfer is recommenced when the transfer request is retriggered. This transfer type is referred to as Packet Transfer, and **Transfer Type (CCR[10])** must be *clear*. The size of the buffer to be transferred must be programmed into the channel **Packet Size Register (PSR)**.

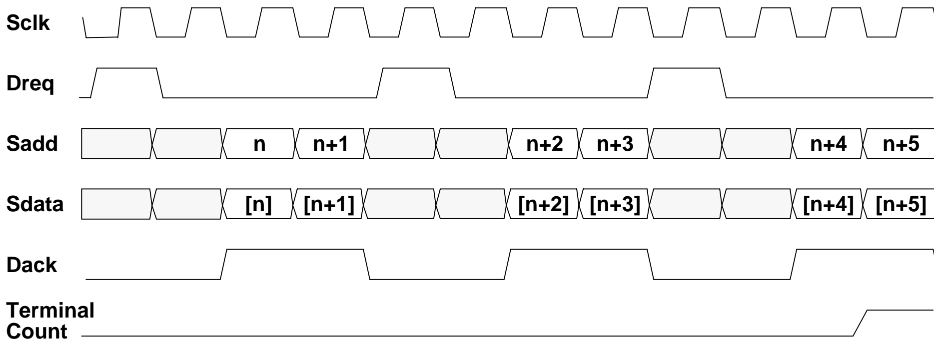


Figure 10-5: Edge Triggered Packet Transfer (Size = 2)

Note that if Packet Transfer and Level Sensitive input are selected and the request is negated BEFORE the number of transfers completed equals the number set in the **Packet Size Register**, then the remaining words to transfer will be discarded by the DMA controller; retriggering will cause the DMA Controller to reload the packet size from the channel **Packet Size Register**. It is recommended that the input be Edge Sensitive.

Software Triggered Transfer

Software requests are issued when **Software Request (CCR[2])** is *set*. Software requests are usually terminated by channel completion, but may also be terminated in software by *clear*ing **Software Request (CCR[2])**.

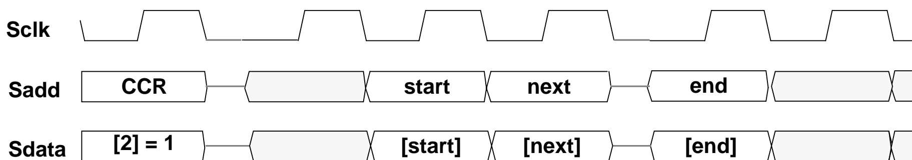


Figure 10-6: Software Triggered Block Transfer

10.2.2 Dual Addressed (Buffered) Transfer

When data must be transferred from one memory block to another, or a peripheral device does not provide support for implicit addressing through a hardware handshake, then an explicit address is required for both source and destination. Each DMA transfer then requires two bus transactions; the first to read the data from the source, and the second to write the data to the destination. An explicit address is generated for each transaction, and the data must be buffered internally by the DMA Controller between read and write.

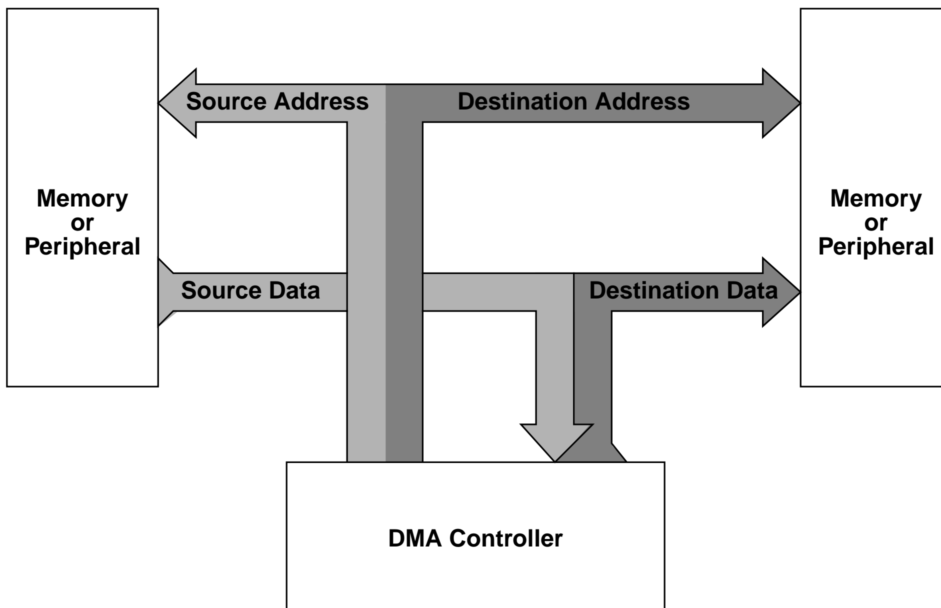


Figure 10-7: Architecture of Dual Addressed DMA

The number of source data read transactions which may be performed before the buffered data must be written to the destination is determined by the internal buffer depth. In this DMA controller, the buffer depth is one word.

If dual addressed transfers are required, then two adjacent channels must be selected in order to provide the source and destination addresses. The higher priority channel (lower channel number) becomes the Read Channel, and **Transfer Direction (CCR bit [8])** must be *clear*. The lower priority channel of the pair becomes the Write Channel, and **Transfer Direction (CCR[8])** must be *set*. **Transfer Mode (CCR[9])** should be *set* for both channels.

Block Transfer

Some devices, such as memory, can process data at the full bus rate; then the data may be transferred in consecutive bus cycles. This is referred to as a Block Transfer, and **Transfer Type (CCR[10])** must be *set*. The function of the hardware request signal is determined by the value of **Request Trigger Type (CCR[11])**.

Edge Triggered Block Transfer

If **Request Trigger Type (CCR[11])** is *clear*, the channel becomes edge sensitive, and monitors for the assertion of the request input of the Read Channel. Once triggered, the channel will ignore subsequent transitions on the hardware request input until activity is completed on both Read and Write Channels. The input must return to the negated level before the channel can be retriggered.

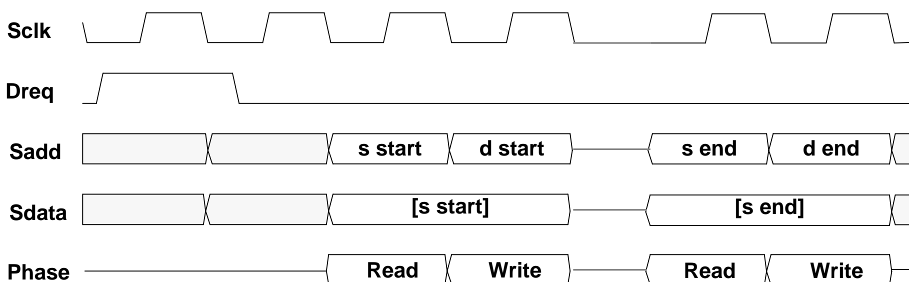


Figure 10-8: Edge Triggered Block Transfer

Level Triggered Block Transfer

If **Request Trigger Type (CCR[11])** is *set*, then the channel continues to test the hardware request input throughout a transaction; if the input is then negated, channel activity is halted after the remaining buffered data has been written to the destination, and the channel returns to the Idle state.

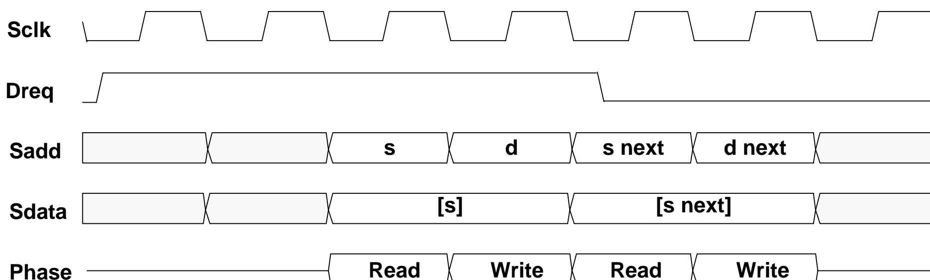


Figure 10-9: Level Triggered Block Transfer

If the request is negated during the Write phase of the DMA transfer with no other channel requesting DMA activity, then at least one empty bus cycle will be inserted before the bus is granted to another bus master.

Edge Triggered Packet Transfer

However, some devices of reduced bandwidth and with limited internal buffering, such as UARTs, require the interruption of DMA service whilst previously transferred data is processed. Transfer is recommenced when the transfer request is retriggered. This transfer type is referred to as Packet Transfer, and **Transfer Type (CCR[10])** must be *clear*. The size of the buffer to be transferred is dictated by the internal buffer depth, and the **Packet Size Register** will be ignored.

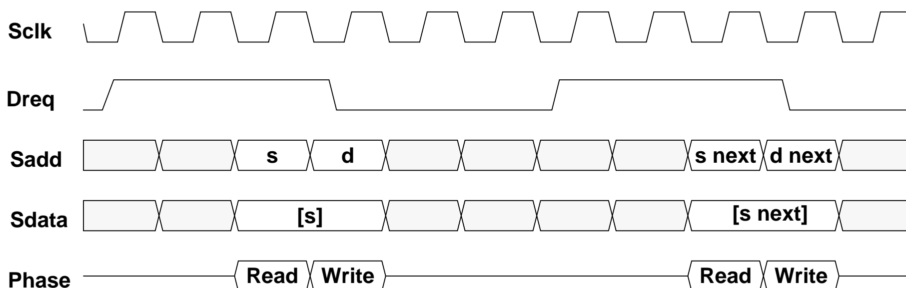


Figure 10-10: Edge Triggered Packet Transfer

Software Transfer Requests

Software requests are issued when **Software Request (CCR[2])** is *set*. Software requests are terminated by channel completion, and may also be terminated by *clear*ing **Software Request (CCR[2])**. If **Transfer Type (CCR[10])** is *set*, then the DMA controller will transfer the entire block of data in consecutive read & write cycles.

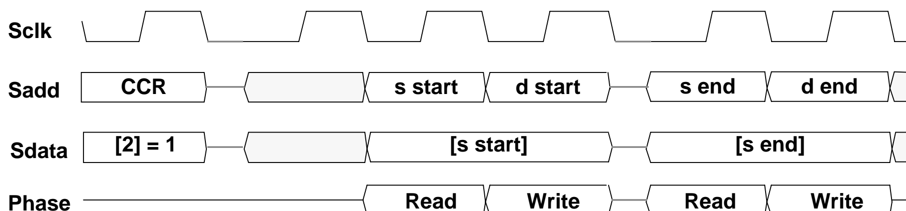


Figure 10-11: Software Triggered Block Transfer

It should be noted that the use of Block Transfer starves the processor of bandwidth. If the user wishes to guarantee some bus bandwidth to the processor, then **Transfer Type (CCR[10])** must be *clear*, indicating Packet Transfers. This guarantees at least one idle bus cycle between each DMA transfer.

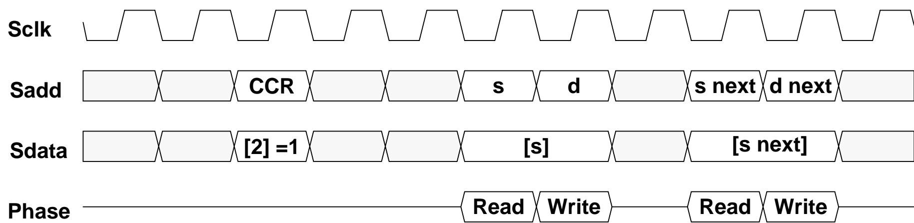


Figure 10-12: Software Triggered Packet Transfer

The Read Channel should be programmed with appropriate values for the **Base Transfer Count Register**, and the **Base Address Register**. Only the **Base Address Register** need be programmed for the Write Channel; for Dual Addressed transfers, the **Packet Size Register** of both channels and the **Base Transfer Count Register** and **Current Transfer Count Register** of the Write channel have no effect on the operation.

10.2.3 Configuration

Transfer Request (Dreq) and Transfer Acknowledge (Dack)

Hardware requests are enabled when **Hardware Request Mask (CCR[1])** is *set* (See Figure 10-4).The trigger polarity is programmed either high or low by **Hardware Request Polarity (CCR[3])**. DMA request sources are detailed in Table 10-4.

DMA transfers are acknowledged by assertion of the Dack output signal. The signal is active high when **Hardware Acknowledge Polarity (CCR[4])** is *clear*, or low when *set*. Single Addressed transfers require the assertion of a hardware acknowledge, and consequently **Hardware Acknowledge Enable(CCR[5])** must be *set*. For Dual Addressed transfers, a hardware acknowledge is not usually necessary, and is masked out when the appropriate bit is *clear*.

Address Calculation

The address calculation performed by the DMAC is determined by the values of **Address Increment (CCR[7])** and **Fixed Address (CCR[6])**.

- If **Fixed Address (CCR[6])** is *set*, then the address will be held for each data transaction, as is required when accessing a FIFO, for example.
- If **Fixed Address (CCR[6])** is *clear*, then the address will be incremented if **Address Increment (CCR[7])** is *set*, otherwise it will be decremented.
- The address offset is dependent on the Value of **Operand Size (CCR[13:12])**, with offsets of 1, 2, and 4 for byte, half-word, and word transfers respectively.

The DMA controller does not directly support byte packing and unpacking, so **Size (CCR[13:12])** must have the same value in each channel. However, the Memory / Peripheral Controller is capable of packing and unpacking operands as appropriate. See Chapter 7 - Memory/Peripheral Controller (MPC) for details.

Re-initialisation

When a DMA transfer is complete, the DMA supports several options for re-initialisation:

- No initialisation - channel returns to Program state. **Auto Initialisation (CCR[14])** should be *clear*, and the state of **Chained Transfer (CCR[15])** will be ignored.
- Auto-initialisation - channel re-initialises with values from the previous transfer as stored in the Base Registers, and returns to Idle state. This mechanism supports repeated transfers into and between memory buffers. **Auto Initialisation (CCR[14])** should be *set*, and **Chained Transfer (CCR[15])** should be *clear.*
- Chained Transfer initialisation - the channel re-initialises with values from the Base Registers, which are updated by the supervising processor as a result of an interrupt

service request, see below. Both **Auto Initialisation (CCR[14])** and **Chained Transfer (CCR[15])** should be *set*. Interrupts must be enabled, so **Interrupt Enable (CCR[16])** should also be *set.* This option provides for scatter / gather type operations, as described below.

Chained Transfers

Many applications require support for the scattering and gathering of blocks of data. The DMA controller offers a Chained Transfer mode, which allows multiple blocks of data to be transferred with minimal processor intervention.When a channel programmed for chained transfers is enabled, the current Registers are updated with the Base Register contents, and an internal flag, Last Block, is *set*. **Block Request (CSR[26])** is also *set*, triggering an interrupt if **Interrupt Enable (CCR[16])** is *set*.

Reading the **Channel Status Register** will *clear* **Block Request (CSR[26])**, negating the interrupt. New block values may now be written to the **Base Registers**. The **Base Address Register** (in the case of Dual Addressed transfers, of the Read Channel) should be written last, which will clear the Last Block flag. If this occurs before Terminal Count (or Early Termination) is reached, then the chain is established, and the new **Base Register** values are transferred to the **Current Registers** upon completion of the current block. The first transfer of the new block will then proceed when appropriately triggered. Since the new register values are loaded automatically, there is no requirement to re-enable the channel - **Channel Enable (CCR[0])** will be *set* from the previous block transfer.

If the new **Base Register** values are not written (the Last Block internal flag remains set), and the channel reaches Terminal Count or Early Termination occurs, then the chain is broken, and the channel returns to the Program state, and indicates that the chain has broken with **Chain End (CSR[27])** *set* and **Channel Enable (CCR[0])** *clear*. If new block parameters are programmed, **Channel Enable (CCR[0])** must be *set* to re-enable the channel.

If the service routine does not read the **Channel Status Register**, **Block Request (CSR[26])** is also cleared after the **Base Address Register** is written.

Interrupts

Interrupts are enabled for a given channel by **Interrupt Enable (CCR[16])**. This permits the generation of an interrupt under the following conditions:

- No initialisation is programmed and Terminal Count is reached.
- A new transfer block is required, or the block chain is exhausted, when Chained Initialisation is programmed.
- Upon detection of a bus error whilst undertaking a bus transaction.

The actual generating condition can be determined by reading the **DMA Status Register** to identify the DMA channel requesting interrupt service, then by testing the corresponding **Channel Status Register**. Note that this action also *clears* the register (with the exception of the **Alternate Phase (CSR[31])** indicator), thereby negating the interrupt request. In Dual Addressed transfers, **Interrupt Enable (CCR[16])** of the write channel should be *clear*, disabling interrupts from that channel.

Channel Priorities and Bus Locking

Channel priorities are hard-wired, with channel 1 having the highest priority. A higher priority channel may pre-empt a lower priority channel, except under one of the following conditions:-

- For Single Addressed transfers, if **Bus Lock (CCR[17])** is *set*. This guarantees the full bus bandwidth to the currently active DMA transfer.
- For Dual Addressed transfers, if the Read or Write phase is currently active. The bus is also automatically locked **between** the Read and Write phases to ensure atomic DMA operations. Dual Addressed transfers may only be pre-empted between the last write of one buffer and the first read of the next, unless **Bus Lock (CCR[17])** is *set*. This guarantees data consistency.

Other higher priority bus masters are also prevented from pre-empting the DMA controller under these conditions.

DMA Devices External to the Microcontroller

As the implicitly addressed DMA targets of Single Addressed transfers may be either internal or external to the microcontroller, it is necessary to indicate to the memory/ peripheral controller the location of the DMA target, so that the bus interface may be switched to pipe data in the correct direction, or to remove bus drive as appropriate.

External Device (CCR[18]) should be *clear* if the target device is internal to the microcontroller, and should be *set* if external. Note that for Dual Addressed transfers, this bit should always be *clear*, as the data buffer for dual-addressed transfers is internal to the DMA controller.

10.3 Programmer's Model

Upon reset, each DMA channel is set to an inactive Program state, from which it may be configured by a supervising processor. Since values written into the channel in any other state may have undesired side effects, **it is essential** that the channel be put into Program state first. **Channel Enable (CCR[0])** must be *clear*; no other register bits should be altered at this time. The programmer may verify that the channel is in Program state by checking that **Channel Enable (CCR[0])** is *clear* by reading the **Control & Status Register**. When the channel is in Program State, there is no channel activity. However, all internal registers may be read and written.

Channel Initialisation

The **Packet Size Register**, **Base Transfer Count Register**, and **Base Address Register** may be modified in Program state; changes will not be immediately reflected in the **Current Transfer Count Register** and **Current Address Register**. Once these registers are configured, the channel may be enabled by setting **Channel Enable (CCR[0])**, at which point the values programmed into the Base Registers are transferred to the Current Registers, and the channel enters Idle state. The settings of the other bits in the **Control & Status Register** will dictate the subsequent operation of the channel.

10.3.1 DMA Registers

The DMA controller is of modular construction. A single DMA channel comprises one DMA single address (Fly-by) engine. The exact behaviour of each channel and any channel pairing is configurable by software. The software programmable internal registers are mapped into the DMA controller address space by channel, with an additional status register. Addresses are specified as offsets from the DMA controller base address.

Address Offsets	DMA Register Base Addresses
+0x000	DMA channel 1 base address
+0x010	DMA channel 2 base address
+0x020... ...+0x1F0	Reserved
+0x200	DMA controller Status Register (DSR)

Table 10-2: DMA Controller Registers

Each DMA channel contains a set of registers which allow the appropriate configuration of the channel in software to perform a variety of DMA operations:

Word Offset	Bit	Reset Value	Type	Register
+0x0	23:0	0x000000	R/W	Channel Control Register (CCR)
	31:24	0x00	R	Channel Status Register (CSR)
+0x4	7:0	0x00	R/W	Packet Size Register (PSR)
+0x8	15:0	Undefined	W	Base Transfer Count Register (BTR)
+0x8	15:0	0x0000	R	Current Transfer Count Register (CTR)
+0xC	31:0	Undefined	W	Base Address Register (BAR)
+0xC	31:0	0x00000000	R	Current Address Register (CAR)

Table 10-3: DMA Controller Channel Registers

The register word offset must be added to the channel address offset and controller base address to form the register address.

The DMA controller also integrates an internal data buffer, which is not visible to an external processor. It is used to hold temporary data between the read and write phases of a memory-memory transfer. On the Butterfly microcontroller, this buffer is one word.

Channel Control Register (CCR)

The **Channel Control Register** is a 24 bit read/write register, occupying bits [23:0]. It defines the configuration of the DMA channels, and their operating behaviour. Upon reset, the register is reset to zero. The register may be both read and written by the supervising processor.

Channel Status Register (CSR)

The **Channel Status Register** is an 8-bit read-only register, occupying bits [31:24]. It indicates the current status and termination conditions for the DMA channel. The register is *cleared* to zero at reset, or when the register is read (with the exception of **Alternate Phase (CSR[31])**). Writing to the status register has no effect on the status values.

Bit	Name	Description
0	Channel Enable	[0] - Channel is in Program state, all DMA operations are disabled, and the clock to the channel circuitry is halted. [1] - Internal current registers are updated, and the channel enters Idle state, ready for operation.
1	Hardware Request Mask	[0] - Activity on the request input is ignored. [1] - Channel responds to activity on the request input in accordance with the setting of the Hardware Request Polarity and Request Trigger Type bits.
2	Software Request	[0] - No activity [1] - Channel responds by performing the requested transfer; upon completion, the bit is *clear*.
3	Hardware Request Polarity	[0] - active high, [1] - active low
4	Hardware Acknowledge Polarity	[0] - active high, [1] - active low
5	Hardware Acknowledge Enable	[0] - Hardware acknowledge signal remains in inactive state as defined by Hardware Acknowledge Polarity [1] - Hardware acknowledge signal is set during bus cycles during which a peripheral is implicitly addressed
6	Fixed Address	[0] - Address is modified on consecutive transfers according to the Address Increment bit [1] - Address is static
7	Address Increment	[0] - Address is decremented, [1] - Address is incremented
8	Transfer Direction	[0] - Data read from memory, [1] - Data write to memory.
9	Transfer Mode	[0] - Memory - peripheral, with peripheral implicitly addressed using hardware handshake mechanism [1] - Memory - memory, with both source and destination explicitly mapped into memory
10	Transfer Type	[0] - Packet transfer, size dictated by Packet Size Register or by depth of internal data buffer [1] - Block transfer, size equal to Transfer Count
11	Request Trigger Type	[0] - Edge trigger; request must return to inactive state before new transfer can be triggered [1] - Level trigger; request must remain in active state for successive transfers

Table 10-4: Channel Control Register (CCR) - offset +0x0

12,13	Operand Size	13,12 : [00] - Byte, [01] - Half Word, [10] - Word, [11] - Reserved, not permitted
14	Auto initialisation	[0] - no auto initialisation [1] - channel is reinitialised from channel base registers when current transfer is complete
15	Chained Transfer	[0] - Reinitialisation from base registers only [1] - Reinitialisation triggers interrupt request
16	Interrupt Enable	[0] - disabled, [1] - enabled
17	Bus Lock	[0] - disabled, may be interrupted by higher priority device [1] - enabled, locks bus for duration of transfer
18	External Device	[0] - Peripheral is internal to controller [1] - Peripheral is external to controller
19-23	Reserved	Not writable, read as zero (0)

Table 10-4: Channel Control Register (CCR) - offset +0x0

24	Terminal Count	[1] Channel word count has decremented to zero, completing the transfer and halting channel operation
25	Reserved	-
26	Block Request	[1] In chained transfer mode, a new transfer block should be programmed into the channel base registers
27	Chain End	[1] In chained transfer mode, a new transfer buffer was not programmed before the current buffer was exhausted
28	Reserved	-
29	Bus Halt	[1] Indicates that an error condition was signalled on b_error or the bus mode signals
30	Reserved	-
31	Alternate Phase	[1] In dual address mode, indicates that read channel is idling whilst awaiting the completion of the write channel (only valid on read channel)

Table 10-5: Channel Status Register (CSR) - offset +0x0

Packet Size Register (PSR)

This register defines the size, in number of data transfers, of device register queues, such as FIFOs, on implicitly addressed peripherals in single addressing mode, thereby avoiding buffer overflow. The **Packet Size Register** should be loaded with the required packet size minus one; thus a value of zero indicates a packet size of one, while a value of 0xFF indicates a packet size of 256. The register is *cleared* at reset, and may be both read and written.

Bit	Name	Description
7:0	Packet Size Register	Size limit of single DMA transfer packet - 1
31:8	Reserved	Not writable; all bits read as 0

Table 10-6: Packet Size Register (PSR) - offset +0x4

Base Transfer Count Register & Current Transfer Count Register (BTR, CTR)

The **Base** and **Current Transfer Count Registers** define the initial transfer block size, and the remaining number of data transfers, respectively. During channel initialisation, the **Base Transfer Count Register** is loaded with the required transfer block size minus one; thus a value of zero indicates one transfer, whilst 0xFFFF indicates 65536 transfers. This value is subsequently written through to the **Current Transfer Count Register** when the channel is enabled. During a DMA transfer cycle on that channel, the transfer count held in the **Current Transfer Count Register** is read, decremented and rewritten to the **Current Transfer Count Register**. If the current transfer count value equals zero, the transfer of data is completed with the current transaction, and the channel halts subsequent transfers. Upon completion of the block transfer, the **Current Transfer Count Register** may optionally be updated from the **Base Transfer Count Register** in order to auto-initialise the channel for repetitive DMA activity.

Bits	Name	Description
15:0 (Write)	Base Transfer Count Register	Initialisation value for transfer block size - Programmed value should be block size - 1
15:0 (Read)	Current Transfer Count Register	Current value indicating remaining transfer count -1
31:16	Reserved	Not writable; all bits read as 0

Table 10-7: Transfer Count Registers - offset +0x8

Base Address Register & Current Address Register (BAR, CAR)

The **Base** and **Current Address Registers** define the initial and current addresses within the block of memory or I/O being transferred. During channel initialisation, the **Base Address register** is loaded with the starting address of the block; this is subsequently written through to the **Current Address Register** when the channel is enabled. During a DMA transfer cycle on that channel, the address held in the **Current Address Register** is output from the controller, simultaneously incremented or decremented as appropriate, and then rewritten to the **Current Address Register**, which then points to the next data item to be transferred. Upon completion of the block transfer, the **Current Address Register** may optionally be updated from the **Base Address Register** in order to auto-initialise the channel for repeated DMA transfers. If the size of the transfer (as indicated by the Transfer Count registers) causes the **Current Address Register** to overflow or underflow the 32-bit address space, the address wraps around.

Bits	Name	Description
31:0 (Write)	Base Address Register	Initial address of transfer block
31:0 (Read)	Current Address Register	Calculated address of next transfer

Table 10-8: Address Registers - offset +0xC

DMA Status Register (DSR)

The **DMA Status Register** indicates the current condition of each channel, and hence allows the programmer to rapidly determine which DMA channels require service. Each bit corresponds to one channel, with the lowest order bit corresponding to channel 1. A status bit is set in the **DMA Status Register** if interrupts are enabled for the associated channel and any bit (with the exception of **Alternate Phase (CSR[31])**) is set in the **Channel Status Register**. (table 10-5). This register is read-only.

Bits	Name	Description
31:0 (Read)	Channel Status	Reflects service request status of each channel

Table 10-9: DMA Controller Status Register (DSR) - offset +0x200

Chapter 11 - Timer/Counter (TIC)

11.1 Overview

Butterfly contains two Timer/Counter modules. A Timer/Counter module comprises a pair of identical, generic 32 bit interval timer elements. These are controlled by the ARM7 core processor via the Bµild bus.

Each timer-element includes a prescaler that is driven by the system clock, to generate internal timer clocks. The main time interval is then generated by counting these timer clock periods. An interrupt may be generated after the desired number of clock periods.

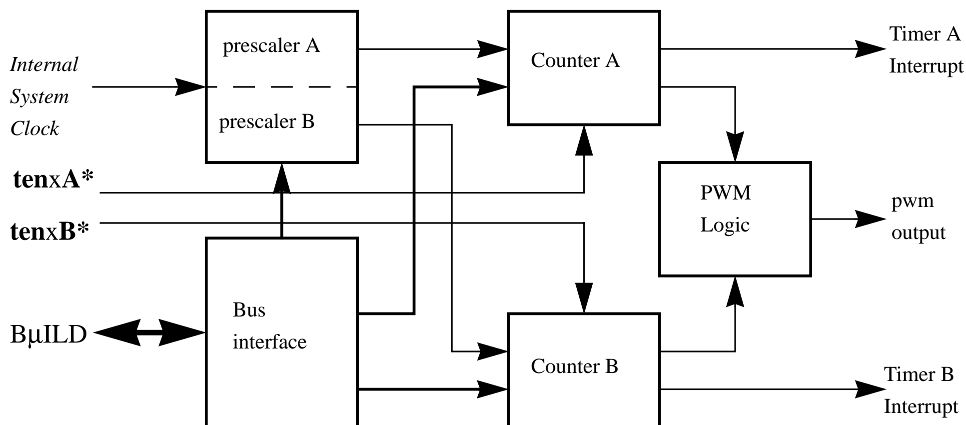
A pulse width modulation (PWM) mode can be configured by combining both timer elements in a Timer/Counter module together. In this mode one timer element controls the high period of an external output, with the other controlling the low period, offering the user complete control over the period and shape of the external output pulse train.

11.1.1 Design Features

The main features of each Timer/Counter module are:

- Two, independently controlled Timer/Counter elements
- Prescalers generating a Timer clock from the system clock
  - Prescale count of 8 bits
  - Division ratio selection by software
- Multiple Timer/Counter modes:
  - countdown to zero
  - free running
  - reload and count on trigger
  - pulse width modulation
- Fully software programmable time-out period
- Maskable interrupt on time-out
- PWM waveform generation by combining the two Timer/Counter elements

11.2 Architecture



* Where x = Timer (e.g.: 1, or 2). see section Figure 1-3: on page 14 for external interface connections

Figure 11-1: Timer/Counter Architecture

11.3 Operational Description

The Timer/Counter consists of two independent synchronous counter elements, both 32 bits long. Each counter has an associated prescale counter, which is used to scale the input clock to its main counter. Both prescalers are fed from the internal system clock, which is derived from the Phase Locked Loop. Reference should be made to the Chapter describing the PLL operating modes for a description of how those modes affect the supplied clock.

The counters are started by controlling the **C/H** (Count/Halt) bit in the Control/Status (**CONSTAT**) register. When a counter reaches zero the Overflow (**OVF**) status flag is set in the **CONSTAT** register and an interrupt pulse will be generated, however this interrupt may be masked by resetting the **MSK** bit in the **CONSTAT** register.

Each counter has the following operating modes:

- Count-Down to zero
- Free Running
- Reload and count on trigger
- Pulse Width Modulation

An external Pulse Width Modulated output is available from Timer 2 only.

11.3.1 Prescaler Operation

The prescaler operates as an 8 bit auto-reload counter, clocked from the system clock. The prescaler continually counts down towards zero. Upon reaching zero it outputs a clock pulse to the main counter and is reloaded with the value in the reload field of the control/status register.

The PreScaler is disabled when the PreScale register is loaded with the value 0x01, and the counter counts system clock cycles.

When a value other than 0x01 is stored in the register, the Prescaler counts from that value down to zero, generating a pre-scaled clock pulse as it reloads, thus dividing the system clock by the value (**PreScale + 1**).

The value 0x00, when programmed into the PreScaler causes the clock to be divided by 257.

The PreScaler counts the falling clock edges and the counter is activated on the following rising clock edge, thus all interrupts generated by the counter reaching zero are activated on the rising edge of the system clock.

11.3.2 Halt on Zero (Mode 0)

The counter decrements from '**0xFFFFFFFF**'. When the counter reaches zero, it is reloaded from the reload register and halts until it is retriggered. This is achieved either by a transition on the external enable, or by setting the **C/H** bit under software control. On reaching zero the **OVF** flag is set and a Timer interrupt may be generated.

11.3.3 Free Running (Mode 1)

When the counter is initially started in Free Running mode, it starts from the value '**0xFFFFFFFF'.** Upon reaching zero the counter is reloaded with the preset value and resumes counting from that value. A Timer interrupt may be generated, note that in this mode **OVF** will not be set and the counter will continue to run.

If required, the initial count from **'0xFFFFFFFF'** may be avoided by starting the counter in Mode 2 with the required value, then stopping before it reaches zero and changing the mode to Free Running. In this case, the counter will restart in the new mode from the value at which it last stopped. This can also apply to Mode 0.

11.3.4 Reload on Trigger (Mode 2)

The counter is reloaded when it receives a start trigger on the external enable input or **C/H** bit, and will then begin to decrement. Upon reaching zero the counter will halt, setting the **OVF** flag and generate an interrupt (if enabled).

11.3.5 Pulse Width Modulation (Mode 3)

One counter is reloaded and started automatically when the other counter reaches zero. Upon reaching zero the counter halts, toggles the state of the **PWM** output and triggers the other counter to run. This mode generates a variable width pulse train on the **PWM** output with counter A controlling the low phase and counter B controlling the high phase.

Proper functioning of the **PWM** mode requires that both counters A and B are set to mode 3. Counter B should be set to start first, as the PWM waveform always starts with a high phase. If counter A is started first the Timer will not begin correct PWM operation until both counters A and B have timed out for the first time.

On BUTTERFLY the PWM output (external pin **Tpwm**) is generated by Counter/Timer 2. The PWM output of Counter/Timer 1 is not available.

As each counter uses a PreScaler, which have two modes of operation, there are two algorithms for the **PWM output**.

PreScale = 0x01;

pwmLo = RLDA + 1;

pwmHi = RLDB +1;

PreScale <>0x01;

pwmLo = ((PreScale+1) * RLDA) -1 ;

pwmHi = ((PreScale+1) * RLDB) +1 ;

11.4 Programmer's Model

Address Offset (Hexadecimal)	Function	Reset Value	R/W
+ 000	Timer A control/status register (CONSTATA)	See Table 11-2	RW
+ 004	Timer A reload register (RLDA)	0xFFFFFFFF	RW
+ 008	Timer A read Register (RDA)	0xFFFFFFFF	R
+ 00C - + 01C	Reserved		-
+ 020	Timer B control/status register (CONSTATB)	See Table 11-2	RW
+ 024	Timer B reload register (RLDB)	0xFFFFFFFF	RW
+ 028	Timer B read register (RDB)	0xFFFFFFFF	R
+ 02C - + FFC	Reserved		-

Table 11-1: Timer/Counter Address Map

Addresses are specified as offsets from the system defined base address.

All the registers are accessed as 32-bit registers, with unused bits defined as zero for read operations. Attempts to access any register with a half-word or byte transfer results in a bus error, as do any accesses to reserved registers or writes to read-only registers. In all cases the following register descriptions apply equally to timer A and timer B.

11.4.1 Register Descriptions

Control/Status Register (CONSTATA/CONSTATB)

This register contains the configuration information and status flags for the associated interval timer.

31–23	22	21	20–19	18	17	16	15–8	7–0
	MSK	OVF	MODE	C/H	ACT	RUN		Prescale Reload

Figure 11-2: Control/Status Register

BIT	Number	Name	Description	Reset Value	R/W
MSK	22	Interrupt Mask Bit	[0] Prevents interrupt generation when zero	0	R/W
OVF	21	Timer Overflow	[1] Set whenever the main counter reaches zero, except when the counter is in free running mode. Reset by any action which restarts the counter, e.g. writing C/H = 1 or setting the external enable pin active	0	R
MODE	20, 19	Timer Mode Control	A two bit field controlling the operating mode of the interval timer. **Bit 20/19:** 00 = Halt on Zero 01 = Free Running 10 = Reload in Trigger 11 = Pulse Width Modulation	00	R/W
C/H	18	Count [1]/ Halt [0]	Controls the state of the main counter. When the counter is externally enabled, setting this bit causes a halted counter to begin counting. Re-setting this bit will cause a running counter to halt. Stopping a counter which has already halted or attempting to start a running counter has no effect.	1	R/W
ACT	17	External Enable Polarity	Determines the sense of the external counter enable input (when present). 0 = active low 1 = active high	1	R/W
RUN	16	Timer Run Status	When set, indicates that the main counter is currently enabled and is running.	0	R
Prescale	msb.....lsb 7....... 0	Clock Prescale Factor	Clock division ratio applied by the pre-scaler. Division ratios of 1 to 255 correspond to values 0x01 to 0xFF, 0x00 gives the ratio 256. For correct prescale operation, on the first occurrence of Timer overflow (OVF) the prescale should only be changed when the C/H bit is reset.	0xFF	R/W

Table 11-2: Control/Status Register bit descriptions

The Counter Reload Register (RLDA/RLDB)

This register holds the 32-bit reload value for the main counter which, in conjunction with the prescaler's reload value, determines the period of the count. Thus the interval timed can be calculated as:

$$T_{timer} = \frac{ReloadValue \times (PreScale + 1)}{f_{BCLK}}$$

BCLK is the internal Bµild bus clock which is derived from SCLK. For full details of this, refer to Chap 4 - Phase Locked Loop, which describes how the different PLL modes affect the clock.

At Reset the initial value shall be 0xFFFFFFFF.

The Counter Read Register (RDA/RDB)

This register holds the current 32-bit count value of the counter. At Reset the initial value shall be 0xFFFFFFFF.

Chapter 12 - Watchdog Timer (WDOG)

12.1 Overview

The function of the Watchdog Timer is to detect hardware or run-time software errors. It performs this function by requiring the processor to write to one of its registers periodically. Should this not occur, the Watchdog will timeout and reset the system. This ensures that hardware/software lock-ups are recoverable.

The watchdog timer generates periodic interrupts to the ARM processor. Should the processor not respond to the interrupt within a certain time a secondary counter times out and resets the system. To respond, the processor needs to write a specific, predefined value into the restart register. This “key” is necessary to guard against errant software accidentally restarting the watchdog, since it is extremely unlikely that it would write the correct 32-bit number into the watchdog restart register.

12.1.1 Design Features

The main features of this watchdog timer are:

- External enable/disable watchdog pin
- “Key” mechanism for restarting the watchdog prevents accidental restarts
- Adjustable interrupt frequency
- Adjustable timeout delay

12.2 Architecture

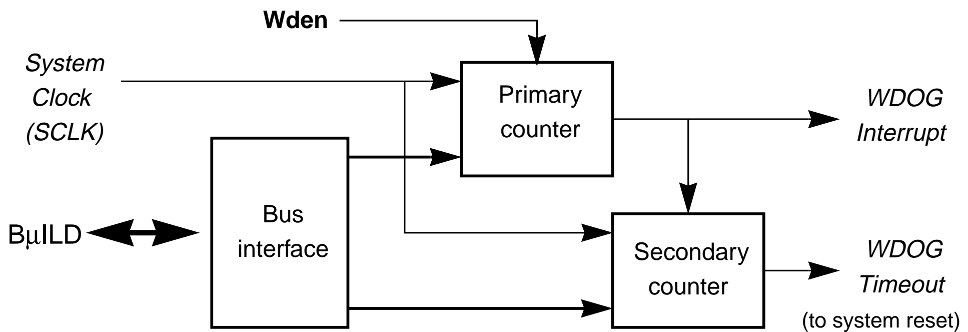


Figure 12-1: Watchdog Structure

12.3 Operational Description

The watchdog consists of two counters, the primary, which is 32 bits long, and the secondary, which is 8 bits long. Both counters are clocked off the system clock, Sclk; the primary directly, and the secondary via a divide-by-sixteen pre-scaler.

12.3.1 Timer Operation and Watchdog Restart Key

When enabled, the primary counter counts down from its 32 bit reload value towards zero. On reaching zero, the Watchdog will generate an interrupt to the ARM processor. The interrupt can be masked out by using the **MSK** bit in the watchdog control register.

It then starts the secondary counter counting down to zero and sets the overflow (**OVF**) flag in the control register. If the processor fails to restart the watchdog before the secondary counter reaches zero, it will generate a timeout signal which causes a system reset.

To restart the watchdog counter, a specific 32 bit value (**=0xECD9F7BD**) must be programmed into the watchdog restart register. This 32 bit “key” activates the restart mechanism which performs the following actions:

- Loads the watchdog primary counter from the reload register
- Resets the **OVF** (overflow) flag in the control register
- Restarts the primary counter
- Reloads and disables the secondary timeout counter
- Removes the timeout signal

The reload register enables the time before an interrupt is generated to be varied in software. Changes to the reload register value will only be reflected in the main watchdog counter after the next software reload.

12.4 Programmer's Model

Addresses are specified as offsets from the **WDOG** base address as defined by the system configuration.

Address Offset (Hexadecimal)	Function	Reset Value	Type	Comments
+ 000	Control/Delay Register	0x000000FF	RW	See 12.4.1.
+ 004	Reload Register	0xFFFFFFFF	RW	controls interrupt frequency
+ 008	Read Register	0xFFFFFFFF	R	current primary counter value
+ 00C	Restart Register	0x00000000	W	"key" input Key value = 0xECD9F7BD
+ 010.....FFF	Reserved		-	

Table 12-1: Register Map

12.4.1 Control Register Format

The control register is 32 bits wide, with unused bits defined as zero. Attempts to access the register as a byte or a half-word will cause a bus error exception. The register is defined as follows, with all control bits active high:

- A single bit mask (**MSK**) which controls whether the zero count state of the watchdog generates an interrupt to the ARM processor.
- A single read-only bit field (**OVF**) which is set whenever the watchdog reaches zero. **OVF** can only be reset by restarting the watchdog timer.
- An additional (**RUN**) bit, valid only when reading the control register, shows whether the watchdog counter is currently enabled.
- An 8 bit field containing the timeout delay used by the secondary watchdog timer.

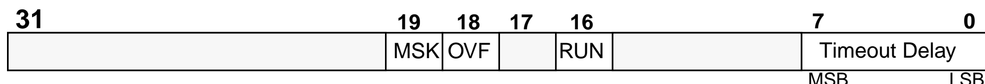


Figure 12-2: Control Register Format

12.5 External Interface

The **WDOG** has one external port, the enable input (**nWden**). The timing relationship between **nWden** and the main system clock (**SCLK**) is shown in Figure 12-3.

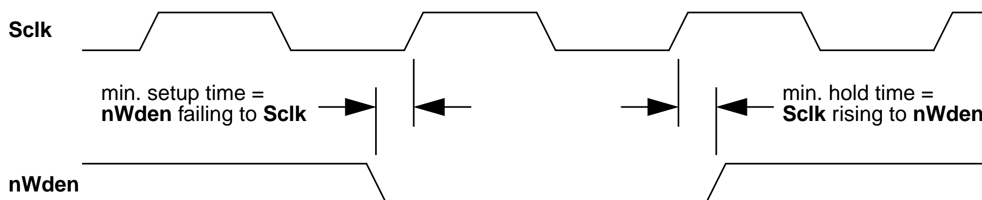


Figure 12-3: Timing Relationship Between nWden and Sclk

Appendix A - BμILD Bus Operation

A.1 Introduction

The Mitel' Microcontroller product range uses a system Bus known as BμILD (Bus for μController Integration in Low-power Designs)

Although the designer need not know details of the internal operation of this bus for most applications, the implementation details are included for information purposes.

This chapter contains a technical overview of the protocols associated with bus arbitration and bus transactions

This represents sufficient information to give a working knowledge of the implementation of the BμILD Bus within Mitel embedded ARM systems. The BμILD architecture is optimised for efficient on-chip embedded systems. It is primarily designed to support ARM CPUs and support modules, but is extensible to other processors and logic.

The ARM7 CPU is a small and powerful 32-bit RISC CPU module suitable for integration in larger ASIC designs. However when designing such embedded systems the following problems must be addressed:

- Ad-hoc system design from scratch for each deeply embedded design significantly extends time to market and/or project development resource requirements.
- Manufacturing test requires fast efficient access to all component modules and I/Os, together with a rapid route for test pattern generation.
- The development of microprocessor based systems requires the use of Debug techniques such as Logic analysis and ICE. These techniques can require special chip bond-outs to gain access to the embedded CPU core and to enhance system visibility.

Through the use of Bμild, a modular approach is taken to solve these problems. Libraries of macro-functions (modules) complete with associated test patterns and software drivers may be constructed and subsequently re-used within differing designs.

The following text describes the key aspects of BμILD including the principal functional elements and protocol definitions.

A.1.1 Bus Masters

The bus master is the controller of the current bus transaction. A bus master initiates bus requests (which automatically managed by Butterfly's internal Bus arbitor), generates addresses and controls data transfers while it has bus access, by reading or writing data over the data bus.

Bus masters on Butterlfy are:

- The CPU(s)
- Direct Memory Access (multi-channel) controller(s)
- BμILD Broadcast Module (for external test and debug)

A.1.2 Bus Slaves

A bus slave responds to addresses present on the internal Bus that are in its allocated range within the address map. It will supplies or receives data during read or write cycles on demand. A slave may set a wait signal to delay access using the synchronous bus transfer protocol.

Example slave devices are:

- UARTs
- The MPC
- Timers
- POCO

A.1.3 System Arbitration and Multiple Bus Master Support

The bus specification supports multiple bus masters. Bus masters must request the bus using a centralised arbiter. For example, DMA controllers may share the bus with one or more CPUs with the priorities resolved by the arbitration logic. The arbitration is pipelined by one clock cycle to avoid the need for redundant bus cycles during bus mastership changes .

Bus Masters

All bus masters are able to arbitrate for the bus using a request and grant protocol. When the bus is not granted, the master isolates itself from the bus and not drive or respond to the bus control signals. A bus master will drive the full 32 bit address bus during bus ownership, and drive the full 32 bit data bus during bus writes. The bus master also samples and responds to the bus control signals set by slave devices to indicate their status.

Bus Slaves

Bus slaves, both internal memory and I/O devices, are selected by the decoding of the master's broadcast address. When selected during the address decode phase, the slave drives the bus status signals to indicate the condition of the current transaction. Invalid or illegal transaction types are indicated either directly by the slave or by an associated protection mechanism. The slave drives the full 32 bit data bus during read transactions. The data transaction phase may be extended if necessary for multiple bus cycles to allow for slaves with long transaction latencies.

Appendix B - Physical and Electrical Specifications

B.1 Device I/O Summary

Mnemonic	Type	Function
Dreq1,2	I	DMA request
Dack1,2	O	DMA acknowledge
Sclk	IO	System clock input or output
Oscen	I	Oscillator Enable
Set_3V	I	Select 3 Volt/ $\overline{\text{5 Volt}}$ Supply mode
Pllpd	I	Disable Phase Locked Loop
Oscin/Bypass	I	Crystal connection/Clock Mode
Oscout	O	Crystal connection
nSreset	I	System reset input, has 100K pull up resistor for use with external capacitor
Sdata<31:0>	IO	Data inputs/outputs
Sadd<21:0>	IO	Address Outputs
nScs<3:0>	O	Chip selects for external memory system
nSwe<3:0>	O	Write enables for external memory system
nSoe	O	Output enable for external memory system
Swait	I	Causes wait states within the MPC
Sr_size<1:0>	I	Reset bus size
*Sbigendian	I	Bigendian select
Ten1	I	Timer enable-1
Ten2	I	Timer enable-2
Tpwm	O	Timer PWM output
nWden	I	Watchdog enable
Utxd1,2	O	UART transmit data

Table B-1: Device I/O Summary

Mnemonic	Type	Function
Urxd1,2	I	UART receive data
Ueclk/nUdcd 1,2	I	UART external clock/ Data Carrier Detect
nUrts1,2	O	UART ready to send
nUcts1,2	I	UART clear to send
Pdat<7:0>	IO	Parallel port data port
nPstrb	I	Parallel port data strobe
nPack	I	Parallel port acknowledge
nPobf	O	Parallel port output buffer full
Pibf	O	Parallel port input buffer full
Iextint1,2	I	External interrupt inputs
Bdiag<3:0>	O	Encoded bus state
VDD	-	Power
GND	-	Ground

Table B-1: Device I/O Summary (Continued)

* Sbigendian applies to the MPC, to change the orientation of how bytes are read/written. Note internally to the device, ARM core BIGEND pin is tied low.

B.2 Pin Position Details

Key:

Shaded areas: Outputs

n = active low signal

Reserved <DNC> = Reserved, Do Not Connect

Reserved <Low> = Reserved, Tie to 0 Volts

Reserved <High> = Reserved, Tie to VDD

Please see next page for pin assignments.

1	CP (VDD)	37	CP (GND)	73	CP (VDD)	109	CP (GND)
	Sadd<14>		Sdata<5>		Pibf	110	Reserved <Low>
	Sdata<13>		Sadd<5>	75	nPobf		Reserved <Low>
	Sdata<29>	40	Sdata<4>		nPack		Reserved <DNC>
5	Sadd<13>		Sdata<20>		nPstrb		Reserved <Low>
	Sdata<12>		Sadd<4>		Pllpd		Reserved <Low>
	Sdata<28>		Sdata<3>		GND	115	Bdiag<0>
	Sadd<12>		Sdata<19>	80	VDD		Bdiag<1>
	Sdata<11>	45	Sadd<3>		Pdat<7>		Bdiag<2>
10	Sdata<27>		Sdata<2>		Pdat<6>		Bdiag<3>
	GND		GND		Pdat<5>		Sr_size<0>
	VDD		VDD		Pdat<4>	120	Sr_size<1>
	Sadd<11>		Sdata<18>	85	Pdat<3>		GND
	Sdata<10>	50	Sadd<2>		Pdat<2>		VDD
15	Sdata<26>		Sdata<1>		Pdat<1>		Tpwm
	Sadd<10>		Sdata<17>		Pdat<0>		Ten2
	Sdata<9>		Sadd<1>		VDD	125	Ten1
	Sdata<25>		Sdata<0>	90	GND		Iextint2
	Sadd<9>	55	Sdata<16>		Oscin/Bypass		Iextint1
20	GND		Sadd<0>		Oscout		nSoe
	VDD		nSreset		Set_3V		nWden
	Sdata<8>		Dreq2		nUcts2	130	Sadd<19>
	Sdata<24>		Dack2	95	nUrts2		Sadd<21>
	Sadd<8>	60	VDD		Urxd2		Sadd<18>
25	Sdata<7>		GND		Utxd2		Oscen
	Sdata<23>		Dreq1		Ueclk2/nUdcd2		GND
	Sadd<7>		Dack1		VDD	135	VDD
	VDD		nScs<2>	100	GND		Sadd<17>
	GND	65	nScs<3>		Sclk		Sadd<20>
30	Sbigendian		nScs<1>		nUcts1		Sadd<16>
	Swait		nScs<0>		nUrts1		Sdata<15>
	Sdata<22>		nSwe<3>		Urxd1	140	Sdata<31>
	Sdata<21>		nSwe<2>	105	Utxd1		Sdata<30>
	Sdata<6>	70	nSwe<1>		Ueclk1/nUdcd1		Sadd<15>
35	Sadd<6>		nSwe<0>		Reserved <LOW>		Sdata<14>
36	CP (VDD)	72	CP (GND)	108	CP (VDD)	144	CP (GND)

B.3 Package Options

The standard package for the Butterfly is a 144 Plastic Quad Flat Pack and is detailed below. A 144 TQFP (Body size 20 x 20 mm) version is also available on special order, please contact your local Mitel Representative for details.

Pin pitch : 0.65mm

Body size: 28 x 28mm

Packing: Supplied in Trays of 24 pieces

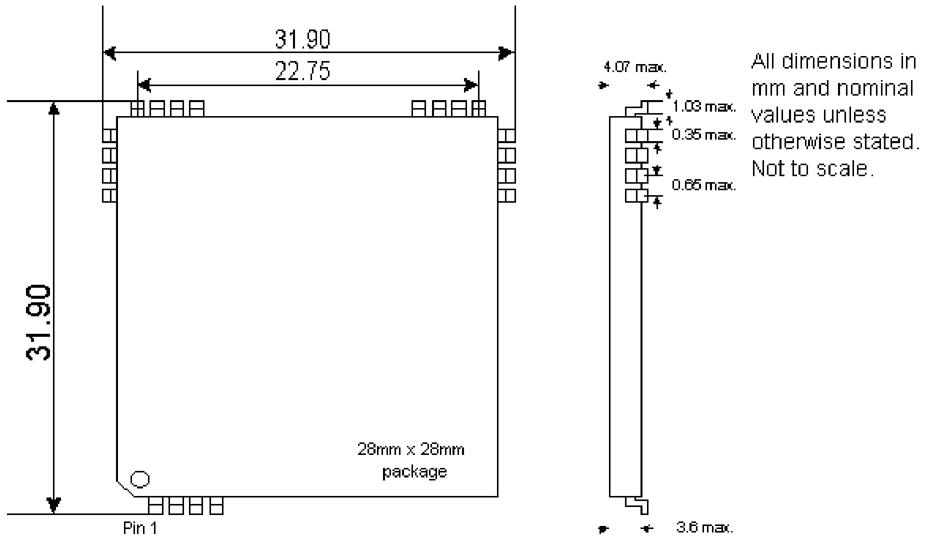


Figure B-1: BUTTERFLY 144 Plastic Quad Flat Pack

B.4 Electrical performance Characteristics.

Details of these can be found in a separate document entitled “Butterfly Microcontroller Performance Supplement” publication number SP4578

Appendix C - Further Information

Further information on Mitel’ ARM and other products may be found on our World-Wide Web page: www.mitelsemi.com

C.1 Related Documents

The following documents may be of use when evaluating and developing ARM based systems:

Document	Publication Number
Data Sheets and Specifications:	
Butterfly Microcontroller Performance Supplement	SP4578
ARM60-B Microprocessor (incorporating an ARM6 core plus JTAG)	DS3553
ARM610-B Microprocessor: (incorporating an ARM6 core, 4K Cache, MMU and JTAG)	DS3554
Product Overviews	
Butterfly High Performance, Low Cost 32-bit Microcontroller	LF4097
Microcontroller Family Development Support	LF4380
ARM Software Development Toolkit for Windows	LF4317
An Introduction to Thumb	MS4417
Application Notes:	
HP Logic Analyser Support for Butterfly	
ARM6 in DSP Applications: Use of the MUL Instruction	
Benchmarking, Performance, Analysis and Profiling	
Customising the ARMulator	

Table C-1: Related Documents

Document	Publication Number
Implementing Fast Fourier Transforms on the ARM Risc Processor	
JPEG compression on the ARM processor	
Rules for ARM Code Writers	
..... Please call for copies or information on other topics	

Table C-1: Related Documents

C.2 Worldwide Offices

Key: CS = Customer Services, SD = Semi-Custom Design Services

Europe

France	CS SD	Mitel Semiconductor, France. Tel: (1) 69 18 90 00 Fax: (1) 64 46 06 07.
Germany	CS SD	Mitel Semiconductor, Germany. Tel: (089) 3609060 Fax: (089) 36090655
Italy	CS	Mitel Semiconductor, Italy. Tel: (02) 6607151 Fax: (02) 66040993.
Sweden	CS	Mitel Semiconductor, Sweden.Tel: (08) 7029770 Fax: (08) 6404736.
UK, Eire, Denmark, Finland, Norway	CS SD	Mitel Semiconductor, UK, Tel: (01793) 726666 Fax: (01793) 518582.

Far East

Korea	CS	Mitel Semiconductor, Seoul, Korea. Tel: (2) 5668141 Fax: (2) 5697933
Japan	CS SD	Mitel Semiconductor, Tokyo, Japan. Tel: (03) 5276-5501 Fax: (03) 5276 5510.
S.E Asia	CS	Mitel Semiconductor, Singapore. Tel: 3827708 Fax: 3828872
Taiwan	CS	Mitel Semiconductor, Taipei, Taiwan. Tel: (02) 5461260 Fax: (02) 719 0260

USA & Canada

USA & Canada	CS SD	Mitel Semiconductor, Scotts Valley, CA, USA.Tel: (408) 438 2900 Fax: (408) 438 7023
South West	CS SD	Mitel Semiconductor, Irvine, CA, USA. Tel: (714) 852-3900 Fax: (714) 852-3910
North West	CS SD	Mitel Semiconductor, San Jose, CA, USA. Tel: (408) 451-4700 Fax: (408) 451-4710
Central and East	CS SD	Mitel Semiconductor, Dedham, MA, USA. Tel: (617) 251-0100. Fax: (617) 251-0104

Mitel also has many other franchised Distributor outlets World-wide. Details can provided by contacting any of the above numbers.

Appendix D - PCB Layout Guidelines

The Butterfly device is a fully featured microcontroller capable of being used with an 8, 16, or 32 bit external memory bus. As such, whilst the power consumption of the embedded ARM is low, in common with other microcontrollers, significant near-instantaneous step-changes in dynamic current consumption can occur during state changes on the external address and data buses. It is therefore most important that a well regulated power supply source with sufficient dynamic current sourcing capacity must be used to prevent temporary VDD depression under conditions of high instantaneous current sourcing.

D.1 Considerations regarding Power distribution

A minimal resistance ground connection must be provided to prevent temporary rises in potential on Ground (GND) pins under conditions of high instantaneous current sinking.

To minimise these effects the following recommendations are made for production systems:-

(i) The Microcontroller should be soldered directly on to the same PCB as all it's memory devices. Ideally, a single PCB only should be used.

(ii) The PCB should have a GND power plane, and a separate VDD power plane.

(iii) Each adjacent GND/VDD pin pair of the microcontroller must be adequately and locally decoupled by a capacitor or capacitors connecting between the VCC supply and GND pins of the pair. Decoupling must be as close to the pin-pairs as physically possible, and within 1cm's distance maximum. A suggested value is 0.1 µF Ceramic. Surface mount

capacitors are preferable to pinned capacitors, due to their lower inductance.

(iv) A power supply reservoir capacitor may be used to smooth out small transients on the VDD line. A suggested value is between 50 and 100 µF electrolytic.

D.2 Considerations regarding PCB track lengths

The output stages of the Butterfly device, like other microcontrollers, are capable of switching output state in a matter of nanoseconds. Consequently, transmission line effects can occur in connected PCB tracks, and can cause ground-potential undershoot and VDD overshoot of the signal concerned to occur.

Example

in a typical small system (with typical PCB composition) having track lengths of 8" with 50pF total loading, a reflection would return in around 4nS which can be very close to the edge transition time of a low going signal. The reflection would cause undershoot of the signal together with some 'ringing' around the GND potential. The frequency of 'ringing' is proportional to the length of the line. The amplitude of ringing is proportional to "the mismatch" (of impedances).

Reflection transmission times vary according to a tracks' length, load, position (inside the PCB or on the surface), cross section, and distance from other layers. Some helpful formulae to allow calculation of PCB track impedances and propagation characteristics can be found in appendix C of *Reference 1* (see end of this Chapter).

The output drivers on the Butterfly microcontroller have the following characteristics:-

The no load edge speeds are

DOWN: 184ps

UP: 341ps.

The loading factors for the edges are:-

DOWN: 21ps/pf

UP: 53ps/pf

Assuming a PCB made of industry standard commercial grade materials and spacing rules. Applying the equations in *Reference 1* to a Signal X with a 50pF load connected by 16" of track {including spurs}, and 8" worst case single track length, results in:

Track impedance results

(embedded track)-47 Ohms

(surface track) - 90 Ohms

Propagation delay results (Tpd)

(embedded)	3.15nS/foot	= 2.1nS for 8" of track.
(surface)	2.5nS/foot	= 1.66 nS for 8 " of track.

Note that the signal has to travel both up and back down the track.

The above example is taken from a test system in which a 50 Ohm series resistor placed at the signal source was found to almost eliminate both undershoot and overshoot.

SUMMARY

Generally, a signal track should be examined for transmission line effects if:-

2 x Tpd x line length > Transition time of the edge

(Tpd = Transmission time of signal along the PCB track in one direction)

To minimise under/over-shoot effects it is recommended that the following guidelines are noted when implementing a Butterfly/PCB system:

First estimate and analyse whether your system signals will be prone to significant transmission line effects. For those signals that are, the following steps should be considered:

A low value resistor, placed in series between the microcontroller output and the PCB track will minimise both signal-undershoot and overshoot (A value of 50 Ohms has been found to be highly effective in a specific system, but this will depend upon the particular signal, the PCB track length, and loading). It is important however to check that if other nodes of the track are connected to components with significant sink or source currents, the voltage drop across the series resistor will not compromise the signal integrity.

Reference 1: "Second edition - EMC for Product designers" by Tim Williams,

ISBN 0 7506 24663. Published by Newnes.