

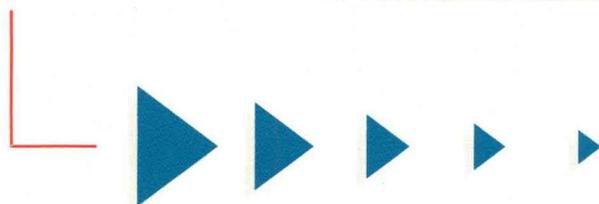


SPARClite Embedded Processor User's Manual

FUJITSU

SPARClite

Embedded Processor
User's Manual



MB86930
MB86931
MB86932
MB86933

1993

FUJITSU



SPARClite User's Manual



Fujitsu Microelectronics, Inc.
Semiconductor Division

CREDITS



Book design & illustration by Communication Graphics. This book, excluding the cover, was illustrated, and produced on Macintosh Computers using FrameMaker® workstation publishing software.

Cover design by Gregg Robles.

TRADEMARKS



NICE is a trademark of Fujitsu Microelectronics, Inc.

SPARC is a registered trademark of SPARC International, Inc. based on technology developed by Sun Microsystems, Inc.

SPARC*lite* is a trademark of SPARC International exclusively licensed to Fujitsu Microelectronics, Inc.

SPARCstation is a trademark of SPARC International, Inc. Products bearing the SPARC trademarks are based on an architecture developed by Sun Microsystems, Inc.

Macintosh is a registered trademark of Apple Computer, Inc. FrameMaker is a registered trademark of Frame Technology Corporation.

Copyright © 1993 Fujitsu Microelectronics, Inc., Semiconductor Division.

All rights reserved. This publication contains information considered proprietary by Fujitsu Limited and Fujitsu Microelectronics, Inc. No part of this document may be copied or reproduced in any form or by any means or transferred to any third party without the prior written consent of Fujitsu Microelectronics, Inc.

Circuit diagrams utilizing Fujitsu products are included as a means of illustrating typical semiconductor applications. Consequently, complete information sufficient for design purposes is not necessarily given.

Fujitsu Limited and its subsidiaries reserve the right to change products or specifications without notice. **Fujitsu advises its customers to obtain the latest version of device specifications to verify, before placing orders, that the information being relied upon by the customer is current.**

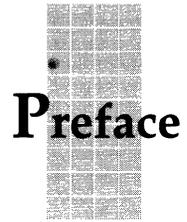
The information contained in this document does not convey any license under copyrights, patent rights or trademarks claimed and owned by Fujitsu Limited or its subsidiaries. Fujitsu assumes no liability for Fujitsu applications assistance, customer's product design, or infringement of patents arising from use of semiconductor devices in such systems' designs. Nor does Fujitsu warrant or represent that any patent right, copyright, or other intellectual property right of Fujitsu covering or relating to any combination, machine, or process in which such semiconductor devices might be or are used.

Fujitsu Microelectronics, Inc.'s Semiconductor Division's products are not authorized for use in life support devices or systems. Life support devices or systems are device or systems which are:

1. **Intended for surgical implant into the human body.**
2. **Designed to support or sustain life; and when properly used according to label instructions, can reasonably be expected to cause significant injury to the user in the event of failure.**

The information contained in this document has been carefully checked and is believed to be entirely accurate. However, Fujitsu Limited and Fujitsu Microelectronics, Inc. assume no responsibility for inaccuracies.

This document is published by the marketing department of Fujitsu Microelectronics, Inc., Semiconductor Division, 3545 North First Street, San Jose, California, U.S.A. 95134-1804.



Preface



About This Manual

SPARClite™ is a family of microprocessors which conform to Version 8 of the SPARC architecture and which have been optimized for use in embedded control. This manual is the definitive guide for understanding this family of embedded processors. It describes both the SPARClite architecture and the first four members of the family - the MB86930, MB86931, MB86932, and MB86933. The intended audience for this manual is both hardware systems designers and applications programmers.

Organization

This manual is divided into four sections, each with its own table of contents.

- Section 1 describes the SPARClite architecture and specifically, the MB86930 microprocessor (the first member of the SPARClite family). This section can be read by itself for an understanding of the SPARClite architecture or the MB86930 processor.
- Section 2 describes the MB86931 which is a superset of the MB86930. This section describes only the additional feature set of the MB86931 and therefore should be read after section 1.
- Section 3 describes the MB86932 which is a superset of the MB86930. This section describes only the additional feature set of the MB86932 and therefore should be read after section 1.

- Section 4 describes the MB86933 which is a subset of the MB86930. Unlike sections 2 and 3, this section contains a complete description of the MB86933 and can be read independently of all other sections.

Notation

This manual uses the following notational conventions:

- Active-low signal names are preceded with a dash, as in `-RESET`.
- Numerals without any special prefix are in base 10. Hexadecimal numerals are preceded by `0x`, and binary numerals are preceded by `0b`. Thus, $28 = 0x1C = 0b11100$.

Related Literature

Additional information can be found in the following documents:

- MB86930 SPARClite 32-Bit RISC Embedded Processor Data Sheet—Describes the MB86930 processor in detail, including complete physical, electrical, and timing characteristics. Available from Fujitsu Microelectronics' Semiconductor Division.
- MB86931 SPARClite 32-Bit RISC Embedded Processor Data Sheet—Describes the MB86931 processor in detail, including complete physical, electrical, and timing characteristics. Available from Fujitsu Microelectronics' Semiconductor Division.
- MB86932 SPARClite 32-Bit RISC Embedded Processor Data Sheet—Describes the MB86932 processor in detail, including complete physical, electrical, and timing characteristics. Available from Fujitsu Microelectronics' Semiconductor Division.
- MB86933 SPARClite 32-Bit RISC Embedded Processor Data Sheet—Describes the MB86933 processor in detail, including complete physical, electrical, and timing characteristics. Available from Fujitsu Microelectronics' Semiconductor Division.
- SPARClite Application Notes — Discuss specific design issues in detail. Available from Fujitsu Microelectronics' Semiconductor Division.
- The SPARC Architecture Manual (version 8) — This document is a more detailed description of the version 8 SPARC architecture on which the SPARClite family is based. Available from SPARC International, Menlo Park, California.



Section 1

.....

MB86930

Chapter 2: Programmer's Model

2.1 Program Modes	2-1
2.2 Memory Organization	2-2
2.3 Registers	2-3
2.3.1 Register Windows	2-4
2.3.2 Special Uses of the r Registers	2-7
2.3.3 SPARC-Defined Special-Purpose Registers.....	2-7
2.3.4 Memory-Mapped Control Registers	2-12
2.4 Data Types	2-22
2.5 Instructions	2-22
2.5.1 Instruction Formats	2-24
2.5.2 Logical Instructions	2-25
2.5.3 Arithmetic and Shift Instructions.....	2-26
2.5.4 Control Transfer Instructions.....	2-32
2.5.5 Load and Store Instructions.....	2-39
2.5.6 Read and Write Control Register Instructions	2-42
2.6 Data and Instruction Caches	2-44
2.6.1 Structure	2-44
2.6.2 Operation	2-47
2.7 Interrupts and Traps.....	2-50
2.7.1 Trap Types	2-51
2.7.2 Trap Behavior.....	2-53
2.7.3 Reset and Error Modes	2-54
2.8 Debug Support Unit.....	2-55
2.8.1 Monitor Mode	2-56
2.8.2 Breakpoint Registers	2-57
2.8.3 Breakpoint Traps.....	2-60
2.9 SPARC Compliance	2-63

Chapter 3: Internal Architecture

3.1 Integer Unit.....	3-2
3.1.1 I Block.....	3-3
3.1.2 A Block	3-8
3.1.3 E Block.....	3-10
3.1.4 Programmer-Visible State and Processor State	3-15
3.1.5 IU Support for Debugging.....	3-16
3.2 Data and Instruction Caches	3-16
3.3 Bus Interface Unit.....	3-17
3.3.1 Buffers.....	3-17
3.3.2 Exception Handling	3-18
3.3.3 Effect on the Pipeline.....	3-18

Chapter 4: External Interface

4.1 Signals.....	4-1
4.1.1 Processor Control and Status.....	4-3
4.1.2 Memory Interface	4-4
4.1.3 Bus Arbitration	4-6
4.1.4 Peripheral Functions.....	4-7
4.1.5 Emulator Bus	4-7
4.1.6 Test and Boundary-Scan	4-7
4.2 Bus Operation.....	4-8
4.2.1 Exception Handling	4-9
4.2.2 Bus Cycles.....	4-10
4.3 System Support Functions	4-16
4.3.1 System-Configuration Registers.....	4-16
4.3.2 Same-Page Detection	4-18
4.3.3 Programmable Timer.....	4-19

Chapter 5: Programming Considerations

5.1 Initialization	5-1
5.1.1 Establishing the Processor State	5-2
5.1.2 Configuring the System	5-2
5.1.3 Initializing the On-Chip Cache.....	5-4
5.2 Trap Handling	5-5
5.3 Register and Stack Management	5-11
5.3.1 Registers	5-11
5.3.2 Memory Stack	5-16
5.3.3 Functions Returning Aggregate Values	5-17
5.3.4 Leaf Procedure Optimization	5-18
5.3.5 Register Allocation Within a Window.....	5-22
5.3.6 Other Register and Window Usage Models	5-23
5.4 Cache Management	5-24
5.5 Division Routines Using the DIVSc Instruction	5-25
5.5.1 Simple Divide Step Examples.....	5-25
5.5.2 Signed Division with Doubleword Dividend (divs2)	5-27
5.5.3 Signed Division with Word Dividend (divs1)	5-30
5.5.4 Unsigned Division with Doubleword Dividend (divu2).....	5-32
5.5.5 Unsigned Division with Word Dividend (divu1).....	5-33
5.5.6 Divide Step In Support Of A To D Converter Compensation	5-34
5.6 Using the SCAN Instruction	5-37
5.6.1 Scan in Support of Software Floating Point.....	5-37
5.6.2 Scan in Support of Run Length Encoding	5-39
5.7 Multiply Routines Using the MULSc Instruction	5-41
5.7.1 Simple Multiply Step Examples	5-42
5.7.2 Signed Multiplication Using Multiply Step	5-44
5.7.3 Unsigned Multiplication Using Multiply Step	5-45
5.7.4 Corner Turning Buffer Using Multiply Step	5-46

Chapter 6: System Design Considerations

6.1 Clocks	6-2
6.2 Memory and I/O Interfacing	6-2
6.2.1 Interfacing SRAM	6-3
6.2.2 Interfacing Page-Mode DRAM	6-4
6.2.3 Interfacing EPROM and Other Devices with Slow Turn-off	6-6
6.2.4 Illegal Memory Accesses	6-7
6.2.5 I/O Interfacing Example: Ethernet Device	6-7
6.3 DMA and Bus Arbitration	6-9
6.4 MB86940 Peripheral Chip	6-10
6.4.1 Interrupt Control	6-10
6.4.2 Counter/Timers	6-11
6.4.3 USARTs	6-11
6.5 In-Circuit Emulation	6-11
6.6 Physical Design Issues	6-12

Chapter 7: Instruction Set

7.1 Suggested Assembly Language Syntax	7-1
7.1.1 Register Names	7-2
7.1.2 Special Symbol Names	7-2
7.1.3 Values	7-3
7.1.4 Labels	7-3
7.1.5 Comments	7-3
7.2 Syntax Design	7-3
7.3 Synthetic Instructions	7-3
7.4 Binary OpCodes	7-3
7.5 Instruction Set	7-16

Chapter 8: JTAG

8.1 Introduction	8-1
8.2 Test Access Ports (TAP)	8-2
8.2.1 TCK.....	8-2
8.2.2 TMS	8-2
8.2.3 TDI.....	8-3
8.2.4 TDO.....	8-3
8.2.5 –TRST	8-3
8.3 Test Instructions.....	8-3
8.3.1 BYPASS	8-4
8.3.2 SAMPLE/PRELOAD	8-4
8.3.3 EXTEST	8-5
8.3.4 JTAG Cells	8-5
8.3.5 Input Cell	8-5
8.3.6 Output Cell	8-6
8.3.7 I/O Cell	8-6
8.3.8 Output Cell with Set	8-6
8.4 Operation	8-8
8.5 The TAP Controller.....	8-10
8.5.1 TAP Controller State Diagram	8-10
8.6 MB86930 JTAG Pin List.....	8-16

CHAPTER 1



Overview

The SPARClite family is a collection of SPARC-based microprocessors optimized for use in embedded systems. Processors in the SPARClite family conform to the SPARC version 8 architecture definition; in particular, they are fully compatible with existing SPARC code and existing SPARC development environments. The MB86930 processor is the first member of the SPARClite family. This chapter provides a quick introduction to the processor architecture and the MB86930 in particular. Subsequent chapters will review this material in more detail.

1.1 General Description

The MB86930 is a high-performance processor suitable for use in embedded control applications such as printers, scanners, robotic machinery, telecom switches and monitors, and I/O subsystems. It operates at clock speeds up to 50 MHz, executing SPARC instructions at a maximum rate of 46 MIPs, and includes 2 Kbytes of instruction and 2 Kbytes of data cache on chip. It is available in a variety of packages, depending on clock-speed and power-dissipation requirements.

The processor consists of a Harvard (Aiken) architecture Integer Unit (IU) core, instruction and data caches, a Bus Interface Unit (BIU), and an In-Circuit Emulator Unit (EMU). These units are connected internally over separate instruction and data buses, and to external memory and I/O over a unified (instruction and data) bus which carries 32 bits of address and 32 bits of data.

The register file in the IU implements 8 register windows. An integer multiply unit (MU) within the IU speeds applications which require integer multiplication. The processor uses software to emulate floating-point instructions at rates up to 1 MFLOP.

The internal instruction and data caches make it possible to sustain a processing rate close to one cycle per instruction by providing the IU at 50 MHz with a maximum aggregate data throughput of 400 Mbytes/sec (two 32-bit words per cycle). The maximum external data throughput is 200 Mbytes/sec (1 word per cycle). In many applications, the internal caches make it possible to maintain high throughput even with slow external memory; SPARClite is therefore a cost-effective solution in embedded control applications that require high processing throughput but cannot tolerate the cost of large, high-speed memories.

The MB86930 is designed with Fujitsu's AS technology, a 1 μ and 3-level metal process with minimum drawn transistor lengths of 0.8 μ . The design of the data path and other arrayed blocks is fully custom to optimize die area and speed. Random control blocks are based on standard cells. All circuits are fully static.

While it does provide a mechanism for code and data protection, the MB86930 is optimized for embedded applications which do not require virtual-to-physical address translation. Using an MB86930 processor in a virtual-memory system, while possible, would require an external Memory Management Unit for address translation.

1.2 Special Features

This section lists some of the features which give the MB86930 its superior speed, flexibility and efficiency and make it an ideal choice for a wide variety of low cost, high-performance embedded systems.

- **Fast Instruction Execution:** The instruction set is streamlined and hardwired for fast execution, with most instructions executing in a single cycle. At 50 (40,30,20) MHz, the MB86930 executes instructions at a peak rate of 50 (40,30,20) MIPs, and can sustain performance of 46 (37,28,18) MIPs. The Integer Unit (IU) features a 5-stage pipeline which has been designed to handle data interlocks, has an optimized branch handler for efficient control transfers, and a bus interface to handle single cycle bus accesses to on-chip cache.
- **Large Register Set:** An internal register file consisting of 136 registers organized into eight overlapping windows speeds interrupt response time and context switches. The register file minimizes accesses to memory during procedure linkages and facilitates passing of parameters and assignment of variables, reducing code in many programs. Reduced code, in turn, can fit more easily into the instruction cache.

- **On-Chip Caches:** On-chip data and instruction caches decouple the processor from external memory latency. The caches are organized as two-way set-associative for improved hit rates, as compared with direct-mapped caches.
- **Cache Locking:** Both data and instruction entries can be locked into their respective caches to ensure deterministic response and highest performance for critical or frequently recurring routines. Maximum flexibility has been designed into the cache to allow all or selected portions to be locked.
- **Separate Instruction and Data Paths On-Chip:** Separate 32-bit instruction and data buses provide a high-bandwidth interface between the IU and on-chip cache. These buses support single cycle instruction execution as well as single cycle data transfers with the cache. The on-chip bus design also supports future expansion of the MB86930.
- **System Support Functions:** The requirement for glue logic between the MB86930 and the system is minimized by providing programmable chip selects, programmable wait-state circuitry, and support for connection to fast page-mode DRAM. Multiple bus masters are supported through a simple handshake protocol.
- **Clock Generator:** To simplify clock design, a crystal can be connected directly to the on-chip oscillator, or an external clock source can be used. A phase-locked loop minimizes the skew between on- and off-chip clocks.
- **Enhanced Instruction Set:** The MB86930 incorporates a fast integer multiply instruction which executes in a fast 5, 3 or 2 cycles for 32-bit, 16-bit or 8-bit operands. An integer divide-step instruction cuts divide times by a factor of 5 to 10 over previous SPARC implementations. A scan instruction supports a single-cycle search for the most significant non-sign bit in a word.
- **Fully Static Circuit Design:** Its static design gives the MB86930 superior noise immunity. Future members of the SPARC_{lite} family will support a low-power mode, in which the processor clock can be slowed or stopped for arbitrary periods of time to reduce operating current with no loss of internal state.
- **Test and Debug Interface:** The MB86930 supports production test through industry standard JTAG boundary scan. Hardware emulation is supported with on-chip breakpoint and single step logic. A dedicated emulator bus provides a means to trace transactions between the integer unit and on-chip cache.

1.3 Programmer's Model

This section briefly introduces those aspects of the SPARC_{lite} processor architecture which are visible to software: the user and supervisor modes of program execution; the organization of the address space; the processor's register set, supported data types, and instruction set; the on-chip caches; and interrupts and traps. Each of the topics discussed here is developed more fully in subsequent chapters.

1.3.1 Program Modes

The SPARClite architecture supports protection in multitasking environments by providing two mutually exclusive modes of program execution, *user mode* and *supervisor mode*. Certain instructions are privileged, and can only be executed when the processor is in supervisor mode. Any attempt to execute a privileged instruction in user mode causes a trap.

Typically, application programs run in user mode, while operating systems run in supervisor mode. On reset, the processor is in supervisor mode. To enter user mode, software must clear a bit in the Processor State Register. The processor enters supervisor mode from user mode only when a hardware reset, an interrupt, or a trap occurs.

1.3.2 Memory Organization

The processor can directly address up to 1 Terabyte of memory, organized into 256 address spaces of 4 GB each. Every external access involves an 8-bit Address Space Identifier (ASI), as well as a 32-bit address. The ASI selects one of the address spaces, and the 32-bit address selects a location within that space.

The use of four of the address spaces are defined in the SPARC architecture: the User Instruction, Supervisor Instruction, User Data, and Supervisor Data spaces. SPARClite defines additional address spaces, which are used for memory-mapped control registers and for the data and instruction caches; two further address spaces are reserved for hardware debug. The remaining spaces are application-definable; any of them can be used for either data memory or I/O. All I/O is memory-mapped. The organization of the entire addressable range is illustrated in Figure 1-1.

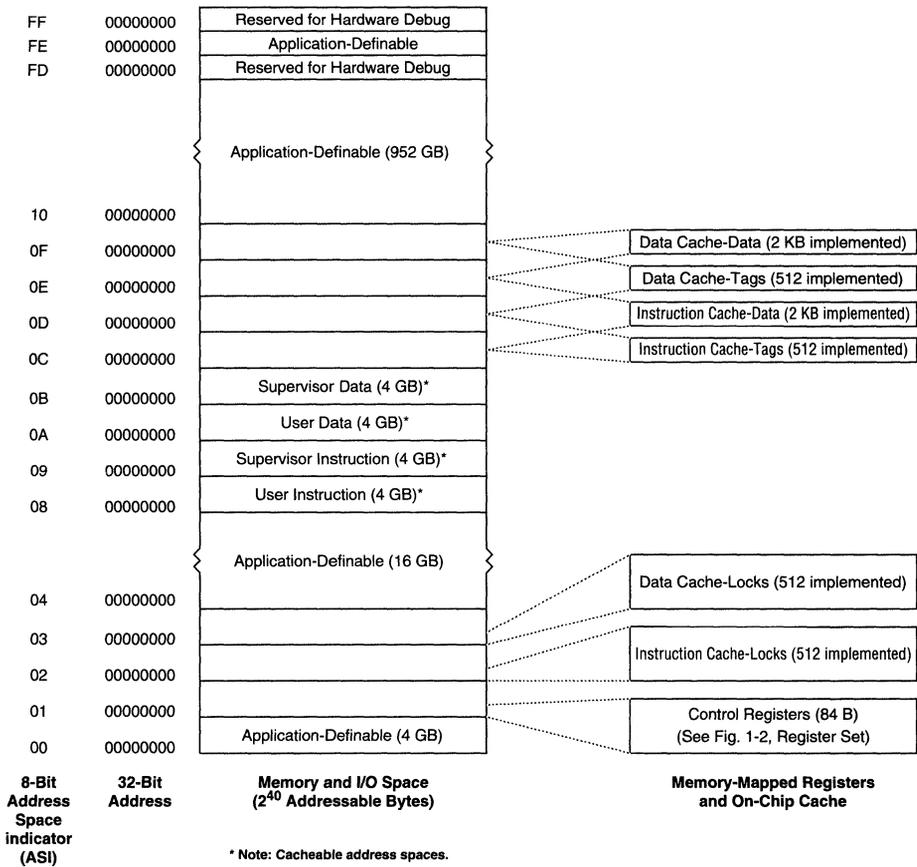


Figure 1-1. Address-Space Organization

Loads and stores are the only instructions that cause external accesses. Versions of these instructions exist for transferring bytes, half-words, words and double words between external memory (or I/O) and processor registers. In user mode, only the user instruction and data spaces are accessible; accessing any of the remaining 254 address spaces requires the processor to be in supervisor mode.

The MB86930 processor does not contain memory-management hardware; virtual-address translation can be handled by software, or by an external memory-management unit with the on-chip caches disabled.

1.3.3 Registers

All registers are 32 bits wide. There are *general-purpose registers*, whose contents have no pre-assigned meaning, and *special-purpose registers*, which contain control and status information or special data values. Some of the special-purpose registers are defined in the SPARC architecture; the rest are SPARC*lite*- or device-specific. The non-SPARC special-purpose registers are memory-mapped. The general-purpose registers, and the special-purpose Y Register, are the only ones which can be accessed in user mode. The register set is illustrated in Figure 1-2.

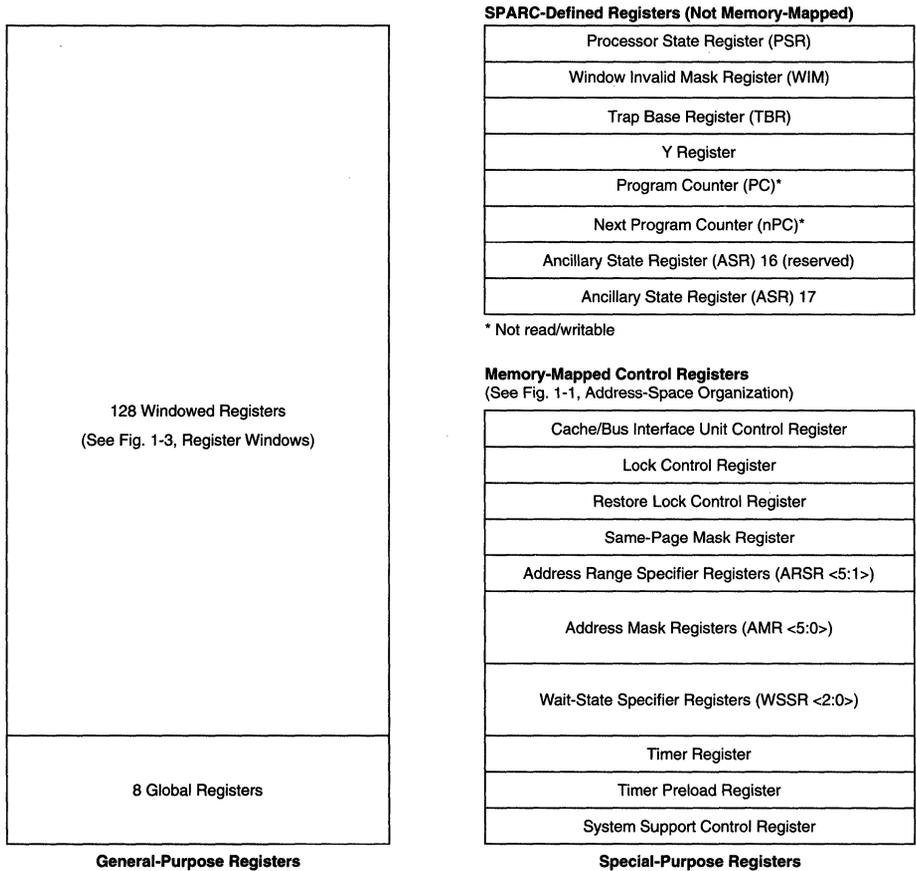


Figure 1-2. Register Set

General-Purpose Registers

In the MB86930, there are 136 general-purpose registers; 8 of these are *global registers*; the other 128 are divided into 8 overlapping blocks, or *windows*. Each

window contains 24 registers. Of these, 8 are *local* to the window, 8 are “out” registers shared with the adjacent window below, and 8 are “in” registers shared with the adjacent window above. This organization is illustrated in Figure 1-3.

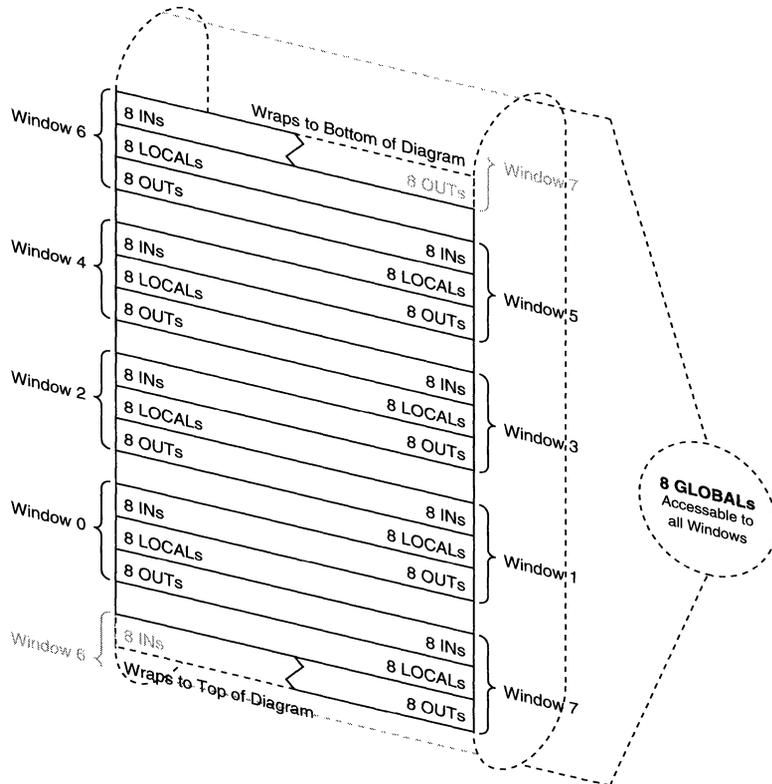


Figure 1-3. Register Windows

At any given time, 32 general-purpose registers can be accessed directly: the 8 global registers, and the 24 registers of the currently active window. The value in the Current Window Pointer (CWP) field of the Processor State Register (PSR) determines which window is active.

The overlap between adjacent windows makes it easy to pass parameters to a subroutine. Values to be passed are written to the “out” registers of the current window, which are the same as the “in” registers of the adjacent window. A SAVE instruction can then be used to decrement the Current Window Pointer, making the parameter values available to the subroutine without moving any data. A RESTORE instruction can be used to increment the CWP upon return

from the subroutine. In effect, the general-purpose registers cache the top portion of the run-time stack.

The window overlap also speeds interrupt handling, as interrupts automatically decrement the CWP, giving the interrupt routing its own window. The SPARC architecture requires a free window to be available to handle these traps.

Special-Purpose Registers

The special-purpose registers include the control and status registers defined by the SPARC architecture, plus a collection of memory-mapped registers which control peripheral functions.

Special instructions exist for reading and writing each of the SPARC control and status registers, except for the Program Counter and the Next Program Counter. The Y Register can be read and written in user mode; the instructions that access the other SPARC-defined registers are privileged.

The memory-mapped registers can be read and written with the alternate-space load and alternate-space store instructions, which are also privileged.

The SPARC-defined registers, shown in Figure 1-2 above, are:

- Processor State Register (PSR)—The primary processor control and status register. It contains *mode* fields, which are set by the operating system to configure the processor, and *status* fields, which are set by the processor to indicate the effects of instruction execution.
- Window Invalid Mask Register (WIM)—Used by software to detect the occurrence of register file underflows and overflows. It contains one mask bit for each register window. If an operation which normally increments or decrements the Current Window Pointer would cause the CWP to point to a window whose corresponding WIM bit equals 1, a trap occurs.
- Trap Base Register (TBR)—Contains three fields used by the processor to generate the address of the service routine when an interrupt or trap occurs.
- Y Register—Used in stepwise multiplication and division routines based on the MULSc and DIVSc instructions. Also used for integer multiply operations.
- Program Counter (PC)—Contains the word address of the instruction currently being executed by the Integer Unit. The PC cannot be directly read or written.
- Next Program Counter (nPC)—Contains the word address of the next instruction to be executed, assuming that no trap occurs. The nPC cannot be directly read or written.
- Ancillary State Registers (ASR[31:1])—The SPARC definition includes 31 Ancillary State Registers, 15 of which (ASR[15:1]) are reserved for future use.

The remaining ASR's can be defined and used in any way by SPARC implementations. SPARClite defines the following ASR:

ASR17— Used to enable and disable single-vector trapping. (When this feature is enabled, all traps vector to a single location.) Single vector trapping provides a small memory alternative to the standard 1K word trap table.

The memory-mapped SPARClite-specific registers, shown in Figure 1-2, are:

- Cache/Bus Interface Unit Control Register—Controls the operation of the data and instruction caches, and the write and prefetch buffers of the Bus Interface Unit.
- Lock Control Register—Controls the locking of individual entries in the data and instruction caches.
- Restore Lock Control Register—Enables or disables the restoration of the Lock Control Register upon return from an interrupt or a hardware trap.
- Same-Page Mask Register—Controls the operation of the same-page detection logic by specifying which bits of the current ASI and address are to be compared with those of the previous ASI and address.
- Address Range Specifier Registers (ARSR[5:1])—Control the assertion of the Chip-Select outputs (–CS[5:1]). –CS_n is asserted when the value on the address bus falls in the address range specified by ARSR_n. –CS₀ is asserted on accesses to the lowest address range in Supervisor Instruction Space.
- Address Mask Registers (AMR[5:0])—AMR_n controls the comparison of the current address with ARSR_n by specifying which bits are to be compared and which are “don't cares.”
- Wait-State Specifier Registers (WSSR[2:0])—Determine, for each address range, the number of clock cycles between the time an address in that range appears on the address bus and the time the processor automatically generates the –READY signal. This makes it possible for memory and I/O devices with different access times to be connected to the processor without additional logic.
- Timer Register—Contains the current timer count.
- Timer Pre-Load Register—Contains the value which is loaded into the timer when the timer overflows.
- System Support Control Register—Enables or disables same-page detection, chip-select, programmable wait-states, and the timer, independently of one another.

1.3.4 Data Types

SPARClite instructions support the Signed Integer, Unsigned Integer, and Tagged data formats of the SPARC definition. The Integer types are supported in byte (8-bit), half-word (16-bit), word (32-bit), and double-word (64-bit) widths. The

Tagged type is one word (32 bits) in width. Hardware support is not provided for the floating-point types; these can be handled in software.

1.3.5 Instructions

SPARClite provides an upward-compatible superset of the SPARC (version 8) instruction set. The additional instructions—integer divide-step, and scan for first changed bit — are supported for the sake of higher performance in embedded applications. Table 1-1 lists the SPARClite instruction set. In the MB86930 processor, the floating-point and coprocessor instructions defined in the SPARC architecture are trapped for software emulation.

Each instruction is a single 32-bit word. The instruction set can be divided into five functional groups:

1. *Logical*—Bit-wise boolean operations. Each logical instruction comes in two versions: one leaves the integer condition codes in the Processor State Register unchanged; the other changes the condition codes as a side effect.
2. *Arithmetic and Shift*—Integer arithmetic, logical and arithmetic shifts. Besides the standard arithmetic operations, SPARC provides instructions to perform tagged arithmetic. In tagged arithmetic, the two least-significant bits of each operand are used to indicate the (user-defined) data type of the operand. The tagged arithmetic instructions set a condition code if the tag of an operand is not zero.

Besides the arithmetic instructions defined in the SPARC architecture, SPARClite provides:

- A *divide-step* instruction, which can be used to construct efficient iterative integer division algorithms.
- A *scan* instruction, which determines the first bit in a word which differs from the most-significant bit. The scan instruction can be used to simplify and accelerate many important operations, like normalizing numbers with redundant sign bits.

Most of the arithmetic instructions come in two versions: one of them leaves the integer condition codes unchanged, while the other changes the condition codes as a side effect of execution.

3. *Control Transfer*—Branches, calls, jumps, returns from trap, and conditional traps. The target address of the control transfer is computed either by adding a specified offset to the value in the Program Counter, or by adding two source operands. The transfer of control either occurs immediately after the control transfer instruction, or is delayed for one further instruction.
4. *Load and Store*—External accesses. Load and store are the only instructions that read and write to external devices (including memory). Bytes, half-words, words and double words can be transferred to and from processor registers.

Special instructions access alternate address spaces. Attempts at unaligned accesses are trapped, and must be carried out under software control.

5. *Read and Write Control Registers*—Access the Program State Register, Window-Invalid Mask Register, Trap-Base Register, Y Register, and Ancillary State Registers. There are also instructions for incrementing and decrementing the Current Window Pointer. With one exception, writes to the control registers are delayed for three instruction cycles. The three instructions following a write, therefore, should not attempt to use or modify the values written. A write to the Y Register, however, is not delayed: it is completed before the next instruction is executed.

Table 1-1: Instruction Set

Group	Opcode	Name
Logical	AND (ANDcc)	And (and modify cc)
	ANDN (ANDNcc)	And Not (and modify icc)
	OR (ORcc)	Inclusive-Or (and modify icc)
	ORN (ORNcc)	Inclusive-Or Not (and modify icc)
	XOR (XORcc)	Exclusive-Or (and modify icc)
	XNOR (XNORcc)	Exclusive-Nor (and modify icc)
Arithmetic	ADD (ADDcc)	Add (and modify icc)
	ADDX (ADDXcc)	Add with Carry (and modify icc)
	TADDcc (TADDccTV)	Tagged Add and modify icc (and Trap on overflow)
	SUB (SUBcc)	Subtract (and modify icc)
	SUBX (SUBXcc)	Subtract with Carry (and modify icc)
	TSUBcc (TSUBccTV)	Tagged Subtract and modify icc (and Trap on overflow)
	MULScc	Multiply Step and modify icc
Shift	SMUL	Signed Multiply
	UMUL	Unsigned Multiply
	SMULcc	Signed Multiply (and modify icc)
	UMULcc	Unsigned Multiply (and modify icc)
	DIVScc	Divide-Step (and Modify icc)
	SCAN	Scan for bit different than MSB
Control Transfer	SLL	Shift Left Logical
	SRL	Shift Right Logical
	SRA	Shift Right Arithmetic
Control Transfer	Bicc	Branch on integer condition codes
	CALL	Call
	JMPL	Jump and Link
	RETT	Return from Trap
Control Transfer	Ticc	Trap on integer condition codes

Table 1-1: Instruction Set (Continued)

Group	Opcode	Name
Load and Store	LDSB (LDSBA) LDSH (LDSHA) LDUB (LDUBA) LDUH (LDUHA) LDD (LDDA)	Load Signed Byte (from Alternate space) Load Signed Halfword (from Alternate space) Load Unsigned Byte (from Alternate space) Load Unsigned Halfword (from Alternate space) Load Doubleword (From Alternate space)
	STB (STBA) STH (STHA) ST (STA) STD (STDA)	Store Byte (into Alternate Space) Store Halfword (into Alternate space) Store Word (into Alternate space) Store Doubleword (into Alternate space)
	LDSTUB (LDSTUBA) SWAP (SWAPA)	Atomic Load-Store Unsigned Byte (in Alternate space) Swap r Register with Memory (in Alternate space)
	SAVE RESTORE	Save caller's window Restore caller's window
	SETHI	Set High 22 bits of r register
Read and Write Control Registers	RDY RDPSR RDWIM RDTBR RDASR	Read Y register Read processor State Register Read Window invalid Mask Register Read Trap Base Register Read Ancillary State Register
	WRY WRPSR WRWIM WRTBR WRASR	Write Y register Write processor State Register Write Window invalid Mask Register Write Trap Base Register Write Ancillary State Register
	UNIMP	Unimplemented instruction

1.3.6 Data and Instruction Caches

Each member of the SPARClite family contains separate data and instruction caches on-chip. In the MB86930 processor, each cache is 2 Kbytes in size, organized into two banks of sixty-four 4-word lines. Each cache line has a 22-bit *address tag*, which indicates the memory location to which the line is currently mapped. A cache line, together with its address tag and status bits, is often called a *cache entry*. The organization of each cache is *two-way set associative*; that is, each address in memory can be mapped to either of two locations in the cache.

There are three modes of cache operation: *normal*, *global locking*, and *local locking*. In normal mode, when the integer unit requests a read to a data or instruction address which is not found in the appropriate cache, the memory block containing the requested address is read into the cache, replacing one of the current cache entries. The locking modes prevent either an entire cache, or just selected entries, from being over written in this way. The locking modes thus allow time-critical routines to be locked into cache. Thanks to the set-associative organization, as

much as one whole bank of a cache can be locked while the remaining bank continues to operate as a direct-mapped cache.

In normal mode, the data cache uses a write-through update policy, and allocates a cache entry only on a load. Writes to locked data entries, however, are not written through to main memory. In this way, a portion of the data cache can be used as fast on-chip RAM which is not mapped to external memory.

Cache tags and data are memory-mapped, and can be directly read and written using the alternate-space load and store instructions. These instructions are privileged.

Subsequent chapters discuss the cache in greater detail: *Programmer's Model* discusses cache locking; *Programming Considerations* contains hints for using the on-chip cache to best advantage.

1.3.7 Interrupts and Traps

In this manual, we distinguish between *interrupts*—which are initiated by external interrupt signals, asynchronously with respect to processor operations, and *traps*—which are caused by instructions, and so are necessarily synchronous. During system operation, external interrupts are generally unavoidable; traps, however, can and should be kept to a minimum by careful software design and testing.

Interrupt response time is critical in many embedded applications. The total response time includes the time required for the processor to finish its current task after recognizing an interrupt, and the time required to switch contexts (if necessary) and begin executing the interrupt service routine. In the SPARClike family, non-interruptible multi-cycle events are minimized, (i.e., Cache refills which take multiple cycles to completely fill a cache line, are designed so they can be interrupted after every word load). This reduces both average and maximum interrupt latency. When an interrupt is detected, the processor switches to a new window. In this way, the current values in the general-purpose registers don't have to be saved before interrupt service begins. Furthermore, service routines can be locked into the cache, making them available for immediate access.

The MB86930 processor provides direct support for 15 distinct interrupt priority levels; each level can service multiple interrupt sources. Supervisor-mode software can mask up to 14 of these levels; the highest level is non-maskable (if ET=1).

An interrupt or trap (other than reset) causes control to be transferred to an address generated by the Trap Base Register. One field in the TBR contains the base address of the trap dispatch table. Normally, an 8-bit *trap type number* serves as an offset into this table. When *single-vector trapping* is enabled, however, control

passes to the base address of the trap table (with $tt=0$), regardless of the trap type. Reset always traps to address 0.

Up to 256 trap types can be distinguished on the basis of the 8-bit trap type number. Of these, half are reserved for hardware interrupts and traps; all but one of the others are programmer-initiated (see the discussion of the Ticc instruction in the *Programmer's Model* chapter). One trap type is defined in SPARClite to support in-circuit emulation. The various trap types are listed, in order of priority, in Table 1-2.

Table 1-2: Trap Types and Priorities

Trap	Priority	tt
reset	1	—
instruction_breakpoint	1.5	255
data_breakpoint	1.5	255
instruction_access_exception	2	1
privileged_instruction	3	3
illegal_instruction	4	2
fp_disabled	5	4
cp_disabled	5	36
window_overflow	6	5
window_underflow	7	6
mem_address_not_aligned	8	7
data_access_exception	10	9
tag_overflow	11	10
trap_instruction (Ticc)	12	128 255
interrupt_level_15	14	31
interrupt_level_14	15	30
interrupt_level_13	16	29
interrupt_level_12	17	28
interrupt_level_11	18	27
interrupt_level_10	19	26
interrupt_level_9	20	25
interrupt_level_8	21	24
interrupt_level_7	22	23
interrupt_level_6	23	22
interrupt_level_5	24	21
interrupt_level_4	25	20
interrupt_level_3	26	19
interrupt_level_2	27	18
interrupt_level_1	28	17

The expression *trapped instruction* refers, in the case of a synchronous trap, to the instruction which caused it. In the case of an interrupt, the trapped instruction is the one which was about to execute when the interrupt occurred.

The Integer Unit supports *precise traps*—when an interrupt or trap occurs, the saved state of the processor reflects the completion of all instructions prior to the trapped instruction, but no subsequent instructions (including the trapped

instruction). Hardware guarantees that upon return from the service routine, the Program Counter points to the trapped instruction or the following instruction if the trapped instruction was emulated.

1.4 Internal Architecture

The internal architecture of SPARClite family processors is illustrated in Figure 1-4. The processor core consists of an Integer Unit which implements a superset of the SPARC integer instruction set. Separate on-chip caches are provided for data and instructions. The Bus Interface Unit handles the interface between the processor and the system. A Clock Generator with built-in phase-locked loop simplifies system clock design. Finally, the Debug Support Unit provides hardware support for in-circuit emulation. Internally, the various functional units are connected by separate instruction and data buses. For connection with external memory and I/O, a unified address bus and a unified data bus are extended off-chip. The main functional units are discussed briefly below, and more fully in the *Internal Architecture* chapter.

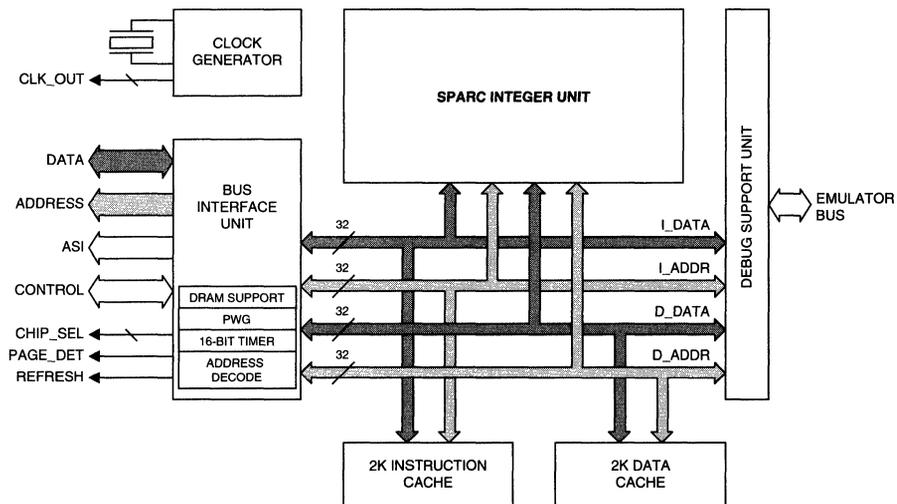


Figure 1-4. Internal Architecture (Block Diagram)

1.4.1 Integer Unit

The Integer Unit (IU) is a compact, fully custom implementation of the SPARC architecture. The IU is hard-wired for high performance. Its internal functional units are designed around a modular architecture and can be customized to meet different application requirements. In the MB86930, for example, this flexibility

was used to provide direct hardware support for integer multiplication, and to extend the SPARC instruction set by supporting divide-step and scan instructions.

The IU implements a five-stage instruction pipeline to allow a sustained execution rate of nearly one instruction per cycle. The operation of the pipeline under ideal conditions is illustrated in Figure 1-5. The pipeline consists of the following stages:

- Fetch (F)—One of the instruction memory spaces is addressed and returns an instruction. (Figure 1-5 below assumes a hit in the instruction cache.)
- Decode (D)—The instruction is decoded; the register file is addressed and returns operands.
- Execute (E)—The ALU computes a result.
- Memory (M)—External memory is addressed (for load and store instructions only; this stage is idle for other instructions).
- Writeback (W)—The result (or loaded memory datum) is written into the register file.

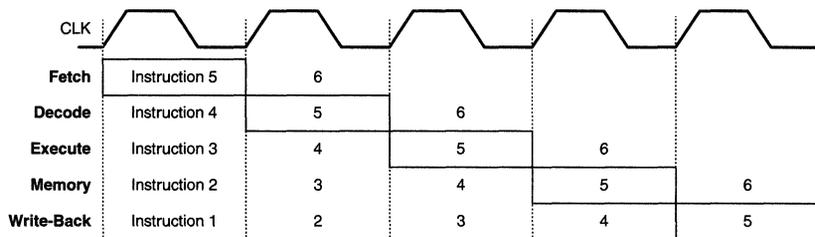


Figure 1-5. Instruction Pipeline

No instructions execute out-of-order; that is, if instruction A enters the pipeline before instruction B, then instruction A necessarily reaches the writeback stage before instruction B does. Conditions which hold up the pipeline, and the effect of traps on pipeline operations, are discussed in the *Internal Architecture* chapter.

1.4.2 Data and Instruction Caches

The on-chip data and instruction caches allow designers to build high-performance systems without incurring the cost of fast external memory and the associated control logic.

In the MB86930 processor, each cache is 2 Kbytes in size, organized into two banks of sixty-four 16-byte lines. Cache lines are refilled in 4-byte increments to avoid the interrupt latency incurred by long, uninterruptible cache line replacements.

The data and instruction caches are accessed independently over separate data and instruction buses, allowing data to be loaded from and stored to cache concurrently with instruction fetches.

1.4.3 Bus Interface Unit

The Bus Interface Unit (BIU) contains the logic which allows the processor to communicate with the system. The BIU receives requests for external memory and I/O accesses from the cache control logic. When the BIU performs a read, it returns the data to both the cache and the IU. Parallel paths make the data available to the IU in the same cycle that it is written to the cache.

The BIU has a one-word (32-bit) write buffer to hide external memory latency from the IU. The BIU also has a one-word prefetch buffer for instruction fetches. These buffers are enabled or disabled by bits in the Cache/Bus Interface Unit Control Register.

1.4.4 Debug Support Unit

The Debug Support Unit supports hardware emulation with on-chip breakpoint and single-step logic. A dedicated emulator bus is extended off-chip from the debug unit; the emulator bus makes it possible to trace transactions between the Integer Unit and on-chip cache.

1.5 External Interface

The processor's external interface consists of signals, bus operations, and system support functions. This section gives an overview; details are discussed more fully in the *External Interface* chapter. The *System Design Considerations* chapter discusses issues that are likely to arise in the design of any SPARClite system.

1.5.1 Signals

The processor's external signals, illustrated in Figure 1-6, can be grouped by function:

- Processor Control and Status—Reset, error, and clock signals.
- Memory Interface—Data and address buses, ASI and byte-enables, chip-selects, and other control signals used to access external memory and memory-mapped devices.
- Bus Arbitration—Signals used by external devices in requesting, and by the processor in granting, control of the bus.
- Peripheral Functions—Interrupt-requests and timer overflow.
- Emulator Bus—Signals to support in-circuit emulation.

- Boundary-Scan—Test signals used for hardware verification.

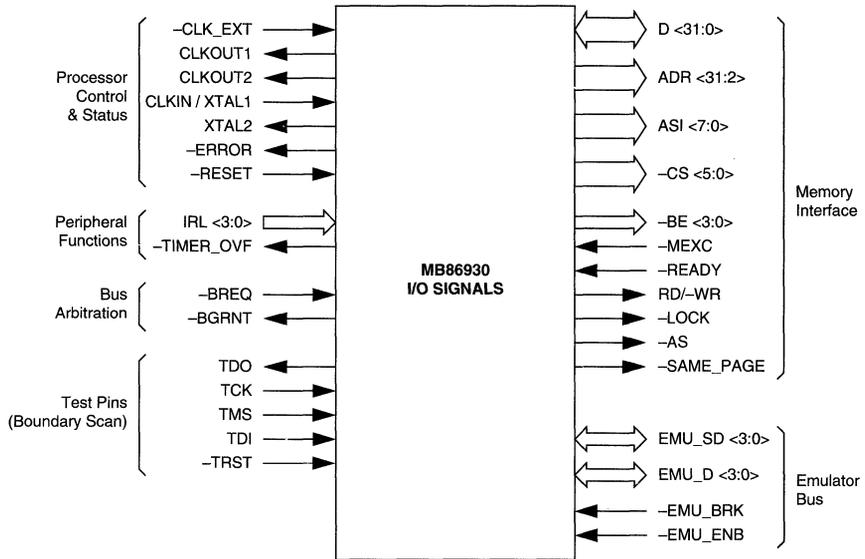


Figure 1-6. Input and Output Signals

1.5.2 Bus Operation

At any given time, the Bus Interface Unit is handling requests for external memory and I/O operations, arbitrating for bus access, or idle. From the point of view of the external system, bus transactions are handled in fairly standard ways:

- **Memory and I/O Operations**—Read and write transactions are initiated with the BIU asserting the $\overline{\text{AS}}$ signal. The $\text{RD}/\overline{\text{WR}}$ output indicates the transaction type. The $\overline{\text{BE}}[3:0]$ outputs indicate the transaction width. The BIU drives the address and ASI signals, and either drives (on stores) or reads (on loads) the signals on the data bus. The transaction ends when the external system or programmable wait-state generator asserts $\overline{\text{READY}}$.

An atomic load-store is executed as a load followed by a store, with no operation allowed in between. The $\overline{\text{LOCK}}$ output is asserted to indicate that the bus is being used for more than one consecutive memory operation.

- **Arbitration**—Any external device can request ownership of the bus by asserting the $\overline{\text{BREQ}}$ signal. The BIU three-states its bus drivers and asserts $\overline{\text{BGRNT}}$ to indicate that it is relinquishing control of the bus. On completion of its transaction, the external device de-asserts $\overline{\text{BREQ}}$; the BIU responds by de-asserting $\overline{\text{BGRNT}}$ in the following cycle.

The *External Interface* chapter gives further details concerning bus operations, with timing diagrams, a bus state diagram, and a discussion of transactions that are interrupted by exceptions.

1.5.3 System Support Functions

Built-in system support functions help to minimize the amount of glue logic required in the external system. The support includes a set of system-configuration registers, a timer for generating refresh requests, and same-page detection logic.

The system-configuration registers (Address Range Specifiers, Address Masks, and Programmable Wait-State Specifiers) allow software to define six different address ranges. When an address driven by the processor is in one of these ranges, the corresponding Chip-Select ($-\text{CS}$) pins are asserted. After a number of clock cycles determined by the corresponding Programmable Wait-State Specifier, the processor automatically generates the $-\text{READY}$ signal. This makes it possible for memory and I/O devices with different access times to be connected to the processor without additional logic.

The programmable timer causes the $-\text{TIMER_OVF}$ output signal to be asserted at software-defined intervals. This signal can be used to initiate DRAM refresh cycles, or to control other periodic events in the external system.

The same-page detection logic determines whether the address of the current memory transaction is on the same page as the previous transaction. If it is, the processor asserts the $-\text{SAME_PAGE}$ signal. The system can then take advantage of the fast consecutive accesses possible within the page boundaries of fast-page mode DRAM.

1.6 Development-Support Tools

A full range of development tools are available to support the development of your SPARC*lite* application. The emergence of SPARC as the industry standard engineering workstation architecture provides a fully supported and cost effective source of native development environments. Furthermore, tools targeted at embedded systems development are available as well.

Solutions are available to meet your emulation, logic analysis, logic modeling, architectural simulation, real-time operating system, PC environment, benchmarking and prototyping requirements. Call the SPARC*lite* customer hotline for a complete list of support solutions.



Programmer's Model

This chapter presents the SPARC^{lite} processor architecture as a collection of resources available to software. It discusses the user and supervisor modes, the organization of the address space, the processor registers, the supported data types, the instruction set, the on-chip caches, interrupts and traps and debug support. A separate section describes the internal state of the processor after reset.

The *Programming Considerations* chapter contains information about how to use these processor resources to best advantage.

2.1 Program Modes

The SPARC architecture provides two mutually exclusive modes of program execution, *user mode* and *supervisor mode*. The processor is in supervisor mode when the S bit of the Processor State Register (PSR) is 1, and in user mode when this bit is 0. Instructions which access either special-purpose registers or alternate memory spaces are privileged; the use of *privileged* instructions is restricted to supervisor mode.

The distinction between user and supervisor modes provides system protection in multitasking environments. System code runs in supervisor mode and has full access to processor resources, while application code runs in user mode and is kept from having unwanted side effects. Embedded systems connected to a network can use a protection scheme based on the distinction between user and supervisor modes. In such a scheme, network service routines intended to have

system-wide effects run in supervisor mode. Routines intended to have only local effects, on the other hand, run in user mode.

In many embedded systems, however, this hierarchy is not required, and the processor can operate exclusively in supervisor mode. In this way, application code can directly manipulate the Current Window Pointer (in the PSR) and other processor control fields.

On reset, the processor is in supervisor mode. To enter user mode, software must clear the S bit in the PSR. The processor enters supervisor mode from user mode only when a hardware reset, an interrupt, or a trap occurs. A return from trap (RETT) instruction restores the value the S bit had before the trap was taken.

2.2 Memory Organization

The processor can directly address up to 1 Terabyte of memory, organized into 256 address spaces of 4 GB each. These address spaces may or may not overlap in physical memory, depending on the system design. Every external access involves an 8-bit Address Space Identifier (ASI) as well as a 32-bit address. The ASI selects one of the address spaces, and the address selects a word within that space (see Table 2-1). Only the user instruction and data spaces are available in user mode; accessing any of the other 254 address spaces requires the processor to be in supervisor mode.

Table 2-1: ASI Address Space Map

ASI <7:0>	Address Space
0x1	Control Register
0x2	Instruction Cache Lock
0x3	Data Cache Lock
0x4 - 0x7	Application Definable
0x8	User Instruction Space
0x9	Supervisor Instruction Space
0xA	User Data Space
0xB	Supervisor Data Space
0xC	Instruction Cache Tag RAM
0xD	Instruction Cache Data RAM
0xE	Data Cache Tag RAM
0xF	Data Cache Data RAM
0x10 - 0xFC, 0xFE	Application Definable
0xFD, 0xFF	Reserved for Debug Hardware

Loads and stores are the only instructions that cause external accesses. Versions of these instructions exist for transferring bytes, half-words, words and double

words between memory (or I/O) and processor registers. Addressing conventions for external accesses are “big-endian”:

- *Bytes*—Increasing the address decreases the significance of a byte within the word. That is, the most significant byte of a word—the “big end” of the word—is accessed when bits [1:0] of the address are both 0. The least significant byte is accessed when address bits [1:0] are both 1.
- *Halfwords*—The most significant halfword of a word is accessed when bit 1 of the address is 0, and the least significant halfword when address bit 1 is 1.
- *Doublewords*—The most significant word of a doubleword is accessed when bit 2 of the address is 0, and the least significant word when address bit 2 is 1.

The address of a halfword, word, or doubleword is the address of its most significant byte. The addressing conventions are illustrated in Figure 2-1.

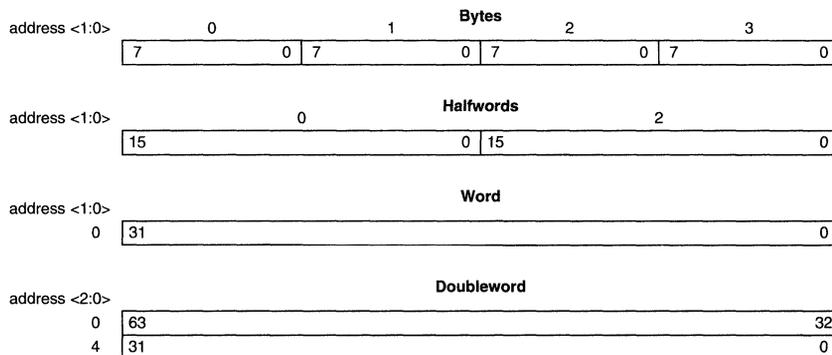


Figure 2-1. Addressing Conventions

Load and store operations require proper alignment of data in memory. An aligned doubleword address is divisible by 8, an aligned word address is divisible by 4, and an aligned half-word address is divisible by 2. If a load or store instruction generates an improperly aligned address, a `memory_address_not_aligned` trap occurs, and the access must be performed piecemeal under software control.

The processor does not contain memory-management hardware; virtual-address translation can be handled by software, or by an external memory-management unit.

2.3 Registers

There are two types of registers: the *general-purpose*, or *r* registers, whose contents have no pre-assigned meaning, and the *special-purpose registers*, which contain

control and status information, or special-purpose data. All registers are 32 bits wide. The register set is illustrated in Figure 1-2 of the *Overview* chapter.

The general-purpose (r) registers can be accessed in user mode. There are 136 r registers; 8 of them are *global registers*; the other 128 are divided into 8 overlapping blocks, called *windows*. The windowing system, and the special uses of certain r registers, are discussed below.

The special-purpose registers are of two kinds: (1) registers defined by the SPARC architecture, and (2) memory-mapped registers which control peripheral functions. Special instructions exist for reading and writing each of the SPARC registers, except for the Program Counter and the Next Program Counter. The memory-mapped registers can be read and written with the alternate-space load and store instructions. Except for reads and writes to the SPARC-defined Y register, all of the instructions which access special-purpose registers are privileged.

Some of the special-purpose have reserved or undefined fields. Therefore, one should not assume particular values for these fields when reading the registers, and one should not assume that only non-reserved fields in the registers are read. In general, it is good practice to write zeros to unused or reserved fields, and to mask reserved fields after reading the registers.

2.3.1 Register Windows

As specified by the SPARC architecture, the general-purpose register set is organized into a set of 8 global registers, plus a collection of overlapping windows. In the MB86930, there are 8 such windows. Each window contains 24 registers. Of these, 8 are *local* to the window, 8 are "*out*" registers shared with the adjacent window below, and 8 are "*in*" registers shared with the adjacent window above. This organization is illustrated in Figure 2-2.

At any given time, 32 general-purpose registers can be accessed directly: the 8 global registers, and the 24 registers of the currently active window. The value in the Current Window Pointer (CWP) field of the Processor State Register (PSR) determines which window is active. (See Section 5.3 for register addressing conventions.)

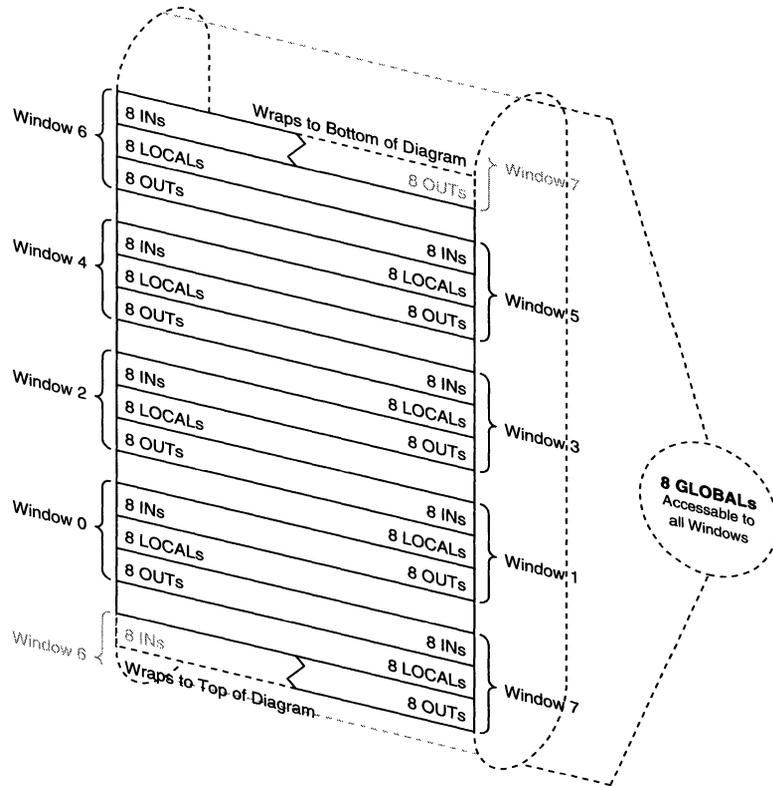


Figure 2-2. Register Windows

Register Addressing

There are up to three address fields associated with a SPARC instruction. In the case of a three-address instruction, these are the *rs1* field, the *rs2* field, and the *rd* field. *Rs1* and *rs2* are the *logical register addresses* of the two source operands of the instruction while *rd* is the logical register address of the destination operand.

These addresses specify the location of the operands within the context of the current window, as shown in Table 2-2.

Table 2-2: Logical Register Addressing

Addresses	Registers
r[0] - r[7]	global[0] - global[7]
r[8] - r[15]	out[0] - out[7]
r[16] - r[23]	local[0] - local[7]
r[24] - r[31]	in[0] - in[7]

The CWP field of the PSR register points to the current window. The combination of a logical register address with the CWP produces a *physical register address*. Physical register addresses are directly decoded by the Register File. Doubleword operands in the register file are assumed to have even-odd alignment. The even numbered register contains the most significant 32 bits of the doubleword. Instructions which act on doublewords must specify even-numbered register addresses.

Since the CWP is part of the PSR register it is possible to change the value of the CWP with software. In particular, the WRPSR, SAVE, RESTORE, and RETT instructions can change the CWP. See the *Instructions* section below for details. Hardware also can change the CWP when a trap or interrupt occurs. See the *Traps and Interrupts* section.

Performance Features

The overlap between adjacent windows makes it easy to pass parameters to a subroutine. Values to be passed should be written to the "out" registers of the current window, which are the same as the "in" registers of the adjacent window. A SAVE instruction can then be used to decrement the Current Window Pointer, making the parameter values available to the subroutine without moving any data.

Register windows improve performance in embedded applications because they function as local variable caches which retain either interrupt, subroutine, context or operating system variables with no additional overhead. Since procedure calls are efficient, optimizing compilers are not forced to replace them with inlined macros; this reduces the size of the compiled code, saving memory space, and making it possible to fit more complicated routines in the instruction cache.

Register windows can be dedicated to individual contexts to enable very fast switching between contexts. When handling interrupts, the hardware immediately moves to the adjacent window to start executing the service routine. In this way, an unused set of registers is made available in less than 3 processor cycles.

Each register in the register file has three read-only and one write-only port. The four-port structure allows even store instructions—which may require three operands to be read out of the register file—to be completed in a single cycle.

2.3.2 Special Uses of the r Registers

Four of the r registers have special uses defined in the SPARC architecture:

- When global register 0 (r[0]) is addressed as a source operand, the constant value 0 is read. When r[0] is used as a destination operand, the data written is discarded, and no r register changes value.
- The CALL instruction writes its own address into out register 7 (r[15]).
- When a trap is taken, the current window pointer is decremented. The program counters PC and nPC are then automatically written into local registers 1 and 2 (r[17] and r[18]) of the *new* register window.

2.3.3 SPARC-Defined Special-Purpose Registers

The registers discussed in this section are defined as part of the SPARC architecture.

Processor State Register (PSR)

The Processor State Register is the primary processor control and status register. It contains 11 mode and status fields which configure the processor and report processor status and exception results. The *mode* fields, shown in upper case in Figure 2-3, are set by the operating system to configure the processor. The *status* fields, shown in lower case, are set by the processor to indicate the effects of instruction execution.

Except for several fields described below, the PSR can be written and read directly with the privileged instructions WRPSR and RDPSR. The PSR can also be modified by the SAVE, RESTORE, Ticc, and RETT instructions, and by any instruction that modifies the condition codes.

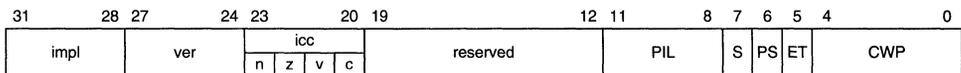


Figure 2-3. Processor State Register

Bits 31-28: Implementation (impl)—Identifies the implementation number of the processor. In the MB86930 processor, it is hardwired to 0. The value in this field cannot be changed by a WRPSR instruction.

- Bits 27-24: Version (ver)—Identifies the processor version, and is intended for factory use. It can be read, but not written. The Version field is hardwired to 2 in the MB86930 processor.
- Bits 23-20: Integer Condition Codes (icc)—Contains the negative (n), zero (z), overflow (v), and carry (c) integer condition-code flags. These bits are modified by the WRPSR instruction, and by arithmetic and logical instructions whose names end with the letters *cc* (for example, ANDcc). The Bicc (Branch on integer condition codes) and Ticc (Trap on integer condition codes) instructions transfer program control based on the values of these bits. The integer condition code flags are defined as follows:
- n (Bit 23) Set to 1 if the ALU result was negative for the last instruction that modified the icc field; equal to 0 otherwise.
 - z (Bit 22) Set to 1 if the ALU result was zero for the last instruction that modified the icc field; equal to 0 otherwise.
 - v (Bit 21) If this bit equals 1, an arithmetic overflow occurred on the last instruction that modified the icc field; it equals 0 otherwise. Logical instructions that modify the icc field always reset the overflow bit to 0.
 - c (Bit 20) If this bit equals 1, either an arithmetic carry out of bit 31 occurred on the last addition that modified the icc, or a borrow out of bit 31 occurred as the result of the last subtraction that modified the icc. The carry bit equals 0 otherwise. Logical instructions that modify the icc field always reset the carry bit to 0.
- Bits 19-12: Reserved (reserved)—This field is reserved. When you use the WRPSR instruction, this field should always be written with 0s.
- Bits 11-8: Processor Interrupt Level (PIL)—Specifies the levels of interrupt which the processor will accept. The processor accepts only interrupts with level 15 (non-maskable interrupts), or with levels higher than the value in the PIL field (maskable interrupts). Bit 11 is the most significant bit, and bit 8 is the least significant.
- Bit 7: Supervisor Mode (S)—Determines whether the processor is in supervisor mode (S=1) or user mode (S=0). Since instructions that write the PSR are available only in supervisor mode, the processor enters supervisor mode from user mode only when a reset, trap, or interrupt occurs.
- Bit 6: Prior S State (PS)—Records the value of the S bit when a trap is taken, so that the processor can return to the proper operating mode (user or supervisor) on return from the trap. Processor hardware changes the PS bit to the state of the S bit when entering a trap, and changes the S bit to the state of the PS bit when returning from the trap.
- Bit 5: Enable Traps (ET)—Enables traps (ET=1). When ET=0, traps are disabled and all interrupts are ignored.
- Bits 4-0: Current Window Pointer (CWP)—Points to the register window which is currently active. The CWP is written and read by the WRPSR and RDPSR instructions, is decremented by traps and the SAVE instruction, and is incremented by the RESTORE and RETT instructions. The SPARClite processor implements 8 out of the 32 windows allowed in the SPARC definition, so only the 3 least significant bits of the CWP field are used. Arithmetic on the CWP is always performed modulo 8. Attempting to write a value to the CWP field which points to an unimplemented window results in an “illegal instruction” error.

with all tt bits set to 0. The trap handler can read the tt field to find out the origin of the current trap.

Bits 3-0: Null (null)—This field is hardwired to 0 to force 4-word increments of the trap vector. The WRTBR instruction does not affect this field.

Y Register

The “Y Register” is composed of a number of 32-bit latches, muxes, and bus drivers which reside in the data path of the Execute Block (see the *Internal Architecture* chapter). It is used during the multiply step instruction (MULScc) to contain the multiplier and the least significant bits of the partial products as they are evaluated. It is used during the divide step instruction (DIVScc) to contain the most significant 32 bits of a 64-bit dividend and the partial remainders as they are evaluated. It is also used by the multiply unit to hold the most significant words of the partial products and, when the multiplication is completed, the high 32 bits of the 64-bit product.

The Y register can be read and written with the RDY and WRY instructions, respectively. WRY is not a “delayed write” instruction: the value written into the Y register is available to the following instruction.



Figure 2-6. Y Register

- *Multiply Step Support*—At the beginning of a multiplication algorithm which uses the MULScc instruction, the 32-bit multiplier is loaded into the Y register with a WRY instruction. When the multiplication is completed, the least significant word of the 64-bit product will be in the Y register.
- *Divide Step Support*—At the beginning of a division algorithm which uses the DIVScc instruction, the most significant word of the dividend is loaded into the Y register with a WRY instruction. At the end of the divide routine, the remainder will be in the Y register and can be read with a RDY instruction.
- *Multiply Unit Support*—The Y register is also used by the Multiply Unit (MU) during the UMUL, UMULcc, SMUL, and SMULcc instructions. The most significant word of the 64-bit product will be in the Y-Register when the multiplication completes.

Program Counter (PC)

The Program Counter contains the word address of the instruction currently being executed by the Integer Unit. The PC cannot be directly read or written.

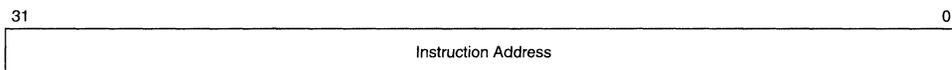


Figure 2-7. Program Counter

Next Program Counter (nPC)

The Next Program Counter contains the word address of the next instruction to be executed, assuming a trap does not occur. The nPC cannot be directly read or written.

In delayed control transfers, the instruction that immediately follows the control transfer (the delay instruction) may be executed before control is transferred to the target. (See the *Instructions section*, below.) The nPC is necessary for implementing this feature. Most instructions complete by copying the contents of the nPC into the PC, then updating the nPC. The nPC is incremented by 4, unless the instruction implies a control transfer, in which case the computed target address is written into the nPC. The PC now points to the instruction which will be executed next, while the nPC points to the instruction which will be executed after that.

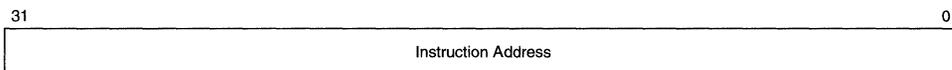


Figure 2-8. Next Program Counter

Ancillary State Registers (ASR[31:1])

The SPARC definition includes 31 Ancillary State Registers, 15 of which (ASR[15:1]) are reserved for future use. The remaining ASR's can be defined and used in any way by SPARC implementations. The MB86930 defines the following ASR:

ASR17—Used to enable and disable single-vector trapping. When this feature is enabled, all traps (except reset and breakpoint traps) vector to a single location, the base address of the trap table, as specified by the TBA field of the TBR

register (tt=0). ASR17 can be read and written with the privileged instructions RDASR and WRASR.

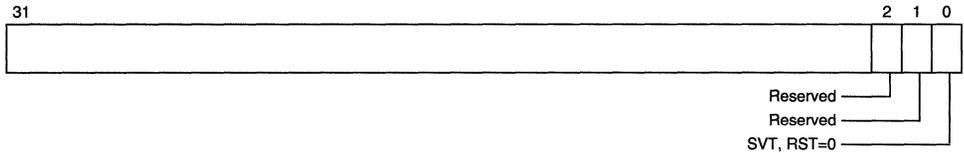


Figure 2-9. Ancillary State Register 17

- Bits 2-1: Reserved Field (reserved)—When writing to ASR17, both of these bits must be written with 0s.
- Bit 0: Single Vector Trapping (SVT)—Enables single vector trapping when set to 1. The SVT bit equals 0 at reset.

2.3.4 Memory-Mapped Control Registers

In addition to the registers defined by the SPARC architecture, the MB86930 provides a collection of memory-mapped registers which control peripheral functions. Figure 2-10 shows these registers and their locations in memory. The memory-mapped registers can be read and written with the alternate-space load and store instructions, which are privileged.

0x00000000	ASI=0x1	Cache/Bus interface Unit Control Register
0x00000004	ASI=0x1	Lock Control Register
0x00000008	ASI=0x1	Lock Control Save Register
0x0000000C	ASI=0x1	Cache Status Register
0x00000010	ASI=0x1	Restore Lock Control Register
0x00000080	ASI=0x1	System Support Control Register
0x00000120	ASI=0x1	Same-Page Mask Register
0x00000124	ASI=0x1	Address Range Specifier Registers (ARSR <5:1>)
0x00000140	ASI=0x1	Address Mask Register (AMR <5:0>)
0x00000160	ASI=0x1	Wait-State Specifier Registers (WSSR <2:0>)
0x00000174	ASI=0x1	Timer Register
0x00000178	ASI=0x1	Timer Preload Register

Figure 2-10. Locations of Memory-Mapped Control Registers

Cache/Bus Interface Unit Control Register

The Cache/BIU Control Register controls the operation of the data and instruction caches, and the write and prefetch buffers of the Bus Interface Unit. This register is located at address 0x00000000 with an ASI of 0x1.

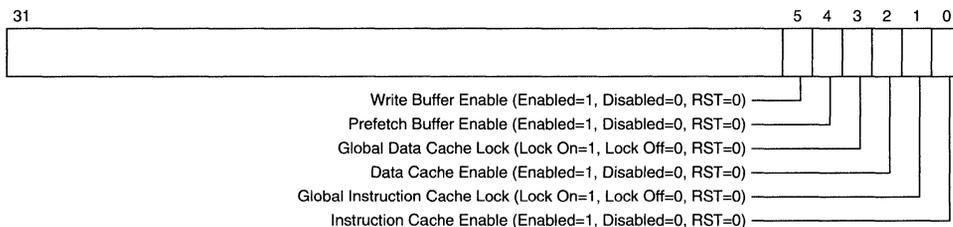


Figure 2-11. Cache/Bus Interface Unit Control Register

- Bit 5: Write Buffer Enabled—When set to 1, enables the write buffer of the BIU only if both the instruction and data caches are enabled. At reset, this bit is 0. This bit should be changed only when the instruction and data caches are off.
- Bit 4: Prefetch Buffer Enabled—When set to 1, enables the prefetch buffer of the BIU only if both the instruction and data caches are enabled. At reset, this bit is 0. This bit should be changed only when the instruction and data caches are off.
- Bit 3: Global Data Cache Lock—Locks the current entries into the on-chip data cache; with this bit set to 1, no valid entry in the data cache will be replaced. To insure the best performance with the cache locked, invalid words in allocated cache locations will be updated. On write hits, with the data cache locked, the data is not written to external memory, allowing the locked cache to be used as scratchpad RAM or a run-time stack, independent of main memory. When the Data Cache Lock bit is 0, the cache operates normally. At reset, this bit is 0.
- Bit 2: Data Cache Enable—Turns the on-chip data cache on (1) and off (0). At reset, this bit is 0.
- Bit 1: Global Instruction Cache Lock—Locks the current entries into the on-chip instruction cache; with this bit set to 1, no valid entry in the instruction cache will be replaced. To insure the best performance with the cache locked, invalid words in allocated cache locations will be updated. When this bit is 0, the cache operates normally. Writes to the Instruction Cache Lock bit do not affect cache operation for the following three instructions. At reset, this bit is 0.
- Bit 0: Instruction Cache Enable—Turns the on-chip instruction cache on (1) and off (0). Writes to the Instruction Cache Enable bit do not affect cache operation for the following three instructions. At reset, this bit is 0.

Lock Control Register

The Lock Control Register controls the locking of individual entries in the data and instruction caches. It is located at address 0x00000004 with an ASI of 0x1.

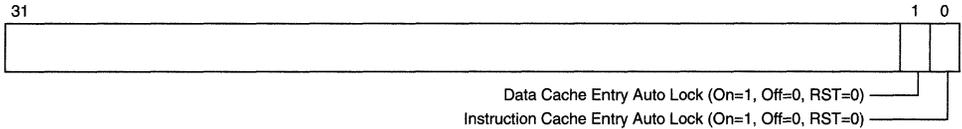


Figure 2-12. Lock Control Register

- Bit 1: Data Cache Entry Auto Lock—Enables (1) and disables (0) auto-locking for entries in the on-chip data cache. All data accessed while this bit is 1 have the lock bits in their cache tags set to 1. Writes to this bit affect all subsequent data accesses. At reset, this bit is 0.
- Bit 0: Instruction Cache Entry Auto Lock—Enables (1) and disables (0) auto-locking for entries in the on-chip instruction cache. All instructions fetched while this bit is 1 have the lock bits in their cache tags set to 1. Writes to this bit do not affect cache operation for the following three instructions. At reset, this bit is 0.

Lock Control Save Register

When an external interrupt or hardware trap occurs, the auto-locking of entries in on-chip cache is disabled. The Lock Control Save Register is used to re-enable auto-locking after the interrupt has been serviced. The register is updated with the contents of the Lock Control Register when there is a hardware interrupt, an exception condition (illegal instruction, memory data alignment error), or a DSU hardware breakpoint. The updated Lock Control Save Register is then used to restore the Lock Control Register after the interrupt or trap. This “autosave” feature allows restoration of the Lock Control Register following interrupts and traps that cannot be anticipated by software. In other cases, the program can save the Lock Control Register directly for later restoration.

The value of the Lock Control Register before the interrupt or trap is automatically saved in the Lock Control Save Register, located at address 0x00000008 with an ASI of 0x1. The correct auto-lock value is restored in the Lock Control Register by setting bit <0> in the Restore Lock Control Register to 1. This causes the value that is saved in the Lock Control Save Register to be moved to the Lock Control Register when a RETT is executed (See Section 2.6.2).

The cache does not have to be enabled for the Lock Control Save Register to be updated, and the register is both readable and writable.

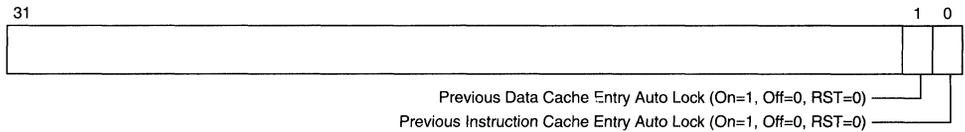


Figure 2-13. Lock Control Save Register

Restore Lock Control Register

On return from an external interrupt or hardware trap service routine, the Lock Control Register can have its previous value restored from the Lock Control Save Register. The Restore Lock Control Register, located at address 0x00000010 with an ASI of 0x1, controls this feature. When bit 0 of this register is set to 1 and a RETT instruction is executed, the value in the Lock Control Save Register is placed into the Lock Control Register.

There should be no traps between writing a 1 to bit 0 of the Restore Lock Control Register and the corresponding RETT instruction. This bit is cleared to 0 on reset, and also when a return from external interrupt or hardware trap is executed.

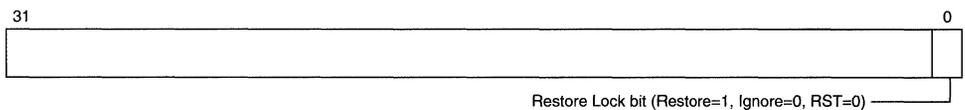


Figure 2-14. Restore Lock Control Register

Cache Status Register

If an attempt is made to lock a cache entry which is already locked, bit 0 in the Cache Status Register is set to 1. This bit can be cleared by software. The Cache Status Register is located at address 0x0000000C with an ASI of 0x1.

The Cache Status Register is meaningful only when auto-locking is utilized. In the case of writing the cache tags manually to lock cache lines (either by writing the

Tag Lock Bit address or the Cache Tag address directly), an attempt to lock a line which is already locked will not be indicated by the Cache Status Register.

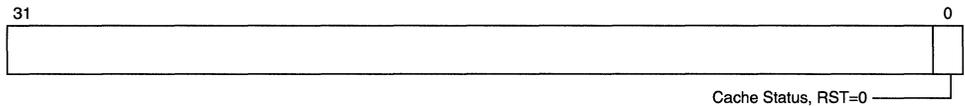


Figure 2-15. Cache Status Register

Same-Page Mask Register

The Same-Page Mask Register controls the operation of the same-page detection logic by specifying which bits of the current ASI and address are to be compared with those of the previous ASI and address. If the specified (i.e., unmasked) bits all match, then the processor recognizes the two accesses as being “in the same page,” and asserts the `-SAME_PAGE` signal. These registers should not be written if the bus interface unit will handle addresses that are affected by the change in the next 3 processor cycles. The Same-Page Mask Register is located at address `0x00000120` with an ASI of `0x1`.

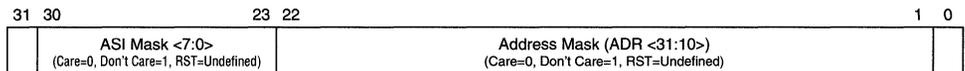


Figure 2-16. Same-Page Mask Register

- Bit 31: Reserved
- Bits 30-23: ASI Mask—Specifies which bits in the ASI of the current external access are to be compared with the corresponding bits in the ASI of the previous access. Only those bits are compared for which the mask bit is 0. Mis-matches in any other bits do not prevent the two accesses from being recognized as “on the same page.” The bits of this field are cleared to 0 on reset.
- Bits 22-1: Address Mask—Specifies which of the 22 most significant bits in the address of the current external access are to be compared with the corresponding bits in the address of the previous access. Only those bits are compared for which the mask bit is 0. Mis-matches in any other bits do not prevent the two accesses from being recognized as “on the same page.” The bits of this field are cleared to 0 on reset.
- Bit 0: Reserved

Address Range Specifier Registers (ARSR[5:1])

Values in the Address Range Specifier Registers define up to five different address ranges, which are used for various system-support functions. The ARSRs

are located in a contiguous block beginning at address 0x00000124 with ASI 0x1 (see Table 2-3).

The ARSRs, together with the Address Mask Registers, can be used to control the assertion of the Chip-Select outputs ($-\text{CS}[5:1]$). $-\text{CS}_n$ is asserted when the value on the address bus falls in the address range specified by ARSR_n and AMR_n . See the discussion of the Address Mask Registers, below. $-\text{CS}_0$ is asserted when the value on the address bus, as masked by AMR_0 , falls into the lowest range of Supervisor Instruction Space. The range of $-\text{CS}_0$ (as masked by AMR_0) is 8K words.

These registers should not be written if the bus interface unit will handle addresses that are affected by the change in the next 3 processor cycles. The user should be careful that two chip selects are never selected at the same time. A programmable wait-state generator is also associated with each address range. See the discussion of the Wait-State Specifier Registers, below.

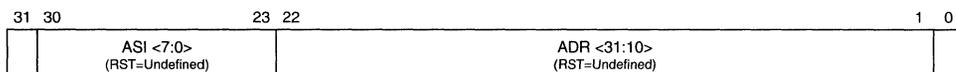


Figure 2-17. Address Range Specifier Registers

Bit 31: Reserved

Bits 30-23: ASI[7:0]—Specifies the ASI of a target address range. The value of this field is undefined on reset.

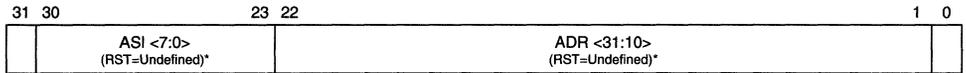
Bits 22-1: ADR[31:10]—Specifies the 22 most significant bits of a target address range. The value of this field is undefined on reset.

Bit 0: Reserved

Address Mask Registers (AMR[5:0])

AMR_n works with ARSR_n to define an address range. AMR_n specifies which bits of the currently driven ASI and address are to be compared with the contents of ARSR_n , and which bits are “don’t cares.” Except for AMR_0 , reset leaves the values in the AMR registers undefined (see Table 2-3). These registers should not be written if the bus interface unit will handle addresses that are affected by the

change in the next 3 processor cycles. The AMRs are located in a contiguous block beginning at address 0x00000140 with ASI 0x1.



* Except AMR[0]. See Table 2-3

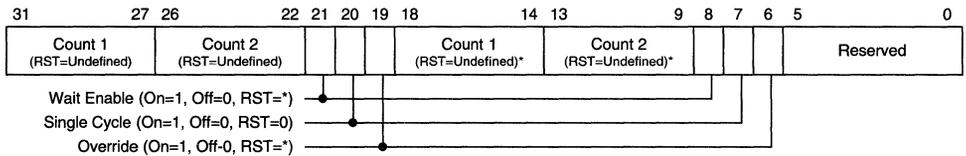
Figure 2-18. Address Mask Registers

- Bit 31: Reserved
- Bits 30-1: Mask—Specifies which bits in the ASI and address of the current external access are to be compared with the corresponding bits in the address-range specifier. Only those bits are compared for which the mask bit is 0. See Table 2-3 for reset value.
- Bit 0: Reserved

Wait-State Specifier Registers (WSSR[2:0])

The wait-state specifiers determine, for each of the address ranges defined by the ARSR and AMR registers, the number of clock cycles between the time an address in a given range appears on the address bus and the time the processor generates an internal –READY signal. This makes it possible for memory and I/O devices with different access times to be connected to the processor without additional logic.

The wait-state specifiers for the six address ranges are kept in three Wait-State Specifier Registers. These registers are located in a contiguous block beginning at address 0x00000160 with ASI 0x1 (see Table 2-3). Each register contains the wait-state specifiers for two address ranges. When the address currently being driven by the processor matches the unmasked bits in one of the Address Range Specifiers, the corresponding wait-state specifier is selected. These registers should not be written if the bus interface unit will handle addresses that are affected by the change in the next 3 processor cycles.



* See Table 2-3

Figure 2-19. Wait-State Specifier Registers

- Bits 31-19: Wait-State Specifier**—When an external access falls within an address range defined by an ARSR and AMR, the corresponding wait-state specifier determines when, and whether, the processor generates an internal --READY signal to terminate the access.
- Count1 (Bits 31-27):** The number of wait-states inserted before the internal --READY , under the following conditions: the Single Cycle bit equals 0 and the current access is not on the same page as the previous access. The number of wait-states is the value of this field +1 (i.e., 0=1 wait-state, 1=2 wait-states, etc.) The value of Count1 is undefined on reset.
 - Count2 (Bits 26-22):** The number of wait-states inserted before the internal --READY , under the following conditions: the Single Cycle bit equals 0 and the current access is on the same page as the previous access. The number of wait-states is the value of this field +1 (i.e., 0=1 wait-state, 1=2 wait-states, etc.) The value of Count2 is undefined on reset.
 - Wait Enable (Bit 21):** Enables and disables the wait-state generator for an individual address range. If the Wait Enable bit of a wait-state specifier equals 0, the internal --READY is not asserted when addresses in the corresponding range are accessed by the processor. If Wait Enable is 1, the single cycle bit must be 0. See Table 2-3 for reset value.
 - Single Cycle (Bit 20):** Specifies the timing of the internal --READY signal. If the Single Cycle bit equals 1 when an address in the appropriate range is accessed, the internal --READY is asserted in the same cycle. If the Single Cycle bit equals 0, and the current transaction is in the same page as the previous transaction, then Count2 is used as the number of cycles after which --READY is asserted internally. If the transaction is not in the same page, Count1 is used instead. If Single Cycle is enabled, the Wait Enable bit must be 0. See Table 2-3 for reset value.
 - Override (Bit 19):** Allows the system to terminate a memory transaction before the internally specified time. If the Override bit equals 1, and external hardware asserts the external --READY signal, then the wait-state generator will stop counting and will wait for the next transaction. This bit is cleared to 0 on reset.
- Bits 18-6: Wait-State Specifier**—The wait-state specifier for a second address range. This field is organized just like bits 31-19.
- Bits 5-0: Reserved**

System Support Control Register

The System Support Control Register enables or disables the various system-support features, independently of one another. However, the chip-select logic for address range 0 is always enabled, regardless of the value in the System Support

Control Register. This register is located at address 0x00000080 with ASI 0x1 (see Table 2-3).

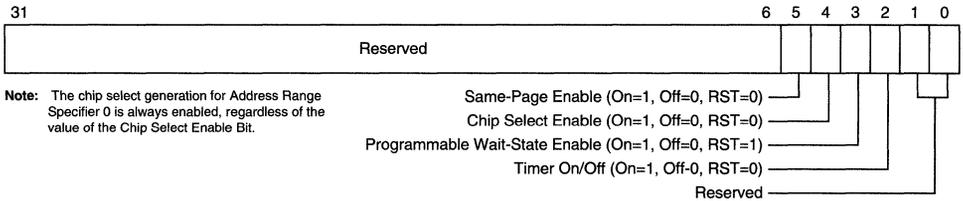


Figure 2-20. System Support Control Register

Bits 31-6: Reserved

Bit 5: Same-Page Enable—Enables (1) and disables (0) the same-page detection logic. When this bit is 1, the `-SAME_PAGE` signal is asserted whenever the address of an external access is on the same page as the previous access. The page size is controlled by the Same-Page Mask Register (see above). When this bit is 0, `-SAME_PAGE` is never asserted. The Same-Page Enable bit is cleared to 0 on reset.

Bit 4: Chip Select Enable—Enables (1) and disables (0) the generation of chip-select signals for external accesses in address ranges 1 through 5. Regardless of the state of this bit, however, `-CS0` is always asserted when the current address lies in address range 0. The Chip Select Enable bit is cleared to 0 on reset.

Note: Before enabling chip selects all chip select Address Mask and Address Range registers should be initialized so that two chip selects are never selected at the same time.

Bit 3: Programmable Wait-State Enable—Enables (1) and disables (0) the programmable wait-state generators for all address ranges. The Programmable Wait-State Enable bit is set to 1 on processor reset.

Bit 2: Timer On/Off—Enables (1) and disables (0) the timer. This bit is cleared to 0 on reset.

Bits 1-0: Reserved

Table 2-3: System Support Register Summary

Chip Selects	Affected by Chip-Select Enable?	Address Range Specifier		Address Mask		Wait-State Specifier	
		Address (ASI=0x01)	Value at Reset	Address (ASI=0x01)	Value at Reset	Address (ASI=0x01)	Value at Reset
0	No	N/A	ASI=0x09 ADR<31:10>=0	0x0000 0140	All mask bits 0 except ADR<14:10> = 1	0x0000 0160 (low halfword)	Count 1,2 = 31 Wait Enable=1 Single Cycle=0 Override=1

Table 2-3: System Support Register Summary

Chip Selects	Affected by Chip-Select Enable?	Address Range Specifier		Address Mask		Wait-State Specifier	
		Address (ASI=0x01)	Value at Reset	Address (ASI=0x01)	Value at Reset	Address (ASI=0x01)	Value at Reset
1	Yes	0x0000 0124	Undefined	0x0000 0144	Undefined	0x0000 0160 (high halfword)	Count 1,2 = Undefined Wait Enable =0 Single Cycle =0 Override=0
2		0x0000 1280		0x0000 0148		0x0000 0164 (low halfword)	
3		0x0000 012C		0x0000 014C		0x0000 0164 (high halfword)	
4		0x0000 0130		0x0000 0150		0x0000 0168 (low halfword)	
5		0x0000 0134		0x0000 0154		0x0000 0168 (high halfword)	

Timer Register

The Timer Register contains the current count of the internal 16-bit timer. When the timer overflows, the processor asserts the -TIMER_OVF signal and reloads the Timer Register with the contents of the Timer Preload Register. The Timer Register can also be loaded directly by writing to the address 0x00000174 with ASI 0x1. The timer is clocked at the processor clock frequency.

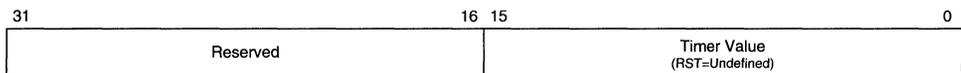


Figure 2-21. Timer Register

Timer Preload Register

The Timer Preload Register contains the value which is loaded into the timer when the timer overflows. In effect, this register specifies the number of clock cycles between assertions of the -TIMER_OVF signal. The Timer Preload Register is located at address 0x00000178 with ASI 0x1.

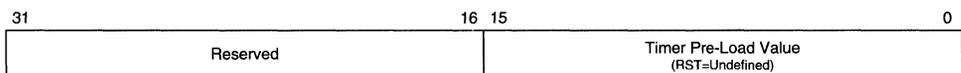


Figure 2-22. Timer Pre-Load Register

2.4 Data Types

Direct support is provided for signed and unsigned integers of various lengths, as illustrated in Figure 2-23. A *tagged word* type is supported for tagged arithmetic, used in artificial intelligence applications. Other data types (character strings, floating-point types, and so on) must be handled in software.

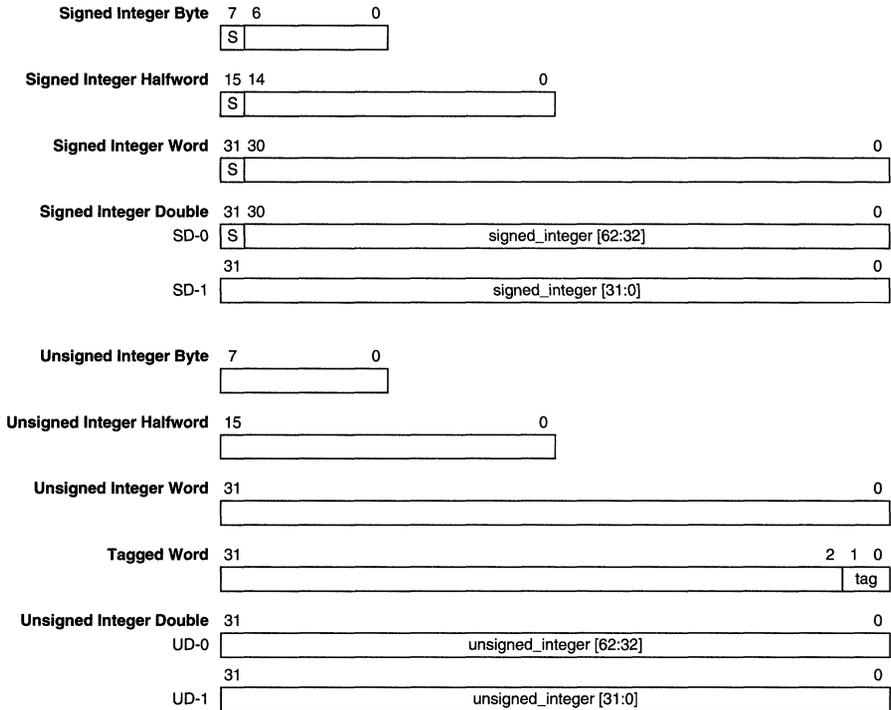


Figure 2-23. Data Types

2.5 Instructions

SPARClite provides an upward-compatible superset of the SPARC integer instruction set. Each instruction is a single 32-bit word. There are only three basic instruction formats, and few addressing modes.

The additional MB86930 instructions—integer divide-step, and scan for first changed bit—are implemented to achieve higher performance in embedded applications. Table 2-4 lists the MB86930 instruction set by function, and shows how to interpret the instruction mnemonics.

Table 2-4: Instruction Mnemonics

Load and Store:

$\left\{ \begin{array}{l} \text{LoaD} \\ \text{STore} \end{array} \right\} \left\{ \begin{array}{l} \text{Signed} \\ \text{Unsigned} \end{array} \right\} \left\{ \begin{array}{l} \text{Byte} \\ \text{Halfword} \\ \text{word} \\ \text{Double word} \end{array} \right\} \left\{ \begin{array}{l} \text{normal} \\ \text{Alternate} \end{array} \right\}$

atomic **SWAP** word
atomic **Load-Store Unsigned Byte**

Control Transfer:

Branch $\left\{ \begin{array}{l} \text{Integer} \\ \text{CC} \end{array} \right\} \left\{ \begin{array}{l} \text{normal} \\ \text{Annul delay instr.} \end{array} \right\}$

CALL
Trap on Integer CC
JuMP and **Link**
RETurn from **Trap**

Logical:

$\left\{ \begin{array}{l} \text{AND} \\ \text{OR} \\ \text{XOR} \end{array} \right\} \left\{ \begin{array}{l} \text{normal} \\ \text{Not} \end{array} \right\} \left\{ \begin{array}{l} \text{normal} \\ \text{set} \end{array} \right\}$

Arithmetic and Shift:

$\left\{ \begin{array}{l} \text{UMUL} \\ \text{SMUL} \end{array} \right\} \left\{ \begin{array}{l} \text{normal} \\ \text{set CC} \end{array} \right\}$

$\left\{ \begin{array}{l} \text{ADD} \\ \text{SUB} \end{array} \right\} \left\{ \begin{array}{l} \text{normal} \\ \text{eXtended} \end{array} \right\} \left\{ \begin{array}{l} \text{normal} \\ \text{set CC} \end{array} \right\}$

Shift $\left\{ \begin{array}{l} \text{Left} \\ \text{Right} \end{array} \right\} \left\{ \begin{array}{l} \text{Logical} \\ \text{Arithmetic} \end{array} \right\}$

Tagged $\left\{ \begin{array}{l} \text{ADD} \\ \text{SUB} \end{array} \right\} \text{set CC} \left\{ \begin{array}{l} \text{normal} \\ \text{Trap oVerflow} \end{array} \right\}$

SCAN
DIVide Step set CC
MULTiPLY Step set CC
SETHI

Read/Write Control Registers:

$\left\{ \begin{array}{l} \text{ReaD} \\ \text{WRite} \end{array} \right\} \left\{ \begin{array}{l} \text{Y} \\ \text{PSR} \\ \text{WIM} \\ \text{TBR} \\ \text{ASR} \end{array} \right\}$

SAVE
RESTORE

In the MB86930 processor, the floating-point and coprocessor instructions defined in the SPARC architecture are trapped for software emulation.

2.5.1 Instruction Formats

Figure 2-24 shows the three basic instruction formats.

Format 1 (op=1): CALL

31	30	29																												0
op			disp30																											

Format 2 (op=0): SETHI & Branches (Bicc, FBfcc, CBccc)

31	30	29	28					25	24			22	21											0
op		rd		op2				imm22																
op		a	cond		op2				disp22															

Format 3 (op=2 or 3): Remaining instructions

31	30	29					25	24					19	18				14	13	12				5	4			0
op		rd		op3				rs1				i=0		asi			rs2											
op		rd		op3				rs1				i=1		simm13														
op		rd		op3				rs1				opf			rs2													

Figure 2-24. Instruction Formats

op, op2, op3

One or more of these fields appear in every format to encode the instruction. The 2-bit *op* field is used in all three formats, and is interpreted as follows:

op Encoding (All Formats)

op	Format	Instructions
0	2	Bicc, FBfcc, CBccc, SETHI
1	1	CALL
2	3	arithmetic, logical, shift and remaining memory instructions
3	3	

The 3-bit *op2* field is used, along with the *op* field, to encode the format 2 instructions, and is interpreted as follows:

op2 Encoding (Format 2)

op2	Instructions
0	unimplemented
1	unimplemented
2	Bicc
3	unimplemented
4	SETHI
5	unimplemented
6	FBfcc
7	CBccc

	The 6-bit <i>op3</i> field is used, along with the <i>op</i> field, to encode the format 3 instructions. An <i>Instruction Index by Operation Code</i> is given in Chapter 7 of this manual.
<i>rd, rs1, rs2</i>	These 5-bit fields contain register addresses, interpreted as discussed in the <i>General-Purpose Registers</i> section, above. The <i>rd</i> field specifies the source operand for a store, or the destination operand for some other operation. The <i>rs1</i> and <i>rs2</i> fields specify source operands.
<i>disp30, disp22</i>	These 30-bit and 22-bit fields contain word-aligned, sign-extended, PC-relative displacements for a call or branch, respectively.
<i>a</i>	This bit is used in branch instructions to specify whether or not the instruction following the branch can be annulled.
<i>cond</i>	This 4-bit field selects the condition codes to test for a conditional branch instruction.
<i>imm22</i>	Contains a 22-bit constant which the SETHI instruction places in the upper end of a specified destination register.
<i>i</i>	Selects the second ALU operand for arithmetic and load/store instructions. If <i>i</i> equals 1, the operand is <i>r[rs2]</i> . If <i>i</i> equals 0, the operand is <i>simm13</i> , sign-extended from 13 to 32 bits.
<i>simm13</i>	Contains a sign-extended 13-bit immediate value used as the second ALU operand for an arithmetic or load/store instruction when <i>i</i> equals 1.
<i>asi</i>	Contains the 8-bit Address Space Identifier required for the load alternate and store alternate instructions.
<i>opf</i>	Encodes a floating-point operate or coprocessor operate instruction. All such instructions are trapped for software emulation.

2.5.2 Logical Instructions

The logical instructions perform bit-wise boolean operations. As shown in Table 2-5, each logical instruction comes in two versions: one leaves the integer condition codes in the Processor State Register unchanged; the other changes the condition codes as a side-effect.

Table 2-5: Logical Instructions

opcode	operation
AND	And
ANDcc	And and modify icc
ANDN	And Not
ANDNcc	And not and modify icc
OR	Inclusive Or
ORcc	Inclusive Or and modify icc
ORN	Inclusive Or Not
ORNcc	Inclusive Or Not and modify icc
XOR	Exclusive Or
XORcc	Exclusive Or and modify icc
XNOR	Exclusive Nor
XNORcc	Exclusive Nor and modify icc

The logical instructions are all format 3 instructions. When the *i* field is 0, they take their arguments from two source registers (*r[rs1]* and *r[rs2]*); when the *i* field is 1, they take one argument from source register *r[rs1]* and the other from the *simm13* field (sign-extended to 32 bits). In both cases, the result is written to the destination register *r[rd]*.

2.5.3 Arithmetic and Shift Instructions

The integer arithmetic instructions are generally three-register instructions which compute a result that is a function of the two source operands, and either write the result into the destination register *r[rd]*, or discard it. One of the source operands is always taken from register *r[rs1]*; the other source depends on the *i* bit in the instruction. If *i* equals 0, the second operand is taken from register *r[rs2]*; if *i* equals 1, the second operand is the value in the *simm13* field of the instruction, sign-extended to 32 bits. By specifying global register 0 as the destination, the instruction effectively discards the result. (See Section 2.3.2, *Special Uses of the r Registers*).

Besides the standard arithmetic operations, SPARC provides instructions to perform tagged arithmetic. In tagged arithmetic, the two least-significant bits of each operand are used to indicate the (user-defined) data type of the operand. The tagged arithmetic instructions set a condition code if the tag of an operand is not zero.

The shift instructions shift the contents of an *r* register by a constant or variable number of bits. They do not affect the condition codes.

Besides the instructions defined in the (Version 8) SPARC architecture, SPARC*lite* provides:

- A *divide-step* instruction, which can be used to construct efficient iterative integer division algorithms.
- A *scan* instruction, which determines the first bit in a word which differs from the most-significant bit. The scan instruction can be used to simplify and accelerate many important operations, like normalizing numbers with redundant sign bits.

Add and Subtract

The integer addition and subtraction instructions, listed in Table 2-6, perform two's-complement arithmetic. Each instruction comes in four versions: these either affect integer condition codes in the Processor State Register or leave them unchanged and either include the carry bit in the result or ignore it.

Table 2-6: Addition and Subtraction Instructions

opcode	operation
ADD	Add
ADDcc	Add and modify icc
ADDX	Add with Carry
ADDXcc	Add with Carry and modify icc
SUB	Subtract
SUBcc	Subtract and modify icc
SUBX	Subtract with Carry
SUBXcc	Subtract with Carry and modify icc

The integer addition and subtraction instructions are format 3 instructions. When the *i* field is 0, they take their arguments from two source registers ($r[rs1]$ and $r[rs2]$); when the *i* field is 1, they take one argument from a source register and the other from the *simm13* field (sign-extended to 32 bits). The result is written to the destination register $r[rd]$.

In subtraction, the second argument, whether register ($r[rs2]$) or immediate (*simm13*), is always subtracted from the first ($r[rs1]$).

The extended addition instructions ADDX and ADDXcc also add the carry bit (*c*) of the Processor Status Register; that is, they compute either " $r[rs1] + r[rs2] + c$ " or " $r[rs1] + \text{sign-extended}(simm13) + c$," and store the result in $r[rd]$.

The extended subtraction instructions SUBX and SUBXcc also subtract the carry bit (*c*); that is, they compute either " $r[rs1] - r[rs2] - c$ " or " $r[rs1] - \text{sign-extended}(-simm13) - c$," and store the result in $r[rd]$.

Overflow occurs on addition if both operands have the same sign and the sign of the sum is different. Overflow occurs on subtraction if the operands have different signs and the sign of the difference differs from the sign of $r[rs1]$.

A special comparison instruction for integer values is not needed, since it can be easily synthesized from the SUBcc instructions (See Chapter 7).

Tagged Add and Subtract

The tagged arithmetic instructions, listed in Table 2-7, perform two's-complement addition or subtraction on their operands.

Table 2-7: Tagged Arithmetic Instructions

opcode	operation
TADDcc	Tagged Add and modify icc
TADDccTV	Tagged Add, modify icc and Trap on Overflow
TSUBcc	Tagged Subtract and modify icc
TSUBccTV	Tagged Subtract, modify icc and Trap on Overflow

If either of operand has a non-zero tag, or if arithmetic overflow occurs, the overflow bit of the Processor Status Register is set to 1. The trapping versions (TADDccTV and TSUBccTV) also cause a tag_overflow trap whenever they set the overflow bit. Except for these special side effects, the tagged arithmetic instructions work just like the ordinary addition and subtraction instructions, which are described above.

TADDcc and TSUBcc modify the integer condition codes; TADDccTV and TSUBccTV also modify the condition codes when they do not trap.

Multiply and Multiply-Step

The integer multiplication instructions, listed in Table 2-8, are directly supported in hardware.

Table 2-8: Integer Multiply Instructions

opcode	operation
UMUL	Unsigned Integer Multiply
SMUL	Signed Integer Multiply
UMULcc	Unsigned Integer Multiply and modify icc
SMULcc	Signed Integer Multiply and modify icc
MULScc	Multiply Step and modify icc

The multiply instructions perform a signed or unsigned multiplication of a 32-bit multiplicand ($r[rs1]$) and a 32-bit multiplier (either $r[rs2]$ or *simm13*, sign-

extended to 32 bits), resulting in a 64-bit product. The low order 32 bits of the product are placed in the destination register ($r[rd]$), and the upper 32 bits of the product are placed in the Y register.

In general, the multiplication requires 5 cycles, but there are three special cases of early termination. If either the multiplier or the multiplicand is zero, the execution takes 1 cycle. If the multiplier is an 8-bit integer or less, the execution takes 2 cycles. If the multiplier is a 9-bit to 16-bit integer, the execution takes 3 cycles.

UMUL and SMUL do not affect the integer condition codes. The effect of UMULcc and SMULcc on the condition codes is shown in Table 2-7.

Table 2-9: Effect of Integer Multiplication on Condition Codes

icc bit	UMULcc	SMULcc
N	Set if product [31] = 1	Set if product [31] = 1
Z	Set if product [31:0] = 0	Set if product [31:0] = 0
V	Zero	Zero
C	Zero	Zero

The multiply-step instruction, MULSc, treats $r[rs1]$ and the Y register as a single, 64-bit, right-shiftable doubleword register. The least significant bit of $r[rs1]$ is treated as if it were the adjacent to the most significant bit of the Y register.

Multiplication with MULSc assumes that the Y register initially contains the multiplicand, $r[rs1]$ contains the most significant bits of the product, and $r[rs2]$ (or *simmm13*) contains the multiplier. Upon completion of the multiplication, the Y register contains the least significant word of the product. The operation of MULSc is described in the *Programming Considerations* chapter.

Divide-Step

The divide-step instruction, DIVSc, performs one bit-cycle of a non-restoring, shift-before-add, signed or unsigned integer division algorithm. It operates on a signed or unsigned dividend, with an unsigned divisor. It uses the integer condition code bits to carry the true sign of the remainder, and the previous quotient bit, from one cycle to the next. Remainder and quotient are kept in correct relative alignment because of the shift-before-add technique. Standard SPARC instructions are therefore sufficient for initializing and terminating both signed and unsigned division routines, eliminating the need for special divide-initialize, divide-terminate or remainder correction instructions.

Division with DIVSc assumes that the Y register initially contains the most significant word of the dividend, $r[rs1]$ contains the least significant word of the dividend, and $r[rs2]$ (or *simmm13*) contains the divisor. Upon completion of the division, the Y register contains the remainder and $r[rd]$ contains the quotient.

When DIVScc is used as expected, it will typically use the same register for *rd* and *rs1*. One exception is a signed division with one word dividend, in which the initial value of *r[rs1]* is saved in the first divide step by using an *rd* different from *rs1*.

DIVScc operates as follows:

1. The *true sign* is formed using the negative (*n*) and overflow (*v*) integer condition codes from the Processor Status Register. True sign = *n* XOR *v*.
2. The *remainder* is formed by upshifting the Y register (initially the most significant word of the dividend) one bit, and setting the least significant bit of remainder equal to most significant bit of *r[rs1]* (initially the least significant word of the dividend).
3. The *divisor* is *r[rs2]* if the *i* field is 0, or *simm13*, sign-extended to 32 bits, if the *i* field is 1.
4. If *true sign* = 0 (+), the ALU computes *remainder - divisor*. If *true sign* = 1 (-), the ALU computes *remainder + divisor*.
5. Carry out from the ALU operation is noted as *c0*. The negative (*n*) condition code is set to bit 31 of the ALU result. The zero (*z*) condition code is set if the ALU result is 0 AND the *true sign* equals Y[31], else cleared.
6. The *new true sign* is formed as (*true sign* AND NOT Y[31]) OR (NOT *c0* AND (*true sign* OR NOT Y[31])).
7. The overflow (*v*) condition code is formed as *new true sign* XOR bit 31 of the ALU result. The carry (*c*) condition code is set to NOT *new true sign*. Y is set to the 32-bit ALU result. If *rd* is not 0, then *r[rd]* is set to *r[rs1]*, upshifted one bit with NOT *new true sign* (the new quotient bit) in the least significant bit position.

See the *Programming Considerations* chapter for sample signed and unsigned division routines based on the DIVScc instruction.

Shift

The shift instructions, listed in Table 2-10, perform logical or arithmetic shifts on values in *r* registers. The shift count for these instructions is either a constant (the least significant 5 bits of *simm13*) or variable (the least significant 5 bits of *r[rs2]*), depending on the value in the *i* field: The least significant 5 bits of the 2's complement of a shift count are the same as 32 minus the shift count. No shift occurs when the shift count is 0.

Table 2-10: Shift Instructions

opcode	operation
SLL	Shift Left Logical
SRL	Shift Right Logical
SRA	Shift Right Arithmetic

SLL and SRL fill vacated bit positions with 0's. SRA fills vacated bit positions with the most significant bit of the $r[rs1]$ operand; that is, SRA treats its result as a two's-complement number, and sign-extends it to 32 bits. The shift instructions do not affect the condition codes.

An arithmetic shift left can be effected using the ADDcc instruction.

Scan

The SCAN instruction scans a register from MSB to LSB looking for either the first changed bit, first 1 or first 0 depending on the value of the source 2 operand. SCAN is a superset to the standard SPARC instruction set. It is decoded in an unused opcode and does not affect compliance with the SPARC architecture standard.

The SCAN instruction is useful for supporting operations like floating-point normalization by finding the number of sign bits in a single processor cycle. Data compression schemes like run length encoding execute significantly faster using SCAN as well.

SCAN works by computing the bitwise XOR of $r[rs1]$ with a mask created by right-shifting $r[rs2]$ by one bit and sign-extending the result. It finds the first 1 in the result, and writes this bit number to the destination register ($r[rd]$). Bit numbers range from 0 for the most significant bit to 31 for the least significant. If the two operands are identical, the value 63 is written into $r[rd]$.

Starting with the same number in $r[rs1]$ and $r[rs2]$, SCAN returns the number of sign bits. Consider the first example shown in Figure 2-25. Both source registers contain 0b00011.... The right-shifted, sign-extended, $rs2$ value is 0b000011..., and the result of the bitwise XOR is 0b0001.... The bit-position of the first 1 in this result (counting from zero, from the left) is 3, which is also the number of sign bits in the $rs1$ value. Similarly, example 2 shows the case where the sign bits are ones.

By using global register 0, which always reads as 0, as the mask operand ($rs2$), the bit position of the first 1 in $rs1$ can be found, as in the third example shown in Figure 2-25. Similarly, by using the immediate value -1, which extends to all 1's, as the mask operand, the bit position of the first 0 in $rs1$ is found. (See example 4).

SCAN does not affect the condition codes.

Example 1: finding the first changed bit (the first 1)

```
r[rs1] = 0b00011... (source 1)
r[rs2] = 0b00011... (source 2)
mask   = 0b000011 (source 2 shifted)
xor     = 0b00010... (xor of source 1 and mask)
r[d]    = 3          (bit location of first changed bit)
```

Example 2: finding the first changed bit (the first 0)

```
r[rs1] = 0b11100... (source 1)
r[rs2] = 0b11100... (source 2)
mask   = 0b111100 (source 2 shifted)
xor     = 0b00010... (xor of source 1 and mask)
r[d]    = 3          (bit location of first changed bit)
```

Example 3: finding the first 1

```
r[rs1] = 0b00011... (source 1)
r[rs2] = 0b00000... (source 2, immediate value 0 or %g0)
mask   = 0b000000 (source 2 shifted)
xor     = 0b00010... (xor of source 1 and mask)
r[d]    = 3          (bit location of first changed bit)
```

Example 4: finding the first 0

```
r[rs1] = 0b10000... (source 1)
r[rs2] = 0b11111... (source 2, immediate value -1)
mask   = 0b111111 (source 2 shifted)
xor     = 0b01111... (xor of source 1 and mask)
r[d]    = 1          (bit location of first changed bit)
```

Figure 2-25. Using the SCAN Instruction

Constants

The SETHI instruction loads a 22-bit immediate constant into an r register. SETHI zeroes the 10 least-significant bits of r[rd], and replaces its 22 high-order bits with the value from the imm22 field of the instruction. SETHI does not affect the integer condition codes. A SETHI instruction with rd = 0 and imm22 = 0 is the SPARC (Version 8) definition of a NOP.

2.5.4 Control Transfer Instructions

A control transfer instruction (CTI) is one which changes the value in the Next Program Counter (nPC) register. There are five basic types of control transfer instructions: conditional branches (Bicc), calls (CALL), jumps (JMPL), returns from trap (RETT), and conditional traps (Ticc).

As shown in Table 2-11, the control transfer instructions can be classified according to two criteria: *how the target address is calculated*, and *when the control transfer takes place*, relative to the CTI.

Table 2-11: Classification of Control Transfer Instructions

Control-Transfer Instruction	Target Address Calculation	Transfer Time Relative to CTI
Bicc	PC-relative	conditional-delayed
CALL	PC-relative	delayed
JMPL, RETT	register-indirect	delayed
Ticc	register-indirect-vectored	non-delayed

Three different schemes are used for computing target addresses:

- *PC-Relative*—Adds an *address displacement* to the current PC value. The *disp30* (CALL) or *disp22* (Bicc) field of the instruction specifies the number of words to be added to the PC; this number can be positive or negative. The *disp* value is sign-extended, then left-shifted by two bits to create the (byte) address displacement.
- *Register-Indirect*—Adds its two source operands (*r[rs1]* is always one of the operands; the other is *r[rs2]* when *i = 0*, and *simm13*, sign-extended to 32 bits, when *i = 1*).
- *Register-Indirect-Vectored*—Calculates the target address in two stages: it first obtains a trap type by adding 128 to the least significant 7 bits of the sum of its two source operands. *r[rs1]* is always one of the operands; the other is *r[rs2]* when *i = 0*, and *simm13*, sign-extended to 32 bits, when *i = 1*. The trap type number is then stored in the *tt* field of the Trap Base Register. The resulting value in the TBR is the target address.

Control transfer can either occur immediately after the CTI, or be delayed. The control transfer instructions fall into three classes:

- *Delayed*—Transfers control to the target address after a one-instruction delay. The *delay instruction*—the one whose address is in the nPC register when a delayed CTI is executed—is executed before the transfer of control to the target address. Special care is required when the delay instruction is itself a CTI; see the section on *Delayed-Control Transfer Couples*, below.
- *Non-Delayed*—Transfers control to the target address immediately after the CTI is executed.
- *Conditional-Delayed*—Delay occurs and the execution of the instruction in the delay slot is conditional, depending on the value of the *a* (annul) bit in the delayed control transfer instruction, and on whether or not the transfer itself is conditional. Details are provided below, under the heading *Branches*.

Branches

The Bicc instructions, listed in Table 2-12, perform program branches, either unconditionally or conditioned on the current values of the integer condition codes (bits 23-20 of the Processor Status Register). The branch target is specified by a PC-relative displacement.

Table 2-12: Branch Instructions

opcode	cond	operation	icc test
BA	1000	Branch Always	1
BN	0000	Branch Never	0
BNE	1001	Branch on Not Equal	not Z
BE	0001	Branch on Equal	Z
BG	1010	Branch on Greater	not (Z or (N xor V))
BLE	0010	Branch on Less or Equal	Z or (N xor V)
BGE	1011	Branch on Greater or Equal	Not N xor V
BL	0011	Branch on Less	N xor V
BGU	1100	Branch on Greater Unsigned	not (Cor Z)
BLEU	0100	Branch on Less or Equal Unsigned	(C or Z)
BCC	1101	Branch on Carry Clear (Greater than or Equal, Unsigned)	not C
BCS	0101	Branch on Carry Set (Less than, Unsigned)	C
BPOS	1110	Branch on Positive	not N
BNEG	0110	Branch on Negative	N
BVC	1111	Branch on Overflow Clear	not V
BVS	0111	Branch on Overflow Set	V

The unconditional branch BA causes a PC-relative delayed control transfer, regardless of the integer condition code values. If the *a* (annul) field is 0, the delay instruction is executed; if the *a* field is 1, the delay instruction is annulled (not executed).

The unconditional branch BN does not cause a transfer of control. BN acts like a NOP when its *a* (annul) field is 0. When its *a* (annul) field is 1, the following instruction (i.e., the delay instruction) is annulled.

The Bicc instructions other than BA and BN perform conditional branches, based on the current values of the integer condition codes. The test condition is coded into the *cond* field of the instruction, as shown in Table 2-12. If the test condition evaluates as true, the branch is taken, otherwise, no transfer of control takes place.

If a conditional branch is taken, the delay instruction is always executed, no matter what the value of the *a* (annul) field. If a conditional branch is not taken, and the *a* (annul) field is 1, then the delay instruction is annulled.

Table 2-13 summarizes the conditions under which the delay instruction is executed, for the various types of branches.

Table 2-13: Conditions for Executing Delay Instructions

a bit	type of branch	Delay instruction executed?
a = 0	unconditional conditional, taken conditional, non taken	YES YES YES
a = 1	unconditional conditional, taken conditional, non taken	NO (annulled) YES NO (annulled)

The effect of a branch instruction on the processor pipeline is shown in Figure 2-26.

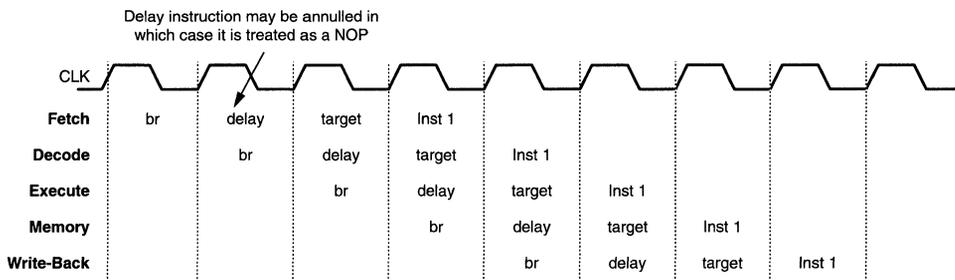


Figure 2-26. Pipeline Sequence: Branch

Call and Link

The CALL instruction writes the contents of the PC (i.e., the address of the CALL itself) into *out* register 7 (r[15]) of the current window. It then causes a delayed control transfer to a PC-relative target address. The instruction field that specifies the address displacement is 30 bits wide, so CALL can be used to transfer control anywhere in the address space. The call instruction pipeline sequence is identical to Figure 2-26, except that the delay instructions cannot be annulled.

Jump and Link

The JMPL instruction writes the contents of the PC (i.e., the address of the JMPL itself) into the destination register r[rd]. It then causes a delayed control transfer to a register-indirect target address. If the target address is not word-aligned, a *mem_address_not_aligned* trap occurs.

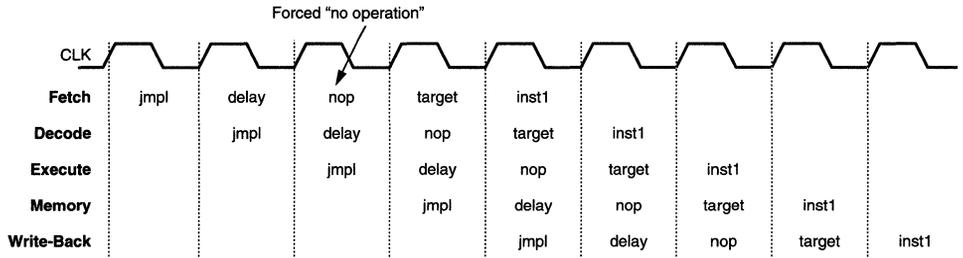


Figure 2-27. Pipeline Sequence: Jump and Link

Return from Trap

Unless it causes a trap, the RETT instruction does four things: it increments the Current Word Pointer (modulo 8), causes a delayed control transfer to the register-indirect target address, restores the processor to the operating mode (user or supervisor) it was in before the trap was taken, and enables traps.

If traps are enabled (i.e., if the ET bit of the Processor Status Register is set to 1), RETT will always cause a trap. A privileged_instruction trap will occur if the processor is in user mode, and an illegal_instruction trap will occur if the processor is in supervisor mode.

If traps are disabled (ET = 0), RETT can cause the following traps, in decreasing order of priority:

- Privileged_instruction, if the processor is in user mode.
- Window_underflow, if the new CWP corresponds to a set bit in the Window Invalid Mask register.
- Mem_address_not_aligned, if the target address of the control transfer is not word-aligned.

In these cases, the processor will write the appropriate trap type number into the tt field of the PSR, enter the error state, and halt.

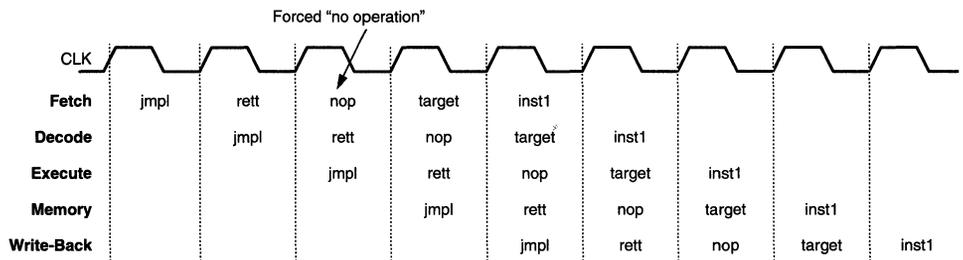


Figure 2-28. Pipeline Sequence: RETT

Software Traps

The Ticc instructions, listed in Table 2-14, generate the trap_instruction trap, either unconditionally or conditioned on the current values of the integer condition codes (bits 23-20 of the Processor Status Register). Ticc can be used to implement breakpoints, traces, and system calls. It can also be used for run-time checks, such as out-of-range array indexes or integer overflow.

Table 2-14: Trap Instructions

opcode	cond	operation	icc test
TA	1000	Trap Always	1
TN	0000	Trap Never	0
TNE	1001	Trap on Not Equal	not Z
TE	0001	Trap on Equal	Z
TG	1010	Trap on Greater	not (Z or (N xor V))
TLE	0010	Trap on Less or Equal	Z or (N xor V)
TGE	1011	Trap on Greater or Equal	Not N xor V
TL	0011	Trap on Less	N xor V
TGU	1100	Trap on Greater Unsigned	not (Cor Z)
TLEU	0100	Trap on Less or Equal Unsigned	(C or Z)
TCC	1101	Trap on Carry Clear (Greater than or Equal, Unsigned)	not C
TCS	0101	Trap on Carry Set (Less than, Unsigned)	C
TPOS	1110	Trap on Positive	not N
TNEG	0110	Trap on Negative	N
TVC	1111	Trap on Overflow Clear	not V
TVS	0111	Trap on Overflow Set	V

The Ticc instructions evaluate a boolean test condition based on the current values of the integer condition codes. The test condition is coded into the *cond* field of the instruction, as shown in Table 2-14. If the test condition evaluates as true, and no higher-priority trap or interrupt request is pending, the trap_instruction trap is generated. Otherwise, the instruction behaves like a NOP. The test condition for TA always evaluates as true, the condition for TN evaluates as false.

When Ticc generates a trap, the trap type is written into the *tt* field of the Trap Base Register. The trap type is calculated by adding 128 to the seven least significant bits of the sum of the two instruction operands. Register *r[rs1]* is always one of the operands; the other is *r[rs2]* when *i* = 0, and *simmm13*, sign-extended to 32 bits, when *i* = 1. The 25 most significant bits of *r[rs2]*, or the 6 most significant bits of *simmm13*, are unused and should be supplied as 0 by software.

Control is then transferred to the address in the TBR. The processor enters supervisor mode, disables traps, decrements the CWP (modulo 8), and saves the PC and nPC into *r[17]* and *r[18]* (*local* registers 1 and 2) of the new window. See the section on *Interrupts and Traps*, below.

Delayed Control-Transfer Couples

When a delayed control-transfer instruction is followed by another control-transfer instruction, the pair of CTI's is called a *delayed control-transfer couple* (DCTI couple). The order of execution for DCTI couples is illustrated by the examples in Table 2-15.

Table 2-15: Order of Execution for Delayed Control-Transfer Couples

Case	12: CTI 40	16: CTI 60	Order of Execution by Address
1	DCTI unconditional	DCTI taken	12, 16, 40, 60, 64...
2	DCTI unconditional	B*cc (a=0) untaken	12, 16, 40, 44,...
3	DCTI unconditional	B*cc (a=1) untaken	12, 16, 44, 48,... (40 annulled)
4	DCTI unconditional	B*A (a=1)	12, 16, 60, 64,... (40 annulled)
5	BA (a=1)	any CTI	12, 40, 44,... (16 annulled)
6	B*cc taken	DCTI	12, 16, 40, 60, 64, 68...

Note: Where the "a" bit is not indicated above, it may be either 0 or 1. See next table for abbreviations.

Abbreviations used in Previous Table

Abbreviation	Refers to Instructions
B*cc	Bicc (including BN, but excluding BA)
DCTI unconditional	CALL, JMPL, RETT, or BA (with a=0)
DCTI taken	CALL, JMPL, RETT, BA (with a=0), or B*cc taken

In the first five cases in Table 2-15, the first instruction causes an unconditional control transfer. Common examples of such DCTI couples are the JMPL, RETT sequences that can be used to return from a trap handler. In Case 6, the first instruction is a conditional branch; the order of execution is implementation-dependent.

Changing Windows with SAVE and RESTORE

The SAVE instruction decrements the Current Window Pointer (CWP) field of the Processor Status Register, thus saving the caller's window. The RESTORE instruction increments the CWP, restoring the caller's window. CWP arithmetic is performed modulo 8, the number of implemented windows.

If the new CWP value corresponds to a bit of the Window Invalid Mask register that is set to 1, a trap is generated: the window_overflow trap for a SAVE, and the window_underflow trap for a RESTORE.

If a trap is not generated, then, besides modifying the CWP, both SAVE and RESTORE act like integer addition instructions. The source operand fields *rs1* and (when *i* = 0) *rs2* are interpreted as register addresses in the old window, while destination field *rd* is interpreted as a register address in the new window.

The SAVE instruction can be used to allocate a new window in the register file, and a new software stack frame in memory, in a single atomic operation. See the *Programming Considerations* chapter for details.

2.5.5 Load and Store Instructions

The load and store instructions are the only ones that access memory and I/O, allowing bytes, half-words, words and doublewords to be transferred to and from processor registers.

Addressing modes are few and simple: the effective memory address is $r[rs1] + r[rs2]$ when $i = 0$, and $r[rs1] + (simm13, \text{sign-extended to 32 bits})$ when $i = 1$. The destination field, *rd*, specifies the register that supplies the data for a store, or receives it for a load.

The SPARC addressing convention is big-endian: the address of a halfword, word, or doubleword is the address of its most significant byte; increasing the address generally decreases the significance of the unit being addressed.

Attempts at unaligned accesses are trapped. An aligned doubleword address is divisible by 8, an aligned word address is divisible by 4, and an aligned half-word address is divisible by 2. If a load or store instruction generates an improperly aligned address, a `memory_address_not_aligned` trap occurs, and the access must be performed piecemeal under software control.

When performing an access, the processor generates an 8-bit Address Space Identifier along with the address. The ASI assignments for SPARClike are shown in Figure 1-1 in the *Overview* chapter. For a normal load or store instruction, the IU automatically supplies an ASI of 0x0A (user data space) or 0x0B (supervisor data space), depending on the current operating mode of the processor.

Privileged instructions exist for accessing the other address spaces. These instructions supply the Address Space Indicator explicitly in their *asi* fields. The “register + immediate” addressing mode is not available for these instructions; they cause an `illegal_instruction` trap if their *i* field is set to 1.

Load

The load integer instructions, shown in Table 2-16, copy data from memory into general-purpose registers. Bytes, half-words and words are copied into the destination register $r[rd]$. Doublewords are copied into an even-next odd r-register pair.

Table 2-16: Load Instructions

opcode	operation
LDSB	Load Signed Byte
LDSH	Load Signed Halfword
LDUB	Load Unsigned Byte
LDUH	Load Unsigned Halfword
LD	Load Word
LDD	Load Doubleword
LDSBA [†]	Load Signed Byte from Alternate space
LDSHA [†]	Load Signed Halfword from Alternate space
LDUBA [†]	Load Unsigned Byte from Alternate space
LDUHA [†]	Load Unsigned Halfword from Alternate space
LDA [†]	Load Word from Alternate space
LDDA [†]	Load Doubleword from Alternate space

[†]. privileged instruction

Fetches bytes and halfwords are right-justified in the destination register $r[rd]$, and either sign-extended or zero-extended on the left, depending on whether the load is signed or unsigned.

For a doubleword load, the effective memory address is that of the most significant word. This word is copied into the even-numbered register $r[rd]$; the last bit of the rd field is ignored, and should be supplied as 0. The least significant word is copied from the effective memory address + 4 into the following odd-numbered register. A successful doubleword load operates atomically.

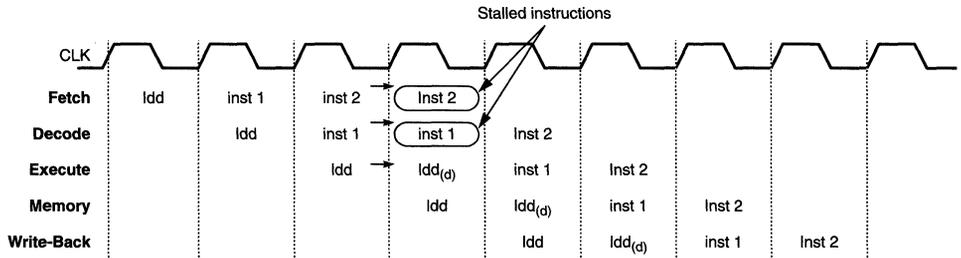


Figure 2-29. Pipeline Sequence: Load Double

Store

The store integer instructions, shown in Table 2-17, copy data from *r* registers into memory. Bytes, half-words and words are copied from the register *r[rd]*. Doublewords are copied from an even-odd *r* register pair.

Table 2-17: Store Instructions

opcode	operation
STB	Store Byte
STH	Store Halfword
ST	Store Word
STD	Store Doubleword
STBA [†]	Store Byte into Alternate space
STHA [†]	Store Halfword into Alternate space
STA [†]	Store Word into Alternate space
STDA [†]	Store Doubleword into Alternate space

†. Privileged instruction.

Byte (and halfword) stores take their data from the least significant byte (or halfword) of the register *r[rd]*.

For a doubleword store, the effective memory address is that of the most significant word. This word is copied from the even-numbered register *r[rd]*; the last bit of the *rd* field is ignored, and should be supplied as 0. The least significant word is copied from the following odd-numbered *r* register to the effective memory address + 4. A successful doubleword store operates atomically.

Atomic Load-Store

The atomic load-store instructions, LDSTUB and LDSTUBA, copy a byte from memory into *r[rd]*, and then rewrite the addressed byte with the value 0xFF. Interrupts and deferred traps cannot separate the load operation from the store.

Table 2-18: Atomic Load-Store Instructions

opcode	operation
LDSTUB	Atomic Load-Store Unsigned Byte
LDSTUBA [†]	Atomic Load-Store Unsigned Byte into Alternate space

†. Privileged instruction.

Swap

The SWAP and SWAPA instructions exchange the contents of $r[rd]$ and the addressed memory location. Interrupts and deferred traps are not permitted to intervene.

Table 2-19: Swap Instructions

opcode	operation
SWAP	SWAP r register with memory
SWAPA [†]	SWAP r register with Alternate space memory

[†]. Privileged instruction.

2.5.6 Read and Write Control Register Instructions

These instructions access the SPARC control and status registers. Except for SAVE and RESTORE, each one reads or writes the contents of an entire register. SAVE and RESTORE decrement and increment (respectively) the Current Word Pointer field of the Processor State Register.

Read Control Register

Each of the instructions shown in Table 2-20 copies data from a particular SPARC register into the destination register $r[rd]$.

Table 2-20: Read Control Register Instructions

opcode	operation
RDASR [†]	Read Ancillary State Register
RDY	Read Y Register
RDPSR [†]	Read Processor State Register
RDWIM [†]	Read Window Invalid Mask Register
RDTBR [†]	Read Trap Base Register

[†]. Privileged instruction.

The $rs1$ field of the RDASR instruction specifies which Ancillary State Register (ASR) is to be read. In SPARC*lite*, only ASR16 and ASR17 are implemented. Attempts to read any other ASR result in an illegal_instruction trap.

Write Control Register

Each of the instructions shown in Table 2-21 copies data into the writable fields of a particular SPARC register. The data to be written is calculated as the bitwise XOR of the two source operands. Register $r[rs1]$ is always one of the sources; the other is $r[rs2]$ when $i = 0$, and $simm13$, sign-extended to 32 bits, when $i = 1$.

The write control register instructions cause *delayed writes*. In a delayed write, the new value of the register is not available for some number of instructions after the write instruction. Table 2-21 shows the number of delay instructions for the SPARClike family processors. (Note: The SPARC architecture allows the number of delay instructions to take up to 3 cycles. If it is important to assure code compatibility with all implementations of SPARC the maximum delay should be assumed).

Table 2-21: Write Control Register Instructions

opcode	operation	write delay (cycles)
WRASR [†]	Write Ancillary State Register	0
WRY	Write Y Register	0
WRPSR [†]	Write Processor State Register	2
WRWIM [†]	Write Window Invalid Mask Register	2
WRTBR [†]	Write Trap Base Register	2

†. Privileged instruction.

Attempts to use or modify the contents of a register (except for the Y Register), after writing to it with a write control register instruction, have the following results:

1. Writing to any field of the same register within the write delay makes the contents of that field undefined.

Exception: A second instance of the *same* write control register instruction, even if it follows within three instructions of the first, will write the register as intended.

Note that many instructions *implicitly* write fields (Current Word Pointer, Integer Condition Codes) of the Program Status Register: the logical and arithmetic instructions whose mnemonics end in "cc"; SAVE and RESTORE; Ticc (when taken); and CALL.

2. Reading any *changed* field of the same register within the write delay yields an unpredictable value.

Note that many instructions *implicitly* read fields of the PSR: ADDX, SUBX, MULScc, DIVScc; SAVE and RESTORE; Bicc and Ticc.

3. If any of the two instructions following a write control register instruction causes a trap, a read control register instruction in the trap handler will get the register's new value.

If any of the two instructions following a WRTBR causes a trap, the Trap Base Address used will be the new value of the TBA field.

If any of the two instructions following a WRPSR causes a trap, the values of the S and CWP fields read from the PSR while taking the trap will be the new values.

WRPSR appears to write the ET and PIL fields immediately with respect to interrupts.

If an WRPSR instruction would cause the CWP field of the Processor Status Register (PSR) to point to an unimplemented window, it causes an `illegal_instruction` trap instead, and does not modify the PSR in any way.

The *rs1* field of the WRASR instruction specifies which Ancillary State Register (ASR) is to be written. In SPARClite, only ASR17 is implemented. Attempts to write any other ASR result in an `illegal_instruction` trap.

2.6 Data and Instruction Caches

Each member of the SPARClite family contains separate data and instruction caches on-chip. The caches are designed for maximum flexibility of operation. Under software control, individual entries or entire banks can be locked. The data cache can be decoupled from external memory and used as a fast on-chip scratch-pad RAM. This section discusses the structure and operation of the caches, as seen from the programmer's point of view.

2.6.1 Structure

In the MB86930 processor, each cache is 2 Kbytes in size, divided into 128 *lines* of 4 words (16 bytes) each. The contents of the cache data memory and tag memory is undefined at reset.

The cache organization, illustrated in Figure 2-30, is *two-way set associative*; that is, each address in memory can be cached in either of two locations. Each cache is divided into two *banks*, with 64 lines per bank. The 64 pairs of lines are called *sets*.

On a cache access, the address bits ADR[9:4] are used to select a set; the corresponding data or instruction values can be in either bank.

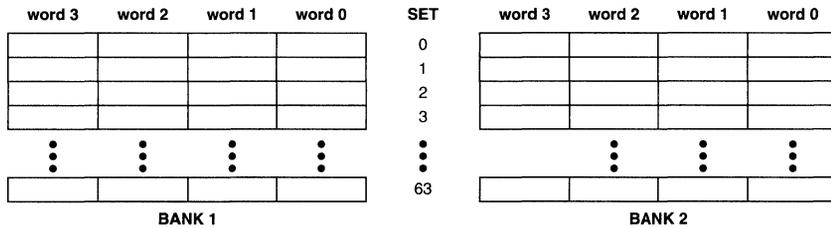


Figure 2-30. Cache Organization

Associated with each cache line is a *tag*, which indicates the memory location to which the line is currently mapped, and contains status information for the cached data or instructions. Data cache tags are located in the address space with ASI 0xE, and instruction cache tags in the address space with ASI 0xC (see Table 2-22). A cache *entry* consists of a cache line together with the corresponding tag. The structure of a cache tag is illustrated in Figure 2-31.

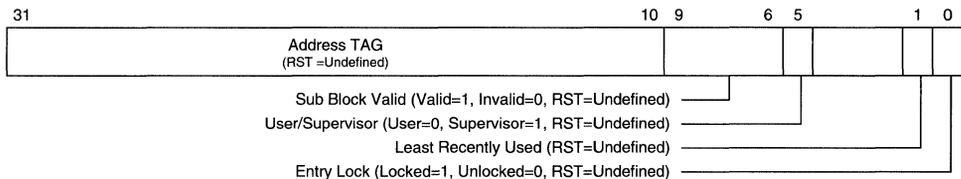


Figure 2-31. Cache Tag

- Bits 31-10: Address Tag—Contains the 22 most significant bits of the memory address of the data or instructions cached in the corresponding line. Undefined on reset.
- Bits 9-6: Sub-Block Valid—Contains one Valid bit for each of the 4 words in the corresponding line. When a Valid bit is 1, it indicates that the corresponding cache word contains a current data or instruction value for the address indicated by the tag. Undefined on reset.
- Bit 5: User/Supervisor—Indicates whether the data or instructions cached in the corresponding line come from user space (User/Supervisor bit = 0) or from supervisor space (User/Supervisor bit = 0). Undefined on reset.
- Bits 4-3: Reserved
- Bit 1: Least Recently Used (Bank 1 Only)—Indicates, for a given set, which bank contains the least recently used entry. When this bit is 1, it indicates that the entry in Bank 1 was the least recently used. Otherwise, Bank 2 was the least recently used. The value of this bit

determines which of the two entries is replaced when a new line needs to be allocated, and both entries are valid. Undefined on reset.

Bit 0: Entry Lock—Locks the current address into the cache tag entry. An access which competes with currently locked entries in both banks of the cache is treated as non-cacheable. Undefined on reset.

A faster way to set and clear the tag entry-lock bits is to write the Tag Lock Bit addresses as shown in Table 2-22. Writes to these locations map to the same entry lock bits in the instruction and data cache tags described in Figure 2-31 above. The advantage of writing the entry lock bit using these alternate memory locations is that only the lock-bit is affected on a write, the reset of the associated tag is not affected. The same operation using the cache tag address would require a read-modify-write so as not to change the rest of the tag value.

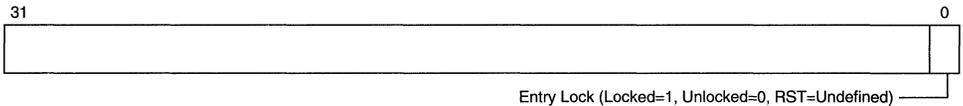


Figure 2-32. Tag Lock Bit

Bit 0: Entry Lock- Locks the current address into the cache tag entry. An access which competes with a currently locked entry in the cache is treated as non-cacheable. Writing this bit has the same effect as writing the corresponding bit in the cache tags except that the rest of the tag remains unaffected by a write to this location.

Table 2-22:Cache Tag Addresses

	Bank 1			Bank 2		
	SET	Cache Tag Address ASI=0xC	Tag Lock Bit ASI=0x2	SET	Cache Tag Address ASI=0xC	Tag Lock Bit ASI=0x2
Instruction Cache	0	0x 0000 0000	0x 0000 0000	0	0x 8000 0000	0x 8000 0000
	1	0x 0000 0010	0x 0000 0010	1	0x 8000 0010	0x 8000 0010
	2	0x 0000 0020	0x 0000 0020	2	0x 8000 0020	0x 8000 0020
	3	0x 0000 0030	0x 0000 0030	3	0x 8000 0030	0x 8000 0030
	4	0x 0000 0040	0x 0000 0040	4	0x 8000 0040	0x 8000 0040
	•	•	•	•	•	•
	•	•	•	•	•	•
	•	•	•	•	•	•
	•	•	•	•	•	•
	63	0x 0000 03F0	0x 0000 03F0	63	0x 8000 03F0	0x 8000 03F0

Table 2-22: Cache Tag Addresses

	Bank 1			Bank 2		
	SET	Cache Tag Address ASI=0xE	Tag Lock Bit ASI=0x3	SET	Cache Tag Address ASI=0xE	Tag Lock Bit ASI=0x3
Data Cache	0	0x 0000 0000	0x 0000 0000	0	0x 8000 0000	0x 8000 0000
	1	0x 0000 0010	0x 0000 0010	1	0x 8000 0010	0x 8000 0010
	2	0x 0000 0020	0x 0000 0020	2	0x 8000 0020	0x 8000 0020
	3	0x 0000 0030	0x 0000 0030	3	0x 8000 0030	0x 8000 0030
	4	0x 0000 0040	0x 0000 0040	4	0x 8000 0040	0x 8000 0040
	•	•	•	•	•	•
	•	•	•	•	•	•
	•	•	•	•	•	•
	63	0x 0000 03F0	0x 0000 03F0	63	0x 8000 03F0	0x 8000 03F0

2.6.2 Operation

This section discusses software initialization of the caches and the various cache operating modes.

Initialization

On reset, both caches are turned off, and all memory requests are sent to the Bus Interface Unit. In order to use the caches, software must initialize the Valid, Least Recently Used and Entry Lock bits by writing 0's to the appropriate alternate address spaces. After initializing the cache, a program can write 1's to the Cache Enable bits of the Cache/BIU control register to turn the caches on. Due to the pipeline in the IU, all writes are delayed by three instruction cycles.

Normal Operation

Accesses to the user and supervisor data spaces, and fetches from the user and supervisor instruction spaces, are generally cacheable. Stores to the instruction address space are not supported. Loads and stores to alternate memory spaces are not cacheable. I/O registers and other locations that need to be prevented from being cached should therefore be mapped to an alternate space. Atomic load/store transactions, including the SWAP instruction, are not cacheable. If an atomic operation references data already in cache, the entry for that data will be invalidated.

On any cacheable access, the address bits ADR[9:4] are used to select a set in the appropriate cache. Address bits ADR[3:2] are used to select a word from each of the two lines in the set; the Valid bits corresponding to those words are checked. The address bits ADR[31:10] are compared with the address tags. The User/Supervisor bit is tested against the ASI indicated by the IU.

A *cache hit* occurs if all of the following are true; otherwise, a *cache miss* occurs:

- ADR[31:10] matches the address tag in either set.
- The User/Supervisor bit corresponds to the ASI indicated by the IU.
- The Valid bit corresponding to the word being accessed is 1.

In the case of a *read hit*, the requested data or instruction is in the cache. The data or instruction is returned to the IU, and the pipeline is not held up. The LRU bit is updated. The lock bit may be updated based on the value of the Cache Entry Auto Lock bit in the Lock Control Register (see *Locking Modes*, below).

A *read miss* freezes the IU pipeline, and sends the request on to external memory. Though each cache line is four words long, only a single word is fetched on a miss. Assuming neither global nor local locking is in force, the fetched word will overwrite the appropriate word in one of the entries in the set. (Under global or local locking, a different policy is followed; see *Locking Modes*, below).

Sometimes a read miss occurs *only* because the Valid bit for the requested word is not set. In this case, a cache line has already been allocated for a 4-word memory block which includes the requested address. The fetched word simply overwrites the appropriate word in this line; the Valid bit for the word is then set.

Otherwise, a new line needs to be allocated on a read miss, and one of the two entries in the set corresponding to the requested address must be selected for replacement. The least recently used entry, as determined by the Least Recently Used bit for the set, is replaced. The fetched word overwrites the appropriate word in this line; its Valid bit is then set, and the Valid bits for the other words in the line are cleared.

The data cache follows a write-through memory update policy. On a *write hit*, the data is written both to the cache and to main memory (write-through). If there is a *write miss*, the data is written only to the external memory (no write-allocate) - the data cache and the corresponding cache tag are not updated or modified. (A different policy is followed if the write is to a locked location; see *Locking Modes*, below.) Data cache write misses can be avoided by first reading the data memory locations that are to be written.

Locking Modes

Without locking, read misses can cause cache lines to be re-allocated. Entire caches, or selected entries corresponding to time-critical routines, however, can be locked into cache. Locked entries cannot be re-allocated. Thanks to the set-associative organization, one bank of each cache can continue to operate as a fully functional direct-mapped cache, no matter how many entries in the other bank are locked.

On a read miss, if one of the entries in the addressed set is locked, the unlocked one is re-allocated, whether or not it was the least recently used. If both entries, or the entire cache, are locked, then the access will be treated as non-cacheable.

Writes to locked data entries, moreover, are not written through to main memory. In this way, a portion of the data cache can be used as fast on-chip RAM which is not mapped to external memory.

There are two modes of cache locking:

- **Global Locking** — Affects an entire cache. When a cache is locked in this way, valid entries are not replaced; invalid words in allocated cache locations will be updated. Bits in the cache/Bus Interface Unit Control Register enable or disable the global locking mode independently for each cache. Enabling global locking does not affect the Entry Lock bits of individual Cache lines; when global locking is subsequently disabled, lines with clear Entry Lock bits are once again subject to re-allocation.
- **Local Locking** — Affects individual cache lines.

Bits in the Lock Control Register enable or disable, independently for each cache, an auto lock mode in which all subsequent cache accesses automatically set the Entry Lock bit of the accessed entry. Software can also lock and unlock an individual entry by writing the lock bit in that entry's tag.

With auto-locking enabled for either the instruction or data cache, any lines accessed in that cache have their entry-lock bit set. This makes it easy to lock a routine into the cache by setting the auto lock bit in the Lock Control Register at the beginning of the routine and then executing the routine to lock the entries. The auto lock bit is cleared in one of two ways. Normally, software clears the auto lock bit at the end of the routine being locked. If a trap or interrupt occurs the auto lock bit will be cleared by hardware. This disables the locking mechanism so that the service routine is not locked into cache by mistake.

Two registers are provided to make it easy to re-enable the auto locking when the processor returns from the interrupt. The value of the Lock Control Register before the interrupt is automatically saved in the Lock Control Save Register when an interrupt or trap occurs. To restore the correct auto-lock value on return from the service routine, software sets a bit in the Restore Lock Control Register.

Figure 2-34. Reset always traps to address 0 and breakpoints always traps to 0x00000FF0.

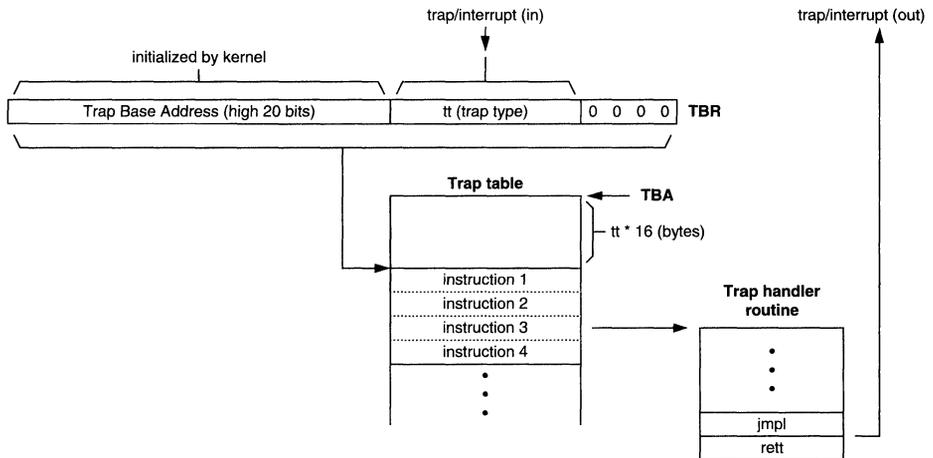


Figure 2-34. Trap and Interrupt Vectoring

A feature called *single vector trapping* allows all traps to vector to a single location, specified by the 20 high-order bits of the TBR, filled out on the right with 0's. After the trap is taken, the trap type can be determined by reading the tt field of the TBR. Single vector trapping can save code space and improve the response time of traps, since all of the trap service routines can potentially fit in cache. This feature, disabled at reset, can be enabled by setting the SVT bit of ASR17.

The Trap Enable bit (ET) of the Processor State Register enables (ET = 1) and disables (ET = 0) interrupts and traps. When ET = 0, interrupts are ignored, and traps cause the Integer Unit to halt and enter the error mode.

The processor provides direct support for 15 interrupt priority levels. The external interrupt request level (on input pins IRL[3:0]) is compared with the value in the Processor Interrupt Level field of the PSR. If the request level equals 15, or if it exceeds the PIL value, the interrupt is taken.

2.7.1 Trap Types

Up to 256 trap types can be distinguished on the basis of the 8-bit trap type number. Of these, half are reserved for external interrupts and hardware-enforced

instruction exceptions. The various trap types are listed in order of priority, with their causes, in Table 2-23.

Table 2-23:Traps

Trap	Priority	tt	Cause
reset	1	–	The external system asserted the –RESET input, signalling a reset request. Alternatively, the processor entered error mode and so generated an internal reset.
breakpoint_trap	1.5	255	Instruction or Data Breakpoint encountered or illegal write access to the breakpoint registers.
instruction_access_exception	2	1	A blocking error exception occurred on an instruction access (for example, an MMU indicated that the page was invalid or read-protected).
privileged_instruction	3	3	An attempt was made to execute a privileged instruction in user mode.
illegal_instruction	4	2	An attempt was made to execute an instruction with an unimplemented opcode, or an UNIMP instruction, or an instruction that would result in illegal processor state (for example, writing an illegal CWP into the PSR). Note that unimplemented FPop and unimplemented CPop instructions generate fp_exception and cp_exception traps.
fp_disabled	5	4	An attempt was made to execute an FPop, FBfcc, or a floating-point load/store instruction.
cp_disabled	5	36	An attempt was made to execute a CPop, CBccc, or a coprocessor load/store instruction.
window_overflow	6	5	A SAVE instruction attempted to cause the CWP to point to a window marked invalid in the WIM.
window_underflow	7	6	A RESTORE or RETT instruction attempted to cause the CWP to point to a window marked invalid in the WIM.
mem_address_not_aligned	8	7	A load/store instruction would have generated a memory address that was not properly aligned according to the instruction, or a JMPL or RETT instruction would have generated a non-word-aligned address.
data_access_exception	10	9	A blocking error exception occurred on a load/store data access. (For example, an MMU indicated that the page was invalid or write-protected).
tag_overflow	11	10	A TADDccTV or TSUBccTV instruction was executed, and either arithmetic overflow occurred or at least one of the tag bits of the operands was nonzero.
trap_instruction (Ticc)	12	128-255	A Ticc instruction was executed and the trap condition evaluated to true.

Table 2-23: Traps (Continued)

Trap	Priority	tt	Cause
interrupt_level_15	14	31	External Interrupt Request
interrupt_level_14	15	30	
interrupt_level_13	16	29	
interrupt_level_12	17	28	
interrupt_level_11	18	27	
interrupt_level_10	19	26	
interrupt_level_9	20	25	
interrupt_level_8	21	24	
interrupt_level_7	22	23	
interrupt_level_6	23	22	
interrupt_level_5	24	21	
interrupt_level_4	25	20	
interrupt_level_3	26	19	
interrupt_level_2	27	18	
interrupt_level_1	28	17	

2.7.2 Trap Behavior

The expression *trapped instruction* refers, in the case of a synchronous trap (instruction exception), to the instruction which caused it. In the case of an interrupt, the *trapped instruction* is the one which was about to enter the Writeback stage of the pipeline when the interrupt occurred.

The Integer Unit supports *precise traps*—when an interrupt or trap occurs, the saved state of the processor reflects the completion of all instructions prior to the trapped instruction, but no subsequent instructions (including the trapped instruction). Hardware guarantees that upon return from the service routine, the Program Counter points to the trapped instruction (or its successor if the trapped instruction was emulated).

The integer unit tests for exceptions generated by an instruction just before that instruction enters the Writeback stage. If an exception is detected, and no higher-priority request is pending, and traps are enabled, the processor takes a trap. If more than one exception is detected, the processor takes the trap with the highest-priority. When a trap is taken, the processor does the following things:

1. Writes the trap type number into the tt field of the Trap Base Register.
2. Saves the current processor mode (user or supervisor) by copying the value of the S bit of the Processor Status Register into the PS bit.
3. Enters supervisor mode by setting the S bit of the PSR to 1.
4. Disables traps by clearing the ET bit of the PSR to 0.

5. Saves the window of the interrupted routine by decrementing the Current Window Pointer (modulo 8). The Window Invalid Mask is *not* checked for window underflow or overflow.
6. Stores the current Program Counter and Next Program Counter values in r[17] and r[18] of the new window.
7. Transfers control to the address specified by the TBR.

An instruction is said to be *squashed* when its execution is aborted after it has entered the pipeline. A taken trap always squashes either 2 or 3 instructions. Asynchronous traps and interrupts squash 3 instructions as shown in Figure 2-35. Software traps (Ticc) only squash 2 instructions because the processor holds the next instruction fetch when the trap instruction reaches the memory stage (in Figure 2-35, instruction 4 is replaced by a hardware generated NOP).

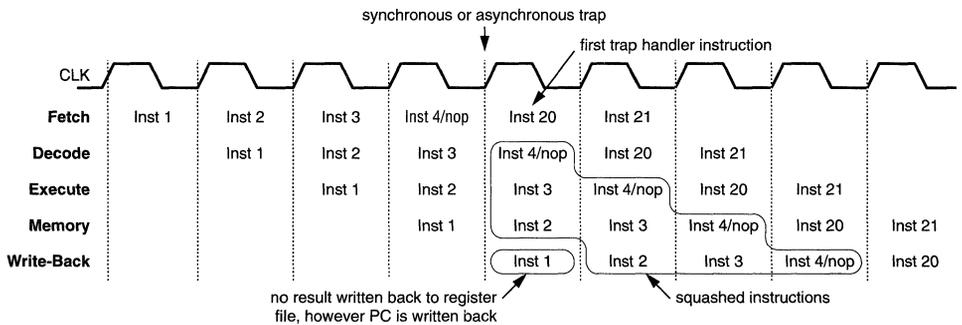


Figure 2-35. Instructions Squashed by Trap

The trap handler must insure that a window is available (for taking another trap), and then re-enable traps by setting ET to 1. The code for handling the exceptional condition that caused the trap can then be executed. Traps must be disabled (ET cleared to 0) before returning, via a RETT instruction, from the service routine.

Unless it causes a trap, the RETT instruction does four things: it increments the Current Word Pointer (modulo 8), causes a delayed control transfer to a register-indirect target address, restores the processor to the operating mode (user or supervisor) it was in before the trap was taken, and enables traps. The trap handler must ensure that a window is available so that RETT can increment the CWP without causing a window underflow and sending the processor into error mode.

2.7.3 Reset and Error Modes

As defined in the SPARC architecture, the SPARClite integer unit has *reset*, *error*, and *execute* modes which are states of the processor. The processor is in execute

mode during the normal execution of instructions. The processor enters error mode if a synchronous trap is encountered while the traps are disabled (the ET bit is 0). The processor enters reset mode when the -RESET input is asserted, and enters execute mode when the -RESET line is de-asserted.

Once it is in error mode, the processor must be reset in order to return to normal operations. The external system can detect an error condition by monitoring the -ERROR signal which is asserted for a minimum of one cycle.

Processor reset occurs whenever the -RESET input is held active for 4 cycles after the clock stabilizes. Reset does the following:

1. Writes 0 into the Program Counter and 4 into the Next Program Counter. When -RESET is de-asserted, the processor will begin fetching instructions at address 0x00000000 in supervisor instruction space (ASI 0x09).
2. Zeroes or sets to the appropriate NOP instruction all registers in the instruction pipeline. This insures that:
 - No instructions are left half-executed in the instruction pipeline.
 - No traps are taken prior to the instruction at address zero.
 - No control transfer instructions are in progress.
 - No interlock or bypass conditions will be detected prior to the instruction at address zero.
 - No state will be written back prior to the instruction at address zero.
3. Enters supervisor mode by setting the S bit in the PSR.
4. Disables traps by clearing the ET bit in the PSR.

2.8 Debug Support Unit

The Debug Support Unit (DSU) consists of a hardware emulator interface, debug support registers, and on-chip breakpoint and single-step logic that support hardware in-circuit emulators (ICE) and debug monitors.

The hardware emulator interface consists of a four-bit emulator data bus ($\text{EMU_D}<3:0>$), a four-bit multiplexed status/data bus ($\text{-EMU_SD}<3:0>$), an emulator break request pin (-EMU_BRK), and an emulator enable signal pin (-EMU_ENB). The emulator interface allows in-circuit emulators and other debug and diagnostic hardware to trace processor activity by monitoring transactions between the IU and cache. These buses and pins should remain open when an in-circuit emulator is not in use.

Debug monitors typically reside in ROM and do not require a dedicated interface. The -EMU_BRK and -EMU_ENB pins, however, are used to enable the DSU for use by debug monitors.

The debug support registers consist of six Breakpoint Descriptor registers, a Debug Control Register, and a Debug Status Register. These registers are used to specify breakpoints, to configure the DSU for desired operation, and to read debug status.

This section describes only DSU debug monitor support, and contains information that is necessary for implementing debug monitors in MB86930-based systems. DSU hardware emulator support is described briefly in Section 6.5. The documentation provided with the emulator contains detailed information for the specific emulator in use.

2.8.1 Monitor Mode

DSU trace and breakpoint debug monitor support operation is enabled and disabled according to the states of the active-low `-EMU_BRK` and `-EMU_ENB` processor input pins at reset as follows:

Table 2-24:

State at Reset		Function
<code>-EMU_ENB</code>	<code>-EMU_BRK</code>	
0	0	Reserved
0	1	Reserved
1	0	Monitor Mode. DSU Registers are cleared at reset, and breakpoint registers can be read and written.
1	1	Normal Mode. DSU Registers are cleared at reset, and all breakpoints are disabled.

The state of the pins are written to bits <1:0> in the Debug Status Register where they can be read by the processor initialization routine to determine whether to jump to the monitor or proceed with normal program execution. After reset, the states of the pins can change with no effect.

The processor jumps to the monitor if in *monitor mode*. The DSU registers are initially cleared at reset in this mode, breakpoints are disabled, and breakpoint registers are readable and writable. In *normal mode*, all breakpoints are disabled, the DSU Status Register can be read, and normal program execution proceeds without breakpoints.

Monitor Mode States

There are two monitor mode states: *break state* and *execute state*.

Break state is a very high-level state in which breakpoints are disabled, the DSU registers can be read and written, and the processor registers that are normally

accessed in Supervisor mode can be accessed. The break state is entered following reset when the monitor mode is selected with the `-EMU_ENB` and `-EMU_BRK` signals, and when the `-Break` flag is cleared to 0 in the Debug Control Register in response to a breakpoint, a software break request (Ticc255 instruction), or a DSU register write exception. The break state allows writes to the DSU registers to allow DSU configuration, and inhibits breakpoints to eliminate debug interrupts while configuring the DSU.

The execute state is the normal debug mode operating state in which breakpoints may be enabled, the DSU registers can be read but not written, and program execution proceeds pending a breakpoint. The execute state is entered from the break state by setting the `-Break` flag in the Debug Control Register to 1 and executing the `JMPL/RETT` pair.

2.8.2 Breakpoint Registers

The DSU contains a Debug Control Register (Figure 2-36) and a Debug Status Register (Figure 2-37) for DSU control and debug status reporting. It also contains six Breakpoint Descriptor Registers for specifying address and data breakpoints.

The breakpoint descriptor and control registers are memory-mapped to ASI 0x1 at the following addresses:

Table 2-25:

0x0000FF00	Instruction Address Descriptor Register 1
0x0000FF04	Instruction Address Descriptor Register 2
0x0000FF08	Data Address Descriptor Register 1
0x0000FF0C	Data Address Descriptor Register 2
0x0000FF10	Data Value Descriptor Register 1
0x0000FF14	Data Value Descriptor Register 2 or Mask Register
0x0000FF18	Debug Control Register
0x0000FF1C	Debug Status Register

Debug Control Register

The Debug Control Register is used to enable the breakpoints that are specified in the Breakpoint Descriptor Registers and to qualify the breakpoints as follows:

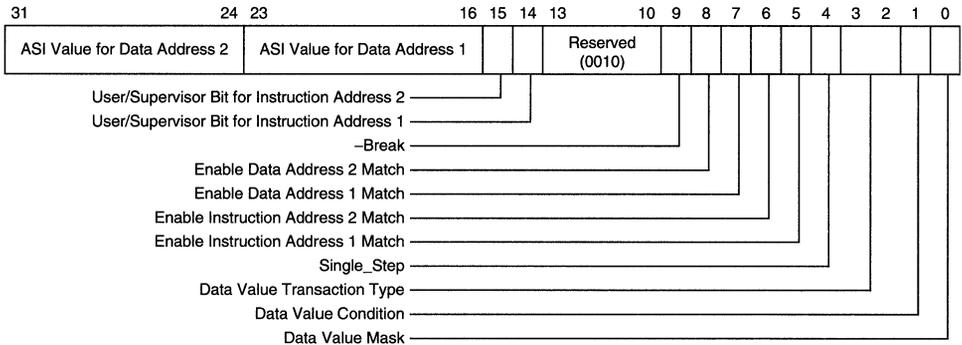


Figure 2-36. Debug Control Register

- Bit 31-24: Data Address 2 ASI: Specifies the ASI match value for Data Address 2.
- Bit 23-16: Data Address 1 ASI: Specifies the ASI match value for Data Address 1.
- Bit 15: Instruction Address 2 User/Supervisor Bit: Specifies either a User (when 0) or Supervisor (when 1) Mode match for instruction address 2.
- Bit 14: Instruction Address 1 User/Supervisor Bit: Specifies either a User (when 0) or Supervisor (when 1) Mode match for instruction address 1.
- Bit 13-10: Reserved, and must be written <0010> (bit 11 = 1; all other bits = 0).
- Bit 9: -Break—Cleared to indicate break state following reset or a breakpoint; set by the monitor to return to the execute state.
- Bit 8: Enable Data Address 2 Match—Enables (1) or disables (0) the breakpoint comparison for Data Address Descriptor 2.
- Bit 7: Enable Data Address 1 Match—Enables (1) or disables (0) the breakpoint comparison for Data Address Descriptor 1.
- Bit 6: Enable Instruction Address 2 Match—Enables (1) or disables (0) the breakpoint comparison for Instruction Address Descriptor 2.
- Bit 5: Enable Instruction Address 1 Match—Enables (1) or disables (0) the breakpoint comparison for Instruction Address Descriptor 1.
- Bit 4: Single Step—Enables single-step operation when set. During single-step operation, a breakpoint trap is issued on every instruction.

Bits 3-2: Data Value Transaction Type—Determines the class of instructions (loads, stores, or both) that can cause a Data Value breakpoint trap.

00	Break only on Loads
01	Break only on Stores
10	Break on Load or Store
11	Break Always

Bit 1: Data Value Condition—Determines whether a Data Value breakpoint trap is caused by values *inside* the range specified by the Data Value Descriptor Registers, or outside this range (assuming that the Data Value Mask bit is 0.)

Bit 0: Data Value Mask—Controls the interpretation of the Data Value Descriptors. When the Data Value Mask bit is 1, Data Value Descriptor 2 is used as a mask for Data Value Descriptor 1. When the Data Value Mask bit is 0, the Data Value Descriptors specify the upper and lower bounds of a value range.

Debug Status Register

The Debug Status Register contains breakpoint status and DSU enable flags as follows:

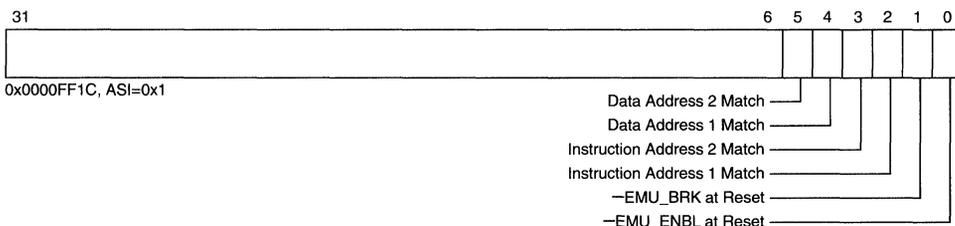


Figure 2-37. Debug Status Register

Bits 31-6: Reserved

Bit 5: Data Address 2 Match—set to (1) if address matched. Software should clear this bit after reading it.

Bit 4: Data Address 1 Match—set to (1) if address matched. Software should clear this bit after reading it.

Bit 3: Instruction Address 2 Match—set to (1) if address matched. Software should clear this bit after reading it.

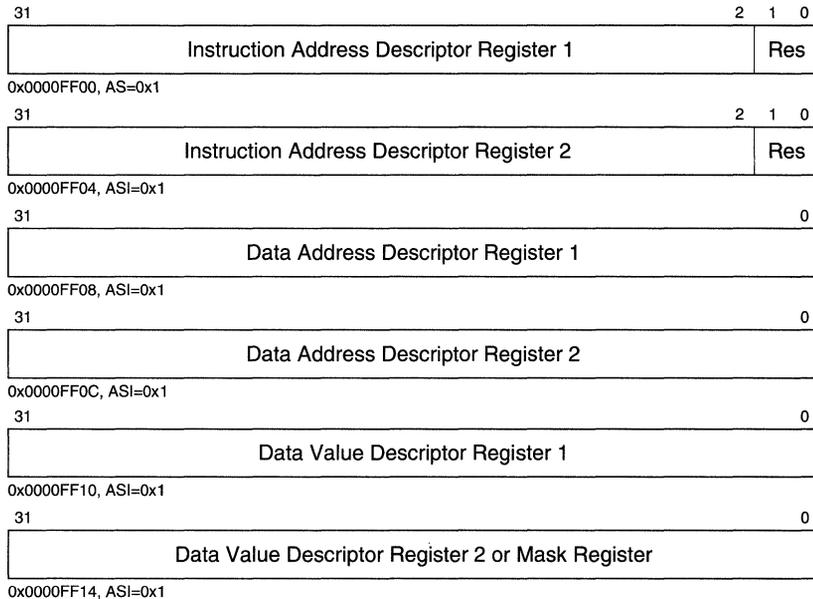
Bit 2: Instruction Address 1 Match—set to (1) if address matched. Software should clear this bit after reading it.

Bit 1: -EMU_BRK Asserted at reset—Holds the state of the -EMU_BRK pin during reset. Maintains its value until the next reset. -EMU_ENBL and -EMU_BRK are used to configure the DSU at reset. This bit is read only.

Bit 0: –EMU_ENBL Asserted at reset—Holds the state of the –EMU_ENBL pin during reset. Maintains its value until the next reset. –EMU_ENBL and –EMU_BRK are used to configure the DSU at reset. This bit is read only.

Breakpoint Descriptor Registers

The DSU contains two instruction address, two data address, and two data value breakpoint descriptor registers as follows:



The instruction addresses, data addresses, and data values in these registers specify breakpoints that force breaks when encountered during program execution and force the DSU to the break state. However, the breakpoints must first be enabled and qualified in the Debug Control Register.

Once a breakpoint occurs, the monitor reads the Debug Status Register to identify the breakpoint.

2.8.3 Breakpoint Traps

A breakpoint is a trap that changes the DSU state from the execute state to the break state, and vectors to the breakpoint trap handler at address 0x00000FF0. A breakpoint can be a hardware breakpoint (breakpoint address match, breakpoint data match, single step trace, or DSU register write exception), or a software breakpoint.

Unlike other traps, the breakpoint trap ignores the Trap Base Register (TBA) and the Single Vector Trap (SVT). The address of the breakpoint service routine is always 0x00000FF0 regardless of the TBA and SVT. A software breakpoint trap updates the Trap Type (tt) field in the Trap Base Register, but a hardware breakpoint trap does not update the tt field.

The breakpoint trap handler is exited by setting the –Break flag in the Debug Control Register, then executing the JMPL/RETT instruction pair to return to normal program execution. It is the responsibility of monitor code to restore all register window values (with the exception of the breakpoint trap window) to their pre-break values before returning from the trap.

Breakpoint traps have Trap Type number 255, and have a higher priority than other traps except RESET.

Instruction Address Breakpoints

An instruction address breakpoint occurs when an instruction address in the code being debugged matches the address in either Instruction Address Descriptor Register 1, or Instruction Address Descriptor Register 2. Each address must be qualified and enabled in the Debug Control Register as follows:

- (1) User or Supervisor mode instruction must be specified in the appropriate bit, <15> or <14>.
- (2) The breakpoint must be enabled in the appropriate bit, <6> or <5>.

The instruction address breakpoint trap is taken after the breakpoint instruction has completed execution.

Data Address Breakpoints

A data address breakpoint occurs when a data address in the code being debugged matches the address in either Data Address Descriptor Register 1, or Data Address Descriptor Register 2. Each address must be qualified and enabled in the Debug Control Register as follows:

- (1) The data address ASI must be specified in the appropriate ASI field, <31:24> or <23:16>.
- (2) The breakpoint must be enabled in the appropriate bit, <8> or <7>.

The data address breakpoint trap is taken after the breakpoint instruction has completed execution. Loads and Stores, for example, complete execution before a resulting breakpoint trap is taken.

Data Value Breakpoints

A data value breakpoint occurs when data that is transferred by the code being debugged falls within the range bounded by the values in Data Value Descriptor Registers 1 and 2, falls outside of the range bounded by the values in Data Value Descriptor Registers 1 and 2, or matches the bits in Data Value Descriptor Register 1 that are not masked by Data Value Descriptor Register 2. The data address must also match the descriptor in Data Address Descriptor Register 1 or 2.

The type of data value breakpoint must be selected in the Debug Control Register as follows:

- (1) The data transaction type must be selected in field <3:2>.
- (2) The data value condition must be selected in bit <1>.
- (3) Masking or no masking must be selected in bit <0>

The Data Value Descriptor Registers work in one of two ways. If the Data Value Mask bit in the Debug Control Register is 1, Data Value Descriptor 2 is used as a mask for Data Value Descriptor 1. In this mode only those bits of the Data Value Descriptor 1 for which the corresponding bits are 0 in Data Value Descriptor 2 are compared with the transferred data. All other bits are ignored in the breakpoint comparison.

If the Data Value Mask bit is 0, Data Value Descriptors 1 and 2 are the lower and upper bounds, respectively, of a comparison range. The break condition is determined by the values of the Data Value Condition bit in the Debug Control Register. If the Data Value Condition bit is a 0, the break condition is as follows:

$$\text{Data Value Descriptor 1} \leq \text{Transferred Data} \leq \text{Data Value Descriptor 2}$$

If the Data Value Condition bit is a 1, this break condition is inverted, changing the comparison into an "out-of-range" test.

The Data Value comparison may be conditioned by the type of transaction (load or store) that is being performed according to the Data Value Transaction Type selection in the Debug Control Register as follows:

Table 2-26:

00	Break only on Loads
01	Break only on Stores
10	Break on Load or Store
11	Break Always

Break Always results in breakpoints based on data address only.

The matching logic automatically masks unused bytes and halfwords in the data transfer.

Single Step Tracing

Single step tracing is initiated by setting the Single Step flag in the Debug Control Register while in break state, then returning to normal program execution by setting the –Break flag in the Debug Control Register and executing the JMPL/RETT instruction pair.

The next instruction in the program then executes, and the following instruction traps if ET=1. If ET=0, the breakpoint trap remains pending until ET is set to 1.

DSU Register Write Exception Breakpoint

Attempted writes to a DSU register while in the execute state of monitor mode results in a breakpoint. This breakpoint can be used by the monitor to force a change from the execute state to the break state. If ET=0 when the breakpoint request occurs, the breakpoint remains pending until ET=1.

Writes to the DSU registers are ignored in normal mode.

Software Breakpoint

A software breakpoint trap (Ticc255) functions the same way as other software traps, and has a trap priority of 12. If ET=0 when the breakpoint occurs, the breakpoint trap is ignored.

2.9 SPARC Compliance

SPARClite processors are fully compliant with the SPARC architectural specification.

Compatibility with existing and planned SPARC standards is a cornerstone of the SPARClite family strategy.

Compatibility assures:

1. a wide range of silicon implementations meeting different price/performance targets.
2. a ready availability of native development environments and tools
3. a large and growing base of application software which is object code compatible
4. an established and commercially viable processor architecture which is likely to be around well into the future.

The SPARC architecture was originally developed by SUN Microsystems, Inc. and first implemented by Fujitsu. SPARC International has since been formed to independently promote and control the evolution of the architecture.

All SPARC processor implementations conform to one of two architecture revision levels. The first commercially available version of the architecture is referred to as SPARC architecture Version 7. All existing silicon implementations and consequently SUN Microsystems, Inc. SPARCstations™ (1, 1+, 2, SLC, ELC, IPC, IPX) and SPARC compatible workstations conform to Version 7. A revised version of the SPARC architecture, Version 8, became final in March 1991. Future SPARC workstations will migrate to SPARC Version 8 processors. All OS and application code written for Version 7 processors will run without modification on SPARC Version 8 processors. SPARC*lite* series processors conform to Version 8 of the SPARC Architecture.

Version 8 of the SPARC Architecture adds these primarily features to Version 7.

- multiply- integer multiply instruction
- divide- integer divide instruction
- write/read ASR- read and write Ancillary State Register instructions which are used as additional control registers and implementation definable control registers

The architecture does not require that all instructions and features be implemented, only that the processor will trap on unimplemented features so that they can be emulated in software. SPARC*lite* implements the Version 8 multiply instruction and read and write ASR instructions. The integer divide instruction is not directly supported in hardware.

The MB86930 implements two instructions not defined by SPARC Version 8. These are the Scan and Divide Step instructions. These instructions are decoded in unused opcodes and provide a superset of SPARC Version 8. If code developed using these instructions is run on Version 7 or Version 8 SPARC processors other than SPARC*lite* an unimplemented instruction trap will occur.

CHAPTER 3



Internal Architecture

The internal architecture of SPARClite family processors is illustrated in Figure 3-1. The processor consists of a Clock Generator, an Integer Unit, separate on-chip caches for data and instructions, a Bus Interface Unit, and a Debug Support Unit to support the use of in-circuit emulators and target monitors. Internally, the various functional units are connected by separate instruction and data buses. For connection with external memory and I/O, a unified address bus and a unified data bus are extended off-chip. This chapter discusses the individual functional units in turn, giving an overview of the flow of data and control signals through the processor.

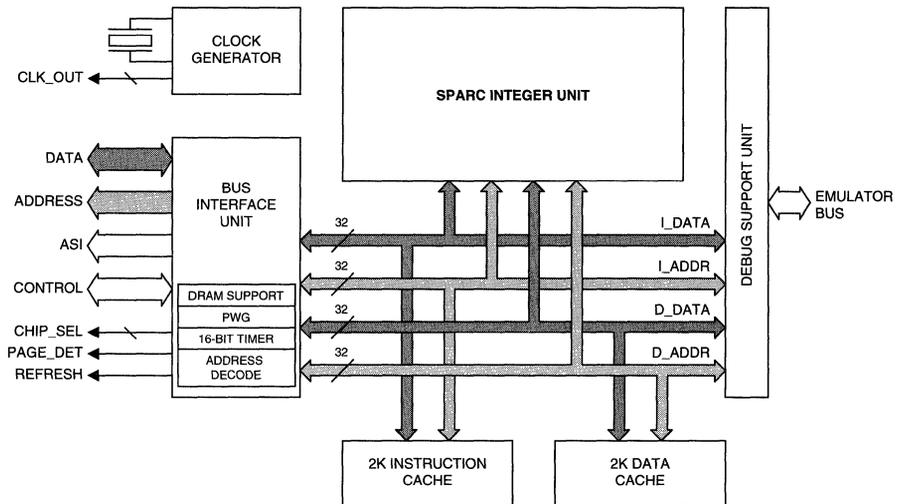


Figure 3-1. Internal Architecture (Block Diagram)

3.1 Integer Unit

The Integer Unit (IU) is a compact, fully custom implementation of the SPARC architecture. It is hard-wired for maximum performance; that is, it uses no micro-code. It contains three functional units:

- *Instruction Block*—Contains the instruction pipeline; decodes instructions into control signals for the other blocks.
- *Address Block*—Performs all instruction-address manipulations.
- *Execute Block*—Performs all data manipulations; generates operand addresses for load and store instructions and effective addresses for some of the control transfer instructions.

As shown in Figure 3-2, the IU is based on a Harvard (Aiken) architecture. There are separate address buses for instructions and data. There are also two 32-bit data interfaces: the instruction data bus, and the data bus. The use of these four

buses allows the IU to retrieve data and instructions simultaneously from on-chip cache.

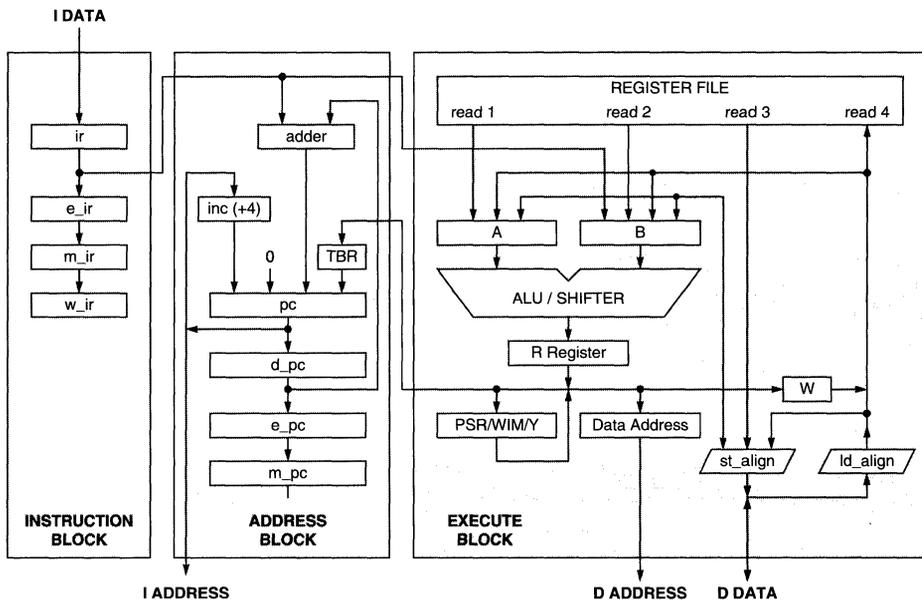


Figure 3-2. Integer Unit Data Path

3.1.1 I Block

The instruction block (I Block) contains the five-stage instruction pipeline and the logic which decodes instructions into control signals for the rest of the IU. The I block detects all bypass and interlock conditions.

The main interfaces to the I block are:

- Instruction data bus from the instruction cache or main memory.
- Immediate data field which goes to the A block for computing PC relative control transfers, and to the E block to be used as immediate data.
- Control signals to the A block and E block, including the register file read and write addresses, register enable signals, multiplexer controls, and partly or fully decoded operation codes for the ALU/Shifter.
- Status signals back from the E block, including possible trap conditions such as `memory_address_not_aligned` or `tag_overflow`.

Instruction Pipeline

The IU implements a five-stage instruction pipeline to allow a sustained execution rate of nearly one instruction per cycle. The operation of the pipeline under ideal conditions is illustrated in Figure 3-3. The pipeline consists of the following stages:

1. Fetch (F)—One of the instruction memory spaces is addressed and returns an instruction. (The figure below assumes a hit in the instruction cache.)
2. Decode (D)—The instruction is decoded; the register file is addressed and returns operands.
3. Execute (E)—The ALU computes a result.
4. Memory (M)—External memory is addressed (for load and store instructions only; this stage is idle for other instructions).
5. Writeback (W)—The result (or loaded memory datum) is written into the register file.

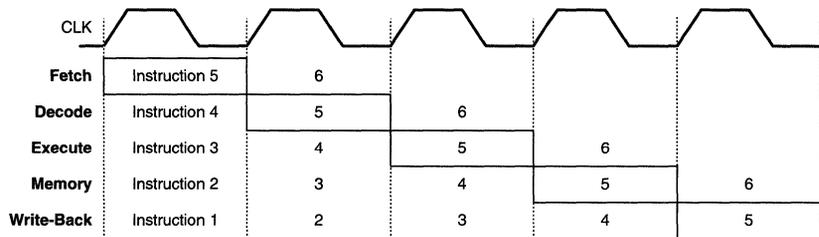


Figure 3-3. Instruction Pipeline

No instructions execute out-of order; that is, if instruction A enters the pipeline before instruction B, then instruction A necessarily reaches the writeback stage before instruction B does.

The control logic for the instruction pipeline is illustrated in Figure 3-4. At each cycle a horizontal control word is available which is wider than 32 bits and controls every multiplexer, latch-enable, and unit op-code in the chip. The horizontal control word is composed of control signals active during the decode stage of instruction N, the execute stage of instruction N-1, the memory stage of instruction N-2 and the writeback stage of instruction N-3. Some control bits require no decoding and are simply hardwired from the appropriate bits in the instruction register. Because the SPARC instruction set is not completely orthogonal (not every instruction field has the same meaning in every instruction) most bits require some decoding based on a single instruction in the pipeline. Some control

bits require decoding using logic that looks at two instructions in the pipeline, as, for example, in controlling multiplexers to select data bypass paths.

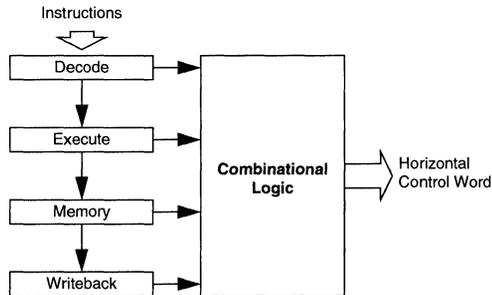


Figure 3-4. Instruction Pipeline Control Logic

Pipeline Hold

The IU does not complete one instruction on absolutely every cycle. On a load instruction, for example, external memory may be slow in returning the requested data. Because the IU does not execute or complete instructions out of order, the pipeline must be held up until the requested data is returned. Only then can the instruction complete and only then can the subsequent instructions continue.

There are also some hazards built into the IU datapath which require interrupting the one-cycle-per-instruction sequence of the pipeline. For example, a double-word load cannot be performed in one cycle because there is not enough memory or register-file bandwidth to move the data through the datapath. Another example is a load to a register which is followed by an instruction which uses that register. Because the operand of the second instruction is required in the decode stage but is not available, this instruction must be delayed until the operand is available.

Conditions which hold up the processor pipeline are handled uniformly by the I Block control logic and are referred to as *hold conditions*. A complete list of possible hold conditions is given in Table 3-1.

Table 3-1: Conditions Which Cause a Pipeline Hold

Name	Description	Pipeline Stage	Instruction Affected
ihold	Processor is attempting to fetch an instruction that is not yet available.	Fetch	Any instruction
dhold	Data is not yet available	Memory	Loads and Stores
mhold	Multiplication in progress	Execute	Integer Multiplication

Table 3-1: Conditions Which Cause a Pipeline Hold

Name	Description	Pipeline Stage	Instruction Affected
Interlock	An instruction in the pipeline must wait for some prior instruction to be completed (through Writeback).		Load/Use and CALL/Use r15 Instruction Pairs
Multicycle Instruction	An instruction which inherently requires more than one cycle in the pipeline	Execute	Load and Store Double-word, Atomic Load/Store

The *interlock conditions* are:

- **Load/Use Instruction Pairs**—If a load instruction which has rd=N as its destination register is followed by an instruction which uses rs=N as one of its source operands, then the load must proceed through Writeback before the following instruction can enter the Execute stage.
- **CALL/Use %r15 Instruction Pairs**—Similarly, since the CALL instruction implicitly writes the current value of the PC into r15, it must proceed to Writeback before any following instruction which uses r15 can enter the Execute stage.

Any time an interlock is detected, a NOP is inserted into the pipeline. The address block is signaled, so that the address of the instruction which causes the interlock is replicated in the address pipe. The NOP itself cannot cause a trap.

The multicycle instructions are LDD, LDDA, STD, STDA, LDSTUB, LDSTUBA, SWAP, and SWAPA. When a multicycle instruction enters the Execute stage, it and the instruction in the d_ir register are frozen for an additional cycle. Although it is possible to detect a multicycle instruction while it is in the Decode stage (unlike interlocks, which cannot be detected without looking at two instructions, those in the d_ir and e_ir registers), the I Block allows it to progress to the Execute stage before a hold is generated and inserted. This simplifies control somewhat because there are fewer points at which the pipeline must be held.

Note that the maximum number of internally generated hold cycles an instruction can cause is two, as in the following case:

```
LDD [%r1+%r2],%0r4
ADD %r5,%r5,%r6
```

The LDD takes two cycles, and it generates an interlock because the next instruction uses the data loaded in the second data memory cycle of the LDD instruction.

When a hold condition occurs, combinational logic generates one or more *freeze signals*, which prevent latches from being updated, and hence keep the pipeline from advancing. For some holds—dhold, for example—the entire pipeline is

frozen, with freeze signals being generated for all stages in the pipeline. For other holds—interlock conditions, for example—later stages in the pipeline must advance for the hold condition to be resolved. Thus only the earlier stages of the pipeline are frozen.

Trap Logic

SPARC_{lite} supports precise traps; that is, when a trap occurs, the saved programmer-visible state of the processor reflects the completion of all instructions prior to the trapped instruction, and no subsequent instructions including the trapped instruction. Thus, when an instruction causes a trap, one of two statements is true:

- No results from that instruction have been written into the programmer-visible registers (the register file or the PSR, TBR, WIM, or Y registers).
- Or, if data has been written into a programmer-visible register, the data contained in that register prior to being written by the trapped instruction is saved by the processor and can be restored when the trap is taken.

Table 3-2 shows the pipeline stages in which the various trap conditions are detected.

Table 3-2: Detection of Trap Conditions

Priority	Trap Type	Stage Detected	Trap
1			reset (hardware reset)
1	-	D	reset
2	1	F	instruction_access_exception
3	3	D	priv_instruction
4	2	D	illegal_instruction
5	4	D	fp_disabled
5	36	D	cp_disabled
6	5	D	window_overflow
7	6	D	window_underflow
8	7	E	mem_address_not_aligned
10	9	M	data_access_exception
11	10	E	tag_overflow
12	128-254	D	trap_instruction (Ticc)
13	255	F	instruction_breakpoint
13	255	M	data_breakpoint
14	31		interrupt_level_15
15	30		interrupt_level_14
.	.		.
.	.		.
.	.		.
28	17		interrupt_level_1

As shown in Table 3-2, the latest stage in which a trap can be detected is the Memory stage (a data memory exception for a load or store). If a programmer-visible register is updated prior to this stage, its original contents must be restored when and if the trap is taken.

Due to the pipelined operation of the IU, a trap condition for one instruction may actually be detected before a trap condition for a prior instruction. Thus, it is necessary to align the detected trap conditions so that all trap conditions for instruction N are considered together, before considering any trap conditions resulting from instruction N+1.

The trap coder is illustrated in Figure 3-5. Its purpose is to align in time the (possibly multiple) trap sources for a single instruction, to determine if a trap is to be taken or not, and if so, to determine the highest priority trap and code its trap type.

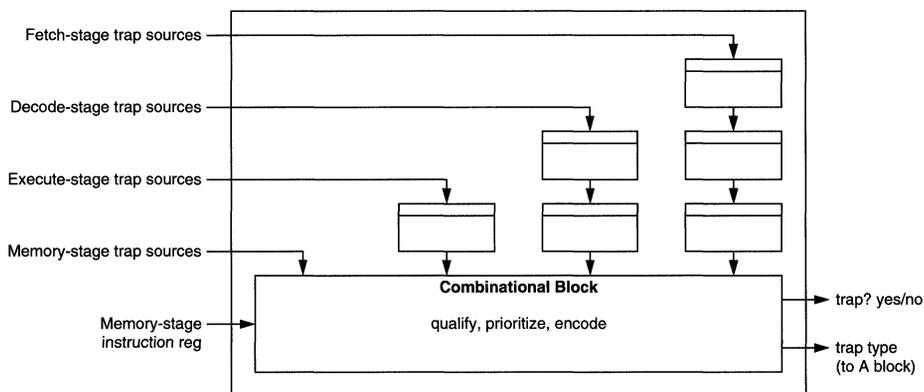


Figure 3-5. Trap Coder

When a trap is taken, the trap type field goes to the A Block where it is used immediately as a trap target address (when concatenated with the Trap Base Address) and is latched into the Trap Base Register.

3.1.2 A Block

The A Block contains the address pipeline. Along with the E Block, it is responsible for all instruction-address manipulations. The A Block executes the CALL and Bicc instructions. The A Block and E Block are used together to execute the JMPL, Ticc, and RETT instructions; in these cases, the A Block controls the update of the Program Counter. The A Block's main interface to the rest of the chip outside the IU is the instruction address bus.

The address pipeline is illustrated in Figure 3-6. The fetch-stage program counter (PC) is used to address instruction memory via the instruction address bus. Because a CALL, JMPL, or trap may require that the address of an instruction be written back to the register file, the address of every instruction tracks the instruction itself in the instruction pipeline so that it is available in the memory stage if it needs to be written back to the register file. These address pipeline registers are the decode, execute, and memory program counters. Each of these registers contains the address from which the instruction in the corresponding instruction register was fetched.

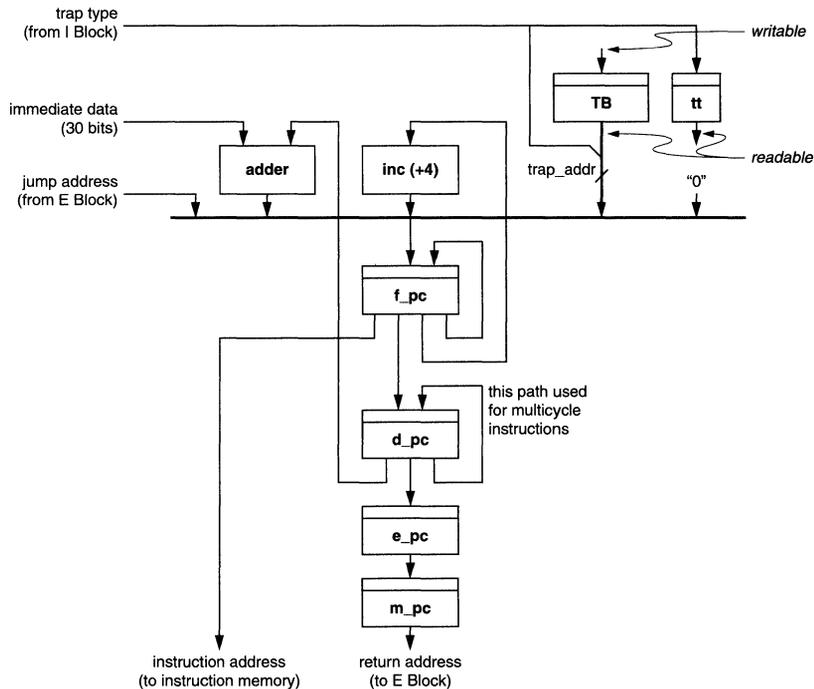


Figure 3-6. Address Pipeline

The PC has five possible sources:

1. +4 incremter, for normal, sequential instruction fetch.
2. The address adder, for PC-relative control transfer (Bicc or CALL instruction). The immediate data field contains offset information and comes from the I Block.

3. The jump address for a JMPL or RETT instruction. The jump address bus contains jump target information, and comes from the E block by way of the register file and ALU.
4. The TBR, concatenated with the trap type (tt) or with zeroes (when Single-Vector Trapping is enabled), on a Ticc instruction or an interrupt or trap. The trap type comes from the trap priority encoder, part of the I Block; when concatenated with TBR[31:12], it gives the target address for a trap.
5. Zeroes, concatenated with the trap type, for reset.

Note that "+4" is used to indicate that the (byte) address is incremented by 4 to fetch the next instruction. In reality, the two least significant bits of the address are not implemented in hardware because they are never used. Word alignment, for the case of a jump address coming from the E Block is verified in the E Block (and to some extent, the I Block).

The return address bus is written back to the register file in the case of a CALL, JMPL or Trap.

Several control signals come from the I block. These include:

- PC input-select signals which control the PC input multiplexer.
- The address adder control signal, which determines whether a 30-bit or a 22-bit immediate address field is added to the previous value of the PC (now found in the decode-stage PC).
- Pipeline freeze signals which can prevent the updating of registers in the pipeline when a hold condition is detected.

3.1.3 E Block

The E Block is responsible for all IU data manipulations. It generates operand addresses for load and store instructions and effective addresses for some of the control transfer instructions.

As shown in Figure 3-7, the E Block contains the Store Align Unit (SAU), the Load Align Unit (LAU), the Register File (RF), and the Adder, Shift, and Logic Unit (ASLU). The E Block also contains the result bypass logic that determines which operands are driven into the ASLU, and the store bypass logic that determines what data is latched for stores.

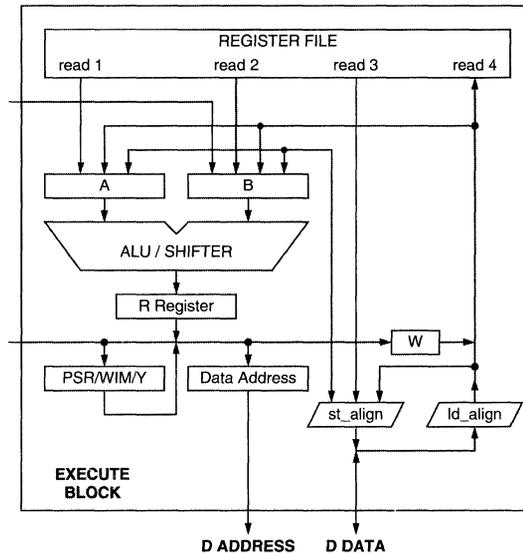


Figure 3-7. Execute Block

Adder, Shift, and Logic Unit (ASLU)

The ASLU incorporates an integer adder, a barrel shifter, a logic unit, and a scan unit. The integer adder calculates the results of the addition, subtraction, multiply-step, and divide-step instructions, and generates the carry, overflow, negative, and zero condition code values. It is used in load and store operations to calculate effective data addresses, and in register-indirect control transfers to calculate the new address to be placed in the PC register of the A Block. The integer adder also serves the multiplication unit by adding the “sum” and “carry” vectors during integer multiplications. The barrel shifter/logic unit executes the logic and shift instructions. The scan unit exists solely to support the scan instruction.

Results from the integer adder, the barrel shifter, the logic unit, and the scan unit are multiplexed into the R (Result) Register. Results from the integer adder are also made available to the Y Register.

Register File

The register file contains 136 registers of 32 bits each. The organization of these registers into windows is discussed in the *Programmer’s Model* chapter. The register file has one write port and three read ports. The write port is used for the instruction destination register (denoted *rd* in instruction descriptions). Two of the read ports are used for the two instruction source registers (*rs1* and *rs2*). The

remaining port is used for the data to be stored when a store or swap instruction is executed. In this way, even store instructions can be executed in a single cycle.

The register file also contains the address decoders for all four ports. Each address presented to the decoders consists of 8 bits derived from an instruction field and the Current Window Pointer. These are physical addresses into the register file memory array.

Bypass Logic

As shown in Figure 3-7, the A and B operand registers have inputs which come from sources other than the register file or immediate data bus. These inputs are results from previous instructions which have not yet written back to the register file. There are two such *bypass paths* in the E Block:

- *Result Bypass*—The result of an ALU operation in the R register is written back to the A or B operand register in the Memory stage of the following ALU operation.
- *Write Bypass*—The data in the W register is written to the A or B operand register, in the Writeback stage.

The result bypass path is selected when one instruction generates a result that can be used by the immediately following instruction. More precisely, if an instruction in the Decode stage of the pipeline has $rs1 = N$ and the instruction in the Execute stage has $rd = N$, the $rs1$ operand will not come from the register file, but directly from the R register in the ALU through the result bypass. Since an intervening SAVE or RESTORE instruction may have changed the Current Word Pointer, it is the *physical addresses* of the register source and destination which are compared, not the logical addresses (which depend on the CWP).

As an example, consider the instruction sequence:

```
add %r1,%r2,%r3           ; r1 + r2 -> r3
add %r3,%r4,%r5           ; r3 + r4 -> r5
```

The second add instruction takes its A source operand not from the register file but directly from the result of the ALU, through the result bypass.

The write bypass is selected when an instruction in the Decode stage has $rs1 = N$ and the instruction in the Memory stage has $rd = N$. In this case, the $rs1$ operand will not come from the register file, but from the W register through the write bypass. In the following instruction sequence, the third instruction uses the write bypass as its A source operand:

```

add %r1,%r2,%r3          ; r1 + r2 -> r3
add %r4,%r5,%r6          ; r4 + r5 -> r6
add %r3,%r7,%r8          ; r3 + r7 -> r8

```

If both bypass conditions apply, the result bypass takes precedence.

There is a third bypass path, called the *store bypass*. It can be seen in Figure 3-7. The register file has a dedicated store port which is used for reading the rd register of a store instruction; this register contains the data to be stored. The store port is read in the Execute stage of the store. When a store and the immediately preceding instruction access the same rd register, a bypass from the Writeback stage of the preceding instruction to the Memory stage of the store is needed. In the code sample below, the result of the first instruction becomes available to the Memory stage of the store by means of the store bypass path.

```

add %r1,%r2,%r3          ; r1 + r2 -> r3
st  %r3,[%r4+%r5]        ; r3 -> mem[r4 + r5]

```

Branch Evaluation Logic

The branch evaluation logic, which forms part of the E Block, evaluates branch conditions based on the current values of the integer condition codes of the PSR register. The icc bits n (negative), z (zero), c (carry) and v (overflow) form part of the branch evaluation block. The interpretation of these bits is discussed in the *Programmer's Model* chapter.

There are several ways the icc bits can be modified. First of all, they can be written and read via the jump address bus by the instructions WRPSR and RDPSR.

Certain arithmetic instructions modify the icc bits as a side effect. When one of these instructions is executing, the new icc values are generated in the E Block during the Execute stage, latched at the end of this stage, and loaded into the PSR during the Memory stage.

Another path leads to the icc bits from the Writeback-stage copy of the PSR. When a trap occurs on an instruction which alters the icc bits, this path allows the pre-trap icc values to be restored to the PSR.

The combinational logic which does the branch evaluation for the IU condition codes has as inputs:

- *Integer Condition Codes*—Directly from the ALU, if the instruction in the Execute stage is one of those that can modify the icc; from the multiplication unit; or from the icc bits of the PSR, if the instruction in the Execute stage is not one that can modify the icc.

- *The cond Field*—From the branch instruction in the Execute stage. (See the discussion of the Bicc instruction in the *Programmer's Model* chapter.)
- *Bicc Indicator*—A control signal indicating whether or not the instruction in the Decode stage is a Bicc instruction. This signal remains valid into the Execute stage.

The output of the combinational logic is a single signal which, when active, causes the branch target address to be loaded into the PC during the Execute stage; otherwise, PC+4 is loaded into the PC.

Load Align Unit (LAU) and Store Align Unit (SAU)

The LAU and SAU align data for loads and stores, respectively. Bytes and half-words to be loaded are right-justified in a 32-bit word, and either sign-extended or zero-extended on the left, depending on whether the load instruction specified signed or unsigned operation. The LAU performs the alignment and extension during Writeback.

Byte and halfword stores take their data from the least significant byte or halfword of the register specified in the instruction's rd field. The SAU performs the necessary alignment for writing the data to the byte or halfword memory address specified in the instruction.

Multiply Unit

The E Block contains hardware to perform integer multiplications. The Multiply Unit (MU) multiplies two 32-bit signed or unsigned integers to produce a 64-bit product. Some multiplication instructions modify the integer condition codes as a side effect; others do not. The multiplication instructions are discussed in the *Programmer's Model* chapter.

The multiply hardware implements a version of *Booth's algorithm*. Booth's algorithm is similar to a "shift and add" multiply algorithm in that it scans the multiplier from the least significant to the most significant bit and, based on the bit string encountered, iteratively adds the multiplicand to produce partial products. It is also similar in that the resulting partial product is right shifted to ready it for the following iteration of the algorithm. Booth's algorithm differs from a "shift and add" algorithm in that it can also be used directly with a negative multiplier (whereas "shift and add" requires a positive multiplier). It differs also in that the hardware must provide for both addition and subtraction of the multiplicand. In particular, a 1-bit Booth's algorithm examines two multiplier bits per iteration, looks for a bit transition, and either adds the multiplicand, subtracts the multiplicand, or adds zero to the existing partial product to produce the new partial product. It "retires" one bit of the multiplier per iteration. For a 1-bit Booth's, Table 3-3

shows the possible bit transitions encountered in the multiplier and the value which is added to the multiplicand for each transition.

Table 3-3: Booth's Algorithm

Multiplier Bits		Add to Shifted Partial Product
Current	Previous	
0	0	+0
0	1	+multiplicand
1	0	-multiplicand
1	1	+0

This technique can be extended so that more than one bit is examined during a given iteration. In particular, the MU performs an 8-bit Booth's algorithm. It examines 9 bits of the multiplier at a time and, based on the eight transitions of these nine bits, determines what multiple of the multiplicand to add to the old partial product to produce the new partial product. The addition is performed in the ALSU.

The MU produces 8 bits of the final product and "retires" 8 bits of the multiplier per cycle, and therefore requires only 5 cycles to do a 32x32 bit multiply (producing a 64-bit result).

The execution of the instruction is controlled by a synchronous state machine which generates control signals for the multiply hardware. Since instructions do not execute out of order, the Integer Unit (IU) must be frozen during the multiply instructions which take more than 1 cycle. Conceptually, the multiply instruction goes through all the pipeline stages (F,D,E,M,W), but its Execute stage is from 1 to 5 machine cycles long. During the Fetch and Decode stages, the multiply instruction progresses like other instruction.

3.1.4 Programmer-Visible State and Processor State

The SPARC Architecture defines the *programmer-visible state* of the processor as a collection of registers, and then specifies the effects of instructions in terms of these registers. These definitions implicitly assume that every instruction completes before the next one begins. The SPARC*lite* processor, however, is pipelined, so that normally four subsequent instructions begin before the first one completes. The actual *processor state* (excluding the register file) therefore encompasses more than the programmer-visible state. For most of the programmer-visible registers, there is a corresponding register in the processor associated with the Writeback stage of the pipeline. That is, instructions normally update the register file and programmer-visible state registers in the Writeback stage.

An instruction may update staged copies of the PSR before Writeback, making the new values available to subsequent instructions sooner, but these staged copies are not user visible. The PSR associated with the Writeback stage can never be updated early; if an instruction traps, it will not have altered any state which can not be restored.

3.1.5 IU Support for Debugging

The IU supports the on-chip Debug Support Unit as well as external ICE circuitry and software with the following features:

- A special breakpoint trap type instruction_breakpoint/data_breakpoint: This is a synchronous trap with trap type 255. It is analogous to the instruction_access_exception and data_access exception traps, but has the following special characteristics:
 - Any instruction can cause a breakpoint exception (unlike the data_access_exception, which can only occur for load/store instructions).
 - The trap vector for this taken trap is *not* the TBR concatenated with the trap type, but zero concatenated with the trap type. That is, the trap target address is 0x0000FF0 regardless of the value in the TBR.

3.2 Data and Instruction Caches

The SPARClite architecture provides separate data and instruction caches, allowing designers to build high-performance systems without incurring the cost of fast external memory and its associated control logic. The software-visible features of the caches are discussed in detail in the *Programmer's Model* chapter, above.

The data and instruction caches are accessed independently over separate data and instruction buses, allowing data to be loaded from and stored to cache at peak rates of one cycle per instruction. The instruction cache is read-only, one word at a time. The data memory is readable and writable by bytes, halfwords, words or doublewords.

In the MB86930 processor, each cache is 2 Kbytes in size, organized into two banks of sixty-four 16-byte lines. Cache lines are refilled in 4-byte increments to avoid the interrupt latency incurred by long, uninterruptible cache line replacements. In a unified (instruction and data) external memory, the instruction and data memory segments should be at aligned 4-word (line size) boundaries.

The instruction cache has four major RAM arrays. There are two arrays for instruction memory and two arrays for tags. In addition to the tag memory, the tag arrays also contain the logic to compare the address tag with the address that

is being accessed. It also checks the VALID bits in the tag. The hit-detection logic is illustrated in Figure 3-8.

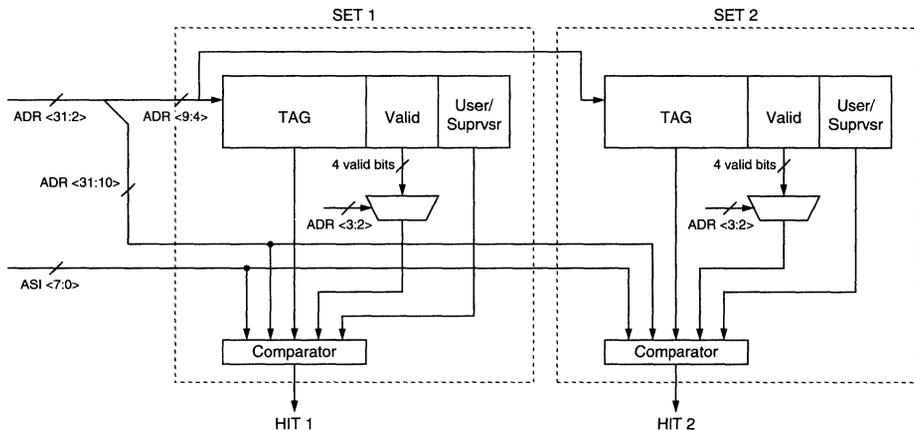


Figure 3-8. Cache Hit Detection Logic

The organization of the data cache is similar to the instruction cache. In addition, the data memory has individual write control for each byte. This makes it possible to do byte or half-word writes without using read-modify-write cycles.

3.3 Bus Interface Unit

The Bus Interface Unit (BIU) contains the logic which allows the processor to communicate with the system. The BIU receives requests for external memory and I/O accesses from the cache control logic. When the BIU performs a read, it returns the data to both the cache and the IU. Parallel paths make the data available to the IU in the same cycle that it is written to the cache. The BIU also handles external requests for control of the bus. The external signals of the BIU, and the relative timing of events in typical bus operations, are discussed in the *External Interface* chapter, below. That chapter also treats the various system-support features of the processor in detail.

3.3.1 Buffers

The BIU has a one-word (32-bit) write buffer to hide external memory latency from the IU. When the BIU receives a request for a write transaction it stores the write data and address in the write buffer and indicates the completion of the write to the IU. It then proceeds to complete the write to external memory. This allows the IU to continue operation from the cache. The write buffer can be

enabled by setting bit 5 of the Cache/BIU Control Register, as discussed in the *Programmer's Model* chapter, above. The write buffer enable bit should be written to, only when the instruction and data caches are off. The write buffer works only when both instruction and data caches are on.

The BIU also has a one-word prefetch buffer for instruction fetches. After an external instruction fetch, the prefetch buffer will initiate an access to the next sequential address, on the next available cycle. Instructions are prefetched only when the BIU does not have a request for a bus transaction from the IU, and no external device is requesting use of the bus. Prefetching is suspended if the buffer is full; this occurs if the prefetched instruction is a hit in the instruction cache or if the prefetched instruction is not used as in the case of a branch to a different address. The buffer restarts again after the next instruction cache miss. If an exception occurs during an instruction prefetch, the exception is not sent to the IU unless the instruction is actually requested by the IU. The prefetch buffer operates only when the instruction cache is on.

3.3.2 Exception Handling

The external memory system can indicate an exception during a memory operation by asserting the -MEXC input. If -MEXC is asserted during an instruction fetch, the BIU indicates an instruction memory exception to the cache control logic and the IU. If -MEXC is asserted during a data fetch, the BIU indicates a data access exception to the cache control logic and the IU.

As indicated above, the IU can continue to operate after putting the data and address for a store into the write buffer. If an exception is detected while completing this buffered write then the BIU indicates a data access exception. Any system which wants to recover from this error should store the address and data for the write causing the exception, in a register. It should also have a status bit to indicate that the exception was caused during a write operation. It will be the responsibility of the data access exception service routine to determine the cause of the exception and recover accordingly.

3.3.3 Effect on the Pipeline

The pipeline hold signals, ihold and dhold , are generated if an instruction or data cannot be made available in the cycle that it is required by the pipeline. Normally ihold and dhold are not asserted if the required instruction or data is already in cache. On the other hand, if a cache miss occurs the cache controller requests that the appropriate data or instruction be fetched from the external system. On a cache miss, the transaction will be available on the bus in the following clock cycle if nothing of higher priority is pending (see below). A bypass exists that allows an

instruction or data word to be made available in the same cycle that it is being written into cache.

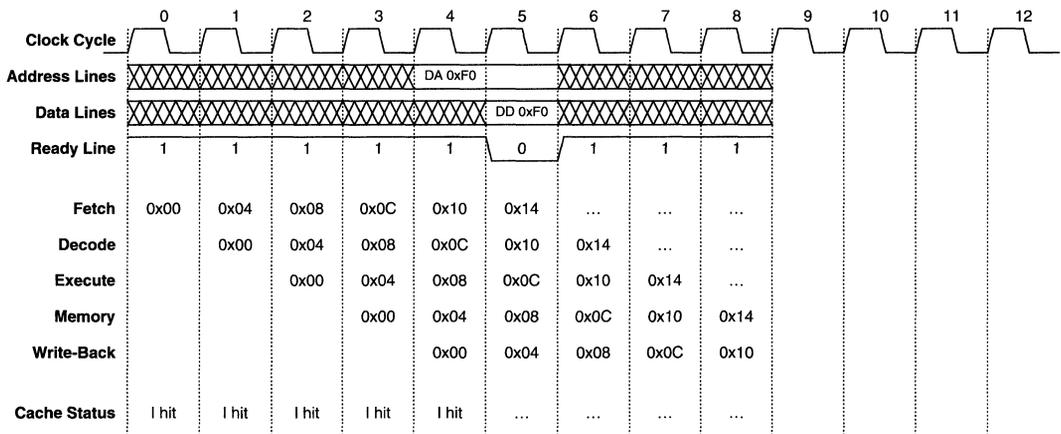
In general the following hierarchy rules apply to the bus interface unit:

- the bus cycle currently in progress will complete
- if the write buffer is full, the buffer will be emptied
- if there is a pending request for a load or store operation it will be serviced
- if there is a pending request for an instruction it will be fetched
- if the prefetch buffer is empty, a prefetch cycle will be initiated

This section illustrates the effect of bus operations on the instruction pipeline for some representative cases.

Case 1: Cache Hits

Figure 3-9 illustrates a sequence of hits in the instruction cache. The instruction fetched in cycle 0 is a STORE to location 0xF0. The data is written to the Write Buffer in cycle 3, and to the bus in cycle 4. Since the write buffer is empty, the pipeline can move at a rate of one instruction per cycle, even when handling a STORE. LOAD instructions also do not hold up the pipeline, provided the source of the load is in the data cache.



Configuration: Instruction Cache: ON Pre-Fetch Buffer: Enabled Memory Wait-State: 1
 Data Cache: - Write Buffer: Enabled

Figure 3-9. Pipeline Operation: Cache Hits

Case 2: Prefetch Buffer Disabled

Figure 3-10 illustrates the operation of the pipeline on instruction cache misses when the prefetch buffer is disabled. The address of each missed instruction is available on the processor external bus in the cycle following the miss. Since data becomes available to both the IU and the cache on the same cycle, the pipeline can proceed in the cycle immediately following the cycle in which the data appears on the external bus.

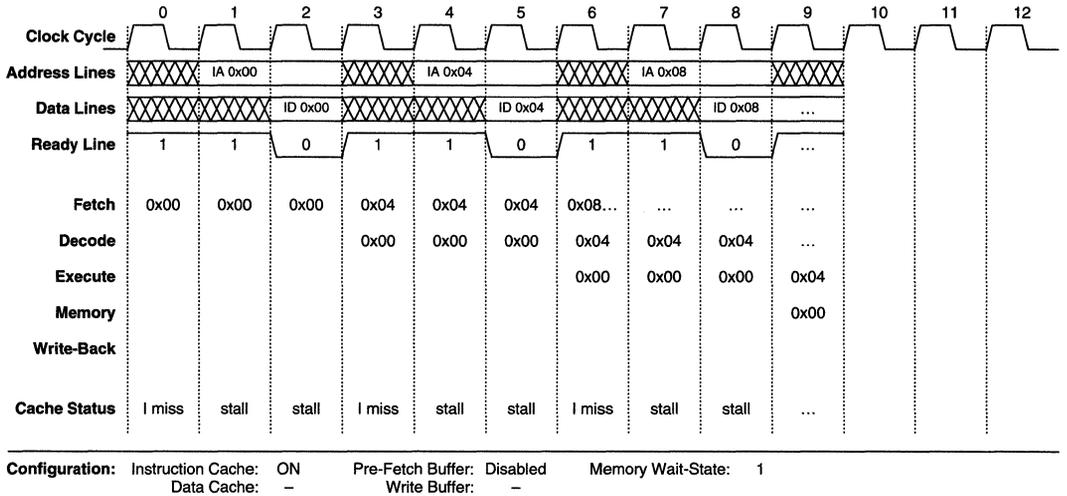


Figure 3-10. Pipeline Operation: Prefetch Buffer Disabled

Case 3: Prefetch Buffer Enabled

Figure 3-11 illustrates the operation of the pipeline on instruction cache misses when the prefetch buffer is enabled. The address of the instruction missed on cycle 0 is available on the system bus in cycle 1. In cycle 3, the pre-fetch buffer logic drives the next sequential word address onto the address lines. The instruction cache miss at this location therefore causes the pipeline to be stalled for only one cycle. Contrast this with Case 2, above. Since the prefetched instruction is actually used by the processor, the prefetch buffer drives the next sequential word address in cycle 5. This saves a cycle on each access when executing sequential code not already in cache.

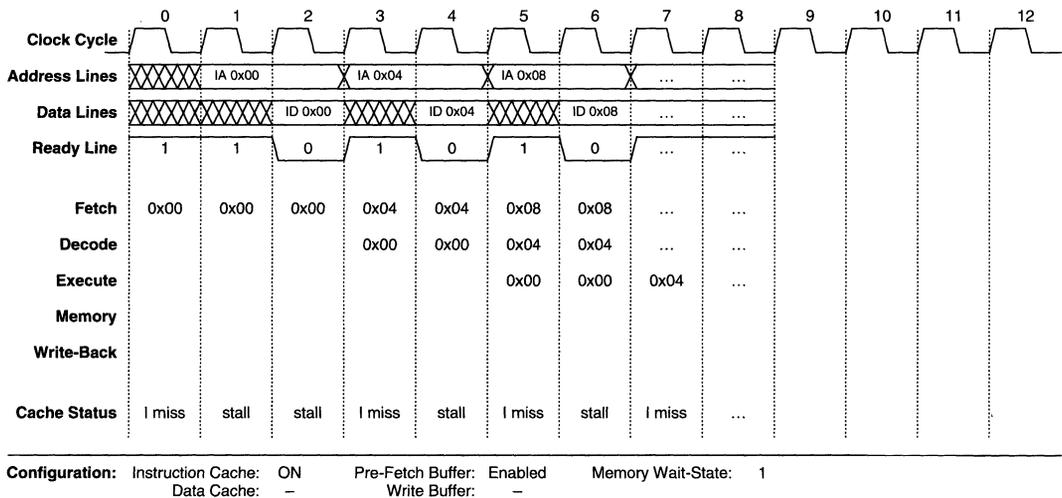


Figure 3-11. Pipelined Operation: Prefetch Buffer Enabled

Case 4: Data Cache Off

Figure 3-12 illustrates the operation of the pipeline on loads, with the data cache turned off and the instruction cache turned on. The instruction fetched in cycle 0 is a LOAD from memory location 0xF0. The data is fetched when this instruction reaches the Memory stage in cycle 7. Since the data cache is off, the data must be fetched externally; this delays the next instruction fetch until cycle 9.

Whenever a prefetch operation is held up by a load or store operation, the pre-fetch buffer address gets updated if the instruction it is pointing to is a hit in the instruction cache. Therefore, when prefetch starts at cycle 9 the IA0x10 instruction address goes out on the address bus instead of 0x0c which has already hit in the cache.

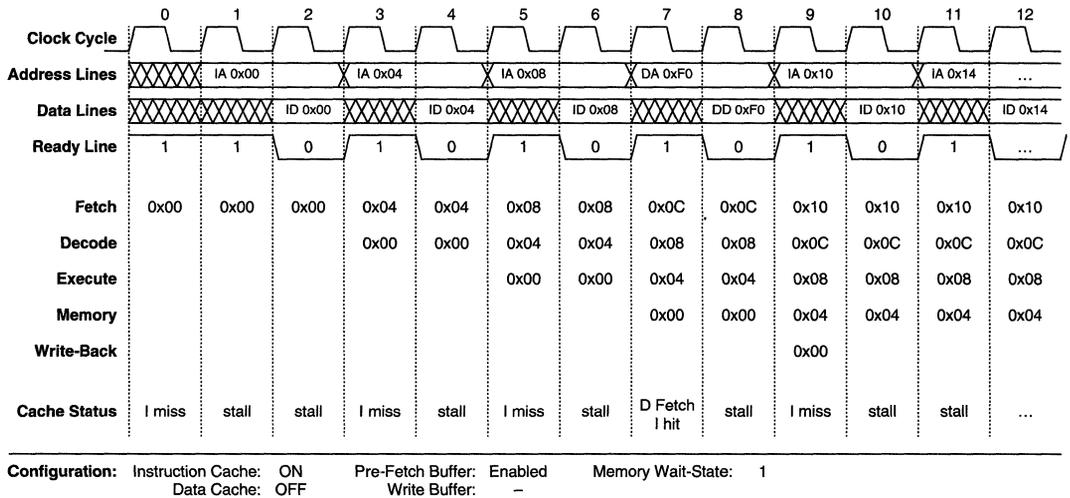
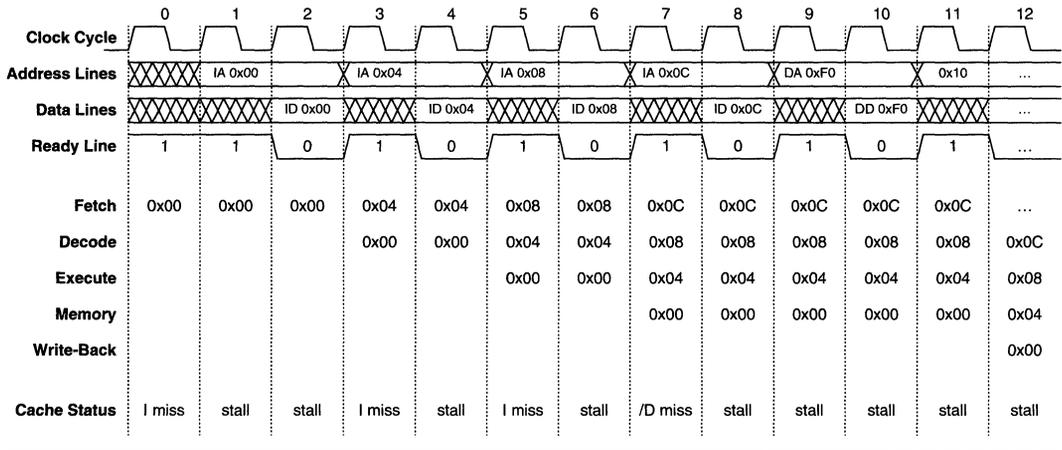


Figure 3-12. Pipeline Operation: LOAD with Data Cache Turned Off

Case 5: Data Cache Miss

Figure 3-13 illustrates the operation of the pipeline on loads, when the data access misses in the cache. The instruction fetched in cycle 0 is a LOAD from memory location 0xF0. The data is required when this instruction reaches the Memory stage in cycle 7. The access misses in the cache, so the data must be fetched externally. At cycle 7, the prefetch operation has already started so the external load operation is delayed until the prefetch completes. At cycle 9, the external load operation takes place. At cycle 11, the now empty prefetch buffer initiates the next sequential instruction fetch at address 0x10.



Configuration: Instruction Cache: ON Pre-Fetch Buffer: Enabled Memory Wait-State: 1
 Data Cache: ON Write Buffer: -

Figure 3-13. Pipeline Operation: Data Cache Miss

CHAPTER 4



External Interface

The processor's external interface consists of signals, bus operations, and system support functions. This chapter details the MB86930 signal set, gives the relative timing of events in the principal types of bus operation, and describes the programmable wait-state generator, on-chip timer, and same-page detection logic. For specific electrical and timing values, see the MB86930 Data Sheet. The System Design Considerations chapter of this document discusses issues that are likely to arise in the design of any SPARClite system.

4.1 Signals

The processor's external signals are illustrated in Figure 1-6 of the *Overview* chapter, and are listed in Table 4-1 below. A dash at the beginning of a signal name, as in *-RESET*, indicates that the signal is active-low.

Table 4-1: Input and Output Signals

Symbol	Type	Symbol	Type	Symbol	Type	Symbol	Type
ADR <31:2>	O S(L) G(Z) I (1)	-CS0, -CS1 -CS2, -CS3 -CS4, -CS5	O S(L) G(1) I (1)	-LOCK	O S(L) G(Z) I (1)	TDO	O
-AS	O S(L) G(Z) I (1)	D <31:0>	I/O S(L) G(Z) I (Z)	-MEXC	I S(L)	-TIMER_OVF	O S(L) G(Q) I (Q)
ASI <7:0>	O S(L) G(Z) I (1)	EMU_BRK	I	-SAME_PAGE	O S(L) G(1) I (1)	TMS	I
-BE 3:0	O S(L) G(Z) I (0)	EMU_D<3:0>	I/O	RD/-WR	O S(L) G(Z) I (1)	-TRST	I
-BGRNT	O S(L) G(0) I (Q)	-EMU_ENB	I	-READY	I S(L)	XTAL1 (CLKIN) XTAL2	I O G(Q) I (Q)
-BREQ	I S(L)	EMU_SD <3:0>	I/O	-RESET	I A(L)		
CLKOUT1 CLKOUT2	O G(Q) I (Q)	-ERROR	O S(L) G(Q) I (Q)	TCK	I		
CLK_ECB	I	IRL <3:0>	I A(L)	TDI	I		

NOTE:

- I = Input Only Pin
- O = Output Only Pin
- I/O = Either Input or Output Pin
- = Pins "must be" connected as described
- S(L) = Synchronous: Inputs must meet setup and hold times relative to CLKIN. Outputs are Synchronous to CLKIN
- A(L) = Asynchronous: Inputs may be asynchronous to CLKOUT.
- G(...) = While the bus is granted to another bus master (-BGRNT=asserted), the pin is
 - G(1) is driven to V_{CC}
 - G(0) is driven to V_{SS}
 - G(Z) floats
 - G(Q) is a valid output
- I(...) = While the bus is between bus cycles (or being reset) and is not granted to another bus master, the pin is
 - I (1) is driven to V_{CC}
 - I (0) is driven to V_{SS}
 - I (Z) floats
 - I (Q) is a valid output

The following sections describe the signal set in detail, arranged by functional group:

- Processor Control and Status—Reset, error, and clock signals.
- Memory Interface—Data and address buses, ASI and byte-enables, chip-selects, and other control signals used to access external memory and memory-mapped devices.
- Bus Arbitration—Signals used by external devices in requesting, and by the processor in granting, control of the bus.
- Peripheral Functions—Interrupt-requests and timer overflow.
- Emulator Bus—Signals to support in-circuit emulation.
- Boundary-Scan—Test signals used for board verification, following JTAG specifications.

4.1.1 Processor Control and Status

Signal	Function
CLKOUT1 CLKOUT2	CLOCK OUTPUTS (O): MB86930 bus transactions can be referenced against these outputs. CLKOUT1 has the same frequency and phase as the internal oscillator, or the signal applied to CLKIN. CLKOUT2 is the same as CLKOUT1, but phase-shifted 180 degrees.
–ERROR	ERROR SIGNAL (O): Asserted by the CPU to indicate that it has halted in an error state as a result of encountering a synchronous trap while traps are disabled. In this situation, the CPU saves the Trap Type (tt) value in the Trap Base Register, enters into an error state and asserts the –ERROR signal. The system can monitor the –ERROR pin and initiate a reset to recover from the error condition.
–RESET	SYSTEM RESET (I): Resets the processor to a known internal state. –RESET should be asserted for at least 4 processor cycles after the clock has stabilized. The internal state of the processor immediately after reset is described in the <i>Programmer's Model</i> chapter.
XTAL1 (CLKIN) XTAL2	EXTERNAL OSCILLATOR (XTAL1, XTAL2): Determines the execution rate and timing of the processor. Connecting a crystal across these pins forms a complete crystal oscillator circuit. The processor operating frequency is the same as the crystal oscillator frequency. The processor can also be driven by an external clock. In this case, the clock signal is applied to XTAL1 (CLKIN); XTAL2 should be left unconnected. The processor operating frequency is the same as the external clock frequency.

4.1.2 Memory Interface

Signal	Function																														
ADR[31:2]	ADDRESS BUS (O): Specifies the data or instruction address of a 32-bit word. Reads are always one word in size while byte, half-word, or word transaction sizes for writes are identified by separate byte-enable signals (–BE3-0). The value on the address bus is valid for the duration of the bus transaction. See note below.																														
–AS	ADDRESS STROBE (O): Asserted by the MB86930 or other bus master to indicate the start of a new bus transaction. A bus transaction begins with the assertion of –AS and ends with the assertion of –READY. During cycles in which neither the processor nor another bus master is driving the bus, the bus is idle, and –AS remains de-asserted. See Table 4-1 for signal values while the bus is idle. The MB86930 asserts –AS for 1 clock cycle.																														
ASI[7:0]	<p>ADDRESS SPACE IDENTIFIERS (O): Indicates which of the 256 available address spaces the current bus transaction is accessing. The ASI values are defined as follows:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>ASI <7:0></th> <th>ADDRESS SPACE</th> </tr> </thead> <tbody> <tr> <td>0x1</td> <td>Control Register</td> </tr> <tr> <td>0x2</td> <td>Instruction Cache Lock</td> </tr> <tr> <td>0x3</td> <td>Data Cache Lock</td> </tr> <tr> <td>0x4 - 0x7</td> <td>Application Definable</td> </tr> <tr> <td>0x8</td> <td>User Instruction Space</td> </tr> <tr> <td>0x9</td> <td>Supervisor Instruction Space</td> </tr> <tr> <td>0xA</td> <td>User Data Space</td> </tr> <tr> <td>0xB</td> <td>Supervisor Data Space</td> </tr> <tr> <td>0xC</td> <td>Instruction Cache Tag RAM</td> </tr> <tr> <td>0xD</td> <td>Instruction Cache Data RAM</td> </tr> <tr> <td>0xE</td> <td>Data Cache Tag RAM</td> </tr> <tr> <td>0xF</td> <td>Data Cache Data RAM</td> </tr> <tr> <td>0x10 - 0xFC</td> <td>Application Definable</td> </tr> <tr> <td>0xFD - 0xFF</td> <td>Reserved for Debug Hardware</td> </tr> </tbody> </table> <p>The ASI values specified as “application definable” can be used by privileged (supervisor mode) instructions such as load and store alternate. The ASI value is available in the same cycle in which the corresponding address value is asserted on the address bus. The values on the ASI pins are valid for the duration of the bus transaction. Transactions with ASI values of 0x8, 0x9, 0xA, and 0xB are cacheable. See note below.</p>	ASI <7:0>	ADDRESS SPACE	0x1	Control Register	0x2	Instruction Cache Lock	0x3	Data Cache Lock	0x4 - 0x7	Application Definable	0x8	User Instruction Space	0x9	Supervisor Instruction Space	0xA	User Data Space	0xB	Supervisor Data Space	0xC	Instruction Cache Tag RAM	0xD	Instruction Cache Data RAM	0xE	Data Cache Tag RAM	0xF	Data Cache Data RAM	0x10 - 0xFC	Application Definable	0xFD - 0xFF	Reserved for Debug Hardware
ASI <7:0>	ADDRESS SPACE																														
0x1	Control Register																														
0x2	Instruction Cache Lock																														
0x3	Data Cache Lock																														
0x4 - 0x7	Application Definable																														
0x8	User Instruction Space																														
0x9	Supervisor Instruction Space																														
0xA	User Data Space																														
0xB	Supervisor Data Space																														
0xC	Instruction Cache Tag RAM																														
0xD	Instruction Cache Data RAM																														
0xE	Data Cache Tag RAM																														
0xF	Data Cache Data RAM																														
0x10 - 0xFC	Application Definable																														
0xFD - 0xFF	Reserved for Debug Hardware																														
Note:	Care must be taken to ensure that software written for SPARClite processors with 32 address and 8 ASI external signals operates correctly with the MB86933 processor, which has only 28 address bits and 4 ASI bits. Inadvertent attempted use of unavailable address and ASI space (i.e. using bits ADR<31:28> and ASI<7:4>) can be detected by programming an MB86933 –CS output to assert to 0 when the high address and ASI nibbles are 0 (not used). External diagnostic hardware, such as a logic analyzer, can then be used to detect when –CS is not asserted, indicating possible use of address and ASI signals that are not available on the MB86933. The –CS signal can be gated with all other –CS signals that are in use to determine if the access is off-chip. If so, the access may be illegal.																														

Signal	Function																
-BE3-0	<p>BYTE ENABLES (O): Indicate whether the current load or store transaction is a byte, half-word or word transaction. The BYTE ENABLE value is available in the same cycle in which the corresponding address value is asserted on the address bus. The values on the byte enable pins are valid for load and store operations and for the duration of the bus transaction (the byte enable signals can be ignored during load operations).</p> <p>Possible values for -BE3-0 are as follows:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td></td> <td style="text-align: center;">31</td> <td></td> <td style="text-align: center;">0</td> </tr> <tr> <td>Byte Writes</td> <td>1 1 1 0</td> <td>1 1 0 1</td> <td>1 0 1 1</td> </tr> <tr> <td>Half-Word Writes</td> <td>1 1 0 0</td> <td>0 0 1 1</td> <td></td> </tr> <tr> <td>Word Writes</td> <td></td> <td>0 0 0 0</td> <td></td> </tr> </table>		31		0	Byte Writes	1 1 1 0	1 1 0 1	1 0 1 1	Half-Word Writes	1 1 0 0	0 0 1 1		Word Writes		0 0 0 0	
	31		0														
Byte Writes	1 1 1 0	1 1 0 1	1 0 1 1														
Half-Word Writes	1 1 0 0	0 0 1 1															
Word Writes		0 0 0 0															
-CS[5-0]	<p>CHIP SELECTS (O): One of these signals is asserted when the value on the address bus lies in the range specified by the corresponding Address Range Specifier Register. The -CS signals are used to decode the current address into one of eight address ranges. Address ranges should not overlap. Each address range has a corresponding wait-state specifier which is used to generate an internal -READY signal after a user-defined number of processor clock cycles. This allows a variety of memory and I/O devices with different access times to be connected to the MB86930 without the need for additional logic. CS0 is enabled at reset (See Chapter 2).</p>																
D[31:0]	<p>DATA BUS (I/O): D31 corresponds to the most significant bit of Byte 0. D0 corresponds to the least significant bit of byte 3. A double word is aligned on an 8-byte boundary, a word is aligned on a 4-byte boundary, and a half-word is aligned on a 2-byte boundary. If a load or store of any of these quantities is not properly aligned, a mem_address_not_aligned Trap will occur in the processor.</p> <p>During write cycles, the point at which data is driven onto the bus depends on the type of the preceding cycle. If the preceding cycle was a write, data is driven in the cycle immediately following the cycle in which -READY was asserted. If the preceding cycle was a read, data is driven one cycle after the cycle in which -READY was asserted, in order to minimize bus contention between the processor and the system.</p>																
-LOCK	<p>BUS LOCK (O): Asserted by the processor to indicate that the current bus transaction requires more than one transfer on the bus. The Atomic Load Store instruction, for example, requires contiguous bus transactions and so causes the BUS LOCK signal to be asserted. The bus will not be granted to another bus master as long as -LOCK is active. -LOCK is asserted with the assertion of -AS and remains active until -READY is asserted at the end of the locked transaction</p>																
-MEXC	<p>MEMORY EXCEPTION (I): Asserted by the memory system to indicate a memory error on either a data or instruction access. Assertion of this signal initiates either a Data or Instruction Access Exception trap in the IU. The current bus access is invalidated by asserting the -MEXC in the same cycle as the -READY signal. The IU ignores the value on the data bus in cycles where -MEXC is asserted.</p>																

Signal	Function
RD/-WR	READ/WRITE BUS TRANSACTION (O): Specifies whether the current bus transaction is a read or a write operation. When -AS is asserted and RD/-WR is high, then the current transaction is a read. With -AS asserted and RD/-WR low, the current transaction is a write. RD/-WR remains active for the duration of the bus transaction and is de-asserted with the assertion of -READY.
-READY	READY (I): Asserted by the external memory system to indicate that the current bus transaction is being completed and that it is ready to start with the next bus transaction in the following cycle. In case of a fetch from memory, the processor will strobe the value on the data bus at the rising edge of CLKIN following the assertion of -READY. In the case of a write, the memory system will assert -READY when the appropriate access time has been met. In most cases, no external logic is required to generate the -READY signal. On-chip circuitry can be programmed to assert -READY internally, based on the address of the current transaction. The external system can override the internal ready generator to terminate the current bus cycle early. Up to 6 address ranges each with different transaction times can be programmed. (See the <i>System Support Functions</i> section, below.)
-SAME_PAGE	SAME-PAGE DETECT (O): Asserted when the address of the current memory access is within the same page as the previous memory access. -SAME_PAGE can be used to take advantage of fast consecutive accesses within page-mode DRAM page boundaries. -SAME_PAGE is asserted with -AS and remains active for one processor cycle. -SAME_PAGE is never asserted in the first transaction following a transaction by another device on the bus. The page size is specified by writing the Same-Page Mask Register. (See the <i>System Support Functions</i> section, below.)

4.1.3 Bus Arbitration

Signal	Function
-BGRNT	BUS GRANT (O): Asserted by the CPU in response to a request from a device wanting ownership of the bus. The CPU grants the bus to other devices only after all transfers for the current transaction are completed. Refer to the data sheet for the output signal states after the assertion of the BUS GRANT signal.
-BREQ	BUS REQUEST (I): Asserted by another device on the bus to indicate that it wants ownership of the bus. The request must be answered with a bus grant (-BGRNT) from the MB86930 before the device can proceed by driving the bus. Once the bus has been granted, the device has ownership of the bus until it de-asserts -BREQ. The user should ensure that devices on the bus do not monopolize the bus to the exclusion of the CPU. The assertion of -BREQ is recognized by the processor even when -RESET is being asserted.

4.1.4 Peripheral Functions

Signal	Function
IRL[3:0]	INTERRUPT REQUEST BUS (I): The value on these pins defines the external interrupt level. IRL[3:0]=1111 forces a non-maskable interrupt. An IRL value of 0000 indicates no pending interrupts. All other values indicate maskable interrupts as enabled in the Processor Interrupt Level field of the Processor Status Register (PSR). Interrupts should be latched and prioritized by external logic and should be held pending until acknowledged by the processor. An interrupt controller is available on the MB86940 peripheral chip. IRL inputs are sampled by the processor in cycle 1, synchronized in the following cycle, and recognized by the processor in the third cycle.
-TIMER_OVF	TIMER OVERFLOW (O): Indicates that the processor's internal 16-bit timer has overflowed. This signal can be used to initiate a DRAM refresh cycle or a one-cycle periodic waveform. On reset, the timer is turned off and -TIMER_OVF is high.

4.1.5 Emulator Bus

Signal	Function
-EMU_BRK	EMULATOR BREAK REQUEST LINE (I): Used to configure the debug unit on reset. See Section 2.8. This pin should be left unconnected.
EMU_D[3:0]	EMULATOR DATA BITS (O): Reserved. These pins should be left unconnected.
-EMU_ENB	EMULATOR ENABLE (I): Used to configure the debug unit on reset. See Section 2.8. This pin should be left unconnected.
EMU_SD[3:0]	EMULATOR STATUS/DATA BITS (I/O): Reserved. These pins should be left unconnected.

4.1.6 Test and Boundary-Scan

Signal	Function
-CLK_ECB	EXTERNAL CLOCK BYPASS (I): When tied high, causes the CLKIN signal to bypass the on-chip phase-locked loop. This signal is intended primarily for testing the chip.
TCK	TEST CLOCK (I): JTAG compatible test clock input.
TDI [†]	TEST DATA IN (I): JTAG compatible test data input.
TDO [†]	TEST DATA OUT (O): JTAG compatible test data output.
TMS [†]	TEST MODE (I): JTAG compatible test mode select pin.
-TRST [†]	TEST RESET (I): Asynchronous reset for JTAG logic. If not using JTAG, this signal must be pulled low.

†. See appendix for more information

4.2 Bus Operation

At any given time, the Bus Interface Unit is handling requests for external memory and I/O operations, arbitrating for bus access, or idle. From the point of view of the external system, bus transactions are handled in fairly standard ways:

- **Memory and I/O Operations**—Read and write transactions are initiated with the processor asserting the -AS signal. The RD/-WR output indicates the transaction type. The $\text{-BE}[3:0]$ outputs indicate the transaction width. The processor drives the address and ASI signals, and either drives (on stores) or reads (on loads) the signals on the data bus. The transaction ends when -READY is asserted.

An atomic load-store is executed as a load followed by a store, with no operation allowed in between. The -LOCK output is asserted to indicate that the bus is being used for more than one consecutive memory operation.

- **Arbitration**—Any external device can request ownership of the bus by asserting the -BREQ signal. The processor three-states its bus drivers and asserts -BGRNT to indicate that it is relinquishing control of the bus. On completion of its transaction, the external device de-asserts -BREQ ; the processor responds by de-asserting -BGRNT in the following cycle.

The BIU receives requests for external memory operations from the Cache Control Logic. In the case of reads from external memory, it performs the read operation and returns the data to the Cache and IU. A parallel path is used to make the data available to the IU in the same cycle that it is written to the cache.

In the case of a write to external memory, the BIU makes use of a write buffer which can hold a one word write transaction. When the BIU receives a request for a write transaction, it stores the write data and address in the write buffer, allowing the IU to continue operating out of on-chip cache. The BIU then proceeds to complete the write to external memory. In most cases the write buffer will hide external memory latency from the IU. The exceptions are in cases where the write buffer is still filled from a previous transaction or if the subsequent IU cycle results in an instruction cache miss. In these cases, IU execution is held until the write buffer is emptied. The write buffer operates only when the instruction and data caches are both on.

The BIU includes a one stage prefetch buffer for instruction fetches. This buffer is used to fetch the next sequential instruction after an instruction cache miss. The instruction is prefetched only if the BIU does not have a request for a bus transaction from the IU nor is any external device requesting use of the bus. The prefetch buffer operation is suspended if the buffer is full. This occurs if the prefetched instruction is a hit in the instruction cache or if a control transfer causes the sequential instruction to be skipped. The buffer restarts after another instruction cache miss. If an exception occurs during an instruction prefetch, the exception is

not sent to the IU unless the instruction is actually requested by the IU. The prefetch buffer operates only when the instruction cache is on.

In any cycle the BIU can receive a request for accesses to either or both instruction and/or data memory. If it receives a request for both in the same cycle, it completes the data memory transaction first.

4.2.1 Exception Handling

The external memory system can indicate an exception during a memory operation. The BIU signals the appropriate data or instruction exception to the IU which will trap accordingly.

As mentioned above, the IU can continue operation after putting the data and address for a store in the write buffer. If an exception is detected while completing this buffered write, then the BIU indicates a data access exception to the IU.

Any system which needs to recover from this error should store the address and data of such write transactions in hardware. If the system can generate both read and write exceptions, then the system must also provide a status bit which indicates whether the exception was generated on a read or on a write transaction. With access to this information the data access exception service routine can determine the cause of the exception and recover accordingly.

4.2.2 Bus Cycles

This section presents the relative timing of events in representative bus transactions.

Load

Whenever an instruction fetch or a load from data memory has a miss in the cache, the BIU performs a read from external memory.

A read transaction begins with the BIU asserting -AS , to indicate a new bus transaction. The -AS signal is de-asserted after one cycle. At the same time the $\text{ADR}<31:2>$ and $\text{ASI}<7:0>$ bits are driven with the location to be read. The BIU drives the $\text{RD}/\text{-WR}$ signal high to indicate a read transaction. Note that the -BE lines indicate byte, halfword or word operations during load operations although their use is optional. The processor loads a word regardless of the size of data requested (byte, halfword, word).

The external memory system responds with the read data on pins $\text{D}<31:0>$. It also asserts the -READY signal when the data is ready (unless internal ready generation is selected). For slow memory, the -READY signal is delayed until data is valid.

A load double operation is treated as back-to-back reads.

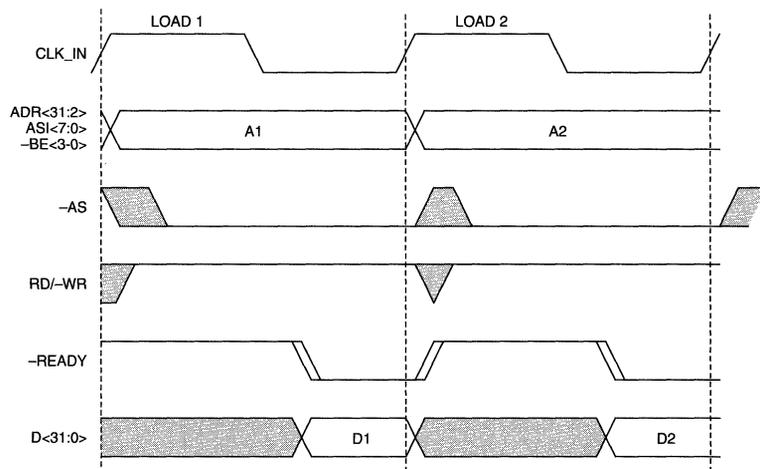


Figure 4-1. Load Timing

Load with Exception

If the external memory system sees a memory exception, it can terminate the current memory transaction by asserting the -MEXC and -READY signals. The data on the data bus is ignored by the MB86930.

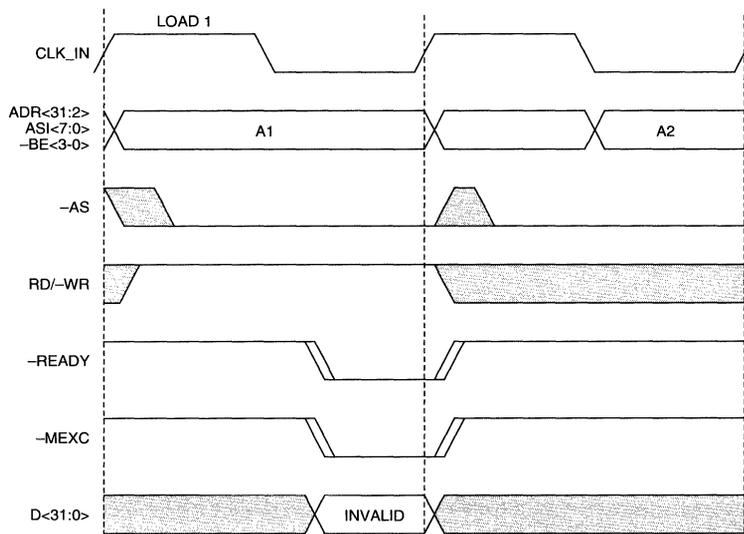


Figure 4-2. Load with Exception Timing

Store

A write transaction begins with the BIU asserting -AS , to indicate a new bus transaction. The -AS signal is de-asserted after one phase. At the same time the $\text{ADR}\langle 31:2 \rangle$ and $\text{ASI}\langle 7:0 \rangle$ pins are driven with the location to be written while the $\text{D}\langle 31:0 \rangle$ pins has corresponding write data. The $\text{-BE}\langle 3:0 \rangle$ pins indicate byte, half-word or word transaction width. The BIU drives the $\text{RD}/\text{-WR}$ signal low to indicate a write transaction.

The external memory system responds by asserting the -READY signal when it has stored the data. There is always one idle bus cycle between the termination of a read cycle and the beginning of a write cycle to provide time for switching of the data bus drivers.

A store double operation is treated as back-to-back writes.

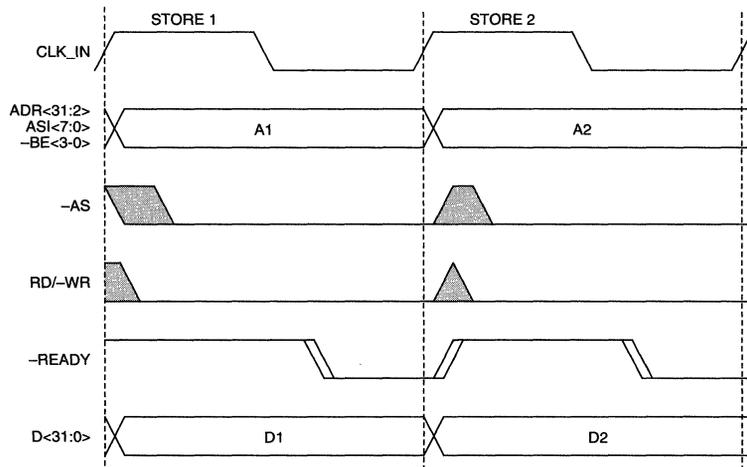


Figure 4-3. Store Timing

Store with Exception

If an access exception occurs on a write, the external memory system can terminate the current memory transaction by asserting the -MEXC and -READY signals. The external memory system is expected to ignore the data on the data bus in this situation.

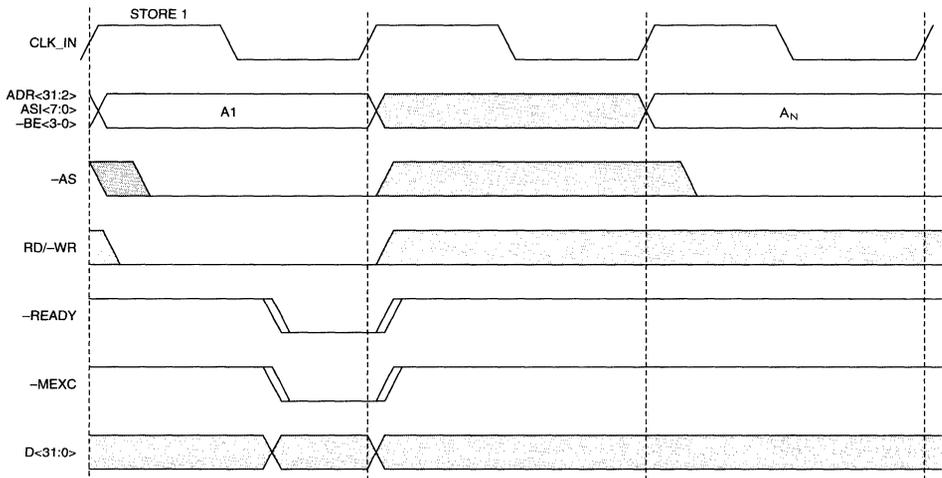


Figure 4-4. Store with Exception Timing

Atomic Load Store

An atomic load store executes as a load followed by a store with no operation allowed in between. The -LOCK signal is asserted to indicate that the bus is being used for more than one external memory operation.

There is one cycle between the termination of the read and the beginning of the write to provide time for the switching of the data bus drivers.

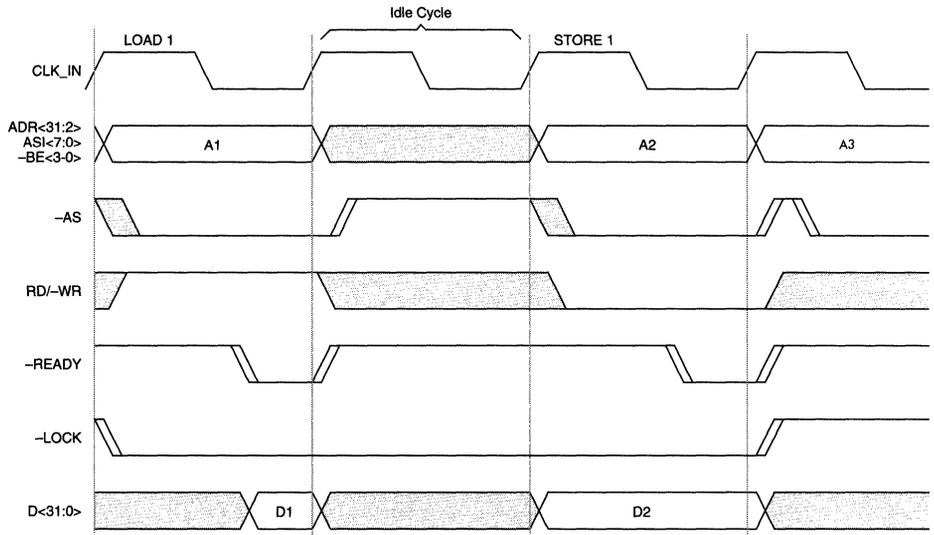


Figure 4-5. Atomic Load Store Timing

External Bus Request and Grant

Any external device can request ownership of the bus by asserting the -BREQ signal. The BIU asserts the -BGRNT signal to indicate that it is relinquishing control of the bus and also three-states all of its bus drivers. In the following cycle, the external device can complete its transaction. On completion of its transaction the external device de-asserts the -BREQ signal. The BIU responds by de-asserting the -BGRNT signal in the following cycle.

The MB86930 is the default owner of the bus.

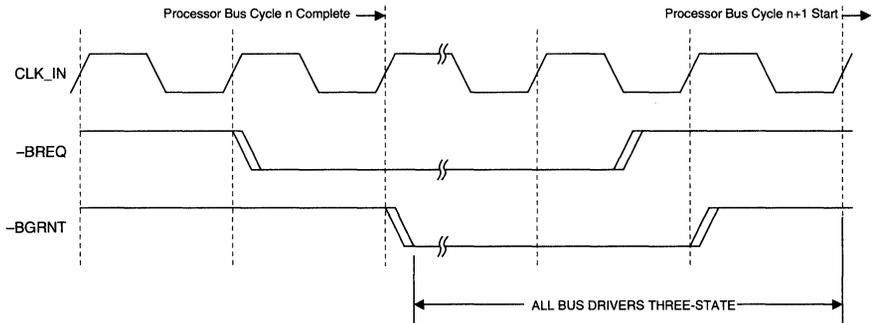


Figure 4-6. External Bus Request and Grant Timing

Processor Reset

The MB86930 is reset by asserting the -RESET signal for a minimum of 4 clock cycles (see Figure). Systems using an external crystal to clock the processor should be sure that -RESET is asserted for at least 4 cycles after the crystal has started up and has stabilized.

If the processor is reset following a halt in Error Mode, and if power to the processor is not removed, the *tt* field after reset will contain the value of the Trap that caused the processor to halt.

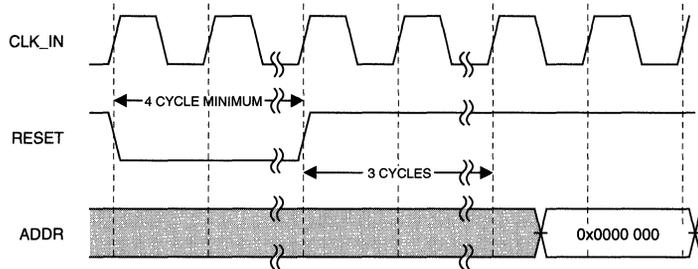


Figure 4-7. Reset Timing

4.3 System Support Functions

Built-in system support functions help to minimize the amount of glue logic required in the external system. The support includes programmable chip select logic, programmable wait-state generation, same-page detection logic and a timer for generating refresh requests. For a more detailed description of the programming of these registers refer to chapter 2.

The System Support Control Register turns the various system support features on and off.

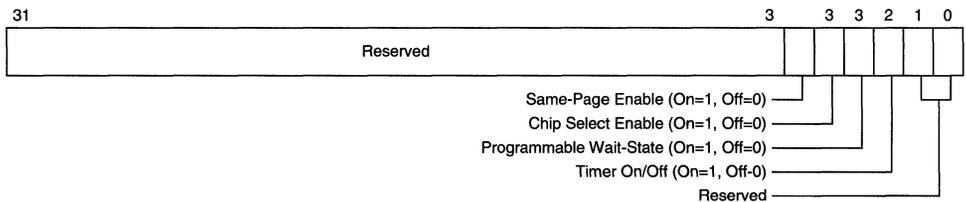


Figure 4-8. System Support Control Register

4.3.1 System-Configuration Registers

The system-configuration registers (Address Range Specifiers, Address Masks, and Programmable Wait-State Specifiers) allow software to define six different address ranges. When an address driven by the processor is in one of these ranges, the corresponding Chip-Select (-CS) pin is asserted. After a number of clock cycles determined by the corresponding Programmable Wait-State Specifier, the processor automatically generates an internal -READY signal. This

makes it possible for memory and I/O devices with different access times to be connected to the processor without additional logic.

The contents of the Address Range Specifier Registers 1-5 (ARSR[5:0]) define five of the six address ranges. An additional address range is available, corresponding to -CS0 . For this address range, ADR is hardwired to 0, and ASI is hardwired to 0x9 (Supervisor Instruction Space). With Mask Register AMR0, -CS0 ranges 8K words. -CS0 is enabled at reset. -CS1 , -CS2 , -CS3 , -CS4 and -CS5 are disabled at reset.



Figure 4-9. Address Range Specifier Register Format

An Address Mask Register is associated with each address range. Any address driven by the chip is compared with the value in all address range specifiers. Only those bits of the register are compared for which the corresponding mask bits are 0. If the specified bits of the current address match one of the address range specifiers, the corresponding chip-select (-CS) pins are asserted. When no bus transaction is being performed, all the -CS pins are high (inactive). The Address Mask Register corresponding to -CS0 is initialized to compare all bits except ADR<14:10>.



Figure 4-10. Address Mask Register Format

A Programmable Wait-State Specifier is associated with each address range. Three registers are used to specify the wait states for the six address ranges. Each register contains the wait-state specifiers for two address ranges.

When the address currently being driven by the processor matches the unmasked bits in one of the Address Range Specifiers, the corresponding wait-state specifier is selected. The format of Wait-State Specifier Registers is shown in Figure 4-11.

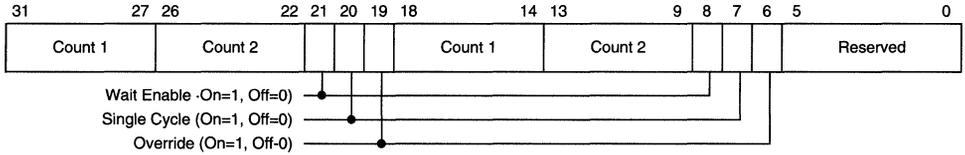


Figure 4-11. Wait-State Specifier Register Format

If the Single Cycle bit equals 1, an internal -READY signal is generated in the same cycle. If the Single Cycle bit equals 0, and the current transaction is in the same page as the previous transaction (see the *Same-Page Detection Logic* section, below), then $\text{Count2} + 1$ is used as the number of cycles after which -READY is asserted internally. If the transaction is not in the same page, $\text{Count1} + 1$ is used instead. If the Wait Enable bit equals 0, the internal -READY is not asserted.

The Override bit allows the user to terminate a transaction earlier than the specified time. If this bit equals 1, and external hardware asserts the external -READY signal, then the wait-state generator will stop counting and will wait for the next transaction, which can occur as soon as the next clock cycle.

The Count1 and Count2 fields of the Wait-State Specifier corresponding to -CS0 have all their bits set to 1 on reset. In this way, 32 wait-state cycles (the maximum number) are inserted into the processor's first instruction accesses. The override bit for -CS0 is enabled as well.

4.3.2 Same-Page Detection

The same-page detection logic determines whether the address of the current memory transaction is on the same page as the previous transaction. If it is, the processor asserts the -SAME_PAGE signal. The system can then take advantage of the fast consecutive accesses possible within fast-page mode DRAM page boundaries. The same-page detection logic consists of a mask register, a register to store the address and ASI bits of the previous transaction, and a comparator.

The Same-Page Mask Register specifies which bits of the current address and ASI must be compared with the previous address and ASI. Only those bits are compared for which the mask bit is 0.

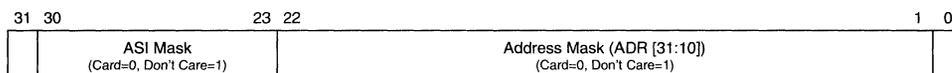


Figure 4-12. Same-Page Mask Register

The `-SAME_PAGE` signal is never asserted for the first transaction following a transaction by another device on the bus. When using the internal wait-state generator, DRAM control logic should issue a bus request when initiating a refresh cycle so that the `-SAME_PAGE` logic is reset appropriately. The `-SAME_PAGE` feature is disabled at reset.

4.3.3 Programmable Timer

The 16-bit programmable timer causes the `-TIMER_OVF` output signal to be asserted at software-defined intervals. This signal can be used to initiate DRAM refresh cycles, or to control other periodic events in the external system.

The current timer count is kept in the Timer Register. When the timer overflows, it is loaded with the value in the Timer Preload Register. The contents of both of these registers are undefined on reset.

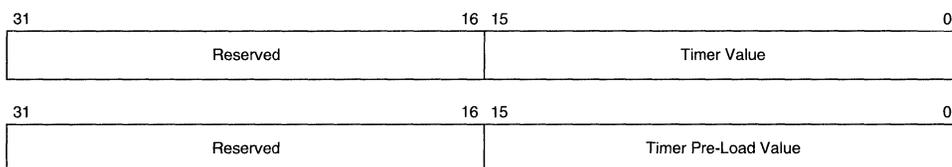


Figure 4-13. Timer and Timer Preload Registers

The timer can also be loaded by writing directly to the Timer Register. The timer can be turned off by writing a 0 to the Timer On/Off bit in the System Support Control register. The timer is clocked at the processor clock frequency.

CHAPTER 5

Programming Considerations

This chapter gives programmers information and advice about how to make the best use of SPARC_{lite} processors. It discusses the initialization of a SPARC_{lite} system, the design of trap handlers, window management, the use of on-chip cache, and SPARC_{lite}-specific instructions.

Because of the availability of high-performance optimizing compilers, real-time operating systems, target monitors and application software, many programmers will never need to program at the detail described in this chapter. However, for those writing their own kernels or operating systems, and for those wanting to hand optimize compiler code, sections in this chapter will prove useful.

Most of the sections in this chapter contain code fragments illustrating the points under discussion. In some sections, complete subroutines are provided which can be used without modification in real systems; the integer multiplication and division routines are a good example.

To follow the discussion and examples in this chapter, you should be familiar with the contents of Chapter 2, *Programmer's Model*. You should also know how to read SPARC assembly language (see Chapter 7).

5.1 Initialization

Processor reset occurs when the external system asserts the -RESET input. Upon reset, the processor is in supervisor mode. It begins fetching and executing instructions starting at address $0x00000000$ in Supervisor Instruction Space (ASI

0x9). The S bit of the PSR is set to 1; the ET bit is cleared to 0. The *tt* field of the Trap Base Register remains unchanged and identifies the last trap encountered if reset occurs without removing power from the processor. This provides a way to trace the origin of a halt to error mode (on power-up, the *tt* field is undefined). All other fields of the SPARC control and status registers (PSR, WIM, TBR, and Y) are undefined on reset.

The Cache/BIU Control Register and System-Support Register are cleared to 0; that is, the various features controlled by these registers are turned *off* (except for $-CS0$). The contents of the on-chip cache and the various system-configuration registers are undefined (see Chapter 2 for details).

5.1.1 Establishing the Processor State

The first task of initialization code is to establish the processor state, as in the following code fragment:

```
! Reset Initialization
wr  %g0, 0x0fa7,%psr      ! Set psr: mask interrupts, mode=S, Pmode=U,
                          ! traps enabled, CWP=7
wr  %g0, 0x0, %wim        ! Initialize wim to window 0
wr  %g0, 0x0, %tbr        ! Initialize tbr to 0
```

Writes to the PSR, WIM, and TBR registers are *delayed* by three instruction cycles; that is, the value in the register undefined for three instructions following the write. Accessing one of these registers, either explicitly or implicitly, within three instructions after a write can lead to unpredictable results.

5.1.2 Configuring the System

Initialization code must also configure the system by writing appropriate values into the system-configuration registers (Address Range Specifiers and Masks, Wait-State Specifiers, Same-Page Mask, and the Timer Registers). Figure 5-1 shows the memory map of a simple example system.

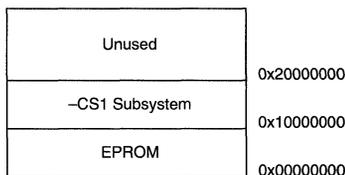


Figure 5-1. Example System Memory Map

The following code sets the various system-configuration registers to values appropriate for the example system.

```

! Address Range Register and Address Mask Register for -CS0 and
! -CS1 are set here. Only the highest nibble of the addresses
! are used for mapping the different -CS signals as shown in Figure 5-1.
! Note: Address range register for -CS0 is preset to 0x04 80 00 00
! ASI=0x9, addr<31:10>=0x0

    sethi %hi(0xfdf<<19), %l0
    xnor %g0, %l0, %l0      ! Set address mask register for -CS0
    or %g0, 0x140, %l1     ! ASI<1>=x, addr<27:0>=0XXXXXXXX
    sta %l0, [%l1] 1      ! SI and SD ASI, addr=0XXXXXXXX
    sethi %hi(0xb1<<19), %l0 ! Set address range register for -CS1:
    or %g0, 0x124, %l1     ! ASI=0xb, addr<31:28>=0x1
    sta %l0, [%l1] 1
    sethi %hi(0xfcfc<<19), %l0
    xnor %g0, %l0, %l0     ! Set address mask register for -CS1
    or %g0, 0x144, %l1     ! ASI<1,0>=xx, addr<27:0>=0XXXXXXXX
    sta %l0, [%l1] 1      ! SI, SD, UI and UD ASI, addr=0x1XXXXXXXX

! Set Wait State Specifier Registers
! Note: count=WS-1, WS+1=cycles, count=cycles-2
! Wait state value is for -CS0 (ROM) and is set to:
! count=6, wait en=1, single cyc=0, override=0
! Wait state value is for -CS1 (subsystem) and is set to:
! count=0, wait en=0, single cyc=0, override=0

    or %g0, 0x160, %l1     ! -CS0 and -CS1 WSS Register
    or %g0, 0x634, %l0
    sll %l0, 6, %l0
    sta %l0, [%l1] 1

.align 4
.word 0xa3802001          ! Set Ancillary Register 17 bit 0
                          ! to enable single vector trapping.
                          ! Machine code is used here for assemblers
                          ! which do not have the WR ASR intruc-
                          ! tion.

    or %g0, 0, %l0        ! Write 0 into Cache/BIU Control Reg
    sta %l0, [%g0] 1      ! disabling all caches

    set 0xffff, %l0       ! Set Timer Pre-Load Register
    or %g0, 0x174, %l1    ! Reload value is set to 0xffff
    sta %l0, [%l1] 1

    set 0x7f800006, %l1    ! Set Same-Page Mask Register
    or %g0, 0x120, %l0    ! Page-size is set to 1K for any ASI
    sta %l0, [%l1] 1

```

```
or    %g0, 0x3c, %l0      ! Set System Support Control Reg:
or    %g0, 0x80, %l1      ! -SAME_PAGE, -CS<5-1>, WS generator and
sta   %l0, [%l1] 1        ! -TIMER_OVF are all enabled
```

5.1.3 Initializing the On-Chip Cache

On reset, both caches are turned off, and all memory requests are sent to the Bus Interface Unit. In order to use the caches, software must initialize the Valid, Least Recently Used and Entry Lock bits by writing 0's to the appropriate alternate address spaces. After initializing the cache, a program can write 1's to the Cache Enable bits of the Cache/BIU control register to turn the caches on. The prefetch and write buffers of the BIU can be turned on in the same operation.

The following code initializes the data and instruction caches, then enables caching and BIU buffering.

```
#define set_size          64
#define ini_tag           0
#define adr1              0
#define adr2              0x80000000
#define CTL_BITS          0x35 /* turn on i-cache, d-cache, prefetch buf., write
                                buf.*/

#define icache_lock_bit  0x1
#define dcache_lock_bit  0x3
#define icache_lock      0x8
#define dcache_lock      0xa
#define icache_enlock    0x1
#define dcache_enlock    0x2
#define lock_reg_adr     0x4
#define lock_save_adr    0x8

.seg "text"
    set    set_size, %l7      /* RAM size */
    set    adr1, %o0          /* start address, set 1 */
    set    adr2, %o2          /* start address, set 2 */
    set    ini_tag, %l0       /* initial tag value */

loopinit:
    sta    %l0, [%o0] 0xc     ! write set 1, itag
    sta    %l0, [%o0] 0xe     ! write set 1, dtag
    sta    %l0, [%o2] 0xc     ! write set 2, itag
    sta    %l0, [%o2] 0xe     ! write set 2, dtag
    add    %o0, 16, %o0       ! inc by 4 words (each tag serves 4 words)
    subcc  %l7, 1, %l7
    bne    loopinit
    add    %o2, 16, %o2       ! delay slot

    set    0, %l1
    set    CTL_BITS, %i7      ! turn on caches.
    sta    %l7, [%l1] 1
    nop
    nop                       ! some nop's for transition
    nop
```

```
nop
nop
```

5.2 Trap Handling

An interrupt or trap (other than reset) causes a vectored transfer of control into a trap table. The first four instructions of each trap handler are in the trap table itself. The Trap Base Address field in the Trap Base Register contains the base address of the table. Associated with each trap type is an 8-bit value, which (left shifted by 4 bits) is used as an offset into the table. From the trap table, control typically passes (via a JMPL or BA instruction) to the appropriate trap handler. A trap table with base address 0x00000000 is shown in the following code fragment.

Note that since $-CS0$ is selected for address range 0x0-0x3fff, the branch after reset at address 0x0 must vector within this address range if the internally generated chip select is being used. There is sufficient space after the trap handler (at label "start" below) yet still within the CS0 default range to write the CS0 mask register if required.

```

0   T_reset:                mov   0xe0, %psr
4                       mov   %0, %tbr
/*
/* 0 -> TBR assumes boot is from fast memory, and that only the
/* first 4 instructions of the response to reset are there. Single
/* Vector Trapping is to remain disabled.
*/
8                       ba    start
c                       mov   %0, %wim

10  T_instr_access_exception: rd   %tbr, %13
14                                rd   %psr, %10
18                                ba   iae_handler
1c                                nop
20  T_unimplemented_instruction: rd  %tbr, %13
24                                rd   %psr, %10
28                                ba   illegal
2c                                nop
30  T_privileged_instruction:  rd   %tbr, %13
34                                rd   %psr, %10
38                                ba   privileged
3c                                nop
40  T_fp_disabled:          rd   %tbr, %13
44                                rd   %psr, %10
48                                ba   fp_disabled
4c                                nop
50  T_window_overflow:      rd   %tbr, %13
54                                rd   %psr, %10

```

```

58                                ba    win_overflow
5c                                nop
60  T_window_underflow:          rd    %tbr, %13
64                                rd    %psr, %10
68                                ba    win_underflow
6c                                nop
70  T_mem_addr_not_aligned:      rd    %tbr, %13
74                                rd    %psr, %10
78                                ba    misaligned_addr
7c                                nop
80  T_fp_exception:             rd    %tbr, %13
84                                rd    %psr, %10
88                                ba    unimplemented_trap
8c                                nop
90  T_data_access_exception:    rd    %tbr, %13
94                                rd    %psr, %10
98                                ba    dae_handler
9c                                nop
a0  T_tag_overflow:             rd    %tbr, %13
a4                                rd    %psr, %10
a8                                ba    tag_overflow
ac                                nop

b0                                rd    %tbr, %13
b4                                rd    %psr, %10
b8                                ba    unimplemented_trap
bc                                nop
c0                                rd    %tbr, %13
c4                                rd    %psr, %10
c8                                ba    unimplemented_trap
cc                                nop

...

100                               rd    %tbr, %13
104                               rd    %psr, %10
108                               ba    unimplemented_trap
10c                               nop
110  T_int_1:                   rd    %tbr, %13
114                               rd    %psr, %10
118                               ba    int_handler
11c                               nop
120  T_int_2:                   rd    %tbr, %13
124                               rd    %psr, %10
128                               ba    int_handler
12c                               nop

...

1f0  T_int_15:                  rd    %tbr, %13
1f4                                rd    %psr, %10

```

```

1f8          ba    int_handler
1fc          nop
200 T_rferr: rd    %tbr, %13
204          rd    %psr, %10
208          ba    unimplemented_trap
20c          nop
210 T_iaerr: rd    %tbr, %13
214          rd    %psr, %10
218          ba    iae_handler
21c          nop
220          rd    %tbr, %13
224          rd    %psr, %10
228          ba    unimplemented_trap
22c          nop
230          rd    %tbr, %13
234          rd    %psr, %10
238          ba    unimplemented_trap
23c          nop
240 T_cp_disabled: rd    %tbr, %13
244          rd    %psr, %10
248          ba    cp_disabled
24c          nop
250          rd    %tbr, %13
254          rd    %psr, %10
258          ba    unimplemented_trap
25c          nop
260          rd    %tbr, %13
264          rd    %psr, %10
268          ba    unimplemented_trap
26c          nop
270          rd    %tbr, %13
274          rd    %psr, %10
278          ba    unimplemented_trap
27c          nop
280 T_cp_exception: rd    %tbr, %13
284          rd    %psr, %10
288          ba    unimplemented_trap
28c          nop
290 T_daerr:  rd    %tbr, %13
294          rd    %psr, %10
298          ba    dae_handler
29c          nop
2a0          rd    %tbr, %13
2a4          rd    %psr, %10
2a8          ba    unimplemented_trap
2ac          nop
2b0          rd    %tbr, %13
2b4          rd    %psr, %10
2b8          ba    unimplemented_trap
2bc          nop

```

```
...  
  
800 software_traps:      rd    %tbr, %13  
804                    rd    %psr, %10  
808                    ba    trap_instr  
80c                    nop  
810                    rd    %tbr, %13  
814                    rd    %psr, %10  
818                    ba    trap_instr  
81c                    nop  
  
...  
  
fe0                    rd    %tbr, %13  
fe4                    rd    %psr, %10  
fe8                    ba    trap_instr  
fec                    nop  
ff0                    rd    %tbr, %13  
ff4                    rd    %psr, %10  
ff8                    ba    emu_exception  
ffc                    nop  
  
...  
1000 start:
```

When a trap is taken, the processor writes the trap type number into the *tt* field of the Trap Base Register, and disables traps by clearing the ET bit of the Processor Status Register. The processor enters supervisor mode ($S=1$), saving the old state of the S bit in the PS field of the PSR. The Current Window Pointer is automatically decremented.

Each of the illustrated trap handlers (except for reset) begins by saving the values of the TBR and PSR, and then jumps, by means of an unconditional branch, to the next instruction in the service routine.

Each trap handler must then:

1. With ET cleared ($ET=0$) by the processor, ensure that a window is available, in case another trap occurs. (When it takes a trap, the processor automatically saves the window of the interrupted routine by decrementing the Current Window Pointer.)
2. Re-enable traps by setting the ET bit of the PSR to 1.
3. Handle the exceptional condition that caused the trap.
4. Disable traps by clearing the ET bit of the PSR to 0.
5. Ensure that a window is available, so that the RETT (return from trap) instruction can restore the window of the interrupted routine by incrementing the CWP.

6. Execute a JMPL/RETT instruction pair. The address for the return is found in r[17] (When it takes a trap, the processor loads r[17] with the value in the PC). The RETT instruction automatically re-enables traps (ET=1).

To re-execute the trapped instruction when returning from a trap handler use the sequence:

```
JMPL %r17, %g0 ! old PC
rett %r18      ! old nPC
```

To return to the instruction after the trapped instruction (e.g., when emulating an instruction) use the sequence:

```
jmp1 %18, %g0 ! old nPC
rett %18 + 4  ! old nPC + 4
```

Two example trap handlers are shown below.

```
! FUNCTION
!   _win_ovf
!
! DESCRIPTION
!   This routine is the trap handler for register window overflow trap.
!   Priority: 0x06
!   Upon entry, the cwp points to the trap window, which is 1 less than
!   the register window that must be saved to the stack. the stack is
!   organized with %i6 = %o6 - (0x40 + local stack used). the ins and
!   locals are saved, and the wim is adjusted for the new window.
!
! INPUTS
!   - None.
!
! INTERNAL DESCRIPTION
!   - Move the invalid window to the next window by rotating the %wim
!     register left by one slot.
!   - Get into the previously invalid window, the one that caused the,
!     trap, and save all of the registers in it.
!   - Get back into the previously valid window and let the trapped
!     routine execute the save again.
!
! RETURNS
!   - %o0 = 1 so execution starts at the trapped instruction.
!
win_overflow:
```

```
rd    %wim, %i4           !read WIM for window handler
wr    %g0, 0, %wim       !clear WIM for now

save                                !decrement into window to be saved

std   %i0, [%sp + 0x0 * 4] !save all local registers
std   %i2, [%sp + 0x2 * 4]
```

```

std    %l4, [%sp + 0x4 * 4]
std    %l6, [%sp + 0x6 * 4]

std    %i0, [%sp + 0x8 * 4] !save all input registers
std    %i2, [%sp + 0xa * 4]
std    %i4, [%sp + 0xc * 4]
std    %i6, [%sp + 0xe * 4]

restore                                !go back to trap window

srl    %l4, 1, %l5                    !rotate original WIM right to obtain the
sll    %l4, 8-1, %l4                  !next window (SPARClite has
or     %l4, %l5, %l4                  !8 windows)
wr     %g0, %l4, %wim                 !install the new WIM

wr     %l0, 0, %psr                   !restore the saved PSR
nop                                         !required nops
nop
nop
jmp    %l1                             !return from the trap
rett   %l2                             !to re-execute SAVE

!
! FUNCTION
!   _win_unf
!
! DESCRIPTION
!   This routine is the trap handler for register window underflow trap.
!   Priority: 0x07
!   Upon entry, the cwp points to the trap window, which is 1 more than
!   the register window that must be restored from the stack. The stack
!   is organized with %i6 = %o6 - (0x40 + local stack used). The ins
!   and locals are restored, and the wim is adjusted for the new window.
!
! INPUTS
!   - None.
!
! INTERNAL DESCRIPTION
!
! RETURNS
!   - %o0 = 1 so execution starts at the trapped instruction.
!
win_underflow:
or     %l0, 0x20, %l0                  ! enable traps
wr     %l0, %psr
mov    %wim, %l4                       ! Get wim.
sll    %l4, 1, %l5                     ! Next WIM = rol(WIM, 1, NWINDOW).
srl    %l4, NWINDOWS-1, %l6
or     %l6, %l5, %l6
mov    %l6, %wim                       ! Install it.

```

```

nop                                ! must delay three instructions
nop                                ! before using these registers, so
nop                                ! put nops in just to be safe
restore                             ! Back to user window.
restore                             ! Get into window to be restored.
ldd    [%sp + 0x0 * 4], %i0        ! Restore all registers
ldd    [%sp + 0x2 * 4], %i2
ldd    [%sp + 0x4 * 4], %i4
ldd    [%sp + 0x6 * 4], %i6
ldd    [%sp + 0x8 * 4], %i0
ldd    [%sp + 0xa * 4], %i2
ldd    [%sp + 0xc * 4], %i4
ldd    [%sp + 0xe * 4], %i6
save
save                                ! Get back to original window.
_rerun_trap_instr:
andn   %i0, 0x20, %i0            ! Disable traps.
wr     %i0, %psr
or     %g0, 0x1, %g1            ! Set Restore Lock bit,
or     %g0, 0x10, %i0          ! in case an autolock sequence
sta    %g1, [%i0] 1             ! is in effect.
jmpl   %i1, %g0                ! Return to instruction at PC.
rett   %i2

```

5.3 Register and Stack Management

This section describes the standard conventions for using the register file. Most SPARC compilers comply with this convention as this is the standard adopted on SPARC workstations. (Compilers are available that optimize code differently for embedded applications if required.)

This section describes standard conventions for using the register file.

5.3.1 Registers

Register usage is typically a critical resource allocation issue for compilers. The SPARClike architecture provides windowed integer registers (*in*, *out*, *local*), and

global integer registers. Figure 5-2 summarizes the SPARC register set, as seen by a user-mode procedure.

in	%i7	(%r31)	return address [†]
	%i6	(%r30)	frame pointer [†]
	%i5	(%r29)	incoming param 6 [†]
	%i4	(%r28)	incoming param 5 [†]
	%i3	(%r27)	incoming param 4 [†]
	%i2	(%r26)	incoming param 3 [†]
	%i1	(%r25)	incoming param 2 [†]
	%i0	(%r24)	incoming param 1 / return value to caller [†]
local	%l7	(%r23)	local 7 [†]
	%l6	(%r22)	local 6 [†]
	%l5	(%r21)	local 5 [†]
	%l4	(%r20)	local 4 [†]
	%l3	(%r19)	local 3 [†]
	%l2	(%r18)	local 2 [†]
	%l1	(%r17)	local 1 [†]
	%l0	(%r16)	local 0 [†]
out	%o7	(%r15)	temporary value / address of CALL instruction [‡]
	%sp, %o6	(%r14)	stack pointer [†]
	%o5	(%r13)	outgoing param 6 [‡]
	%o4	(%r12)	outgoing param 5 [‡]
	%o3	(%r11)	outgoing param 4 [‡]
	%o2	(%r10)	outgoing param 3 [‡]
	%o1	(%r9)	outgoing param 2 [‡]
	%o0	(%r8)	outgoing param 1 / return value from callee [‡]
global	%g7	(%r7)	global 7 (SPARC ABI: use reserved)
	%g6	(%r6)	global 6 (SPARC ABI: use reserved)
	%g5	(%r5)	global 5 (SPARC ABI: use reserved)
	%g4	(%r4)	global 4 (SPARC ABI: global register variable)
	%g3	(%r3)	global 3 (SPARC ABI: global register variable)
	%g2	(%r2)	global 2 (SPARC ABI: global register variable)
	%g1	(%r1)	temporary value [‡]
	%g0	(%r0)	0
state	%y		Y register (used in multiplication/division) [‡]
	(icc field of %psr)		Integer condition codes [‡]

†. assumed by caller to be preserved across a procedure call

‡. assumed by caller to be destroyed (volatile) across a procedure call.

Figure 5-2. SPARC Register Set, as Seen by a User-Mode Procedure

In and Out Registers

The *in* and *out* registers are used primarily for passing parameters to subroutines and receiving results from them, and for keeping track of the memory stack.

Certain routines can also use *out* registers 0 through 5 as fast temporary storage; these include *leaf routines*—which contain no procedure calls—and routines which pass parameters using only shared memory or global registers. In general, when a procedure is called, the caller's *outs* become the callee's *ins*.

One of a procedure's *out* registers (%o6) is used as its stack pointer, %sp. It points to an area in which the system can store %r16 ... %r31 (%l0 ... %l7) when the register file overflows (window_overflow trap); it is used to address most values located on the stack. See Figure 5-3. A trap can occur at any time, which may precipitate a subsequent window_overflow trap, during which the contents of the user's register window at the time of the original trap are spilled to the memory to which its %sp points.

A procedure may store temporary values in its out registers, with the exception of %sp, with the understanding that those values are volatile across procedure calls. %sp cannot be used for temporary values for the reasons described in the *Register Windows and %sp* section below.

Up to six parameters can be passed by placing them in *out* registers %o0...%o5; additional parameters are passed in the memory stack. The stack pointer is implicitly passed in %o6, and a CALL instruction places its own address in %o7.

When an argument is a data aggregate being passed by value, the caller first makes a temporary copy of the data aggregate in its stack frame, then passes a pointer to the copy in the argument *out* register (or on the stack, if it is the 7th or later argument).

After a callee is entered and its SAVE instruction has been executed, the caller's *out* registers are accessible as the callee's *in* registers.

The caller's stack pointer %sp (%o6) automatically becomes the current procedure's frame pointer %fp (%i6) when the SAVE instruction is executed.

The callee finds its first six parameters in %i0 ... %i5, and the remainder (if any) on the stack.

For each passed-by-value data aggregate, the callee finds a pointer to a copy of the aggregate in its argument list. The compiler must arrange for an extra dereferencing operation each time such an argument is referenced in the callee. The additional code in the callee program uses the pointer to access aggregate values on the stack.

If the callee is passed fewer than six parameters, it may store temporary values in the unused *in* registers.

If a register parameter (in %i0 ... %i5) has its address taken in the called procedure, the callee stores that parameter's value on the memory stack. The parameter is then accessed in that memory location for the lifetime of the pointer(s) which

contains its address (or for the lifetime of the procedure, if the compiler doesn't know the pointer's lifetime).

The six words available on the stack for saving the first six parameters are deliberately contiguous in memory with those in which additional parameters may be passed. This supports constructs such as C's *varargs*, for which the callee copies to the stack the register parameters which must be addressable.

A function returns a scalar integer value by writing it into its ins (which are the caller's *outs*), starting with %i0. Aggregate values are returned using the mechanism described in the *Functions Returning Aggregate Values* section.

A procedure's return address, normally the address of the instruction just after the CALL's delay-slot instruction, is simply calculated as %i7 + 8.

Local Registers

The *locals* are used for *automatic* variables—those whose lifetimes are no longer than the lifetimes of their containing procedures—and for most temporary values. For access efficiency, a compiler may also copy parameters (i.e., those past the sixth) from the memory stack into the *locals* and use them from there. Procedures only calling several leaf routines may be more efficient if some of the procedure's automatic variables are referenced by their address rather than have the values passed for each leaf routine call and return. If an automatic variable's address is taken, the variable's value must be stored in the memory stack, and be accessed there for the lifetime of the pointer(s) which contains its address (or for the lifetime of the procedure, if the compiler doesn't know the pointer's lifetime).

If a routine creates variables that can be used by other called routines, these variables should either be stored in the memory stack and referenced by pointers, or stored in the global registers, unless the register window does not change when the other routines are called.

Register Windows and %sp

Some caveats about the use of %sp and the SAVE and RESTORE instructions are appropriate. It is essential that:

- %sp *always* contains the correct value, so that when (and if) a register window overflow or underflow trap occurs, the register window can be correctly stored to or reloaded from memory.
- User (non-supervisor) code use SAVE and RESTORE instructions carefully. In particular, "walking" the call chain through the register windows using RESTOREs, expecting to be able to return to where one started using SAVEs does not work as one might suppose. This fails because the "next" register window (in the "SAVE direction") is reserved for use by trap handlers. Since

non-supervisor code cannot disable traps, a trap could write over the contents of a user register window which has “temporarily” been RESTORE’d.

For example, if a routine at the fourth calling level returns to its caller at third level and restores the third-level window, an intervening trap at third level can change registers in the fourth-level window. A subsequent call and SAVE to a routine at fourth level will not find the register contents the same as they were on exit from the last fourth-level routine.

The safe method is to flush the register windows out to user memory (the stack) in supervisor state using a software trap designed for that purpose. Then, user code can safely “walk” the call chain through user memory, instead of through the register windows.

The rule-of-thumb which will avoid such problems is to consider all memory below %sp on the user’s stack, and the contents of all register windows “below” the current one to be volatile. Below means decreasing memory address and window pointer, corresponding to call space of subsequent routines by the current routine. In embedded control applications complex enough to require partitioning the process into re-usable tasks driven by a master sequencer, this view can be critical to ensure correct functioning in all cases.

Global Registers

Unlike the *ins*, *locals*, and *outs*, the *globals* are not part of any register window. The *globals* are a set of eight registers with global scope, like the register sets of more traditional processor architectures. The *globals* (except %g0) are conventionally assumed to be volatile across procedure calls. However, if they are used on a per-procedure basis and expected to be non-volatile across procedure calls, either the caller or the callee has to take responsibility for saving and restoring their contents.

Global register %g0 has a “hardwired” value of zero. It always reads as zero, and writes to it have no effect.

The *global* registers other than %g0 can be used for temporaries, global variables, or global pointers—either user variables, or values maintained as part of the program’s execution environment. For example, one could use *globals* in the execution environment by establishing a convention that global scalars are addressed via offsets from a global base register. In the general case, memory accessed at an arbitrary address requires two instructions, e.g.:

```
sethi %hi(address), reg
ld    [reg+%lo(address)], reg
```

Use of a global base register for frequently accessed global values would provide faster (single-instruction) access to 2^{13} bytes of those values, e.g.:

```
ld    [%gn+offset], reg
```

Global register *n* would hold the address of the center of a block of global values. The offset, varying from -4096 to 4095 bytes, would point to a particular value.

The current convention is that the global registers (except %g0) are assumed to be volatile across procedure calls. The convention used by the SPARC Application Binary Interface (ABI) is that %g1 is assumed to be volatile across procedure calls, %g2 ... %g4 are reserved for use by the application program (for example, as global register variables), and %g5 ... %g7 are assumed to be nonvolatile and reserved for (as-yet-undefined) use by the execution environment.

5.3.2 Memory Stack

Space on the memory stack, called a *stack frame*, is normally allocated for each procedure. Under certain conditions, optimization may enable a leaf procedure to use its caller's stack frame instead of one of its own. In that case, the leaf procedure allocates no space of its own for a stack frame. The following description of the memory stack applies to all procedures, except leaf procedures which have been optimized as shown in 5.3.4.

The following are *always* allocated at compile time in every procedure's stack frame:

- 16 words, always starting at %sp, for saving the procedure's in and local registers, should a register window overflow occur.

The following are allocated at compile time in the stack frames of non-leaf procedures:

- One word, for passing a "hidden" (implicit) parameter. This is used when the caller is expecting the callee to return a data aggregate by value; the hidden word contains the address of stack space allocated (if any) by the caller for that purpose. See the section titled *Functions Returning Aggregate Values*.
- Six words, into which the callee may store parameters that must be addressable.

Space is allocated as needed in the stack frame for the following at compile time:

- Outgoing parameters beyond the sixth.
- All automatic arrays, automatic data aggregates, automatic scalars which must be addressable, and automatic scalars for which there is no room in registers.
- Compiler-generated temporary values (typically when there are too many for the compiler to keep them all in registers).

Space can be allocated dynamically (at runtime) in the stack frame for the following:

- Memory allocated using the `alloca()` function of the C library

Addressable automatic variables on the stack are addressed with negative offsets relative to `%fp`; dynamically allocated space is addressed with positive offsets from the pointer returned by `alloca()`; everything else in the stack frame is addressed with positive offsets relative to `%sp`.

The stack pointer `%sp` must always be doubleword-aligned. This allows window overflow and underflow trap handlers to use the more efficient `STD` and `LDD` instructions to store and reload register windows.

Figure 5-3 illustrates the stack frame of an active non-leaf procedure.

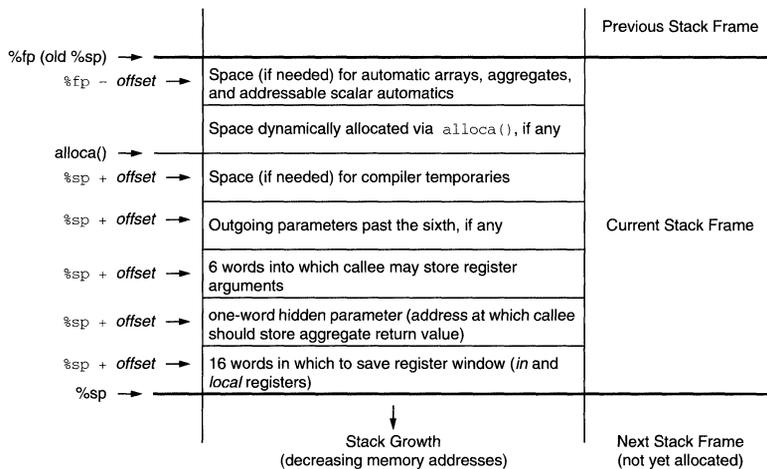


Figure 5-3. User Stack Frame

5.3.3 Functions Returning Aggregate Values

Some programming languages, including C, dialects of Pascal, and Modula-2, allow the user to define functions that return aggregate values. Examples include a C `struct` or `union`, or a Pascal `record`. Since such a value may not fit into the registers, another value-returning protocol must be defined to return the result in memory.

Re-entrancy and efficiency considerations require that the memory used to hold such a return value be allocated by the function's caller. The address of this memory area is passed as the one-word hidden parameter mentioned in section 5.3.2 "Memory Stack", above. Where it is known that re-entrancy is not required, global

or shared memory allocated by the master sequencer can be an effective alternative, especially if the amount of memory required is small enough to be held in locked data cache.

Because of the lack of type safety in the C language, a function should not assume that its caller is expecting an aggregate return value and has provided a valid memory address. Thus, some additional handshaking is required.

When a procedure expecting an aggregate return value from a called function is compiled, an UNIMP instruction is placed after the delay-slot instruction following the CALL to the function in question. The immediate field in this UNIMP instruction contains the low-order twelve bits of the size (in bytes) of the area allocated by the caller for the aggregate value expected to be returned.

When the aggregate-returning function is about to store its value in the memory allocated by its caller, it first tests for the presence of this UNIMP instruction in its caller's instruction stream. If it is found, the callee assumes the hidden parameter to be valid, stores its return value at the given address, and returns control to the instruction following the caller's UNIMP instruction. If the UNIMP instruction is not found, the hidden parameter is assumed not to be valid and no value is returned.

On the other hand, if a scalar-returning function is called when an aggregate return value is expected (which is clearly a software error), the function returns as usual, executing the UNIMP instruction, which causes an unimplemented-instruction trap.

5.3.4 Leaf Procedure Optimization

A *leaf procedure* is one that is a "leaf" in the program's call graph; that is, one that does not call (e.g. via CALL or JMPL) any other procedures.

Each procedure, including leaf procedures, normally uses a SAVE instruction to allocate a stack frame and obtain a register window for itself, and a corresponding RESTORE instruction to de-allocate it. The time costs associated with this are:

- Possible generation of register-window overflow/underflow traps at runtime. This only happens occasionally, but when either underflow or overflow does occur, it costs dozens of machine cycles to process.
- The two cycles expended by the SAVE and RESTORE instructions themselves

There are also space costs associated with this convention, the cumulative cache effects of which may not be negligible. The space costs include:

- The space occupied on the stack by the procedure's stack frame
- The two words occupied by the SAVE and RESTORE instructions

Of the above costs, the trap-processing cycles are typically the most significant.

Some leaf procedures can be made to operate without their own register window or stack frame, using their caller's instead. This can be done when the candidate leaf procedure meets all of the following conditions:

- Contains no references to `%sp`, except in its SAVE instruction
- Contains no references to `%fp`
- Refers to (or can be made to refer to) no more than 8 of the 32 integer registers, inclusive of `%o7` (the "return address").

Such procedures can be converted into routines which share the caller's stack frame and register window—an optimization that saves both time and space. When optimized, such a procedure is known as an optimized leaf procedure. It may only safely use registers that its caller already assumes to be volatile across a procedure call, namely, `%o0 ... %o5`, `%o7`, and `%gl`.

The optimization can be performed at the assembly-language level using the following steps:

- Change all references to registers in the procedure to registers that the caller assumes volatile across the call:
 - Leave references to `%o7` unchanged.
 - Leave any references to `%g0 ... %g7` unchanged.
 - Change `%i0 ... %i5` to `%o0 ... %o5`, respectively. If an in register is changed to an out register that was already referenced in the original unoptimized version of the procedure, all original references to that out register must be changed to refer to an unused out or global register.
 - Change references to each local register into references to any register among `%o0 ... %o5` or `%gl` that remains unused.
- Delete the SAVE instruction. If it was in a delay slot, replace it with a NOP instruction. If its destination register was not `%g0` or `%sp`, convert the SAVE into the corresponding ADD instruction instead of deleting it.
- If the RESTORE's implicit addition operation is used for a productive purpose (such as setting up the procedure's return value), convert the RESTORE to the corresponding ADD instruction. Otherwise, the RESTORE is only used for stack and register-window de-allocation; replace it with a NOP instruction (it is probably in the delay slot of the RET, and so cannot be deleted).
- Change the RET (return) synthetic instruction to RETL (return-from-leaf-procedure synthetic instruction).
- Perform any optimizations newly made possible, such as combining instructions, or filling the delay slot of the RETL with a productive instruction.

After the above changes, there should be no SAVE or RESTORE instructions, and no references to in or local registers in the procedure body. All original references to ins are now to outs. All other register references are to either `%gl`, or other outs.

Costs of optimizing leaf procedures in this way include:

- Additional intelligence in the peephole optimizer to recognize and optimize candidate leaf procedures.
- Additional intelligence in debuggers to properly report the call chain and the stack traceback for optimized leaf procedures.

The following code fragment shows a simple procedure call with a value returned, and the procedure itself:

```
! CALLER:
!   int i;                               /* compiler assigns "i" to register %i7 */
!   i = sum3 (1, 2, 3);
!   ...
!   mov    1,%o0                          ! first arg to sum3 is 1
!   mov    2, %o1                          ! second arg to sum3 is 2
!   call   sum3                            ! the call to sum3
!   mov    3, %o2                          ! last parameter to sum3 in delay slot
!   mov    %o0, %i7                       ! copy return value to %i7 (variable "i")
!   ...

#define SA      (x)(((x) +7) & (~0x07)) /* rounds "x" up to doubleword boundary */
#define MINFRAME ((16+1+6)*4)         /* minimum size frame */

! CALLEE:
!   int    sum3 (a, b, c)
!   int    a, b, c;                       /* args received in %i0, %i1, and %i2 */
!   {
!   return a+b+c;
!   }

sum3:
!   save   %sp, -SA(MINFRAME), %sp        !set up new %sp; alloc min. stack frame
!   add    %i0, %i1, %i7                  ! compute sum in local %i7
!   add    %i7, %i2, %i7                  ! (or %i0 could have been used directly)
!   ret                                         ! return from sum3, and...
!   restore %i7, 0, %o0                    ! move result into output reg & restore
```

Since "sum3" does not call any other procedures (i.e., it is a "leaf" procedure), it can be optimized to become:

```
sum3:
!   add    %o0, %o1, %o0                  !
!   retl                                       ! (must use RETL, not RET,
!   add    %o0, %o2, %o0                  ! to return from leaf procedure)
```

If a leaf routine is being created at the assembly level for use in an environment such as embedded control where all the caller routines are known, then a different approach can be taken.

Form a register map which identifies all of the in and local registers which contain information to be used by the leaf routine. Additionally, to accommodate the most restrictive of caller routines, identify those in and local registers which must be preserved for the caller.

Initially attempt to write the leaf routine so that it changes only out and global registers, but uses information in the in and local registers. If the code requires storing temporary values in memory and retrieving them later in the routine, or regenerating a value in a register later in the routine because the register was overwritten to hold some other value, then examine the in and local registers to see if any of them can be changed by the leaf routine.

If so, modify the routine appropriately. If not, or if after modification there is still temporary memory use or register value regeneration, try to relax the restrictions of caller routines by changing code to regenerate some of the variables saved in registers.

Usually leaf routines are associated with inner loops and are executed much more frequently than the routines that call them. Total program performance will be improved with the most efficient inner loops and leaf routines, even at the expense of less efficient outer-loop and set-up routines.

The following short function code shows an example of a leaf routine written directly at the assembly level and satisfying the requirements for safe calling by other routines:

```

/*RGB_I
*
*Convert red, green, blue pixel planes to intensity pixel plane:
*
*   Y(i,j)= [a*A(i,j)+ b*B(i,j)+ c*C(i,j)]/256
*
*   Since there is no distinction between the i and j indexes as
*   used by this process, the arrays can be accessed linearly with
*   a single index that runs through the total 512 by 512 pixel
*   space. i= 511 -> 0, j= 511 -> 0. Each pixel is one byte.

*Inputs:  base address Y
*         base address A
*         base address B
*         base address C
*         pointer to Scalar Constant Array Base for a,b,c and other
*         constants.
*Outputs: Y(i,j)

*Time:   3932169 + 458753W cycles,
*        where W is number of wait states for DRAM access of data.

```

***REGISTER MAP:**

*i0	[Y(0,0)]	10	o0	aA+bB+cC, Y(i,j)	g0	0
*i1	[A(0,0)]	11	o1	A(i,j)	g1	a
*i2	[B(0,0)]	12	o2	B(i,j)	g2	b
*i3	[C(0,0)]	13	o3	C(i,j)	g3	c
*i4		14	o4	bB, cC	g4	
*i5		15	o5	512j+i(2 ¹⁸ -1 ->0)	g5	[Y(0,0)]+1
*i6	FP	16	o6	SP	g6	
*i7	general return	17	o7	leaf return	g7	SCAB

*The following instructions take one cycle unless otherwise noted.

*/

```
rgb_i: sethi 256,%o5           !preset index to last pixel for fetch.
      sub   %o5,1,%o5         !start at end & work toward beginning.
      add   %i0,1,%g5         !offset store base to compensate for
                              !fetch index being ahead one pass
                              !of store index
      ldub [%g7+cnsta],%g1     !get weighting coefficients
      ldub [%g7+cnstb],%g2     !1+W cycles for 1st byte - cache miss.
      ldub [%g7+cnstc],%g3     !1 cycle each for rest - cache hit.
/*inner loop begin*/
t1:   ldub [%i1+%o5],%o1       !fetch A. 1+W cycles for 1st byte.
                              !1 cycle for remaining 3 bytes in word.
      umul %o1,%g1,%o0         !2 cycles for byte multiplier
      ldub [%i2+%o5],%o2       !fetch B. 1+W/4 cycles.
      umul %o2,%g2,%o4         !2 cycles.
      add  %o0,%o4,%o0         !update accumulator
      ldub [%i3+%o5],%o3       !fetch C. 1+W/4 cycles.
      umul %o3,%g3,%o4         !2 cycles.
      add  %o0,%o4,%o0         !update accumulator
      sra  %o0,8,%o0           !scale sum of products to form Y
      subcc %o5,1,%o5         !decrement & test index
      bg   t1                  !loop if index >0
      stb  %o0,[%g5+%o5]       !store Y using offset base since
                              !index has decremented.
                              !1+W cycles - always cache miss.
/*inner loop end*/
      retl                    !2 cycles
      nop                     !exit
```

5.3.5 Register Allocation Within a Window

The usual SPARC software convention is to allocate eight registers (%10-%17) for local values. A compiler could allocate more registers for local values at the expense of having fewer *outs/ins* available for argument passing.

For example, if instead of assuming that the boundary between local values and input arguments is between r[23] and r[24] (%17 and %i0), software could by con-

vention assume that the boundary is between r[25] and r[26] (%i1 and %i2). As illustrated in Table 5-1, this would provide 10 registers for local values and 6 “in”/“out” registers.

Table 5-1: Alternative Register Allocation

	Standard Register Model	“10-Local” Register Model	Arbitrary Register Model
registers for local values	8	10	n
“in”/“out” registers:			
reserved for %sp/%fp	1	1	1
reserved for return address	1	1	1
available for arg passing	6	4	14-n
total “ins”/“outs”	8	6	16-n

5.3.6 Other Register and Window Usage Models

In general-purpose computers, procedure calls are assumed to be frequent relative to both context switches and User-Supervisor state transitions. A primary goal in these applications is to minimize total overhead, which includes time spent in both context switches and procedure calls. As more register windows are shared among competing processes, total procedure call time decreases (due to execution of fewer window overflow and underflow traps), while total context-switch time may increase (the average number of register windows saved during a context switch increases). The task is to strike a balance to minimize the sum of these two factors.

In embedded and/or real-time systems, the following factors are often more important than total overhead:

- Minimal *average* context-switch time
- A *constant* (or small worst-case deterministic) context-switch time
- A *constant* (or small worst-case deterministic) procedure-call time

In these cases, it can be worthwhile to use a different scheme for managing the SPARC register windows than the standard one described so far. This section provides a few examples of modifications that can be made to the standard conventions. You can then design a register-usage scheme appropriate to the specific needs of your application.

1. Divide the register file into “supervisor mode” register windows and “user mode” register windows. In cases where user/supervisor transitions are frequent, this will reduce register-window overflow and underflow overhead.

To be effective in a workstation environment, where the coding style is characterized by deep nesting of procedure calls, such a scheme would require a

SPARC implementation with at least 14 windows in hardware (a minimum of 7 for user code plus 7 for supervisor code). In embedded control, however, the nesting of procedure calls is typically shallow, and windows will be used more sparingly.

2. Use multiple 1's in the Window Invalid Mask Register (WIM) to partition the register file into groups of at least two registers each. Assign each group of registers to an executing task. This technique can be useful in real-time processing, where extremely fast context switches are desirable. A context switch would consist of loading a new stack pointer, resetting the CWP to the new task's block of register windows, and saving and restoring whatever subset of the global registers is assumed to be nonvolatile. In particular, note that no window registers would need to be loaded or stored during a context switch.

This technique assumes that only a few tasks are present, and, in the simplest case, that all tasks share a single address space. The number of hardware register windows required is a function of the number of windows reserved for the supervisor, the number of windows reserved for each task, and the number of tasks. Register windows could be allocated to tasks unequally, if appropriate.

3. Avoid the normal register-window mechanism, by not using SAVE and RESTORE instructions. Software would effectively see 32 general-purpose registers instead of SPARC's usual windowed register file. In this mode, SPARC would operate like processors with a more traditional flat register architecture. Procedure call times would be more deterministic (since there would be no window overflow or underflow traps), but for most types of software, average procedure call time would significantly increase, due to increased memory traffic for parameter passing and saving and restoring local variables.

A number of existing SPARC compilers produce code using this register organization.

It would be awkward, at best, to attempt to mix (link) code using the SAVE/RESTORE convention with code not using it in the same process. If both conventions *were* used in the same system, two versions of each library would be required.

It would be possible to run user code with one register-usage convention and supervisor code with another. With sufficient intelligence in the supervisor, user processes with different register conventions could be run simultaneously.

5.4 Cache Management

Effective cache usage is based on the following principles:

- *Compactness of Code*—Critical loops should fit entirely in the cache. They can then be locked into the cache to prevent their being displaced when other, less-

often-used routines are called. In some cases, it may be advisable to disable compiler in-lining optimizations in order to keep your code compact.

- *Program Profiling*—Knowing where your program spends its time will help you decide what instructions and data to lock into cache.
- *Data and Instruction Locality*—If possible, a large program or data set should be partitioned in such a way that one portion at a time can be locked into cache and used for a while before another portion needs to be loaded. For example, there are numerical routines which perform as many of their required computations as possible on one block of data before proceeding to the next block.

5.5 Division Routines Using the DIVScc Instruction

This section shows how integer division routines can be created using the DIVScc instruction. Signed and unsigned divisions are included for both word and doubleword dividends. The divisor is always a single word. These routines can serve as models for your own use of DIVScc, or they can be incorporated into your programs and used without modification. These sample routines do not set the integer condition codes in exactly the same way as the SPARC Version 8 integer division instructions.

5.5.1 Simple Divide Step Examples

In each of the following examples, a cycle by cycle view of divide step with reduced word size (3 bits) is given

```
! Register Use:
! out0 most significant half Dividend/ Remainder
! out1 least significant half Dividend/ Quotient
! out2 Divisor
! Note: TS, True Sign = N xor V from condition codes
! Note: adjustment of negative quotient is also
!       conditional on remainder. Details omitted
!       here. See signed division example code.
```

Examples of SIGNED division

```
!      7/2 = +3 & +1 rmdr; 010-> o2, 111-> o1, 000-> o0
!      !Y   o1   TS  ALUin   ALUout
mov   %o0,%y      !                               msh dividend -> Y reg
!      !000 111
tst   %o0         !                               initialize cc with sign dividend
!      !000 1|11  0
divscc %o1,%o2,%o1 !                               divide step 1
!      !111 1|10  1
divscc %o1,%o2,%o1 !                               divide step 2
!      !111+0010 0001
```

```

!001 1101 0
divscc %o1,%o2,%o1 ! 0011-0010 0001 divide step 3
!001 011 0
tst %o0 ! dividend & quotient sign?
!001 011 0
bl,a 1f
!001 011
add %o1,1,%o1 ! adjust quotient if negative from
!001 011 1's to 2's complement form
1:mov %y,%o0 !001 -> o0 retrieve remainder

! -11/3 = -3 & -2 rmdr; 011-> o2, 101-> o1, 110-> o0
!Y o1 TS ALUin ALUout
mov %o0,%y ! msh dividend -> Y reg
!110 101
tst %o0 ! initialize cc with sign dividend
!110 1101 1
divscc %o1,%o2,%o1 ! 1101+0011 0000 divide step 1
!000 0111 0
divscc %o1,%o2,%o1 ! 0000-0011 1101 divide step 2
!101 1110 1
divscc %o1,%o2,%o1 ! 1011+0011 1110 divide step 3
!110 100 1
tst %o0 ! dividend & quotient sign?
!110 100 1
bl,a 1f
!110 100
add %o1,1,%o1 ! 100+001 101 adjust quotient if negative from
!110 101 1's to 2's complement form
1:mov %y,%o0 !110 -> o0 retrieve remainder

```

Examples of UNSIGNED division

```

! 11/3 = 3 & 2 rmdr; 011-> o2, 011-> o1, 001-> o0
!Y o1 TS ALUin ALUout
mov %o0,%y ! msh dividend -> Y reg
!001 011
tst %g0 ! initialize cc as non negative
!001 0111 0 dividend
!111 1110 1 divide step 1
!111+0011 0010 divide step 2
!010 1101 0
divscc %o1,%o2,%o1 ! 0101-0011 0010 divide step 3
!010 011 0 TS is last remainder sign
mov %y,%o0 !010 -> o0 retrieve remainder
!---
! reg o0
!010 011 0
bl,a 1f
!010 011

```

```

add    %o0,%o2,%o0    !          adjust remainder if negative
                        !010 011
1:nop

!      33/5 = 6 & 3 rmdr; 101-> o2, 001-> o1, 100-> o0
                        !Y   o1   TS  ALUin   ALUout
mov    %o0,%y         !100 001          msh dividend -> Y reg
tst    %g0            !          initialize cc as non negative
                        !100 0101 0      dividend
divscc %o1,%o2,%o1    !          1000-0101 0011      divide step 1
                        !011 0111 0
divscc %o1,%o2,%o1    !          0110-0101 0001      divide step 2
                        !001 1111 0
divscc %o1,%o2,%o1    !          0011-0101 1110      divide step 3
                        !110 110 1      TS is last remainder sign
mov    %y,%o0         ! 110 -> o0      retrieve remainder
                        !---
!          reg o0
                        !110 110 1
bl,a   1f
                        !110 110
add    %o0,%o2,%o0    !          110+101 011      adjust remainder if negative
                        !011 110
1:nop

```

5.5.2 Signed Division with Doubleword Dividend (divs2)

This subroutine for signed division of a 64-bit dividend by a 32-bit divisor produces a 32-bit signed quotient and a 32-bit remainder. Special treatment is given to borderline overflow when the absolute value of the quotient is 2^{31} , in order to support the math operator INTEGER PART OF: $Q=-2^{31}$ does not overflow; $Q=+2^{31}$ overflows with a special overflow code.

Remainder is zero if the division is exact; otherwise, the remainder is the same sign as original dividend. There is a check for divide by zero and a check for overflow with non-zero divisor. The check for divide by zero is kept separate to support the SPARC-recommended trap for divide by zero. In applications where the user knows the numerical ranges of the operands, or controls them, these checks can be omitted. Division with divide by zero fault takes 6 cycles, sets the overflow flag in the integer condition code, and leaves 0xffff800 in register out3.

Division with non-zero divisor overflow takes 17 to 23 cycles (17 or 19 if the original dividend is positive, 18 or 23 if the original dividend is negative); it sets the overflow flag in the integer condition code, and leaves 0x800 in register out3.

Division leading to a quotient of absolute value 2^{31} takes 20 cycles if the original dividend is positive, and 23 cycles if the original dividend is negative. It leaves the correct remainder in register out0, -2^{31} in out1 as quotient and 0 in out3. It

clears the overflow condition code if the actual quotient is -2^{31} , and sets the overflow condition code if the actual quotient is $+2^{31}$.

Division without fault takes 49 to 60 cycles; it clears the overflow condition code, and leaves 0 in register out3. Exact division with last partial remainder = 0 takes 49 cycles. Exact division with last partial remainder = \pm divisor, as happens with non-restoring division algorithms, takes 53 or 54 cycles. Inexact division, with non-zero final remainder, takes 56 to 60 cycles.

!Calling Convention

```
! mov    %l0,%o0      !msh dvdnd->o0
! mov    %l1,%o1      !lsh dvdnd->o1
! call   divs2        !DIVISION SUBROUTINE CALL
! orcc   %g0,%l2,%o2  !dvsr->o2 & test
```

!Register Map

```
! reg#
! out0   msh dividend/remainder
! out1   lsh dividend/quotient
! out2   divisor
! out3   overflow indication
!        overflow divide by zero/0xffff800 and V=1
!        overflow divide by non-zero/0x800 and V=1
!        overflow quotient =+2^31/0 and V=1
!        no overflow/0 and V=0
! out4   scratch for final remainder calculations
! out5   absolute value of divisor
! y      msh dividend/successive partial remainders
! call to divs2 must be made with cc indicating sign of divisor
```

.global divs2

```
divs2: bne    0f          !go on if divisor not zero
        mov    %o2,%o5    !copy divisor in o5, D
        sethi  0x1fffff,%o3 !divide by zero indicator
        retl   !exit with
        addcc  %o3,%o3,%o3 !overflow set
0:      bl,a   1f
        sub    %g0,%o5,%o5 !if divsr neg, D=-divsr
1:      mov    %o0,%y      !msh dvdnd->Y
        tst    %o0        !initialize cc for first divide step
        !with sign dividend for signed divide
        bl    2f          !skip ahead for negative dividend
        divscc %o1, %o5, %o1 !divide step 1

!don't change cc except by DIVSCC until last divide step done

        bl    3f          !ok if different
```

```

mov    %g0,%o3          !clear overflow indicator
srl    %o1,1,%o4        !get lsh rmdr
bg     8f               !if msh rmdr >0 then overflow
subcc  %o4,%o5,%g0      !if lsh rmdr <D then Q is +/-2^31
bge    8f               !& o4 is correct final rmdr
sethi  0x200000,%o1     !check if overflow on Q = +2^31
                        !set -2^31 -> Q
                        !else overflow
tst    %o2              !if original divisor >0
bg,a   9f               !which implies quotient =+2^31
addcc  %o1,%o1,%g0      !set ovrlfw cc with o3 = 0
9:     retl              !exit
mov    %o4,%o0          !with correct remainder in o0
8:     sethi 0x200001,%o3 !overflow divide by non-zero indicator
retl   !exit with
addcc  %o3,%o3,%o3      !overflow set
2:     bge    3f         !ok if different
mov    %g0,%o3          !clear overflow indicator
mov    %y,%o0           !get msh rmdr
addcc  %o0,1,%g0        !is it -1
bne    8f               !if <-1 then overflow
srl    %o1,1,%o4        !get lsh rmdr except for leading 1
sethi  0x200000,%o1     !set -2^31 ->Q
or     %o1,%o4,%o4      !insert leading 1 in lsh rmdr
addcc  %o4,%o5,%g0      !if lsh rmdr >-D then q is +/-2^31
ble    8f               !& o4 is correct final rmdr
                        !check if overflow on Q = +2^31
                        !else overflow
tst    %o2              !if original divisor <0
bl,a   9f               !which implies quotient =+2^31
addcc  %o1,%o1,%g0      !set ovrlfw cc with o3 = 0
9:     retl              !exit
mov    %o4,%o0          !with correct remainder in o0
8:     sethi 0x200001,%o3 !overflow divide by non-zero indicator
retl   !exit with
addcc  %o3,%o3,%o3      !overflow set
3:     divscc %o1, %o5, %o1 !divide step 2
divscc %o1, %o5, %o1     !divide step 3
.
.
.
divscc %o1, %o5, %o1     !
divscc %o1, %o5, %o1     !divide step 32

be     6f               !if final remainder is zero,
                        !go fix quotient polarity
mov    %y, %o4          !final remainder from Y to o4
bg     4f               !skip ahead if rmdr+; continue if rmdr-
addcc  %o4,%o5,%g0      !is neg rmdr + abs divsr =0
be,a   6f               !if so, go fix quotient polarity and
mov    %g0,%o4          !clear rmdr. if not, don't clear

```

```

        tst    %o0          !test original dvdnd
        bl    5f          !if neg, go check neg Q
        tst    %o1          !sign Q
        ba    5f
        add    %o4,%o5,%o4    !if orig dvdnd pos and final rmdr neg,
                                !correct rmdr; then go check neg Q
4:      subcc  %o4,%o5,%g0    !is pos rmdr - abs divsr =0
        be,a  6f          !if so, go fix quotient polarity and
        mov    %g0,%o4      !clear rmdr. if not, don't clear
        tst    %o0          !test original dvdnd
        bge   5f          !if pos, go check neg Q
        tst    %o1          !sign Q
        sub    %o4,%o5,%o4    !if orig dvdnd neg and final rmdr pos,
                                !correct rmdr; then go check neg Q
5:      bl,a  6f          !skip ahead if Q pos
        add    %o1,1,%o1    !if neg Q, 1's complement to
                                !2's complement; annul if pos Q
6:      tst    %o2          !check original divisor sign
        bl,a  7f
        sub    %g0,%o1,%o1    !if neg divsr, negate quotient
7:      retl
        mov    %o4,%o0      !with correct remainder in o0

```

5.5.3 Signed Division with Word Dividend (divs1)

This subroutine for signed division of a 32-bit dividend by a 32-bit divisor produces a 32-bit signed quotient and a 32-bit remainder. Remainder is zero if the division is exact; otherwise the remainder is the same sign as the original dividend. There is no check for divide by zero. It is not possible to overflow with non-zero divisor. If the calling routine knows that divide by zero cannot happen, no test is needed. If divide by zero is possible, a simple test just after the call can abort the division.

Division without fault takes 47 to 58 cycles. Exact division with last partial remainder = 0 takes 47 cycles. Exact division with last partial remainder = \pm divisor, as happens with non-restoring division algorithms, takes 51 or 52 cycles. Inexact division, with non-zero final remainder, takes 54 to 58 cycles.

!Calling Convention

```

!      mov    %l1,%o0      !dvdnd->o0
!      orcc  %g0,%l2,%o2    !divsr->o2 & test
!      call  divs1         !DIVISION SUBROUTINE CALL
!      be    dvby0        !abort division if divide by zero

```

!Register Map

```

!      reg#

```

```

!   out0  dividend/remainder
!   out1  quotient
!   out2  divisor
!   out4  scratch for final remainder calculations
!   out5  absolute value of divisor
!   y     initially sign extension of dividend/successive partial
!         remainders. call to divs1 must be made with cc indicating
!         sign of divisor

.global divs1
divs1: mov    %g0,%y           !0 -> Y
      mov    %o2,%o5         !copy divisor in o5, D
      bl,a   1f
      sub    %g0,%o5,%o5     !if divsr neg, D=-divsr
1:     tst    %o0             !initialize cc for first divide step with
      !sign dividend for signed divide

      bl,a   2f
      mov    -1,%y           !-1 -> Y only if dvdnd neg
2:     divscc %o0, %o5, %o1   !divide step 1
      !leave original dividend in o0.
      !do partial remainders & quotient in o1
      !don't change cc except by divscc until
      !last divide step is completed

      divscc %o1, %o5, %o1   !divide step 2
      divscc %o1, %o5, %o1   !divide step 3
      divscc %o1, %o5, %o1   !divide step 4
      .
      .
      .
      divscc %o1, %o5, %o1
      divscc %o1, %o5, %o1   !divide step 32
      be     6f              !if final remainder is zero,
      !go fix quotient polarity

      mov    %y, %o4         !final remainder from Y to o4
      bg     4f              !skip ahead if rmdr+; continue if rmdr-
      addcc  %o4,%o5,%g0     !is neg rmdr + abs divsr =0
      be,a   6f              !if so, go fix quotient polarity and
      mov    %g0,%o4         !clear rmdr. if not, don't clear
      tst    %o0             !test original dvdnd
      bl     5f              !if neg, go check neg Q
      tst    %o1             !sign Q
      ba     5f
      add    %o4,%o5,%o4     !if orig dvdnd pos and final rmdr neg,
      !correct rmdr; then go check neg Q
4:     subcc  %o4,%o5,%g0     !is pos rmdr - abs divsr =0
      be,a   6f              !if so, go fix quotient polarity and
      mov    %g0,%o4         !clear rmdr. if not, don't clear
      tst    %o0             !test original dvdnd
      bge    5f              !if pos, go check neg Q
      tst    %o1             !sign Q
      sub    %o4,%o5,%o4     !if orig dvdnd neg and final rmdr pos,

```

```
5:      bl,a   6f          !correct rmdr; then go check neg Q
      add    %o1,1,%o1    !skip ahead if Q pos
                          !if neg Q, 1's complement to
                          !2's complement; annul if pos Q
6:      tst    %o2          !check original divisor sign
      bl,a   7f
      sub    %g0,%o1,%o1 !if neg divsr, negate quotient
7:      retl          !exit
      mov    %o4,%o0     !with correct remainder in o0
```

5.5.4 Unsigned Division with Doubleword Dividend (divu2)

This subroutine for unsigned division of a 64-bit dividend by a 32-bit divisor produces a 32-bit unsigned quotient and a 32-bit remainder. Remainder is zero if the division is exact, and positive otherwise. There is a check for divide by zero and a check for overflow with non-zero divisor. The check for divide by zero is kept separate in order to support the SPARC-recommended trap for divide by zero. In applications where the user knows the numerical ranges of the operands, or controls them, these checks can be omitted.

Division with divide by zero fault takes 6 cycles; it sets the overflow flag in the integer condition code, and leaves 0xffff800 in register out3. Division with a non-zero divisor overflow takes 9 cycles; it sets the overflow flag and leaves 0x800 in register out3. Division without fault takes 42 cycles, clears the overflow flag, and leaves 0 in register out3.

!Calling Convention

```
! mov    %l0,%o0        !msh dvdnd->o0
! mov    %l1,%o1        !lsh dvdnd->o1
! call   divu2          !DIVISION SUBROUTINE CALL
! orcc   %g0,%l2,%o2    !divsr->o2 & test
```

!Register Map

```
! reg#
! out0   msh dividend/remainder
! out1   lsh dividend/quotient
! out2   divisor
! out3   overflow indication
!        overflow divide by zero/0xffff800 and V=1
!        overflow divide by non-zero/0x800 and V=1
!        no overflow/0 and V=0
! y      msh dividend/successive partial remainders
!        call to divs2 must be made with cc indicating if divisor zero
```

```

global divu2
divu2: bne    1f                !go on if divisor not zero
      mov     %o0,%y           !msh dvdnd->Y
      sethi   0x1ffffff,%o3    !divide by zero indicator
      retl                    !exit with
      addcc   %o3,%o3,%o3      !overflow set
1:     subcc   %o0,%o2,%g0      !is msh dvdnd < dvsr
      bcs     2f                !ok if so
      orcc    %g0,0,%o3        !initialize cc for first divide step
      !with positive sign for unsigned divide
      !clear overflow indicator
      sethi   0x200001,%o3     !overflow divide by non-zero indicator
      retl                    !exit with
      addcc   %o3,%o3,%o3      !overflow set
2:     divscc %o1, %o2, %o1     !divide step 1
      !don't change cc except by divscc until
      !last divide step is completed
      divscc %o1, %o2, %o1     !divide step 2
      divscc %o1, %o2, %o1     !divide step 3
      .
      .
      .
      divscc %o1, %o2, %o1
      divscc %o1, %o2, %o1     !divide step 32

      bl     3f                !skip ahead if rmdr-
      mov     %y,%o0           !final remdr from Y to o0
      retl                    !exit
      addcc   %o0,0,%o0        !clear ovrflw cc if on
3:     retl                    !exit
      addcc   %o0,%o2,%o0      !correct rmdr & clear ovrflw cc if on

```

5.5.5 Unsigned Division with Word Dividend (divu1)

This subroutine for unsigned division of a 32-bit dividend by a 32-bit divisor produces a 32-bit unsigned quotient and a 32-bit remainder. Remainder is zero if the division is exact, and positive otherwise. There is no check for divide by zero. It is not possible to overflow with non zero divisor. If the calling routine knows that divide by zero cannot happen, no test is needed. If divide by zero is possible, a simple test just after the call can abort the division.

If not aborted, the division takes 39 cycles; it clears overflow flag and leaves 0 in register out3. If the remainder is of no interest and only the quotient corresponding to INTEGER(dvdnd/dvsr) or FLOOR(dvdnd/dvsr) for unsigned numbers is wanted, then the last steps of this routine can be modified as indicated. Quotient-only unsigned division takes 36 cycles.

!Calling Convention

```

! mov  %l1,%o1      !dvdnd->o1
! orcc %g0,%l2,%o2  !dvsr->o2 & test
! call divul        !DIVISION SUBROUTINE CALL
! be   dvby0        !abort division if divide by zero

```

!Register Map

```

! reg#
! out0  remainder
! out1  dividend/quotient
! out2  divisor
! out3  0 if divide by non zero
! y     zero/successive partial remainders

```

.global divul

```

divul: mov  %g0,%y      !0->Y
      orcc %g0,0,%o3   !initialize cc for first divide step
                        !with positive sign for unsigned divide
                        !clear divide by zero indicator
      divscc %o1, %o2, %o1 !divide step 1
                        !don't change cc except by divscc until
                        !last divide step is completed
      divscc %o1, %o2, %o1 !divide step 2
      divscc %o1, %o2, %o1 !divide step 3
      .
      .
      .
      divscc %o1, %o2, %o1
      divscc %o1, %o2, %o1 !divide step 31
      retl      !exit for quotient-only divide
      divscc %o1, %o2, %o1 !divide step 32

```

!ALL the following steps may be omitted for quotient-only divide

```

      bl   1f          !skip ahead if rmdr-
      mov  %y,%o0      !final rmdr from Y to o0
      retl      !exit
      addcc %o0,0,%o0  !clear ovrflw cc if on
1:    retl      !exit
      addcc %o0,%o2,%o0 !correct rmdr & clear ovrflw cc if on

```

5.5.6 Divide Step In Support Of A To D Converter Compensation

The following code fragment shows compensation for errors in quantization codes of an analog to digital converter that has been calibrated with the Walsh Transform techniques developed at Schlumberger (Fairchild) Test Systems. Refer to "A System For Converter Testing Using Walsh Transform Techniques" by E.A.

Sloane presented as paper 11.3 at the IEEE International Test Conference, October 1981.

As the paper shows, for well designed and manufactured analog to digital converters, the relation between codes and actual voltage values of the mid point of each quantization bin is as close to linear as technology and economics permit. So the power of two order Walsh coefficients dominate over the cross terms. Consequently, this example only uses the quantization bits as is and doesn't cover the exclusive or combinations between some of the more significant bits. For each bit of additional accuracy, only another instruction pair of add & set condition codes and divide step is required. To do this with table lookup would require doubling the table size, consuming data cache. Simple gain and offset corrections based on least square linear fit don't offer as much accuracy and usually are based on static rather than dynamic tests, which are more suited to actual use.

The operation shown in the code fragment is:

$$Y_{reg} = \pm 2^9 \times A9 \pm 2^8 \times A8 \dots \pm 2^0 \times A0$$

At each stage whether the next term is added or subtracted depends on whether the corresponding bit of quantization in a register pointed to by symbol x is 0/1.

```

mov    0,%y          !clear Yreg
addcc  x,x,x         !left shift code from upper bits of register x
                          !with msb setting N & V to force true sign
divscc %g0,A9,%g0    !only add or subtract immediate value to Yreg
                          !no other register is affected

addcc  x,x,x
divscc %g0,A8,%g0
addcc  x,x,x
divscc %g0,A7,%g0
addcc  x,x,x
divscc %g0,A6,%g0
addcc  x,x,x
divscc %g0,A5,%g0
addcc  x,x,x
divscc %g0,A4,%g0
addcc  x,x,x
divscc %g0,A3,%g0
addcc  x,x,x
divscc %g0,A2,%g0
addcc  x,x,x
divscc %g0,A1,%g0
addcc  x,x,x

```

```
divscc %g0,A0,%g0
mov    %y,%g1    !g1 holds compensated value of quantization code
                !from x scaled by a factor chosen to make most
                !use of the 13 bit precision available for
                !immediate values. Here with 10 bits, results
                !are scaled by 2^9 relative to coefficients.
```

As an example, a 10 bit offset binary analog to digital converter might be set to operate over a range of -5.12 to +5.12 volts with nominal 10 millivolt quantization resolution. If ideal, with no errors, the coefficients for each bit expressed as millivolts would be:

m	9	8	7	6	5	4	3	2	1	0
a(m)	-2560	-1280	-640	-320	-160	-80	-40	-20	-10	-5

If the process technology is limited to ± 0.5% accuracy of the converter's resistive ladder, then the actual coefficients for each bit in millivolts could be:

m	9	8	7	6	5	4	3	2	1	0
a(m)	-2572.59	-1274.24	-642.94	-319.97	-159.87	-80.34	-39.86	-20.02	-10.05	-4.98

These coefficients would be scaled by 2^{9-m} , corresponding to the order of entering Yreg which gets left shifted each time, and rounded to integer.

m	9	8	7	6	5	4	3	2	1	0
A(m)	-2573	-2548	-2572	-2560	-2558	-2571	-2551	-2563	-2572	-2547

Driving the analog to digital converter with a 4.000 Volts, 5 MHz sine wave, sampling at 64 MHz and collecting 64 consecutive samples allows performing spectrum analysis with FFT to determine effective bits under the test conditions. Because of the sine wave frequency relative to the sample frequency, the significant distortion harmonics don't alias into the fundamental frequency analysis bin. Number of effective bits is approximately:

$$\frac{0.5 \times \log \left(\frac{2}{3} \times \frac{\text{power spectrum at fundamental}}{\text{sum of power spectrum at all other frequencies}} \right)}{\log (2)}$$

The nominal 10 bit converter with ideal coefficients at each code bit shows 9.52 effective bits under dynamic rather than static testing. The converter with ± 0.5%

errors in the resistive ladder taken at nominal value without Walsh based calibration shows 7.57 effective bits. With Walsh base calibration, it shows 9.05 effective bits. A least square straight line fit for compensation shows only 7.57 effective bits but with reduced error in measuring peak amplitude.

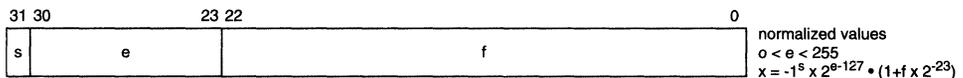
This less obvious use of divide step allows fast compensation for an appropriately calibrated analog to digital converter. Recovery for this example of about 3/4 of the lost number of effective bits at the price of two cycles per quantization bit plus 2 cycles overhead.

5.6 Using the SCAN Instruction

The code examples in this section illustrate the use of the SCAN instruction. In the first example, SCAN is used to simplify and speed up floating-point normalization.

5.6.1 Scan in Support of Software Floating Point

The following code fragment shows post normalization of floating point add or subtract for the case where the result requires calculating the difference of the magnitudes of the numbers. The IEEE754 format, which is used in SPARC architecture as well, is assumed. This uses sign, offset exponent, hidden leading bit when normalized and fraction. Only the logic of normalize numbers is shown here. Number values are in sign and magnitude form rather than two's complement.



The operation is $x+y=z$ or $x-y=z$. If subtract, then sign y is complemented. The magnitudes of the numbers have to be compared and the one with the lesser exponent right shifted to align its decimal point with the greater. If exponents are equal, magnitudes must be compared if signs differ to see what the sign of the result will be. This is assumed to have taken place before the code fragment shown here, which shows the logic of handling numbers with different signs and different exponents. Symbol x points to the larger number; y to smaller.

```
sethi 0x3fe, %g5      !mask for sign and exponent with and
                      !or for fraction with andn
sll  %g5,1,%g4
```

```

xor    %g4,%g5,%g4    !single one at bit 23 for hidden bit
srl    x,23,%g2
and    %g2,0xff,%g2   !x exponent
srl    y,23,%g3
and    %g3,0xff,%g3   !y exponent
sub    %g2,%g3,%g1    !alignment difference
andn   y,%g5,%g3      !y fraction
or     %g3,%g4,%g3    !y hidden bit
srl    %g3,%g1,%g2    !downshift y magnitude to g2
sub    %g0,%g1,%g1    !complement of shift
sll    %g3,%g1,%g3    !upshift left over y for test
addcc  %g3,%g3,%g0    !test left over for rounding
                        !note: not IEEE754 rounding here

andn   x,%g5,%g1      !x fraction
or     %g1,%g4,%g1    !x hidden bit
subx   %g1,%g2,%g1    !difference of magnitudes with
                        !simple rounding

!-----
scan   %g1,0,%g2      !scan difference for leading one.
                        !Use of 0 as the scan mask is because
                        !of sign magnitude arithmetic assumed
                        !in this example. Leading 8 bits are
                        !guaranteed to be zero because of
                        !format. Question is, how many more
                        !till the first one?
                        !If two's complement arithmetic had
                        !been assumed, then there could have
                        !been leading ones or leading zeros
                        !depending on sign of result. Then
                        !instead of 0 as mask, scan would have
                        !used %g1 as mask as well as value.
                        !Question would have been, how many
                        !leading bits are the same as the sign?

subcc  %g2,32,%g0     !test if all significant bits lost
blu    1f             !use unsigned compare for future compatibility
sub    %g2,8,%g2      !remove effect of format's 8 leading 0's
!underflow due to loss of significant bits code would follow here

1:    sll    %g1,%g2,%g1    !normalize result
        andn   %g1,%g4,%g1    !hide leading bit
        srl    x,23,%g3
        and    %g3,0xff,%g4    !x exponent in g4
        subcc  %g4,%g2,%g0     !test exponent underflow
        bgu    2f             !use unsigned compare for future compatibility
        sub    %g3,%g2,%g3     !subtract normalization shift from
                                !result sign and exponent
!exponent underflow code would follow here

2:    sll    %g3,23,%g3     !place sign and exponent result in
                                !format position

```

```
retl          !exit(2 cycles)
or   %g1,%g3,z !combine with fraction
```

Each instruction in this code fragment runs one cycle out of instruction cache except for the leaf return which takes two. That's 32 cycles for this fragment. Without scan as a hardware instruction, the function would have to be performed as a software routine that takes 43 to 52 cycles for usual cases. The fragment would take 74 to 83 cycles, more than double. A software substitute for scan would consume instruction cache space. Attempts to speed up the binary tree search in the software routine by look-up tables based on leading bits would consume data cache space.

5.6.2 Scan in Support of Run Length Encoding

The following code fragment shows compression of long binary strings by looking for runs of all ones or all zeros and coding these so that lossless reconstruction is possible. For the example, runs less than four in length are ignored and directly transmitted and runs greater than sixteen are broken up for coding efficiency and coding simplification. Best compression occurs for low information content long binary strings such as background sections of black and white raster lines.

```
code   value
00000  reserved
00001  "
00010  "
00011  "
-----
00100  00001... or 11110...
00101  000001... or 111110...
00110  0000001... or 1111110...
.
.
.
01111  0000 0000 0000 0001... or 1111 1111 1111 1110...
10000  0000 0000 0000 0000 1... or 1111 1111 1111 1111 0...
-----
10001  0001...
10010  0010...
10011  0011...
.
.
.
11110  1110...
-----
11111  toggle
```

The code fragment omits starting up the loop, reloading buffers with new data, storing code and terminating the loop. Symbol *x* points to data segment in some register ready for compression and symbol *y* points to its immediate successor.

```
0:  scan  x,x,%g1      !scan for how many bits are same as msb.
                        !g1 = 1 to 31 or >32 if all in x register.
                        !x is used as both the value to be scanned(rs1)
                        !and the mask(rs2).
        subcc %g1,4,%g0 !test if run at least length 4
        bgeu 1f        !use unsigned compare for future compatibility
        subcc %g1,16,%g0 !test if run greater than length 16
!handle fixed length code, g1<4
        srl  x,28,%g2  !extract leading 4 bits of x as compression code
        or   %g2,16,%g2 !insert leading bit of code for fixed length
        sll  x,3,x      !shift rest of x in 2 steps
        addcc x,x,x     !complete x shift and test last of 4 bits outgoing
        bcs  2f        !separate cases for 1 or 0
        addcc x,x,%g0  !test without shifting first of remaining bits
        bcs  3f        !if last out bit =0 and first remaining bit =1
        mov  1,%g4     !set new low priority toggle indicator
        ba   3f
        mov  0,%g4     !otherwise clear toggle indicator
                        !fixed length code overwrites any pending toggle
2:  bcc  3f          !if last out bit =1 and first remaining bit =0
        mov  1,%g4     !set new low priority toggle indicator
        mov  0,%g4     !otherwise clear toggle indicator
                        !fixed length code overwrites any pending toggle
3:  srl  y,28,%g3     !extract leading 4 bits of y
        or   x,%g3,x   !move them to right end of x
        sll  y,4,y     !shift rest of y with incoming trailing zeros
        ba   5f
        subcc %g5,4,%g5 !decrement counter of how many bits of x left
!handle run length code
1:  blu  4f          !skip ahead if run less than 16
                        !use unsigned compare for future compatibility
        sll  %g4,1,%g4 !shift incoming toggle indicator to higher priority
!handle runs at least 16
        mov  16,%g2    !set compression code to 16
        sll  x,16,x    !ignore leading 16 bits of x and shift rest of x
        srl  y,16,%g3  !extract leading 16 bits of y
        or   x,%g3,x   !move them to right end of x
        sll  y,16,y    !shift rest of y with incoming trailing zeros
        ba   5f
        subcc %g5,16,%g5 !decrement counter of how many bits of x left
!handle runs of length 4 to 15
4:  mov  %g1,%g2      !set compression code to scan result
        sub  %g0,%g1,%g1 !complement scan result
```

```

sll  x,%g2,x      !ignore leading g2 bits of x and shift rest of x
srl  y,%g1,%g3   !extract leading 32-g1 bits of y
or   x,%g3,x     !move them to right end of x
sll  y,%g2,y     !shift rest of y with incoming trailing zeros
subcc %g5,%g2,%g5 !decrement counter of how many bits of x left
or   %g4,1,%g4   !toggle following compression code too
!one compression code to go
5:  bgu  6f      !skip ahead if there are still bits of x left
      !use unsigned compare for future compatibility
      subcc %g6,1,%g6 !decrement counter of code fields left
!code for reloading y and shifting part of it into x if the old y had
!trailing zeros and resetting g5 to 32-#trailing zeros.
.
.
6:  bg   7f      !skip ahead if room for more codes
      andcc %g4,2,%g0 !test if toggle has priority
!code for storing codes and reinitializing g6
.
.
7:  sll  z,5,z      !make room for new code
      be,a 0b      !if g4 bit1 off then no additional code
      !if g4 bit1 on then insert toggle code first
      or   z,%g2,z  !insert new data code
      andn %g4,2,%g4 !clear high priority toggle indicator
      !without disturbing low priority toggle indicator
      ba   5b      !check on how much code space left and append toggle
      or   z,0x1f,z !back through 5,6,7 just once
.
.

```

Each instruction in this code fragment runs one cycle out of instruction cache if it is in the active path for a particular case. Scan is in the active path for all cases. Without hardware implementation of scan, the function would require a software subroutine taking 43 to 52 cycles instead of 1 cycle. Additionally, that routine would consume instruction cache space. Alternate versions that might attempt to speed up the binary tree search with table look-up using leading bits as an index would consume data cache space.

5.7 Multiply Routines Using the MULScC Instruction

This section shows examples of doing integer multiplication using the multiply step instruction. With hardware implementation of multiply in SPARC*lite*, these routines are not required for usual situations. However, these examples illustrate how MULScC works and may serve as models for use in unusual situations.

These sample routines do not set the integer condition codes in exactly the same way as SMULcc and UMULcc Version 8 integer multiplication.

5.7.1 Simple Multiply Step Examples

In each of the following examples a cycle by cycle view of multiply step is given.

Multiply Step With Reduced Word Size (32 to 3 Bits)

```
! Register Use:
! out0 Multiplier
! out1 Multiplicand
! out2 most significant half Product
! out3 least significant half Product
! Note: TS, True Sign = N xor V from condition codes
```

Examples of SIGNED multiplication

```
!           2 * 3 = 6; 010 -> o1, 011 -> o0
!           o2   Y   TS ALUin  ALUout
mov   %o0, %y   !           multiplier -> Y reg
!           011
andcc %g0, 0, %o2 !           clear product accumulator & cc
!           !00|0  01|1  0
mulsc %o2, %o1, %o2 !           000+010  010  active multiply step 1
!           !01|0  00|1  0
mulsc %o2, %o1, %o2 !           001+010  011  active multiply step 2
!           !01|1  00|0  0
mulsc %o2, %o1, %o2 !           001+000  001  active multiply step 3
!           !00|1  10|0  0
mulsc %o2, 0, %o2   !           000+000  000  final double shift without
!           !000   110  0           add to align result
tst   %o0         !           multiplier sign?
!           !000   110  0
bl,a  1f
sub   %o2, %o1, %o2 !           adjust msh product if
!           !000   110           multiplier negative
l:mov %y, %o3      !           110 -> o3  retrieve lsh product

!           -2 * 3 = -6; 110 -> o1, 011 -> o0
!           o2   Y   TS ALUin  ALUout
mov   %o0, %y   !           multiplier -> Y reg
!           011
andcc %g0, 0, %o2 !           clear product accumulator & cc
!           !00|0  01|1  0
mulsc %o2, %o1, %o2 !           000+110  110  active multiply step 1
!           !11|0  00|1  1
mulsc %o2, %o1, %o2 !           111+110  101  active multiply step 2
!           !10|1  00|0  1
mulsc %o2, %o1, %o2 !           110+000  110  active multiply step 3
!           !11|0  10|0  1
```

```

mulsccl %o2,0,%o2      !           111+000  111      final double shift without
                        !111   010  1          add to align result
tst   %o0              !           111   010  0          multiplier sign?
                        !111   010  0
bl,a  1f               !111   010
sub   %o2,%o1,%o2     !           111   010          adjust msh product if
                        !111   010          multiplier negative
1:mov %y,%o3          !           010 -> o3      retrieve lsh product

!           3 * -2 = -6; 011 -> o1, 110 -> o0
!           o2   Y   TS  ALUin  ALUOut
mov   %o0, %y        !           110          multiplier -> Y reg
andcc %g0,0,%o2     !           110   110  0          clear product accumulator & cc
mulsccl %o2,%o1,%o2 !           000+000  000          active multiply step 1
                        !0010  0111  0
mulsccl %o2,%o1,%o2 !           000+011  011          active multiply step 2
                        !0111  0011  0
mulsccl %o2,%o1,%o2 !           001+011  100          active multiply step 3
                        !1010  1010  0
mulsccl %o2,0,%o2   !           010+000  010          final double shift without
                        !010   010  0          add to align result
tst   %o0              !           110   010  1          multiplier sign?
                        !010   010  1
bl,a  1f               !010   010
sub   %o2,%o1,%o2     !           010-011  111          adjust msh product if
                        !111   010          multiplier negative
1:mov %y,%o3          !           010 -> o3      retrieve lsh product

```

Examples of UNSIGNED multiplication

```

!           3 * 6 = 18; 011 -> o1, 110 -> o0
!           o2   Y   TS  ALUin  ALUOut
mov   %o0, %y        !           110          multiplier -> Y reg
andcc %g0,0,%o2     !           110   110  0          clear product accumulator & cc
mulsccl %o2,%o1,%o2 !           000+000  000          active multiply step 1
                        !0010  0111  0
mulsccl %o2,%o1,%o2 !           000+011  011          active multiply step 2
                        !0111  0011  0
mulsccl %o2,%o1,%o2 !           001+011  100          active multiply step 3
                        !1010  1010  0
mulsccl %o2,0,%o2   !           010+000  010          final double shift without
                        !010   010  0          add to align result
tst   %o1              !           110   010  0          msb multiplicand?
                        !010   010  0
bl,a  1f               !010   010

```

```

add    %o2,%o0,%o2  !                                adjust msh product if unsigned
                                !010   010           multiplicand treated as if
                                !                                negative
1:mov  %y,%o3       !                                retrieve lsh product

!
!      6 * 3 = 18; 110 -> o1, 011 -> o0
!      o2   Y   TS  ALUin  ALUout
mov    %o0, %y      !                                multiplier -> Y reg
                                !      011
andcc  %g0,0,%o2    !                                clear product accumulator & cc
                                !0010  0111  0
mulsc  %o2,%o1,%o2 !                                active multiply step 1
                                !1110  0011  1           000+110  110
mulsc  %o2,%o1,%o2 !                                active multiply step 2
                                !1011  0010  1           111+110  101
mulsc  %o2,%o1,%o2 !                                active multiply step 3
                                !1101  0010  1           110+000  110
mulsc  %o2,0,%o2   !                                final double shift without
                                !111   010   1           add to align result
tst    %o1         !                                msb multiplicand?
                                !111   010   1
bl,a   1f
add    %o2,%o0,%o2 !                                adjust msh product if unsigned
                                !010   010           multiplicand treated as if
                                !                                negative
1:mov  %y,%o3       !                                retrieve lsh product

```

5.7.2 Signed Multiplication Using Multiply Step

```

/*
 * Procedure to perform a 32-bit by 32-bit signed multiply.
 * Pass the multiplier in %o0, and the multiplicand in %o1.
 * The least significant 32 bits of the result are returned in %o0,
 * and the most significant in %o1. Multiplies take 47 to 51 instruction cycles.
 *
 *   call    .mul
 *   nop                    ! (or set up last parameter here)
 *
 * Note that this is a leaf routine; i.e., it calls no other routines and does
 * all of its work in the out registers. Thus, the usual SAVE and RESTORE
 * instructions are not needed.
 */

global .mul
.mul:  mov    %o0, %y      ! multiplier to Y register
      andcc  %g0, %g0, %o4 ! zero the partial product and clear N and V conditions

      mulsc  %o4, %o1, %o4 ! first iteration of 33
      mulsc  %o4, %o1, %o4
      mulsc  %o4, %o1, %o4
      .
      .

```

```

    .
    mulscc %o4, %o1, %o4
    mulscc %o4, %o1, %o4
    mulscc %o4, %o1, %o4 ! 32nd iteration
    mulscc %o4, %g0, %o4 ! last iteration only shifts
!
! if %o0 (multiplier) was negative, the result is:
!   (%o0 * %o1) + %o1 * (2**32)
! We fix that here.
!
    tst    %o0
    rd     %y, %o0
    bl,a   lf
    sub    %o4, %o1, %o4 ! bit 33 and up of the product are in
                                ! %o4, so we don't have to shift %o1
l:   retl                                ! leaf-routine return
    mov    %o4, %o1 ! return high bits

```

5.7.3 Unsigned Multiplication Using Multiply Step

```

/*
 * Procedure to perform a 32-bit by 32-bit unsigned multiply.
 * Pass the multiplier in %o0, and the multiplicand in %o1.
 * The least significant 32 bits of the result are returned in %o0,
 * and the most significant in %o1. Multiplies take 46 or 58 instruction cycles.
 *
 *   call    .umul
 *   nop                    ! (or set up last parameter here)
 *
 * Note that this is a leaf routine; i.e., it calls no other routines and does
 * all of its work in the out registers. Thus, the usual SAVE and RESTORE
 * instructions are not needed.
 */

.global .umul
.mul: mov    %o0, %y        ! multiplier to Y register
      andcc %g0, %g0, %o4 ! zero the partial product and clear N and V conditions

      mulscc %o4, %o1, %o4 ! first iteration of 33
      mulscc %o4, %o1, %o4
      mulscc %o4, %o1, %o4
      .
      .
      mulscc %o4, %o1, %o4
      mulscc %o4, %o1, %o4
      mulscc %o4, %o1, %o4 ! 32nd iteration
      mulscc %o4, %g0, %o4 ! last iteration only shifts
/*
 * Normally, with the shift and add approach, if both numbers are
 * positive, you get the correct result. With 32-bit two's-complement
 * numbers, -x can be represented as ((2 - (x/ (2**32))) mod 2) * 2**32)
 * To avoid a lot of 2**32's, we just move the radix point up to be
 * just to the left of the sign bit. So:
 *

```

```

*   x * y = (xy) mod 2
*  -x * y = (2 - x) mod 2 * y = (2y - xy) mod 2
*   x * -y = x * (2 - y) mod 2 = (2x - xy) mod 2
*  -x * -y = (2 - x) * (2 - y) = 4 - 2x - 2y + xy) mod 2
*
* For signed multiplies, we subtract (2**32) * x from the partial
* product to fix this problem for negative multipliers (see .mul in
* Section 1.
* because of the way the shift into the partial product is calculated
* (N xor V), this term is automatically removed for the multiplicand,
* so we don't have to adjust
*
* But for unsigned multiplies, the high order bit wasn't a sign bit,
* and the correction is wrong. So for unsigned multiplies where the
* high order bit is one, we end up with xy - (2**32) * y. To fix it
* we add y * (2**32).
*/
    tst     %o1
    bl,a   lf
    add    %o4, %o0, %o4
1:   rd     %y, %o0      ! return least sig. bits of prod
    retl                    ! leaf-routine return
    mov    %o4, %o1     ! Delay slot; return high bits

```

5.7.4 Corner Turning Buffer Using Multiply Step

Multiply Step In Support Of Corner Turning Buffer For Image Processing

The following code fragment shows implementation of an 8 by 8 bit corner turning buffer in the local register files. This supports bit plane image rotation by 90 degrees. The form of the implementation uses register files to hold and manipulate the lowest level of data structure and use data cache to reduce access to the larger image plane. The multiply step is used for its ability to couple information from one register to another in a single step in a way not expected from its main purpose.

The total image plane is divided in 8 by 8 bit blocks. Blocks are accessed as groups of 4 that rotate into corresponding positions on edges square to each other. These form concentric squares.

Each byte of block loads to Yreg and controls multiply step with constant, 1 in bit 15, to make local registers 0 to 7 into corner turning buffer. The constant remains in a fixed position but the nominal partial product keeps shifting to the right, making room for new input. Choosing a large enough constant allows old processed data to remain in the local registers long enough so that it can be extracted with shift by a differing amount that depends on which processed byte is desired. This allows overlapping of storing results with fetching new input. To accommodate the need for differing shift amounts, casing is used to select one and only one

instruction out of a block on each pass. A delayed control transfer couple is formed with jump and link immediately followed in the delay slot by branch always. The target address of jump and link steps backwards by one instruction each pass. As soon as new data is removed from target destination, one byte of rotated block is stored there.

FROM this	TO	that
a7 a6 a5 a4 a3 a2 a1 a0		h7 g7 f7 e7 d7 c7 b7 a7
b7 b6 b5 b4 b3 b2 b1 b0		h6 g6 f6 e6 d6 c6 b6 a6
c7 c6 c5 c4 c3 c2 c1 c0		h5 g5 f5 e5 d5 c5 b5 a5
d7 d6 d5 d4 d3 d2 d1 d0		h4 g4 f4 e4 d4 c4 b4 a4
e7 e6 e5 e4 e3 e2 e1 e0		h3 g3 f3 e3 d3 c3 b3 a3
f7 f6 f5 f4 f3 f2 f1 f0		h2 g2 f2 e2 d2 c2 b2 a2
g7 g6 g5 g4 g3 g2 g1 g0		h1 g1 f1 e1 d1 c1 b1 a1
h7 h6 h5 h4 h3 h2 h1 h0		h0 g0 f0 e0 d0 c0 b0 a0

```
local a7a6a5a4a3a2a1a0 input 1st byte - ldub
reg
```

```
0:0...0a7 x x x x x x x x
1:0...0a6 x x x x x x x x
2:0...0a5 x x x x x x x x
3:0...0a4 x x x x x x x x
4:0...0a3 x x x x x x x x
5:0...0a2 x x x x x x x x
6:0...0a1 x x x x x x x x
7:0...0a0 x x x x x x x x
```

```
local b7b6b5b4b3b2b1b0 input 2nd byte - ldub
reg
```

```
0:0...0b7a7 x x x x x x x x
1:0...0b6a6 x x x x x x x x
2:0...0b5a5 x x x x x x x x
3:0...0b4a4 x x x x x x x x
4:0...0b3a3 x x x x x x x x
5:0...0b2a2 x x x x x x x x
6:0...0b1a1 x x x x x x x x
7:0...0b0a0 x x x x x x x x
```

```
local c7c6c5c4c3c2c1c0 input 3rd byte - ldub
reg
```

```
0:0...0c7b7a7 x x x x x x x x
1:0...0c6b6a6 x x x x x x x x
2:0...0c5b5a5 x x x x x x x x
3:0...0c4b4a4 x x x x x x x x
4:0...0c3b3a3 x x x x x x x x
5:0...0c2b2a2 x x x x x x x x
6:0...0c1b1a1 x x x x x x x x
7:0...0c0b0a0 x x x x x x x x
```

*

```

*
*
local  h7h6h5h4h3h2h1h0 input 8th byte - ldub
reg
0:0...0h7g7f7e7d7c7b7a7 x x x x x x x x <1
1:0...0h6g6f6e6d6c6b6a6 x x x x x x x x
2:0...0h5g5f5e5d5c5b5a5 x x x x x x x x
3:0...0h4g4f4e4d4c4b4a4 x x x x x x x x
4:0...0h3g3f3e3d3c3b3a3 x x x x x x x x
5:0...0h2g2f2e2d2c2b2a2 x x x x x x x x
6:0...0h1g1f1e1d1c1b1a1 x x x x x x x x
7:0...0h0g0f0e0d0c0b0a0 x x x x x x x x
      A7A6A5A4A3A2A1A0 next edge byte 1 - ldub
local  h7g7f7e7d7c7b7a7 output rotated byte 1 - stb <1
reg
0:0...0A7h7g7f7e7d7c7b7a7 x x x x x x x x
1:0...0A6h6g6f6e6d6c6b6a6 x x x x x x x x <2
2:0...0A5h5g5f5e5d5c5b5a5 x x x x x x x x
3:0...0A4h4g4f4e4d4c4b4a4 x x x x x x x x
4:0...0A3h3g3f3e3d3c3b3a3 x x x x x x x x
5:0...0A2h2g2f2e2d2c2b2a2 x x x x x x x x
6:0...0A1h1g1f1e1d1c1b1a1 x x x x x x x x
7:0...0A0h0g0f0e0d0c0b0a0 x x x x x x x x
      B7B6B5B4B3B2B1B0 next edge byte 2 - ldub
local  h6g6f6e6d6c6b6a6 output rotated byte 2 - stb <2
reg
0:0...0B7A7h7g7f7e7d7c7b7a7 x x x x x x x x
1:0...0B6A6h6g6f6e6d6c6b6a6 x x x x x x x x
2:0...0B5A5h5g5f5e5d5c5b5a5 x x x x x x x x <3
3:0...0B4A4h4g4f4e4d4c4b4a4 x x x x x x x x
4:0...0B3A3h3g3f3e3d3c3b3a3 x x x x x x x x
5:0...0B2A2h2g2f2e2d2c2b2a2 x x x x x x x x
6:0...0B1A1h1g1f1e1d1c1b1a1 x x x x x x x x
7:0...0B0A0h0g0f0e0d0c0b0a0 x x x x x x x x
      C7C6C5C4C3C2C1C0 next edge byte 3 - ldub
      h5g5f5e5d5c5b5a5 output rotated byte 3 - stb <3
*
*
*
local
reg
0:0...0G7F7E7D7C7B7A7h7g7f7e7d7c7b7a7 x
1:0...0G6F6E6D6C6B6A6h6g6f6e6d6c6b6a6 x
2:0...0G5F5E5D5C5B5A5h5g5f5e5d5c5b5a5 x
3:0...0G4F4E4D4C4B4A4h4g4f4e4d4c4b4a4 x
3:0...0G3F3E3D3C3B3A3h3g3f3e3d3c3b3a3 x
5:0...0G2F2E2D2C2B2A2h2g2f2e2d2c2b2a2 x
6:0...0G1F1E1D1C1B1A1h1g1f1e1d1c1b1a1 x
7:0...0G0F0E0D0C0B0A0h0g0f0e0d0c0b0a0 x <8
      H7H6H5H4H3H2H1H0 next edge byte 8 - ldub
      h0g0f0e0d0c0b0a0 output rotated byte 8 - stb <8

```

```

*
*
*
/* INNER LOOP 0 for each square, position, edge, byte */
t0: ldub [%i1+%i4],%o1 !get input for next pass
      !i1 is base of fetch, controlled elsewhere
      !i4 is pointer to target byte
      mulsc %l1,%o5,%l1 !finish corner turning with previous input
      mulsc %l0,%o5,%l0 !garbage 1st time, reg o5 = 2^15
      sra %i4,4,%i4 !downshift adrs pointer for extract pointer
      mov %o1,%y !new input
      jmp %g1+%i4,%g0 !for input registers i=7->0 (%i4=-i)
      ba t2 !select 1 extract result instruction
!only one srl %lx,z,%o0 done on each pass
!use of casing keeps code compact while still avoiding self modifying code
!g1 points to t1
      srl %l0,8,%o0
      srl %l1,7,%o0
      srl %l2,6,%o0
      srl %l3,5,%o0
      srl %l4,4,%o0
      srl %l5,3,%o0
      srl %l6,2,%o0
t1: srl %l7,1,%o0
t2: sll %i4,4,%i4 !upshift extract pointer for adrs offset
      stb %o0,[%i0+%i4] !store 1 result
      !i0 is base of store, controlled elsewhere
      !i0 = i1 3 times out of 4
      mulsc %l7,%o5,%l7 !start corner turning with new input
      mulsc %l6,%o5,%l6
      mulsc %l5,%o5,%l5
      mulsc %l4,%o5,%l4
      mulsc %l3,%o5,%l3
      mulsc %l2,%o5,%l2
      addcc %i4,64,%i4 !dec adrs offset
      ble t0
      orcc %g0,1,%g0 !set N & V =0
      !keep left input to multiply partial
      !product zero
*
*
*

```

This less obvious use of multiply step and less common use of delayed control transfer couple allow efficient implementation of a fast corner turning buffer to support bit plane image processing.

CHAPTER 6



System Design Considerations

The MB86930 SPARClite microcontroller is suitable for a wide range of embedded controller applications due to its high performance and low unit cost. In designing a system, several issues and trade-offs must be considered to balance the needs of performance, low hardware cost, low development cost, and short time to market. This chapter provides detailed information on some specific design considerations:

- The clock signals and type of clock source
- The sizes, types, and interface requirements of the system memory and peripherals
- The possible need for DMA capability and bus arbitration
- The possible use of an MB86940 Peripheral Chip for interrupt control, timers, and USARTs
- In-circuit emulation capability
- Other hardware implementation issues

6.1 Clocks

Either of two possible clock sources can be used to drive a SPARClite system: the internal oscillator of the MB86930 processor, or a separate external oscillator. In the former case, a crystal is connected across inputs XTAL1 and XTAL2. In the latter case, the clock signal is connected to the XTAL1 input pin; XTAL2 is left unconnected. Using the internal oscillator has a lower hardware cost, but is less flexible than using an external oscillator.

There are two clock output signals from the processor, CLKOUT1 and CLKOUT2. CLKOUT1 has the same frequency and phase as the internal oscillator or the signal applied to XTAL1. CLKOUT2 is the same as CLKOUT1, but phase-shifted 180 degrees. The rising edge of either CLKOUT1 or CLKOUT2 can be used by the external system for timing purposes.

The output clocks are controlled by a phase-locked loop implemented in the processor. The phase-locked loop minimizes the skew between the input clock signal and CLKOUT1, and controls the duty cycles of the output clocks. The input clock signal applied to XTAL1 can have a relatively wide range of duty cycles. (See the data sheet for the clock timing specifications.) The duty cycle of the output clocks is somewhat less than 50%, reflecting the fact that the processor requires its internal clock phases to have non-overlapping transitions.

The drive capability of the clock output signals is limited. Depending on the number of inputs that must be driven and the clock speed, it may be necessary to buffer these signals for use elsewhere in the system. To minimize clock skew for systems that exceed the drive capability of CLKOUT1 or CLKOUT2, a buffered external clock can be used to drive both the processor and the system.

6.2 Memory and I/O Interfacing

The SPARClite processor minimizes the need for external logic by providing a programmable on-chip address decoder and six independent chip-select output signals. The address decoder compares the current address against the programmed address ranges, and automatically asserts the appropriate chip-select signal. The on-chip address decoder is more economical than a separate external decoder, and also operates faster.

Each programmable address range has an associated wait-state generator, which generates a Ready signal internally at a programmed number of access cycles. Either this internal Ready signal can be used, or the conventional -READY signal input from the external memory controller can be used to end the transaction. The processor can also be programmed to use the internal wait-state generator, while allowing the -READY signal to override the internal count to end the bus cycle

sooner. The internally generated Ready signal is not visible external to the processor.

If you use a single chip-select signal from the processor to select multiple memory or I/O devices, all those devices will have the same number of wait states generated when they are accessed. Different chip select signals, however, can be individually programmed to different numbers of wait states.

Any area of memory not mapped to one of the chip selects (-CS5-0) will use the external -READY.

6.2.1 Interfacing SRAM

The address bus, data bus, and chip select signals of the SRAM can be connected directly to the address bus, data bus and a chip select of the processor. The output enable signal can be generated by gating RD/-WR high and Chip select low to produce output enable low. Write enable for the SRAMs requires more consideration.

The processor data hold time for a write is specified as zero hold after rising edge of clock. RD/-WR hold time at the end of a write operation can be 0 after rising edge of clock, or can be held low if the next cycle is also a write. Thus an implementation cannot use RD/-WR directly as -WE for the SRAMs.

Figure 6-1 shows a timing diagram for an example implementation using 2 cycle access SRAM running at 40 MHz. It was implemented in a combinatorial PAL (see Figure 6-4). Individual -WE signals are generated for each of the 4 bytes in the data word.

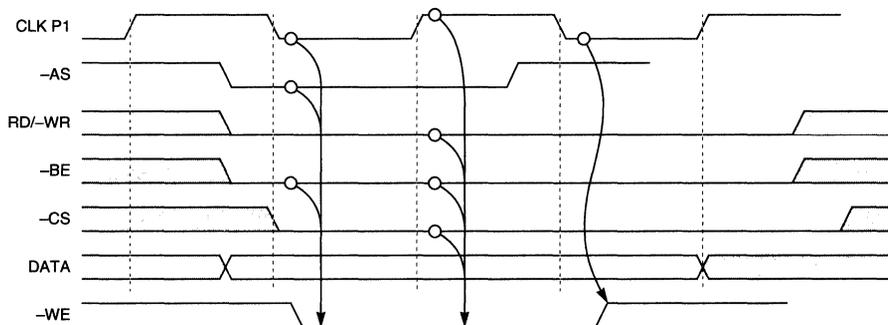


Figure 6-1. SRAM Interfacing Example

```
!clkd = !clkp1;
!soe_ = rw & !scs_;
!swe3_ = !rw & !as_ & !be3_ & !clkp1
        # !rw & !as_ & !be3_ & !clkd
        # !rw & !scs_ & !swe3_ & clkp1
        # !rw & !scs_ & !swe3_ & clkd;

!swe2_ = !rw & !as_ & !be2_ & !clkp1
        # !rw & !as_ & !be2_ & !clkd
        # !rw & !scs_ & !swe2_ & clkp1
        # !rw & !scs_ & !swe2_ & clkd;

!swe1_ = !rw & !as_ & !be1_ & !clkp1
        # !rw & !as_ & !be1_ & !clkd
        # !rw & !scs_ & !swe1_ & clkp1
        # !rw & !scs_ & !swe1_ & clkd;

!swe0_ = !rw & !as_ & !be0_ & !clkp1
        # !rw & !as_ & !be0_ & !clkd
        # !rw & !scs_ & !swe0_ & clkp1
        # !rw & !scs_ & !swe0_ & clkd;
```

Clock low and -AS low and -BE low and $\text{RD}/\text{-WR}$ low cause -WE to be asserted. Clock high and -CS low and -BE low and $\text{RD}/\text{-WR}$ low cause -WE to stay low. When clock goes low again, -WE is negated. This way there is sufficient data hold time.

For this implementation, CLKOUT1 from the processor was used since it has better duty cycle control than an oscillator clock.

6.2.2 Interfacing Page-Mode DRAM

Interfacing Dynamic RAM requires a DRAM controller for generating RAS and CAS (Row Address Strobe and Column Address Strobe), and for handling refresh. The DRAM controller is typically implemented as a state machine. The DRAM controller and signal interfaces should be designed carefully to accommodate refresh operations and fast page mode access.

The programmable 16-bit timer provided in the SPARClite processor can be used for timing the refresh interval. The timer output signal, -TIMER_OVF (Timer Overflow), goes low for a single clock cycle at the end of each timer interval. The timer interval is programmed in software, the correct amount of time depending on how the refresh operation is implemented.

There are two ways to implement the correct number of wait states: either the processor's internal wait-state generator can be used, or the DRAM controller can generate a -READY signal for the processor.

The processor supports fast "page mode" access to DRAM. When the current DRAM address is within the same page as the previous DRAM access, the -SAME_PAGE (Same-Page Detect) signal is asserted. This tells the DRAM controller that DRAM can be accessed using CAS only, without selecting a new row of the DRAM, saving time. Page-mode accesses thus provide timing advantages comparable to the burst-mode accesses of some other processors.

To take advantage of page hits, RAS is asserted and left asserted to continuously select a row. CAS is asserted, one access at a time, to select a memory location in that row. Accesses need not be in consecutive locations. As long as each access is in the same row, RAS can be left asserted and CAS asserted once to access each memory location. RAS remains asserted between accesses.

The wait-state generator can be programmed to use a different (smaller) number of clock cycles for a "page hit" (when the current address is within the same page as the previous DRAM access).

When using the internal wait-state generator instead of the external -READY signal, the processor has no way of detecting a refresh operation that occurs during an access. One solution is to have the DRAM controller take control of the bus during refresh using -BREQ (Bus Request), thereby preventing the processor from requesting a memory access for the duration of the refresh operation. The disadvantage of this solution is that the processor is forced to remain idle. An alternative solution is to disable the internal wait-state generator and let the DRAM controller generate the -READY signal for all DRAM accesses.

Figure 6-2 is a simplified state diagram for a DRAM memory controller. Upon reset, the state machine starts in the RAS Precharge and Idle state, and remains in that state until a memory access or refresh request occurs.

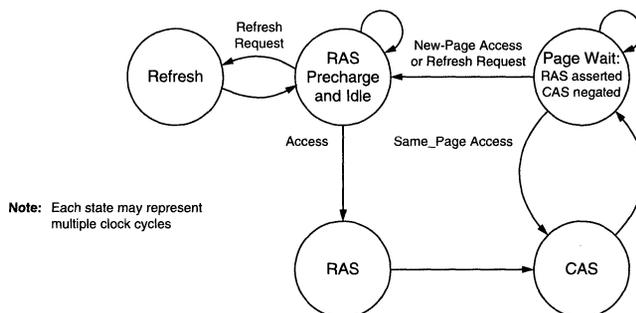


Figure 6-2. Simplified State Diagram for DRAM Controller

If a refresh request occurs, the state machine goes into the Refresh state. (In practice, this will actually be a number of sequential states.) When the refresh operation is complete, the state machine returns to the RAS Precharge and Idle state.

When the processor requests a DRAM memory access, the state machine enters the RAS state, in which the RAS signal is asserted to select the row. From there it goes to the CAS state, in which the CAS signal is asserted to select the column. At this point, data is clocked into the appropriate part and the bus cycle ends.

From there the state machine enters the Page Wait state, in which the state machine waits for something to happen; either another memory access or a refresh request. In this state, RAS is asserted and CAS is negated. If there is a memory access to the same page of DRAM (as indicated by the `-SAME_PAGE` signal), the state machine goes directly to the CAS state, and CAS is asserted to select the memory location. If there is a memory access to a different page of DRAM, or if a refresh request occurs, the state machine goes to the RAS Precharge and Idle state, and from there to the requested operation. Until one of these events occurs, the state machine waits with RAS asserted.

For more information, refer to SPARClite Application Note #1 on DRAM interfacing.

6.2.3 Interfacing EPROM and Other Devices with Slow Turn-off

One characteristic of EPROM memory to consider is its relatively long turn-off time—the delay from the negation of the Chip Select input or Output Enable input to the three-stating of the data outputs. In high-speed systems, contention on the data bus between different peripheral devices can occur, depending on the organization of different memory and peripherals in the system.

When using EPROM in the system (or other memory or I/O devices that are slow to turn off), carefully study the timing diagrams in the External Interface chapter of this manual and in the data sheet, and determine the worst-case access situations. If contention on the data bus can occur, consider adding fast data buffers between the EPROM outputs and the system data bus. These data buffers will allow the EPROM outputs to be quickly isolated from the data bus at the end of an EPROM access cycle.

The worst-case timing situation typically involves two consecutive loads from different devices. In back-to-back loads from different devices, there must be sufficient time for the first device to get off the data bus before the second device tries to drive its data. A load followed by a store is not critical since the processor inserts a “dead cycle” in this sequence to allow the external device to fully relinquish the bus.

6.2.4 Illegal Memory Accesses

The external memory or I/O interface circuit can detect illegal memory accesses and prevent the processor from completing such accesses by asserting the -MEXC (Memory Exception) and -READY signals. (See Figure 4-2, Load with Exception Timing, and Figure 4-4, Store with Exception Timing.) The current bus access is invalidated by the assertion of this signal, and the processor ignores the value on the data bus in that cycle. An instruction-access or data-access exception trap is initiated in the processor, allowing the software to handle the illegal memory access.

The memory-exception mechanism can be used for protection, by preventing user-mode accesses to certain regions of the processor's address space. External logic can also be used to detect and signal out-of-range access attempts.

6.2.5 I/O Interfacing Example: Ethernet Device

As an example of an I/O device interface, consider the MB86960 Ethernet interface device, also known as the NICE™ chip, used on the SPARClite Evaluation Board. In the evaluation board implementation, a PAL and two data transceivers are used to handle the interface. A block diagram of the interface is shown in Figure 6-3.

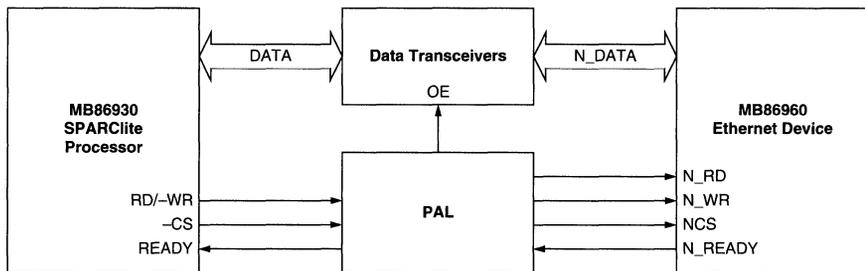


Figure 6-3. MB86960 Interface Block Diagram

The MB86960 NICE chip is completely asynchronous, has a non-deterministic access time, and has a long turn-off delay for the data pins. The PAL handles the synchronization of the control signals (Read, Write, Chip-Select, and Ready) between the processor and the NICE chip. The two data transceivers are used to

isolate the output pins from the data bus when a data access is complete. Figure 6-4 is a state diagram for the PAL.

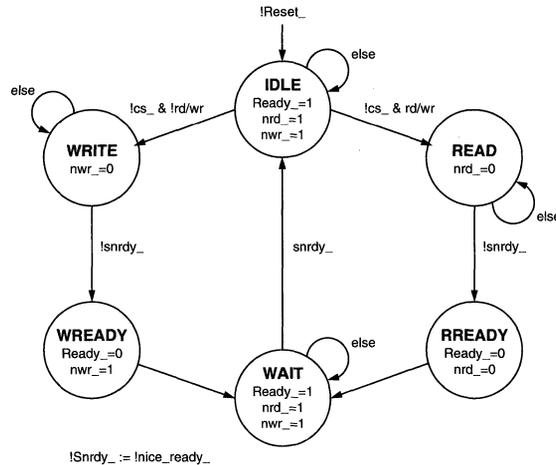


Figure 6-4. MB86960 Interface PAL State Diagram

Read and write operations are strobed by the assertion of the signals N_RD and N_WR (the read and write input pins of the NICE chip). To ensure that the address and the NICE chip Select signals are stable during strobing, the state machine waits one clock cycle before asserting N_RD or N_WR. When a transaction is finished, the NICE chip asserts its N_READY signal. Since N_READY is asynchronous, it is synchronized by a flip-flop in the PAL, producing a synchronized ready signal, which can then be used elsewhere inside the PAL and by the processor.

In a write operation, the synchronized Ready signal causes N_WR to be negated and the processor's -READY signal to be asserted. The data input setup and hold times of the NICE chip are based on the transition of the N_WR signal from asserted to negated; early negation ensures that there will be enough hold time because the processor won't stop driving the data bus until the next clock cycle.

In a read operation, the synchronized Ready signal causes the processor's -READY signal to be asserted, and on the next cycle, the -READY signal and N_RD are negated. Since data setup and hold times of the processor are based on the rising edge of the clock while -READY is asserted, enough hold time is ensured. The setup time requirement is ensured because there are almost two clock cycles between N_READY and the processor sampling the data.

In the case of back-to-back reads of the NICE chip, a new cycle can't start until N_READY is negated from the previous cycle.

The data transceivers are enabled by -CS asserted and -AS negated. Thus, during the uncertain period at the beginning of a bus cycle, the transceivers are not driving the data bus.

The byte order for the NICE chip (little-endian) is opposite that of the SPARClite processor (big-endian). The byte order is swapped in hardware: SPARClite data bits 8-15 connect to NICE bits 0-7, and SPARClite data bits 0-7 connect to NICE bits 8-15. The NICE chip can operate in both 8-bit and 16-bit modes.

6.3 DMA and Bus Arbitration

Some systems require support for multiple bus masters, such as for DMA (Direct Memory Access). An external device requests control of the bus by asserting the -BREQ (Bus Request) signal. External bus requests take precedence over internal requests. The processor, upon completing the current bus transaction, three-states its bus drivers and asserts -BGRNT (Bus Grant) to indicate that it is relinquishing control of the bus. The external device then takes control of the bus.

Upon completion of the DMA transfer or other bus operation, the external device de-asserts the -BREQ signal. The processor responds by de-asserting the -BGRNT signal and taking control of the bus, continuing with the next processor transaction.

The chip-select logic of the processor does not monitor the address bus and does not operate during the time that the bus is granted to another bus master. Therefore, an external address decoder should be used to generate the chip select signals for the external bus master. Also, the -CS outputs of the processor are held high (negated), but not three-stated, while the bus is granted to the external bus master. Therefore, for each memory device that is to be accessed by the external bus master, an OR gate must be provided at the chip select input to accept the signal from either the processor or the external address decoder. An alternative method is to not use the -CS signals from the processor at all, and to use the external address decoder all of the time (although the propagation delay for on-board chip selects is less).

A DMA operation that writes to system memory must be designed in such a manner that it will not modify cached data. Otherwise, the external memory data would no longer match the data stored in the processor's cache, resulting in errors. One way to meet this requirement is to locate the DMA-accessed memory in an address space that is not cached. The only address spaces that are cached are the User/Supervisor Instruction and Data spaces, corresponding to ASI (Address Space Identifier) values 0x8, 0x9, 0xA, and 0xB. Locating the DMA-accessible memory only in other address spaces (i.e., ASI values 0x10-0xFE) will ensure that no cached data will be modified.

Another way to handle this requirement is to use software to invalidate the data stored in cache when the external memory is modified. The software must keep track of what is cached and what is being modified. Each time a cached memory space is modified, the software invalidates the corresponding data stored in cache, in effect forcing an update to the cache whenever its contents are out-of-date.

Alternatively, embedded control task monitor software can be used to control the dynamic assignment of buffers between DMA inputs and outputs and processing inputs and outputs. The software can then ensure that no DMA transfers involve currently cached memory.

6.4 MB86940 Peripheral Chip

The MB86940 is an optional peripheral device that interfaces directly with the MB86930 SPARClite processor, and operates at the same clock speeds. It provides a variety of support features; a 15-level interrupt controller, a set of four counter/timers, and a set of two USARTs. With a MB86940 Peripheral Chip in the system, you can use any or all of these support features. The Peripheral Chip is a low-power CMOS device in either 120-pin PQFP or 135-pin CPGA packages.

A brief overview of the Peripheral Chip features is provided below. For detailed information on the chip functions, interfacing, and specifications, refer to the MB86940 User's Guide.

6.4.1 Interrupt Control

The interrupt controller on the Peripheral Chip has 15 separate interrupt-request inputs. The trigger conditions and active signal levels are individually programmable. The interrupt controller arbitrates the pending requests, and based on the SPARClite priority levels, issues an asynchronous interrupt to the processor. The interrupt is held pending until acknowledged by the processor.

The SPARClite processor has four interrupt inputs, (IRL3-IRL0). The value on these pins defines the level of the external interrupt. The value 0000 indicates no pending interrupt, while 1111 forces a non-maskable interrupt. Intermediate values indicate maskable interrupts with the corresponding priority levels.

6.4.2 Counter/Timers

The Peripheral Chip has four general-purpose 16-bit counter/timers. Each timer can be individually programmed to operate in any of several modes: time-out interrupt mode, rate generation mode, square wave generation mode, external-trigger one-shot mode, and software-trigger one-shot mode. Each timer can be reloaded at any time. Two prescalers are provided to optionally reduce the operating frequency of the timers.

6.4.3 USARTs

Two USART (Universal Synchronous/Asynchronous Receiver/Transmitter) channels are provided in the Peripheral Chip. The channels are individually programmable. Each channel is capable of sending and receiving serial data at rates up to 64K baud in synchronous mode and up to 19.2K baud in asynchronous mode. Data can be five to eight bits per character.

6.5 In-Circuit Emulation

SPARClite processors have ten pins used for in-circuit emulation: four emulator status/data bits, four emulator data bits, an emulator break request line, and an emulator enable pin. All of these pins should be left unconnected in the design for proper system operation.

To allow for compatibility with an in-circuit emulator, the system's reset circuit should be designed to allow the in-circuit emulator to take control of the $\overline{\text{RESET}}$ signal. For example, a jumper in the $\overline{\text{RESET}}$ input line close to the processor can be included, allowing the normal Reset circuit to be easily disconnected from the processor.

To simplify the task of emulating the processor especially for boards that do not socket the processor, it is recommended that the processor's emulator pins be connected to a standard format 20-pin connector. Access to these pins allow the emulator to take full control of the processor as well as to trace processor activity. If this socket is included on production boards, an emulator can be used for board diagnostics and maintenance later in the product life cycle. For more information contact Fujitsu Microelectronics' Advanced Products Division or your emulator vendor.

6.6 Physical Design Issues

Multiple VCC and VSS pins are provided on the SPARClite device for power and ground connections. The circuit board should be designed using separate power and ground planes for power distribution. Every VCC pin must be connected to the power plane, and every VSS pin must be connected to the ground plane. Any pins identified in the data sheet as "NC" must be left unconnected in the system.

To minimize the effects of spikes on output transitions, a generous amount of decoupling capacitance should be connected near the MB86930 device. It is important to use low-inductance capacitors and interconnections, especially in high-speed systems. Inductance can be minimized by making the board traces as short as possible between the processor and the decoupling capacitors.

For reliable operation, alternate bus masters must drive any signals that are three-stated by the processor when the processor grants control of the bus. Among the signals that must be driven are -LOCK , ADR31 through ADR2, ASI7 through ASI0, -BE3 through -BE0 , -AS , and RD/ -WR . These pins are normally driven by the processor during active and idle bus states, and don't require external pullups. D31 through D0 should be pulled up.

When designing the system, take into account the amount of load on the signal lines driven by the processor. The standard load is specified in the data sheet. If the actual load in the system is larger, the system may not be able to operate at the speeds specified in the data sheet timing diagrams, making it necessary to use a slower clock or to use buffers for the heavily loaded signals.

CHAPTER 7

.....

Instruction Set

This chapter presents the SPARC_{lite} processor instruction set. Sections discussing recommended assembly language syntax, a table of instructions listed by opcode, and an alphabetized instruction set reference are included.

7.1 Suggested Assembly Language Syntax

This section provides guidelines that describe the typical SPARC syntax accepted by most SPARC assemblers. It is intended to be a guide to help in understanding the code examples shown throughout this manual. Consult your assembler manual for a complete syntax description.

7.1.1 Register Names

reg A *reg* is an integer register name¹. It can have one of the following values:

<code>%r0 ... %r31</code>	
<code>%g0 ... %g7</code>	(<i>global</i> registers; same as <code>%r0 ... %r7</code>)
<code>%o0 ... %o7</code>	(<i>out</i> registers; same as <code>%r8 ... %r15</code>)
<code>%l0 ... %l7</code>	(<i>local</i> registers; same as <code>%r16 ... %r23</code>)
<code>%i0 ... %i7</code>	(<i>in</i> registers; same as <code>%r24 ... %r31</code>)
<code>%fp</code>	(frame pointer, conventionally same as <code>%i6</code>)
<code>%sp</code>	(stack pointer, conventionally same as <code>%o6</code>)

Subscripts further identify the placement of the operand in the binary instruction as one of the following:

<code>reg_{rs1}</code>	(<i>rs1</i> field)
<code>reg_{rs2}</code>	(<i>rs2</i> field)
<code>reg_{rd}</code>	(<i>rd</i> field)

asr_reg An *asr_reg* is an Ancillary State Register name². It can have one of the following values:

`%asr1 ... %asr31`

Subscripts further identify the placement of the operand in the binary instruction as one of the following:

<code>asr_reg_{rs1}</code>	(<i>rs1</i> field)
<code>asr_reg_{rd}</code>	(<i>rd</i> field)

7.1.2 Special Symbol Names

The symbol names and the registers or operators to which they refer are as follows:

<code>%psr</code>	Processor State Register
<code>%wim</code>	Window Invalid mask Register
<code>%tbr</code>	Trap Base Register
<code>%y</code>	Y register
<code>%hi</code>	Unary operator which extracts high 22 bits of its operand
<code>%lo</code>	Unary operator which extracts low 10 bits of its operand

1. In actual usage, the `%sp`, `%fp`, `%gn`, `%on`, `%ln` and `%in` forms are preferred over `%rn`

2. The MB86930 allows only `%asr17`.

7.1.3 Values

Some instructions use operands comprising values as follows:

<i>sim13</i>	A signed immediate constant that can be represented in 13 bits
<i>const22</i>	A constant that can be represented in 22 bits
<i>asi</i>	An alternate address space identifier (0 to 255)

7.1.4 Labels

A label is a sequence of characters comprised of alphabetic letters (a-z, A-Z {upper and lower case distinct}), underscores (_), dollar signs (\$), periods (.), and decimal digits (0-9). A label may contain decimal digits, but cannot begin with one.

7.1.5 Comments

Two types of comments are accepted by most SPARC assemblers: C-style “/*...*/” comments (which may span multiple lines), and “!...” comments, which extend from the “!” to the end of the line.

7.2 Syntax Design

The suggested SPARC assembly language syntax is designed so that:

- The destination operand (if any) is consistently specified as the last (right-most) operand in an assembly language statement.
- A reference to the **contents** of a memory location (in a Load, Store, or SWAP instruction is always indicated by square brackets ([]). A reference to the **address** of a memory location (such as in a JMPL, CALL, or SETHI) is specified directly, without square brackets.

7.3 Synthetic Instructions

Table 7-1 describes the mapping of a set of synthetic (or “pseudo”) instructions to actual SPARC instructions. These synthetic instructions may be provided in a SPARC assembler for the convenience of assembly language programmers.

Note that synthetic instructions should not be confused with “pseudo-ops”, which typically provide information to the assembler but do not generate instruc-

tions. Synthetic instructions always generate instructions; they provide more mnemonic syntax for standard SPARC instructions.

Table 7-1: Mapping of Synthetic Instructions to SPARC Instructions

Synthetic Instruction		SPARC Instruction(s)		Comment
cmp	reg_{rs1}, reg_{rs2}	subcc	$reg_{rs1}, reg_{rs2}, \%g0$	compare
cmp	$reg_{rs1}, simm13$	subcc	$reg_{rs1}, simm13, \%g0$	
jmp	$reg_{rs1} + reg_{rs2}$	jmp1	$reg_{rs1} + reg_{rs2}, \%g0$	
jmp	$reg_{rs1} +/- simm13$	jmp1	$reg_{rs1} +/- simm13, \%g0$	
call	$reg_{rs1} + reg_{rs2}$	jmp1	$reg_{rs1} + reg_{rs2}, \%o7$	
call	$reg_{rs1} +/- simm13$	jmp1	$reg_{rs1} +/- simm13, \%o7$	
tst	reg_{rs2}	orcc	$\%g0, reg_{rs2}, \%g0$	<i>test</i>
ret		jmp1	$\%i7+8, \%g0$	return from subroutine
retl		jmp1	$\%o7+8, \%g0$	return from leaf subroutine
restore		restore	$\%g0, \%g0, \%g0$	<i>trivial</i> restore
save		save	$\%g0, \%g0, \%g0$	<i>trivial</i> save (Warning: trivial save should only be used in kernel code!)
set	$value, reg_{rd}$	sethi	$\%hi(value), reg_{rd}$	(when $((value \& 0x1fff) == 0)$)
		or	$\%g0, value, reg_{rd}$	(when $-4096 \leq value \leq 4095$)
		sethi	$\%hi(value), reg_{rd}$	(otherwise)
		or	$reg_{rd}, \%lo(value), reg_{rd}$	<i>Warning: do not use set in the delay slot of a DCTI.</i>
not	reg_{rs1}, reg_{rd}	xnor	$reg_{rs1}, \%g0, reg_{rd}$	one's complement
not	reg_{rd}	xnor	$reg_{rd}, \%g0, reg_{rd}$	one's complement
neg	reg_{rs1}, reg_{rd}	sub	$\%g0, reg_{rs2}, reg_{rd}$	two's complement
neg	reg_{rd}	sub	$\%g0, reg_{rd}, reg_{rd}$	two's complement
inc	reg_{rd}	add	$reg_{rd}, 1, reg_{rd}$	increment by 1
inc	$simm13, reg_{rd}$	add	$reg_{rd}, simm13, reg_{rd}$	increment by const13
inccc	reg_{rd}	addcc	$reg_{rd}, 1, reg_{rd}$	increment by 1 and set icc
inccc	$simm13, reg_{rd}$	addcc	$reg_{rd}, simm13, reg_{rd}$	increment by const13 and set icc
dec	reg_{rd}	sub	$reg_{rd}, 1, reg_{rd}$	decrement by 1
dec	$simm13, reg_{rd}$	sub	$reg_{rd}, simm13, reg_{rd}$	decrement by const13
deccc	reg_{rd}	subcc	$reg_{rd}, 1, reg_{rd}$	decrement by 1 and set icc
deccc	$simm13, reg_{rd}$	subcc	$reg_{rd}, simm13, reg_{rd}$	decrement by const13 and set icc

Table 7-1: Mapping of Synthetic Instructions to SPARC Instructions

Synthetic Instruction		SPARC Instruction(s)		Comment
btst	$reg_{rs1} + reg_{rs2}$	andcc	$reg_{rs1} + reg_{rs2}, \%g0$	bit test
btst	$reg_{rs1} +/- simm13$	andcc	$reg_{rs1} +/- simm13, \%g0$	bit test
bset	$reg_{rs1} + reg_{rs2}$	or	$reg_{rs1} + reg_{rs2}, \%g0$	bit set
bset	$reg_{rs1} +/- simm13$	or	$reg_{rs1} +/- simm13, \%g0$	bit set
bclr	$reg_{rs1} + reg_{rs2}$	or	$reg_{rs1} +/- simm13, \%g0$	bit clear
bclr	$reg_{rs1} +/- simm13$	or	$reg_{rs1} +/- simm13, \%g0$	bit clear
btog	$reg_{rs1} + reg_{rs2}$	andn	$reg_{rs1} + reg_{rs2}, \%g0$	bit toggle
btog	$reg_{rs1} +/- simm13$	andn	$reg_{rs1} +/- simm13, \%g0$	bit toggle
		xor	$reg_{rs1} + reg_{rs2}, \%g0$	
		xor	$reg_{rs1} +/- simm13, \%g0$	
clr	reg_{rd}	or	$\%g0, \%g0, reg_{rd}$	clear (zero) register
clrb	$[reg_{rs1} + reg_{rs2}]$	stb	$\%g0, [reg_{rs1} + reg_{rs2}]$	clear byte
clrb	$[reg_{rs1} +/- simm13]$	stb	$\%g0, [reg_{rs1} +/- simm13]$	clear byte
clrh	$[reg_{rs1} + reg_{rs2}]$	sth	$\%g0, [reg_{rs1} + reg_{rs2}]$	clear halfword
clrh	$[reg_{rs1} +/- simm13]$	sth	$\%g0, [reg_{rs1} +/- simm13]$	clear halfword
clr	$[reg_{rs1} + reg_{rs2}]$	st	$\%g0, [reg_{rs1} + reg_{rs2}]$	clear word
clr	$[reg_{rs1} +/- simm13]$	st	$\%g0, [reg_{rs1} +/- simm13]$	clear word
mov	reg_{rs1}, reg_{rd}	or	$\%g0, reg_{rs1}, reg_{rd}$	
mov	$reg_{rs1} +/- simm13, reg_{rd}$	or	$\%g0, reg_{rs1} +/- simm13, reg_{rd}$	
mov	$\%y, reg_{rd}$	rd	$\%y, reg_{rd}$	
mov	$\%asrn, reg_{rd}$	rd	$\%asrn, reg_{rd}$	
mov	$\%psr, reg_{rd}$	rd	$\%psr, reg_{rd}$	
mov	$\%wim, reg_{rd}$	rd	$\%wim, reg_{rd}$	
mov	tbr, reg_{rd}	rd	tbr, reg_{rd}	
mov	$reg_{rs1}, \%y$	wr	$reg_{rs1}, \%y$	
mov	$simm13, \%y$	wr	$simm13, \%y$	
mov	$reg_{rs1}, \%asr_reg$	wr	$reg_{rs1}, \%asr_reg$	
mov	$simm13, \%asr_reg$	wr	$simm13, \%asr_reg$	
mov	$reg_{rs1}, \%psr$	wr	$reg_{rs1}, \%psr$	
mov	$simm13, \%psr$	wr	$simm13, \%psr$	
mov	$reg_{rs1}, \%wim$	wr	$reg_{rs1}, \%wim$	
mov	$simm13, \%wim$	wr	$simm13, \%wim$	
mov	$reg_{rs1}, \%tbr$	wr	$reg_{rs1}, \%tbr$	
mov	$simm13, \%tbr$	wr	$simm13, \%tbr$	

7.4 Binary Opcodes

The following table provides a mapping by binary opcode of the SPARC instructions mnemonics. In the table, the 32-bits that make up an instruction are divided into 4 fields. Field 1 for bits 31-30, field 2 for bits 24-19, field 3 for bits 29-25, and field 4 for bits 13-5. When using the table, look first for a match in field 1, then a match in field 2, followed by fields 3 and 4 until the desired mnemonic is found.

Table 7-2: SPARC Instructions Sorted by Opcode

Bits 31:30 Field 1	Bits 29...25 Field 3	Bits 24...19 Field 2	Bits 13...5 Field 4	Instruction Mnemonic
00	xxxxx	000xxx	xxxxxxxx x	UNIMP
00	x0000	010xxx	xxxxxxxx x	BN
00	x0001	010xxx	xxxxxxxx x	BE
00	x0010	010xxx	xxxxxxxx x	BLE
00	x0011	010xxx	xxxxxxxx x	BL
00	x0100	010xxx	xxxxxxxx x	BLEU
00	x0101	010xxx	xxxxxxxx x	BCS
00	x0110	010xxx	xxxxxxxx x	BNEG
00	x0111	010xxx	xxxxxxxx x	BVS
00	x1000	010xxx	xxxxxxxx x	BA
00	x1001	010xxx	xxxxxxxx x	BNE
00	x1010	010xxx	xxxxxxxx x	BG
00	x1011	010xxx	xxxxxxxx x	BGE
00	x1100	010xxx	xxxxxxxx x	BGU
00	x1101	010xxx	xxxxxxxx x	BCC
00	x1110	010xxx	xxxxxxxx x	BPOS

Table 7-2: SPARC Instructions Sorted by Opcode (Continued)

Bits 31:30 Field 1	Bits 29...25 Field 3	Bits 24...19 Field 2	Bits 13...5 Field 4	Instruction Mnemonic	
00	x1111	010xxx	xxxxxxxx x	BVC	
00	xxxxx	100xxx	xxxxxxxx x	SETHI	
00	00000	100xxx	xxxxxxxx x	NOP	
00	x0000	110xxx	xxxxxxxx x	FBN	†
00	x0001	110xxx	xxxxxxxx x	FBNE	†
00	x0010	110xxx	xxxxxxxx x	FBLG	†
00	x0011	110xxx	xxxxxxxx x	FBUL	†
00	x0100	110xxx	xxxxxxxx x	FBL	†
00	x0101	110xxx	xxxxxxxx x	FBUG	†
00	x0110	110xxx	xxxxxxxx x	FBG	†
00	x0111	110xxx	xxxxxxxx x	FBU	†
00	x1000	110xxx	xxxxxxxx x	FBA	†
00	x1001	110xxx	xxxxxxxx x	FBE	†
00	x1010	110xxx	xxxxxxxx x	FBUE	†
00	x1011	110xxx	xxxxxxxx x	FBGE	†
00	x1100	110xxx	xxxxxxxx x	FBUGE	†
00	x1101	110xxx	xxxxxxxx x	FBLE	†
00	x1110	110xxx	xxxxxxxx x	FBULE	†
00	x1111	110xxx	xxxxxxxx x	FBO	†
00	x0000	111xxx	xxxxxxxx x	CBN	†

Table 7-2: SPARC Instructions Sorted by Opcode (Continued)

Bits 31:30 Field 1	Bits 29...25 Field 3	Bits 24...19 Field 2	Bits 13...5 Field 4	Instruction Mnemonic	
00	x0001	111xxx	xxxxxxxx x	CB123	†
00	x0010	111xxx	xxxxxxxx x	CB12	†
00	x0011	111xxx	xxxxxxxx x	CB13	†
00	x0100	111xxx	xxxxxxxx x	CB1	†
00	x0101	111xxx	xxxxxxxx x	CB23	†
00	x0110	111xxx	xxxxxxxx x	CB2	†
00	x0111	111xxx	xxxxxxxx x	CB3	†
00	x1000	111xxx	xxxxxxxx x	CBA	†
00	x1001	111xxx	xxxxxxxx x	CB0	†
00	x1010	111xxx	xxxxxxxx x	CB03	†
00	x1011	111xxx	xxxxxxxx x	CB02	†
00	x1100	111xxx	xxxxxxxx x	CB023	†
00	x1101	111xxx	xxxxxxxx x	CB01	†
00	x1110	111xxx	xxxxxxxx x	CB013	†
00	x1111	111xxx	xxxxxxxx x	CB012	†
01	01xxx	xxxxxx	xxxxxxxx x	CALL	
10	xxxxx	000000	xxxxxxxx x	ADD	
10	xxxxx	000001	xxxxxxxx x	AND	
10	xxxxx	000010	xxxxxxxx x	OR	
10	xxxxx	000011	xxxxxxxx x	XOR	

Table 7-2: SPARC Instructions Sorted by Opcode (Continued)

Bits 31:30 Field 1	Bits 29...25 Field 3	Bits 24...19 Field 2	Bits 13...5 Field 4	Instruction Mnemonic	
10	xxxxxx	000100	xxxxxxxx x	SUB	
10	xxxxxx	000101	xxxxxxxx x	ANDN	
10	xxxxxx	000110	xxxxxxxx x	ORN	
10	xxxxxx	000111	xxxxxxxx x	xNOR	
10	xxxxxx	001000	xxxxxxxx x	ADDx	
10	xxxxxx	001010	xxxxxxxx x	UMUL	
10	xxxxxx	001011	xxxxxxxx x	SMUL	
10	xxxxxx	001100	xxxxxxxx x	SUBx	
10	xxxxxx	001110	xxxxxxxx x	UDIV	†
10	xxxxxx	001111	xxxxxxxx x	SDIV	†
10	xxxxxx	010000	xxxxxxxx x	ADDcc	
10	xxxxxx	010001	xxxxxxxx x	ANDcc	
10	xxxxxx	010010	xxxxxxxx x	ORcc	
10	xxxxxx	010011	xxxxxxxx x	XORcc	
10	xxxxxx	010100	xxxxxxxx x	SUBcc	
10	xxxxxx	010101	xxxxxxxx x	ANDNcc	
10	xxxxxx	010110	xxxxxxxx x	ORNcc	
10	xxxxxx	010111	xxxxxxxx x	xNORcc	
10	xxxxxx	011000	xxxxxxxx x	ADDxcc	
10	xxxxxx	011010	xxxxxxxx x	UMULcc	

Table 7-2: SPARC Instructions Sorted by Opcode (Continued)

Bits 31:30 Field 1	Bits 29...25 Field 3	Bits 24...19 Field 2	Bits 13...5 Field 4	Instruction Mnemonic	
10	xxxxx	011011	xxxxxxxx x	SMULcc	
10	xxxxx	011100	xxxxxxxx x	SUBxcc	
10	xxxxx	011101	xxxxxxxx x	DIVScc	
10	xxxxx	011110	xxxxxxxx x	UDIVcc	†
10	xxxxx	011111	xxxxxxxx x	SDIVcc	†
10	xxxxx	100000	xxxxxxxx x	TADDcc	
10	xxxxx	100001	xxxxxxxx x	TSUBcc	
10	xxxxx	100010	xxxxxxxx x	TADDccTV	
10	xxxxx	100011	xxxxxxxx x	TSUBccTV	
10	xxxxx	100100	xxxxxxxx x	MULScc	
10	xxxxx	100101	xxxxxxxx x	SLL	
10	xxxxx	100110	xxxxxxxx x	SRL	
10	xxxxx	100111	xxxxxxxx x	SRA	
10	00000	101000	xxxxxxxx x	STBAR	†
10	xxxxx	101000	xxxxxxxx x	RDASR (or RDY if rs1=0)	
10	xxxxx	101001	xxxxxxxx x	RDPSR	
10	xxxxx	101010	xxxxxxxx x	RDWIM	
10	xxxxx	101011	xxxxxxxx x	RDTBR	
10	xxxxx	101100	xxxxxxxx x	SCAN	
10	xxxxx	110000	xxxxxxxx x	WRASR	

Table 7-2: SPARC Instructions Sorted by Opcode (Continued)

Bits 31:30 Field 1	Bits 29...25 Field 3	Bits 24...19 Field 2	Bits 13...5 Field 4	Instruction Mnemonic	
10	00000	110000	xxxxxxxxx x	WRY	
10	xxxxx	110001	xxxxxxxxx x	WRPSR	
10	xxxxx	110010	xxxxxxxxx x	WRWIM	
10	xxxxx	110011	xxxxxxxxx x	WRTBR	
10	xxxxx	110100	01100011 1	FqTOs	†
10	xxxxx	110100	01100011 1	FdTOs	†
10	xxxxx	110100	01100010 0	FiTOs	†
10	xxxxx	110100	01100100 0	FiTOs	†
10	xxxxx	110100	00110100 1	FsMULd	†
10	xxxxx	110100	00100111 1	FDIVd	†
10	xxxxx	110100	01100100 1	FsTOD	†
10	xxxxx	110100	00110111 0	FsMULq	†
10	xxxxx	110100	01100110 0	FiTOq	†
10	xxxxx	110100	01101001 0	FdTOi	†
10	xxxxx	110100	01101001 1	FqTOi	†
10	xxxxx	110100	01101000 1	FsTOi	†
10	xxxxx	110100	01100111 0	FdTOq	†
10	xxxxx	110100	00100111 1	FDIVq	†
10	xxxxx	110100	01100110 1	FsTOq	†
10	xxxxx	110100	01100101 1	FqTOD	†

Table 7-2: SPARC Instructions Sorted by Opcode (Continued)

Bits 31:30 Field 1	Bits 29...25 Field 3	Bits 24...19 Field 2	Bits 13...5 Field 4	Instruction Mnemonic	
10	xxxxx	110100	00000000 1	FMOVs	†
10	xxxxx	110100	00100000 1	FADDs	†
10	xxxxx	110100	00100001 0	FADDd	†
10	xxxxx	110100	00100001 1	FADDq	†
10	xxxxx	110100	00010101 1	FSQRTq	†
10	xxxxx	110100	00010101 0	FSQRTd	†
10	xxxxx	110100	00100110 1	FDIVs	†
10	xxxxx	110100	00000100 1	FABSS	†
10	xxxxx	110100	00010100 1	FSQRTs	†
10	xxxxx	110100	00100010 1	FSUBs	†
10	xxxxx	110100	00000010 1	FNEGs	†
10	xxxxx	110100	00100101 0	FMULd	†
10	xxxxx	110100	00100101 1	FMULq	†
10	xxxxx	110100	00100011 0	FSUBd	†
10	xxxxx	110100	00100100 1	FMULs	†
10	xxxxx	110100	00100101 1	FMULd	†
10	xxxxx	110100	00100100 1	FMULq	†
10	xxxxx	110100	00100100 1	FMULs	†
10	xxxxx	110100	00100011 1	FSUBq	†
10	xxxxx	110101	00101011 1	FCMPEq	†

Table 7-2: SPARC Instructions Sorted by Opcode (Continued)

Bits 31:30 Field 1	Bits 29...25 Field 3	Bits 24...19 Field 2	Bits 13...5 Field 4	Instruction Mnemonic	
10	xxxxxx	110101	00101000 1	FCMPs	†
10	xxxxxx	110101	00101001 1	FCMPq	†
10	xxxxxx	110101	00101011 0	FCMPed	†
10	xxxxxx	110101	00101010 1	FCMPes	†
10	xxxxxx	110101	00101001 0	FCMPd	†
10	xxxxxx	110110	xxxxxxxx x	CPop1	†
10	xxxxxx	110111	xxxxxxxx x	CPop2	†
10	xxxxxx	111000	xxxxxxxx x	JMPL	
10	xxxxxx	111001	xxxxxxxx x	RETT	
10	x0000	111010	xxxxxxxx x	TN	
10	x0001	111010	xxxxxxxx x	TE	
10	x0010	111010	xxxxxxxx x	TLE	
10	x0011	111010	xxxxxxxx x	TL	
10	x0100	111010	xxxxxxxx x	TLEU	
10	x0101	111010	xxxxxxxx x	TCS	
10	x0110	111010	xxxxxxxx x	TNEG	
10	x0111	111010	xxxxxxxx x	TVS	
10	x1000	111010	xxxxxxxx x	TA	
10	x1001	111010	xxxxxxxx x	TNE	
10	x1010	111010	xxxxxxxx x	TG	

Table 7-2: SPARC Instructions Sorted by Opcode (Continued)

Bits 31:30 Field 1	Bits 29...25 Field 3	Bits 24...19 Field 2	Bits 13...5 Field 4	Instruction Mnemonic	
10	x1011	111010	xxxxxxx x	TGE	
10	x1100	111010	xxxxxxx x	TGU	
10	x1101	111010	xxxxxxx x	TCC	
10	x1110	111010	xxxxxxx x	TPOS	
10	x1111	111010	xxxxxxx x	TVC	
10	xxxxx	111011	xxxxxxx x	FLUSH	†
10	xxxxx	111100	xxxxxxx x	SAVE	
10	xxxxx	111101	xxxxxxx x	RESTORE	
11	xxxxx	000000	xxxxxxx x	LD	
11	xxxxx	000001	xxxxxxx x	LDUB	
11	xxxxx	000010	xxxxxxx x	LDUH	
11	xxxxx	000011	xxxxxxx x	LDD	
11	xxxxx	000100	xxxxxxx x	ST	
11	xxxxx	000101	xxxxxxx x	STB	
11	xxxxx	000110	xxxxxxx x	STH	
11	xxxxx	000111	xxxxxxx x	STD	
11	xxxxx	001001	xxxxxxx x	LDSB	
11	xxxxx	001010	xxxxxxx x	LDSH	
11	xxxxx	001101	xxxxxxx x	LDSTUB	
11	xxxxx	001111	xxxxxxx x	SWAP	

Table 7-2: SPARC Instructions Sorted by Opcode (Continued)

Bits 31:30 Field 1	Bits 29...25 Field 3	Bits 24...19 Field 2	Bits 13...5 Field 4	Instruction Mnemonic	
11	xxxxx	010000	xxxxxxxx x	LLDA	
11	xxxxx	010001	xxxxxxxx x	LDUBA	
11	xxxxx	010010	xxxxxxxx x	LDUHA	
11	xxxxx	010011	xxxxxxxx x	LDDA	
11	xxxxx	010100	xxxxxxxx x	STA	
11	xxxxx	010101	xxxxxxxx x	STBA	
11	xxxxx	010110	xxxxxxxx x	STHA	
11	xxxxx	010111	xxxxxxxx x	STDA	
11	xxxxx	011001	xxxxxxxx x	LDSBA	
11	xxxxx	011010	xxxxxxxx x	LDSHA	
11	xxxxx	011101	xxxxxxxx x	LDSTUBA	
11	xxxxx	011111	xxxxxxxx x	SWAPA	
11	xxxxx	100000	xxxxxxxx x	LDF	†
11	xxxxx	100001	xxxxxxxx x	LDFSR	†
11	xxxxx	100011	xxxxxxxx x	LDDF	†
11	xxxxx	100100	xxxxxxxx x	STF	†
11	xxxxx	100101	xxxxxxxx x	STFSR	†
11	xxxxx	100110	xxxxxxxx x	STDFQ	†
11	xxxxx	100111	xxxxxxxx x	STDF	†
11	xxxxx	110000	xxxxxxxx x	LDC	†

Table 7-2: SPARC Instructions Sorted by Opcode (Continued)

Bits 31:30 Field 1	Bits 29...25 Field 3	Bits 24...19 Field 2	Bits 13...5 Field 4	Instruction Mnemonic	
11	xxxxx	110001	xxxxxxxx x	LDCSR	†
11	xxxxx	110011	xxxxxxxx x	LDDC	†
11	xxxxx	110100	xxxxxxxx x	STC	†
11	xxxxx	110101	xxxxxxxx x	STCSR	†
11	xxxxx	110110	xxxxxxxx x	STDCQ	†
11	xxxxx	110111	xxxxxxxx x	STDC	†

†. These instructions are not implemented in hardware.

7.5 Instruction Set

This section provides a reference of all instructions supported in hardware on the SPARC*lite* MB86930. For additional information on the instructions refer to Chapter 2 "*Programmer's Model*" and to Chapter 5 "*Programming Considerations*" for code use examples.

ADD

ADD

Add

Description:

Computes either “ $r[rs1]+r[rs2]$ ” if the i field is zero, or “ $r[rs1] + \text{sign_ext}(\text{simm13})$ ” if the i field is one, and places the result in the destination specified by the rd field.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd		000000			rs1	i=0	unused (zero)			rs2	
31	30	29	25	24	19	18	14	13	12	0		
10	rd		000000			rs1	i=1	simm13				

Syntax:

```
add    regrs1, regrs2, regrd
add    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
mov    2, %l1
mov    4, %l2
add    %l1, %l2, %l3 ! %l3= 6
```

ADDcc**ADDcc****Add and modify icc****Description:**

Computes either “ $r[rs1]+r[rs2]$ ” if the *i* field is zero, or “ $r[rs1] + \text{sign_ext}(\text{simm13})$ ” if the *i* field is one, and places the result in the destination specified by the *rd* field.

ADDcc modifies the integer condition codes.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd		010000		rs1		i=0	unused (zero)			rs2	
31	30	29	25	24	19	18	14	13	12			
10	rd		010000		rs1		i=1	simm13				

Syntax:

```
addcc    regrs1, regrs2, regrd
addcc    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

n,z,v,c

Example:

```
mov      2, %l1
addcc    %l1, -5, %l3    ! %l3= -3
                        ! nzvc=1000
```

ADDX

ADDX

Add with carry

Description:

Computes either “ $r[rs1]+r[rs2]+c$ ” if the i field is zero, or “ $r[rs1] + \text{sign_ext}(\text{simm13})+c$ ” if the i field is one, and places the result in the destination specified by the rd field.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0	
10	rd		001000		rs1		i=0	unused (zero)			rs2		
31	30	29	25	24	19	18	14	13	12				
10	rd		001000		rs1		i=1	simm13					0

Syntax:

```

addx    regrs1, regrs2, regrd
addx    regrs1, immediate, regrd

```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```

mov     -1, %l1
addcc   %l1, %l1, %l2
addx    %g0, %g0, %l3 ! %l3= 1

```

ADDXcc**ADDXcc****Add with carry and modify icc****Description:**

Computes either “ $r[rs1]+r[rs2]+c$ ” if the i field is zero, or “ $r[rs1] + \text{sign_ext}(\text{simm13})+c$ ” if the i field is one, and places the result in the destination specified by the rd field.

ADDXcc modifies the integer condition codes.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd			011000			rs1	i=0	unused (zero)			rs2
31	30	29	25	24	19	18	14	13	12	0		
10	rd			011000			rs1	i=1	simm13			

Syntax:

```
addxcc    regrs1, regrs2, regrd
addxcc    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

n, z, v, c

Example:

```
mov      -1, %l1
mov      %l1, %l3
addcc    %l1,%l1,%l2    ! nzvc=1001
addxcc   %l3,0,%l3      ! %l3=0, nzvc=0101
```

AND

AND

And

Description:

Implements a bitwise logical And to compute either “r[rs1] and r[rs2]” if the *i* field is zero, or “r[rs1] and sign_ext(simmm13)” if the *i* field is one, and places the result in the destination specified by the *rd* field.

Format:

	31	30	29		25	24		19	18		14	13	12		5	4	0
	10			rd			000001			rs1		i=0		unused (zero)			rs2
	31	30	29		25	24		19	18		14	13	12				0
	10			rd			000001			rs1		i=1		simmm13			

Syntax:

```
and    regrs1, regrs2, regrd
and    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
mov    0x5, %l1
mov    0x3 %l2
and    %l1, %l2, %l3 ! %l3= 0x1
```

ANDcc**ANDcc****And and modify icc****Description:**

Implements a bitwise logical And to compute either “r[rs1] and r[rs2]” if the *i* field is zero, or “r[rs1] and sign_ext(simm13)” if the *i* field is one, and places the result in the destination specified by the *rd* field.

ANDcc modifies the integer condition codes.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd			010001			rs1	i=0	unused (zero)			rs2
31	30	29	25	24	19	18	14	13	12	0		
10	rd			010001			rs1	i=1	simm13			

Syntax:

```
andcc    regrs1, regrs2, regrd
andcc    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

n, z, v=0, c=0

Example:

```
mov      0x5, %l1
and      %l1, 0xa, %l3 ! %l3= 0x0, nzvc=0100
```

ANDN

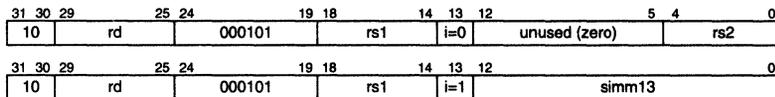
ANDN

And Not

Description:

Implements a bitwise logical And Not to compute either “ $r[rs1]$ andn $r[rs2]$ ” if the i field is zero, or “ $r[rs1]$ andn sign_ext(simm13)” if the i field is one, and places the result in the destination specified by the rd field.

Format:



Syntax:

```
andn    regrs1, regrs2, regrd
andn    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
mov     0x5, %11
mov     0x3 %12
andn    %11, %12, %13 ! %13= 0x4
```

ANDNcc**ANDNcc****And Not modify icc****Description:**

Implements a bitwise logical And Not to compute either “ $r[rs1]$ andn $r[rs2]$ ” if the i field is zero, or “ $r[rs1]$ andn sign_ext(simm13)” if the i field is one, and places the result in the destination specified by the rd field.

ANDNcc modifies the integer condition codes.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd			010101			rs1	i=0	unused (zero)			rs2
31	30	29	25	24	19	18	14	13	12	0		
10	rd			010101			rs1	i=1	simm13			

Syntax:

```
andncc    regrs1, regrs2, regrd
andncc    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

$n, z, v=0, c=0,$

Example:

```
mov       0x5, %l1
andncc   %l1, 0x3, %l3 ! %l3= 0x4, nzvc=0000
```

BA

BA

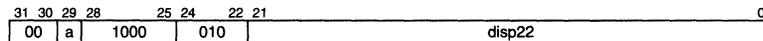
Branch Always

Description:

BA causes a PC-relative, delayed control transfer to the address “PC + (4 x sign_ext(dispatch2))”, regardless of the value of the condition code bits.

If the annul field of the branch instruction is 1, the delay instruction is annulled (not executed). If the annul field is 0, the delay instruction is executed. (Note: this is the reverse of the case for other conditional branches)

Format:



Syntax:

```
ba          label
ba,a       label          ! annul bit set
```

Traps:

(none)

Condition Code Modified:

(none)

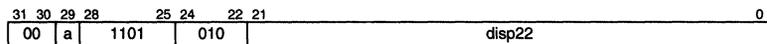
Example:

```
ba          xyz
mov        0x4, %l1      ! delay slot
```

BCC**BCC****Branch on Carry Clear (Branch Greater or Equal Unsigned)****Description:**

BCC causes a PC-relative, delayed control transfer to the address "PC + (4 x sign_ext(dispatch22))", if the carry (C) bit in the PSR is clear.

The annul bit only affects execution if the branch is not taken. With the annul (a) bit set, the delay instruction is annulled (not executed). With the annul (a) bit clear, the delay instruction is executed.

Format:**Syntax:**

```

bcc      label
bgeu    label          ! alternate mnemonic
bcc,a   label          ! annul bit set
bgeu,a  label

```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```

bcc,a   xyz
mov     0x4, %11          ! delay slot not executed if branch not taken

```

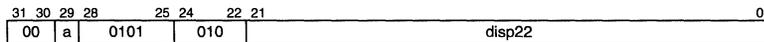
Branch on Carry Set (Branch on Less Than, Unsigned)

Description:

BCS causes a PC-relative, delayed control transfer to the address “PC + (4 x sign_ext(dispatch))”, if the carry (C) bit in the PSR is set.

The annul bit only affects execution if the branch is not taken. With the annul (a) bit set, the delay instruction is annulled (not executed). With the annul (a) bit clear, the delay instruction is executed.

Format:



Syntax:

```

bcs      label
blu      label           ! alternate mnemonic
bcs,a    label          ! annul bit set
blu,a    label
    
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

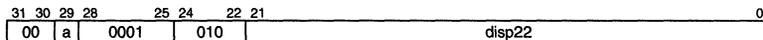
```

bcs      xyz
mov      0x4, %l1      ! delay slot
    
```

BE**BE****Branch on Equal (Branch on Zero)****Description:**

BE causes a PC-relative, delayed control transfer to the address "PC + (4 x sign_ext(dispatch22))", if Z is set.

The annul bit only affects execution if the branch is not taken. With the annul (a) bit set, the delay instruction is annulled (not executed). With the annul (a) bit clear, the delay instruction is executed.

Format:**Syntax:**

```

be          label
bz          label          ! alternate mnemonic
be,a       label          ! annul bit set
bz,a       label

```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```

bz          xyz
mov        0x4, %l1      ! delay slot

```

BG

BG

Branch on Greater

Description:

BG causes a PC-relative, delayed control transfer to the address "PC + (4 x sign_ext(dispatch22))", if "not(Z or (N xor V))" is true.

The annul bit only affects execution if the branch is not taken. With the annul (a) bit set, the delay instruction is annulled (not executed). With the annul (a) bit clear, the delay instruction is executed.

Format:



Syntax:

```

bg          label
bg, a      label          ! annul bit set
    
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

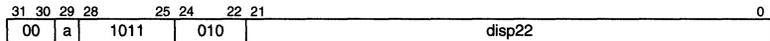
```

bg          xyz
mov        0x4, %l1          ! delay slot
    
```

BGE**BGE****Branch on Greater or Equal****Description:**

BGE causes a PC-relative, delayed control transfer to the address “PC + (4 x sign_ext(dispatch22))”, if “not(N xor V)” is true.

The annul bit only affects execution if the branch is not taken. With the annul (a) bit set, the delay instruction is annulled (not executed). With the annul (a) bit clear, the delay instruction is executed.

Format:**Syntax:**

```

bge      label
bge,a    label      ! annul bit set

```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```

bge      xyz
mov      0x4, %l1      ! delay slot

```

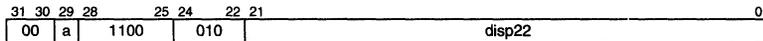
Branch on Greater, Unsigned

Description:

BGU causes a PC-relative, delayed control transfer to the address "PC + (4 x sign_ext(dispatch))", if "not(C or Z)" is true.

The annul bit only affects execution if the branch is not taken. With the annul (a) bit set, the delay instruction is annulled (not executed). With the annul (a) bit clear, the delay instruction is executed.

Format:



Syntax:

```

bgu      label
bgu, a   label      ! annul bit set
    
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

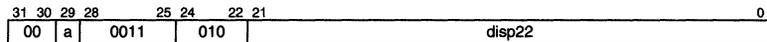
```

bgu      xyz
mov      0x4, %l1    ! delay slot
    
```

BL**BL****Branch on Less****Description:**

BL causes a PC-relative, delayed control transfer to the address "PC + (4 x sign_ext(dispatch22))", if "N xor V" is true.

The annul bit only affects execution if the branch is not taken. With the annul (a) bit set, the delay instruction is annulled (not executed). With the annul (a) bit clear, the delay instruction is executed.

Format:**Syntax:**

```
bl          label
bl,a       label      ! annul bit set
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
bl          xyz
mov        0x4, %l1      ! delay slot
```

BLE

BLE

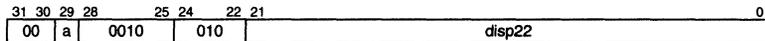
Branch on Less or Equal

Description:

BLE causes a PC-relative, delayed control transfer to the address "PC + (4 x sign_ext(dispatch22))", if "Z or (N xor V)" is true.

The annul bit only affects execution if the branch is not taken. With the annul (a) bit set, the delay instruction is annulled (not executed). With the annul (a) bit clear, the delay instruction is executed.

Format:



Syntax:

```
ble      label
ble,a    label      ! annul bit set
```

Traps:

(none)

Condition Code Modified:

(none)

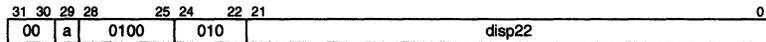
Example:

```
ble      xyz
mov      0x4, %l1      ! delay slot
```

BLEU**BLEU****Branch on Less or Equal, Unsigned****Description:**

BLEU causes a PC-relative, delayed control transfer to the address "PC + (4 x sign_ext(dispatch22))", if "C or Z" is true.

The annul bit only affects execution if the branch is not taken. With the annul (a) bit set, the delay instruction is annulled (not executed). With the annul (a) bit clear, the delay instruction is executed.

Format:**Syntax:**

```
bleu      label
bleu,a    label      ! annul bit set
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
bleu      xyz
mov       0x4, %l1      ! delay slot
```

BN

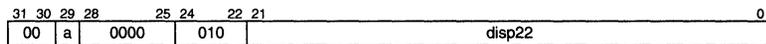
BN

Branch Never

Description:

BN acts like a “NOP” except that if the annul field is one, the delay instruction is not executed (annulled). If the annul (a) field is zero, the delay instruction is executed.

Format:



Syntax:

```
bn          label
bn,a       label      ! annul bit set
```

Traps:

(none)

Condition Code Modified:

(none)

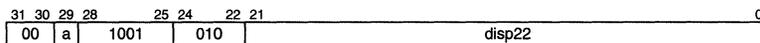
Example:

```
bn          xyz
mov        0x4, %l1      ! delay slot
```

BNE**BNE****Branch on Not Equal (Branch on Not Zero)****Description:**

BNE causes a PC-relative, delayed control transfer to the address "PC + (4 x sign_ext(dispatch22))", if Z is clear.

The annul bit only affects execution if the branch is not taken. With the annul (a) bit set, the delay instruction is annulled (not executed). With the annul (a) bit clear, the delay instruction is executed.

Format:**Syntax:**

```

bne      label
bnz      label           ! alternate mnemonic
bne,a    label           ! annul bit set
bnz,a    label

```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```

bnz      xyz
mov      0x4, %l1       ! delay slot

```

BNEG**BNEG****Branch on Negative****Description:**

BNEG causes a PC-relative, delayed control transfer to the address "PC + (4 x sign_ext(dispatch22))", if N is set.

The annul bit only affects execution if the branch is not taken. With the annul (a) bit set, the delay instruction is annulled (not executed). With the annul (a) bit clear, the delay instruction is executed.

Format:**Syntax:**

```

bneg      label
bneg, a   label      ! annul bit set

```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```

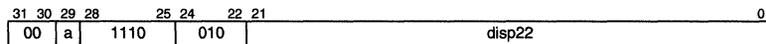
bneg      xyz
mov       0x4, %l1      ! delay slot

```

BPOS**BPOS****Branch on Positive****Description:**

BPOS causes a PC-relative, delayed control transfer to the address "PC + (4 x sign_ext(dispatch))", if N is clear.

The annul bit only affects execution if the branch is not taken. With the annul (a) bit set, the delay instruction is annulled (not executed). With the annul (a) bit clear, the delay instruction is executed.

Format:**Syntax:**

```
bpos      label
bpos,a    label      ! annul bit set
```

Traps:

(none)

Condition Code Modified:

(none)

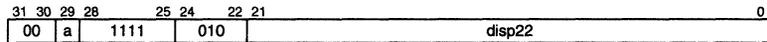
Example:

```
bpos      xyz
mov       0x4, %l1      ! delay slot
```

BVC**BVC****Branch on Overflow Clear****Description:**

BVC causes a PC-relative, delayed control transfer to the address "PC + (4 x sign_ext(dispatch))", if V is clear.

The annul bit only affects execution if the branch is not taken. With the annul (a) bit set, the delay instruction is annulled (not executed). With the annul (a) bit clear, the delay instruction is executed.

Format:**Syntax:**

```
bvc      label
bvc,a    label      ! annul bit set
```

Traps:

(none)

Condition Code Modified:

(none)

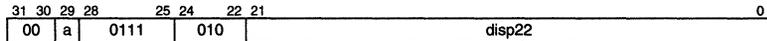
Example:

```
bvc      xyz
mov      0x4, %l1      ! delay slot
```

BVS**BVS****Branch on Overflow Set****Description:**

BVS causes a PC-relative, delayed control transfer to the address "PC + (4 x sign_ext(dispatch22))", if V is set.

The annul bit only affects execution if the branch is not taken. With the annul (a) bit set, the delay instruction is annulled (not executed). With the annul (a) bit clear, the delay instruction is executed.

Format:**Syntax:**

```
bvs          label
bvs,a       label      ! annul bit set
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
bvs          xyz
mov          0x4, %l1      ! delay slot
```

CALL

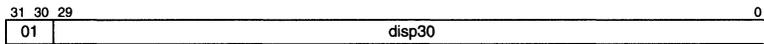
CALL

Call Instruction

Description:

The CALL instruction causes an unconditional, delayed, PC-relative control transfer to address "PC + (4 x disp30)". Since the word displacement field is 30 bits wide, the target address can be arbitrarily distant. The CALL instruction also writes the value of PC, which contains the address of the CALL, into %o7 (r[15]).

Format:



Syntax:

```
call    label
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
call    xyz
mov     0x4, %l1    ! delay slot
```

DIVSCC

DIVSCC

Divide Step

Description:

The DIVSCC instruction performs one bit-cycle of a non-restoring, shift-before-add, signed or unsigned division. Initially, the most significant half of the dividend is in the Y register, the least significant half is in $r[rs1]$. The divisor is in $r[rs2]$. Subsequently, the most significant half of the partial remainder is in the Y register, the least significant half is in $r[rs1]$.

DIVSCC operates as follows:

1. The *true sign* is formed using the negative (n) and overflow (v) integer condition codes from the Processor Status Register. $\text{True sign} = n \text{ XOR } v$.
2. The *remainder* is formed by upshifting the Y register (initially the most significant word of the dividend) one bit, and setting the least significant bit of remainder equal to most significant bit of $r[rs1]$ (initially the least significant word of the dividend).
3. The *divisor* is $r[rs2]$ if the *i* field is 0, or *simm13*, sign-extended to 32 bits, if the *i* field is 1.
4. If *true sign* = 0 (+), the ALU computes *remainder* - *divisor*. If *true sign* = 1 (-), the ALU computes *remainder* + *divisor*.
5. *Carry out* from the ALU operation is noted as *c0*. The negative (n) condition code is set to bit 31 of the ALU result. The zero (z) condition code is set if the ALU result is 0 AND the *true sign* equals $Y[31]$, else cleared.
6. The *new true sign* is formed as $(\text{true sign AND NOT } Y[31]) \text{ OR } (\text{NOT } c0 \text{ AND } (\text{true sign OR NOT } Y[31]))$.
7. The overflow (v) condition code is formed as *new true sign* XOR bit 31 of the ALU result. The carry (c) condition code is set to NOT *new true sign*. Y is set to the 32-bit ALU result. If *rd* is not 0, then $r[rd]$ is set to $r[rs1]$, upshifted one bit with NOT *new true sign* (the new quotient bit) in the least significant bit position.

Divide Step (Continued)

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10			rd	011101		rs1		i=0	reserved		rs2	
31	30	29	25	24	19	18	14	13	12			0
10			rd	011101		rs1		i=1	simm13			

Syntax:

```
divscc    regrs1, regrs2, regrd
divscc    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

n, z, v, c,

Example:

See Chapter 5 “Programming Considerations” for sample signed and unsigned division routines based on the DIVScc instruction as well as some application examples.

JMPL**JMPL****Jump and Link****Description:**

The JMPL instruction causes a register-indirect control transfer to an address specified by either “ $r[rs1] + r[rs2]$ ” if the *i* field is zero, or “ $r[rs1] + \text{sign_ext}(\text{simm13})$ ” if the *i* field is one.

The JMPL instruction writes the PC, which contains the address of the JMPL instruction, into the destination *r* register specified in *rd* field.

If either of the low-order two bits of the jump address is nonzero, a `mem_address_not_aligned` trap occurs.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0	
10	rd		111000			rs1	i=0	unused (zero)			rs2	0	
31	30	29	25	24	19	18	14	13	12				
10	rd		111000			rs1	i=1	simm13					0

Syntax:

```

jmpl      regrs1, regrs2, regrd
jmpl      regrs1, immediate, regrd

```

Traps:

`mem_address_not_aligned`

Condition Code Modified:

(none)

Jump and Link (Continued)

Example:

```
jmp1    %l2+0xf8, %g0
mov     0xfe, %l1    ! delay slot
```

notes:-JMPL with $rd=\%g0$ can be used to return from a subroutine.

- For a non-leaf subroutine the typical return address is “r[31]+8”, if the subroutine was entered by a call instruction. (Note: The pseudo operation “ret” invokes this return address). A leaf subroutine (no use of save, no call to other subroutines) can use “r[15]+8” as the return address. (Note: Pseudo operation “retl” invokes this return address).
- JMPL with $rd = 15$ can be used as a register-indirect CALL.
- When the delay slot instruction of JMPL is RETT, the target of the JMPL is the address space pointed to by the state of the machine after the RETT is executed (this is important when returning from a trap (which is supervisor space) to user address space).

LD

LD

Load Word

Description:

The LD instruction moves a word from memory into the r register defined by the *rd* field. The source value is loaded from either “r[rs1] + r[rs2]” if the *i* field is zero, or “r[rs1] + sign_ext(simm 13)” if the *i* field is one.

The address space identifier (ASI) indicates either user data (0xA) or supervisor data (0xB) according to the S bit of the PSR.

If the LD instruction traps, the destination register (*rd*) remains unchanged.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
11	rd		000000			rs1	i=0	unused (zero)			rs2	
31	30	29	25	24	19	18	14	13	12			0
11	rd		000000			rs1	i=1	simm13				

Syntax:

```
ld      [regrs1+ regrs2], regrd
ld      [regrs1 +/- immediate], regrd
```

Traps:

- mem_address_not_aligned
- data_access_exception

Condition Code Modified:

(none)

Example:

```
ld      [%g0 + 0xfe0], %l4
ld      [0xfe0], %l4           !recognized as equivalent
```

LDA

LDA

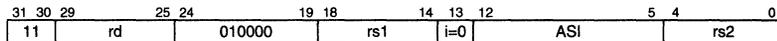
Load Word from Alternate Space

Description:

The LDA instruction moves a word from memory into the r register defined by the *rd* field. The source value is loaded from “r[*rs1*] + r[*rs2*]” with the ASI field designating the ASI value.

If the LDA instruction traps, the destination register (*rd*) remains unchanged. LDA is a privileged instruction which can only be executed in supervisor mode.

Format:



Syntax:

```
lda      [regrs1 + regrs2]ASI, regrd
```

Traps:

- mem_address_not_aligned
- data_access_exception
- privileged_instruction (if not supervisor mode)
- illegal_instruction (if i=1)

Condition Code Modified:

(none)

Example:

```
lda      [%l1 + %l2]0xf, %l4      ! ASI value 15 decimal
```

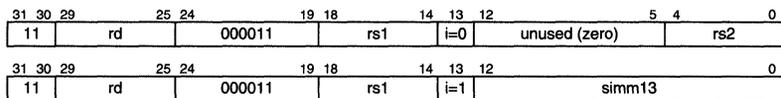
LDD**LDD****Load Doubleword****Description:**

The LDD instruction moves two words from memory into an r register pair. The most significant word at the effective memory address is moved into the even r register. The least significant word, which is at the effective memory address + 4, is moved into the odd r register. The least significant bit of the *rd* field is ignored.

The source value is loaded from either “r[*rs1*] + r[*rs2*]” if the *i* field is zero, or “r[*rs1*] + sign_ext(simmm 13)” if the *i* field is one.

The address space identifier (ASI) indicates either user data (0xA) or supervisor data (0xB) according to the S bit of the PSR.

If the LDD instruction traps while loading the second word the even destination register (*rd_{even}*) will have been changed.

Format:**Syntax:**

```
ldd      [regrs1+ regrs2], regrd
ldd      [regrs1 +/- immediate], regrd
```

Traps:

```
mem_address_not_aligned
data_access_exception
```

Condition Code Modified:

(none)

Example:

```
ldd      [%i5 + %l2], %g2
```

LDDA

LDDA

Load Doubleword from Alternate Space

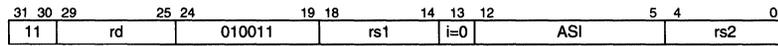
Description:

The LDDA instruction moves two words from memory into an r register pair. The most significant word at the effective memory address is moved into the even r register. The least significant word, which is at the effective memory address + 4, is moved into the odd r register. The least significant bit of the rd field is ignored.

The source value is loaded from “r[rs1] + r[rs2]” with the ASI field designating the ASI value.

If the LDD instruction traps while loading the second word the even destination register (rd_{even}) will have been changed.

Format:



Syntax:

```

ldda    [regrs1 + regrs2]ASI, regrd
ldda    [regrs1 +/- immediate]ASI, regrd

```

Traps:

- mem_address_not_aligned
- data_access_exception
- privileged_instruction (if not supervisor mode)
- illegal_instruction (if i=1)

Condition Code Modified:

(none)

Example:

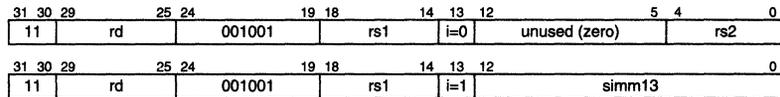
```
ldda    [%r7 - 5]0x1, %r4
```

LDSB**LDSB****Load Signed Byte****Description:**

The LDSB instruction moves a byte from memory into the *r* register defined by the *rd* field. The fetched byte is right-justified in *rd* and is sign-extended. The source value is loaded from either “*r*[*rs1*] + *r*[*rs2*]” if the *i* field is zero, or “*r*[*rs1*] + sign_ext(simm 13)” if the *i* field is one.

The address space identifier (ASI) indicates either user data (0xA) or supervisor data (0xB) according to the S bit of the PSR.

If the LD instruction traps, the destination register (*rd*) remains unchanged.

Format:**Syntax:**

```
ldsb    [regrs1 + regrs2], regrd
ldsb    [regrs1 +/- immediate], regrd
```

Traps:

data_access_exception

Condition Code Modified:

(none)

Example:

```
ldsb    [%g0 + 0xfe0], %l4
```

LDSBA

LDSBA

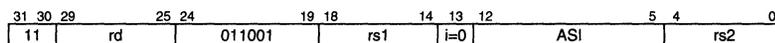
Load Signed Byte from Alternate Space

Description:

The LDSB instruction moves a byte from memory into the r register defined by the *rd* field. The fetched byte is right-justified in *rd* and is sign-extended. The source value is loaded from “r[*rs1*] + r[*rs2*]” with the ASI field designating the ASI value.

If the LDSBA instruction traps, the destination register (*rd*) remains unchanged. LDSBA is a privileged instruction which can only be executed in supervisor mode.

Format:



Syntax:

```
ldsba    [regrs1 + regrs2]ASI, regrd
```

Traps:

- data_access_exception
- privileged_instruction (if not supervisor mode)
- illegal_instruction (if i=1)

Condition Code Modified:

(none)

Example:

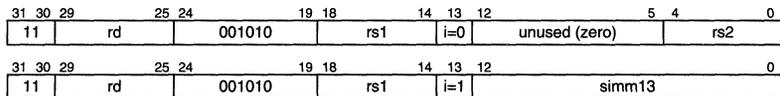
```
ldsba    [%11 + %12]0xf, %14    ! ASI value 15 decimal
```

LDSH**LDSH****Load Signed Halfword****Description:**

The LDSH instruction moves a halfword from memory into the *r* register defined by the *rd* field. The fetched halfword is right-justified in *rd* and is sign-extended. The source value is loaded from either “*r*[*rs1*] + *r*[*rs2*]” if the *i* field is zero, or “*r*[*rs1*] + sign_ext(simm 13)” if the *i* field is one.

The address space identifier (ASI) indicates either user data (0xA) or supervisor data (0xB) according to the S bit of the PSR.

If the LDSH instruction traps, the destination register (*rd*) remains unchanged.

Format:**Syntax:**

```
ldsh    [regrs1 + regrs2], regrd
ldsh    [regrs1 +/- immediate], regrd
```

Traps:

```
data_access_exception
mem_address_not_aligned
```

Condition Code Modified:

(none)

Example:

```
ldsh    [%g0 + 0xfe0], %l4
```

LDSHA

LDSHA

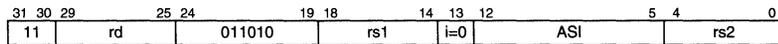
Load Signed Halfword from Alternate Space

Description:

The LDSH instruction moves a halfword from memory into the *r* register defined by the *rd* field. The fetched halfword is right-justified in *rd* and is sign-extended. The source value is loaded from “*r*[*rs1*] + *r*[*rs2*]” with the ASI field designating the ASI value.

If the LDSHA instruction traps, the destination register (*rd*) remains unchanged. LDSHA is a privileged instruction which can only be executed in supervisor mode.

Format:



Syntax:

```
ldsha    [regrs1 + regrs2]ASI, regrd
```

Traps:

- data_access_exception
- mem_address_not_aligned
- privileged_instruction (if not supervisor mode)
- illegal_instruction (if i=1)

Condition Code Modified:

(none)

Example:

```
ldsha    [%11 + %12]0xf, %14    ! ASI value 15 decimal
```

LDSTUB**LDSTUB****Atomic Load-Store Unsigned Byte****Description:**

The LDSTUB instruction moves a byte from memory into an r register identified by the *rd* field and then rewrites the same byte in memory to all ones atomically (without allowing intervening asynchronous traps). The value in the *rd* register is right justified and zero-filled.

The source value is loaded from either “*r[rs1] + r[rs2]*” if the *i* field is zero, or “*r[rs1] + sign_ext(simm 13)*” if the *i* field is one.

The address space identifier (ASI) indicates either user data (0xA) or supervisor data (0xB) according to the S bit of the PSR.

If the LDSTUB instruction traps, memory remains unchanged.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
11	rd		001101		rs1		i=0	unused (zero)			rs2	
31	30	29	25	24	19	18	14	13	12			
11	rd		001101		rs1		i=1	simm13				

Syntax:

```
ldstub    [regrs1 + regrs2], regrd
ldstub    [regrs1 +/- immediate], regrd
```

Traps:

data_access_exception

Condition Code Modified:

(none)

Example:

```
ldstub    [%g7 - 0xfb], %o1
```

LDSTUBA

LDSTUBA

Atomic Load-Store Unsigned Byte into Alternate Space

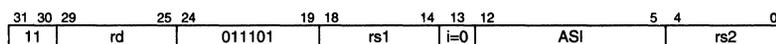
Description:

The LDSTUBA instruction moves a byte from memory into an r register identified by the *rd* field and then rewrites the same byte in memory to all ones atomically (without allowing intervening asynchronous traps). The value in the *rd* register is right justified and zero-filled.

The source value is loaded from “r[rs1] + r[rs2]” with the ASI field designating the ASI value.

If the LDSTUBA instruction traps, memory remains unchanged. LDSTUBA is a privileged instruction which can only be executed in supervisor mode.

Format:



Syntax:

```
ldstuba [regrs1 + regrs2]ASI, regrd
```

Traps:

- data_access_exception
- privileged_instruction (if not supervisor mode)
- illegal_instruction (if i=1)

Condition Code Modified:

(none)

Example:

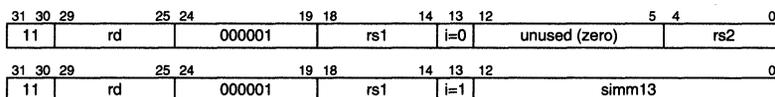
```
ldstuba [%11 + %12]0xf, %14 ! ASI value 15 decimal
```

LDUB**LDUB****Load Unsigned Byte****Description:**

The LDUB instruction moves an unsigned byte from memory into the *r* register defined by the *rd* field. The fetched halfword is right-justified in *rd* and is zero-filled. The source value is loaded from either "*r*[*rs1*] + *r*[*rs2*]" if the *i* field is zero, or "*r*[*rs1*] + sign_ext(simm 13)" if the *i* field is one.

The address space identifier (ASI) indicates either user data (0xA) or supervisor data (0xB) according to the S bit of the PSR.

If the LDUB instruction traps, the destination register (*rd*) remains unchanged.

Format:**Syntax:**

```
ldub    [regrs1 + regrs2], regrd
ldub    [regrs1 +/- immediate], regrd
```

Traps:

data_access_exception

Condition Code Modified:

(none)

Example:

```
ldub    [%g0 + 0xfe0], %l4
```

LDUBA

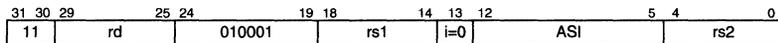
LDUBA

Load Unsigned Byte from Alternate Space

Description:

The LDUBA instruction moves a byte from memory into the r register defined by the *rd* field. The fetched byte is right-justified in *rd* and is zero-filled. The source value is loaded from “r[*rs1*] + r[*rs2*]” with the ASI field designating the ASI value. If the LDUBA instruction traps, the destination register (*rd*) remains unchanged. LDUBA is a privileged instruction which can only be executed in supervisor mode.

Format:



Syntax:

```
lduba    [regrs1 + regrs2]ASI, regrd
```

Traps:

- data_access_exception
- privileged_instruction (if not supervisor mode)
- illegal_instruction (if i=1)

Condition Code Modified:

(none)

Example:

```
lduba    [%11 + %12]0xf, %14    !ASI value 15 decimal
```

LDUH**LDUH****Load Unsigned Halfword****Description:**

The LDUH instruction moves a halfword from memory into the *r* register defined by the *rd* field. The fetched halfword is right-justified in *rd* and is zero-filled. The source value is loaded from either "*r*[*rs1*] + *r*[*rs2*]" if the *i* field is zero, or "*r*[*rs1*] + sign_ext(simm 13)" if the *i* field is one.

The address space identifier (ASI) indicates either user data (0xA) or supervisor data (0xB) according to the S bit of the PSR.

If the LDUH instruction traps, the destination register (*rd*) remains unchanged.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
11	rd			000010			rs1	i=0	unused (zero)			rs2
31	30	29	25	24	19	18	14	13	12	0		
11	rd			000010			rs1	i=1	simm13			

Syntax:

```
lduh      [regrs1 + regrs2, regrd
lduh      [regrs1 +/- immediate], regrd
```

Traps:

```
data_access_exception
mem_address_not_aligned
```

Condition Code Modified:

(none)

Example:

```
lduh      [%g7 - 0xfeb], %l4
```

LDUHA

LDUHA

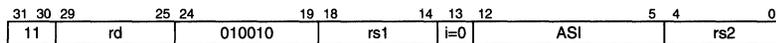
Load Unsigned Halfword from Alternate Space

Description:

The LDUHA instruction moves a halfword from memory into the *r* register defined by the *rd* field. The fetched halfword is right-justified in *rd* and is zero-filled. The source value is loaded from “*r*[*rs1*] + *r*[*rs2*]” with the ASI field designating the ASI value.

If the LDUHA instruction traps, the destination register (*rd*) remains unchanged. LDUHA is a privileged instruction which can only be executed in supervisor mode.

Format:



Syntax:

```
lduha    [regrs1 + regrs2]ASI, regrd
```

Traps:

- data_access_exception
- privileged_instruction (if not supervisor mode)
- illegal_instruction (if *i*=1)

Condition Code Modified:

(none)

Example:

```
lduha    [%g7 - 0xfeb]0xee, %l3
```

MULScc**MULScc****Multiply Step Instruction****Description:**

The MULScc can be used to generate up to 64-bit products of two signed or unsigned words. MULScc works as follows:

1. Compute the value obtained by shifting "r[rs1]" (the incoming partial product) right by one bit and replacing its high-order bit by "N xor V" (the sign of the previous partial product).
2. If the least significant bit of the Y register (the multiplier) is set, the value from step (1) is added to the multiplicand. The multiplicand is "r[rs2]" if the *i* field is zero or is "sign_ext(simmm13)" if the *i* field is one. If the LSB of the Y register is not set, then zero is added to the value from step (1).
3. The result from step (2) is written into "r[rd]" (the outgoing partial product). The PSR's integer condition codes are updated according to the addition performed in step (2).
4. The Y register (the multiplier) is shifted right by one bit and its high_order bit is replaced by the least significant bit of "r[rs1]" (the incoming partial product).

It should be noted that, for most applications, the UMUL/SMUL instructions are a faster and more efficient means of multiplying integer values. However MULScc can be used for other bit manipulations. See Chapter 5 "Programming Considerations" for details.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd			100100			rs1	i=0	reserved			rs2
31	30	29	25	24	19	18	14	13	12	0		
10	rd			100100			rs1	i=1	simmm13			

Syntax:

```

mulsc   regrs1, regrs2, regrd
mulsc   regrs1, immediate, regrd

```

Multiply Step Instruction (Continued)

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
mulsc  %04, %01, %04
```

NOP

NOP

No Operation

Description:

The NOP instruction changes no program-visible state (except the PC and nPC)

Format:**Syntax:**

nop

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
bz      target
nop                                !delay slot
```

OR

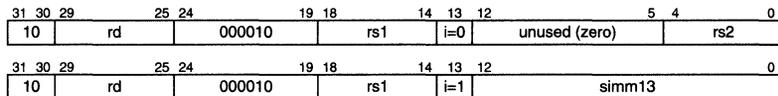
OR

Inclusive OR

Description:

Implements a bitwise logical inclusive Or to compute either “r[rs1] or r[rs2]” if the *i* field is zero, or “r[rs1] or sign_ext(simm13)” if the *i* field is one, and places the result in the destination specified by the *rd* field.

Format:



Syntax:

```
or      regrs1, regrs2, regrd
or      regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
or      %g0, -1, %o3 ! mov -1, %o3 equivalent
```

ORcc**ORcc****Inclusive OR and modify icc****Description:**

Implements a bitwise logical inclusive Or to compute either “r[rs1] or r[rs2]” if the *i* field is zero, or “r[rs1] or sign_ext(simm13)” if the *i* field is one, and places the result in the destination specified by the *rd* field.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10		rd		010010		rs1		i=0		unused (zero)		rs2
31	30	29	25	24	19	18	14	13	12			0
10		rd		010010		rs1		i=1		simm13		

Syntax:

```
orcc    regrs1, regrs2, regrd
orcc    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

n, z, v=0, c=0

Example:

```
mov     -1, %o3
orcc    %o3, 0, %g0    ! tst %o3 equivalent, nzvc=1000
```

ORN

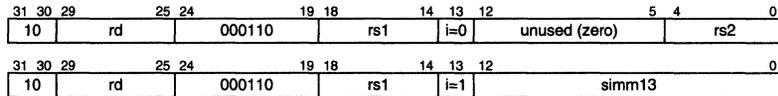
ORN

Inclusive Or Not

Description:

Implements a bitwise logical inclusive Or Not to compute either “ $r[rs1] \text{ or } r[rs2]$ ” if the i field is zero, or “ $r[rs1] \text{ or } \text{sign_ext}(\text{simm13})$ ” if the i field is one, and places the result in the destination specified by the rd field.

Format:



Syntax:

```

orn      regrs1, regrs2, regrd
orn      regrs1, immediate, regrd
    
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```

orn      %g0, 3, %o1      ! all 1's except bottom two bits to reg
                           o1
    
```

ORNcc**ORNcc****Inclusive Or Not and modify icc****Description:**

Implements a bitwise logical inclusive Or Not to compute either “ $r[rs1] \text{ orn } r[rs2]$ ” if the i field is zero, or “ $r[rs1] \text{ orn sign_ext(simm13)}$ ” if the i field is one, and places the result in the destination specified by the rd field.

Format:

31 30 29	25 24	19 18	14 13 12	5 4	0
10	rd	010110	rs1	i=0	unused (zero) rs2
31 30 29	25 24	19 18	14 13 12		
10	rd	010110	rs1	i=1	simm13

Syntax:

```
orncc    regrs1, regrs2, regrd
orncc    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

$n, z=0, v, c=0$

Example:

```
orncc    %g0, -1, %o3
```

RDASR

RDASR

Read Ancillary State Register

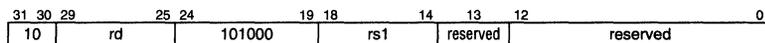
Description:

Reads the contents of the ancillary state register specified by the *rs1* field into the destination register *rd*.

On the SPARClite MB86930 a valid value for *rs1* is 17. All other values of *rs1* will generate an illegal instruction trap.

All reserved fields should be programmed as 0. RDASR is a privileged instruction.

Format:



Syntax:

```
rd          asr_regrs1, regrd
```

Traps:

```
illegal_instruction  
privileged_instruction
```

Condition Code Modified:

(none)

Example:

```
rd          %asr17, %g1
```

RDPSR**RDPSR****Read Processor State Register****Description:**

RDPSR reads the contents of the Processor State Register into the destination register *rd*.

All reserved fields should be programmed as 0. RDPSR is a privileged instruction.

Format:

31	30	29	25	24	19	18	14	13	12	0
10	rd			101001	reserved		reserved	reserved		

Syntax:

```
rd      %psr, regrd
```

Traps:

```
privileged_instruction
```

Condition Code Modified:

```
(none)
```

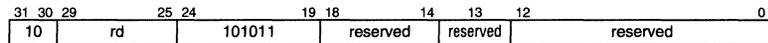
Example:

```
rd      %psr, %g1
```

RDTBR**RDTBR****Read Trap Base Register****Description:**

RDTBR reads the contents of the Trap Base Register into the destination register *rd*.

All reserved fields should be programmed as 0. RDTBR is a privileged instruction.

Format:**Syntax:**

```
rd    %tbr, regrd
```

Traps:

```
privileged_instruction
```

Condition Code Modified:

```
(none)
```

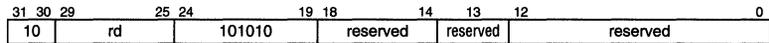
Example:

```
rd    %tbr, %g1
```

RDWIM**RDWIM****Read Window Invalid Mask Register****Description:**

RDWIM reads the contents of the Window Invalid Mask Register into the destination register *rd*.

All reserved fields should be programmed as 0. RDWIM is a privileged instruction.

Format:**Syntax:**

```
rd          %wim, regrd
```

Traps:

privileged_instruction

Condition Code Modified:

(none)

Example:

```
rd          %wim, %g0
```

RDY

RDY

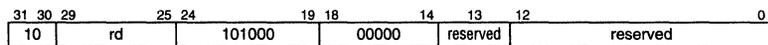
Read Y Register

Description:

RDY reads the contents of the Y register into the destination register *rd*.

Unlike the other read state register instructions, RDY is not privileged. All reserved fields should be programmed as 0.

Format:



Syntax:

`rd %y, regrd`

Traps:

(none)

Condition Code Modified:

(none)

Example:

`rd %y, %o0`

RESTORE

RESTORE

Restore Caller's Window

Description:

The RESTORE instruction adds one (modulo 8) to the Current Window Pointer (CWP) of the PSR and compares this value (new_CWP) against the Window Invalid Mask (WIM) register. If the WIM bit corresponding to the new_CWP is 0, the new_CWP is written into the CWP field of the PSR. This causes the CWP+1 window to become the current window, thereby restoring the caller's window. If the WIM bit corresponding to the new_CWP is 1, a window_underflow trap is generated and the CWP is left unchanged.

If an underflow trap is not generated, RESTORE behaves like an ADD instruction except that the source operands $r[rs1]$ and $r[rs2]$ are read from the old window and the sum is written into $r[rd]$ of the new window.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10		rd		111101		rs1		i=0		unused (zero)		rs2
31	30	29	25	24	19	18	14	13	12			0
10		rd		111101		rs1		i=1		simm13		

Syntax:

```
restore    regrs1, regrs2, regrd
restore    regrs1, immediate, regrd
```

Traps:

window_underflow

Condition Code Modified:

(none)

Example:

```
ret                ! return from non-leaf subroutine
restore    %i5, %l1, %o5 ! add number sampled processed with this call
                !   to running total kept in callee's reg i5
                !   and same register, caller's reg o5.
```

RETT

RETT

Return from Trap Instruction

Description:

If RETT does not cause a trap, it adds 1 to the CWP (modulo 8), causes a delayed control transfer to the target address, restores the S field of the PSR from the PS field, and sets the ET field of the PSR to 1. The target address is “r[rs1] + r[rs2]” if the *i* field is zero, or “r[rs1] + sign_ext(simm13)” if the *i* field is one.

RETT can cause one of several traps. In order of highest to lowest priority:

- If traps are enabled (ET=1) and the processor is in user mode (S=0), a `privileged_instruction` trap occurs.
- If traps are enabled (ET=1) and the processor is in supervisor mode (S=1), an `illegal_instruction` trap occurs.
- If traps are disabled (ET=0) and the processor is in user mode (S=0), `privileged_instruction` trap code is placed in *tt* (trap type) field of TBR and the processor enters `error_mode` state.
- If traps are disabled (ET=0) and a window underflow condition is detected, `window_underflow` trap is placed in *tt* (trap type) field of TBR and the processor enters `error_mode` state.
- If traps are disabled (ET=0) and either of the low-order two bits of the target address is nonzero, then `memory_address_not_aligned` code is placed in *tt* (trap type) field of TBR and the processor enters `error_mode` state.

The instruction executed immediately before an RETT must be a JMPL instruction.

RETT is a privileged instruction.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	reserved			111001			rs1	i=0	reserved			rs2
31	30	29	25	24	19	18	14	13	12	0		
10	reserved			111001			rs1	i=1	simm13			

Return from Trap Instruction (Continued)

Syntax:

```
rett      regrs1, regrs2
rett      regrs1, immediate
```

Traps:

```
privileged_instruction
illegal_instruction
window_underflow
mem_address_not_aligned
```

Condition Code Modified:

(none)

Example:

To re-execute the trapped instruction when returning from the trap handler use the sequence:

```
jmp1      %r17,%r0      !old PC
rett      %r18          !old nPC
```

To return to the instruction after the trapped instruction (for example, after emulating an instruction) use the sequence:

```
jmp1      %r18,%r0      !old nPC
rett      %r18+4        !old nPC + 4
```

SAVE

SAVE

Save Caller's Window

Description:

The SAVE instruction subtracts one (modulo 8) from the Current Window Pointer (CWP) of the PSR and compares this value (new_CWP) against the Window Invalid Mask (WIM) register. If the WIM bit corresponding to the new_CWP is 0, the new_CWP is written into the CWP field of the PSR. This causes the CWP -1 window to become the current window, thereby saving the caller's window. Otherwise a window_overflow trap is generated and the CWP is left unchanged.

If an overflow trap is not generated, SAVE behaves like an ADD instruction except that the source operands $r[rs1]$ and $r[rs2]$ are read from the old window and the sum is written into $r[rd]$ of the new window.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd			111100			rs1		i=0	unused (zero)		rs2
31	30	29	25	24	19	18	14	13	12	0		
10	rd			111100			rs1		i=1	simm13		

Syntax:

```
save    regrs1, regrs2, regrd
save    regrs1, immediate, regrd
```

Traps:

window_overflow

Condition Code Modified:

(none)

Example:

```
save    %sp, -64, %sp ! equivalent statements to make
save    %o6, -64, %o6 ! room for 16 more words in call stack
```

SCAN

SCAN

Scan for MSB

Description:

The scan instruction returns the location of the first nonsign bit or the location of either the most significant one or most significant zero of source register $r[rs1]$.

SCAN works as follows:

- (1) The $r[rs1]$ value is "xored" on a bit-wise basis with the value obtained by shifting right by one bit and sign extending the value in $r[rs2]$.
- (2) The bit position of the first "1" in the value obtained above is returned to the destination register $r[rd]$. A "1" in the MSB positions returns a value of 0, while the first "1" in the LSB position returns a value of 31. If no bit is set, a value of 63 is returned. For future compatibility, use unsigned compares of the SCAN value against unsigned thresholds. Use threshold equal WORDSIZE=32 to detect if no bit is set. See 5.6.1 "Scan in Support of Software Floating Point" and 5.6.2 "Scan in Support of Run Length Encoding" for illustration. (See Fig. 2-25, Using the SCAN Instruction) for additional details.

Format:

31 30 29	25 24	19 18	14 13 12	5 4	0
10	rd	101100	rs1	i=0	unused (zero) rs2
31 30 29	25 24	19 18	14 13 12	0	
10	rd	101100	rs1	i=1	simm13

Syntax:

```
scan    regrs1, regrs2, regrd
scan    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

(none)

Scan for MSB (Continued)

Example:

```
scan    %g1, 0, %g2      ! scan reg g1 for position of first one
                          ! from the msb end and put position
                          ! number in reg g2
scan    %g1, %g1, %g2    ! scan reg g1 for position of first bit
                          ! that differs from msb reg g1
```

SETHI**SETHI****Set High 22 bits****Description:**

SETHI zeroes the least significant 10 bits of the destination register ($r[rd]$), and replaces its high-order 22 bits with the value from the immediate field.

A SETHI instruction with $rd=0$ and $imm22=0$ is defined to be a NOP instruction.

Format:**Syntax:**

```
sethi    const22, regrd
sethi    %hi(value), regrd
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
sethi    %hi(label_trig_table), %l7
or       %l7, %lo(label_trig_table), %l7    ! address pointer of
                                           ! trig_table to %l7
```

SLL

SLL

Shift Left Logical

Description:

SLL shifts the value of $r[rs1]$ left by the count specified by the lower 5 bits of either " $r[rs2]$ " if the i field is zero, or " $imm13$ " if the i field is one. The vacated positions (least significant bits) are filled with zeroes. The shifted result is placed in the register specified by the rd field.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd			100101			rs1		i=0	unused (zero)		rs2
31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd			100101			rs1		i=1	unused (zero)		shcnt

Syntax:

```
sll      regrs1, regrs2, regrd
sll      regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
sll      %l1, %g1, %o1      ! left justify least significant part of
                             regl1
                             ! by shift count in reg g1
sub      %g0, %g1, %g1      ! negate reg g1
srl      %l1, %g1, %o0      ! right justify most significant part of reg
                             l1
                             ! by 32 - original shift count
or       %o0, %o1, %o0      ! join parts to complete left rotate by
                             ! original shift count
```

SMUL**SMUL****Signed Integer Multiply****Description:**

SMUL performs either “ $r[rs1] \times r[rs2]$ ” if the i field is zero, or “ $r[rs1] \times \text{sign_ext}(\text{simm13})$ ” if the i field is one. The 32 least significant bits of the product are written to the destination register $r[rd]$. The most significant bits of the product are written to the Y register.

The SMUL operation takes 5 cycles to compute a 32 bit x word operation, 3 cycles to compute a 32 bit x halfword operation, and 2 cycles to compute a 32 bit x byte operation. To do this, the hardware tests the most significant 16, 24 or 32 bits of $r[rs2]$ against the sign bit at run time. If the bits match, the SMUL instruction will terminate in 3, 2 or 1 cycle respectively.

SMUL assumes a signed integer word operand and computes a signed integer doubleword product.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd			001011			rs1		i=0	unused (zero)		rs2
31	30	29	25	24	19	18	14	13	12			0
10	rd			001011			rs1		i=1	simm13		

Syntax:

```
smul    regrs1, regrs2, regrd
smul    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
smul    %o2, %o3, %o1 ! least significant half product to %o1
rd      %y, %o0      ! most significant half product to %o0
```

Signed Integer Multiply and Change Condition Codes

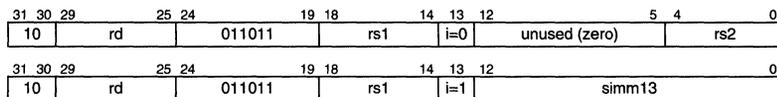
Description:

SMULcc performs either “ $r[rs1] \times r[rs2]$ ” if the i field is zero, or “ $r[rs1] \times \text{sign_ext}(\text{simm13})$ ” if the i field is one. The 32 least significant bits of the product are written to the destination register $r[rd]$. The most significant bits of the product are written to the Y register.

The SMUL operation takes 5 cycles to compute a 32 bit x word operation, 3 cycles to compute a 32 bit x halfword operation, and 2 cycles to compute a 32 bit x byte operation. To do this, the hardware tests the most significant 16, 24 or 32 bits of $r[rs2]$ against the sign bit at run time. If the bits match, the SMUL instruction will terminate in 3, 2 or 1 cycle respectively.

SMULcc assumes a signed integer word operand and computes a signed integer doubleword product. SMULcc writes the integer condition code (see below).

Format:



Syntax:

```
smulcc    regrs1, regrs2, regrd
smulcc    regrs1, immediate, regrd
```

Signed Integer Multiply and Change Condition Codes (Continued)

Traps:

(none)

Condition Code Modified:

Table 7-3:

icc bit	SMULcc
N	Set if product [31] = 1
Z	Set if product [31:0] = 0
V	Zero
C	Zero

Example:

```
smulcc    %o2, %o3, %o1 ! least significant half product to %o1
rd        %y, %o0      ! most significant half product to %o0
```

SRA

SRA

Shift Right Arithmetic

Description:

SRA shifts the value of $r[rs1]$ right by the count specified by the lower 5 bits of either “ $r[rs2]$ ” if the i field is zero, or “ $simm13$ ” if the i field is one. The vacated positions (most significant bits) are filled with the most significant bit of $r[rs1]$. The shifted result is placed in the r register specified by the rd field.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd		100111		rs1		i=0	unused (zero)			rs2	
31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd		100111		rs1		i=1	unused (zero)			shcnt	

Syntax:

```
sra    regrs1, regrs2, regrd
sra    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
sra    %g1, 4, %g1    ! right shift reg g1 4 bits and extend
                        sign
```

SRL**SRL****Shift Right Logical****Description:**

SRL shifts the value of $r[rs1]$ right by the count specified by the lower 5 bits of either " $r[rs2]$ " if the i field is zero, or "simm13" if the i field is one. The vacated positions (most significant bits) are filled with zeroes. The shifted result is placed in the r register specified by the rd field.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10			rd		100110		rs1	i=0		unused (zero)		rs2
31	30	29	25	24	19	18	14	13	12	5	4	0
10			rd		100110		rs1	i=1		unused (zero)		shcnt

Syntax:

```
srl      regrs1, regrs2, regrd
srl      regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
sll      %l1, %g1, %o1      ! left justify least significant part of
                             regl1
                             ! by shift count in reg g1
sub      %g0, %g1, %g1      ! negate reg g1
srl      %l1, %g1, %o0      ! right justify most significant part of reg
                             l1
                             ! by 32 - original shift count
or       %o0, %o1, %o0      ! join parts to complete left rotate by
                             ! original shift count
```

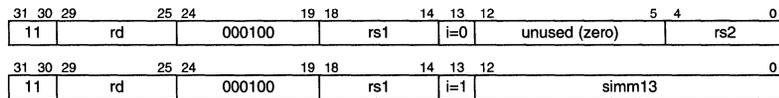
Store Word

Description:

The ST instruction moves a word from the r register specified by the *rd* field into memory. The effective memory address is either “*r[rs1] + r[rs2]*” if the *i* field is zero, or “*r[rs1] + sign_ext(simm13)*” if the *i* field is one. If the ST instruction traps, memory remains unchanged.

The address space identifier (ASI) indicates either user data (0xA) or supervisor data (0xB) according to the S bit of the PSR.

Format:



Syntax:

```

st      regrd, [regrs1 + regrs2]
st      regrd, [regrs1 +/- immediate]
  
```

Traps:

```

mem_address_not_aligned
data_access_exception
  
```

Condition Code Modified:

(none)

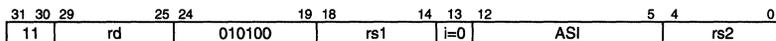
Example:

```

st      %l4, [%g0 + 0xfe0]
st      %l4, [0xfe0] ! recognized as equivalent
  
```

STA**STA****Store Word in Alternate Space****Description:**

The STA instruction moves a word from the *r* register specified by the *rd* field into memory. The source value is stored to "*r[rs1] + r[rs2]*" with the ASI field designating the ASI value. If the STA instruction traps, memory remains unchanged. STA is privileged and may only be executed in supervisor mode.

Format:**Syntax:**

```
sta      regrd, [regrs1 + regrs2]ASI
```

Traps:

- mem_address_not_aligned
- data_access_exception
- illegal_instruction (if i=1)
- privileged_instruction (if not supervisor mode)

Condition Code Modified:

(none)

Example:

```
sta      %l4, [%l1 + %l2]0xf          ! ASI value 15 decimal
```

Store Byte

Description:

The STB instruction moves the least significant byte from the r register specified by the *rd* field into memory. The effective memory address is either “*r[rs1] + r[rs2]*” if the *i* field is zero, or “*r[rs1] + sign_ext(simmm13)*” if the *i* field is one. If the STB instruction traps, memory remains unchanged.

The address space identifier (ASI) indicates either user data (0xA) or supervisor data (0xB) according to the S bit of the PSR.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
11	rd		000101		rs1		i=0	unused (zero)			rs2	
31	30	29	25	24	19	18	14	13	12			
11	rd		000101		rs1		i=1	simmm13				

Syntax:

```

stb      regrd, [regrs1 + regrs2]
stb      regrd, [regrs1 +/- immediate]

```

Traps:

data_access_exception

Condition Code Modified:

(none)

Example:

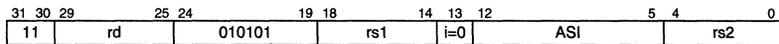
```

stb      %g2, [%i5 + %i2]

```

STBA**STBA****Store Byte in Alternate Space****Description:**

The STBA instruction moves the least significant byte from the r register specified by the *rd* field into memory. The source value is stored to “r[rs1] + r[rs2]” with the ASI field designating the ASI value. If the STBA instruction traps, memory remains unchanged. STBA is privileged and may only be executed in supervisor mode.

Format:**Syntax:**

```
stba    regrd, [regrs1+ regrs2]ASI
```

Traps:

- data_access_exception
- illegal_instruction (if i=1)
- privileged_instruction (if not supervisor mode)

Condition Code Modified:

(none)

Example:

```
stba    %o4, [%g7 - 5]0x1
```

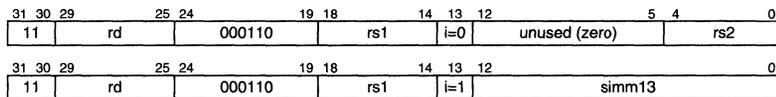
Store Halfword

Description:

The STH instruction moves the least significant halfword from the *r* register specified by the *rd* field into memory. The effective memory address is either “*r*[*rs1*] + *r*[*rs2*]” if the *i* field is zero, or “*r*[*rs1*] + sign_ext(simm13)” if the *i* field is one. If the STH instruction traps, memory remains unchanged.

The address space identifier (ASI) indicates either user data (0xA) or supervisor data (0xB) according to the *S* bit of the PSR.

Format:



Syntax:

```
sth      regrd, [regrs1 + regrs2]
sth      regrd, [regrs1 +/- immediate]
```

Traps:

```
data_access_exception
mem_address_not_aligned
```

Condition Code Modified:

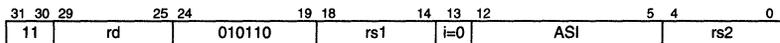
(none)

Example:

```
sth      %14, [%g0 + 0xfe0]
```

STHA**STHA****Store Halfword in Alternate Space****Description:**

The STHA instruction moves the least significant byte from the r register specified by the *rd* field into memory. The source value is stored to “r[rs1] + r[rs2]” with the ASI field designating the ASI value. If the STHA instruction traps, memory remains unchanged. STHA is privileged and may only be executed in supervisor mode.

Format:**Syntax:**

```
stha      regrd, [regrs1+ regrs2]ASI
```

Traps:

- data_access_exception
- illegal_instruction (if i=1)
- mem_address_not_aligned
- privileged_instruction (if not supervisor mode)

Condition Code Modified:

(none)

Example:

```
stha      %i4, [%i2 + %i3]0x3
```

Store Doubleword into Alternate space

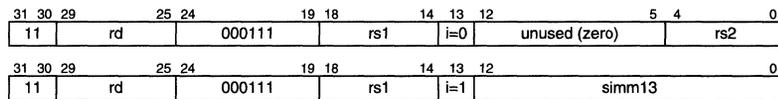
Description:

The STD instruction moves a doubleword from an even/next-odd r register pair into memory. The even r register (which contains the most significant word) is written into memory at the effective address and the odd r register (with the least significant word) is written into memory at the effective address + 4. The effective memory address is either “r[rs1] + r[rs2]” if the *i* field is zero, or “r[rs1] + sign_ext(simm13)” if the *i* field is one.

The address space identifier (ASI) indicates either user data (0xA) or supervisor data (0xB) according to the S bit of the PSR.

If the STD instruction traps while writing the first word to memory, memory remains unchanged. If the STD instruction traps while the second word is being written, the first word written (the most significant word at the highest address) will have been changed.

Format:



Syntax:

```
std      regrd, [regrs1 + regrs2]
std      regrd, [regrs1 +/- immediate]
```

Traps:

```
data_access_exception
mem_address_not_aligned
```

Condition Code Modified:

(none)

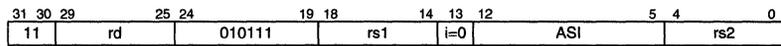
Example:

```
std      %o2, [%l3 - 4]
```

STDA**STDA****Store Doubleword in Alternate Space****Description:**

The STDA instruction moves a doubleword from an even/next-odd r register pair into memory. The even r register (which contains the most significant word) is written into memory at the effective address and the odd r register (with the least significant word) is written into memory at the effective address + 4. The source value is stored to “r[rs1] + r[rs2]” with the ASI field designating the ASI value. STDA is privileged and may only be executed in supervisor mode.

If the STD instruction traps while writing the first word to memory, memory remains unchanged. If the STD instruction traps while the second word is being written, the first word written (the most significant word at the highest address) will have been changed.

Format:**Syntax:**

```
stda      regrd, [regrs1+ regrs2]ASI
```

Traps:

- data_access_exception
- illegal_instruction (if i=1)
- mem_address_not_aligned
- privileged_instruction (if not supervisor mode)

Condition Code Modified:

(none)

Example:

```
stda      %i4, [%i2 + %i3]0x3
```

SUB

SUB

Subtract

Description:

Computes either “ $r[rs1]-r[rs2]$ ” if the i field is zero, or “ $r[rs1] - \text{sign_ext}(\text{simm13})$ ” if the i field is one, and places the result in the destination specified by the rd field.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd		000100		rs1		i=0	unused (zero)			rs2	
31	30	29	25	24	19	18	14	13	12	0		
10	rd		000100		rs1		i=1	simm13				

Syntax:

```
sub    regrs1, regrs2, regrd
sub    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
mov    4, %11
mov    2, %12
sub    %11, %12, %13 ! %13= 2
```

SUBcc**SUBcc****Subtract and modify icc****Description:**

Computes either “ $r[rs1]-r[rs2]$ ” if the i field is zero, or “ $r[rs1] - \text{sign_ext}(\text{simm13})$ ” if the i field is one, and places the result in the destination specified by the rd field.

SUBcc modifies the integer condition codes. Overflow occurs on subtraction if the operands have different signs and the sign of the difference differs from the sign of $r[rs1]$.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd			010100			rs1	i=0	unused (zero)			rs2
31	30	29	25	24	19	18	14	13	12	0		
10	rd			010100			rs1	i=1	simm13			

Syntax:

```
subcc    regrs1, regrs2, regrd
subcc    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

n, z, v, c

Example:

```
mov      4, %l1
subcc    %l1, 0x2, %l3 ! %l3= 2
                                ! nzvc = 0000
subcc    %l1, 0x7, %l4 ! %l4 = -3
                                ! nzvc = 1001
```

SUBX

SUBX

Subtract with Carry

Description:

Computes either “ $r[rs1]-r[rs2]-c$ ” if the i field is zero, or “ $r[rs1] - \text{sign_ext}(\text{simm13})-c$ ” if the i field is one, and places the result in the destination specified by the rd field.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10		rd		001100		rs1		i=0		unused (zero)		rs2
31	30	29	25	24	19	18	14	13	12			0
10		rd		001100		rs1		i=1		simm13		

Syntax:

```
subx    regrs1, regrs2, regrd
subx    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
subcc   %g0, 255, %g3 ! reg g3 = -255, nzvc = 1001
subx    %g0, 0, %g2  ! reg g2 = -1, sign extended
```

SUBXcc

SUBXcc

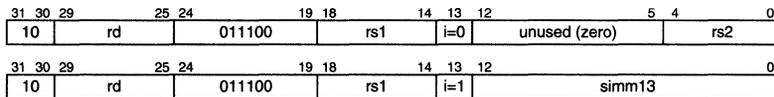
Subtract and modify icc

Description:

Computes either “ $r[rs1]-r[rs2]-c$ ” if the i field is zero, or “ $r[rs1] - \text{sign_ext}(\text{simm13})-c$ ” if the i field is one, and places the result in the destination specified by the rd field.

SUBXcc modifies the integer condition codes. Overflow occurs on subtraction if the operands have different signs and the sign of the difference differs from the sign of $r[rs1]$.

Format:



Syntax:

```
subxcc    regrs1, regrs2, regrd
subxcc    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

n, z, v, c

Example:

```
mov       -1, %l1      ! reg l1 = 0xffffffff
srl       %l1, 1, %l2  ! reg l2 = 0x7fffffff
orcc      %g0, 0, %g0  ! nzvc = 0100
subxcc    %l2, %l1, %g1 ! reg g1 = 0x80000000, nzvc = 1011
subxcc    %l2, %l1, %g2 ! reg g2 = 0x7fffffff, nzvc = 0001
```

SWAP

SWAP

SWAP Register with Memory

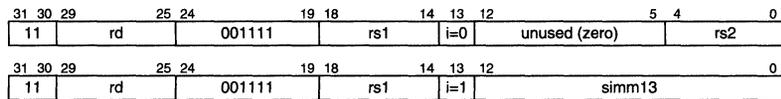
Description:

The SWAP instruction exchanges the contents of the r register identified by the *rd* field with the contents of the addressed memory location. This is performed atomically without allowing intervening asynchronous traps.

The effective address of the swap instruction is either “ $r[rs1] + r[rs2]$ ” if the *i* field is zero, or “ $r[rs1] + \text{sign_ext}(\text{simm13})$ ” if the *i* field is one.

If the SWAP instruction traps, memory remains unchanged.

Format:



Syntax:

```
swap    [regrs1 + regrs2], regrd
swap    [regrs1 + immediate], regrd
```

Traps:

```
data_access_exception
mem_address_not_aligned
```

Condition Code Modified:

(none)

Example:

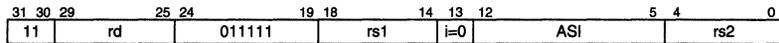
```
swap    [%g7-23], %g6
```

SWAPA**SWAPA****SWAP Register with Alternate Space Memory****Description:**

The SWAPA instruction exchanges the *r* register identified by the *rd* field with the contents of the addressed memory location. This is performed atomically without allowing intervening asynchronous traps.

The effective address of the swap instruction is "*r*[*rs1*] + *r*[*rs2*]" with the ASI field designating the ASI value.

If the SWAPA instruction traps, memory remains unchanged. SWAPA is privileged and may only be executed in supervisor mode.

Format:**Syntax:**

```
swapa      [regrs1 + regrs2] ASI, regrd
```

Traps:

- data_access_exception
- illegal_instruction (if *i*=1)
- mem_address_not_aligned
- privileged_instruction (if not supervisor mode)

Condition Code Modified:

(none)

Example:

```
swapa      [%15 + 125]oxf, %14
```

Trap Always (Trap on Zero)

Description:

The TA instruction generates a trap_instruction trap if no higher priority traps are pending. The trap_instruction trap causes the *tt* field of the Trap Base Register (TBR) to be written with 128 plus the least significant seven bits of either “r[rs1] + r[rs2]” if the *i* field is zero, or “r[rs1] + sign_ext(simm13)” if the *i* field is one.

All bits indicated as reserved in the instruction formats should be supplied as zero as should the most significant 25 bits of r[rs2] if the *i* field is 0.

(note: If single vector trapping is enabled, the trap_instruction trap will vector to the location pointed to by the Trap Base Address in the TBR, and the *tt* field will be ignored)

Format:

31	30	29	28	25	24	19	18	14	13	12	5	4	0
10	reserved	1000	111010	rs1	i=0	reserved	rs2						
31	30	29	28	25	24	19	18	14	13	12	7	6	0
10	reserved	1000	111010	rs1	i=1	reserved	software trap #						

Syntax:

```
ta      regrs1, regrs2
ta      regrs1, immediate
```

Traps:

```
trap_instruction
```

Condition Code Modified:

(none)

Example:

```
ta      %g0+35      ! tt=163
```

TADDcc**TADDcc****Tagged Add and modify icc****Description:**

The TADDcc instruction computes either “ $r[rs1] + r[rs2]$ ” if the *i* field is zero, or “ $r[rs1] + \text{sign_ext}(\text{simm13})$ ” if the *i* field is one. An overflow condition exists if bit 1 or 0 of either operand is not zero, or if the addition generates an arithmetic overflow.

If TADDcc causes an overflow condition, the overflow bit (*v*) of the PSR is set; if it does not cause an overflow, the overflow bit is cleared. In either case, the remaining integer condition codes are also updated and the result of the addition is written into the *r* register specified by the *rd* field.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd		100000			rs1	i=0	unused (zero)			rs2	
31	30	29	25	24	19	18	14	13	12			0
10	rd		100000			rs1	i=1	simm13				

Syntax:

```
taddcctv  regrs1, regrs2, regrd
taddcctv  regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

n, z, v, c

Example:

```
taddcc    %g0, 1, %g0    ! nzvc = 0010
```

Tagged Add and modify icc and Trap on Overflow

Description:

The TADDccTV instruction computes either “ $r[rs1] + r[rs2]$ ” if the *i* field is zero, or “ $r[rs1] + \text{sign_ext}(\text{simm13})$ ” if the *i* field is one. An overflow condition exists if bit 1 or 0 of either operand is not zero, or if the addition generates an arithmetic overflow.

If TADDccTV causes an overflow condition, a tag_overflow trap is generated and the destination register and condition codes remain unchanged. If TADDccTV does not cause an overflow condition, all the integer condition codes are updated (in particular, the overflow bit (*v*) is set to 0) and the result of the addition is written into the *r* register specified by the *rd* field.

Format:

31	30	29		25	24		19	18		14	13	12		5	4		0	
10			rd			100010			rs1		i=0			unused (zero)			rs2	
31	30	29		25	24		19	18		14	13	12						0
10			rd			100010			rs1		i=1			simm13				

Syntax:

```
taddcctv  regrs1, regrs2, regrd
taddcctv  regrs1, immediate, regrd
```

Traps:

```
tag_overflow
```

Condition Code Modified:

```
n, z, v, c
```

Example:

```
taddcctv  %g0, 1, %g0    ! nzvc=0010
```

TCC**TCC****Trap on Carry Clear (Trap on Greater Than or Equal, Unsigned)****Description:**

The TCC instruction causes a trap_instruction trap if (not C)=1 and if no higher priority trap is pending. The trap_instruction trap causes the *tt* field of the Trap Base Register (TBR) to be written with 128 plus the least significant seven bits of either “r[rs1] + r[rs2]” if the *i* field is zero, or “r[rs1] + sign_ext(simm13)” if the *i* field is one.

If (not C)=0, a trap_instruction trap does not occur and the instruction behaves like a NOP. All bits indicated as reserved in the instruction formats should be supplied as zero as should the most significant 25 bits of r[rs2] if the *i* field is 0.

Note: if single vector trapping is enabled, the trap_instruction trap will vector to the location pointed to by the Trap Base Address in the TBR and the *tt* field will be ignored)

Format:

31	30	29	28	25	24	19	18	14	13	12	5	4	0
10	reserved	1101	111010	rs1	i=0	unused (zero)	rs2						
31	30	29	28	25	24	19	18	14	13	12	7	6	0
10	reserved	1101	111010	rs1	i=1	reserved	software trap #						

Syntax:

```

tcc      regrs1, regrs2
tcc      regrs1, immediate
tgeu    regrs1, regrs2           !alternate mnemonic
tgeu    regrs1, immediate       !alternate mnemonic

```

Traps:

trap_instruction

Condition Code Modified:

(none)

Example:

```
tcc      %g0 + 33      ! tt = 161
```

Trap on Carry Set (Trap on Less Than, Unsigned)

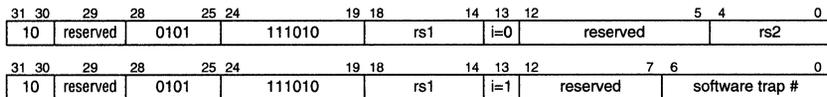
Description:

The TCS instruction causes a trap_instruction trap if C=1 and if no higher priority trap is pending. The trap_instruction trap causes the *tt* field of the Trap Base Register (TBR) to be written with 128 plus the least significant seven bits of either “r[rs1] + r[rs2]” if the *i* field is zero, or “r[rs1] + sign_ext(simm13)” if the *i* field is one.

If C=0, a trap_instruction trap does not occur and the instruction behaves like a NOP. All bits indicated as reserved in the instruction formats should be supplied as zero as should the most significant 25 bits of r[rs2] if the *i* field is 0.

(note: If single vector trapping is enabled, the trap_instruction trap will vector to the location pointed to by the Trap Base Address in the TBR, and the *tt* field will be ignored)

Format:



Syntax:

```

tcs      regrs1, regrs2
tcs      regrs1, immediate
tlu      regrs1, regrs2           ! alternate mnemonic
tlu      regrs1, immediate       ! alternate mnemonic
    
```

Traps:

trap_instruction

Condition Code Modified:

(none)

Example:

```

tcs      %g0 + 34      ! tt = 162
    
```

TE

TE

Trap on Equal

Description:

The TE instruction causes a trap_instruction trap if Z=1 and if no higher priority trap is pending. The trap_instruction trap causes the *tt* field of the Trap Base Register (TBR) to be written with 128 plus the least significant seven bits of either “r[rs1] + r[rs2]” if the *i* field is zero, or “r[rs1] + sign_ext(simm13)” if the *i* field is one.

If Z=0, a trap_instruction trap does not occur and the instruction behaves like a NOP. All bits indicated as reserved in the instruction formats should be supplied as zero as should the most significant 25 bits of r[rs2] if the *i* field is 0.

(note: If single vector trapping is enabled, the trap_instruction trap will vector to the location pointed to by the Trap Base Address in the TBR, and the *tt* field will be ignored)

Format:

31	30	29	28	25	24	19	18	14	13	12	5	4	0
10	reserved	0001		111010		rs1		i=0		reserved		rs2	
31	30	29	28	25	24	19	18	14	13	12	7	6	0
10	reserved	0001		111010		rs1		i=1		reserved		software trap #	

Syntax:

```
te      regrs1, regrs2
te      regrs1, immediate
```

Traps:

trap_instruction

Condition Code Modified:

(none)

Example:

```
te      %g0 + 36      ! tt = 164
```

Trap on Greater

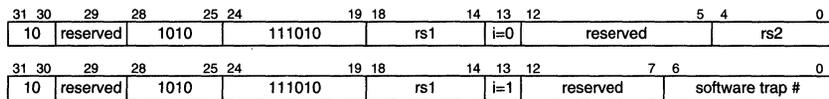
Description:

The TG instruction causes a trap_instruction trap if “not(Z or (N xor V))” is true and if no higher priority trap is pending. The trap_instruction trap causes the *tt* field of the Trap Base Register (TBR) to be written with 128 plus the least significant seven bits of either “r[rs1] + r[rs2]” if the *i* field is zero, or “r[rs1] + sign_ext(simm13)” if the *i* field is one.

If “not (Z or (N xor V))” is false, a trap_instruction trap does not occur and the instruction behaves like a NOP. All bits indicated as reserved in the instruction formats should be supplied as zero as should the most significant 25 bits of r[rs2] if the *i* field is 0.

(note: If single vector trapping is enabled, the trap_instruction trap will vector to the location pointed to by the Trap Base Address in the TBR, and the *tt* field will be ignored)

Format:



Syntax:

```
tg          regrs1, regrs2
tg          regrs1, immediate
```

Traps:

trap_instruction

Condition Code Modified:

(none)

Example:

```
tg          %g0+36          ! tt=164
```

Trap on Greater Than or Equal

Description:

The TGE instruction causes a trap_instruction trap if “not(N xor V)” is true and if no higher priority trap is pending. The trap_instruction trap causes the *tt* field of the Trap Base Register (TBR) to be written with 128 plus the least significant seven bits of either “r[rs1] + r[rs2]” if the *i* field is zero, or “r[rs1] + sign_ext(simm13)” if the *i* field is one.

If “not(N xor V)” is false, a trap_instruction trap does not occur and the instruction behaves like a NOP. All bits indicated as reserved in the instruction formats should be supplied as zero as should the most significant 25 bits of r[rs2] if the *i* field is 0.

(note: If single vector trapping is enabled, the trap_instruction trap will vector to the location pointed to by the Trap Base Address in the TBR, and the *tt* field will be ignored)

Format:

31	30	29	28	25	24	19	18	14	13	12	5	4	0
10	reserved	1011	111010	rs1	i=0	reserved	rs2						
31	30	29	28	25	24	19	18	14	13	12	7	6	0
10	reserved	1011	111010	rs1	i=1	reserved	software trap #						

Syntax:

```
tge      regrs1, regrs2
tge      regrs1, immediate
```

Traps:

trap_instruction

Condition Code Modified:

(none)

Example:

```
tge      %g0+37      ! tt=165
```

Trap on Greater Unsigned

Description:

The TGU instruction causes a trap_instruction trap if “not (C or Z)” is true and if no higher priority trap is pending. The trap_instruction trap causes the *tt* field of the Trap Base Register (TBR) to be written with 128 plus the least significant seven bits of either “*r[rs1]* + *r[rs2]*” if the *i* field is zero, or “*r[rs1]* + sign_ext(simm13)” if the *i* field is one.

If “not (C or Z)” is false, a trap_instruction trap does not occur and the instruction behaves like a NOP. All bits indicated as reserved in the instruction formats should be supplied as zero as should the most significant 25 bits of *r[rs2]* if the *i* field is 0.

(note: If single vector trapping is enabled, the trap_instruction trap will vector to the location pointed to by the Trap Base Address in the TBR, and the *tt* field will be ignored)

Format:

31	30	29	28	25	24	19	18	14	13	12	5	4	0
10	reserved	1100	111010	rs1	i=0	reserved	rs2						
31	30	29	28	25	24	19	18	14	13	12	7	6	0
10	reserved	1100	111010	rs1	i=1	reserved	software trap #						

Syntax:

```
tgu      regrs1, regrs2
tgu      regrs1, immediate
```

Traps:

```
trap_instruction
```

Condition Code Modified:

(none)

Example:

```
tgu      %g0+38      ! tt=166
```

TL

TL

Trap on Less

Description:

The TL instruction causes a trap_instruction trap if “N xor V” is true and if no higher priority trap is pending. The trap_instruction trap causes the *tt* field of the Trap Base Register (TBR) to be written with 128 plus the least significant seven bits of either “r[rs1] + r[rs2]” if the *i* field is zero, or “r[rs1] + sign_ext(simm13)” if the *i* field is one.

If “N xor V” is false, a trap_instruction trap does not occur and the instruction behaves like a NOP. All bits indicated as reserved in the instruction formats should be supplied as zero as should the most significant 25 bits of r[rs2] if the *i* field is 0.

(note: If single vector trapping is enabled, the trap_instruction trap will vector to the location pointed to by the Trap Base Address in the TBR, and the *tt* field will be ignored)

Format:

31	30	29	28	25	24	19	18	14	13	12	5	4	0
10	reserved	0011	111010	rs1	i=0	reserved	rs2						
31	30	29	28	25	24	19	18	14	13	12	7	6	0
10	reserved	0011	111010	rs1	i=1	reserved	software trap #						

Syntax:

```
t1      regrs1, regrs2
t1      regrs1, immediate
```

Traps:

```
trap_instruction
```

Condition Code Modified:

(none)

Example:

```
t1      %g0 + 40      ! tt=168
```

Trap on Less Than or Equal

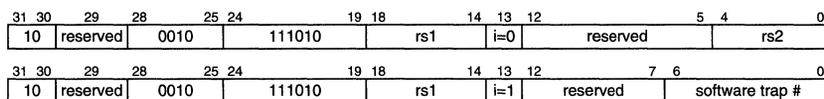
Description:

The TLE instruction causes a trap_instruction trap if “Z or (N xor V)” is true and if no higher priority trap is pending. The trap_instruction trap causes the *tt* field of the Trap Base Register (TBR) to be written with 128 plus the least significant seven bits of either “*r[rs1] + r[rs2]*” if the *i* field is zero, or “*r[rs1] + sign_ext(simm13)*” if the *i* field is one.

If “Z or (N xor V)” is false, a trap_instruction trap does not occur and the instruction behaves like a NOP. All bits indicated as reserved in the instruction formats should be supplied as zero as should the most significant 25 bits of *r[rs2]* if the *i* field is 0.

(note: If single vector trapping is enabled, the trap_instruction trap will vector to the location pointed to by the Trap Base Address in the TBR, and the *tt* field will be ignored)

Format:



Syntax:

```
tle      regrs1, regrs2
tle      regrs1, immediate
```

Traps:

```
trap_instruction
```

Condition Code Modified:

(none)

Example:

```
tle      %g0 + 41      ! tt = 169
```

TLEU

TLEU

Trap on Less Than or Equal Unsigned

Description:

The `tleu` instruction causes a `trap_instruction` trap if “C or Z” is true and if no higher priority trap is pending. The `trap_instruction` trap causes the `tt` field of the Trap Base Register (TBR) to be written with 128 plus the least significant seven bits of either “`r[rs1] + r[rs2]`” if the `i` field is zero, or “`r[rs1] + sign_ext(simm13)`” if the `i` field is one.

If “C or Z” is false, a `trap_instruction` trap does not occur and the instruction behaves like a NOP. All bits indicated as reserved in the instruction formats should be supplied as zero as should the most significant 25 bits of `r[rs2]` if the `i` field is 0.

(note: If single vector trapping is enabled, the `trap_instruction` trap will vector to the location pointed to by the Trap Base Address in the TBR, and the `tt` field will be ignored)

Format:

31	30	29	28	25	24	19	18	14	13	12	5	4	0
10	reserved	0100	111010	rs1	i=0	reserved	rs2						
31	30	29	28	25	24	19	18	14	13	12	7	6	0
10	reserved	0100	111010	rs1	i=1	reserved	software trap #						

Syntax:

```
tleu    regrs1, regrs2
tleu    regrs1, immediate
```

Traps:

```
trap_instruction
```

Condition Code Modified:

(none)

Example:

```
tleu    %g0+42    ! tt =170
```

TN

TN

Trap Never

Description:

The TN instruction acts like a “NOP”.

All bits indicated as reserved in the instruction formats should be supplied as zero as should the most significant 25 bits of r[rs2] if the *i* field is 0.

Format:

31	30	29	28	25	24	19	18	14	13	12	5	4	0
10	reserved	0000	111010	rs1	i=0	reserved	rs2						
31	30	29	28	25	24	19	18	14	13	12	7	6	0
10	reserved	0000	111010	rs1	i=1	reserved	software trap #						

Syntax:

```
tn      regrs1, regrs2
tn      regrs1, immediate
```

Traps:

trap_instruction

Condition Code Modified:

(none)

Example:

```
tn      %g0 + 39      ! nop
```

TNE**TNE****Trap on Not Equal (Trap on Not Zero)****Description:**

The TNE instruction causes a trap_instruction trap if Z=0 and if no higher priority trap is pending. The trap_instruction trap causes the *tt* field of the Trap Base Register (TBR) to be written with 128 plus the least significant seven bits of either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one.

If Z=1, a trap_instruction trap does not occur and the instruction behaves like a NOP. All bits indicated as reserved in the instruction formats should be supplied as zero as should the most significant 25 bits of r[rs2] if the *i* field is 0.

(note: If single vector trapping is enabled, the trap_instruction trap will vector to the location pointed to by the Trap Base Address in the TBR, and the *tt* field will be ignored)

Format:

31	30	29	28	25	24	19	18	14	13	12	5	4	0
10	reserved	1001	111010	rs1	i=0	reserved	rs2						
31	30	29	28	25	24	19	18	14	13	12	7	6	0
10	reserved	1001	111010	rs1	i=1	reserved	software trap #						

Syntax:

```
tne      regrs1, regrs2
tne      regrs1, immediate
tnz      regrs1, regrs2
tnz      regrs1, immediate
```

Traps:

```
trap_instruction
```

Condition Code Modified:

```
(none)
```

Example:

```
tne      %g0 + 43      !tt=171
```

TNEG

TNEG

Trap on Negative

Description:

The TNEG instruction causes a trap_instruction trap if N=1 and if no higher priority trap is pending. The trap_instruction trap causes the *tt* field of the Trap Base Register (TBR) to be written with 128 plus the least significant seven bits of either “r[rs1] + r[rs2]” if the *i* field is zero, or “r[rs1] + sign_ext(simm13)” if the *i* field is one.

If N=0, a trap_instruction trap does not occur and the instruction behaves like a NOP. All bits indicated as reserved in the instruction formats should be supplied as zero as should the most significant 25 bits of r[rs2] if the *i* field is 0.

(note: If single vector trapping is enabled, the trap_instruction trap will vector to the location pointed to by the Trap Base Address in the TBR, and the *tt* field will be ignored)

Format:

31	30	29	28	25	24	19	18	14	13	12	5	4	0
10	reserved	0110	111010	rs1	i=0	reserved	rs2						
31	30	29	28	25	24	19	18	14	13	12	7	6	0
10	reserved	0110	111010	rs1	i=1	reserved	software trap #						

Syntax:

```
tneg    regrs1, regrs2
tneg    regrs1, immediate
```

Traps:

trap_instruction

Condition Code Modified:

(none)

Example:

```
tneg    %g0 + 44    ! tt = 172
```

TPOS

TPOS

Trap on Positive

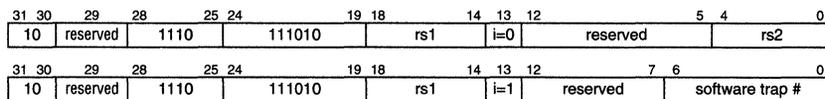
Description:

The TPOS instruction causes a trap_instruction trap if N=0 and if no higher priority trap is pending. The trap_instruction trap causes the *tt* field of the Trap Base Register (TBR) to be written with 128 plus the least significant seven bits of either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(sim13)" if the *i* field is one.

If N=1, a trap_instruction trap does not occur and the instruction behaves like a NOP. All bits indicated as reserved in the instruction formats should be supplied as zero as should the most significant 25 bits of r[rs2] if the *i* field is 0.

(note: If single vector trapping is enabled, the trap_instruction trap will vector to the location pointed to by the Trap Base Address in the TBR, and the *tt* field will be ignored)

Format:



Syntax:

```
tpos    regrs1, regrs2
tpos    regrs1, immediate
```

Traps:

```
trap_instruction
```

Condition Code Modified:

(none)

Example:

```
tpos    %g0 + 45    ! tt = 173
```

TSUBcc

TSUBcc

Tagged Subtract and modify condition codes

Description:

Computes either “ $r[rs1]-r[rs2]$ ” if the i field is zero, or “ $r[rs1] - \text{sign_ext}(\text{simm13})$ ” if the i field is one, and places the result in the destination specified by the rd field.

TSUBcc modifies the condition codes. The overflow bit of the PSR is set if bit 1 or bit 0 of either operand is nonzero. The overflow bit is also set if the operands have different signs and the sign of the difference differs from the sign of $r[rs1]$.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd			100001			rs1	i=0	unused (zero)			rs2
31	30	29	25	24	19	18	14	13	12	0		
10	rd			100001			rs1	i=1	simm13			

Syntax:

```
tsubcc    regrs1, regrs2, regrd
tsubcc    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

n, z, v, c

Example:

```
tsubcc    %g0, 2, %g0    ! nzvc = 1011
```

TSUBccTV**TSUBccTV****Tagged Subtract, modify condition codes and Trap on Overflow****Description:**

Computes either “ $r[rs1]-r[rs2]$ ” if the i field is zero, or “ $r[rs1] - \text{sign_ext}(\text{simmm13})$ ” if the i field is one, and places the result in the destination specified by the rd field.

A `tag_overflow` occurs if bit 1 or bit 0 of either operand is nonzero, or if the subtraction generates an arithmetic overflow (the operands have different signs and the sign of the difference differs from the sign of $r[rs1]$).

If TSUBccTV causes a `tag_overflow`, a `tag_overflow` trap is generated and the destination register (rd) and condition codes remain unchanged. If a `tag_overflow` does not occur, the integer condition codes are updated ($v=0$).

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0	
10	rd			100011	$rs1$		$i=0$	unused (zero)			$rs2$		
31	30	29	25	24	19	18	14	13	12				0
10	rd			100011	$rs1$		$i=1$	$simmm13$					

Syntax:

```
tsubccv  regrs1, regrs2, regrd
tsubccv  regrs1, immediate, regrd
```

Traps:

`tag_overflow`

Condition Code Modified:

n, z, v, c

Example:

```
tsubccv  %g0, 2, %g0    ! nzvc = 1011
```

Trap on Overflow Clear

Description:

The TVC instruction causes a trap_instruction trap if V=0 and if no higher priority trap is pending. The trap_instruction trap causes the *tt* field of the Trap Base Register (TBR) to be written with 128 plus the least significant seven bits of either “r[rs1] + r[rs2]” if the *i* field is zero, or “r[rs1] + sign_ext(simm13)” if the *i* field is one.

If V=1, a trap_instruction trap does not occur and the instruction behaves like a NOP. All bits indicated as reserved in the instruction formats should be supplied as zero as should the most significant 25 bits of r[rs2] if the *i* field is 0.

(note: If single vector trapping is enabled, the trap_instruction trap will vector to the location pointed to by the Trap Base Address in the TBR, and the *tt* field will be ignored)

Format:

31	30	29	28	25	24	19	18	14	13	12	5	4	0
10	reserved	1111		111010		rs1		i=0		reserved			rs2
31	30	29	28	25	24	19	18	14	13	12	7	6	0
10	reserved	1111		111010		rs1		i=1		reserved			software trap #

Syntax:

```
tvc      regrs1, regrs2
tvc      regrs1, immediate
```

Traps:

trap_instruction

Condition Code Modified:

(none)

Example:

```
tvc      %g0, + 146      ! tt = 174
```

TVS

TVS

Trap on Overflow Set

Description:

The TVS instruction causes a trap_instruction trap if V=1 and if no higher priority trap is pending. The trap_instruction trap causes the *tt* field of the Trap Base Register (TBR) to be written with 128 plus the least significant seven bits of either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one.

If V=0, a trap_instruction trap does not occur and the instruction behaves like a NOP. All bits indicated as reserved in the instruction formats should be supplied as zero as should the most significant 25 bits of r[rs2] if the *i* field is 0.

(note: If single vector trapping is enabled, the trap_instruction trap will vector to the location pointed to by the Trap Base Address in the TBR, and the *tt* field will be ignored)

Format:

31	30	29	28	25	24	19	18	14	13	12	5	4	0
10	reserved	0111		111010		rs1		i=0		reserved		rs2	
31	30	29	28	25	24	19	18	14	13	12	7	6	0
10	reserved	0111		111010		rs1		i=1		reserved		software trap #	

Syntax:

```

tvs      regrs1, regrs2
tvs      regrs1, immediate

```

Traps:

```
trap_instruction
```

Condition Code Modified:

(none)

Example:

```
tvs      %g0 + 147      ! tt = 175
```

Unsigned Integer Multiply

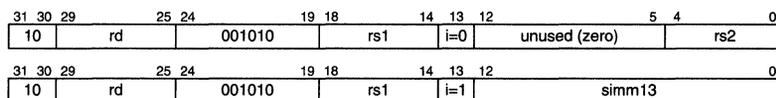
Description:

UMUL performs either “ $r[rs1] \times r[rs2]$ ” if the *i* field is zero, or “ $r[rs1] \times \text{sign_ext}(\text{simm13})$ ” if the *i* field is one. The 32 least significant bits of the product are written to the destination register $r[rd]$. The most significant bits of the product are written to the Y register.

The UMUL operation takes 5 cycles to compute a 32 bit x word operation, 3 cycles to compute a 32 bit x halfword operation, and 2 cycles to compute a 32 bit x byte operation. To do this, the hardware tests the most significant 16, 24 or 32 bits of $r[rs2]$ against the sign bit at run time. If the bits match, the UMUL instruction will terminate in 3, 2 or 1 cycle respectively.

UMUL assumes an unsigned integer word operand and computes an unsigned integer doubleword product.

Format:



Syntax:

```
umul    regrs1, regrs2, regrd
umul    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
umul    %o2, %o3, %o1 ! least significant half product to reg
                    o1
rd      %y, %o0      ! most significant half product to reg
                    o0
```

UMULcc**UMULcc****Signed Integer Multiply and Change Condition Codes****Description:**

UMULcc performs either “ $r[rs1] \times r[rs2]$ ” if the *i* field is zero, or “ $r[rs1] \times \text{sign_ext}(\text{simm13})$ ” if the *i* field is one. The 32 least significant bits of the product are written to the destination register *r[rd]*. The most significant bits of the product are written to the Y register.

The UMULcc operation takes 5 cycles to compute a 32 bit x word operation, 3 cycles to compute a 32 bit x halfword operation, and 2 cycles to compute a 32 bit x byte operation. To do this, the hardware tests the most significant 16, 24 or 32 bits of *r[rs2]* against the sign bit at run time. If the bits match, the UMULcc instruction will terminate in 3, 2 or 1 cycle respectively.

UMULcc assumes an unsigned integer word operand and computes an unsigned integer doubleword product. UMULcc writes the integer condition code bits (see below)

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd		011010		rs1		i=0	unused (zero)			rs2	
31	30	29	25	24	19	18	14	13	12			0
10	rd		011010		rs1		i=1	simm13				

Syntax:

```
umulcc    regrs1, regrs2, regrd
umulcc    regrs1, immediate, regrd
```

Signed Integer Multiply and Change Condition Codes (Continued)

Traps:

(none)

Condition Code Modified:.

Table 7-4:

icc bit	UMULcc
N	Set if product [31] = 1
Z	Set if product [31:0] = 0
V	Zero
C	Zero

Example:

```

umulcc    %o2, %o3, %o1 ! least significant half product to reg
              o1
rd        %y, %o0      ! most significant half product to reg
              o0
    
```

WRASR

WRASR

Write Ancillary State Register

Description:

WRASR writes “ $r[rs1] \text{ xor } r[rs2]$ ” if the i field is zero, or “ $r[rs1] \text{ xor sign_ext(simmm13)}$ ” if the i field is one, to the writable fields of the ASR register specified in rd (16-31).

On the SPARClite MB86930 a valid rd value is 17. All other values of rd will generate an illegal instruction trap.

WRASR is a privileged instruction.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd			110000			rs1	i=0	unused (zero)			rs2
31	30	29	25	24	19	18	14	13	12			0
10	rd			110000			rs1	i=1	simmm13			

Syntax:

```
wr      regrs1, regrs2, asr_regrd
wr      regrs1, immediate, asr_regrd
```

Traps:

```
illegal_instruction
privileged_instruction
```

Condition Code Modified:

(none)

Example:

```
wr      %g0, 1, %asr17    ! enable single vector trapping
wr      %g0, 0, %asr17    ! disable single vector trapping
```

WRPSR

WRPSR

Write Processor State Register

Description:

WRPSR causes a delayed write of “ $r[rs1] \text{ xor } r[rs2]$ ” if the i field is zero, or “ $r[rs1] \text{ xor sign_ext(simm13)}$ ” if the i field is one, to the writable fields of the PSR register.

WRPSR is a privileged instruction. See section 2.4.7 for programming considerations.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	reserved			110001			rs1	i=0	unused (zero)			rs2
31	30	29	25	24	19	18	14	13	12	5	4	0
10	reserved			110001			rs1	i=1				

Note: reserved fields should be programmed as 0.

Syntax:

```
wr      regrs1, regrs2, %psr
wr      regrs1, immediate, %psr
```

Traps:

privileged_instruction

Condition Code Modified:

(none)

Example:

```
wr      %g0, 0xec7, %psr ! e to pil, 1 to S & PS, 0 to et, 7 to cwp
```

WRTBR**WRTBR****Write Trap Base Register****Description:**

WRTBR causes a delayed write of “r[rs1] xor r[rs2]” if the *i* field is zero, or “r[rs1] xor sign_ext(simm13)” if the *i* field is one, to the writable fields of the TBR register.

WRPSR is a privileged instruction.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	reserved			110011			rs1	i=0	unused (zero)			rs2
31	30	29	25	24	19	18	14	13	12	0		
10	reserved			110011			rs1	i=1	simm13			

Note: reserved fields should be programmed as 0.

Syntax:

```
wr      regrs1, regrs2, %tbr
wr      regrs1, immediate, %tbr
```

Traps:

privileged_instruction

Condition Code Modified:

(none)

Example:

```
wr      %g0, 0x1000, %tbr
```

WRWIM

WRWIM

Write Window Invalid Mask Register

Description:

WRWIM causes a delayed write of “r[rs1] xor r[rs2]” if the *i* field is zero, or “r[rs1] xor sign_ext(simm13)” if the *i* field is one, to the writable fields of the WIM register.

WRWIM is a privileged instruction.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	reserved			110010			rs1	i=0	unused (zero)			rs2
31	30	29	25	24	19	18	14	13	12			0
10	reserved			110010			rs1	i=1	simm13			

Note: reserved fields should be programmed as 0.

Syntax:

```
wr      regrs1, regrs2, %wim
wr      regrs1, immediate, %wim
```

Traps:

privileged_instruction

Condition Code Modified:

(none)

Example:

```
wr      %g0, -256, %wim      ! only windows 0 to 7 valid
                                ! windows 8 and above invalid
```

WRY**WRY****Write Y Register****Description:**

WRY writes “r[rs1] xor r[rs2]” if the *i* field is zero, or “r[rs1] xor sign_ext(simm13)” if the *i* field is one, to the Y register.

Unlike the other write state register instructions, WRY is not a privileged instruction.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10		00000		110000		rs1		i=0		unused (zero)		rs2
31	30	29	25	24	19	18	14	13	12			0
10		00000		110000		rs1		i=1		simm13		0

Note: reserved fields should be programmed as 0.

Syntax:

```
WY      regrs1, regrs2, %y
WY      regrs1, immediate, %y
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
WY      %g0, 0, %y      ! clear reg y
```

XNOR

XNOR

Exclusive NOR

Description:

Implements a bitwise logical exclusive Nor to compute either “ $r[rs1] \text{ xnor } r[rs2]$ ” if the i field is zero, or “ $r[rs1] \text{ xnor sign_ext(simm13)}$ ” if the i field is one, and places the result in the destination specified by the rd field.

Format:

31	30	29		25	24		19	18		14	13	12		5	4		0
10			rd			000111			rs1		i=0			unused (zero)			rs2
31	30	29		25	24		19	18		14	13	12					0
10			rd			000111			rs1		i=1			simm13			

Syntax:

```
xnor    regrs1, regrs2, regrd
xnor    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
xnor    %l1, 0, %l1    ! complement reg l1
```

XNORcc**XNORcc****Exclusive NOR and modify icc****Description:**

Implements a bitwise logical exclusive Nor to compute either “ $r[rs1] \text{ xnor } r[rs2]$ ” if the i field is zero, or “ $r[rs1] \text{ xnor sign_ext(simm13)}$ ” if the i field is one, and places the result in the destination specified by the rd field.

XNORcc modifies the integer condition codes.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd			010111			rs1		i=0	unused (zero)		rs2
31	30	29	25	24	19	18	14	13	12			0
10	rd			010111			rs1		i=1	simm13		

Syntax:

```
xnorcc    regrs1, regrs2, regrd
xnorcc    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

$n, z=0, v, c=0$

Example:

```
xnorcc    %l1, %l2, %g0    ! do any bits in reg l1 match corresponding
                           bits
                           !   in reg l2?
bne       xyz              ! skip ahead if not
```

XOR

XOR

Exclusive OR

Description:

Implements a bitwise logical exclusive Or to compute either “ $r[rs1] \text{ xor } r[rs2]$ ” if the i field is zero, or “ $r[rs1] \text{ xor sign_ext(simm13)}$ ” if the i field is one, and places the result in the destination specified by the rd field.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0	
10	rd		000011		rs1		i=0	unused (zero)			rs2		
31	30	29	25	24	19	18	14	13	12				0
10	rd		000011		rs1		i=1	simm13					

Syntax:

```
xor      regrs1, regrs2, regrd
xor      regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

(none)

Example:

```
xor      %l1, -1, %l1 ! complement reg l1
```

XORcc**XORcc****Exclusive NOR and modify icc****Description:**

Implements a bitwise logical exclusive Or to compute either “ $r[rs1] \text{ xor } r[rs2]$ ” if the i field is zero, or “ $r[rs1] \text{ xor sign_ext(simm13)}$ ” if the i field is one, and places the result in the destination specified by the rd field.

XORcc modifies the integer condition codes.

Format:

31	30	29	25	24	19	18	14	13	12	5	4	0
10	rd			010011			rs1		i=0	unused (zero)		rs2
31	30	29	25	24	19	18	14	13	12			0
10	rd			010011			rs1		i=1	simm13		

Syntax:

```
xorcc    regrs1, regrs2, regrd
xorcc    regrs1, immediate, regrd
```

Traps:

(none)

Condition Code Modified:

$n, z=0, v, c=0$

Example:

```
xorcc    %l1, -1, %l1    ! complement reg l1 and test result
```


8.2 Test Access Ports (TAP)

SPARClite has five dedicated pins for JTAG.

Name	Input/Output	Weak pull-up	Function
TCK	Input	No	Test Clock
TMS	Input	Yes	Test Mode Select
TDI	Input	Yes	Test Data Input
TDO	Output	No	Test Data Output
-TRST	Input	Yes	Test Reset

8.2.1 TCK

JTAG uses a test clock independent of component-specific system clock. This is necessary to be able to shift the serial test data through components with different operating frequencies. An independent test clock allows shifting of test data concurrently with the system operation of the component and without changing the state of the on-chip system logic. Following are the JTAG requirements and clock specifications.

1. The JTAG test logic state will remain unchanged indefinitely when TCK=0.
2. A 50% duty cycle clock is recommended.

8.2.2 TMS

The sequence of TMS inputs is used to put the JTAG test logic into a particular test mode. The test logic must be in the correct test mode to shift-in instructions, to do data-shifts and do other operations.

1. TMS input is sampled by the test logic at the rising edge of TCK.
2. Undriven TMS input appears as a logic "1" to the test logic. This is to ensure that the test logic will sequence to the Test_Logic_Reset state if the TMS is held high for at least five rising edges of TCK. The test logic will remain in the Test_Logic_Reset state as long as TMS=1. (See "Test Logic Reset" on page 8-10.)

8.2.3 TDI

The TDI pin is used to input test instructions and test data.

1. The TDI input is sampled by the test logic at the rising edge of TCK.
2. Undriven TDI input appears as a logic "1" to the test logic.
3. No logic inversion takes place when data is being shifted from TDI towards TDO.
4. TDI input change at the falling edge of TCK is recommended.

8.2.4 TDO

TDO is the serial output for the test instructions and data from the test logic.

1. TDO output is valid after the falling edge of TCK.
2. TDO output is in the high-impedance state when data or instruction is not scanned.

8.2.5 -TRST

-TRST is an asynchronous test logic reset pin.

1. The test logic is forced into the Test_Logic_Reset state asynchronously when a logic "0" is applied to the -TRST pin.
2. If it is not being driven, -TRST pin appears as a logic "1" to the test logic. This is to ensure normal test operation in the event of an unterminated -TRST.
3. -TRST does not initialize any system logic within the component.
4. To ensure deterministic operation of the test logic, the TMS input should be held at 1 while the -TRST signal changes from 0 to 1.

8.3 Test Instructions

SPARClite implements the three JTAG public instructions; BYPASS, SAMPLE/PRELOAD and EXTEST.

SPARClite contains a two bit JTAG instruction register which receives the instruction serially from the TDI input. The instruction bits are shifted-in at the rising edge of TCK. For fault isolation of the board level serial test data path, a constant binary "01" pattern is loaded into the instruction shift register at the start of the instruction-shift cycle. Therefore, a "01" pattern will appear at the TDO output in the beginning of the instruction-shift cycle.

When shifting the instruction into the instruction register, the least significant bit of the instruction needs to be shifted in first, followed by the most significant bit.

8.3.1 BYPASS

The BYPASS instruction is used to bypass a component that is connected in series with other components. This allows more rapid movement of test data through the components of the board, bypassing the ones that do not need to be tested. The BYPASS operation enables the bypass register, which is a single stage shift register, between TDI and TDO.

1. The binary code for the BYPASS instruction is 11.
2. The BYPASS instruction is forced into the instruction register output latches during the Test_Logic_Reset state. Note the distinction between the "01" content of the instruction shift register and the "11" content of the instruction register output latch. Therefore, at the start of the instruction-shift cycle, a "01" pattern will be seen instead of "11".
3. The BYPASS operation does not interfere with the component operation at all. If the TDI input trace to the component is somehow disconnected, the test logic will see a "11" at TDI input during the instruction-shift state. Therefore, no unwanted interference with the on-chip system logic occurs.

8.3.2 SAMPLE/PRELOAD

The SAMPLE/PRELOAD instruction is used to sample the state of the component pins. The sampled values can be examined by shifting out the data through TDO. This instruction can also be used to preload the boundary-scan cell output latches with specific values. The preloaded values are then enabled to the output pins by the EXTEST.

1. The binary code for the instruction is 01.
2. The SAMPLE/PRELOAD instruction selects the boundary-scan cells to be connected between TDI and TDO in the Shift_DR TAP controller state (see section 8.4).
3. The values of the component pins are sampled on the rising edge of TCK in the Capture_DR TAP controller state.
4. The preload values shifted into the boundary-scan cells are latched into the boundary-scan output latch at the falling edge of TCK in the Update_DR TAP controller state.

8.3.3 EXTEST

EXTEST instruction allows testing of off-chip circuitry and board level interconnections. The PRELOAD/SAMPLE instruction is used to preload the data into the latched parallel outputs of the boundary-scan shift register stages. Then, the EXTEST instruction enables the preloaded values to the components output pins.

1. The binary code for the instruction is 00.
2. SPARClite outputs the preloaded data to the pins at the falling edge of TCK in the Update_IR TAP controller state at which point the JTAG instruction register is updated with the EXTEST.
3. The EXTEST instruction selects the boundary-scan cells to be connected between TDI and TDO in the Shift_DR test logic controller state.
4. Once the EXTEST instruction is effective, the output pins can change at the falling edge of TCK in the Update_DR TAP controller state.

8.3.4 JTAG Cells

SPARClite's JTAG test data scan path is composed of input cells, output cells, I/O cells and output cells with set control. The basic structures of the cells are shown in the accompanying figures. As the name implies, the input cell is used for input-only pins and the output cell is used for output-only pins. The I/O cell is used for the I/O pins and the output cell with set control is used for I/O buffer control.

With each group of I/O pins there is an I/O buffer control JTAG cell which is used to control the direction of the I/O pins during EXTEST operation. This implies that within the data-scan path there are cells which do not correspond to a pin, but are used for I/O buffer control during EXTEST operation.

Note that the output cell and the I/O cell have an output latch separate from the shift register. This allows the output to remain unchanged during a data-shift operation during the EXTEST mode. The cell output latches are updated during the Update_DR state (see section 8.4).

8.3.5 Input Cell

For SPARClite, an input cell structure with signal capture only capability has been chosen to minimize the propagation delay from the input pins to the on-chip system logic. Using the SAMPLE/PRELOAD instruction, the user can sample the input pin and scan out the sampled value.

8.3.6 Output Cell

The output cell has the capability to output a preloaded value to the output pin during EXTEST. During EXTEST, the source of the output changes from the chip logic to the output latch of the JTAG output cell. The output value in the cell is preloaded using the SAMPLE/PRELOAD instruction.

8.3.7 I/O Cell

The I/O cell is actually composed of an input cell and an output cell. Therefore, for each I/O pin there are two cells associated with the pin. Hence, when the data is shifted out through TDO, two bits for each I/O pin will be seen. As mentioned previously, an I/O buffer control cell is associated with each group of I/O pins. For example, the 32-bit data bus is controlled by the data I/O buffer control cell. The I/O buffer control cell is also in the data scan path through which the user can control the direction of the I/O buffer for the EXTEST.

8.3.8 Output Cell with Set

This cell is used as the I/O buffer control cell. The output latch of the cell is set during Test_Logic_Reset state so that if EXTEST is entered after reset, the I/O pins are in the input mode. There is one I/O buffer control cell for each group of I/O signals.

I/O buffer control cell name	I/O pins
emudiojo	EMU_D<3:0>, EMU_SD<3:0>
emuenblo	-EMU_ENB
dbusiojo	D<31:0>
tstatejo	Output Pins†

†. Not all output pins are three-statable

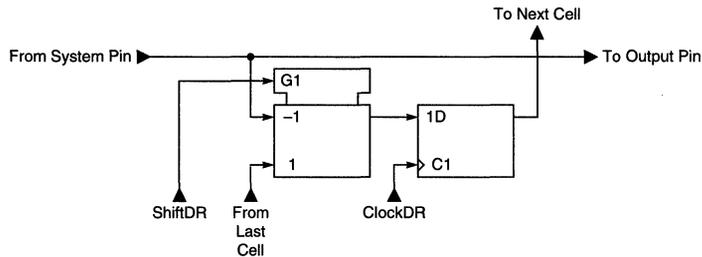


Figure 8-1. Input Cell Allowing Signal Capture Only

8.4 Operation

The JTAG control logic, which is also referred to as the TAP controller, is implemented with a synchronous finite state machine. The asynchronous reset input ($-\text{TRST}$) and the TMS input control the state transition of the TAP controller. To shift instructions into the instruction register and to do test data-scans, the TAP controller needs to be in the appropriate state (see Figure 8-5 and Figure 8-6 for timing relationship). A TAP state transition diagram is provided with examples in the following pages.

The usual sequence of operations is as follows. Initially, the TAP controller is forced into the reset state, *Test_Logic_Reset*, by $-\text{TRST}=0$. Next, TMS is set to a "1" and the $-\text{TRST}$ is deasserted at the falling edge of TCK. At the next rising edge of TCK, the TMS=1 value is sampled by the test logic and the TAP controller remains in the reset state. The first thing that needs to be done is to shift in the 2 bit instruction into the JTAG instruction register.

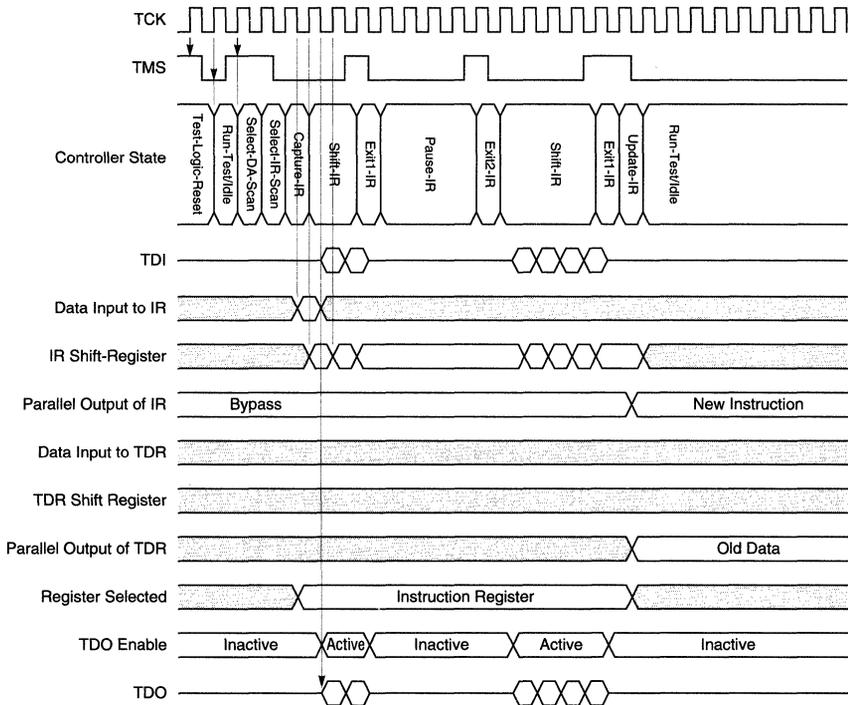


Figure 8-5. Test Logic Operation: Instruction Scan

To do so, the TAP controller needs to be transitioned to the *Shift_IR* state. In order to make the state transition from *Test_Logic_Reset* to *Shift_IR* state, the correct

TMS sequence would have to be 0 -> 1 -> 1 -> 0 -> 0. Remember that the TMS input should change at the falling edge of TCK so that enough setup time is available with respect to the rising edge of TCK at which point the TMS input is sampled. The TAP controller changes state at the rising edge of TCK. Once in the Shift_IR state, the instruction bits at TDI will be shifted into the JTAG instruction register at the rising edge of TCK. Suppose the instruction shifted in was a SAMPLE/PRELOAD. Then as soon as the instruction is shifted in, the TAP controller must transition to the Exit1_IR state to terminate the instruction-scan. Otherwise, more than 2 bits will be shifted into the instruction register.

For the SAMPLE/PRELOAD instruction, data shifts need to take place either to output the sampled value of the pins or to shift in the preload value for EXTEST. Therefore, the TAP controller needs to change state from Exit1_IR to the Shift_DR state. This is accomplished by giving the 1 -> 0 -> 1 -> 0 -> 0 TMS sequence. Once, in the Shift_DR state, the TDI input will be scanned into the shift register portion of the boundary scan cells at the rising edge of TCK. Once data-scan is finished, the TAP controller state can be transitioned to the Run_Test/Idle state for the next JTAG instruction.

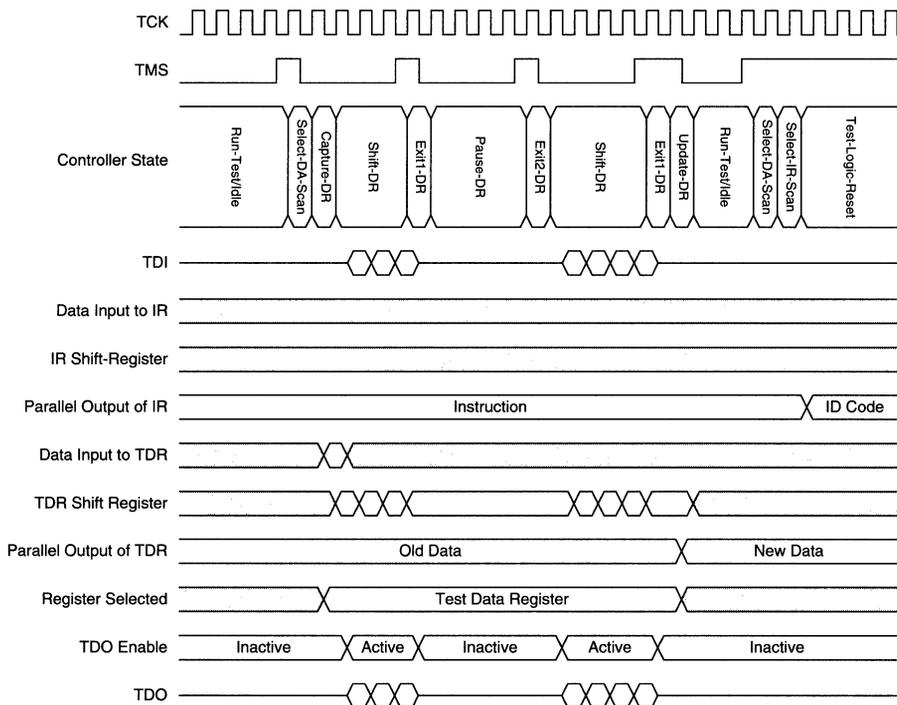


Figure 8-6. Test Logic Operation: Data Scan

8.5 The TAP Controller

8.5.1 TAP Controller State Diagram

Specifications

Rules

1. The state diagram for the TAP controller is shown in Figure 8-7. (Note the value shown adjacent to each state transition arc in this figure represents the signal present at TMS at the time of a rising edge at TCK.)
2. All state transition of the TAP controller must occur based on the value of TMS at the time of a rising edge of TCK.
3. Actions of the test logic occur on either the rising or the falling edge of TCK in each controller state.

Description

The behavior of the TAP controller and other test logic in each of the controller states is briefly described as follows. Note the term, Test Data Registers, refers to either the Bypass Register or the 152 JTAG cells connected as a shift register.

Test Logic Reset

The test logic is disabled so that normal operation of the on-chip system logic (i.e., in response to stimuli received through the system pins only) can continue unhindered. This is achieved by initializing the instruction register with the BYPASS instruction. No matter what the original state of the controller may be, the controller will enter Test-Logic-Reset when the TMS input is held high for at least five rising edges of TCK. The controller remains in this state while TMS is high.

If the controller should leave the Test-Logic-Reset controller state as a result of an erroneous low signal on the TMS line at the time of a rising edge on TCK (for example, a glitch due to external interference), it will return to the Test-Logic-Reset state following three rising edges of TCK with the TMS line at the intended high logic level. The operation of the test logic is such that no disturbance is caused to on-chip system logic operation as the result of such an error. On leaving the Test-Logic-Reset controller state, the controller moves into the Run-Test/Idle controller state where no action will occur because the current instruction has been set to select operation of the bypass register. The test logic is also inactive in the Select-DR-Scan and Select-IR-Scan controller states.

Note that the TAP controller will also be forced to the Test-Logic-Reset controller state by applying a low logic level to the TRST* input.

Run-Test/Idle

A controller state between scan operations. In the Run-Test/Idle controller state, activity in selected test logic occurs only when certain instructions are present.

For instructions which do not cause functions to execute in the Run-Test/Idle controller state, all test data registers selected by the current instruction must retain their previous state (i.e., Idle).

The instruction does not change while TAP controller is in this state.

Select-DR-Scan

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state.

If TMS is held low and a rising edge is applied to TCK when the controller is in this state, then the controller moves into the Capture-DR state and a scan sequence for the selected test data register is initiated. If TMS is held high and a rising edge is applied to TCK the controller moves on to the Select-IR-Scan state.

The instruction does not change while the TAP controller is in this state.

Select-IR-Scan

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state.

If TMS is held low and a rising edge is applied to TCK when the controller is in this state, then the controller moves into the Capture-IR state and a scan sequence for the instruction register is initiated. If TMS is held high and a rising edge is applied to TCK the controller returns to the Test-Logic-Reset state.

The instruction does not change while TAP controller is in this state.

Capture-DR

In this controller state data may be parallel loaded into test data registers selected by the current instruction on the rising edge of TCK.

The instruction does not change while TAP controller is in this state.

Shift-DR

In this controller state, the test data register connected between TDI and TDO as a result of the current instruction shifts data one stage towards its serial output on each rising edge of TCK.

The instruction does not change while the TAP controller is in this state.

Exit1-DR

This is a temporary controller state. If TMS is held high, a rising edge applied to TCK while in this state causes the controller to enter the Update-DR state, which terminates the scanning process. If TMS is held low and a rising edge is applied to TCK, the controller enters the Pause-DR state.

All test data registers selected by the current instruction retain their previous state unchanged.

The instruction does not change while TAP controller is in this state.

Pause-DR

This controller state allows shifting of the test data register in the serial path between TDI and TDO to be temporarily halted. All test data registers selected by the current instruction retain their previous state unchanged.

The instruction does not change while TAP controller is in this state.

Exit2-DR

This is a temporary controller state. If TMS is held high and a rising edge is applied to TCK while in this state, the scanning process terminates and the TAP controller enters the Update-DR controller state. If TMS is held low and a rising edge is applied to TCK, the controller enters the Shift-DR state.

All test data register selected by the current instruction retain their previous state unchanged.

The instruction does not change while the TAP controller is in this state.

Update-DR

Some test data registers are provided with a latched parallel output to prevent changes at the parallel output while data is shifted in the associated shift-register path in response to certain instruction (e.g., EXTEST). Data is latched onto the parallel output of these test data register from the shift-register path on the falling edge of TCK in the Update-DR controller state. The data held at the latched parallel output should not change other than in this controller state.

All shift-register stages in test data registers selected by the current instruction retain their previous state unchanged.

The instruction does not change while the TAP controller is in this state.

Capture-IR

In this controller state the shift-register contained in the instruction register loads a pattern of fixed logic values on the rising edge of TCK.

Test data registers selected by the current instruction retain their previous state. The instruction does not change while the TAP controller is in this state.

Shift-IR

In this controller state the shift-register contained in the instruction register is connected between TDI and TDO and shifts data one stage towards its serial output on each rising edge of TCK.

Test data register selected by the current instruction retain their previous state. This instruction does not change while the TAP controller is in this state.

Exit1-IR

This is a temporary controller state. If TMS is held high, a rising edge applied to TCK while in this state causes the controller to enter the Update-IR state, which terminates the scanning process. If TMS is held low and a rising edge is applied to TCK, the controller enters the Pause-IR state.

Test data registers selected by the current instructions retain their previous state. The instruction does not change while the TAP controller is in this state and the instruction register retains its state.

Pause-IR

This controller state allows shifting of the instruction register to be temporarily halted.

Test data registers selected by the current instruction retain their previous state. The instruction does not change while the TAP controller is in this state and the instruction register retains its state.

Exit2-IR

This is temporary controller state. If TMS is held high and a rising edge is applied to TCK while in this state causes termination of the scanning process and the TAP controller enters the Update-IR controller state. If TMS is held low and a rising edge is applied to TCK the controller enters the Shift-IR state.

Test data registers selected by the current instruction retain their previous state. The instruction does not change while the TAP controller is in this state and the instruction register retains its state.

Update-IR

The instruction shifted into the instruction register is latched onto the parallel output from the shift-register path on the falling edge of TCK in this controller state. Once the new instruction has been latched it becomes the current instruction.

Test data registers selected by the current instruction retain their previous state.

The Pause-DR and Pause-IR controller states are included so that shifting of data through the test data or instruction register can be temporarily halted. For example, this might be necessary in order to allow an ATE system to reload its pin memory from disc during application of a long test sequence.

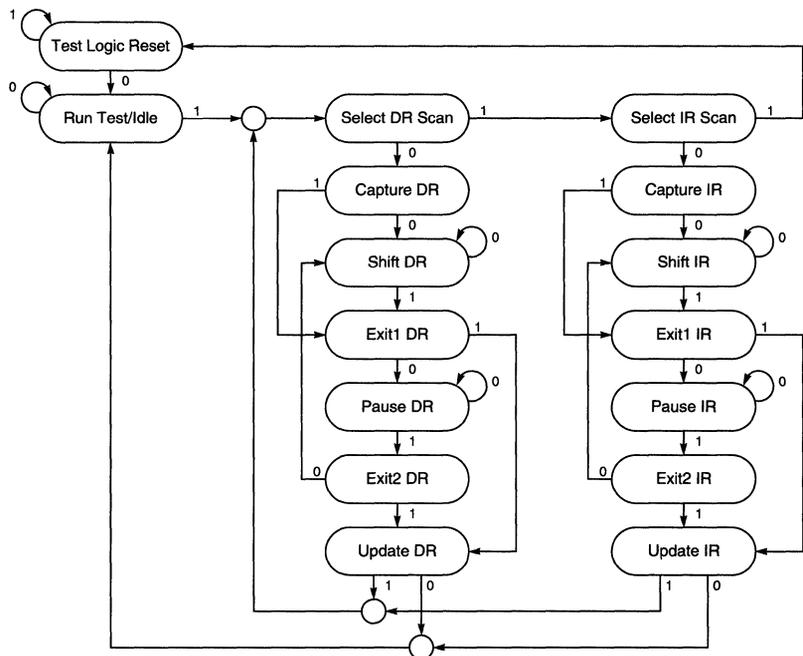


Figure 8-7. TAP Controller State Diagram

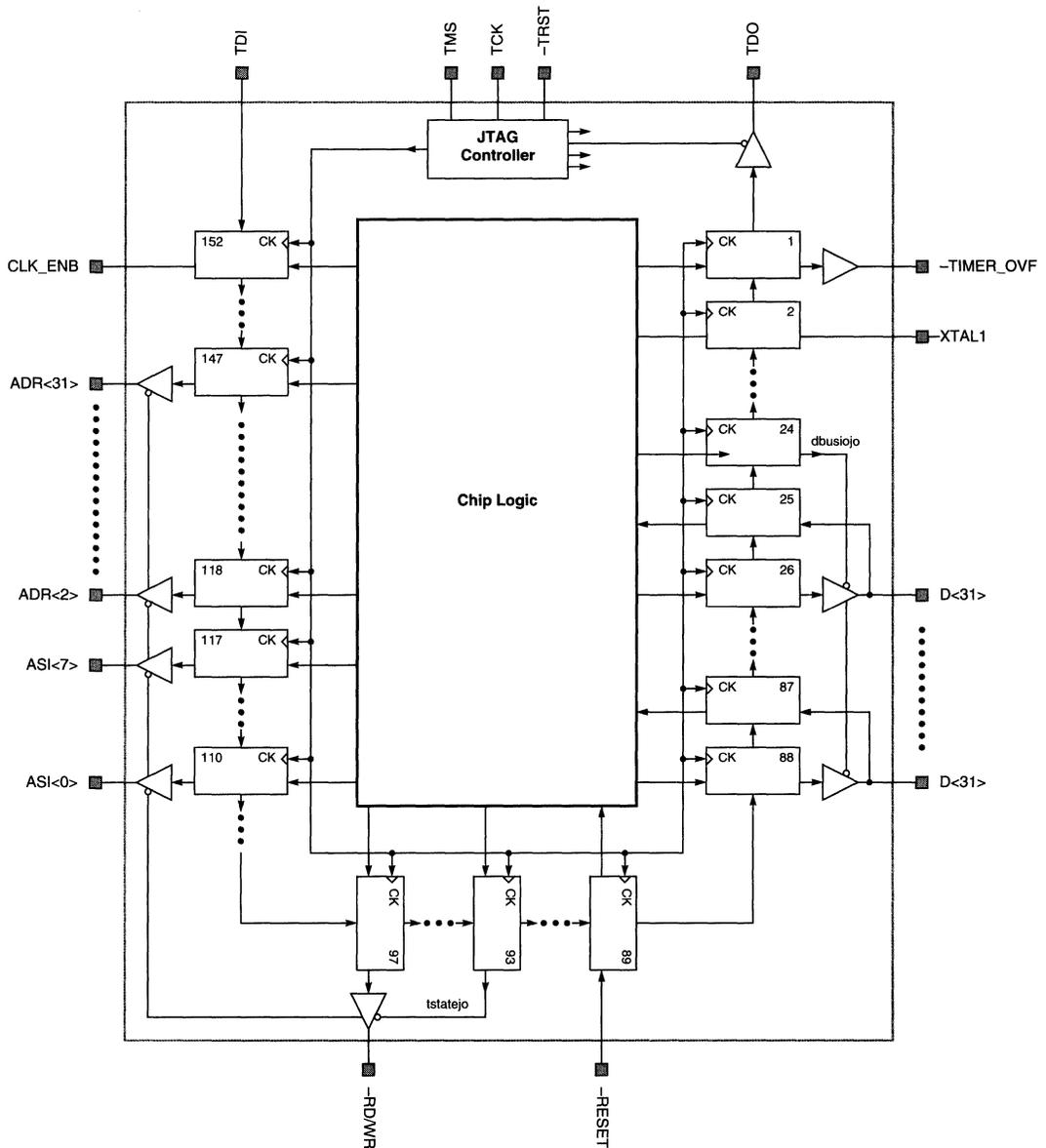


Figure 8-8. JTAG Cell Organization

8.6 MB86930 JTAG Pin List

The JTAG cells are arranged in a shift register configuration (see Figure 8-8). When shifting in a JTAG pattern through TDI, the LSB should correspond to the JTAG cell value for `-TIMER_OVF` pin whereas, the MSB of the pattern should correspond to the `CLK_ENB` pin's JTAG cell. As far as JTAG output through TDO is concerned, the first bit out corresponds to `-TIMER_OVF` JTAG cell value and the last output bit corresponds to the `CLK_ENB` JTAG cell value. Table 8-1 lists the order of all of the JTAG cells.

Table 8-1: JTAG Pin Order

Order	JTAG Cell	JTAG Cell Type	Function
1	<code>-TIMER_OVF</code>	output	Timer Overflow pin
2	<code>XTAL1</code>	input	Crystal input
3	<code>_EMU_BRK</code>	input	Emulator break input
4	<code>icediojo[†]</code>	output	EMU_D bus bidirectional control signal emudiojo = 1: EMU_D bus is input emudiojo = 0: EMU_D bus is output
5	<code>EMU_D_i<3></code>	input	Input bit 3 of EMU_SD<3:0> bus
6	<code>EMU_D_o<3></code>	output	Output bit 3 of EMU_SD<3:0> bus
7	<code>EMU_D_i<2></code>	input	Input bit 2 of EMU_SD<3:0> bus
8	<code>EMU_D_o<2></code>	output	Output bit 2 of EMU_SD<3:0> bus
9	<code>EMU_D_i<1></code>	input	Input bit 1 of EMU_SD<3:0> bus
10	<code>EMU_D_o<1></code>	output	Output bit 1 of EMU_SD<3:0> bus
11	<code>EMU_D_i<0></code>	input	Input bit 0 of EMU_SD<3:0> bus
12	<code>EMU_D_o<0></code>	output	Output bit 0 of EMU_SD<3:0> bus
13	<code>EMU_D_i<3></code>	input	Input bit 3 of EMU_D<3:0> bus
14	<code>EMU_D_o<3></code>	output	Output bit 3 of EMU_D<3:0> bus
15	<code>EMU_D_i<2></code>	input	Input bit 2 of EMU_D<3:0> bus
16	<code>EMU_D_o<2></code>	output	Output bit 2 of EMU_D<3:0> bus
17	<code>EMU_D_i<1></code>	input	Input bit 1 of EMU_D<3:0> bus
18	<code>EMU_D_o<1></code>	output	Output bit 1 of EMU_D<3:0> bus
19	<code>EMU_D_i<0></code>	input	Input bit 0 of EMU_D<3:0> bus
20	<code>EMU_D_o<0></code>	output	Output bit 0 of EMU_D<3:0> bus
21	<code>iceenblo[†]</code>	output	-EMU_ENB bus bidirectional control signal emuenblo = 1: -EMU_ENB bus is an input emuenblo = 0: -EMU_ENB bus is an output
22	<code>-EMU_ENB_i</code>	input	Input bit of -EMU_ENB pin
23	<code>-EMU_ENB_o</code>	output	Output bit of -EMU_ENB pin

Table 8-1: JTAG Pin Order (Continued)

Order	JTAG Cell	JTAG Cell Type	Function
24	dbusiojo [†]	output	D<31:0> bus bidirectional control signal dbusiojo = 1: D<31:0> bus is an input dbusiojo = 0: D<31:0> bus is an output
25	D_i<31>	input	Input bit 31 of D<31:0> bus
26	D_o<31>	output	Output bit 31 of <31:0> bus
:			:
87	D_i<0>	input	Input bit 0 of <31:0> bus
88	D_o<0>	output	Output bit 0 of <31:0> bus
89	–RESET	input	Chip reset pin
90	–BREQ	input	Bus request input
91	–MEXC	input	Memory exception input
92	–READY	input	External memory transaction complete signal
93	tstatejo [†]	output	Three-state control signal If tstatejo=1 then the following pins are three-stated. ADR<31:2>, ASI<7:0>, –BE<3:0>, –AS, RD/WR, –LOCK
94	–BGRNT	output	Bus grant output signal
95	–ERROR	output	Error output signal
96	–LOCK	output	Bus lock output signal
97	–RD/WR	output	Memory Read/Write output signal
98	–AS	output	Start of memory transaction output signal
99	–CS<0>	output	LSB of chip select output signal
:			:
104	–CS<5>	output	MSB of chip select output signal
105	–SAME_PAGE	output	Same-Page output signal
106	–BE<3>	output	Byte 3 enable output signal
:			:
109	–BE<0>	output	Byte 0 enable output signal
110	ASI<0>	output	LSB of ASI output pins
:			:
117	ASI<7>	output	MSB of ASI output pins
118	ADR<2>	output	LSB of Address output pins
:			:
147	ADR<31>	output	MSB of Address output pins

Table 8-1: JTAG Pin Order (Continued)

Order	JTAG Cell	JTAG Cell Type	Function
148	IRL<3>	input	MSB of interrupt request pin
:			:
151	IRL<0>	input	LSB of address output pins
152	CLK_ENB	input	PLL control pin. CLK_ENB=1: PLL on CLK_ENB=0: PLL off

†. These are internal I/O control signals. Therefore, there are no corresponding external pins.

1. The following pins are not three-statable: -SAME_PAGE, -CS<5:0>, -BGRNT, TIMER_OVF, -ERROR.

2. The following pins have no corresponding JTAG cells: CLKOUT1, CLKOUT2, XTAL2, -TRST, TCK, TMS, TDI, TDO.

Chapter A3: MB86931 Timers

- 3.1 Timer Registers.....A3-2**
 - 3.1.1 Prescaler Register..... A3-3
 - 3.1.2 Timer Control Registers (TCR)..... A3-4
 - 3.1.3 Reload Register..... A3-5
 - 3.1.4 Count Register A3-6
- 3.2 Prescaler OperationA3-6**
 - 3.2.1 Output Clock Duty Cycles A3-6
 - 3.2.2 Counter Loading A3-7
- 3.3 Timer OperationA3-7**
 - 3.3.1 In Signal A3-8
 - 3.3.2 Out Signal A3-9
 - 3.3.3 Starting and Stopping the Timer..... A3-10
 - 3.3.4 Timer Operating Modes..... A3-10

Chapter A4: MB86931 Serial Data Transmitters And Receivers

- 4.1 SDTR RegistersA4-2**
 - 4.1.1 Hidden Register Access A4-2
 - 4.1.2 SDTR Register Map A4-4
 - 4.1.3 Control Data Buffer Register A4-4
 - 4.1.4 Mode Register A4-5
 - 4.1.5 Command Register..... A4-7
 - 4.1.6 Synchronizing Character Registers A4-8
 - 4.1.7 Status Register A4-8
 - 4.1.8 Transmit Data Register A4-10
 - 4.1.9 Receive Data Register..... A4-10
- 4.2 Asynchronous Mode OperationA4-10**
 - 4.2.1 Operation Description A4-11
 - 4.2.2 Asynchronous Mode Timing A4-12
- 4.3 Synchronous Mode OperationA4-19**
 - 4.3.1 Operation Description A4-20
 - 4.3.2 Synchronous Mode Timing A4-21
- 4.4 Status Flag Operation And TimingA4-29**

Chapter A5: External Interface

5.1 Signals.....	A5-1
5.1.1 Processor Signals Descriptions	A5-2
5.1.2 Interrupt Request Signal Description	A5-6
5.1.3 Timer Signal Descriptions	A5-7
5.1.4 Serial Port Signal Descriptions.....	A5-7

Chapter A6: MB86931 JTAG

6.1 MB86931 JTAG Pin List	A6-1
--	-------------



Overview of MB86931

This section of the manual provides a functional description of the MB86931 with emphasis on the interrupt controller, timer, serial data transmitter/receiver, and signal descriptions. The core MB86930 processor is fully described in section 1 of this manual.

1.1 General Description

The MB86931 is a member of the SPARC_{lite} family whose function is a superset of the MB86930. The MB86931 features the high-performance MB86930 processor core combined with a 15-channel interrupt request controller (IRC), four additional independent 16-bit timers, and two independent serial data transmitters/receivers.

The processor is based on the SPARC architecture and is upward code compatible with previous processor implementations. On-chip data and instruction caches help decouple the processor from external memory latency. Separate on-chip instruction and data paths provide a high bandwidth interface between the IU and caches.

The interrupt request controller supports 15 maskable, prioritized interrupts. The system processor can program each interrupt channel to trigger in response to a high level, a low level, a rising edge, or a falling edge. The IRC latches the interrupt requests and asserts the encoded level number of the highest-priority inter-

rupt on the internal Interrupt Request Bus to interrupt the processor and identify the interrupt.

Four general-purpose timers can generate periodic interrupts and square waves, and feature two watchdog modes. They can be clocked by two prescalers, by external clocks, or by an internal MB86931 clock. A fifth timer in the MB86930 core generates an underflow signal, and is typically used for memory refresh timing.

The two Serial Data Transmitter and Receiver (SDTR) units support both synchronous and asynchronous modes, and are program-compatible with standard serial communication devices. They operate independently and can be clocked with the internal clock, with external clocks, or with clocks generated by the on-chip timers. Each SDTR supports the communication protocol and handshaking signals necessary for modem interface and control.

The following is a summary of the MB86931 features:

- 20 MHz (50 ns/cycle) operating frequency
- SPARC® high performance RISC processor
 - 2-way set associative instruction and data caches, 2 Kbytes each
 - Flexible cache data locking mechanism
 - Harvard architecture
 - 8 window, 136 word register file
 - Fast interrupt response time
 - 247 address spaces, 4 Gbyte each
 - User and supervisor modes
 - Buffered writes and instruction pre-fetching
 - Fast page-mode DRAM support
 - 16-bit DRAM refresh timer
 - Programmable address decoder and wait-state generator
 - On-chip clock generator circuit
 - JTAG test interface
 - Emulator support hardware
 - Single vector trapping
- 15-channel Interrupt Request Controller
 - Individual interrupt masks
 - Positive and negative level and edge trigger options for each channel
- Four independent general-purpose 16-bit timers
 - Prescalers for two timers
 - Five modes of operation for each timer

- Two Serial Data Transmitter and Receiver Units
 - Compatible with the MB89251
 - Asynchronous and synchronous operation
 - 5 to 8 bit character length selection
 - Parity bit option
 - Internal or external synchronous mode options
 - One (MONOSYNCH) or two (BISYNC) synchronous character options
- 0.8 micron gate CMOS technology.

1.2 Programmer's Model of the MB86931

The MB86931 contains all of the registers that are defined in the MB86930 processor. The chip also contains IRC, general-purpose timer, and SDTR control and status registers.

All registers (except ASR registers) on the MB86931 are read with the LOAD ALTERNATE (LDA) instruction, and written with the STORE ALTERNATE (STA) instruction. Only loads and stores to word addresses are supported for the MB86931-specific registers. Reserved register fields are undefined when read, and should be written 0.

The following are listings of the MB86931 registers and their addresses. The registers are grouped according to function. All addresses are word addresses.

Cache/BIU Control and Status Registers:

ASI: 0x01

Address range: 0x00000000 - 0x000000FF

0x00000000	ASI=0x1	Cache/BIU Control Register
0x00000004	ASI=0x1	Lock Control Register
0x00000008	ASI=0x1	Lock Control Save Register
0x0000000C	ASI=0x1	Cache Status Register
0x00000010	ASI=0x1	Restore Lock Control Register
0x00000080	ASI=0x1	System Support Control Register

Peripheral control and status registers:

ASI: 0x01

Address range: 0x00000100 - 0x000001FF

0x00000120	ASI=0x1	Same Page Mask Register
0x00000124	ASI=0x1	Address Range Specifier Register 1
0x00000128	ASI=0x1	Address Range Specifier Register 2
0x0000012C	ASI=0x1	Address Range Specifier Register 3
0x00000130	ASI=0x1	Address Range Specifier Register 4
0x00000134	ASI=0x1	Address Range Specifier Register 5
0x00000140	ASI=0x1	Address Mask Register 0
0x00000144	ASI=0x1	Address Mask Register 1
0x00000148	ASI=0x1	Address Mask Register 2
0x0000014C	ASI=0x1	Address Mask Register 3
0x00000150	ASI=0x1	Address Mask Register 4
0x00000154	ASI=0x1	Address Mask Register 5
0x00000160	ASI=0x1	Wait State Specifier Register
0x00000164	ASI=0x1	Wait State Specifier Register
0x00000168	ASI=0x1	Wait State Specifier Register
0x00000174	ASI=0x1	Timer Register (MB86930 core)
0x00000178	ASI=0x1	Timer Preload Register (MB86930 core)

MB86931-specific control and status registers:

ASI: 0x01

Address range: 0x00000200 - 0x000002FF

0x00000200	ASI=0x1	IRC Trigger Mode Register 0
0x00000204	ASI=0x1	IRC Trigger Mode Register 1
0x00000208	ASI=0x1	IRC Request Sense Register
0x0000020C	ASI=0x1	IRC Request Clear Register
0x00000210	ASI=0x1	IRC Mask Register
0x00000214	ASI=0x1	IRC IRL Latch/Clear Register
0x00000220	ASI=0x1	STDR0 Transmit/Receive Data Register
0x00000224	ASI=0x1	STDR0 Control/Status Register
0x00000230	ASI=0x1	STDR1 Transmit/Receive Data Register
0x00000234	ASI=0x1	STDR1 Control/Status Register
0x00000240	ASI=0x1	Timer 0 Prescaler Register
0x00000244	ASI=0x1	Timer 0 Control Register
0x00000248	ASI=0x1	Timer 0 Reload Register
0x0000024C	ASI=0x1	Timer 0 Count Register
0x00000250	ASI=0x1	Timer 1 Prescaler Register
0x00000254	ASI=0x1	Timer 1 Control Register
0x00000258	ASI=0x1	Timer 1 Reload Register
0x0000025C	ASI=0x1	Timer 1 Count Register
0x00000264	ASI=0x1	Timer 2 Control Register
0x00000268	ASI=0x1	Timer 2 Reload Register
0x0000026C	ASI=0x1	Timer 2 Count Register
0x00000274	ASI=0x1	Timer 3 Control Register
0x00000278	ASI=0x1	Timer 3 Reload Register
0x0000027C	ASI=0x1	Timer 3 Count Register

Emulation Registers:

ASI: 0x01

Address range: 0x0000FF00 - 0x0000FFFF

0x0000FF00	ASI=0x1	Instruction Address Descriptor Register 1
0x0000FF04	ASI=0x1	Instruction Address Descriptor Register 2
0x0000FF08	ASI=0x1	Data Address Descriptor Register 1
0x0000FF0C	ASI=0x1	Data Address Descriptor Register 2
0x0000FF10	ASI=0x1	Data Value Descriptor Register 1
0x0000FF14	ASI=0x1	Data Value Descriptor Register 2 or Mask Register
0x0000FF18	ASI=0x1	Debug Control Register
0x0000FF1C	ASI=0x1	Debug Status Register

Instruction Cache Lock Registers:

ASI: 0x02

Address range: 0x00000000 - 0x000003F0 (Bank 1)
0x80000000 - 0x800003F0 (Bank 2)

Note: The lock bit for each line in the instruction cache can be initialized by writing to every fourth word address in this space.

Data Cache Lock Registers:

ASI: 0x03

Address range: 0x00000000 - 0x000003F0 (Bank 1)
0x80000000 - 0x800003F0 (Bank 2)

Note: The lock bit for each line in the instruction cache can be initialized by writing to every fourth word address in this space.

Instruction Cache Tag RAM:

ASI: 0x0C

Address range: 0x00000000 - 0x000003F0 (Bank 1)
0x80000000 - 0x800003F0 (Bank 2)

Note: The tag for each line in the instruction cache can be initialized by writing to every fourth word address in this space.

Instruction Cache Data RAM:

ASI: 0x0D

Address range: 0x00000000 - 0x000003FF (Bank 1)
0x80000000 - 0x800003FF (Bank 2)

Note: The instruction cache can be initialized by writing to word addresses in this space.

Data Cache Tag RAM:

ASI: 0x0E
Address range: 0x00000000 - 0x000003FF (Bank 1)
0x80000000 - 0x800003FF (Bank 2)
Note: The tag for each line in the data cache can be initialized by writing to every fourth word address in this space.

Data Cache Data RAM:

ASI: 0x0F
Address range: 0x00000000 - 0x000003FF (Bank 1)
0x80000000 - 0x800003FF (Bank 2)
Note: The instruction cache can be initialized by writing to word addresses in this space.

1.3 Internal Architecture of the MB86931

The MB86931 is an integration of the MB86930 SPARClite RISC processor and the MB86940 SPARClite Companion Chip. The chip consists of a processor core ported from the MB86930 processor and peripheral logic ported from the MB86940 SPARClite Companion Chip. Figure A1-1 shows a block diagram of the MB86931; Figure A1-2 shows a detailed block diagram of the MB86931 peripheral logic.

The processor core Integer Unit supports a superset of the SPARC instruction set. The separate instruction and data caches maximize processor throughput. The Bus Interface unit interfaces the processor to the system. The Clock Generator with integrated phase-lock loop simplifies system clock design. The Debug Support Unit supports in-circuit emulation.

The peripheral logic Interrupt Request Controller (IRC) generates prioritized interrupt levels for as many as 15 interrupts. The timers allow square-wave generation and support watchdog functions. The SDTRs (serial data transmitters/receivers) allow serial communication using standard communication protocols.

Separate internal data and instruction buses connect the functional units. A unified external data bus and a unified external address bus extend off-chip to interface the functional units to memory and I/O.

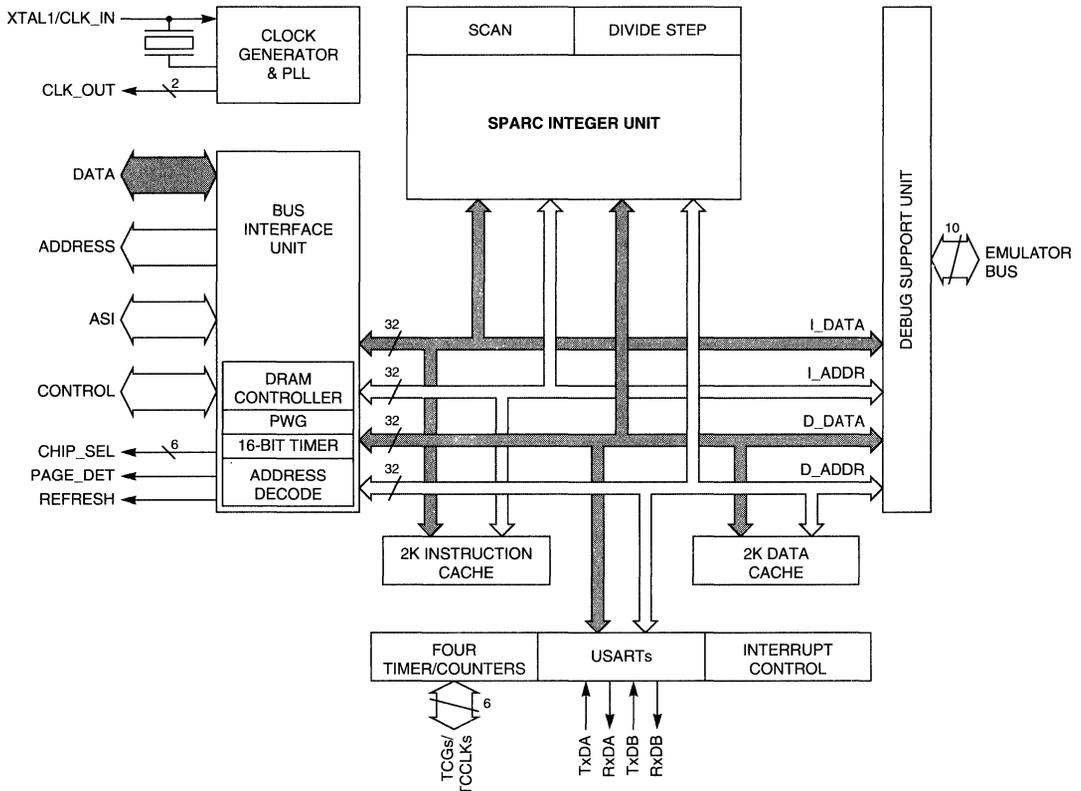


Figure A1-1. MB86931 Block Diagram

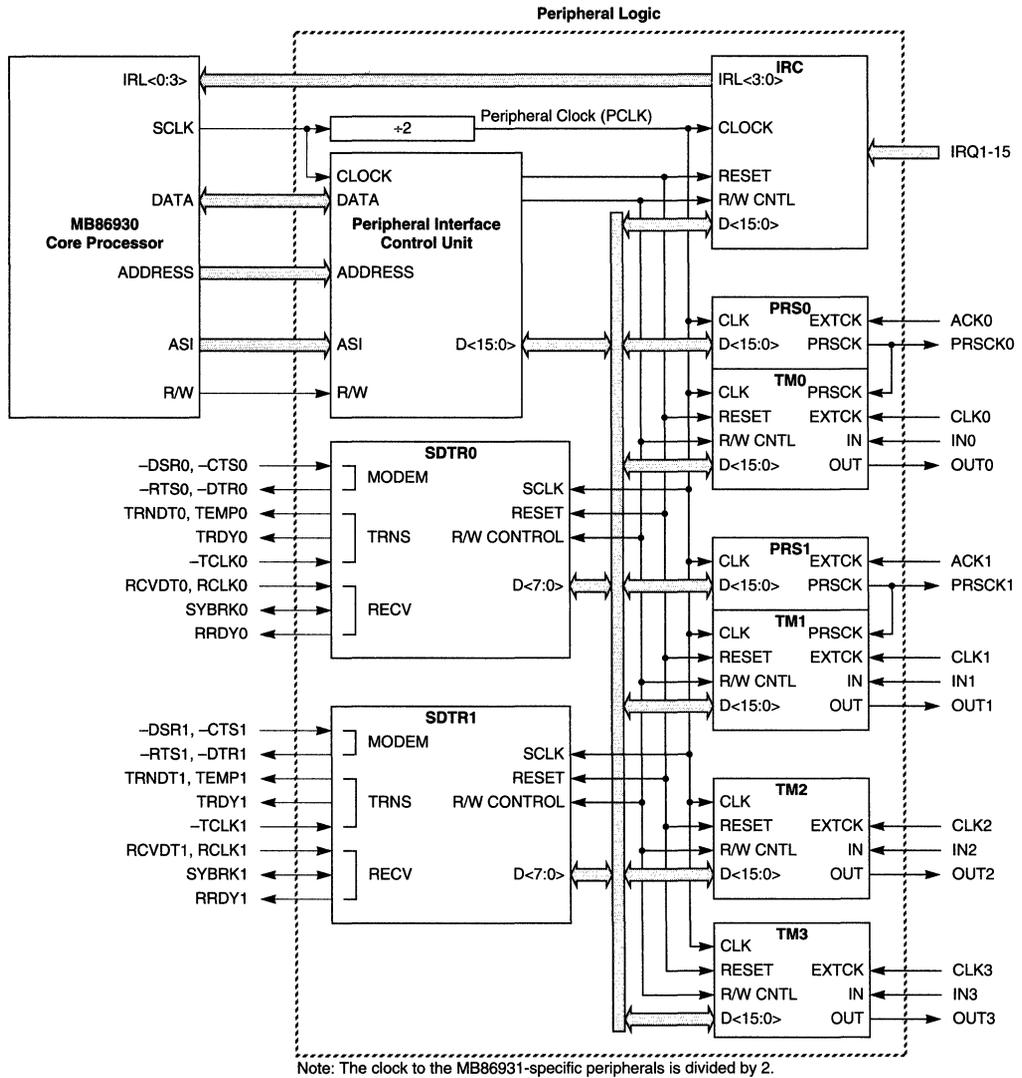


Figure A1-2. Peripheral Logic Block Diagram

The IRL Mask logic allows masking of all interrupt requests by forcing the interrupt level asserted on IRL<3:0> to 0. The processor can still poll for pending interrupts by reading the Request Sense register even if the interrupt level is masked. The processor controls interrupt level masking by writing to the Mask register.

2.1 IRC Registers

The IRC features six internal registers that allow the processor to control IRC operation and to monitor system interrupt requests that may be pending. Register addressing is shown in Table A2-1.

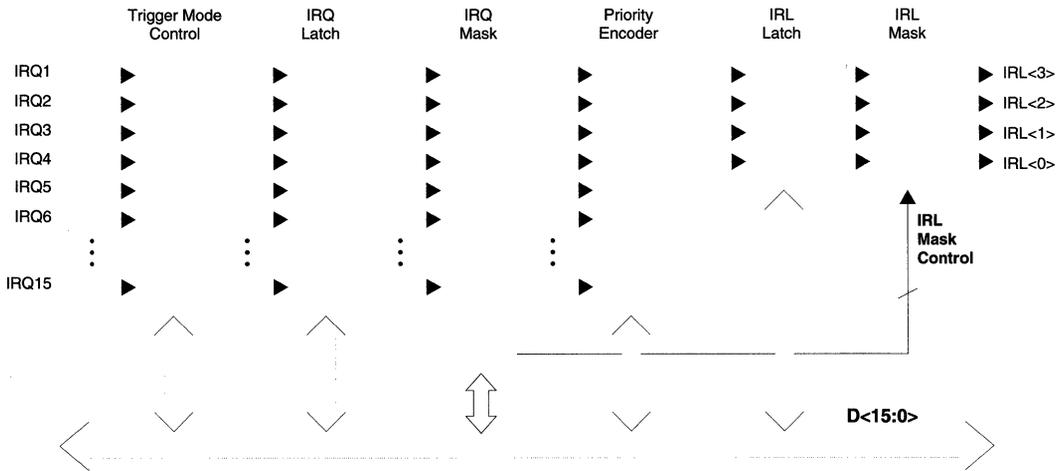


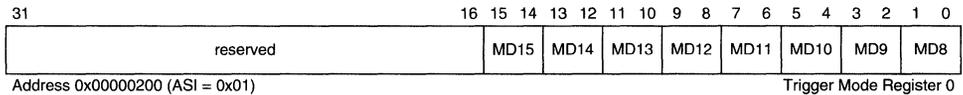
Figure A2-1. IRC Block Diagram

Table A2-1: IRC Register Map

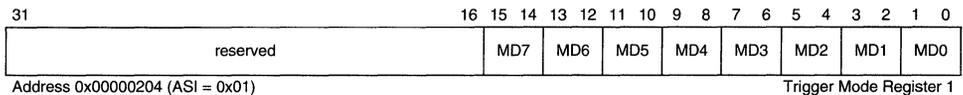
Address	Register	Access
0x00000200	Trigger Mode 0	W
0x00000204	Trigger Mode 1	W
0x00000208	Request Sense	R
0x0000020C	Request Clear	W
0x00000210	Mask	W
0x00000214	IRL Latch/Clear	R/W

2.1.1 Trigger Mode Registers

The Trigger Mode registers control the trigger mode for each interrupt channel. Trigger Mode Register 0 controls trigger modes for interrupt channels 8-15; Trigger Mode Register 1 controls trigger modes for interrupt channels 1-7.



Bits 15-0: Trigger Mode Selects - Select trigger modes for channels 8-15.



Bits 15-2: Trigger Mode Selects - Select trigger modes for channels 1-7.

Bits 1-0: Reserved.

Two-bit fields in the registers select one of four trigger modes for each channel as follows:

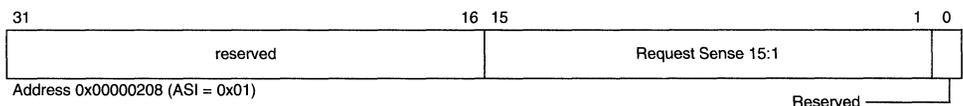
MDx Value*	Trigger Mode
0	High Level
1	Low Level
2	Rising Edge
3	Falling Edge

Reset clears the Trigger Mode registers, resulting in high level triggering for each interrupt channel.

Note: An interrupt channel should be masked before its trigger mode is changed, or a false interrupt may occur.

2.1.2 Request Sense Register

The processor reads the state of the IRQ Latch through the Request Sense register to identify pending interrupts.



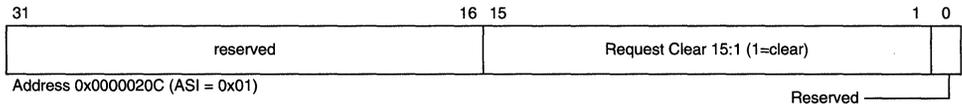
Bits 15-1: Sense IRQ Latch - Correspond to interrupt channels 15-1 and indicate, when high, that the corresponding interrupts are latched and pending.

Bit 0: Reserved.

Reset clears the Request Sense Register.

2.1.3 Request Clear Register

The processor writes to the Request Clear register to clear the IRQ Latch. The processor typically uses this register to clear the latch associated with an interrupt when it services the interrupt.



Bits 15-1: Clear IRQ Latch - Correspond to interrupt channels 15-1, and writing the bits to 1 clears the corresponding interrupt latches.

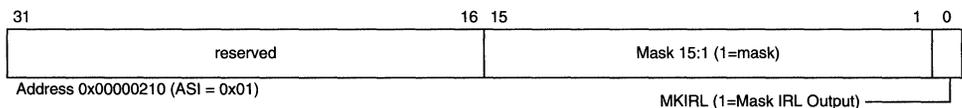
Bit 0 Reserved.

Reset clears the Request Clear Register.

Note: The processor should clear the latch associated with an interrupt following a change in its trigger mode, or a false interrupt may occur.

2.1.4 Mask Register

The Mask register is used to mask the outputs of the IRQ Latch from the Priority Encoder, and the output of the IRL latch from the IRL<3:0> bus. The processor uses the Mask register to mask unused interrupt channels, to temporarily mask individual interrupt requests, and to mask all interrupt requests.



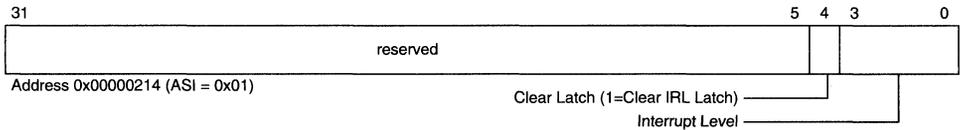
Bits 15-1: Interrupt Request Mask - Correspond to interrupt channels 15-1, and writing them to 1 masks the corresponding interrupt requests.

Bit 0: Mask IRL - Masks the output of the IRL Latch. When MKIRL is set to 1, the IRL Latch output is masked, and the IRL<3:0> bus is forced to 0. When MKIRL is 0, the encoded interrupt level number in the IRL latch is asserted on the IRL<3:0> bus to interrupt the processor. MKIRL is typically set to 1 (mask enabled) in systems that poll interrupt requests.

Reset clears the Mask register.

2.1.5 IRL Latch/Clear Register

The processor uses the IRL Latch/Clear register to clear and read the IRL Latch.



Bit 4: Clear IRL Latch - Clears the IRL Latch when written to 1.

Bits 3:0: Interrupt Level - Holds the value of the IRL Latch. The processor typically reads IRL to identify the highest-priority interrupt level in systems that poll the interrupts.

Reset clears the IRL Latch/Clear Register.

2.2 IRC Operation

The IRC latches interrupt requests into the IRQ Latch according to the trigger mode option selected for each interrupt channel. The Priority Encoder prioritizes the unmasked interrupts and generates an encoded interrupt level number for the highest-priority interrupt. The IRL Latch latches the encoded interrupt level number, which is then transferred through the IRL Mask logic to the IRL<3:0> bus to interrupt the processor. The processor responds by servicing the interrupt identified on IRL<3:0>, clearing the latched interrupt from the IRQ Latch through the IRL Latch/Clear register and clearing the IRL latch. The IRC then generates a new level number for the highest-priority interrupt that may be latched in the IRQ Latch.

The interrupt request latency is ten system clock cycles. That is, the corresponding interrupt level is asserted on IRL<3:0> ten clock cycles after an interrupt request is recognized by the IRC.

2.2.1 Polling

The processor can poll interrupts by reading either the IRQ Latch via the Request Sense register, or the IRL Latch via the IRL Latch/Clear register.

The processor may mask interrupts that it polls via the Request Sense register by masking either the IRQ Latch or the IRL Latch. The processor then periodically reads the IRQ Latch and clears interrupts from the latch when they are serviced. The IRL Latch may remain unmasked to allow interrupt-driven servicing of some interrupts if the polled interrupts are masked with the IRQ Latch mask.

The processor may mask all interrupts when it polls interrupts via the IRL Latch/Clear register by masking the IRL Latch. The processor then periodically reads

the IRL Latch for the highest-level pending interrupt and clears both the IRL Latch and the interrupt from the IRQ Latch once the interrupt is serviced.

2.2.2 Initialization

All IRC registers are cleared to 0 by Reset. This results in high-level trigger mode for all interrupts, and all masks disabled.

After reset, the interrupt trigger modes should be changed after the interrupts are masked with the IRQ mask to eliminate false interrupts. The masks can then be disabled.

2.2.3 Noise Immunity

The IRQ pins are sampled at the rising edge of the IRC internal clock. The pin value must be verified by three successive samples for recognition by the IRC. For example, a level trigger must be asserted for at least two internal clock periods (four system clock periods) for recognition.

Figure A2-2 shows the IRQ pin sample timing.

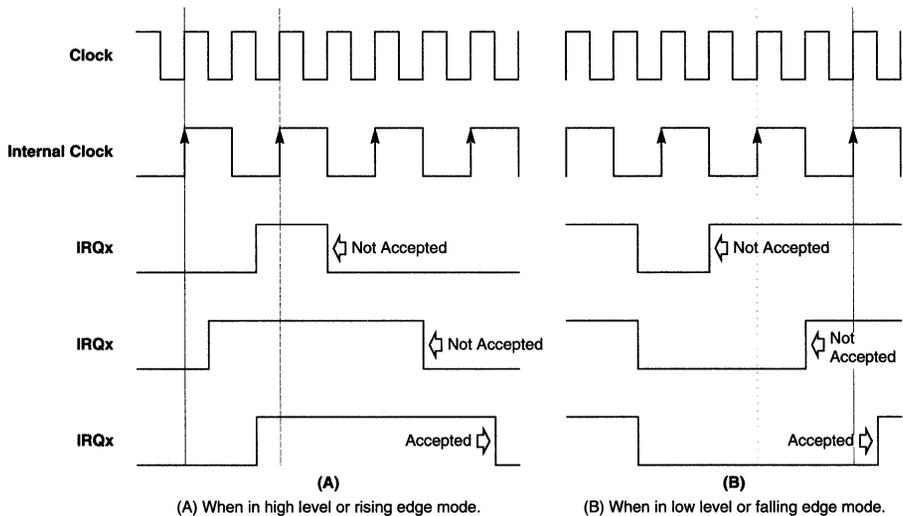


Figure A2-2. -IRQ Pin Sample Timing



MB86931 Timers

The MB86931 features four independent general-purpose 16-bit timers. Each timer can be independently programmed to operate in one of the following five modes:

- Mode 0 - Periodic Interrupt Mode
- Mode 1 - Timeout Interrupt Mode
- Mode 2 - Square Wave Generator Mode
- Mode 3 - Software Trigger Watchdog Mode
- Mode 4 - External Trigger Watchdog Mode.

The peripheral clock (PCLK) is an internal MB86931 clock that operates at one half the frequency of the processor clock.

Timer 0 and Timer 1 have clock prescalers that can be independently clocked by PCLK, or by asynchronous external clocks (ACKx). The timers themselves can be independently clocked by PCLK, by the prescaler output clock (PRSCKx), or by an external asynchronous clock (CLKx).

Timer 2 and Timer 3 have no clock prescalers but can be clocked by PCLK, or by external asynchronous clocks.

Figure A3-1 shows a block diagram of the timers and prescalers and their clock options. The external prescaler input clocks are labeled ACKx, the prescaler output clocks are labeled PRSCKx, the internal clock is labeled PCLK, and the exter-

nal timer clocks are labeled CLKx. Note that the asynchronous external clocks are synchronized internally with the peripheral clock.

3.1 Timer Registers

Each timer has a Timer Control register, a Reload register, and a Count register for timer configuration and control. Timer 0 and Timer 1 also have Prescaler registers for prescaler control. Table A3-1 shows the timer register map.

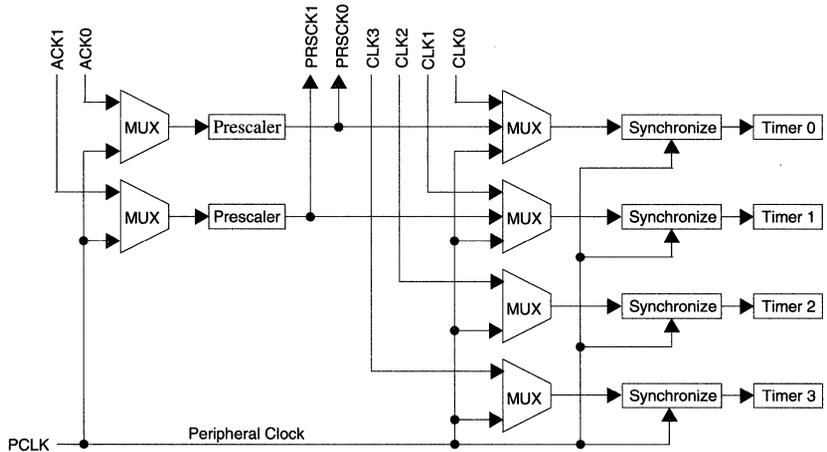


Figure A3-1. Timer Prescaler Block Diagram

Table A3-1: Timer Register Map

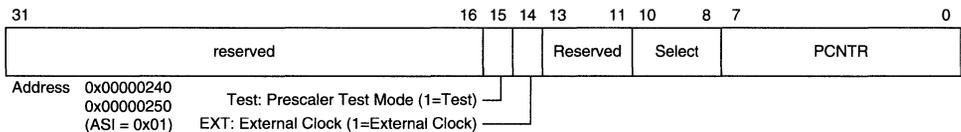
Address	Functional Unit	Register Name	Access	Reset State
0x0000240	Prescaler 0	Prescale Register 0	R/W	0x01
0x0000244	Timer 0	Timer Control Register 0	R/W	0
0x0000248		Reload Value 0	R/W	0
0x000024C		Count Value 0	R	0
0x0000250	Prescaler 1	Prescale Register 1	R/W	0x01
0x0000254	Timer 1	Timer Control Register 1	R/W	0
0x0000258		Reload Value 1	R/W	0
0x000025C		Count Value 1	R	0
0x0000260	Reserved	*****	-	-
0x0000264	Timer 2	Timer Control Register 2	R/W	0
0x0000268		Reload Value 2	R/W	0
0x000026C		Count Value 2	R	0

Table A3-1: Timer Register Map (Continued)

Address	Functional Unit	Register Name	Access	Reset State
0x00000270	Reserved	*****	-	-
0x00000274	Timer 3	Timer Control Register 3	R/W	0
0x00000278		Reload Value 3	R/W	0
0x0000027C		Count Value 3	R	0

3.1.1 Prescaler Register

The Prescaler register allows selection of the prescaler clock, the prescaler output, and the prescaler counter value.



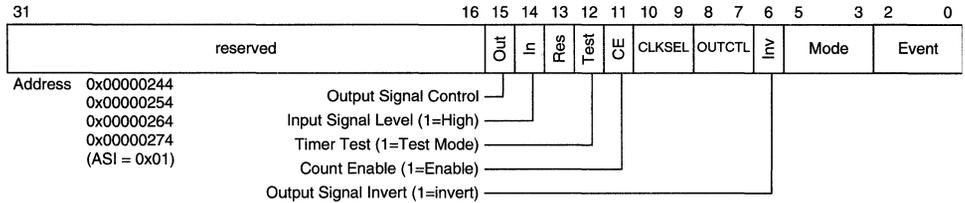
- Bit 15: External Clock - Selects the prescaler clock source as follows:
 0:PCLK.
 1:External clock.
- Bit 14: Prescaler Test Mode - Set to 1 for testing. The prescaler test mode is intended for factory use only, and Test should therefore remain 0 during normal operation.
- Bits 13-11: Reserved.
- Bits 10-8: Prescaler Output Select - Selects one of the eight prescaler outputs for PRSCKx, the prescaler clock output. Each selection is one half the frequency of the previous selection. A 0 in this field selects the prescaler counter output; a 1 selects one half the frequency of the prescaler counter output, etc.
- Bits 7-0: Prescaler Counter Value - Determines the prescaler counter output frequency. The value in this field is loaded into the prescaler counter when it is written, and when timeout occurs. The prescaler counter value must be 1 or greater; a value of 1 forces the prescaler output clock (PRSCKx) low.

Reset initializes the Prescaler registers to 0x01. This initial state selects internal prescaler clock, the highest prescaler output clock frequency, and a Prescaler value of 1 (PRSCK forced low).

The reserved fields should be written "0" for future software compatibility.

3.1.2 Timer Control Registers (TCR)

The TCR enables and disables the timer and allows selection and control of the timer In and Out signals, clock sources, and operation modes.



- Bit 15: Output Signal Level - A read-only status bit for reading the current Out signal level. When the Out signal level is high, the OUT status bit is 1.
- Bit 14: Input Signal Level - A read-only status bit for reading the current In signal level. When the In signal level is high, the In status bit is 1.
- Bit 13: Reserved.
- Bit 12: Timer Test Mode - Set to 1 for testing. The timer test mode is intended for factory use only, and should therefore remain 0.
- Bit 11: Count Enable - Enables the timer when set to 1; disables the timer when cleared to 0. The timer and its prescaler should be configured for desired operation when the timer is enabled.
- Bits 10-9: Clock Select - Selects the timer clock source as follows:

CLKSEL	Clock Source
0	Internal Clock
1	External Clock
2	Prescaler Output Clock (Timers 0 and 1 only)
3	Reserved

The external and prescaler clocks are synchronized with the internal clock before being applied to the timer.

Caution: The external clock frequency must be no higher than 1/3 of the peripheral clock (PCLK) frequency.

- Bits 8-7: Out Signal Control - Selects the state of the Out pin while the timer is stopped as follows:

OUTCTL	Out State
0	Remains in the current state
1	Asserted high
2	Asserted low
3	Reserved.

Bit 6: Invert - Inverts the timer Out signal when set to 1.

Bits 5-3: Mode Select - Selects the timer mode of operation as follow:

Mode	Timer Operating Mode
0	Periodic Interrupt Mode
1	Timeout Interrupt Mode
2	Square Wave Generator Mode
3	Software Trigger Watchdog Mode
4	External Trigger Watchdog Mode
5-7	Reserved

Bits 2:0: Event Select - Selects the timer event gate or trigger as follow:

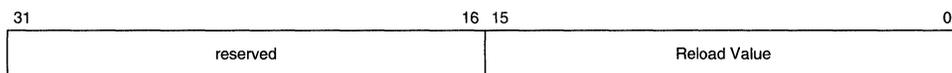
Event	Gate or Trigger	Applicable Modes
0	Low Level Gate	0, 1, 2
1	High Level Gate	0, 1, 2
2	Rising Edge Trigger	4
3	Falling Edge Trigger	4
4	Rising and Falling Edge Triggers	4

The gate or trigger is the In signal.

Reset initializes the Timer Control register to 0. The reserved fields should be written "0" for future software compatibility.

3.1.3 Reload Register

The Reload register holds the initial value of the timer counter.



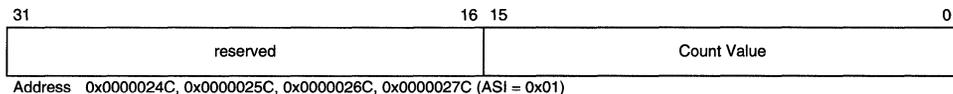
Address 0x00000248, 0x00000258, 0x00000268, 0x00000278 (ASI = 0x01)

Bits 15-0: Timer Reload Value - In Modes 0 and 2, the Timer Reload Value is automatically loaded into the counter when a timeout occurs. In Mode 2, the Timer Reload Value is compared with the Count Register value to control the Out signal.

Reset initializes the Reload register to 0. The reserved field should be written "0" for future software compatibility.

3.1.4 Count Register

The Count register is a read-only register that holds the current timer counter value.



Bits 15-0: Timer Count Value - The current timer count value.

Reset initializes the Count register to 0.

3.2 Prescaler Operation

Figure A3-2 shows a prescaler block diagram consisting of an 8-bit counter, cascaded divide-by-two flip-flops, and selector logic.

Once the prescaler counter is loaded, the counter decrements at its clocked frequency and generates an output to the cascaded flip-flops. The flip-flops successively divide by two to provide eight frequencies for selection by the selector logic. The selector logic selects the output of the counter or one of the divided outputs as the prescaler clock output according to the value in the Prescaler register Select field. The clock output, PRSCKx, may be used to clock the timer, and is available for external use at the PRSCKx package pin.

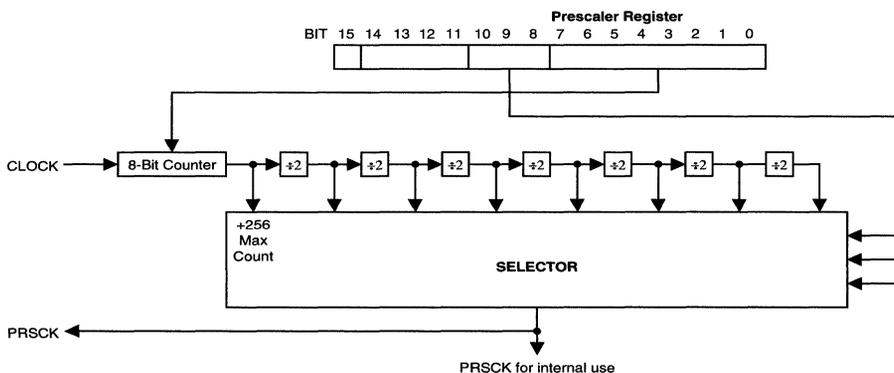


Figure A3-2. Prescaler Block Diagram

3.2.1 Output Clock Duty Cycles

The clocks generated by the cascaded flip-flops have 50% duty cycles when selected with 1-7 in the Prescaler register Select field.

The clock generated directly by the prescaler counter, selected with 0 in the Prescaler Select field, is not a 50% duty cycle clock. The clock is asserted high until the counter reaches 1, and is then asserted low for one internal clock cycle. The clock is then asserted back to the high level while the counter reloads and counts down to 1 again. The clock is therefore low for one internal clock cycle during the count-down period.

The timer operation is independent of the prescaler clock duty cycle.

3.2.2 Counter Loading

The 8-bit prescaler counter is loaded with the value in the Prescaler Register PCNTR field in three ways as follows:

1. When the 8-bit prescaler counter decrements to 0.
2. By writing to the PCNTR field.
3. When the timer reload value is loaded or reloaded into the companion timer if the timer is clocked by the prescaler output clock, and the prescaler is clocked by PCLK. CLKSEL must be 2 in the companion timer's Timer Control register (prescaler output clock selected to clock the timer) and Ext must be 0 in the Prescaler register for this to occur.

The cascaded flip-flops in the divide chain are cleared when the prescaler counter is loaded.

When the prescaler is operating in the external clock mode, a new counter value written into the Prescaler register PCNTR field is not loaded into the Prescaler counter until the next rising edge of the PRSCKx prescaler clock output. The prescaler should therefore be changed to internal clock mode before writing the PCNTR field to minimize latency in loading the counter.

3.3 Timer Operation

Figure A3-3 shows a block diagram of a timer. Each timer is identical, but only Timer 0 and Timer 1 have prescaler clock sources.

Timer 0 and Timer 1 can be clocked with the internal clock, an external clock, or a prescaler clock. Timer 2 and Timer 3 can be clocked with the internal clock or with an external clock. Timer clock selection is controlled by the CLKSEL field in the TCR.

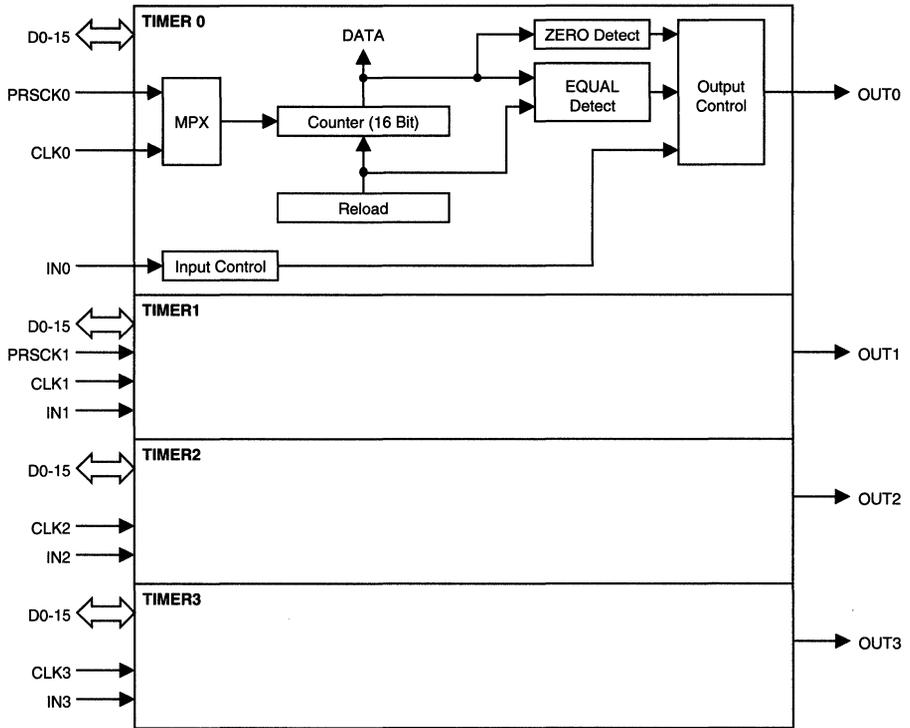


Figure A3-3. Timer Block Diagram

3.3.1 In Signal

The In signal can be used as a gating signal (i.e. to temporarily stop the timer by masking the timer clock) in Modes 0, 1, 2 and 3, and as a trigger event for counter operation in Mode 4.

To use the In signal as a gating signal in Modes 0, 1, 2 and 3, the In signal active level (low level or high level gate) is selected in the TCR Event Select field. The In signal level must be asserted at the active level during an entire clock cycle to mask the clock.

Figure A3-4 shows In signal gate timing.

When using the In signal as a triggering signal in Mode 4, the event field in the TCR determines the In signal event to be a rising edge, a falling edge, or both a rising edge and a falling edge. When the In signal triggers the timer, the Out signal is asserted (to low if the Inv bit in the TCR is 0; to high if the Inv bit in the TCR

is 1), and count down from the value in the Reload Value register begins. At time-out, the Out signal is set if INV = 0 (and reset if INV = 1).

Figure A3-5 shows In signal trigger timing.

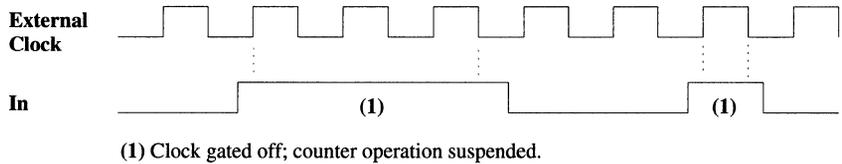


Figure A3-4. In Signal Gate Timing

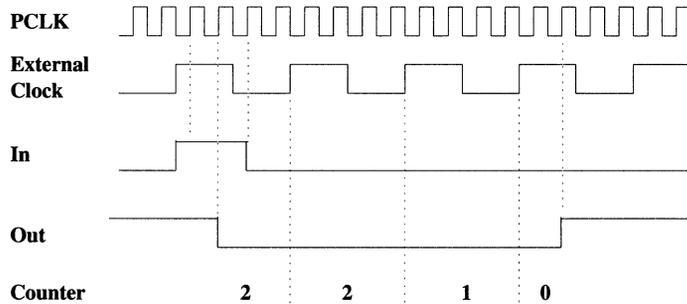


Figure A3-5. In Signal Trigger Timing (Rising Edge Trigger)

3.3.2 Out Signal

The Out signal is used to indicate timeout, the occurrence of an event at the In pin, or the half-value of the Reload Value register during countdown. The Out signal active level is controlled by the Out Signal Control field and the Invert control bit in the TCR. The Out Signal Control field controls the state of the Out signal while the timer is stopped. The Invert bit inverts the Out signal when set to 1.

The Out signal is typically tied to an interrupt request controller to generate a processor interrupt at timeout in Modes 0, 1, 3, and 4, and is used as a square wave in Mode 2.

The following are the conditions for resetting and setting the Out signal level for the various timer modes during timer operation, with the TCR Invert bit cleared to 0:

Mode	Out Signal Reset	Out Signal Set
0	Writing Reload Register; Reading Count Value Register	Timeout
1	Writing Reload Register; Reading Count Value Register	Timeout
2	When the half-value of the Reload Register is reached.	Timeout
3	Writing the Reload Register	Timeout
4	When a trigger event occurs at the In pin	Timeout

The Out signal is inverted when the Invert bit is set to 1.

3.3.3 Starting and Stopping the Timer

The timers are stopped following reset. Timer operation is initiated in all modes by first writing the timer mode in the TCR Mode field and setting the Count Enable control bit in the TCR to 1.

Timer operation in Modes 0, 1, 2, and 3 begins when the Reload register is written. The Reload register value is transferred to the timer counter when the Reload register is written, and the counter begins decrementing.

Timer operation in Mode 4 begins when a trigger event occurs at the In pin. The Reload register value is transferred to the timer counter when the trigger event occurs, and the counter begins decrementing.

Once operating, each timer is stopped in the various operating modes as follows:

- Modes 0: Writing the TCR or active In gate.
- Mode 1: Writing the TCR or active In gate or timeout
- Mode 2: Writing the TCR or active In gate.
- Mode 3: Writing the TCR or active In gate or timeout.
- Mode 4: Writing the TCR or timeout.

Note that the timers can be halted in all operating modes by writing to the TCR.

In gate timer control is described in Section 3.3.1; timer starting and halting is summarized in Table A3-2.

3.3.4 Timer Operating Modes

Each timer supports five operating modes: periodic interrupt mode (Mode 0), timeout interrupt mode (Mode 1), square wave generator mode (Mode 2), soft-

ware trigger watchdog mode (Mode 3), and external trigger watchdog mode (Mode 4). The timer operating mode is controlled by the Mode field in the TCR.

Periodic Interrupt Mode (Mode 0)

The Out signal is initially set to the high or low state, depending on the OUTCTL field in the TCR. The timer is enabled (CE=1) and the mode selected. The counter then begins decrementing and the Out signal is driven low when the Reload register is written with the reload value.

When timeout occurs (counter = 0), the timer Out signal transitions to the high level. The Reload register value loads into the counter at timeout, and the counter continues decrementing. The Out signal remains at the high level until the Counter register is read or the Reload register is written.

The Out levels are inverted if Inv = 1 in the TCR.

Timeout Interrupt Mode (Mode 1)

This mode differs from Mode 0 at timeout. In Mode 1, the timer halts at timeout instead of reloading and decrementing the counter.

The Out signal is initially set to the high or low state, depending on the OUTCTL field in the TCR. The timer is enabled (CE=1) and the mode selected. The counter then begins decrementing and the Out signal is driven low when the Reload register is written with the reload value.

When timeout occurs (counter = 0), the timer Out signal transitions to the high level, and the counter halts. The Out signal remains at the high level and the counter remains halted until the Count register is read or the Reload register is written. When the Count register is read or the Reload register is written, the Out signal is asserted low, the Reload register value loads into the timer counter, and the counter decrements.

The Out levels are inverted if Inv = 1 in the TCR.

Square Wave Generator Mode (Mode 2)

This mode differs from Mode 0 in the transition of the Out signal.

The Out signal is initially set to the high or low state, depending on the OUTCTL field in the TCR. The timer is enabled (CE=1) and the mode selected. The counter then begins decrementing when the Reload register is written with the reload value.

When the counter decrements to half of the reload value, the Out signal is driven to the low level. When timeout occurs (counter = 0), the timer Out signal transi-

tions to the high level. The counter reloads at timeout, and continues decrementing, repeating the Out level changes. The Out signal is therefore a square wave.

The following are the square wave high and low times for various Reload register values represented by "N":

N	Period (N+1)	High Level (N+1)/2+1	Low Level N/2
0	-	-	
1	2	1	
2	3	2	1
3	4	3	1
4	5	3	2
5	6	4	2
6	7	4	3

For $N \geq 2$, the period of the square wave is $N+1$, the high level width is $(N+1)/2+1$, and the low level is $N/2$. $N = 0$ and $N = 1$ are special cases, as shown in the table.

The Out levels are inverted if $Inv = 1$ in the TCR.

Software Trigger Watchdog Mode (Mode 3)

The Out signal is initially set to the high or low state, depending on the OUTCTL field in the TCR. The timer is enabled ($CE=1$) and the mode selected. The counter then begins decrementing and the Out signal is driven low when the Reload register is written with the reload value.

At timeout, the counter halts and the Out signal transitions to the high level. However, writing to the Reload register before timeout updates the counter with the reload value, delaying timeout and the Out signal transition to the high level.

The timer is restarted after halting at timeout by writing to the Reload register. The value written to the Reload register is loaded into the timer counter by timer logic, restarting the watchdog operation.

The Out levels are inverted if $Inv = 1$ in the TCR.

Hardware Trigger Watchdog Mode (Mode 4)

The Out signal is initially set to the high or low state, depending on the OUTCTL field in the TCR.

The timer is enabled ($CE=1$) and the mode selected. The counter then begins decrementing and the Out signal is driven low when a trigger event occurs at the In pin.

At timeout, the counter halts and the Out signal transitions to the high level. However, the occurrence of another event at the In pin before timeout updates the counter with the reload value, delaying timeout and the Out signal transition to the high level.

The timer is restarted after halting at timeout by another trigger event at the In pin. The In signal event is determined by the Event field in the TCR and can be a rising edge, falling edge, or both rising and falling edges.

The Out levels are inverted if $Inv = 1$ in the TCR.

Table A3-2 summarizes the timer operating modes. Figures A3-6 through A3-10 show timing for the timer modes.

Table A3-2: Timer Operating Mode Summary

	Go/Halt		Initial Value Loading	Out Signal Control		Function of "IN" Signal
	Go	Halt		Reset	Set	
Mode0 Periodic Interrupt	Reload Reg Write after Mode Set and CE=1	TCR Write, In Gate	Reload Reg Write, Timeout	Reload Reg Write, Count Reg Read	Timeout	Gate ("H" Level) ("L" Level)
Mode1 Timeout Interrupt	Reload Reg Write After Mode Set and CE=1	TCR Write, In Gate, Timeout	Reload Reg Write	Reload Reg Write, Count Reg Read	Timeout	Gate ("H" Level) ("L" Level)
Mode2 Square Wave Generator	Reload Reg Write After Mode Set and CE=1	TCR Write, In Gate	Reload Reg Write, Timeout	Equality Detection (1/2 Reload Value)	Timeout	Gate ("H" Level) ("L" Level)
Mode3 Software Trigger Watchdog	Reload Reg Write After Mode Set and CE=1	TCR Write, Timeout	Reload Reg Write	Reload Reg Write	Timeout	Gate ("H" Level) ("L" Level)
Mode4 Hardware Trigger Watchdog	Input Event	TCR Write, Timeout	Input Event	Input Event	Timeout	Rise-Edge/ Fall-Edge/ Both

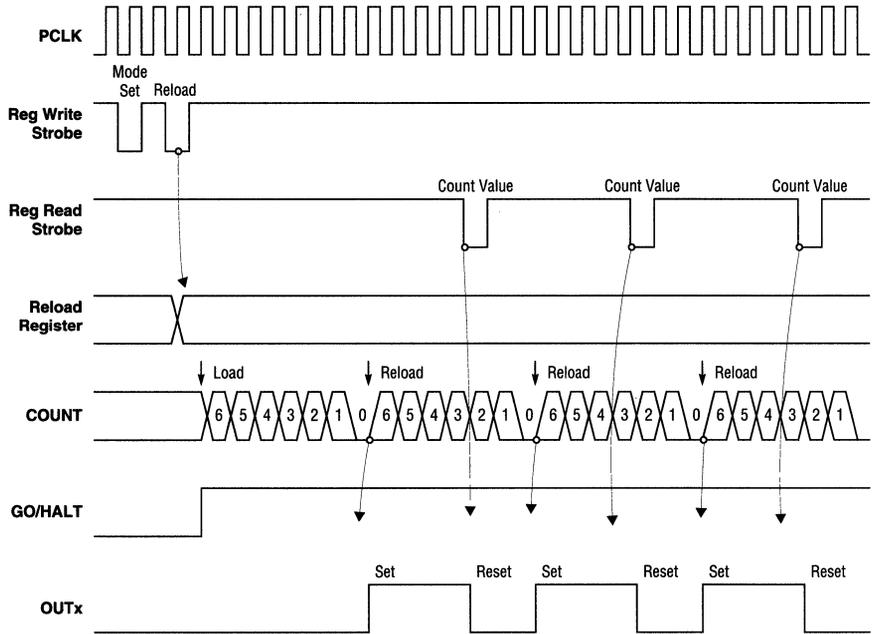


Figure A3-6. Periodic Interrupt Timing (Mode 0) Using the Internal Peripheral Clock

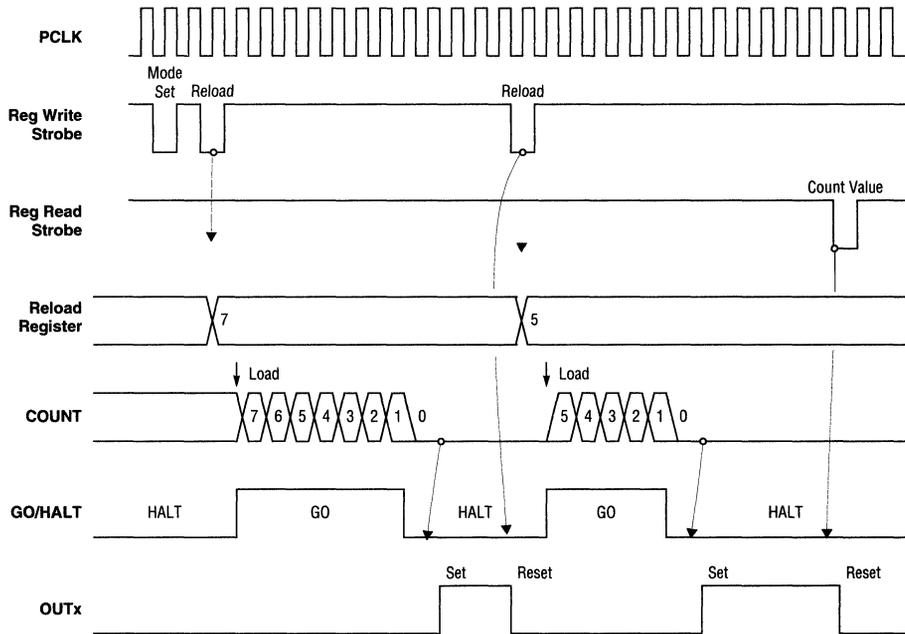


Figure A3-7. Timeout Interrupt Timing (Mode 1) Using the Internal Peripheral Clock

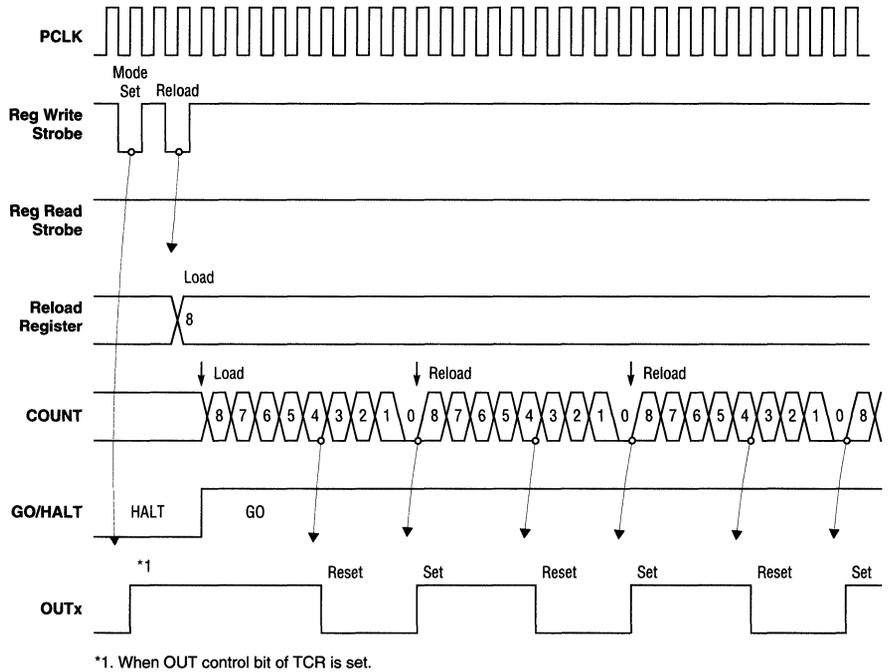
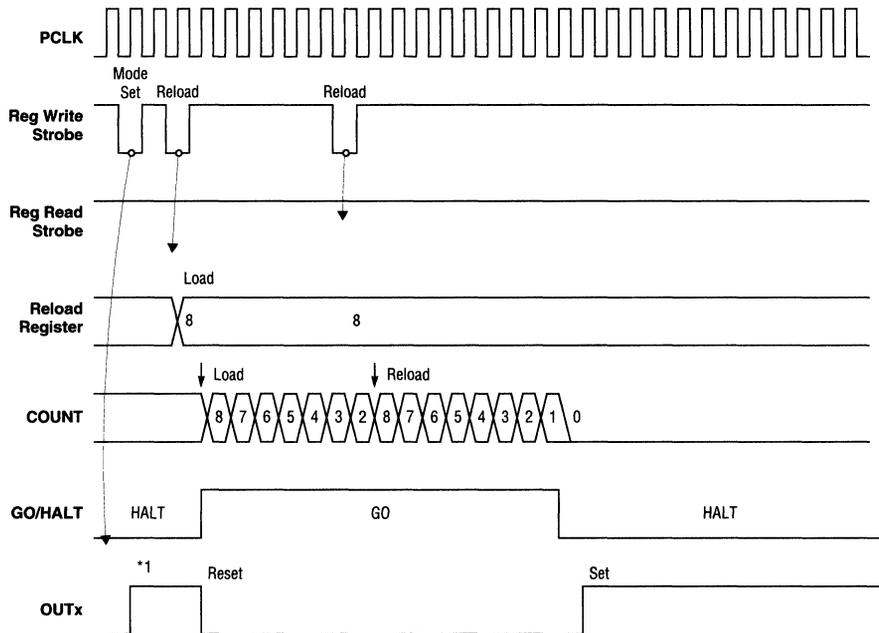


Figure A3-8. Square Wave Generator Timing (Mode 2) Using the Internal Peripheral Clock



*1. When OUT control bit is set.

Figure A3-9. Software Trigger Watchdog Timing (Mode 3) Using the Internal Peripheral Clock

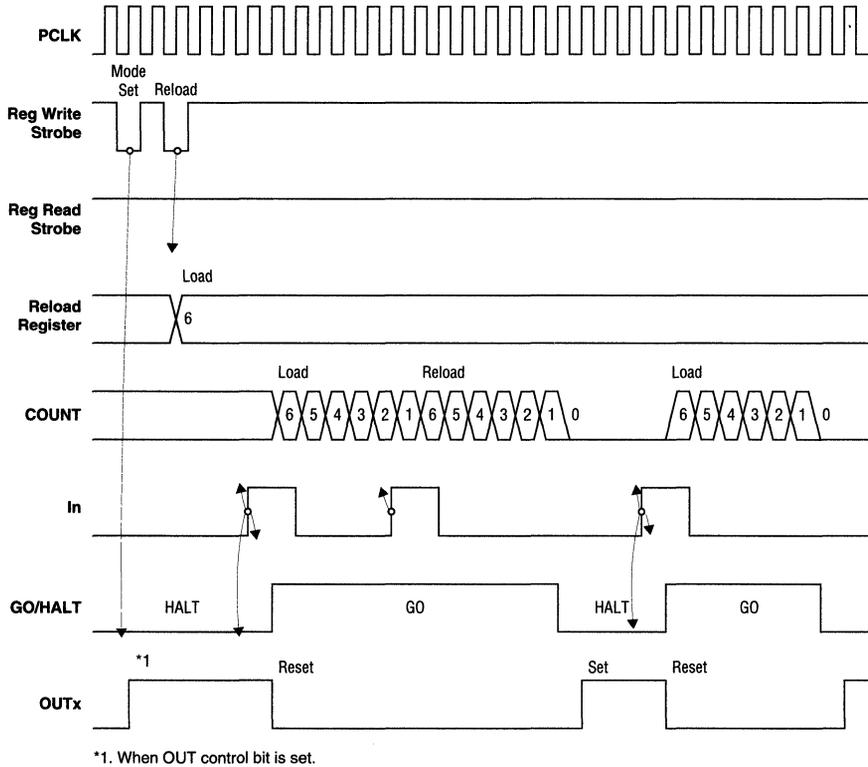


Figure A3-10. Hardware Trigger Watchdog Timing (Mode 4) Using the Internal Peripheral Clock



MB86931 Serial Data Transmitters And Receivers

The MB86931 features two independent serial communication units designated SDTR0 and SDTR1. The SDTRs support synchronous and asynchronous data transfer modes, and are program-compatible with existing industry-standard serial communication devices.

Each SDTR supports the following synchronous mode features:

- 5 to 8 bit data character lengths
- Parity option
- One (MONOSYNC) or two (BISYNC) synchronizing characters

Each SDTR supports the following asynchronous mode features:

- 5 to 8 bit data character lengths
- Parity and stop bit options
- Parity, overrun, and framing error detection
- Divide by 16 or 64 clock options.
- 1, 1.5, or 2 bit length option for stop bit
- Break detection.

The SDTR transmitters and receivers are double buffered and operate independently to allow full-duplex operation. The transmit/receive clock can be exter-

nally generated, or generated by an MB86931 timer. Each SDTR features handshaking signals for modem control.

Figure A4-1 shows a block diagram of an SDTR.

4.1 SDTR Registers

Each SDTR has eight 8-bit registers that can be accessed by the processor. Four registers, the Transmit Data register, the Receive Data register, the Status register, and the Control Data Buffer register, are directly accessed by the processor. The remaining four registers, the Mode register, the Command register, and the two Synchronizing Character registers, are hidden registers that are indirectly accessed by the processor through the Control Data Buffer register.

SDTR registers require 14 system clock cycles to initialize, 20 system clock cycles to update when written during asynchronous mode operation, and 40 system clock cycles to update when written during synchronous mode operation

4.1.1 Hidden Register Access

The Mode, Command, and Synchronous Character registers are accessed sequentially by writing to the Control Data Buffer register (see flowcharts, Figure A4-2 and Figure A4-7).

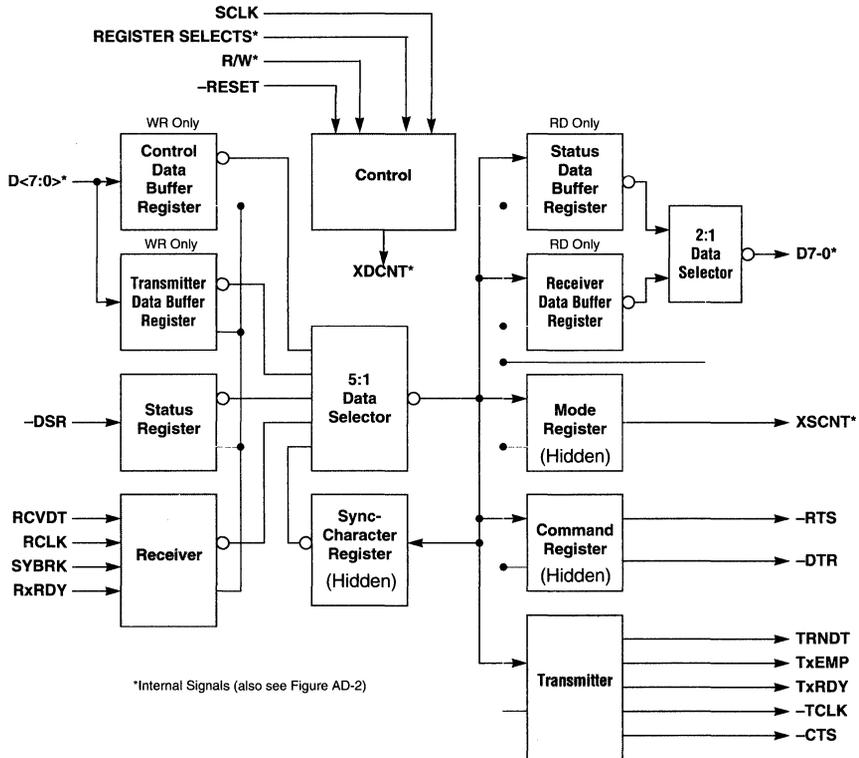


Figure A4-1. SDTR Block Diagram

After a hardware or software reset, the first byte written to the Control Data Buffer register is loaded into the Mode register. The data written into the Mode register determines whether SDTR operates in synchronous or asynchronous mode, and selects the number of SYNCH characters if the mode is synchronous. (See *Command Register* for a software reset description)

Asynchronous Mode Register Access

In the asynchronous mode, all bytes written to the Control Data Buffer register after the Mode register is written are loaded into the Command register. The Synchronizing Character registers are not accessed in the asynchronous mode.

Synchronous Mode Register Access

In the synchronous mode, the second byte written to the Control Data Buffer register after reset is loaded into the first Synchronous Character register.

If one SYNCH character was specified in the Mode register, the third byte written to the Control Data Buffer register is loaded into the Command register, and further writes to the Control Data Buffer register are loaded into the Command register until a reset occurs.

If two SYNCH characters were specified in the Mode register, the second and third bytes written to the Control Data Buffer register are loaded into the two Synchronizing Character registers, and further writes to the Control Data Buffer register are loaded into the Command register until a reset occurs.

4.1.2 SDTR Register Map

Table AD-4 shows the SDTR register map. Note that the Transmit Data register and the Receive Data register in each SDTR share the same address, and that the Control Data Buffer register and the Status register share the same address. Selection of one of the registers at each address is determined by whether the access operation is a read or a write. The Transmit Data register and the Control Data Buffer registers are write-only registers that are selected during write operations; the Receive Data register and the Status register are read-only registers that are selected during read operations.

4.1.3 Control Data Buffer Register

This is a write-only register through which the processor writes to the Mode register, the Command register, and the Synchronous Character registers.

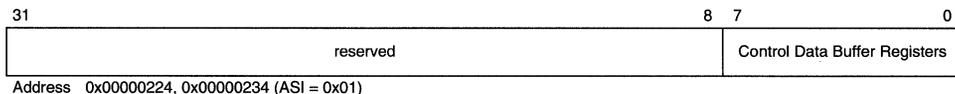


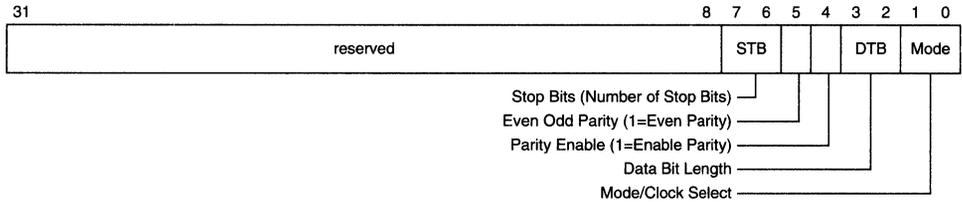
Table A4-1: SDTR Register Map

Functional Unit	Access	Address	Bits <31:8>	Bits <7:0>
SDTR 0	Write	0x00000220	0x00 (RSVD)	Transmit Data Register
	Read	0x00000220	0x00 (RSVD)	Receive Data Register
	Write	0x00000224	0x00 (RSVD)	Control Data Buffer Register
	Read	0x00000224	0x00 (RSVD)	Status Data Buffer Register
SDTR 1	Write	0x00000230	0x00 (RSVD)	Transmit Data Register
	Read	0x00000230	0x00 (RSVD)	Receive Data Register
	Write	0x00000234	0x00 (RSVD)	Control Data Buffer Register
	Read	0x00000234	0x00 (RSVD)	Status Data Buffer Register

4.1.4 Mode Register

The Mode register has two formats according to the mode selected in the Mode field.

In the asynchronous mode, the register controls stop bit length, parity, data character length, and data transfer clock frequency as follow:



Bits 7-6: Stop Bit Length - Selects the length of the stop bits as follows:

Bit 7	Bit 6	Number of Stop Bits
0	0	None
0	1	1
1	0	1.5
1	1	5

Bit 5: Even Odd Parity - Selects parity as follows:

0: Odd parity.
 1: Even parity.

Bit 4: Parity Enable - Enables parity when set to 1.

Bits 3-2: Data Bit Length - Selects the number of character bits as follows:

Bit 3	Bit 2	Number of Bits
0	0	5
0	1	6
1	0	7
1	1	8

Bits 1-0: Mode/Clock Select- -Selects the operating mode and the asynchronous mode Baud rate as follows:

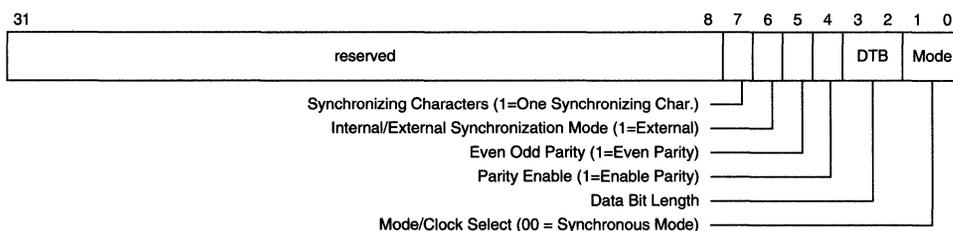
Bit 1	Bit 0	Mode/Clock Selection
0	0	Synchronous Mode
0	1	-TCLK/RCLK Freq.

Bit 1	Bit 0	Mode/Clock Selection
1	0	1/16 -TCLK/RCLK Freq.
1	1	1/64 -TCLK/RCLK Freq.

-TCLK and RCLK may have different frequencies, resulting in different transmitter and receiver Baud rates. The division factor selected in the Mode register applies to both the transmitter clock and the receiver clock.

Each clock option selects the asynchronous mode.

In the synchronous mode, the Mode register controls the number of synchronizing characters, internal or external synchronous mode operation, parity, and character length as follows:



Bit 7: Synchronizing Characters - Selects the number of synchronizing characters as follows:

- 0: Two synchronizing characters.
- 1: One synchronization character.

Bit 6: Internal/External Synchronization Mode - Selects the synchronization mode as follows:

- 0: Internal synchronization mode.
- 1: External synchronization mode.

Bit 5: Even Odd Parity - Selects parity as follows:

- 0: Odd parity.
- 1: Even parity.

Bit 4: Parity Enable - Enables parity when set to 1.

Bits 3-2: Data Bit Length - Selects the number of character bits as follows:

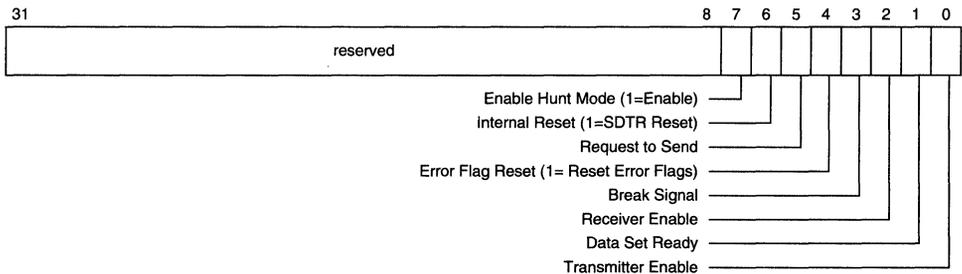
Bit 3	Bit 2	Number of Bits
0	0	5
0	1	6
1	0	7
1	1	8

Bits 1-0: Mode/Clock Select - This field must be 0 to select synchronous mode.

Reset forces the Mode register to 0x42. This represents asynchronous mode with 1/16 transmit/receive clock, 5-bit characters, disabled odd parity, and one stop bit.

4.1.5 Command Register

The Command register enables the transmitter and receiver, resets the SDTR and the Error Flag Reset flag in the status register, controls modem handshaking signals, and enables hunt mode as follows:



Bit 7: Enable Hunt Mode - Enables hunt mode in the asynchronous mode as follows:

- 0: Disable hunt mode.
- 1: Enable hunt mode.

The hunt mode enables the receiver to synchronize with the character stream by comparing the received characters with the synchronizing characters in the Synchronizing Character registers. (See the SYBRK signal description).

Bit 6: Internal Reset - Resets the SDTR as follows:

- 0: No effect.
- 1: SDTR reset.

During operation, the processor must reset the SDTR by setting IRST to 1 to access the Mode register.

Bit 5: Request to Send - The processor asserts the RTS modem handshaking output signal as follows:

- 0: High level.
- 1: Low level

RTS is typically set to 1 to request the modem to establish a carrier.

Bit 4: Error Flag Reset - Resets all error flags in the Status register as follows:

- 0: No effect.
- 1: Resets error flags.

Bit 3: Break Signal - Asserts break on the TRNDT output signal as follows:

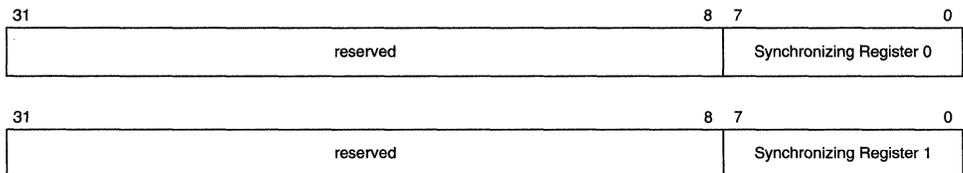
- 0: No effect.
- 1: The TRNDT signal is forced low.

- Bit 2:** Receiver Enable - Enables the receiver as follows:
 0: Receiver disabled.
 1: Receiver enabled.
- Bit 1:** Data Terminal Ready - The processor asserts the DTR modem handshaking output signal as follows:
 0: High level.
 1: Low level
- The DTR signal can be used to prepare the modem for transmission.
- Bit 0:** Transmitter Enable - Enables the transmitter as follows:
 0: Transmitter disabled.
 1: Transmitter enabled.

Reset does not affect the Command register.

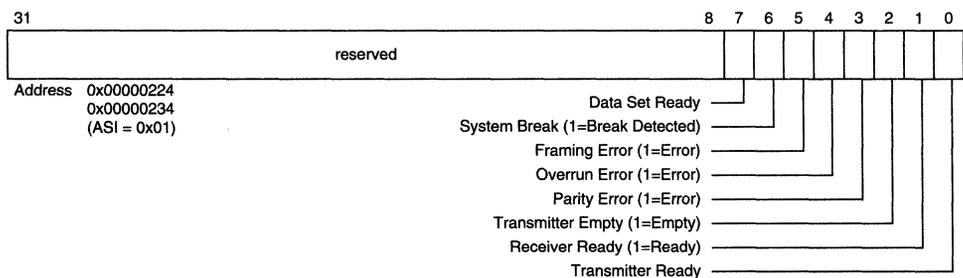
4.1.6 Synchronizing Character Registers

The Synchronizing Character registers hold the synchronizing characters that are used in the synchronous mode. One synchronizing character is written to the first Synchronizing Character register in both the MONOSYNCH and the BISYNCH modes; a second synchronizing character is written to the second Synchronizing Character register in the BISYNCH mode.



4.1.7 Status Register

The Status register is a read-only register that contains the Data Set Ready flag, transmitter status and error flags, and receiver status and error flags as follows:



- Bit 7:** Data Set Ready (DSR) - Indicates the state of the DSR modem input signal as follows:

0: High level.

1: Low level.

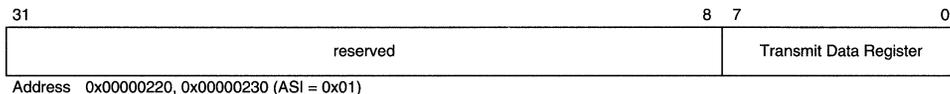
The DSR signal is asserted by the modem to indicate that it is ready for data transfer.

- Bit 6:** System Break (SYBRK) - Indicates Synchronizing Character detection in the synchronous mode and break code detection in the asynchronous mode as follows:
- 0: No detection.
1: Detection.
- Bit 5:** Framing Error (FERR) - Indicates detection of a framing error as follows:
- 0: No framing error.
1: Framing error.
- This flag is set to 1 in the asynchronous mode if the number of stop bits following a character is not correct.
- Bit 4:** Overrun Error (OERR) - Indicates detection of an overrun error as follows:
- 0: No overrun error.
1: Overrun error.
- This flag is set to 1 to indicate that data was transmitted to the receiver while the receiver buffer was full.
- Bit 3:** Parity Error (PERR) - Indicates the detection of a parity error as follows:
- 0: No parity error.
1: Parity error.
- Bit 2:** Transmitter Empty (TxEMP) - Indicates whether the transmitter data buffer is empty as follows:
- 0: Transmitter buffer not empty.
1: Transmitter buffer empty.
- Bit 1:** Receiver Ready (RxRDY) - Indicates whether the receiver is ready for more data as follows:
- 0: Receiver not ready.
1: Receiver ready.
- Bit 0:** Transmitter Ready (TxRDY) - Indicates that the transmitter is ready for more data as follows:
- 0: Transmitter not ready.
1: Transmitter ready.

Reset sets the FERR, OERR, and PERR flags to 1. All other flags are undefined.

4.1.8 Transmit Data Register

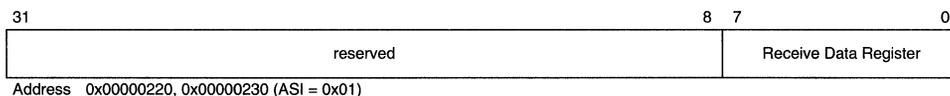
The processor writes data to this write-only register for transfer to the transmit data buffer.



Reset forces the Transmit Data register to FF.

4.1.9 Receive Data Register

The processor reads data from the receiver data buffer through this read-only register.



Reset leaves the Receive Data register undefined.

4.2 Asynchronous Mode Operation

In the Asynchronous mode, each transmitted character is preceded by a low-level start bit. The start bit is immediately followed by 5 to 8 character bits, an optional parity bit, and one or two high-level stop bits. The number of character bits, the number of stop bits, and type of parity is selected in the Mode register.

The receiver uses the high-to-low transition of the start bit to synchronize with the data stream. The interval between each character is a high level due to either the stop bit of the preceding character, or "marking" if the line is idle. When the receiver detects a start bit, it samples the received bit stream at bit-wide intervals based on the Baud rate to identify the character bits, the parity bit, and the stop bit(s). The parity bit that follows the character must be correct or a parity error occurs, and the stop bit(s) must be correct or a framing error occurs.

4.2.1 Operation Description

Figure A4-2 shows a flowchart for asynchronous mode operation. The flowchart begins with power-on reset, which must be held for a least six system clock cycles to ensure proper reset. Reset forces the SDTR I/O signals to the following states:

Signal	Initial Level
-DTR	High
-RTS	High
TxRDY	Low
RxRDY	Low
TRNDT	High
TxEMP	High
SYBRK	Low

The Mode and Command registers are then written to program the SDTR. The SDTR can then be software-reset through the Command register to access the Mode register, or can be used to receive and transmit data. The Command register can be accessed at any time during transmit/receive operations.

Note that the transmitter must be enabled in the Command register and the -CTS input signal must be low to transmit data, and that the receiver must be enabled in the Command register to receive data.

Writes to the Mode and Command registers may not have effect for as many as 10 -TCLK/RCLK cycles.

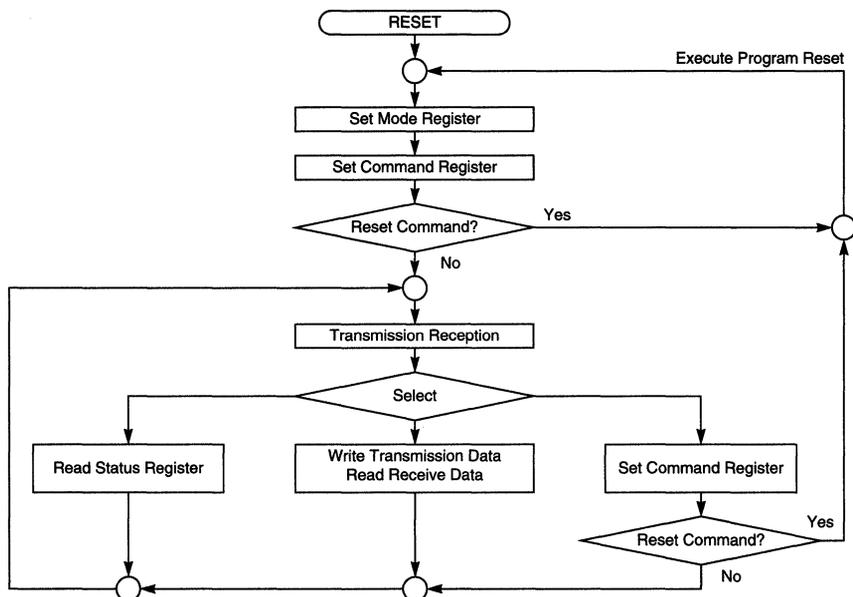
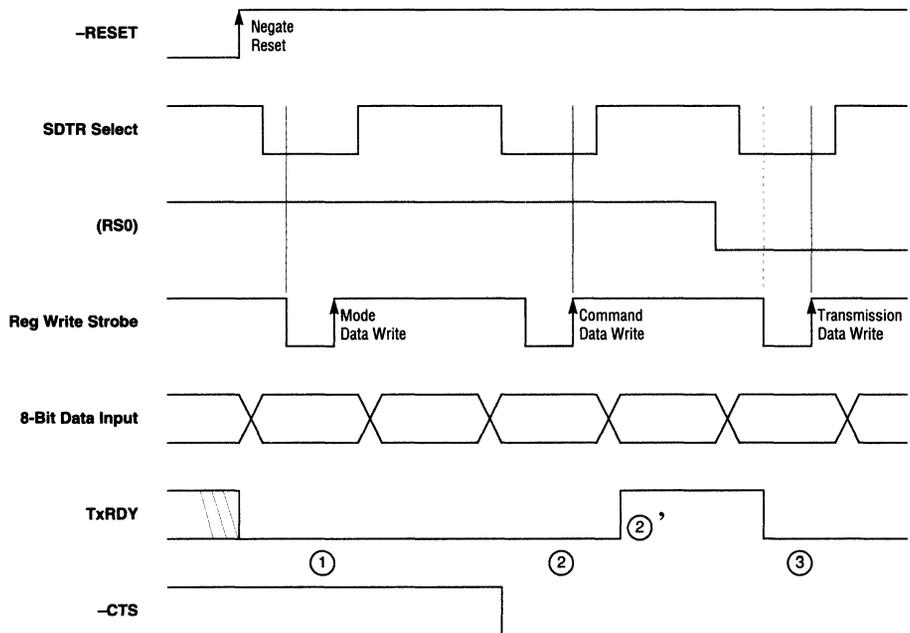


Figure A4-2. Asynchronous Mode Operation Flowchart

4.2.2 Asynchronous Mode Timing

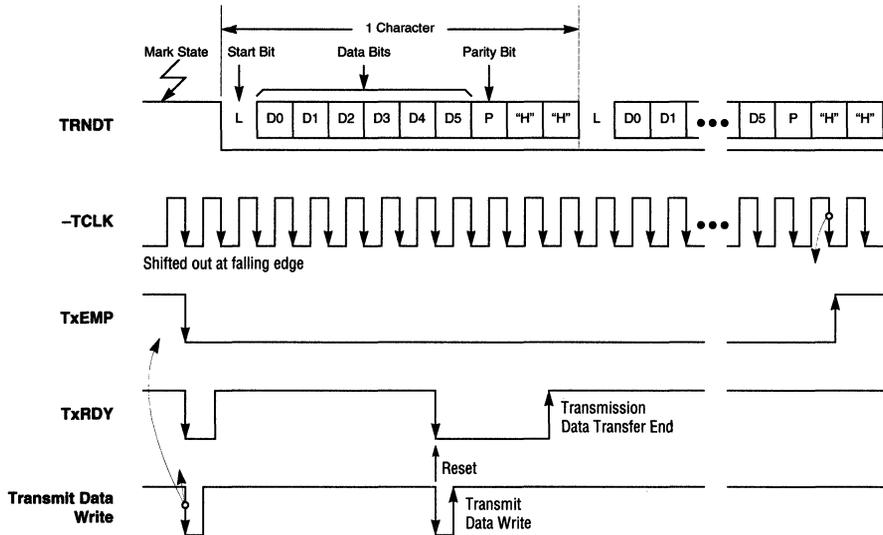
Figures A4-3 through A4-10 show timing for various SDTR asynchronous transmitter and receiver operations. The operations are typical and should be understood before using the SDTR.



Notes:

1. ① indicates the Mode register write interval; ② indicates the Command register write interval; ③ indicates the data transmission interval.
2. TxRDY is asserted high at ②' because the -CTS input is low and the transmitter was enabled in the Command register.

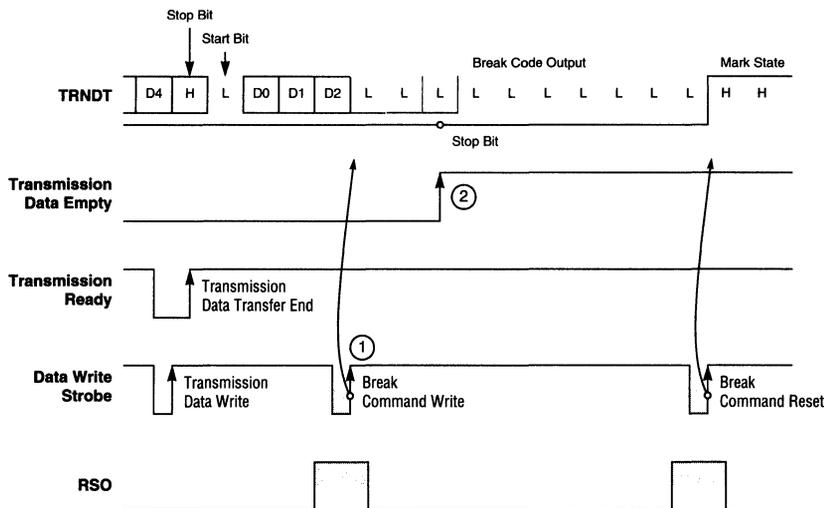
Figure A4-3. Asynchronous Mode Transmitter Initialization



Notes:

1. Start and stop bits are added by the SDTR character-by-character in the asynchronous mode.
2. The frame format is 6 bit character length, parity, and 2 stop bits.
3. The TRNDT pin remains high after reset until the transmit data is written to the SDTR.
4. When the CPU writes transmit data to the SDTR, the SDTR appends start, parity, and stop bits to the data to form a frame. The SDTR transmits the frame to an external unit bit-by-bit at the falling edge of the -TCLK transmitter clock.
5. The TxRDY input signal must be high before the CPU writes the transmit data. The TxRDY signal transitions to the low level when the CPU writes the data, then transitions back to the high level when the SDTR transfers the data to the transmit shift register. The CPU can then write more data to the transmitter.
6. The TxEMP signal transitions to the low level when the CPU writes the transmit data to the SDTR, then transitions to the high level when the SDTR has transmitted all data in its transmit data buffer.

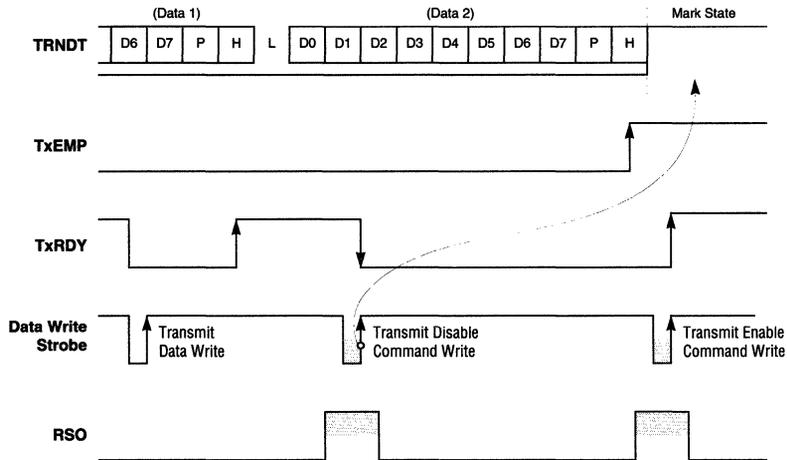
Figure A4-4. Asynchronous Mode Data Transmission Timing



Notes:

1. The frame format is 5-bit character length, no parity, and one stop bit.
2. A break is forced in interval ① by setting the Break bit to 1 in the Command register. The transmitter continues normal operation, but the transmitter output signal (TRNDT) is forced low. Note that the TxEMP output signal is asserted in interval ② to indicate that the transmitter buffer is empty even during the break condition.
3. The transmitter output transitions to the high level until more data is transmitted once the break is terminated by clearing the Break bit to 0.

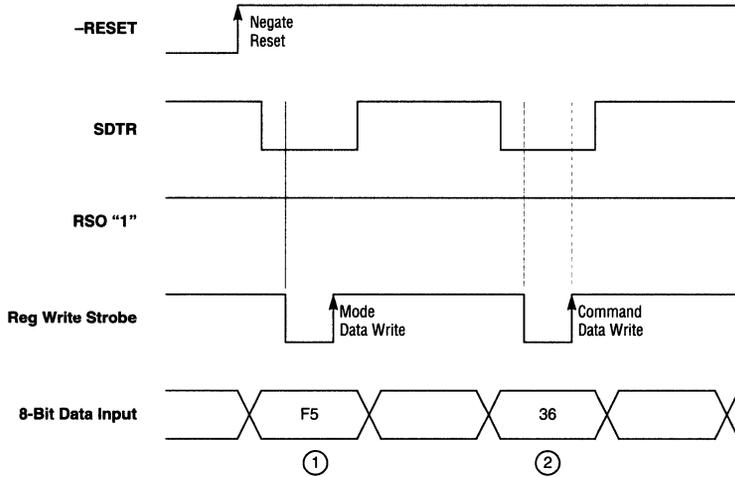
Figure A4-5. Break Timing



Notes:

1. The frame format is 8-bit character length, parity, and 1 stop bit length.
2. When the transmitter is disabled by clearing the TxEN bit in the Command register, the SDTR continues transmitting all data in the transmitter buffer, then enter the mark state. While the transmitter is disabled, data that the CPU writes to the transmitter is not transmitted until the transmitter is enabled. TxRDY remains low while the transmitter is disabled, even if the transmitter data buffer is empty.

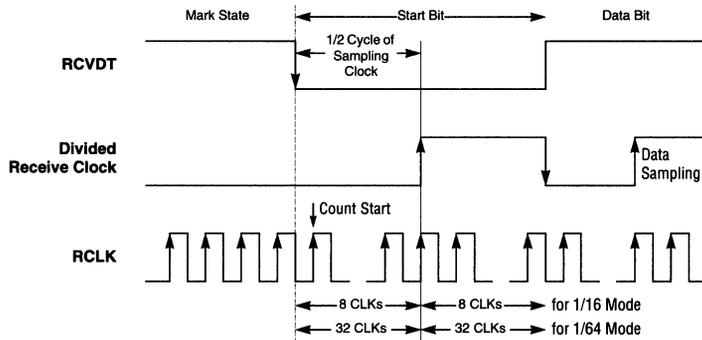
Figure A4-6. Asynchronous Mode Transmit Disable Timing



Notes:

1. ① indicates the Mode register write interval.
2. ② indicates the Command register write interval.

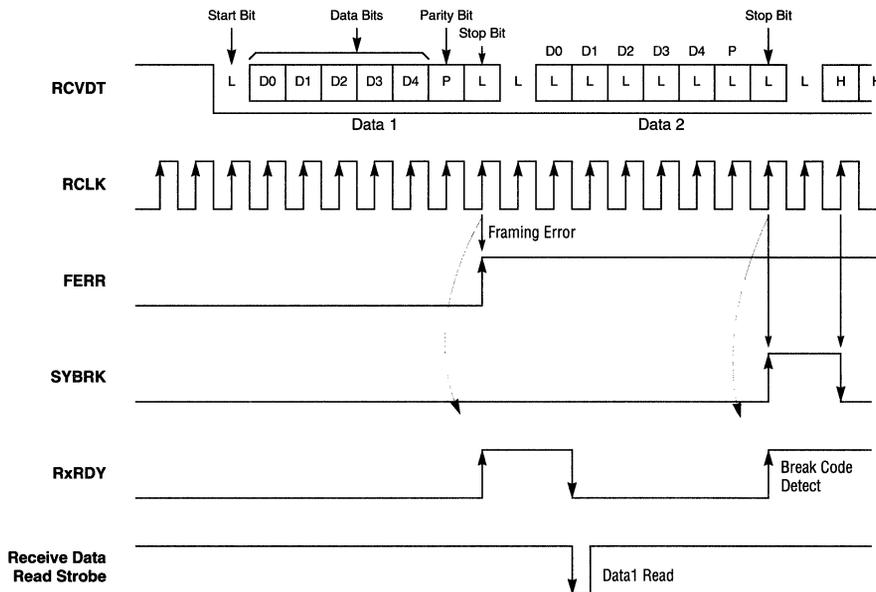
Figure A4-7. Asynchronous Mode Receiver Initialization



Notes:

1. This figure shows start bit detection when the Baud rate is 1/16 or 1/64 the RCLK (receiver clock) frequency. The start bit must be low for at least 8 RCLK cycles if the Baud rate is 1/16 the RCLK frequency, and must be low for at least 32 RCLK cycles if the Baud rate is 1/64 the RCLK frequency.

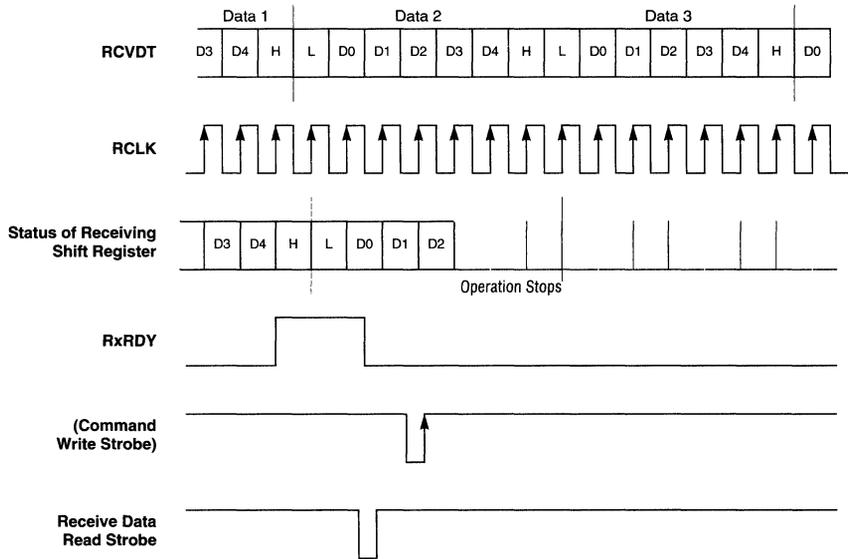
Figure A4-8. Start Bit Detection Timing



Notes:

1. The frame format is 5-bit character length, parity, and 1 stop bit.
2. In this example, the receiver detects a low level in the stop bit position and reports a framing error (FERR). The receiver then detects a low level that lasts an entire character interval and asserts SYBRK to report a break. SYBRK remains asserted until the RCVDT signal returns to the high level, or the SDTR is reset.

Figure A4-9. Break Code Detection Timing



Notes:

1. The frame format is 5-bit character length, no parity, 1 stop bit.
2. When the receiver is disabled by clearing RxEN (bit 2) in the Command register to 0, the receiver stops all operation, including error detection. When the receiver is enabled, the receiver begins operation. If RCVDT is low when the receiver is enabled, the receiver recognizes the low level as a start bit and begins sampling the received data.

Figure A4-10. Asynchronous Mode Receiver Disable Timing

4.3 Synchronous Mode Operation

In the synchronous mode, the receiver maintains bit synchronization with the received data by phase-locking its clock with the received data or by using an external clock that is already synchronized with the data. This allows the receiver to receive an indefinite number of successive characters without start bits or stop bits.

The receiver must determine, however, when a character string, sometimes called a frame, begins. It does so with either SYNCH (synchronization) characters if operating in the internal synchronization mode (IESM = 0 in the Mode register), or with an external synchronization signal at the SYBRK pin (IESM = 1 in the Mode register). If the data transfer is interrupted, the transmitter re-establishes frame synchronization by re-transmitting the SYNCH characters.

4.3.1 Operation Description

Figure A4-11 shows a flowchart for synchronous mode operation.

The Mode register is written immediately after reset. A synchronizing character is then written into the Synchronizing register, and a second synchronizing character is written into the Synchronizing register if in the BISYNCH mode. The Command register is then written.

The STDR can then be software-reset through the Command register to access the Mode register, or can be used to receive and transmit data. The Command register can be accessed at any time during transmit/receive operations.

Writes to the Mode and Command registers may not have effect for as many as 20 -TCLK/RCLK cycles.

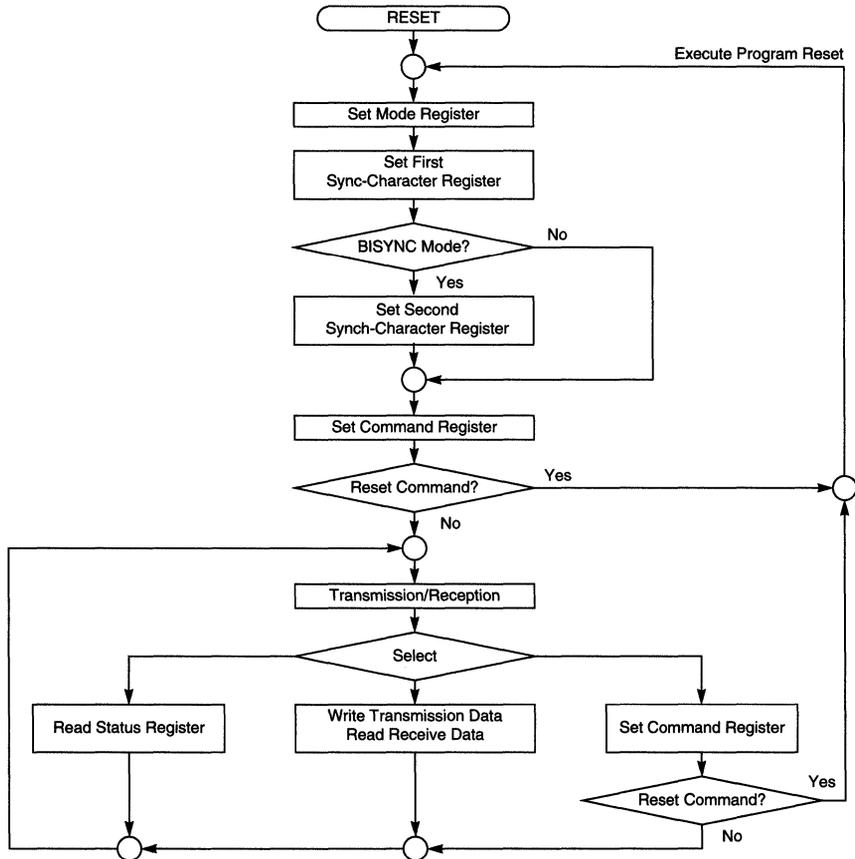
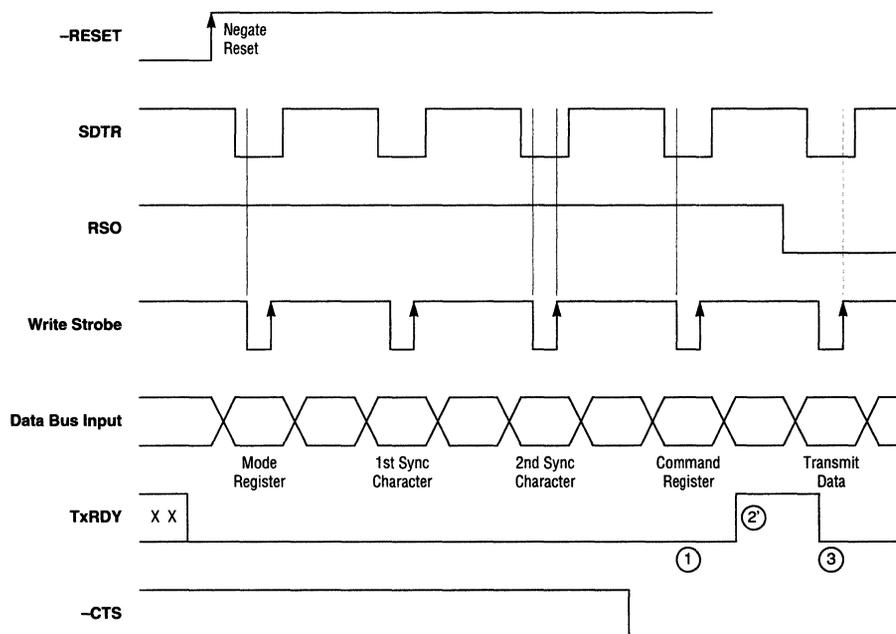


Figure A4-11. Synchronous Mode Operation Flowchart

4.3.2 Synchronous Mode Timing

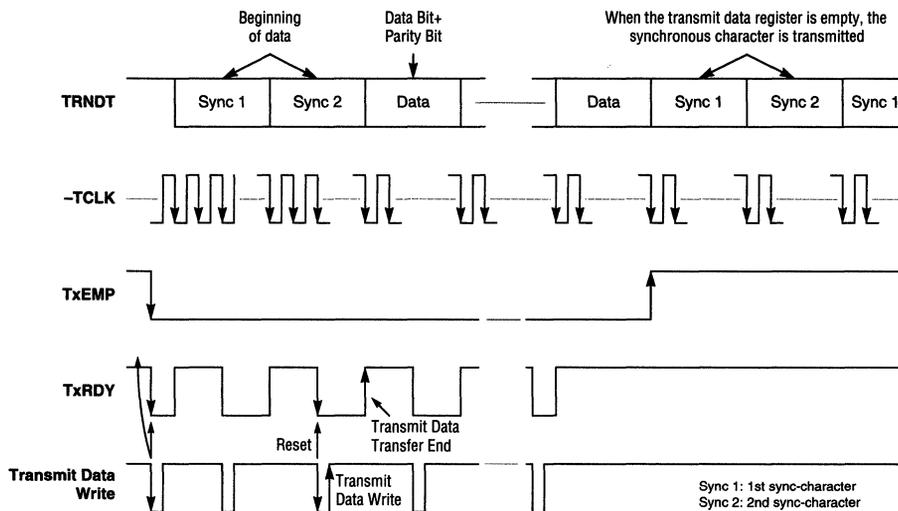
Figures A4-12 through A4-20 show timing for various SDTR synchronous transmitter and receiver operations. The operations are typical and should be understood before using the SDTR.



Notes:

1. The Mode, Synchronizing Character, and Command registers are written as shown in the figure. The TxRDY flag is asserted in interval ② because the transmitter was enabled in the Command register in interval ① and -CTS is low. TxRDY is released in interval ③ because data is written to the SDTR for transmission.

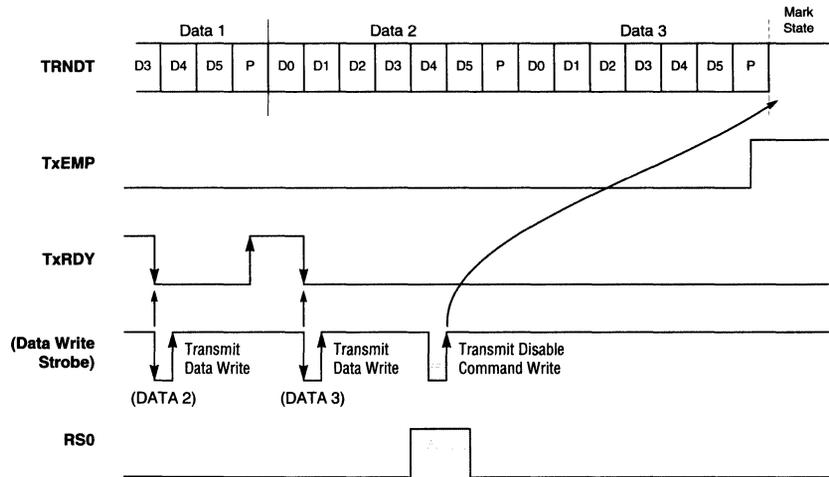
Figure A4-12. Synchronous Mode Transmitter Initialization



Notes:

1. The frame format is BISYNC mode with parity.
2. The data to be transmitted is written to the SDTR when TxRDY is high. TxRDY transitions to the low level when the data is transmitted.
3. The SDTR transmits synchronization characters while the transmitter buffer is empty (TxEMP = 1) to maintain synchronization.

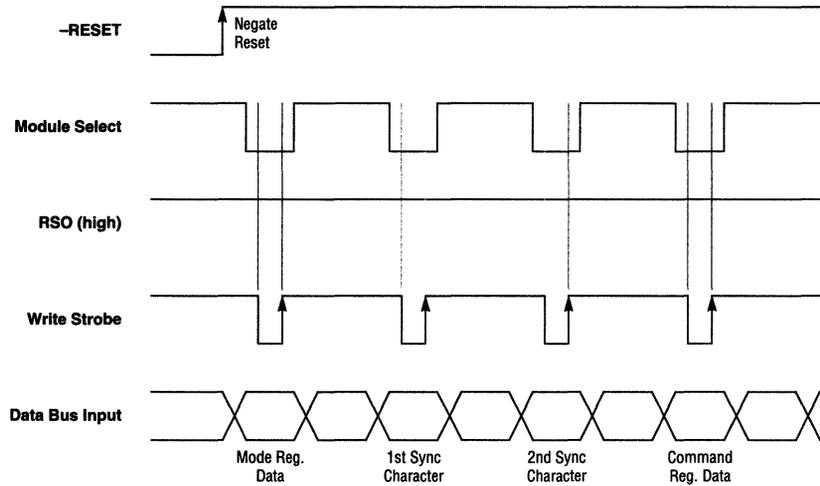
Figure A4-13. Synchronous Data Transmission Timing



Notes:

1. The frame format is 6-bit character length and parity.
2. If the transmitter is disabled during transmission (TxEN = 0 in Command register), the SDTR transmits all data in its data buffer then enters the mark state. If the transmitter is disabled during transmission of the first SYNC character in the BISYNC mode, the SDTR transmits the second SYNC character, then enters the mark state.
3. The TxEMP signal remains high and the RxRDY signal remains low while the transmitter is disabled.

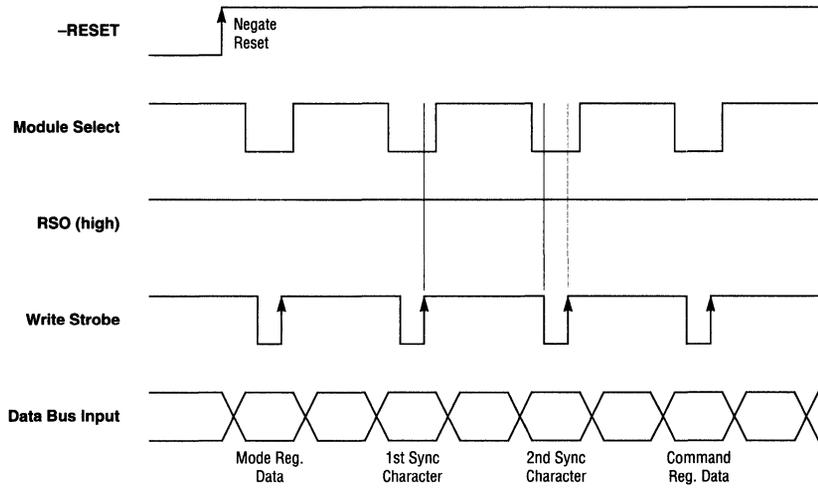
Figure A4-14. Synchronous Mode Transmitter Disable Timing



Notes:

1. The Mode, Synchronizing Character, and Command registers are written as shown. The receiver must be enabled and the -DTR and -RTS input signals must be low to receive data.

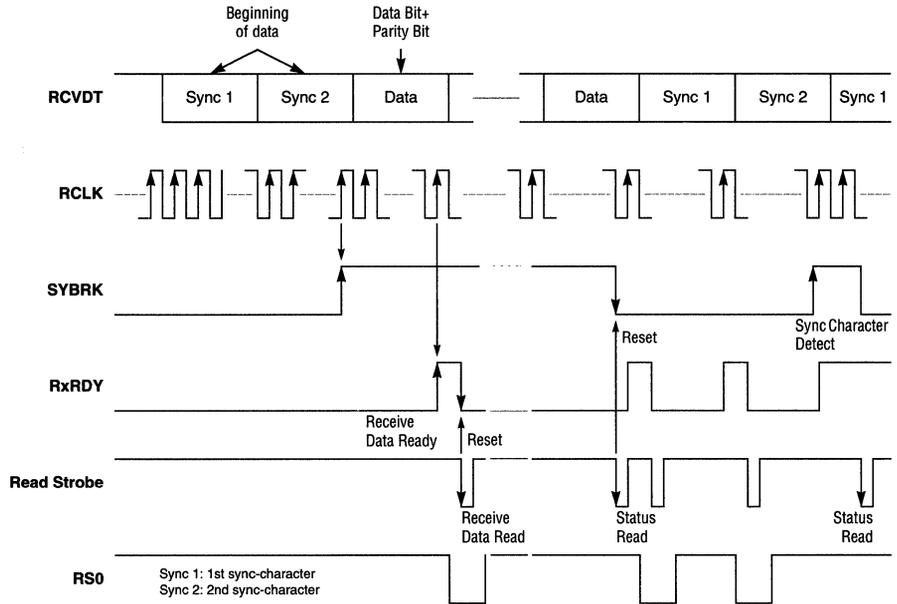
Figure A4-15. Synchronous Mode Receiver Initialization Timing



Notes:

1. The Mode, Synchronizing Character, and Command registers are written as shown. The receiver must be enabled and the -DTR and -RTS input signals must be low to receive data.

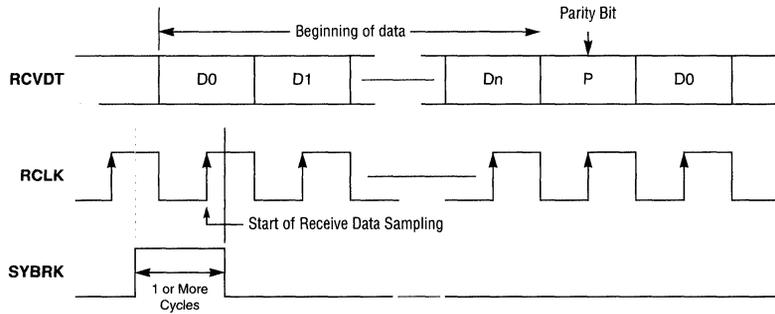
Figure A4-16. Synchronous Mode Receiver Initialization Timing



Notes:

1. The frame format is BISYNC mode and parity.
2. The SDTR compares the received synchronizing characters with the characters in the Synchronizing Character register. If there is a match, the SDTR asserts the SYBRK flag in the Status register to 1 and the SYBRK output signal to the high level to indicate that synchronization is established. The SDTR then reads the data and parity bit, and asserts RxRDY to indicate to the processor that data is available. The SDTR will not assert RxRDY unless frame synchronization is established.
3. The SYBRK and RxRDY signals transition to the low state and the SYBRK flag clears to 0 when the processor reads the data.

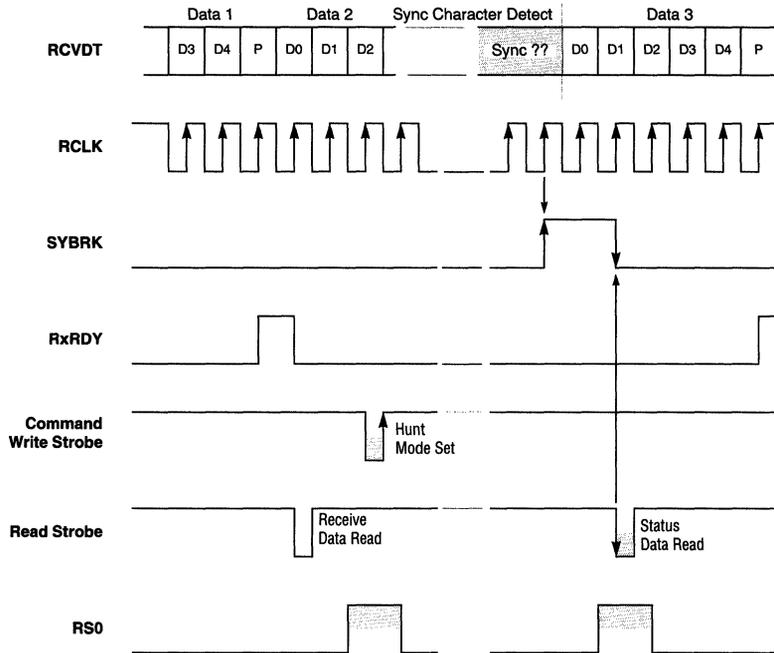
Figure A4-17. Synchronous Mode Data Reception Timing



Notes:

1. In the external synchronization mode, a high SYBRK external signal identifies the beginning of a frame. Synchronization is established when SYBRK is asserted to the high level while RCLK is high, and data is sampled on the following low-to-high transitions of RCLK.
2. The SYBRK synchronizing signal should be asserted for at least one RCLK cycle.

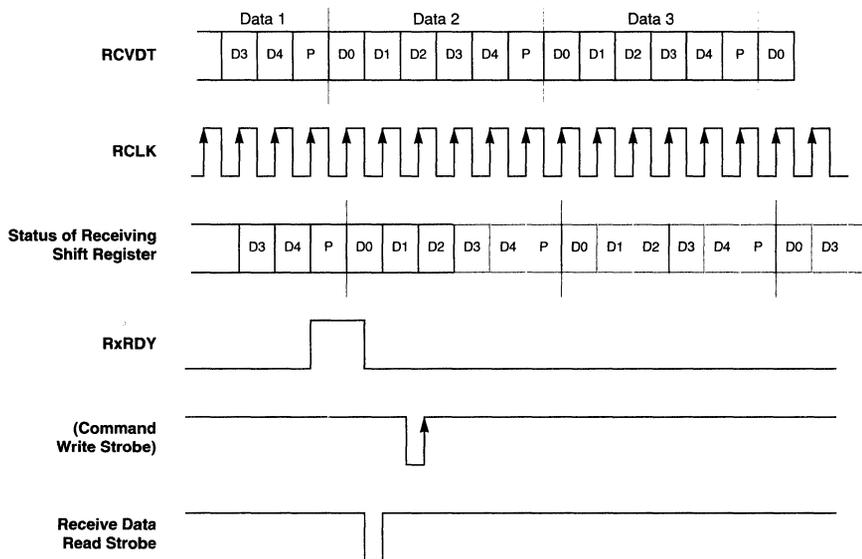
Figure A4-18. External Synchronization Mode Timing



Notes:

1. The frame format is 5-bit character length and parity.
2. Hunt mode is enabled in the Command register (EHM = 1) to initiate synchronizing character search by the receiver. Data stored in the receiver when hunt mode is enabled is cleared.

Figure A4-19. Hunt Mode Timing



Notes:

1. The frame format is 5-bit character length and parity.
2. The RxRDY status flag and the RxRDY output signal are masked when the receiver is disabled in the Command register (RxEN = 0). The RxRDY flag and the error flags operate normally while synchronization is maintained and data is received.
3. The RxRDY status flag and output signal are unmasked when the receiver is enabled.

Figure A4-20. Synchronous Mode Receiver Disable Timing

4.4 Status Flag Operation And Timing

The TxRDY flag sets to 1 to indicate that the transmitter buffer can accept more data from the processor for transmission. The TxRDY flag sets even if the transmitter is disabled.

The TxRDY output signal transitions to the high state to indicate that the transmitter can accept more data *and* that the transmitter is enabled.

The TxEMP flag sets to 1 to indicate that the transmitter is empty.

Figure A4-21 shows TxRDY flag and output signal timing.

Figure A4-22 shows parity error timing for odd parity. The data in Figure A4-22(a) has proper odd parity; the data in Figure A4-22(b) has incorrect even parity and generates a parity error (PERR = 1).

Figure A4-23 shows receiver overrun. Character #5 is transmitted to the receiver while the receiver buffer is full, causing the error (OERR = 1).

Figure A4-24 shows a framing error. The stop bit is low rather than high, causing the error (FERR = 1)

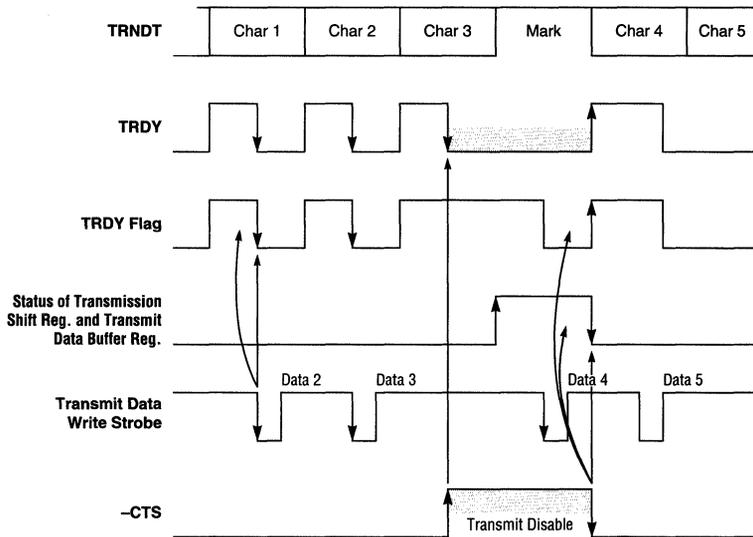


Figure A4-21. TxRDY Timing

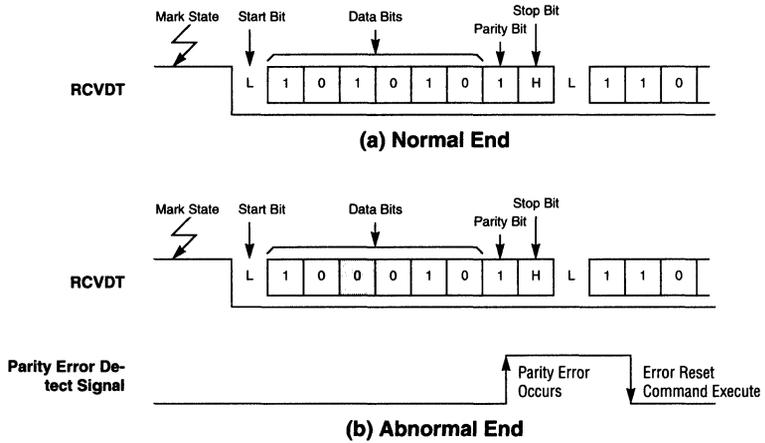


Figure A4-22. Parity Error Timing

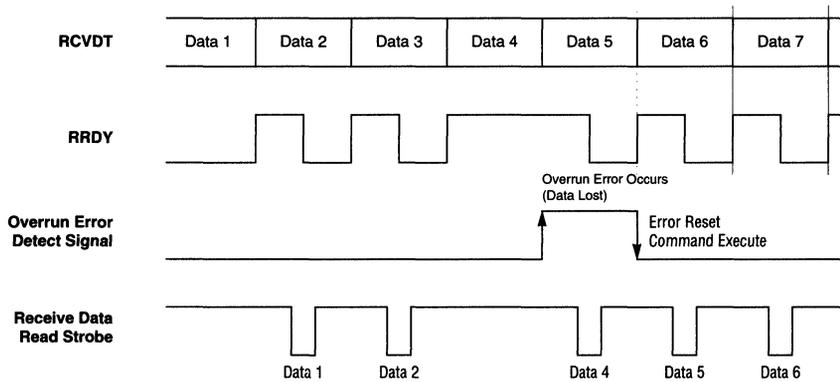


Figure A4-23. Overrun Error Timing

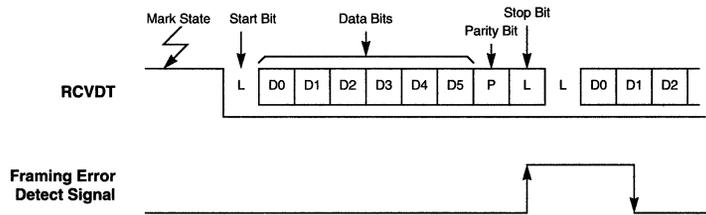


Figure A4-24. Framing Error Timing



External Interface

The following tables list the MB86931 external signals. The signals are grouped according to MPU interface, interrupt request, timer, and SDTR functions.

5.1 Signals

Signals in the tables that are preceded by a hyphen (-) are active low. Dual-function signals have two names that are separated by a slash (/).

5.1.1 Processor Signals Descriptions

Signal	Type	Function
-RESET	I A(L)	SYSTEM RESET: Asserting reset for at least 4 processor cycles after the clock has stabilized, causes the MB86931 to be initialized.
XTAL1, (CLK_IN) XTAL2	I O G(Q) I (Q)	EXTERNAL OSCILLATOR: The crystal inputs determine execution rate and timing of the MB86931 processor. Connecting a crystal to these pins forms a complete crystal oscillator circuit. The crystal oscillator frequency is the same as the processor operating frequency. When driving the processor with an external clock, XTAL2 pin should be left floating.
CLKOUT1	O G(Q) I (Q)	CLOCK OUTPUT 1: This is an output signal against which MB86931 bus transactions can be referenced. The CLKOUT1 frequency is the same as the frequency applied to XTAL1 and is the same as the processor operating frequency. CLKOUT1 is in phase with CLK_IN.
CLKOUT2	O G(Q) I (Q)	CLOCK OUTPUT 2: This is an output signal against which MB86931 bus transactions can be referenced. The CLKOUT2 frequency is the same as the frequency applied to XTAL1 and is the same as the processor operating frequency. CLKOUT2 is out of phase with CLK_IN.
-LOCK	O S(L) G(Z) I (1)	BUS LOCK: This is a control signal asserted by the processor to indicate to the system that the current bus transaction requires more than one transfer on the bus. The Atomic Load Store instruction for example requires contiguous bus transactions which cause the assertion of the bus lock signal. The bus may not be granted to another bus owner as long as -LOCK is active. -LOCK is asserted with the assertion of -AS and remains active until -READY is asserted at the end of the locked transaction.
-BREQ	I S(L)	BUS REQUEST: Asserted by another device on the bus to indicate that it wants ownership of the bus. The request must be answered with a bus grant (-BGRNT) from the MB86931 before the device can proceed by driving the bus. Once the bus has been granted, the device has ownership of the bus until it de-asserts -BREQ. The user should ensure that devices on the bus cannot monopolize the bus to the exclusion of the CPU. Inputs to -BREQ while -RESET is active are valid and cause Bus Grant to be asserted.
-BGRNT	O S(L) G(O) I (Q)	BUS GRANT: Asserted by the CPU in response to a request from a device wanting ownership of the bus. The CPU grants the bus to other devices only after all transfers for the current transaction are completed. All bus drivers are three-stated with the assertion of the bus grant signal.
-ERROR	O A(L) G(Q) I (Q)	ERROR SIGNAL: Asserted by the CPU to indicate that it has halted in an error state as a result of encountering a synchronous trap while traps are disabled. In this situation the CPU saves the PC and nPC registers, sets the tt value in the TBR, enters into an error state and asserts the -ERROR signal. The system can monitor the -ERROR pin and initiate a reset under the error condition. This pin is high on reset.

Signal	Type	Function
-MEXC	I S(L)	MEMORY EXCEPTION: Asserted by the memory system to indicate a memory error on either a data or instruction access. Assertion of this signal initiates either a data or instruction access exception trap in the IU. The current bus access is invalidated by asserting the -MEXC in the same cycle as the -READY signal. Assertion in any other bus cycle gives indeterminate results. The IU ignores the contents of the data bus in cycles where -MEXC is asserted.
-TIMER_OVF	O S(L) G(Q) I(Q)	TIMER UNDERFLOW: Asserted by the processor to indicate that the internal 16-bit timer has underflowed. This signal can be used to initiate a DRAM refresh cycle or a one cycle periodic waveform. On reset, the timer is turned off and -TIMER_OVF is high.
-SAME_PAGE	O S(L) G(1) I(1)	SAME-PAGE DETECT: The -SAME_PAGE is used to take advantage of fast consecutive accesses within Fast Page Mode DRAM page boundaries. This signal is an output asserted by the processor when the current address is within the same page as the previous memory access. The -SAME_PAGE signal is asserted with -AS and remains active for one processor cycle. -SAME_PAGE is never asserted in the first transaction following a transaction by another device on the bus. The page size is specified by writing the SAME-PAGE MASK register.
-CS0, -CS1, -CS2, -CS3, -CS4, -CS5	O S(L) G(1) I(1)	CHIP SELECTS: These outputs are asserted when the value on the address bus matches the address range in one of the corresponding ADDRESS RANGE registers. The signals are used to decode the current address into one of six address ranges. Address ranges should not overlap. Each address range has a corresponding wait specifier which is used to automatically assert the -READY signal after a user defined number of processor clock cycles. This allows a variety of memory and I/O devices with different access times to be connected to the MB86931 without the need for additional logic.
ADR <31:2>	O S(L) G(Z) I(1)	ADDRESS BUS: The 30-bit ADDRESS BUS (A31-A2) is an output which identifies the data or instruction address of a 32-bit word. Reads are always one word in size while byte, half-word, or word transaction sizes for writes is identified by separate byte-enable signals (-BE0-3). The address bus is valid for the duration of the bus transaction.

Signal	Type	Function																																																																																	
ASI <7:0>	O S(L) G(Z) I (1)	<p>ADDRESS SPACE IDENTIFIERS: The ADDRESS SPACE IDENTIFIERS are outputs which indicate to which of 256 available spaces the current ADDRESS BUS value corresponds. The ASI values are defined as follows:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>ASI <7:0></th> <th>ADDRESS SPACE</th> </tr> </thead> <tbody> <tr><td>0x1</td><td>Control Registers</td></tr> <tr><td>0x2</td><td>Instruction Cache Lock</td></tr> <tr><td>0x3</td><td>Data Cache Lock</td></tr> <tr><td>0x4 - 0x7</td><td>Application Definable</td></tr> <tr><td>0x8</td><td>User Instruction Space</td></tr> <tr><td>0x9</td><td>Supervisor Instruction Space</td></tr> <tr><td>0xA</td><td>User Data Space</td></tr> <tr><td>0xB</td><td>Supervisor Data Space</td></tr> <tr><td>0xC</td><td>Instruction Cache Tag RAM</td></tr> <tr><td>0xD</td><td>Instruction Cache Data RAM</td></tr> <tr><td>0xE</td><td>Data Cache Tag RAM</td></tr> <tr><td>0xF</td><td>Data Cache Data RAM</td></tr> <tr><td>0x10 - 0xFD</td><td>Application Definable</td></tr> <tr><td>0xFE - 0xFF</td><td>Reserved for Debug Hardware</td></tr> </tbody> </table> <p>The ASI values specified as “application definable” can be used by supervisor mode instructions such as Load Alternate and Store Alternate. The ASI value is available in the same cycle in which the corresponding address value is asserted on the address bus. The ASI pins are valid for the duration of the bus transaction. ASI values 0x8, 0x9, 0xA, and 0xB are cacheable.</p>	ASI <7:0>	ADDRESS SPACE	0x1	Control Registers	0x2	Instruction Cache Lock	0x3	Data Cache Lock	0x4 - 0x7	Application Definable	0x8	User Instruction Space	0x9	Supervisor Instruction Space	0xA	User Data Space	0xB	Supervisor Data Space	0xC	Instruction Cache Tag RAM	0xD	Instruction Cache Data RAM	0xE	Data Cache Tag RAM	0xF	Data Cache Data RAM	0x10 - 0xFD	Application Definable	0xFE - 0xFF	Reserved for Debug Hardware																																																			
ASI <7:0>	ADDRESS SPACE																																																																																		
0x1	Control Registers																																																																																		
0x2	Instruction Cache Lock																																																																																		
0x3	Data Cache Lock																																																																																		
0x4 - 0x7	Application Definable																																																																																		
0x8	User Instruction Space																																																																																		
0x9	Supervisor Instruction Space																																																																																		
0xA	User Data Space																																																																																		
0xB	Supervisor Data Space																																																																																		
0xC	Instruction Cache Tag RAM																																																																																		
0xD	Instruction Cache Data RAM																																																																																		
0xE	Data Cache Tag RAM																																																																																		
0xF	Data Cache Data RAM																																																																																		
0x10 - 0xFD	Application Definable																																																																																		
0xFE - 0xFF	Reserved for Debug Hardware																																																																																		
-BE3-0	O S(L) G(Z) I (O)	<p>BYTE ENABLES (O): Indicate whether the current load or store transaction is a byte, half-word or word transaction. The BYTE ENABLE value is available in the same cycle in which the corresponding address value is asserted on the address bus. The values on the byte enable pins are valid for load and store operations and for the duration of the bus transaction. Since the processor extracts the appropriate byte or halfword from the word being read, the byte enable signals can be ignored during load operations.</p> <p>Possible values for -BE3-0 are as follows:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th></th> <th colspan="4">Byte 0</th> <th colspan="4">Byte 1</th> <th colspan="4">Byte 2</th> <th colspan="4">Byte 3</th> </tr> <tr> <th></th> <th>31</th><th>24</th><th>23</th><th>16</th> <th>15</th><th>8</th><th>7</th><th>0</th> <th>31</th><th>24</th><th>23</th><th>16</th> <th>15</th><th>8</th><th>7</th><th>0</th> </tr> </thead> <tbody> <tr> <td>Byte Writes</td> <td>1</td><td>1</td><td>1</td><td>0</td> <td>1</td><td>1</td><td>0</td><td>1</td> <td>1</td><td>1</td><td>0</td><td>1</td> <td>1</td><td>0</td><td>1</td><td>1</td> </tr> <tr> <td>Half-Word Writes</td> <td colspan="4">1 1 0 0</td> <td colspan="4">0 0 1 1</td> <td colspan="4"></td> </tr> <tr> <td>Word Writes</td> <td colspan="8">0 0 0 0</td> <td colspan="8"></td> </tr> </tbody> </table>		Byte 0				Byte 1				Byte 2				Byte 3					31	24	23	16	15	8	7	0	31	24	23	16	15	8	7	0	Byte Writes	1	1	1	0	1	1	0	1	1	1	0	1	1	0	1	1	Half-Word Writes	1 1 0 0				0 0 1 1								Word Writes	0 0 0 0															
	Byte 0				Byte 1				Byte 2				Byte 3																																																																						
	31	24	23	16	15	8	7	0	31	24	23	16	15	8	7	0																																																																			
Byte Writes	1	1	1	0	1	1	0	1	1	1	0	1	1	0	1	1																																																																			
Half-Word Writes	1 1 0 0				0 0 1 1																																																																														
Word Writes	0 0 0 0																																																																																		

Signal	Type	Function
D <31:0>	I/O S(L) G(Z) I (Z)	<p>DATA BUS: The bus interface has 32 bidirectional data pins (D31-D0) to transfer data in thirty-two bit quantities. D(31) corresponds to the most significant bit of the least significant byte of the 32-bit word. A double word is aligned on an 8-byte boundary, a word is aligned on a 4-byte boundary, and a half-word is aligned on a 2-byte boundary. If a load or store of any of these quantities is not properly aligned, a Not Aligned Trap will occur in the processor.</p> <p>In write bus cycles, the point at which data is driven onto the bus depends on the type of the preceding cycle. If the preceding cycle was a write, data is driven in the cycle immediately following the cycle in which -READY was asserted. If the preceding cycle was a read, data is driven one cycle after the cycle in which -READY was asserted to minimize bus contention between the processor and the system. All bits of the data bus are driven regardless of word size. The values on the pins not corresponding to the byte or half-word being written are undefined.</p>
-AS	O S(L) G(Z) I (1)	<p>ADDRESS STROBE: A control signal asserted by the MB86931 or other bus master to indicate the start of a new bus transaction. A bus transaction begins with the assertion of -AS and ends with the assertion of -READY. -AS remains asserted for 1 clock cycle. During cycles in which neither the processor nor another bus master is driving the bus the bus is idle, and -AS remains de-asserted.</p>
RD/ -WR	O S(L) G(Z) I (1)	<p>READ/BUS TRANSACTION: This signal specifies whether the current bus transaction is a read or a write operation. When -AS is asserted and RD/-WR is low, then the current transaction is a write. With -AS asserted and RD/-WR high, the current transaction is a read. RD/-WR remains active for the duration of the bus transaction and is de-asserted with the assertion of -READY.</p>
-READY	I S(L)	<p>READY: This is a control signal asserted by the external memory system to indicate that the current bus transaction is being completed and that it is ready to start with the next bus transaction in the following cycle. In case of a fetch from memory, the processor will strobe the value on the data bus at the rising edge of CLK_IN following the assertion of -READY. For the case of a write, the memory system will assert -READY when the appropriate access time has been met.</p> <p>In most cases, no additional logic is required to generate the -READY signal. On-chip circuitry can be programmed to assert -READY based on the address of the current transaction. The external system can override the internal ready generator to terminate the current bus cycle early. Up to 6 address ranges each with different transaction times can be programmed.</p>
CLK_ECB	I	<p>EXTERNAL CLOCK BYPASS: Tying this signal high causes the CLK_IN signal to bypass the Phases Lock Loop (PLL). This signal is used for testing of the chip.</p>
EMU_SD <3:0>	I/O	<p>EMULATOR STATUS/DATA BITS: Bi-directional pins used by a hardware emulator to control and monitor MB86931 execution. These pins should be left unconnected.</p>
EMU_D<3:0>	I/O	<p>EMULATOR DATA BITS: Bi-directional pins used by a hardware emulator to control and monitor MB86931 execution. These pins should be left unconnected.</p>

Signal	Type	Function
EMU_BRK	I	EMULATOR BREAK REQUEST LINE: Input used by a hardware emulator to request a trap when emulation is enabled. This pin should be left unconnected.
-EMU_ENB	I	EMULATOR ENABLE: Tied low while the MB86931 is being reset to enable hardware emulator mode on the chip. This pin should be left unconnected.
TCK	I	TEST CLOCK: JTAG compatible test clock input.
TMS	I	TEST MODE: JTAG compatible test mode select pin.
TDI	I	TEST DATA IN: JTAG compatible test data input.
TDO	O	TEST DATA OUT: JTAG compatible test data output.
-TRST	I	TEST RESET: Asynchronous reset for JTAG logic. If not using JTAG, this signal must be pulled low.

NOTE:

I = Input Only Pin	G(...)	I(...)
O = Output Only Pin	= While the bus is granted to another bus master (-BGRNT=asserted), the pin is	= While the bus is between bus cycles (or being reset) and is not granted to another bus master, the pin is
I/O = Either Input or Output Pin	G(1) is driven to V _{CC}	I (1) is driven to V _{CC}
- = Pins "must be" connected as described	G(0) is driven to V _{SS}	I (0) is driven to V _{SS}
A(L) = Asynchronous: Inputs may be asynchronous to CLKOUT.	G(Z) floats	I (Z) floats
S(L) = Synchronous: Inputs must meet setup and hold times relative to CLK_IN. Outputs are Synchronous to CLK_IN	G(Q) is a valid output	I (Q) is a valid output

5.1.2 Interrupt Request Signal Description

Signal	Type	Function
IRQ<15:1>	I	Interrupt Request. These are prioritized system interrupt requests. IRQ15 has the highest priority, and IRQ1 the lowest. The trigger for each interrupt can be programmed for a high level, a low level, a rising edge, or a falling edge. The level-trigger interrupt request signals are sampled during three successive internal clock periods to minimize false interrupts.

5.1.3 Timer Signal Descriptions

Signal	Type	Function
CLK<3:0>	I	Timer external clock input. In the external clock mode, this signal is synchronized with the internal clock before use. These pins should be tied high or low when not used.
OUT<3:0>	O	Timer output pin. According to the mode, the output wave functions as (1) periodic interrupt signal output; (2) square wave output; (3) one-shot pulse output. These pins are low during reset.
IN<3:0>	I	Count control input. These inputs are used as gate signals in Modes 0 to 3, and as external triggers in Mode 4.
ACK0 ACK1	I	Asynchronous clock. These are prescaler input clocks that are used when selected in the Prescaler registers. The clocks are synchronized with the internal clock and are divided and output to the PRSCKx pin. When not used, they should be tied low.
PRSCK0 PRSCK1	O	Prescaler output.

5.1.4 Serial Port Signal Descriptions

Signal	Type	Function
-DSR0 -DSR1	I	Modem Data Set Ready signal. The status of these pins is loaded into bit 7 of the corresponding SDTR status register.
-RTS0 -RTS1	O	Modem Request to Send signal. When bit 5 of the command register is set to 1, these signals are driven low.
-DTR0 -DTR1	O	Modem Data Terminal Ready or Rate Select signal. When bit 1 of the command register is set to 1, these signals are driven low.
-CTS0 -CTS1	I	Modem Clear to Send signal. A transmitter is enabled only when its corresponding -CTSx signal is low.
TRNDT0 TRNDT1	O	Serial transmit data. Parallel data written in the data register is converted into serial data, then transmitted through these pins. In the asynchronous mode, start and stop bits are added to data, and a parity bit can be added. If there is no data to be transmitted, the SDTR transmits synchronous characters in the synchronous mode, and enters the mark state in the asynchronous mode. The mark state also occurs after a transmit disable command is specified (bit 0 of the command register is set to 0) or when -CTS is High. Note that the mark state occurs during transmission after: (1) One byte is transmitted if a transmit disable command is specified during transmission; (2) the second synchronous character is transmitted if the first synchronous character was transmitted (with the synchronous state held) in the BISYNC mode.

Signal	Type	Function
TxEMP0 TxEMP1	O	These signals are driven high if there is no data to be transmitted in the SDTR. These signals are driven low at the falling edge of the write signal when the processor writes a byte to be transmitted.
TxRDY0 TxRDY1	O	These signals are driven low if the transmit data buffer register becomes empty with the -CTS pin low and the transmitter is enabled.
-TCLK0 -TCLK1	I	<p>Clock for determining the transmission baud rate.</p> <p>In the synchronous mode, since the baud rate is fixed at transmit clock x 1, the frequency of the clock to be input to the -TCLK pin is the transmission baud rate.</p> <p>In the asynchronous mode, the transmit clock x 1/16 and x 1/64 frequencies will be the transmission baud rate in accordance with the baud rate set in the mode register. For example, if a clock of 19.2 kHz is input to the -TCLK pin, the transmission baud rate is 1200 bauds at x 1/16, and 300 bauds at x 1/64. The transmit data is synchronized with the falling edge of this transmit clock.</p>
RCVDT0 RCVDT1	I	Serial receive data input. The input data is converted to parallel data in the SDTR and can be read via the system data bus.
RCLK0 RCLK1	I	<p>Clock for determining the receive baud rate. In the synchronous mode, since the baud rate is fixed at receive clock x 1, the frequency of the clock to be input to the RCLK pin is the receive baud rate. In the asynchronous mode, the receive clock x 1/16 and x 1/64 frequencies will be the receive baud rate in accordance with the baud rate set in the mode register. For example, if a clock of 19.2 kHz is input to the RCLK pin, the receive baud rate is 1200 bauds at x 1/16, and 300 bauds at x 1/64. The receive data is sampled at the falling edge of this receive clock.</p>
SYBRK0 SYBRK1	I/O	<p>SYBRK0/SYBRK1. When the external synchronous mode is set in the mode register, synchronous signals are output from these pins. If H-level signals are input to these pins when RCLK is high during hunt, the data sampled at the rising edge of the next RCLK will be the start bit of the received data. When the internal synchronous mode is selected, these pins are used as synchronous character detection pins. If the received data coincides with the data loaded in the synchronous character register (in the BISYNC mode, data for two characters coincide with each other), these are driven high. Then, when the MPU reads data out of the status register, these pins are driven low at the end of the read-out signal strobe. When used in the asynchronous mode, these signals function as break code detection signals. If the received data (including start, stop, and parity bits) is all 0s immediately after a framing error occurs, these signals are driven high. The signals are released when reset is executed or when 1 data is received.</p>
RxRDY0 RxRDY1	O	These pins are driven high when the serial data received at the RCVDT pin is converted to parallel data in the SDTR, allowing the processor to read the data. The signals are driven low when the processor reads the data.

CHAPTER A6



MB86931 JTAG

6.1 MB86931 JTAG Pin List

The JTAG cells are arranged in a shift register configuration (see Figure A6-8). When shifting in a JTAG pattern through TDI, the LSB should correspond to the JTAG cell value for `-TIMER_OVF` pin whereas, the MSB of the pattern should correspond to the `CLK_ENB` pin's JTAG cell. As far as JTAG output through TDO is concerned, the first bit out corresponds to `-TIMER_OVF` JTAG cell value and the last output bit corresponds to the `CLK_ENB` JTAG cell value. Table A6-1 lists the order of all of the JTAG cells.

Table A6-1:JTAG Pin Order

Order	JTAG Cell	JTAG Cell Type	Function
1	<code>-TIMER_OVF</code>	output	Timer Overflow pin
2	<code>XTAL1</code>	input	Crystal input
3	<code>_EMU_BRK</code>	input	Emulator break input
4	<code>icediojo[†]</code>	output	Bidirectional control for EMU_D/EMU_SD buses icediojo = 1: EMU_D and EMU_SD buses are input icediojo = 0: EMU_D and EMU_SD buses are output
5	<code>EMU_SD_i<3></code>	input	Input bit 3 of EMU_SD<3:0> bus
6	<code>EMU_SD_o<3></code>	output	Output bit 3 of EMU_SD<3:0> bus
7	<code>EMU_SD_i<2></code>	input	Input bit 2 of EMU_SD<3:0> bus
8	<code>EMU_SD_o<2></code>	output	Output bit 2 of EMU_SD<3:0> bus

Table A6-1: JTAG Pin Order (Continued)

Order	JTAG Cell	JTAG Cell Type	Function
9	EMU_SD_i<1>	input	Input bit 1 of EMU_SD<3:0> bus
10	EMU_SD_o<1>	output	Output bit 1 of EMU_SD<3:0> bus
11	EMU_SD_i<0>	input	Input bit 0 of EMU_SD<3:0> bus
12	EMU_SD_o<0>	output	Output bit 0 of EMU_SD<3:0> bus
13	EMU_D_i<3>	input	Input bit 3 of EMU_D<3:0> bus
14	EMU_D_o<3>	output	Output bit 3 of EMU_D<3:0> bus
15	EMU_D_i<2>	input	Input bit 2 of EMU_D<3:0> bus
16	EMU_D_o<2>	output	Output bit 2 of EMU_D<3:0> bus
17	EMU_D_i<1>	input	Input bit 1 of EMU_D<3:0> bus
18	EMU_D_o<1>	output	Output bit 1 of EMU_D<3:0> bus
19	EMU_D_i<0>	input	Input bit 0 of EMU_D<3:0> bus
20	EMU_D_o<0>	output	Output bit 0 of EMU_D<3:0> bus
21	iceenblio [†]	output	bidirectional control signal for –EMU_ENB pin iceenblio = 1: –EMU_ENB pin is an input iceenblio = 0: –EMU_ENB pin is an output
22	–EMU_EN_i	input	Input bit of –EMU_ENB pin
23	–EMU_EN_o	output	Output bit of –EMU_ENB pin
24	dbusiojo [†]	output	Bidirectional control signal D<31:0> bus dbusiojo = 1: D<31:0> bus is an input dbusiojo = 0: D<31:0> bus is an output
25	D_i<31>	input	Input bit 31 of D<31:0> bus
26	D_o<31>	output	Output bit 31 of <31:0> bus
:	:	:	:
87	D_i<0>	input	Input bit 0 of <31:0> bus
88	D_o<0>	output	Output bit 0 of <31:0> bus
89	–RESET	input	Chip reset pin
90	–BREQ	input	Bus request input
91	–MEXC	input	Memory exception input
92	–READY	input	External memory transaction complete signal
93	tstatejo [†]	output	Three-state control signal for ADR, ASI, –BE, –AS, RD/WR and –LOCK If tstatejo = 1: signals are three-stated. If tstatejo = 0: signals are outputs.
94	–BGRNT	output	Bus grant output signal
95	–ERROR	output	Error output signal
96	–LOCK	output	Bus lock output signal

Table A6-1: JTAG Pin Order (Continued)

Order	JTAG Cell	JTAG Cell Type	Function
97	-RD/WR	output	Memory Read/Write output signal
98	-AS	output	Start of memory transaction output signal
99	-DSR0	input	
100	-CTS0	input	
101	-TCLK0	input	
102	-RCLK0	input	
103	RCVDT0	input	
104	xscnto0jo	output	Bidirectional control signal for SYBRK0 pin xscnto1jo = 1: SYBRK0 is an input xscnto1jo = 0: SYBRK0 is an output
105	SYBRK0_i	input	
106	SYBRK0_o	output	
107	-RTS0	output	
108	-DTR0	output	
109	TRNDT0	output	
110	TxEMP0	output	
111	TxRDY0	output	
112	RxRDY0	output	
113	PRSCK0	output	
114	OUT0	output	
115	IN0	input	
116	ACK0	input	
117	CLK0	input	
118	CLK2	input	
119	IN2	input	
120	OUT2	output	
121	IRQ1	input	
:	:	:	
135	IRQ15	input	
136	OUT3	output	
137	IN3	input	
138	CLK3	input	
139	CLK1	input	
140	ACK1	input	

Table A6-1: JTAG Pin Order (Continued)

Order	JTAG Cell	JTAG Cell Type	Function
141	IN1	input	
142	OUT1	output	
143	PRSCK1	output	
144	RxRDY1	output	
145	TxRDY1	output	
146	TxEMP1	output	
147	TRNDT1	output	
148	-DTR1	output	
149	-RTS1	output	
150	xscnto1jo	output	Bidirectional control signal for SYBRK1 pin xscnto0jo = 1: SYBRK1 is an input xscnto0jo = 0: SYBRK1 is an output
151	SYBRK1_i	input	
152	SYBRK1_o	output	
153	RCVDT1	input	
154	RCLK1	input	
155	-TCLK1	input	
156	-CTS1	input	
157	-DSR1	input	
158	-CS<0>	output	
159	-CS<1>	output	
160	-CS<2>	output	
161	-CS<3>	output	
162	-CS<4>	output	
163	-CS<5>	output	
164	-SAMEPAGE	output	
165	BE<3>	output	
166	BE<2>	output	
167	BE<1>	output	
168	BE<0>	output	
169	ASI<0>	output	
170	ASI<1>	output	
171	ASI<2>	output	
172	ASI<3>	output	

Table A6-1: JTAG Pin Order (Continued)

Order	JTAG Cell	JTAG Cell Type	Function
173	ASI<4>	output	
174	ASI<5>	output	
175	ASI<6>	output	
176	ASI<7>	output	
177	ADR<2>	output	
:	:	:	
206	ADR<31>	output	
207	TEST<3>	input	Factory test pin
208	TEST<2>	input	Factory test pin
209	TEST<1>	input	Factory test pin
210	TEST<0>	input	Factory test pin
211	CLK_ENB	input	

†. These are internal I/O control signals. Therefore, there are no corresponding external pins.

1. The following pins are not three-statable: -SAME_PAGE, -CS<5:0>, -BGRNT, TIMER_OVF, -ERROR.

2. The following pins have no corresponding JTAG cells: CLKOUT1, CLKOUT2, XTAL2, -TRST, TCK, TMS, TDI, TDO.



Section 3

.....

MB86932

2.2 Programmer's Model	B2-6
2.2.1 Cache/Bus Interface Unit Control Register	B2-6
2.2.2 Context Table Pointer Register	B2-6
2.2.3 Context Register	B2-7
2.2.4 TLB Exceptions	B2-7
2.2.5 Instruction Fault Status Register	B2-8
2.2.6 Data Fault Status Register	B2-9
2.2.7 TLB Control Register	B2-10
2.2.8 TLB Data Fault Address Register	B2-11
2.2.9 Most Recently Used Register	B2-12
2.2.10 The TLB Entry	B2-12
2.2.11 The TLB CAM Entry	B2-12
2.2.12 The TLB RAM Entry	B2-14
2.2.13 ITLB Description	B2-16
2.2.14 TLB Lookup	B2-16
2.3 Internal Architecture	B2-16
2.3.1 Details of TLB Logic	B2-16
2.3.2 Address Translation: Logical and Physical Steps	B2-17
2.3.3 Basic TLB Exception Timings	B2-18
2.3.4 TLB Timing Considerations	B2-19
2.3.5 TLB Emulation Support Logic	B2-20
2.4 Programming Considerations	B2-21
2.4.1 MMU Architecture Example: the SPARC Reference MMU	B2-21
2.4.2 Virtual Address format	B2-23
2.4.3 Physical Address format	B2-24
2.5 Conformity to SPARC Reference MMU Architecture	B2-24

Chapter B3: MB86932 Caches

3.1 Overview of MB86932 Caches	B3-1
3.2 Programmer's Model	B3-2
3.2.1 Operation of the Instruction Cache	B3-3
3.2.2 Operation of the Data Cache	B3-3

3.3 Internal Architecture of MB86932 Caches	B3-3
3.3.1 Instruction Cache.....	B3-4
3.3.2 Read Hit.....	B3-5
3.3.3 Miss Processing.....	B3-5
3.3.4 Data Cache	B3-6
3.3.5 Read Hit.....	B3-8
3.3.6 Write Hit.....	B3-8
3.3.7 Miss Processing.....	B3-8
3.3.8 Atomic Load and Store.....	B3-8

Chapter B4: MB86932 Bus Interface Unit

4.1 Overview of Bus Interface Unit.....	B4-1
4.2 Burst Mode	B4-1
4.2.1 Overview	B4-1
4.2.2 Burst Mode Interface Pins	B4-2
4.2.3 Burst Mode Fetch Sequence	B4-2
4.2.4 Bus Mode control bits	B4-3
4.2.5 PROM Address Space.....	B4-3
4.2.6 Prefetch Buffer.....	B4-3
4.2.7 Cache Off	B4-3
4.2.8 Bus Request	B4-3
4.2.9 Memory Exception (Instruction fetches or Data loads)	B4-4
4.2.10 Memory Exception (DMA)	B4-4
4.2.11 Non-cacheable Accesses.....	B4-4
4.2.12 Interface Timing.....	B4-4
4.3 Parity	B4-6
4.4 Wait State Specifier Register	B4-7
4.4.1 Purpose.....	B4-7
4.4.2 Format	B4-7
4.4.3 Same Page Mode.....	B4-8
4.4.4 Burst Mode	B4-8

4.5 ROM Interface	B4-9
4.5.1 Purpose	B4-9
4.5.2 Features	B4-9
4.5.3 Bus Configuration on Reset	B4-9
4.5.4 System Interface	B4-10
4.5.5 PROM Address Space	B4-10
4.5.6 Load/Stores	B4-11
4.5.7 Burst Mode	B4-12
4.5.8 Memory Exception	B4-12
4.5.9 Bus Request	B4-12
4.5.10 Timing	B4-12
4.6 Processor Bus Request	B4-13
4.6.1 Purpose	B4-13
4.6.2 Features	B4-13
4.7 BIU Timing	B4-14
4.7.1 Effect of TLB	B4-14
4.8 BIU Priorities	B4-15

Chapter B5: MB86932 DMA

5.1 Overview	B5-1
5.2 Programmer's Model	B5-4
5.2.1 DMA Priority	B5-4
5.2.2 DP/Source/Destination ASI Register	B5-5
5.2.3 Current Source Address Register	B5-5
5.2.4 Current Destination Address Register	B5-6
5.2.5 Current Byte Count Register	B5-6
5.2.6 Descriptor Pointer Register	B5-7
5.2.7 Channel Control Register	B5-7
5.2.8 Channel Status Register	B5-9
5.2.9 Channel Initialization	B5-9
5.2.10 Buffer Chaining Data Structure	B5-10
5.2.11 DMA Initialization	B5-11
5.2.12 Basic DMA Timing	B5-11
5.2.13 Error Conditions	B5-11
5.3 External Interface	B5-12
5.3.1 Transfer Protocols	B5-12

Chapter B6: MB86932 DSU

6.1 Overview B6-1

6.2 Programmer's Model B6-1

 6.2.1 New Registers and Flags B6-1

 6.2.2 Logic of Context Comparison B6-3

Chapter B7: MB86932 External Interface

7.1 SIGNAL DESCRIPTIONS B7-1

Chapter B8: MB86932 JTAG

8.1 MB86932 JTAG Pin List B8-16

each word of the line. The data cache is a physical cache; that is, it is accessed with a physical, not a virtual, address. When data is to be removed from the cache, the cache can be invalidated in a single cycle; likewise, "locked" data in the cache can be unlocked in a single cycle.

- **On-Chip DMA:** The MB86932 has two DMA channels. Each channel supports two transfer types: contiguous block and chained block transfers. The DMA also supports three transfer protocols: single-datum transfer, block transfer, and demand transfer (where data moves continue as long as an external device requests it). Four data types are supported: byte, halfword, word, and quad-word. For byte and halfword, the DMA does all the required packing/unpacking. Each channel also supports either fly-by or flow-thru transfer modes, and each can be started by either software or external hardware requests. The addressing convention for accesses is "big_endian."
- **Configurable External Data Bus:** The MB86932 includes a data bus that can be configured at Reset as 8, 16, or 32 bits wide (when in the address space selected by chip select 0). This enables the MB86932 to boot from a single by-8 or by-16 ROM.
- **Burst Mode:** The MB86932 supports two data- and instruction-accessing modes to external memory: normal and burst. In normal mode, it accepts a single datum per address, driven externally. In burst mode, it accepts 4 words per address, driven externally. Burst mode stores are supported only as part of DMA requests, and no burst mode transfers are supported in 8/16 bit mode.

1.2 Programmer's Model of the MB86932

1.2.1 User-visible Registers

All the special-purpose registers and ASR registers defined on the MB86930 exist also on the MB96832.

All on-chip control/status/data registers which exist in alternate address spaces in the MB86930, with one exception, exist also on the MB86932 in backwards-compatible format. The one exception is the Instruction Tags, whose format has changed.

The increase in cache and the addition of new peripherals in the MB86932 have made it necessary to add new registers, accessible through alternate address spaces; these are described below in 3.4.1.1. All on-chip memory-mapped control/status registers for these new features are mapped into ASI=0x01, 0x02, 0x03, 0x0C, 0x0D, 0x0E, or 0x0F. The BIU recognizes that these ASI's are mapped to internal registers rather than memory, and does not assert the external ASI pins (or any other pins) when doing accesses in these ASI spaces. Since the address calculated by the IU for any register of this class is its physical address, no address translation is necessary, and the TLB is not involved

In the lists that follow, an appended asterisk (*) = “new in MB86932”; a double asterisk (**)= “changed from equivalent in MB86930.”

Cache/BIU control/status registers:

ASI: 0x01

Address range: 0x00000000-0x000000FF

0x00000000	ASI=0x1	Cache/BIU Control Register** (TLB enable bit added)
0x00000004	ASI=0x1	Lock Control Register
0x00000008	ASI=0x1	Lock Control Save Register
0x0000000C	ASI=0x1	Cache Status Register
0x00000010	ASI=0x1	Restore Lock Control Register
0x00000020	ASI=0x1	Bus Control Register*
0x00000080	ASI=0x1	System Support Control Register** (DMA priority; even/odd paritybits added)

Peripheral control/status registers:

ASI: 0x01

Address range: 0x00000100-0x000001FF

0x00000120	ASI=0x1	Same Page Mask Register
0x00000124	ASI=0x1	Address Range Specifier Register 1
0x00000128	ASI=0x1	Address Range Specifier Register 2
0x0000012C	ASI=0x1	Address Range Specifier Register 3
0x00000130	ASI=0x1	Address Range Specifier Register 4
0x00000134	ASI=0x1	Address Range Specifier Register 5
0x00000140	ASI=0x1	Address Mask Register 0
0x00000144	ASI=0x1	Address Mask Register 1
0x00000148	ASI=0x1	Address Mask Register 2
0x0000014C	ASI=0x1	Address Mask Register 3
0x00000150	ASI=0x1	Address Mask Register 4
0x00000154	ASI=0x1	Address Mask Register 5
0x00000160	ASI=0x1	Wait State Specifier Register** (SGL cycle/parity bit added)
0x00000164	ASI=0x1	Wait State Specifier Register** (SGL cycle/parity bit added)
0x00000168	ASI=0x1	Wait State Specifier Register** (SGL cycle/parity bit added)
0x00000174	ASI=0x1	Timer Register
0x00000178	ASI=0x1	Timer Preload Register
0x00000180	ASI=0x1	Source/Destination ASI Register (DMA0)*
0x00000184	ASI=0x1	Current Source Address Register (DMA0)*
0x00000188	ASI=0x1	Current Destination Address Reg (DMA0)*
0x0000018C	ASI=0x1	Current Byte Count Register (DMA0)*
0x00000190	ASI=0x1	Descriptor Pointer (DP) Register (DMA0)*
0x00000194	ASI=0x1	Channel Control Register (DMA0)*
0x00000198	ASI=0x1	Channel Status Register (DMA0)*
0x000001A0	ASI=0x1	Source/Destination ASI Register (DMA1)*
0x000001A4	ASI=0x1	Current Source Address Register (DMA1)*
0x000001A8	ASI=0x1	Current Destination Address Reg (DMA1)*
0x000001AC	ASI=0x1	Current Byte Count Register (DMA1)*
0x000001B0	ASI=0x1	Descriptor Pointer (DP) Register (DMA1)*

0x000001B4	ASI=0x1	Channel Control Register (DMA1)*
0x000001B8	ASI=0x1	Channel Status Register (DMA1)*

ASSP Control/status registers: *

ASI: 0x01

Address range: 0x00000200-0x000002FF

Note: This space is reserved for additional control/status registers for possible future derivatives of the SPARClite family of products.

TLB Entries: *

ASI: 0x01

Address range: 0x00000300-0x000003FF--

Note: This allows up to 32 entries in the TLB, although only 16 entries are used in the MB86932. The TLB can be read or written by the "lda" or "sta" instructions.

0x00000300	ASI=0x1	TLB RAM Entry 1
0x00000304	ASI=0x1	TLB CAM Entry 1
...
...	...	***other TLB entries***
...
0x00000378	ASI=0x1	TLB RAM Entry 16
0x0000037C	ASI=0x1	TLB CAM Entry 16

TLB Status/Control Registers: *

ASI: 0x01

Address range: 0x00000400-0x000004FF

Note: The TLB enable bit is in the Cache/BIU Control Register.

0x00000400	ASI=0x1	ITLB Register "RAM" Entry
0x00000404	ASI=0x1	ITLB Register "CAM" Entry
0x00000408	ASI=0x1	Context Register
0x0000040C	ASI=0x1	Context Table Pointer Register
0x00000410	ASI=0x1	TLB Control Register
0x00000414	ASI=0x1	Data Fault Status Register
0x00000418	ASI=0x1	Instruction Fault Status Register
0x0000041C	ASI=0x1	TLB Most Recently Used Register
0x00000420	ASI=0x1	TLB Data Fault Address Register

Emulation Registers:

ASI: 0x01

Address range: 0x0000FF00-0x0000FFFF

0x0000FF00	ASI=0x1	Instruction Address Descriptor Register 1
0x0000FF04	ASI=0x1	Instruction Address Descriptor Register 2
0x0000FF08	ASI=0x1	Data Address Descriptor Register 1
0x0000FF0C	ASI=0x1	Data Address Descriptor Register 2
0x0000FF10	ASI=0x1	Data Value Descriptor Register 1
0x0000FF14	ASI=0x1	Data Value Descriptor Register 2 or Mask Register
0x0000FF18	ASI=0x1	Debug Control Register **
0x0000FF1C	ASI=0x1	Debug Status Register
0x0000FF20	ASI=0x1	Context Compare Register **

Instruction Cache Lock Registers: **

ASI: 0x02

Address range: 0x00000000-0x00000FFF (Bank 1)

0x80000000-0x80000FFF (Bank 2)

Note: Writing to every eighth *word* address in this space can be used to initialize the lock bit for each line in the instruction cache. This differs from the MB86930, where every *fourth* word location is accessed.

Data Cache Lock Registers:

ASI: 0x03

Address range: 0x0000FF00-0x000003FF (Bank 1)

0x8000FF00-0x800003FF (Bank 2)

Note: Writing to every fourth *word* address in this space can be used to initialize the lock bit for each line in the data cache. This is unchanged from the MB86930

Instruction Cache Tag RAM: **

ASI: 0x0C

Address range: 0x00000000-0x00000FFF (Bank 1)

0x80000000-0x80000FFF (Bank 2)

Note: Writing to every eighth *word* address in this space can be used to initialize the tags for each line in the instruction cache. This differs from the MB86930, where every *fourth* word location is accessed.

Instruction Cache Invalidate Registers: *

ASI: 0x0C

Note: These registers are in addition to the Instruction Cache Tags which are accessed using ASI 0x0C.

0x00001000 Bank 1 Instruction Cache Invalidate (write only)

0x80001000 Bank 2 Instruction Cache Invalidate (write only)

Instruction Cache Data RAM: **

ASI: 0x0D

Address range: 0x00000000-0x00000FFF (Bank 1)

0x80000000-0x80000FFF (Bank 2)

Note: Writing to *word* addresses in this space can be used to initialize the values in the instruction cache.

Data Cache Tag RAM:

ASI: 0x0E

Address range: 0x00000000-0x000003FF (Bank 1)

0x80000000-0x800003FF (Bank 2)

Note: Writing to every fourth *word* address in this space can be used to initialize the tag bit for each line in the data cache. This is unchanged from the MB86930

Data Cache Invalidate Registers: *

ASI: 0x0E

Note: These registers are in addition to the Data Cache Tags which are accessed using ASI 0x0E.

0x00001000 Bank 1 Data Cache Invalidate (write only)

0x80001000 Bank 2 Data Cache Invalidate (write only)

Data Cache Data RAM:

ASI: 0x0F

Address range: 0x00000000-0x000003FF (Bank 1)

0x80000000-0x800003FF (Bank 2)

Note: Writing to *word* addresses in this space can be used to initialize the data RAM. This is unchanged from the MB86930

1.3 Internal Architecture of the MB86932

Figure B1-1, shows the general block diagram of the MB86932. Figure B1-2 shows in more detail the major units and buses connecting them. The solid lines show the bus connections that are used when the accesses are within the user or supervisor data/instruction ASI spaces (ASI 08, 09, 0A, and 0B). The dashed lines show the additional connections required to access the control/status or data registers through the alternate ASI spaces. The major buses are:

- **Data Data Bus (DD)**—A 32-bit bus used to transfer data to and from MB86932 functional units. In general, when a load is executed, data is transferred to the Integer Unit (IU) from one of the other units, and when a store is executed, data is transferred from the IU to one of the other units. When loads/stores to user or supervisor data space are performed, the DD gives the IU access to the Data Cache, the BIU (if the data is not in the cache), or the DSU (if the data is to be accessed out of DSU memory).

When doing Load Alternates or Store Alternates, the DD bus can access all units except the Instruction Cache and Instruction Tags, which can be accessed only through the ID bus. In such a case, the IU can read data (load alternate) or write data (store alternate) to the control/status/data registers of all units. Since the TLB can be accessed by the IU only through alternate space, their connection is shown as a dashed line.

- **Virtual Data Address bus (VDA)**—This 32-bit bus connects the IU, where the virtual address is generated, to the TLB, where the virtual address is translated to a 32-bit physical address. It also connects to the Debug Support Unit.
- **Physical Data Address bus (PDA)**—This 32-bit bus carries the physical address generated by the TLB. During loads/stores to user or supervisor data space, it is used both to access the Data Cache, and to compare against the Data Tags. The PDA bus also goes to the BIU for use when data is not cached, and has to be accessed from external memory.

For load alternates and store alternates, the PDA goes to all units (except for the I_cache and the I_tags), so that control/status/data registers can be accessed.

- **Instruction Data bus (ID)**—This 32-bit bus normally transfers instructions from either the Instruction Cache, the Bus Interface Unit, or the DSU (when code is being run out of DSU memory).

Note: When a store alternate is being performed to the I_cache or the I_tags (during cache initialization, for example), the data are first transferred from the IU to the BIU on the DD bus. The BIU then transfers the data on the ID bus to the I_cache or the I_tags. When a load alternate from the I_cache or the I_tags to the IU occurs, the reverse operation takes place. This obviates the need to extend both the ID and the DD busses to the I_cache and I_tags. (In the fig-

ure below, the connections for reading/writing the tags through alternate space are shown as dashed lines.)

- Virtual Instruction Address bus (VIA)—This 30-bit wide bus carries the 30-bit virtual address generated by the IU to the TLB for translation into a Physical Instruction Address (PIA). (Since instructions must fall on word boundaries, their addresses need only specify a full word address, for which 30 bits suffices.) The VIA also goes to the Debug Unit.
- Physical Instruction Address bus (PIA)—This carries the translated address from the TLB to the I_cache and I_tags, and to the BIU for use when the instruction is not cached, and access must be made to off-chip memory. The PIA can also be driven by the BIU when doing a store or load alternate to the I_cache or I_tags. In this case, the item to be stored is first sent to the BIU over the DD bus, with the address on the PDA bus. The BIU accepts this item, and drives it back on the ID bus, with the address on the PIA bus. This obviates the need to connect both the ID and DD buses to the I_cache and I_tags.
- Alternate Space Identifier address bus (ASI)—This 8-bit bus is driven by the IU, and indicates which address space a load/store is transferring data from/to. Load- and Store-Alternate instructions are used to read/write status/control/data registers in the various units of the MB86932.
- DMA Data Data bus (DDD)—This 32-bit local bus goes from the DMA to the BIU, and is used to send/receive data to/from the BIU during a DMA operation.
- DMA Data Address bus (DDA)—This 30-bit local bus goes from the DMA to the BIU, and is used to send the source or destination address to the BIU during a DMA operation.
- DMA ASI bus (DDASI)—This 8-bit local bus goes from the DMA to the BIU, and is used to send the ASI value to the BIU for cases where the DMA is addressing an alternate address space.

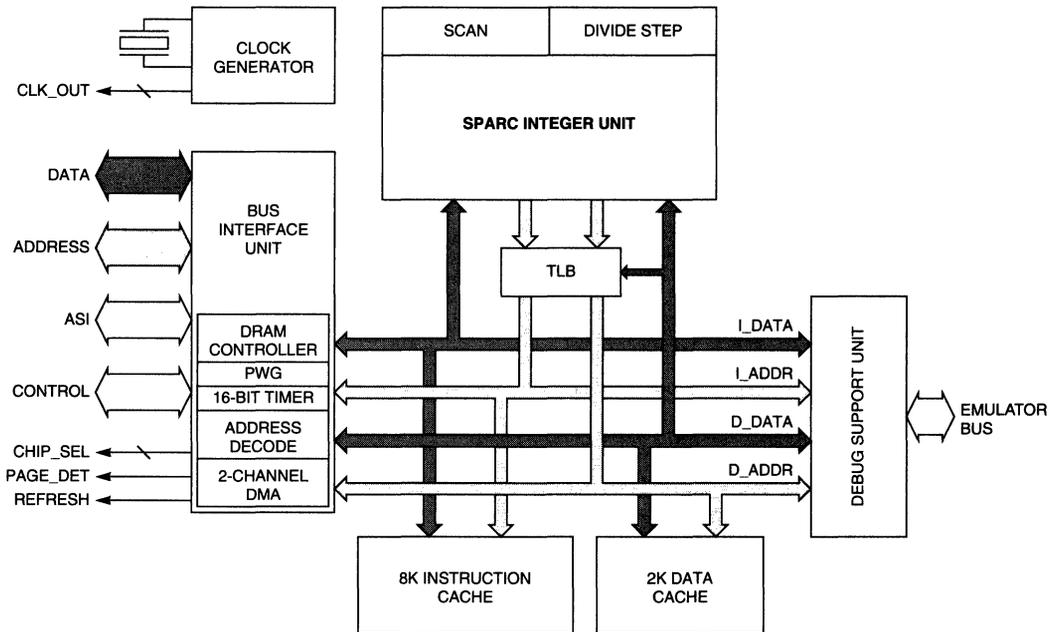


Figure B1-1. MB86932 Block Diagram

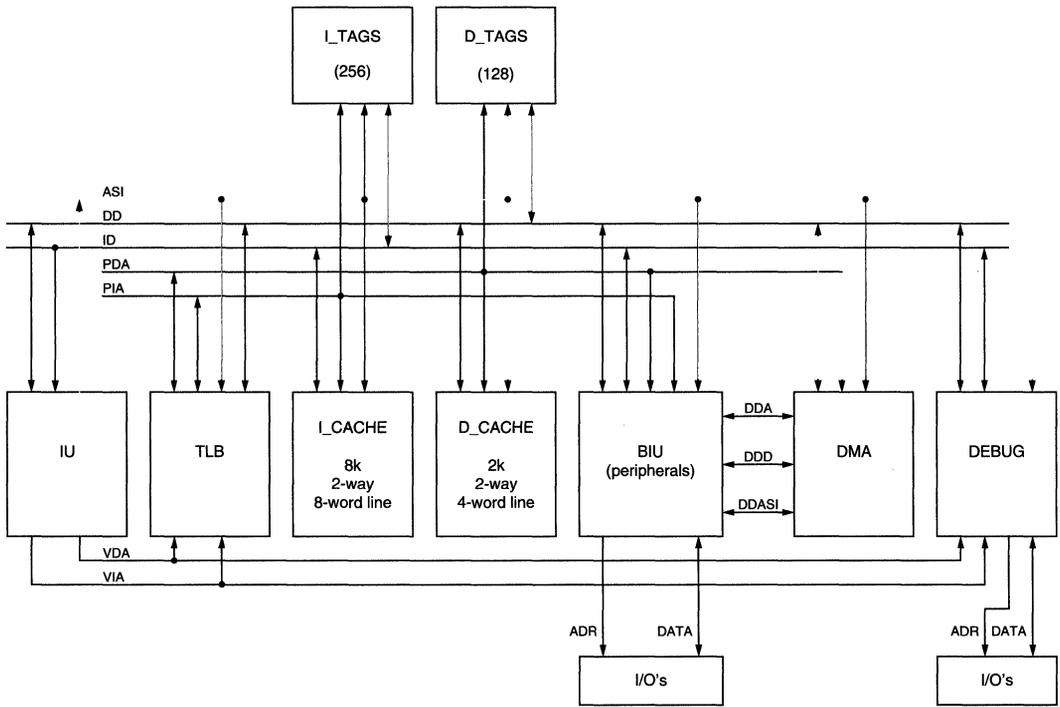


Figure B1-2. MB86932 Detailed Block Diagram

2.1.1 Memory Management Units: A General Description

This section provides a general description of MMU's, their function and benefits, for users that may be unfamiliar with them. It also defines terms that will be used throughout this chapter.

Figure B2-1 shows a block diagram of how an MMU fits with the CPU, cache, and memory. The MMU is responsible for doing address translations of the "virtual address" coming from the CPU to the "physical address" going to the cache and main memory. The "virtual address" is the address that the running program generates (from 0 up to 2^{32} for the SPARC Architecture). The "physical address" is the address that the hardware cache and memory receives. The CPU, as it runs code, produces virtual addresses which are dynamically translated to the physical address translation provided by the MMU. The benefits of virtual to physical address translation provided by the MMU are the following:

- Supports Virtual Memory
- Supports Multiple executing processes
- Supports Memory Protection

2.1.2 Virtual Memory

"Virtual memory" is the memory space that the program can address. For example, for the SPARC Architecture the virtual memory is 2^{32} bytes. "Physical memory" is the actual amount of memory (RAM, ROM, etc.) that is implemented in the system. Usually the physical memory is significantly smaller than the virtual memory.

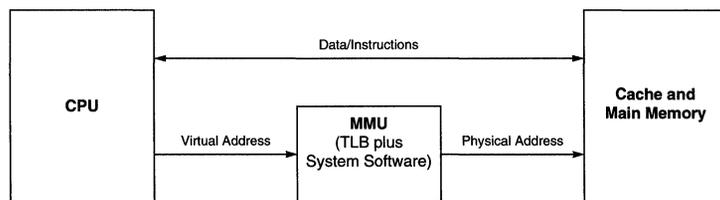


Figure B2-1. MMU's Role in Address Translation

Because this is true, it is necessary to dynamically allocate sections of this physical memory to code and data which is accessed by the program virtual address.

This is accomplished in the following way. The virtual memory space is broken into segments called "pages" (4k bytes in the SPARC Reference MMU). Similarly, the physical address space is broken into equivalent sized segments called "page

frames". The complete program and data can be stored in mass storage (e.g. disk) until requested by the running program. When the program requests data not in physical memory, the required page needs to be retrieved from mass storage and put into an available page frame in physical memory. The MMU keeps track of where each page is placed in physical memory through the use of an "address translation table", also called a "page table". In the most general case, the address translation table contains, for each virtual memory page address, either a corresponding physical memory page frame address or a pointer to mass storage where that virtual page can be found. As long as the page is in physical memory the MMU uses this table to translate virtual addresses to physical addresses as the program executes. When the page is not found in physical memory the MMU is responsible for retrieving pages in mass storage and placing them in physical memory so that they can be accessed by the program.

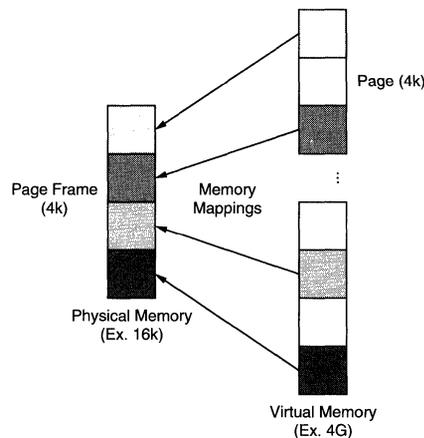


Figure B2-2. Memory Mappings

As an example, Figure B2-2 shows a physical address space of 16k bytes and a virtual address space of 4G bytes. The page size and the page frame size are both 4k bytes. The figure shows four virtual pages residing in physical memory. Other parts of the program would reside in mass storage (e.g. on disk).

Conceptually, both the virtual and physical address can be thought of as having two fields—the msb's making up the page number and the lsb's making up the offset (within the page). Effectively, the MMU's does address translation by taking the page number from the virtual address and replacing it with the corresponding physical page number from the address translation table. The offset remains the same.

2.1.3 Multiple Processes

A process is an “executing” program. At any time a process can be “running” on the CPU or “waiting” (e.g., waiting for I/O). Multiple processes can be executing at the same time but there can be only one running process. Each process may be using a number of physical page frames in memory. For example, the four page frames in Figure B2-2 could be holding 4 pages each of which could be associated with a different process.

Each executing program (or process) sees its own 2^{32} virtual address space. To support multiple processes there must be a way to translate between a process's virtual addresses and the physical addresses of that process's pages in memory. To accomplish this the MMU uses a “context register” the value of which is used to identify the process which is currently running. Also, required is a “context table pointer register”. The context table pointer register contains a pointer of the head of a table which in turn contains pointers to address translation tables for each process. The context register is used as an offset into this table. Thus, when a particular process is running the MMU must add the context to the context table pointer register to get the head of the address translation table for that process. See Figure B2-3. Once the table is found the virtual to physical address translation can complete as described in section 2.1.2.

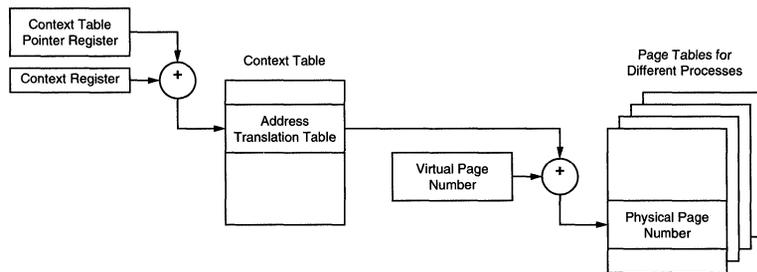


Figure B2-3. Schematic of Address Translation

2.1.4 Memory Protection

Memory protection can occur at two levels: the process level and the page level.

Memory protection at the process level is supported by the context register and the context table pointer register. Since each process can only go through its own address translation table when doing a virtual to physical address translation (as shown in Figure B2-3) one process can be prevented from accessing another process's instruction and data.

Since memory is segmented into pages, it is possible to associate with each page a protection field which can give permissions to the running program. These per-

missions can include whether the page can be read, written, executed, etc. by the program. For each page the permissions allowed are stored in the address translation table in the entry corresponding to that page.

2.1.5 How MMU's are Constructed

MMU's are constructed from a combination of hardware and software.

Software:

On the software side, the MMU is composed of the address translation table(s). There is one table for each process although this table can be either single level or multi-level as is defined in the SPARC Reference MMU (see Figure B2-3). These address translation tables reside in physical memory.

The MMU also is composed of the system software needed to search these tables to do virtual to physical address translations. When the running program generates a virtual address the MMU must conceptually translate that address by adding the context table pointer register to the context register to get a pointer which is added to the virtual page number to finally get the physical page number. This physical page number replaces the virtual page number to generate the physical address.

Finally, MMU software is required to move pages of instructions/data between mass storage and main memory as different pages of the running program are accessed.

Hardware:

The price of virtual addressing is that virtual addresses must be translated into physical addresses on the fly, at the time they are needed during execution. If each translation of virtual to physical addresses required a table lookup, as described above, the processor would run exceedingly slowly. Fortunately, instruction and data accesses exhibit a property known as "locality" - that is, they tend to occur not at random locations, but near each other on one or more recently-used pages.

To take advantage of this property, the MB86932 stores in one on-chip structure, the Translation Lookaside Buffer (TLB), 16 recently-used virtual page numbers, together with their corresponding physical page numbers. In another structure, the Instruction Translation Lookaside Buffer (ITLB), it stores the virtual and physical page numbers of the instruction page currently being accessed. Together these structures act as small cache of the most recently used virtual/physical address translations. Because of locality of address translations, most of the time the translation is done using the TLB. Only rarely does the CPU have to go to the address translation table to find a physical page number. Since the TLB transla-

tions occur in parallel with cache/memory access there is no time penalty as long as the translation pair is in the TLB.

Hardware is also provided to cause a trap whenever a requested virtual address is not found in the TLB. The trap software can be written to use the context register and context pointer register to find the head of the current process address translation table in physical memory. The virtual page number can be used as an offset in this table to find the physical page number. The trap software can then store this virtual page number/physical page number pair in the TLB for future use.

2.2 Programmer's Model

This section describes the user visible relations and their functions. Many of the registers depicted below are very similar to those provided in the MB86930, except for a few bits or fields that support the MMU in the MB86932.

2.2.1 Cache/Bus Interface Unit Control Register

The cache/bus interface unit control register is identical to that on the MB86930 except for the addition of the "TLB Enable Bit" (TE), bit 6 of the register. When cleared, the TLB is disabled, and translations from virtual to physical addresses do not occur. When set, translations are enabled, contingent on the state of the TLB. The TE bit is cleared on reset.

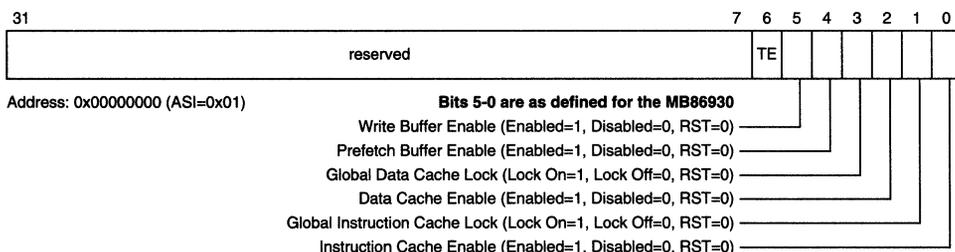


Figure B2-4. Cache/Bus Interface Unit Control Register

2.2.2 Context Table Pointer Register

This register holds the physical address of the base of the context table, which resides in main memory. When a software table walk is being done, the lower 8

bits of the context register can be added to the context table pointer register to create an offset into the context table in memory.

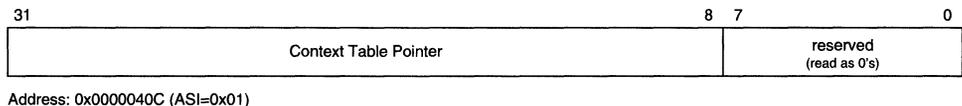


Figure B2-5. Context Table Pointer Register

2.2.3 Context Register

Bits 7 through 2 of the context register are implemented. This register has two functions: first, it provides protection between processes. During a TLB access, the context field is compared against the corresponding field in the TLB entry. If the two match—or if the global bit is set to show that context is irrelevant in this case—a virtual-to-physical address translation occurs. Second, it can be used during a software table walk, when the context field is used as a word offset into the context table in main memory. This is done by adding the context register to the context table pointer register, producing the physical address of the desired root pointer in the context table in main memory.

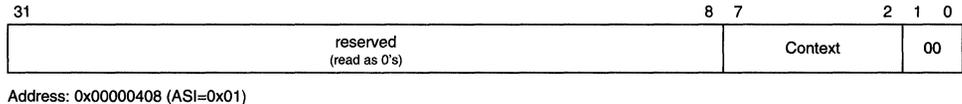


Figure B2-6. Context Register

2.2.4 TLB Exceptions

There are only two kinds of faults that can be caused by a TLB access: the *instruction_access_exception* and the *data_access_exception*., resulting respectively from an instruction address translation fault and a data address translation fault.

The MB86932 uses two of the existing traps defined in the SPARC (version 8) instruction set to support the TLB. The traps used are:

1. `Instruction_access_exception`
 - Version 8 ⇒ Priority=5; trap type=0x01
 - (a) A TLB miss occurred on an instruction access.
 - (b) A blocking error such as “protection violation” occurred on an instruction access.

- (c) A first reference to the instruction page was made, and the RT bit in the TLB Control Register was set.
- (d) An external mexc signal occurred during an external instruction fetch.
- (e) A parity error was detected on an external instruction fetch.

The cause of the `instruction_access_exception` is indicated by the Instruction Fault Status Register.

2. `Data_access_exception`

Version 8 ➔ Priority=13; trap type=0x09

- (a) A TLB miss occurred on a data access.
- (b) A blocking error such as “protection violation” occurred on a data access.
- (c) A first reference or first modification to this data page was made, and the RT or DMT bit in the TLB Control Register was set.
- (d) An external mexc signal occurred during an external read or write.
- (e) A parity error was detected on an external data read.

The cause of the `data_access_exception` is indicated by the Data Fault Status Register.

Since these two exceptions can be generated by several different causes, both TLB- and non-TLB related, two registers, described below, have been included to indicate the source of the exceptions.

2.2.5 Instruction Fault Status Register

There can be multiple causes for the `instruction_access_exception`; in particular, the TLB can cause this exception for a number of reasons. The Instruction Fault Status Register exists to indicate the exact reason for the fault, whether TLB-related or not. If the `instruction_access_exception` occurred, the address of the faulting instruction is in `r[17]`.

The instruction Fault Status Register is a read-only register. The bits in this register are set by hardware when an `instruction_access_exception` occurs and indicate the cause of the `instruction_access_exception`. This register is cleared when either

the Instruction Fault Status Register or the Data Fault Status Register is read by software.

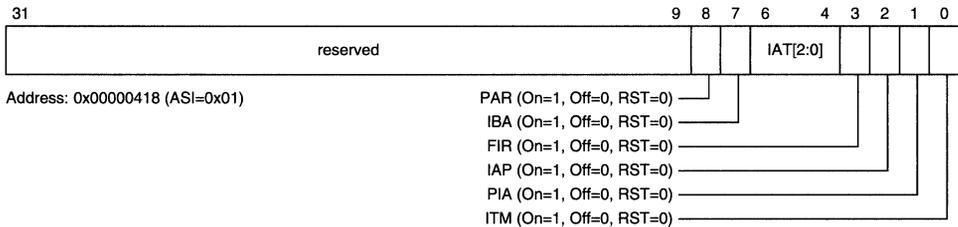


Figure B2-7. Instruction Fault Status Register

- Bits 31-9: Reserved
- Bit 8: Parity bit (PAR)—If IBA bit set, a set PAR indicates parity error; if PAR is cleared, “mexc” pin strobed, but no parity error detected.
- Bit 7: Instruction Bus Access exception (IBA)—Set when either external “mexc” or parity error occurs during external instruction fetch.
- Bits 6-4: Instruction Access Type (IAT[2:0])—This is the ACC field (from TLB) for the instruction causing the exception.
- Bit 3: First Instruction Reference (FIR)—Set when first reference is made to so-far “unreferenced” instruction page; this causes a trap only if the “RT” bit of “TLB Control Register” is set.
- Bit 2: Instruction Access Protection violation (IAP)—Set when an instruction lacks access permission sought.
- Bit 1: Privileged Instruction Access violation (PIA)—Set when user-mode instruction seeks access to supervisor-mode area.
- Bit 0: Instruction TLB Miss (ITM)—Set when address translation not in TLB or ITLB.

2.2.6 Data Fault Status Register

There can be multiple causes for the `data_access_exception`; in particular, the TLB can cause this exception for a number of reasons. The Data Fault Status Register exists to indicate the exact reason for the fault, whether TLB-related or not. If the `data_access_exception` occurred, the address of the datum causing the fault is held in the Data Fault Address Register.

The Data Fault Status Register is a read-only register. The bits in this register are set by hardware when a `data_access_exception` occurs and indicate the cause of the `data_access_exception`. This register is cleared when either the Instruction Fault Status Register or the Data Fault Status Register is read by software.

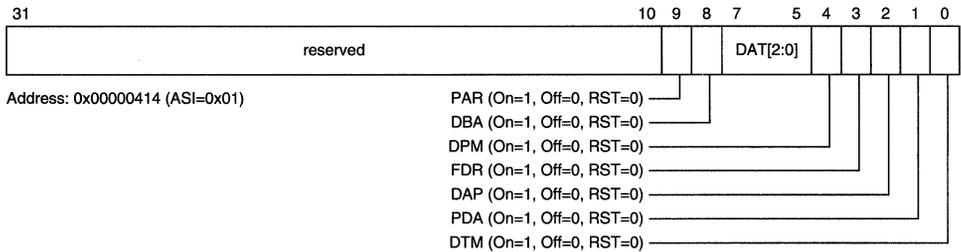


Figure B2-8. Data Fault Status Register

- Bits 31-9: Reserved
- Bit 9: Parity bit (PAR)—If IBA bit set, a set PAR indicates parity error; if PAR is cleared, “mexc” pin strobed, but no parity error detected.
- Bit 8: Data Bus Access exception (DBA)—Set when either external “mexc” or parity error occurs during external data read.
- Bits 7-5: Data Type (DAT[2:0])—This is the ACC field (from TLB) for the data causing the exception.
- Bit 4: Data Page Modification (DPM)—Set when first store is done to so-far unmodified data page; causes trap only if DMT bit in TLB Control Register is set.)
- Bit 3: First Data Reference (FDR)—Set when first reference is made to so-far unreferenced data page; causes trap only if RT bit in TLB Control Register is set.
- Bit 2: Data Access Protection violation (DAP)—Set when data access is attempted without permission for type of access sought.
- Bit 1: Privileged Data Access violation (PDA)—Set when data access sought to supervisor area when in user mode.
- Bit 0: Data TLB Miss (DTM)—Set when data address translation not in TLB.

2.2.7 TLB Control Register

One bit in this register is used to control whether a fault can occur on the first write to an unmodified page, and the other bit is used to control whether a fault can occur on the first reference to a previously “unreferenced” page. These bits

can be used to support different page-replacement schemes; they are cleared to 0 on reset.

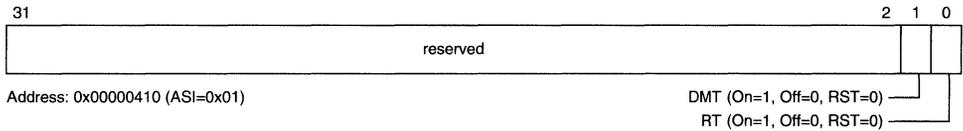


Figure B2-9. TLB Control Register

Bits 31-2: Reserved

Bit 1: Data Modify Trap (DMT)—Control bit, enables trapping on first modification of a datum in a so-far unmodified data page.

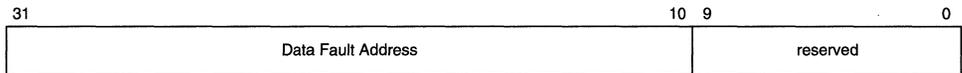
Bit 0: Reference Trap (RT)—control bit, enables trapping on first reference to so-far “unreferenced” page; cleared on reset.

2.2.8 TLB Data Fault Address Register

When a TLB fault occurs, it is necessary for the trap code to have access to the virtual address of the faulting instruction or data access. This allows the trap handler to know what address caused the fault. In the case of a TLB miss, this address is needed to perform a software table walk in main memory to find the correct translation. The instruction address is also necessary so that the access can be retried once the reason for the fault has been corrected.

In the case of a TLB fault during an instruction access, the virtual address of the faulting instruction is held in r[17] of the trap handler window. Thus, no special register is needed for this address. For data addresses the situation is different, since the effective data address is not saved during a trap; the TLB Data Fault Address Register is used instead to hold this address value. When a TLB fault is recognized during the memory stage of the pipeline, the address on the Data Address Bus is latched and held. This register can be read by the trap software. The TLB Data Fault Address Register contains the 22 most significant bits (MSB's) of the faulting data address, which is sufficient information for the table walk. The format is shown in Figure B2-10.

The TLB Data Fault Address Register is a read-only register. When a TLB data_access_exception occurs the virtual data address is captured and held. This register is cleared when read by software.



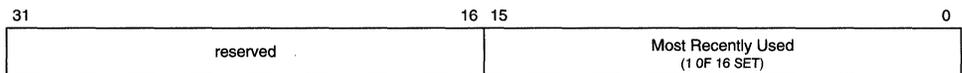
Address: 0x00000420 (ASI=0x01)

Figure B2-10. TLB Data Fault Address Register

2.2.9 Most Recently Used Register

Bits 0 through 15 of the Most Recently Used Register correspond to the 16 entries of the TLB. The register is updated every time a TLB match occurs: the bit corresponding to the matched entry is set, and all others are cleared. On a TLB miss, this register can be read, and supports replacement algorithms whose policy is to leave the most recently used address in the TLB.

The Most Recently Used Register is a read-only register. This register is cleared when read by software.



Address: 0x0000041C (ASI=0x01)

Figure B2-11. Most Recently Used Register

2.2.10 The TLB Entry

The TLB has 16 fully associative entries, each of them consisting of an entry in the CAM (content-addressible memory) array and the corresponding RAM array.

2.2.11 The TLB CAM Entry

The CAM-array entry is shown in the diagram below. Each CAM entry consists of a 20-bit virtual page number (VPN) that contains three index fields, a 2-bit fragment index, and a 6-bit context number (process identifier) that are compared against the virtual page address from the Integer Unit (IU) and the content of the context register. In addition, a global bit (G), two level bits (1:0), and a fragment enable bit (FE) are included. The CAM provides simultaneous comparison of all 16 TLB entries against the current virtual page address and context number. If a CAM entry matches the virtual page address, the corresponding RAM entry in the TLB provides a physical page number (PPN) to generate a physical address.

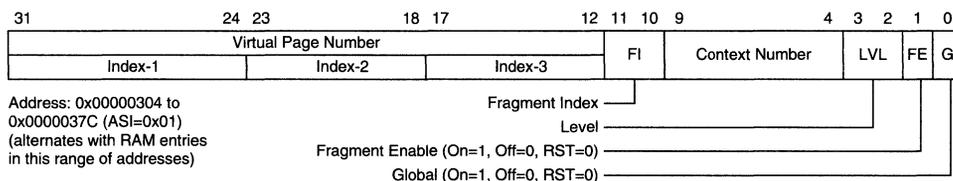


Figure B2-12. CAM Entry Format

Bits 31-12: Virtual Page Number (VPN)—Some page table subfields may be masked by Index-3, -2, and -1. Index-1, -2 and -3 support different page sizes.

During translation, the context field in the CAM is compared against the value in the context register. Only when these two values match can a RAM entry be selected. The only exception is when the corresponding Global bit is set, indicating that context is irrelevant. The Global bit is, in effect, an enabling switch for the Context field; if set, it masks out that field.

The two Level bits determine what page sizes are part of the MMU architecture. They work by acting as (encoded) masks, excluding from the comparison process one or more of the index subfields, as detailed in the following table:

Table B2-1: Level-bits Decoding Table

LVL[1:0]	Address Mapping	M3M2M1	TLB Field masked
11	4-kbyte	0 0 0	None
10	256-kbyte	1 0 0	Index 3
01	16-Mbyte	1 1 0	Indexes 2 and 3
00	4-Gbyte	1 1 1	Indexes 1, 2 and 3

As summarized in the table, the mask bits M3, M2 and M1 are used to exclude from comparison index 3 (bits 17-12), index 2 (bits 23-18), and index 1 (bits 31-24), respectively. If M3 is set, index 3 is masked out, and a RAM entry is selected based on the match of indexes 1 and 2, and the context number. The 6-bit index 3 combined with the 12-bit page offset can provide an index to a 256-Kbyte linear addressing region. If both M3 and M2 are set, the RAM entry is selected based on the match of the index 1 and the context number. The 12-bit index (bits 23-12) combined with the 12-bit page offset can provide an index to a 16-Mbyte addressing region. If all three masks are set, the RAM is selected based on the context number match alone, and the RAM entry provides an index to a 4-Gbyte addressing region.

2.2.12 The TLB RAM Entry

The RAM-array entry is shown in the diagram below. It consists of a 20-bit (maximum) physical page number (PPN), a 3-bit access-level protection, a cacheable bit, a modify bit, and a valid bit. The mask bits (M1, M2, and M3) are a decoded version of the LVL field in the corresponding CAM entry.

The mask bits allow the TLB to generate a correct index into a page for different page sizes. The index fields that are excluded from the CAM comparison process are used as part of this index into the page—used as part of the offset into the selected page instead of part of the PPN. If all mask bits are clear, the 20-bit PPN drives the upper 20 bits of the physical address, and the 12-bit offset drives the lower 12 bits. If M3 is set, the lower 6-bits of the PPN are replaced by the bits 17-12 of the virtual address. Therefore, the physical address contains a 18-bit untranslated address (page offset) and a 14-bit page number. If both M2 and M3 are set, the lower 12 bits of the PPN are replaced by bits 23-12 of the virtual address. The physical address then contains a 24-bit untranslated address (page offset), and a 8-bit page number. If all mask bits are set, the virtual address outputs to the physical address.

Associated with each virtual page are coded values that indicate what kind of accesses (read, write, execute, or none) may be made to this page by this program. When loaded into the TLB these values are stored in the ACC field [5:2], and are compatible with the specification given in the *SPARC Reference MMU Architecture*. Note that entries in the TLB make no explicit reference to ASI spaces; this information is implicit in the access bits of the TLB.

Table B2-2: Access Protection available through PTE[4:2]

ACC Field Value	Accesses Allowed	
	User Access (ASI=0x8 or 0xA)	Supervisor Access (ASI=0x9 or 0xB)
0	Read Only	Read Only
1	Read/Write	Read/Write
2	Read/Execute	Read/Execute
3	Read/Write/Execute	Read/Write/Execute
4	Execute Only	Execute Only
5	Read Only	Read/Write
6	No Access	Read/Execute
7	No Access	Read/Write/Execute

Note: ASI=Alternate Space Identifier; an 8-bit value that indicates whether a load/store instruction transfers data to/from external units, or registers on the chip.

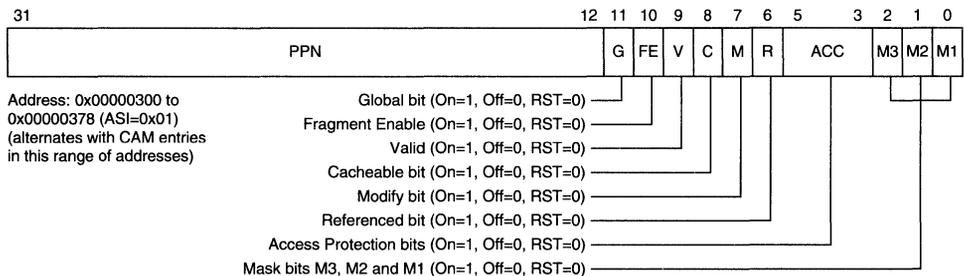


Figure B2-13. RAM Entry Format

- Bits 31-12: (Maximum) Physical Page Number (PPN)—Some page table subfields may be masked by M3, M2, and M1
- Bit 11: Global bit (G)—This bit and the fragment enable bit (FE) are duplicates of the same bits in the corresponding RAM entry.
- Bit 10: Fragment Index bit (FE)—If set, this bit asserts that the Fragment Index bits are to be included in the comparison. If so, the result is to establish access protection at the sub-page 1K level, the “fragment” level. When the FE bit is set, the Fragment Index bits are compared against virtual address bits 10 and 11 (which do not themselves go through translation). A RAM entry is then selected only when there is a VPN match *and* a FI match in the CAM entry. Since each TLB entry can set its own access level or protection, protection at the 1-Kbyte level is thus available. When the FE bit is not set, the FI bits are excluded from the comparison, and access protections apply only to the full 4-Kbyte (or larger) whole pages.
- Bit 9: Valid bit (V)—This bit reports the current validity of the TLB entry. The V bit of each entry should be cleared by software to invalidate those entries before the TLB is enabled.
- Bit 8: Cacheable bit (C)—This bit indicates whether the memory addressed by the TLB is cacheable or not.
- Bit 7: Modify bit (M)—This bit in the TLB is set when the memory page is modified by a write operation.
- Bit 6: Referenced bit (R)—This bit is set by the TLB when the page in question is accessed. This bit can be used in TLB replacement algorithms.
- Bits 5-3: Access Protection bits (ACC)—The access-level protection for the address region mapped by the RAM entry. Access-level protection is checked during TLB access. If a TLB match occurs, but access-level protection is violated, the TLB will generate a trap.
- Bits 2-0: Mask bits (M3, M2 and M1)—A decoded version of the LVL field in the corresponding CAM entry.

2.2.13 ITLB Description

Fully static implementation of a one-entry Instruction Translation Lookaside Buffer (ITLB) allows immediate access to the physical address of the last page entry stored there. The registers associated with the ITLB are in locations 0x00000400 ("RAM" entry) and 0x00000404 ("CAM" entry). These two registers are the same as the TLB RAM and CAM entries. The instruction cache hit/miss and access permissions are determined by the PTE in the ITLB, so there is no performance penalty in using the ITLB. If an access-level violation is detected, the ITLB generates an `instruction_access_exception` trap.

2.2.14 TLB Lookup

If an instruction address translation is not found in the ITLB, an ITLB hold is asserted, and the virtual instruction address is looked up in the TLB on the next (the second) cycle, preempting any data address translation. On a match in the TLB, the TLB entry is output to the ITLB. On the third cycle, the translation is retried using the ITLB, with guaranteed success. If the translation is not found in the TLB, an `instruction_access_exception` trap is asserted, and a software routine to access the address translation table in main memory can be executed. This is known as a "software table walk". Due to the locality of instructions, and the availability of the TLB in case of ITLB miss, the performance price of instruction address translation is minimal.

For data address translation, the virtual data address is used directly to compare against each entry in the TLB. If a TLB match occurs, the TLB outputs the physical address, the cacheable bit, and the access protection bits. If the translation is not found in the TLB, or if an access-level violation is detected, a `data_access_exception` trap is asserted. If the trap occurred because of a TLB miss, a software table walk can be initiated.

2.3 Internal Architecture

2.3.1 Details of TLB Logic

Because of the normally sequential nature of instruction addresses, it is likely that the next required instruction is on the current page. Accordingly, the virtual instruction address is sent directly to the ITLB for translation; only if the desired address is not found there is it sent to the TLB. If it is found in the TLB, it is loaded into the ITLB, and instruction address translation proceeds. If the physical address is not found in the TLB either, an "instruction_access_exception" trap is asserted, and a software table walk can be performed. The physical address found

in the Page Tables is entered into the ITLB and TLB for use in later instruction address translations.

Because data is more widely scattered, data address translations go directly to the TLB. If the desired physical address is not found in the TLB, a “data_access_ exception” trap is asserted, a table walk is performed, and the physical address when found in the Page Tables is entered into the TLB for later data address translations.

The net result for overall system performance is that few instruction or data references in a normally structured program need be translated by accessing the Page Tables in memory; the great majority of the physical addresses needed are found on-chip in the TLB/ITLB, and two physical addresses—one data address and one instruction address—can be acquired simultaneously. The details of the process of translating a virtual into a physical address are illustrated in the diagram and flowchart in Figures B2-14 and B2-15:

2.3.2 Address Translation: Logical and Physical Steps

The diagram below illustrates the registers and fields involved in TLB-based translation of a virtual address into a physical address. The flowchart supplements it by correlating the physical steps taken with their meaning from the user's point of view.

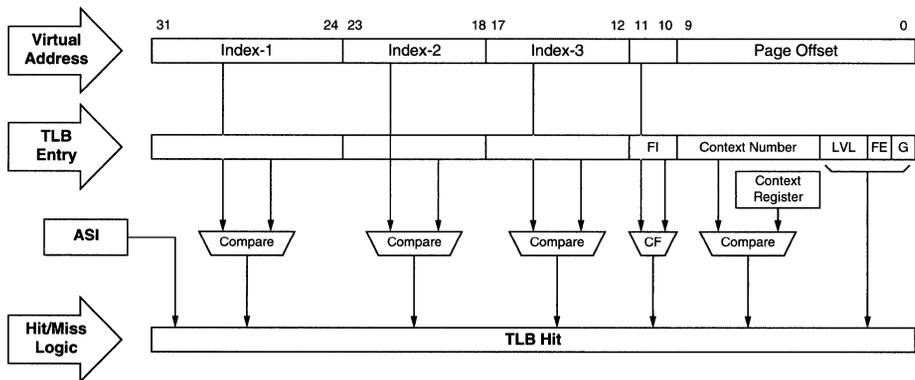


Figure B2-14. Address Translation by TLB: Fields and Registers

In this flowchart, the logic of each translation step is given in the left-hand box, and its physical realization in the corresponding right-hand one.

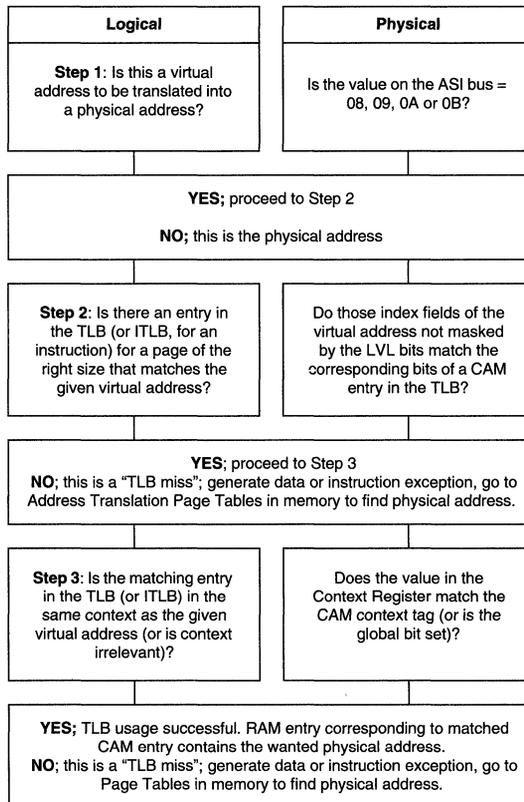


Figure B2-15. Flowchart of TLB Address Translation

2.3.3 Basic TLB Exception Timings

The diagram below indicates at what stage of the pipe the various TLB exceptions occur, and when the processor recognizes these exceptions. Instruction-access TLB faults occur during the fetch stage, while data-access TLB faults occur during the memory stage. All of the exceptions are recognized at the end of the memory stage.

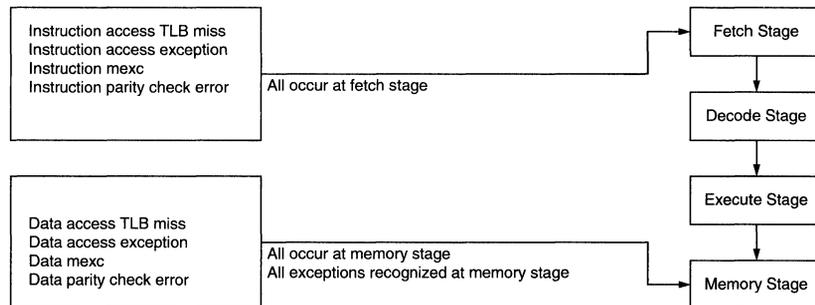


Figure B2-16. TLB Exception Timings

2.3.4 TLB Timing Considerations

The TLB does the instruction/data address translation in parallel with accessing the instruction/data caches (see Figure B3-2 in Chapter 3 "MB86932 Caches"). Thus, there is no additional cycle penalty when executing with the TLB enabled if the virtual/physical address translation is in the TLB. There are two exceptions to this rule:

1. If the virtual/physical address pair being accessed is in the TLB but not in the ITLB, the translation will require an additional two cycles. The first of these is used to access the TLB for the translation pair and load it into the ITLB; the second is used to retry the translation with the ITLB.
2. If either cache is disabled and the TLB is enabled, the TLB translation requires an extra cycle be inserted before the address can be driven on the Address pins.

Figure B2-17 shows the timing for TLB virtual-to-physical address translations. The "ITLB Status" indicates whether an ITLB hit or miss occurs for the address of the instruction in the fetch stage. The "TLB Status" normally indicates whether a TLB hit or miss occurs for the data address of the instruction in the memory stage.

Four situations are indicated in Figure B2-17. Cycle 0 is an example of both an ITLB hit for the instruction address of INST4, and a TLB hit for the data address of INST1.

Cycles 1 through 3 show what occurs when INST5 misses in the ITLB in cycle 1. The INST5 instruction translation is retried using the TLB in cycle 2. If a hit occurs, the ITLB is updated with the value from the TLB. Finally, the instruction address of INST5 goes through translation using the ITLB in cycle 3.

Cycles 4 and 5 indicate what happens when the instruction address translation for INST6 is in neither the ITLB nor the TLB. After accessing the ITLB and missing, the TLB is accessed with the same instruction address. When a miss is detected here, this causes an instruction memory exception to occur. Note that this does not cause a trap until this instruction reaches its memory stage in cycle 8.

Cycle 8 shows a data address TLB miss. (The data address from the instruction in the memory stage always goes to the TLB except in the cases when the instruction address preempts it after an ITLB miss.) This data-address TLB miss causes a `data_memory_exception` which that is recognized in the same cycle in which it occurs.

For the last two cases, the exceptions cause the processor to vector to the trap routine (INST20) in cycle 9. The instructions in the decode, execute, and memory steps are “squashed” at that time. The faulting instruction (INST6 in both cases) does not write back a result to the register file. Instead, the PC (virtual address) of the faulting instruction is written to the register file. In addition, if a data address translation caused the fault, the value of the faulting virtual data address is written into the data fault address register.

2.3.5 TLB Emulation Support Logic

When the MB86932 chip is executing from the In-Circuit Emulation (ICE) port, all accesses, including those to supervisor instructions and data, should be untranslated. This is required because the ICE logic has a fixed memory map, and will not be able to handle translated addresses. The ICE code will have the responsibility of doing the table walk before accessing any location. It should access everything using physical addresses.

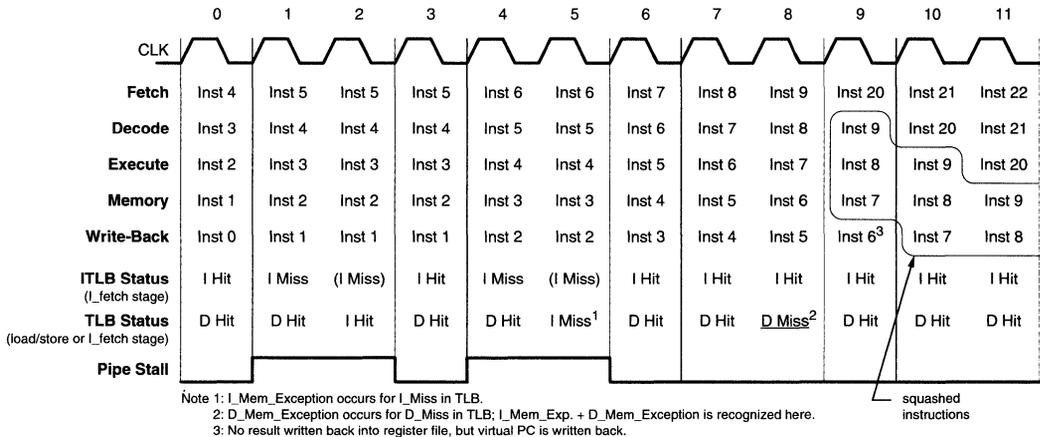


Figure B2-17. Sequence of Events for ITLB and TLB Misses

2.4 Programming Considerations

2.4.1 MMU Architecture Example: the SPARC Reference MMU

One MMU architecture that the MB86932 can support is the SPARC Reference MMU. This architecture can be implemented as follows.

The information the MMU required to perform virtual-to-physical address translation is put in a hierarchy of physically-addressed structures, the Page Tables, that reside in main memory. In the SPARC Reference MMU architecture, with three Page Tables, Page Tables 1 and 2 would contain two kinds of entries:

- *Page Table Pointers*, which contain the physical address of the logically next-lower table, and thereby link the tables together as a hierarchy (Note that PTPs are never found in the TLB.)
- *Page Table Entries*, which contain the physical address of a page of the size associated with the table (along with other page-specific information). Since the TLB caches PTEs, the information *content* of PTEs in main memory should be compatible with that of PTEs in the TLB, although the exact format may differ.

Page Table 3, can contain only Page Table Entries (PTEs), since there is no next-lower table for it to point to. SPARC reference compatible formats of the PTP and PTE are:

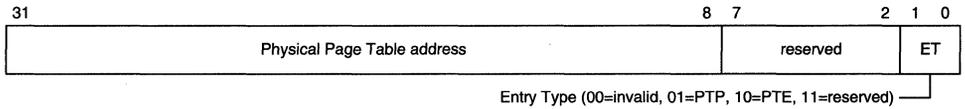


Figure B2-18. Page Table Pointer (PTP)

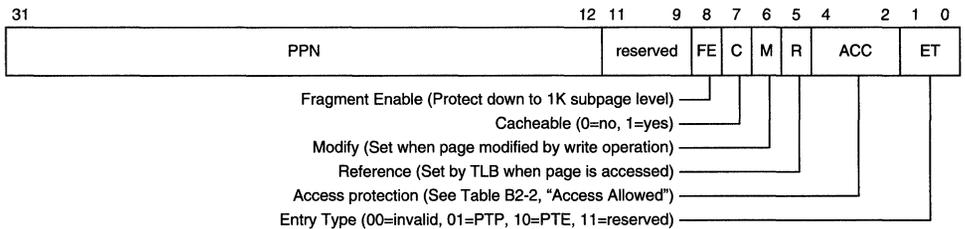


Figure B2-19. MB86932 Page Table Entry (PTE)

Note that the FE field is not part of the SPARC Reference MMU Architecture, but is introduced by the MB86932. (The reservation of bits [11:9] is similarly an MB86932 feature.)

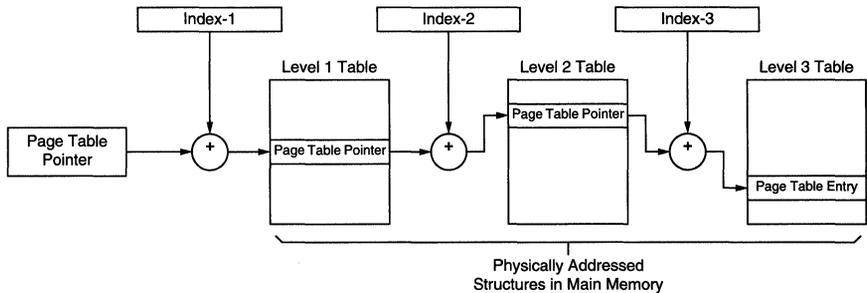


Figure B2-20. Pointer chaining from Page Table to Page Table

The table walk, or search from table to table for a matching virtual address, follows a simple logic: after the root pointer locates the Level-1 table in memory, and index-1 of the virtual address picks out a particular entry in that table, each suc-

ceeding table (if necessary) is located by the PTP just found, and the entry in that new table is picked by the next index field of the virtual address.

The reason for dividing the page pointers among three tables is to support sparse addressing efficiently; the root and the PTEs in the three tables point respectively to pages of 4 gigabytes, 16 megabytes, 256 kilobytes, and 4 kilobytes, so memory can be used in block sizes appropriate to an application's routines and data structures.

2.4.2 Virtual Address format

The format of a virtual address as generated by the Integer Unit (IU) and passed for translation to the TLB is as follows: the Page Offset field for the maximum (four page sizes) configuration is nominally 12 bits wide (bits 11-0) and specifies a particular byte within a 4K-byte page; the three index fields, collectively known as the Virtual Page Number (VPN) field, enable the TLB to identify the correct page. Each index field is an offset into the correspondingly-numbered page table.

If the page whose physical address is sought is larger than 4K-bytes, a 12-bit offset field is insufficient to identify a specific byte within it. But the PTE of a page size larger than 4K will not be in table 3, and will not require that all three Page Tables be walked during the translation process, so one or more of the index fields becomes available for use in forming a bigger page offset field, exactly as required. In effect, then, the offset field is always as big as needed for the current address translation process.

For example, if the information sought is in a 256K-byte page, the index-3 field, needed only when the PTE is in Table 3, is effectively made part of the Page Offset field for that translation, giving an Offset field of 18 bits; this supports the identification of an individual byte in a 256K-byte page. Similarly, a 16M-byte page will not require index fields 2 or 3, yielding the effective 24-bit Offset field that is needed for addressing a 4M-byte address space.

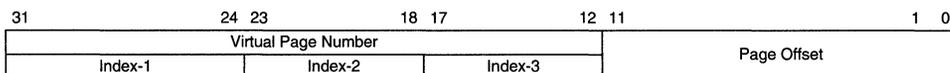


Figure B2-21. Virtual Address Format

2.4.3 Physical Address format

The MB86932 physical address format differs from that specified in the *SPARC Reference MMU Architecture* in being 32 bits wide rather than 36. It is otherwise as specified in that document: Since all pages begin on 4K-byte boundaries, the low-order 12 bits of the physical address are the same as those of the virtual address, and do not require translation.

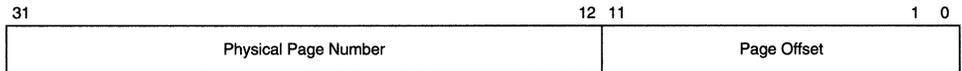


Figure B2-22. Physical Address format

2.5 Conformity to SPARC Reference MMU Architecture

The MMU architecture of the MB86932 is as specified in *The SPARC Reference MMU Architecture* (Sun Microsystems, Revision 1.4, 23 Jan 1989), with the following exceptions:

- The physical address format is 32 bits wide rather than the 36 bits specified in the Reference Architecture.
- As a consequence of the difference in physical address format, the Physical Page Number (PPN) portion of the Page Table Entry (PTE) is 20 bits wide rather than the 24 bits specified in the Reference Architecture.
- Bit 8 of the PTE, the Fragment Enable (FE) bit, has been reserved in the 932 to support protection down to 1K sub-page boundaries.
- A few of the registers and bits specified in the Reference Architecture are not implemented, or not implemented in full. Specifically: only bits 7-2 of the Context Register are implemented.
- The *instruction_address_MMU_miss* and the *data_address_MMU_miss* exceptions defined in the Reference are not implemented; on the occurrence of one of these misses, the TLB will instead vector to the *instruction_access_exception* trap routine, or the *data_access_exception* trap routine, respectively.



MB86932 Caches

3.1 Overview of MB86932 Caches

The MB86932 offers enhanced support for cacheing: its instruction cache is 8K-bytes in size, and has 8-word lines. (The corresponding values for the MB86930 are 2K-bytes and 4-word lines.) The data cache of the MB86932 remains the same as the MB86930's at 2K-bytes and 4-word lines. The increased instruction cache size is reflected in a new format for the Instruction Cache Tag, which has four new "valid" bits to control the four new words per cache line (the other four valid bits remain in the same positions they occupy in the I_Cache Tag in the MB86930, making for backward compatibility).

Both caches are "physical" caches; that is, the cache arrays are accessed with physical, not virtual, addresses. The addresses stored in the tag arrays are also physical, not virtual, addresses. Since the caches are accessed with physical addresses, the reading and writing of the caches is expedited by the MB86932's restriction of the minimum page size to 4K-bytes. This allows the lower 12 bits of the physical address to be identical to the lower 12 bits of the virtual address, which in turn means that, given 2-way set associativity, the cache can be up to 8K bytes without requiring any address translation when being accessed. The MB86932 uses this full 8K-byte space in its instruction cache, while in the data cache only 2K-bytes (of the possible 8K-bytes) are implemented.

3.2 Programmer's Model

The cache control/status registers of the MB86932 form a superset of those in the MB86930. The registers common to the two chips are the:

0x00000000	ASI=0x1	Cache/BIU Control Register** (TLB enable bit added)
0x00000004	ASI=0x1	Lock Control Register
0x00000008	ASI=0x1	Lock Control Save Register
0x0000000C	ASI=0x1	Cache Status Register
0x00000010	ASI=0x1	Restore Lock Control Register

To this set (all in the ASI=0x01 space) the MB86932 adds two Instruction_Cache_Invalidate Registers, one for each bank of the instruction cache, and two Data_Cache_Invalidate Registers, one for each bank of the data cache. All four are write-only; their format is shown below.

Bank 1 of the instruction cache is controlled by the register at address 0x00001000, while bank 2 is controlled by the register at address 0x80001000 both in ASI space 0x0C. Bank 1 of the data cache is controlled by the register at address 0x00001000, while bank 2 is controlled by the register at address 0x80001000, both in ASI space 0x0E.

Invalidating the cache, and clearing lock and lru bits, is an easy way to remove old code/data from the caches when a new page is brought into physical memory, or after a DMA has been made to cacheable locations in main memory. Clearing only the lock and lru bits is an easy way to allow locked code to be replaced after use. Note that the invalidate bits are written during the M stage of the instruction; thus, their effect is not felt until the fourth instruction after the instruction that writes to these registers.

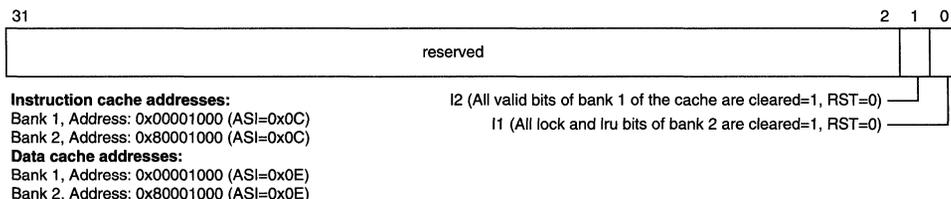


Figure B3-1. Cache Invalidate Register Format

3.2.1 Operation of the Instruction Cache

At reset the cache is turned off, and the valid bits, lock bits, and LRU bits are set to 0. Initialization of the cache to particular values can be done by doing stores to an alternate address space 0x0C. When the cache is off, all requests are sent to the external memory. After the cache is initialized, the user writes a 1 to the cache-on bit to turn on the cache.

3.2.2 Operation of the Data Cache

At reset, the cache is turned off, and the valid bits, lock bits, and LRU bits are set to 0. Initialization of the cache to particular values can be done by doing writes to alternate address space 0x0E. When the cache is off, all requests are sent to the external memory. After the cache is initialized, the user writes a 1 to the cache-on bit to enable the caches.

Accesses to the ASI's corresponding to user and supervisor data space are cached. No loads or stores from any other ASI are cached.

3.3 Internal Architecture of MB86932 Caches

Figure 3-Cache-1, below, shows how the TLB works with the caches (in the example shown, the Instruction Cache):

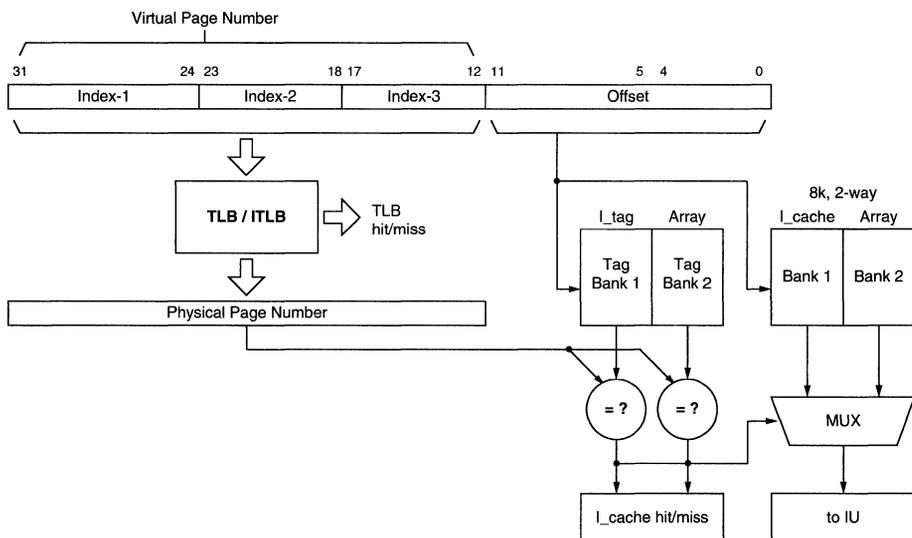
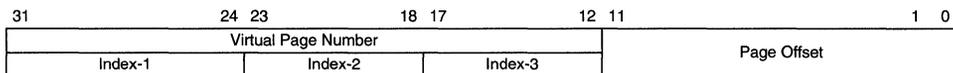


Figure B3-2. TLB/Cache Interaction

3.3.1 Instruction Cache

The instruction cache is an 8K-byte, 2-way associative, sectored cache, with 8-word lines. The basic operation of the cache is as follows: the IU sends the virtual address to the TLB, I_cache, and I_cache tags. Since the lower 12 bits of the virtual address are not translated, they are available immediately at the I_cache and tag array. Thus, the tag array can be accessed and the I_cache address can be decoded simultaneously with the TLB translation of the virtual page number to the physical page number. Once this is completed, the tag read from the tag array can be compared to bits 31-12 of the translated physical address to determine hit or miss.

The virtual instruction address format is shown below. The virtual page number has three index fields that are conditionally translated by the TLB, based on the mapped memory region size. The address coming out of the TLB is the physical address, and goes to the I_cache and tags. Bits 31-12 go to the tag array for comparison. Bits 11-5, which do not go through translation, select two tags (one for each bank) out of the 256-entry tag array, and also choose two lines (one for each bank) out of the 8K I_cache. Bits 4-2 select a word out of the 8-word line. In each of the diagrams below, bits 0-11 are the untranslated part of the address.



(from IU to ITLB)

Figure B3-3. Virtual Instruction Address



(from ITLB to Instruction Cache)

Figure B3-4. Physical Instruction Address



Figure B3-5. Address to I_cache and Tag Array

The instruction cache tag format is shown below. Twenty bits make up the address tag. Four bits, 9-6, are Valid bits for four of the words of the 8-word line. These bits are in the same location as the valid bits of the MB89630 I_cache tag array. Four additional Valid bits have been added for the other four words of the 8-word line. Bit 5 is used to indicate whether the line can be accessed by supervi-

sor only. Bit 1 is the least-recently used bit, which is used when doing a line replacement in the I_cache. Note that because of the increase in cache size and line size, the tag format of the MB86932 differs from that of the MB86930.

How the valid bits in a tag correspond to the words in the corresponding line is shown below:

Word Address [4:2]	000	001	010	011	100	101	110	111
Valid Bit Location	6	7	8	9	2	3	4	10

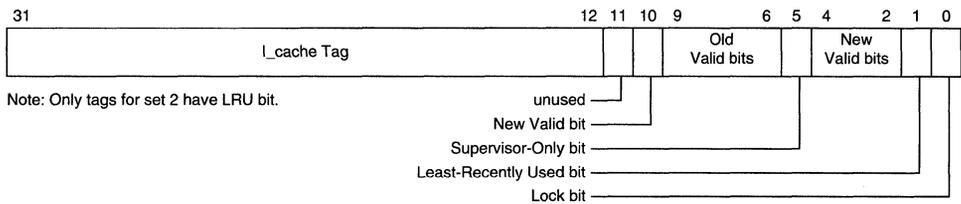


Figure B3-6. I_cache Tag Format

Note that any access that competes with a currently locked entry in the cache is treated as non-cacheable. In addition to the lock bits in the tag array, there is a global cache lock bit for each of the caches. Whenever these global lock bits are set, all accesses that do not result in a hit in the cache are treated as non-cacheable.

Writes to the instruction address space are not supported. The tag and instruction memory can be updated by doing writes to alternate address spaces 0x0C and 0x0D.

3.3.2 Read Hit

On an instruction fetch, the tag and the instruction are accessed in parallel, using the untranslated lower 12 bits of the address. If the translated bits of the address match one of the accessed tags, and the U/S fields match, and the “valid” bit corresponding to the word being accessed is set, then the required instruction is in the cache. The instruction is returned to the IU, and the LRU bit is updated. The lock bit may be updated, based on the value of the Instruction lock bit in the “lock control register.”

3.3.3 Miss Processing

If the address field in the tag does not match the translated address bits (31-12) coming from the TLB, or the U/S bit does not correspond to the ASI indicated by the IU, or the corresponding “valid” bit is not set, the result is a cache miss. In this

case, the “hold” signal to the IU, and the “miss” signal, are asserted. This freezes the IU pipeline. The request is sent to external memory via the BIU.

If the address field in the tag matches the translated address bits (31-12), and the U/S bit corresponds to the ASI indicated by the IU, and at least one of the valid bits is set (but the valid bit for the requested word is not set), it implies that an entry has already been allocated for this word. There is no need to select an entry to be replaced.

If the miss is due to the address field in the tag not matching the translated address bits (31-12), or the U/S bit does not correspond to the ASI indicated by the IU, or none of the valid bits is set, then an entry needs to be selected for replacement (or allocation). The LRU bit for this entry is checked, and the least-recently used entry is chosen to be replaced (or allocated).

The entry that is chosen for replacement will also depend on the “lock” bits. Consider two sets, A and B. If the lock bit for a given entry in A is set, and the corresponding bit of B is clear, then the entry in B will be replaced regardless of the value of the LRU bit. The LRU bit will be updated to show the entry in A to be the least-recently used. If the lock bit for both entries, or the lock bit for the whole cache, is set, then the access will be treated as a non-cacheable access.

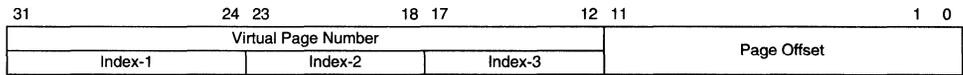
In the case of an instruction fetch, when the required instruction is accessed from main memory, it is returned to the IU and stored in the cache. The “hold” signal freezing the IU is deasserted. If a line was replaced or allocated because of the cache miss, the valid bit for the accessed word is set, and the other valid bits are reset. If the word being accessed is part of an already allocated line, then only the “valid” bit for the accessed word is set. All other bits remain unchanged. The lock bit may also be updated based on the value of the Instruction lock bit in the “lock control register.”

3.3.4 Data Cache

The data cache is a 2K-byte, 2-way associative, sectored cache, with 4-word lines. The basic operation of the cache is as follows: the IU sends the virtual address to the TLB, D_cache, and D_cache tags. Since the lower 12 bits of the virtual address are not translated, they are available immediately at the D_cache and tag array. Thus, the tag array can be accessed and the D_cache address can be decoded simultaneously with the TLB translation of the virtual page number to the physical page number. Once this is completed, the tag read from the tag array can be compared to bits 31-10 of the translated physical address to determine hit or miss.

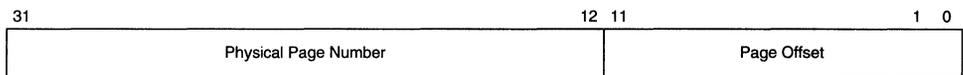
The virtual data address format is shown below. The virtual page number has three index fields that are conditionally translated by the TLB, based on the mapped memory region size. The address coming out of the TLB is the physical address, and goes to the D_cache and tags. Bits 31-10 go to the tag array for com-

parison. Bits 9-4, which do not go through translation, select two tags (one for each bank) out of the 128-entry tag array, and also choose two lines (one for each bank) out of the 2K D_cache. Bits 3-2 select a word out of the 4-word line. In each of the diagrams below, bits 11-0 are the untranslated part of the address.



(from IU to TLB)

Figure B3-7. Virtual Data Address



(from TLB to Data Cache)

Figure B3-8. Physical Data Address from TLB

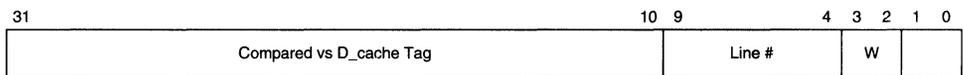
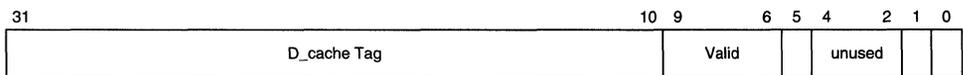


Figure B3-9. Address to D_cache and Tag Array

The data cache tag format is shown below. Twenty-two bits make up the address tag. Four bits, 9-6, are valid bits for each word of a D_cache line. Bit 5 is used to indicate whether the line can be accessed by supervisor only. Bit 1 is the least-recently used bit, which is used when doing a line replacement in the D_cache. Finally, bit 0 is used to lock the entry into the cache. Note that this format is identical to that of the MB86930.



Note: Only tags for set 2 have LRU bit.



Figure B3-10. D_cache Tag Format

The data cache follows a write-through update policy. On a write hit, the data is written to both the cache and main memory. If there is a write miss, the data is written only to the external memory. A different write policy is followed if the write is to a locked location.

The lock bit in the data cache has the effect of locking the current data in the cache. Any access that does not result in a hit in the cache, and maps to a location that is currently locked, is treated as non-cacheable. Any writes to locked data cache entries are not written to main memory. Only the data in the cache is updated.

3.3.5 Read Hit

On a load, the tag and the data are accessed in parallel, using the untranslated lower 12 bits of the address. If the translated portion of the address field coming from the TLB matches the tag, and the U/S bit corresponds to the ASI indicated by the IU, and the "valid" bit corresponding to the word being accessed is set, then the required data is in the cache. Since a hit is detected, the data is returned to the IU, and the "hold" signal to the IU is not asserted. The LRU bit is updated. The lock bit may be updated, based on the value of the Data lock bit in the "lock control register."

3.3.6 Write Hit

On a store(ST, STB, STH), if a hit is detected, the IU hold signal is not asserted. The LRU bit is updated. The lock bit may be updated, depending on the value of the Data lock bit in the "lock control register." If the lock bit for this entry is not set, or the Data lock bit in the "lock control register" does not indicate that the entry is to be locked, then the transaction is also sent to the BIU to be completed in external memory.

3.3.7 Miss Processing

If the address field in the tag does not match the translated address bits (31-10) coming from the TLB, or the U/S bit does not correspond to the ASI indicated by the IU, or the corresponding "valid" bit is not set, the result is a cache miss.

In the case of a write miss, the cache is left unchanged, and the request is sent to the BIU to be completed in external memory.

A read miss is processed in exactly the same way as a miss for an instruction fetch, except that the lock bit may be updated depending on the value of the Data lock bit in the "lock control register."

3.3.8 Atomic Load and Store

All atomic load and store transactions are treated as non-cacheable transactions.

CHAPTER B4

.....

MB86932 Bus Interface Unit

4.1 Overview of Bus Interface Unit

The BIU on the MB86932 includes all the features of the MB86930, and in addition offers the following:

- A four-word burst mode for instruction fetches and data loads,
- Byte-based parity generation/checking for the external data bus,
- A modified Wait State Specifier Register that supports burst mode and parity generation/checking on specified address ranges,
- A ROM/PROM interface that allows the MB86932 to boot from either 8-bit wide or 16-bit wide ROM/PROM,
- A processor bus request feature that enables the MB86932 to request access to external address and data buses,
- Modified timing on the external address bus when the TLB is enabled while caches are off.

4.2 Burst Mode

4.2.1 Overview

The Bus Interface Unit (BIU) supports the fetching of instructions and data from external memory to the appropriate cache in 'bursts' of four words at a time. A burst mode transfer is initiated either by a cache miss or by a DMA request. For a

cache miss, burst mode is supported only for instruction fetches and data loads, not for stores. The IU is held until all four words are fetched. For DMA burst access, both data burst reads and data burst writes are supported. (Note, however, that the DMA does not support movement of data to/from cache.)

When burst mode is triggered by a cache miss, it replaces four words in the cache line where the miss occurred. Such a burst-mode transfer can take place only if (a) the enabling bit (see "Bus Control Register," below) is set, and (b) the external memory supports burst mode. In the case of an *i_cache* miss, only half the line is replaced, since *i_cache* lines are eight words long. In the case of a *d_cache* miss, the entire four-word line is replaced by a burst-mode fetch. The four-word sequence fetched in burst mode starts with the word that caused the miss, followed by three more words in a standard order.

4.2.2 Burst Mode Interface Pins

Two pins are dedicated to burst mode:

- BMREQ: Output pin to inform the memory system that the current bus transaction is a burst mode.
- BMACK: Input pin to inform the processor that the memory system can support burst mode.

Note: When a cache miss occurs, -BMREQ will be asserted only if the corresponding bit of the Bus Control Register (DBE for data, IBE for instructions) is set. However, for a DMA transaction, -BMREQ is asserted whenever a quad word transfer is requested, regardless of the status of the DBE bit.

4.2.3 Burst Mode Fetch Sequence

In burst-mode accesses, the BIU automatically uses the two least significant bits (LSBs) of the address of the requested word, ADR[3:2], to determine the sequence in which the other three words will be fetched. (The sequence is optimized for a 2-way interleaved memory.) The table below shows the four possible sequences of words, in terms of their address LSBs, depending on the LSBs of the word causing the miss. Note that the first word accessed in a burst is always the one requested by the IU and that during a burst access, bits ADR[3:2] do not change.

Table B4-1: Sequence of Words Fetched in Burst Mode

LSBs of Missed Word	SEQUENCE OF WORDS TRANSFERRED (in terms of their LSBs)			
	1st word	2nd word	3rd word	4th word
00	00	01	10	11
01	01	00	11	10
10	10	11	00	01
11	11	10	01	00

4.2.4 Bus Mode control bits

Two bits in the Bus Control Register are used to control burst mode for instruction fetches and data loads.

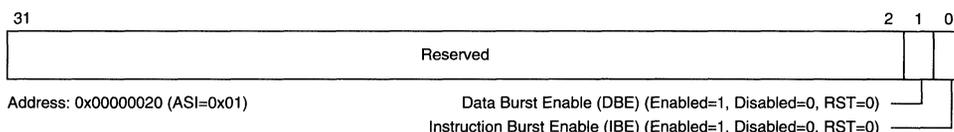


Figure B4-1. Bus Control Register

On reset, burst mode for both instruction and data misses is disabled. The user must explicitly enable one or both after reset. Bus operations already in progress are not affected by modification of the burst-enable bits.

4.2.5 PROM Address Space

Burst mode access from the PROM address space is not supported for 8- or 16-bit bus mode. If burst mode is enabled, and the address lies within the PROM address space for a non-32-bit bus mode transfer, the burst mode request output signal (-BMREQ) will still be asserted, but the burst acknowledge signal (-BMACK) should not be asserted by the external memory. If -BMACK is asserted under these conditions, the BIU operation is undefined.

4.2.6 Prefetch Buffer

The prefetch buffer is not used when burst-mode instruction fetches are enabled, and is automatically disabled if the IBE bit is set, regardless of the state of the Prefetch Buffer Enable bit in the Cache/BIU Control Register. If the external memory system cannot handle burst mode operations, the instruction burst mode should be left disabled, so that the prefetch buffer can be used.

4.2.7 Cache Off

Instruction and data burst mode is automatically disabled if the corresponding cache is turned off.

4.2.8 Bus Request

The bus will be released to service another request only after the completion of the burst mode transaction.

4.2.9 Memory Exception (Instruction fetches or Data loads)

All four word accesses of a burst mode access will be completed even if a memory exception occurs on any of the word accesses. During a burst access, word accesses that cause an external memory exception ($-MEXC$ asserted) are not written into the cache, while any words that do not cause a memory exception are written to cache. Note that the Interger Unit will recognize a memory exception only when it is accessing the specific word with which the memory exception is associated.

For example, if the IU requested word 00, the BIU would burst-read 00, 01, 10 and 11. If an external memory exception occurred only on word 10, this word would not be written to the cache; the other three words, however, would be written to the cache. The IU would not vector to the `memory_exception` trap handler, since there was no memory exception on the specific word it requested.

If, however, the IU ever tried to access word 10, which was not written into the cache because of the memory exception, a miss would occur which would cause the BIU to fetch that word from memory again. If a $-MEXC$ were asserted on this access of word 10, the processor would vector to the `memory_exception` trap handler, since this was the word specifically requested by the IU.

4.2.10 Memory Exception (DMA)

When a memory exception ($-MEXC$ strobed) occurs on any word of a DMA burst read, the DMA will complete all four reads. The corresponding four writes, needed to complete the transaction, will not occur.

When a memory exception occurs on any word of a DMA burst write, the DMA will continue, completing all four writes.

A memory exception on a DMA transfer will not cause the IU to vector to the `data_memory_exception` trap routine.

4.2.11 Non-cacheable Accesses

Burst mode fetches from a non-cacheable address space are not supported. The burst request signal ($-BMREQ$) will not be asserted, and only a single-word fetch will be performed.

4.2.12 Interface Timing

Figure 3-BIU-1 below shows the timing of a burst mode transaction for an instruction fetch, data load, or DMA read. To start the transaction, the MB86932 outputs a burst mode request signal ($-BMREQ$) to the memory system. The memory system asserts the burst mode acknowledge signal ($-BMACK$) to the processor when

the first word is fetched, indicating that a burst mode request can be handled. The -BMACK should be asserted only in the cycle when the -RDY for the first access is asserted. The memory latency involved in the first word fetch is the same as in a non-burst access, and subsequent fetches are usually shorter; as in the Figure, a single cycle. This does not mean that each fetch following the first will occur in one cycle; subsequent fetches can take any number of cycles, depending on the -RDY assertion. The -BMREQ signal is deasserted after the completion of the first word fetch.

If the memory system cannot handle a burst mode transaction, -BMACK will remain deasserted. Once the burst mode logic detects an inactive -BMACK , the burst mode access will terminate. The burst mode logic will not attempt to complete the fetch of the remaining words in the cache line. However, -BMREQ will be asserted again for any subsequent misses. Therefore, for a certain address segment in which the memory system cannot handle a burst mode operation, the -BMACK signal can remain deasserted. An example is shown in Figure B4-3.

Figure B4-4 shows the timing for the write portion of a DMA burst operation. The timing is identical to that in Figure B4-2, except that the RD/-WR line is low, indicating a write operation is in progress.

Note that $\text{ADR}[31:2]$ is the address of the first word fetched. This address remains constant through the burst.

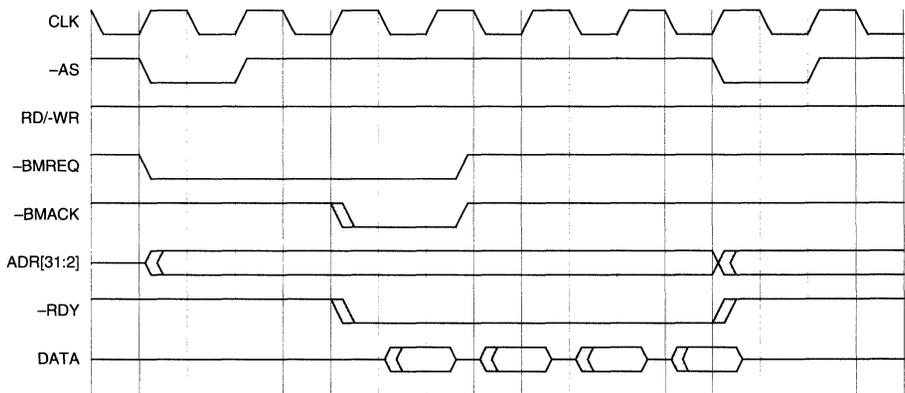


Figure B4-2. Burst Mode (0 wait state)

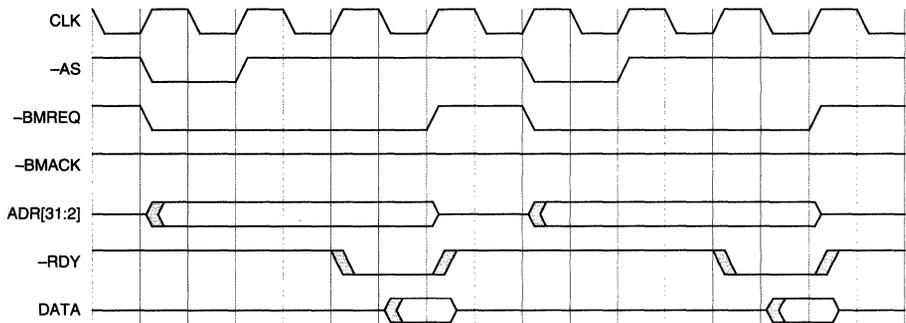


Figure B4-3. Terminated Burst Mode Due to -BMACK=1

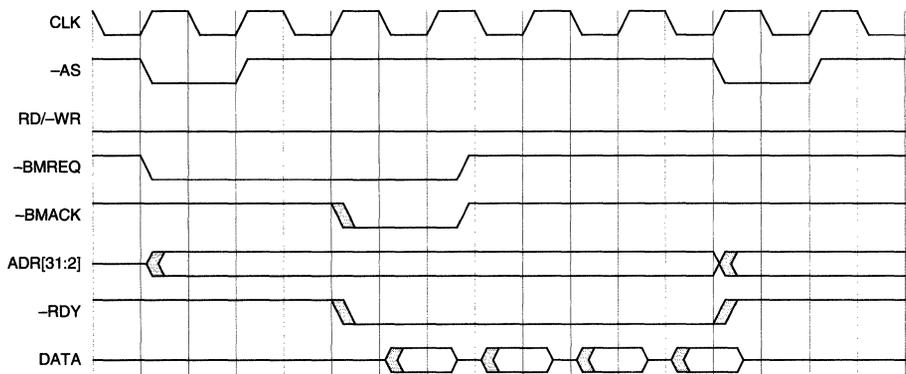


Figure B4-4. DMA Burst Mode, Write Portion

4.3 Parity

The MB86932 provides parity generation/checking for the 32-bit external data bus. Parity can be enabled/disabled for specified address ranges by setting/clearing bits in the Wait-State Specifier Register (see section on that register, below). Parity can be set even or odd by setting bit 0 in the System Support Control Register: set to 1, odd parity is generated/checked; set to 0, even parity is generated/checked. On reset, the value of this bit is cleared to 0.

Parity is generated/checked for every byte of data (resulting in four parity bits). If parity is odd, the parity bit is set to 1 when there are an odd number of 1's in the data; if parity is even, the parity bit is set to 1 when there are an even number of 1's in the data. When enabled, parity is generated for all writes to external mem-

ory. Incoming parity is checked only for the address ranges for which the “PE” bit in the corresponding Wait-State Specifier Register is set to 1. If a parity error is detected on an instruction fetch, an instruction_memory_exception occurs, and bit 8 in the Instruction Fault Status Register is set (see TLB section). If a parity error is detected on a data fetch, a data_memory_exception occurs, and bit 9 in the Data Fault Status Register is set (see TLB section). The parity bits will have a longer setup/delay time than the other data bits.

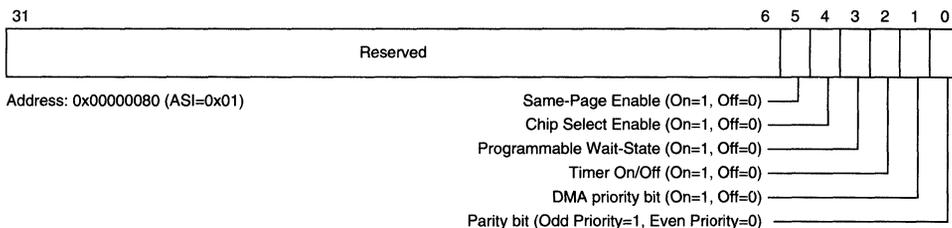


Figure B4-5. System Support Control Register

4.4 Wait State Specifier Register

4.4.1 Purpose

The Wait-State Specifier Register (WSSR) format on the MB86932 has been changed from that on the MB86930 to accommodate the burst mode bus transaction using internal $\overline{\text{READY}}$ and Parity generation/checking.

4.4.2 Format

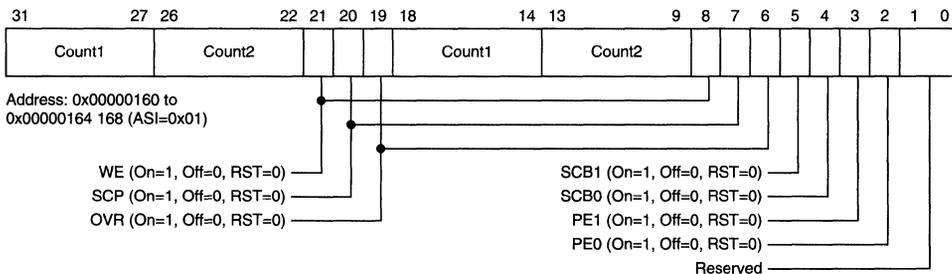


Figure B4-6. Wait State Specifier Register

The bits in the WSSR can have two different meanings depending on whether burst mode is enabled or disabled.

4.4.3 Same Page Mode

Burst mode disabled or burst mode enabled and –BMACK not asserted for this region.

- Count1: Count1 +1 is the number of wait states inserted before internal –READY is asserted, under the following conditions: SCP=0, and current access is not in the same page as the previous access.
- Count2: Count2 +1 is the number of wait states inserted before internal –READY is asserted, under the following conditions: SCP=0, and current access is in the same page as the previous access.
- WE: Wait Enable, enables or disables the internal wait state generation for the individual address range. IF WE is 1 SCP must be 0.
- SCP: If this bit is 1 the internal –READY is generated in the same cycle when an access is started. All accesses to external memory in this address range will be single cycle. IF SCP is 1 WE must be 0.
- OVR: Allows the system to terminate the memory operation before the internally specified time. If the OVR bit is set to 1, and the external hardware asserts external –READY signal, the wait state generator will stop counting and will wait for the next transaction.
- SCB: Unused; should be 0.
- PE: Enable checking of Parity. PE1, PE0 correspond to address ranges for WSSR[31:19] and WSSR[18:16] respectively.

4.4.4 Burst Mode

Burst mode enabled and –BMACK is asserted.

- Count1: Count1 +1 is the number of wait states inserted before internal –READY is asserted, for the first access of a burst mode transfer.
- Count2: Count2 +1 is the number of wait states inserted before internal –READY is asserted, for the 2nd, 3rd and 4th access of a burst mode access if SCB=0.
- WE: Wait Enable, enables or disables the internal wait state generation for the individual address range. If WE is 1, SCP must be 0.
- SCP: If this bit is 1, the internal –READY is generated in the same cycle when an access is started. All accesses to external memory in this address range will be single cycle. If SCP is 1, WE must be 0.
- OVR: Allows the system to terminate the memory operation before the internally specified time. If the OVR bit is set to 1, and the external hardware asserts external –READY signal, the wait state generator will stop counting and will wait for the next transaction.
- SCB: If this bit is 1, in the burst mode all accesses after the first access take a single cycle. If this is 1, Count2 is ignored. SCB1 and SCB0 correspond to address ranges for WSSR[31:19] and WSSR[18:6] respectively.
- PE: Enable checking of Parity. PE1, PE0 correspond to address ranges for WSSR[31:19] and WSSR[18:6] respectively.

Table B4-2: RESET State

WSSR reset state for $-\text{CS}[1]$ to $-\text{CS}[5]$:	WSSR reset state for $-\text{CS}[0]$:
Count2=0	Count2=31
Count1=0	Count1=31
WE=0	WE=1
SCP=0	SCP=0
SCB=0	SCB=0
OVR=0	OVR=1
PE=0	PE=0

4.5 ROM Interface

4.5.1 Purpose

The data bus of the MB86932 can be configured upon reset to 8- and 16-bit bus modes as well as the standard 32-bit mode. This flexibility accommodates those cases in which boot code resides in PROMs organized as blocks of bytes or half-words.

4.5.2 Features

Bus Configuration: the data bus configurations are fixed to specific segments of the bus:

- 8-bit mode: D[7:0]
- 16-bit mode: D[15:0]
- 32-bit mode: D[31:0]

4.5.3 Bus Configuration on Reset

Two external pins, $-\text{BMODE}16$ and $-\text{BMODE}8$ are used to determine the bus configuration. The two bus configuration pins have weak pull-ups, so that if unconnected, the bus configuration will default to a 32-bit bus.

(reserved): $-\text{BMODE}16=0$, $-\text{BMODE}8=0$

8-bit mode: $-\text{BMODE}16=1$, $-\text{BMODE}8=0$

16-bit mode: $-\text{BMODE}16=0$, $-\text{BMODE}8=1$

32-bit mode: $-\text{BMODE}16=1$, $-\text{BMODE}8=1$

4.5.4 System Interface

In order to minimize external “glue logic” required for interfacing to the 8- or 16-bit bus, the BE bits are encoded to reflect the two LSBs of a byte address or the LSB of a halfword address. Therefore, the ADR[31:2] and selected $\overline{\text{BE}}$ bits can be concatenated to form a complete address for a non-32 bit bus mode.

Table B4-3: System Interface BE Bits

Bus Mode	Byte	BE[0:3]
8-bit bus	0	0000
	1	0001
	2	0010
	3	0011
16-bit bus	0 & 1	0000
	2 & 3	0010

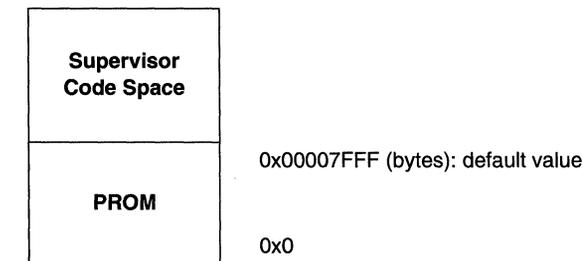
8-bit bus mode address= {ADR[31:2], $\overline{\text{BE}}$ [2], $\overline{\text{BE}}$ [3]}
 16-bit bus mode address={ADR[31:2], $\overline{\text{BE}}$ [2]}

$\overline{\text{CS}}[0]$, which is enabled on reset, and the internal $\overline{\text{READY}}$ generation logic, can be used to minimize any glue logic required to interface to the PROM. On reset, the wait state generator, corresponding to $\overline{\text{CS}}[0]$ for internal $\overline{\text{READY}}$ generation, is set to 32 cycles. Later on in the boot code, the wait state generator can be changed to a more appropriate value.

4.5.5 PROM Address Space

The PROM address space is defined by the $\overline{\text{CS}}[0]$ address-range specifier. On reset, the $\overline{\text{CS}}[0]$ address range defaults to 32K bytes (starting address=0x0), and the ASI is initialized to 0x9. The PROM address range can be changed later using the mask bit register associated with $\overline{\text{CS}}[0]$. An example of the supervisor address space (ASI=0x9) memory map is shown below:

Figure B4-7. Supervisor Address Space (ASI=0x9) Memory Map



Any memory access from the PROM address space, in a non-32 bit mode, will make the -BE bit encodings reflect the LSBs of a byte/halfword address. Furthermore, the fetched bytes/halfwords will be assembled into a 32-bit word. On the other hand, any access from the non-PROM address range will result in a normal, 32-bit memory access.

4.5.6 Load/Stores

One of the functions of the boot code is to set the processor and system configuration. This might involve loading system parameters from PROM, loading data from memory mapped I/O, and storing data to non-PROM address space. All loads from the PROM address space behave the same way as instruction fetches, in that, for a non-32 bit bus mode -BE , bit encoding and word assembly are done. Loads from a non-PROM address space behave in the normal (32-bit) manner. In order to meet the -BE AC timing, the -BE bits on the MB86932 need to be all 0's for all types of loads—word, halfword, and byte—from the non-PROM address space. This requires a functional change from the current specification of the MB86930's -BE bits, which reflect the byte information for loads. This change does not cause a problem, since the processor fetches a full 32-bit word on a load, and the IU selects the byte appropriately. As on the MB86930 -BE bits should be ignored for 32-bit loads.

Since stores to the PROM will never occur, for all stores, regardless of address space, the -BE bits will reflect the byte information of the store. Therefore, byte and halfword stores to the PROM address space becomes meaningless, since the $\text{-BE}[2]$ and $\text{-BE}[3]$ bits no longer reflect the byte address. Furthermore, store word operations to the PROM address space will not result in a dis-assembly process for a non-32 bit bus mode. Since stores to PROM address space are not disabled, the user would have to qualify $\text{-CS}[0]$ with the R/ -W signal to use it as a PROM chip select signal. This will not be necessary if the user can be sure that a store to PROM space never occurs.

A summary of the $\text{-BE}[0:3]$ bit behavior for loads from the PROM address space is shown below. For all load instructions (byte, halfword, word), a full 32-bit fetch occurs. For example, in the 8-bit bus mode, four bytes will be fetched for all loads, and the BE bits will sequence with the proper 2 LSBs of the byte address.

Table B4-4: Load $\text{-BE}[0:3]$ Bit Behavior

Bus Mode	Operation	BE[0:3] in PROM space
8-bit bus	Loads (all)	0000=>0001=>0010=>0011
16-bit bus	Loads (all)	0000=>0010
32-bit bus	Loads (all)	0000

4.5.7 Burst Mode

Since speed is not a critical issue when executing boot code out of PROM, and because there is no industry-wide standard for a burst-mode EPROM interface, burst-mode interface is not supported for accesses from PROM address space. When the system has a 8/16 bit memory being used for boot code, it should not assert -BMACK for any accesses to -CS0 .

4.5.8 Memory Exception

Any memory exception that occurs during a fetch from the PROM address space in a non-32 bit bus mode will be held off until the entire word is fetched.

4.5.9 Bus Request

Any bus request happening during the non-32 bit bus mode fetch will not be recognized until the end of the complete 32-bit fetch operation.

4.5.10 Timing

Timing examples for the 8- and 16-bit bus modes with 1 wait-state memory are shown below. Note that -AS is asserted at the beginning for one cycle.

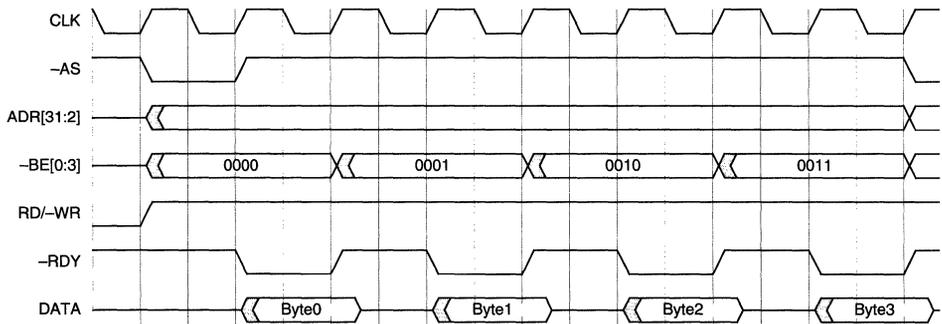


Figure B4-8. 8-bit Bus Mode (1 Wait State)

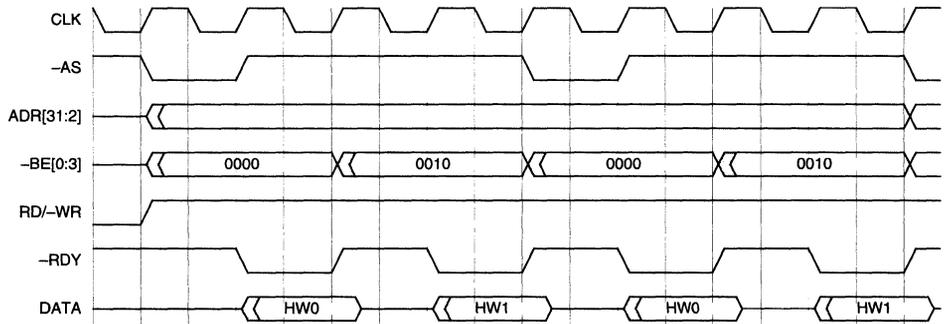


Figure B4-9. 16-bit Bus Mode (1 Wait State)

4.6 Processor Bus Request

4.6.1 Purpose

When the bus is released in response to an external device's request for the bus (by asserting -PBREQ), the MB86932 processor cannot access the bus as long as the bus request signal remains asserted. An external bus arbiter may never be aware that the processor needs the bus back. To remedy this problem, a processor bus request signal is asserted whenever the external bus is required by the processor. The external bus arbiter then can release the bus to the processor requesting it. Also, in a bus-based multiprocessor system, a processor bus request signal is useful to the external bus arbiter in deciding which processor requires the bus.

4.6.2 Features

-PBREQ pin: An external pin is used to output the processor bus request signal, -PBREQ . The -PBREQ will be asserted whenever the MB86932 requires the bus while the bus is granted to an external device. The external device using the bus

can monitor the -PBREQ signal, and remove the -BREQ signal at an appropriate time. An example of the -PBREQ timing is shown in the figure below:

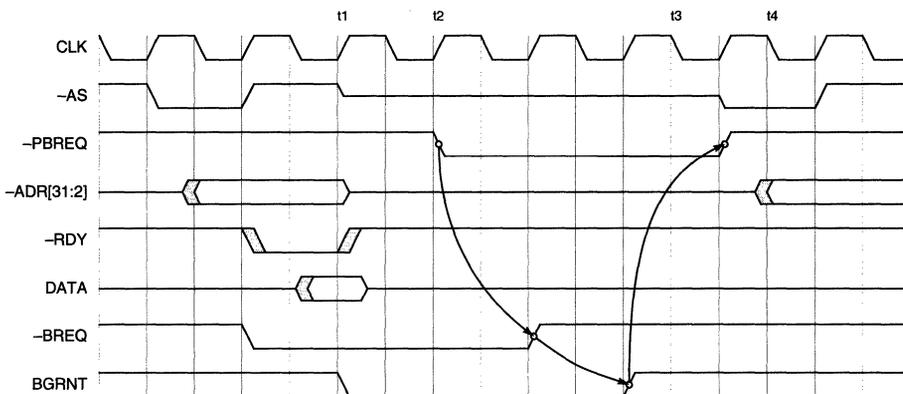


Figure B4-10. Example of -PBREQ timing

In the example above, the bus is released at the beginning of cycle t1 in response to an external bus request. At t2, -PBREQ is asserted because of a pending bus cycle in the processor. The external bus arbiter de-asserts -BREQ , and returns the bus to the processor. -PBREQ remains asserted until the end of the cycle t3. At t4, the processor drives the bus.

4.7 BIU Timing

4.7.1 Effect of TLB

Since the TLB can be used with the cache turned off, a one-cycle delay is introduced at the beginning of each memory operation to complete the TLB translation. For cache-on cases, the TLB does not introduce an additional delay since address translation occurs during the one-cycle already available for cache hit/miss detection. The first figure below shows the timing for the cache-off, TLB-off

case; the second figure shows the timing for the cache-off, TLB-on case. Note in the second figure the one-cycle delay for each new memory operation.

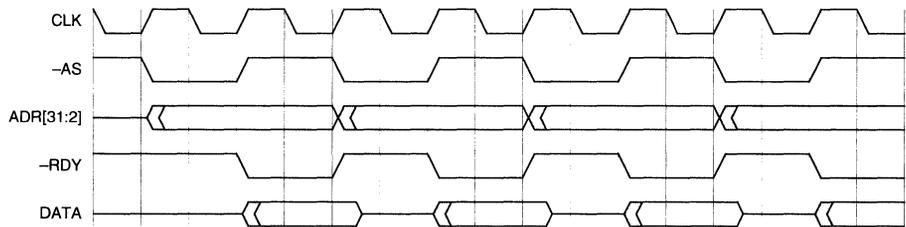


Figure B4-11. Cache=off, TLB=off (1 wait state)

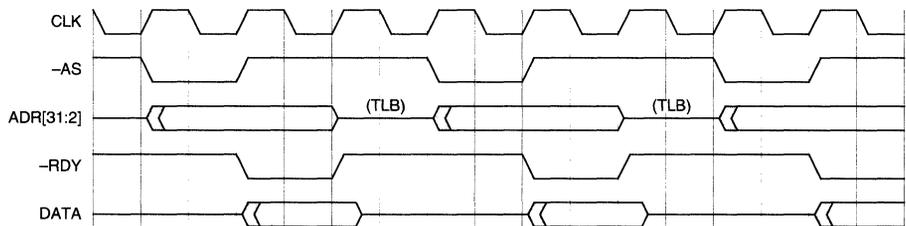


Figure B4-12. Cache=off, TLB=on (1 wait state)

4.8 BIU Priorities

In general the following hierarchical rules apply when multiple requests are made to the bus interface unit:

- The bus cycle currently in progress will complete.
- If there is a pending external bus request, the bus will be granted to the external requestor.
- If there is a pending DMA request, the bus will be granted to the DMA controller.
- If the write buffer is full, the buffer will be emptied.
- If there is a pending load or store operation it will be serviced.
- If there is a pending request for an instruction it will be fetched.
- If the prefetch buffer is empty, a prefetch cycle will be initiated.

Note that bit1 in the System Support Control Register can be used to allow the IU to “steal” cycles from the DMA. When this bit is set the DMA will de-assert its

request after each datum is moved. When cleared the DMA will keep the bus until the whole DMA transaction has completed.



MB86932 DMA

5.1 Overview

The Direct Memory Access Controller (DMAC) module provides high-speed memory-to-memory and memory-to-peripheral data transfers. The DMAC executes independently of the CPU, making it possible for the processor to execute from cache while DMA transfers are taking place. The DMAC operates on physical addresses.

The DMAC supports two independent DMA channels concurrently. It supports byte, half-word, word and quad-word transfers. The DMA mechanism provides three different methods of performing DMA transfers: Single transfer, Demand transfer, and Block transfer. Single transfer and Demand transfer use the DMA request (-DREQ) and DMA acknowledge (-DACK) signals to synchronize transfers with external devices. Block transfers do not use -DREQ and -DACK, they are typically used to transfer data from memory to memory.

“Fly-by” transfer mode is supported for high speed DMA transfers. In this mode, a single bus transaction transfers the data from source to destination. “Flow-Thru” transfer mode is also supported. In this mode, two bus transactions, a read followed by a write, need to be performed to complete the transfer of data from source to destination.

The DMA channels can be configured to perform a single buffer transfer, or to operate in the buffer-chaining mode. The buffer-chaining mode is provided to simplify operations such as scatter/gather. In this mode, the DMAC is configured

with a series of descriptors in memory. Each descriptor describes a single buffer transfer, which is part of the complete DMA transfer.

The two figures that follow give, respectively, an overall picture of the relationship of the DMAC to other major functional components of the MB86932, and a detailed picture of the flow within the DMAC.

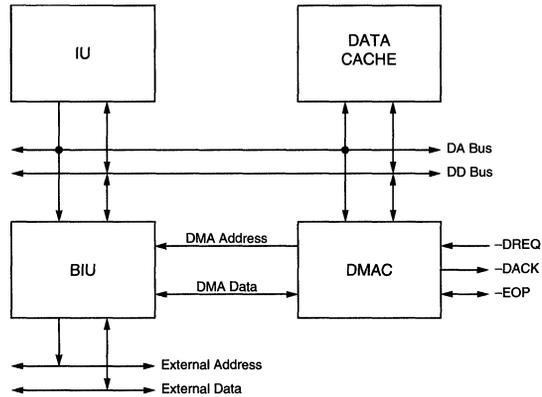


Figure B5-1. Relation of DMAC to Other Major Components

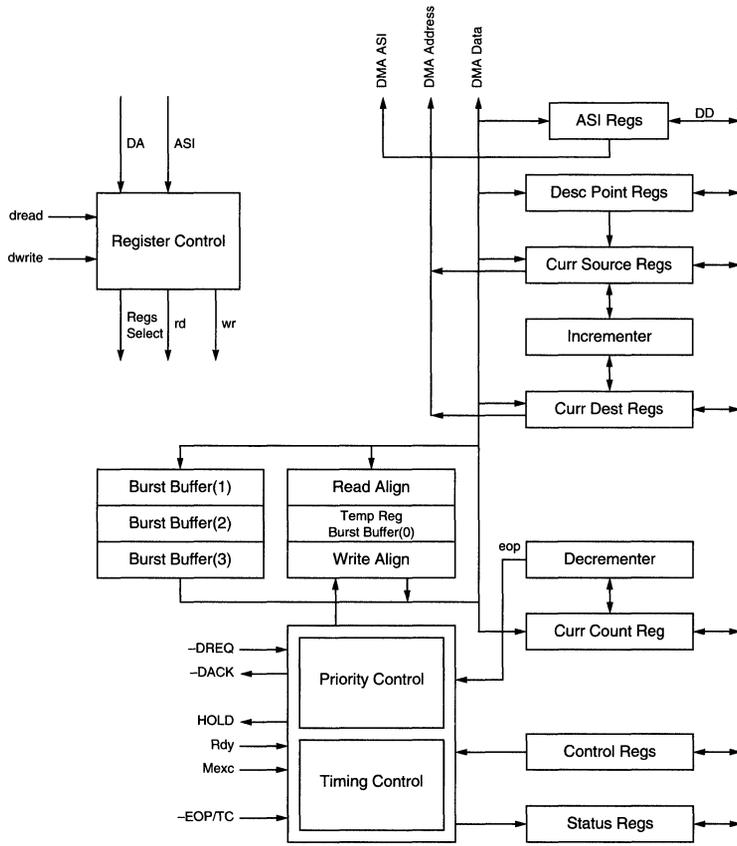


Figure B5-2. DMA Block Diagram

5.2 Programmer's Model

Table B5-1: DMA Signal Descriptions

Signal	Function
-DREQ1 / -DREQ0	DMA REQUEST (I): This input signal indicates that an external device is requesting DMA transfer. It is an edge-sensitive signal for single transfer, and a level-sensitive signal for demand transfer.
-DACK1 / -DACK0	DMA ACKNOWLEDGE (O): This output signal is sent to the external device to acknowledge the DMA request, and is active when the requesting device is accessed.
-EOP1 / -EOP0	END OF PROCESS (I/O): This pin is used as input when an external device wants to cause the DMA process to terminate. It functions as output when the byte count reaches zero. When not active, -EOP output will be tristated. For signalling the Terminal Count (TC) , -EOP will be pulled down, and then be pulled up for one cycle. A high impedance internal pull up is used to hold the signal high when -EOP is tristated. The -EOP issued by the DMAC can be used as input to the interrupt controller. If -EOPx is asserted by the external device, channel x will be disabled. Reprogramming is needed to enable a channel.

Six pins are dedicated to the DMAC, three for each channel. In the table above, the pin number corresponds to the channel number. For example, the -DREQ0 pin is the request pin for channel 0.

5.2.1 DMA Priority

The DMA Priority Bit in the System Support Control Register can be programmed to indicate whether the DMA is to release the bus for one clock cycle so that the IU can use it. When this bit is set, the DMAC will deassert the HOLD signal to the BIU for one clock cycle after a DMA entry has been transferred. In this way, the IU can steal a bus cycle when service is needed. When this bit is cleared, the DMA blocks the IU from using the BUS until the whole DMA transaction completes.

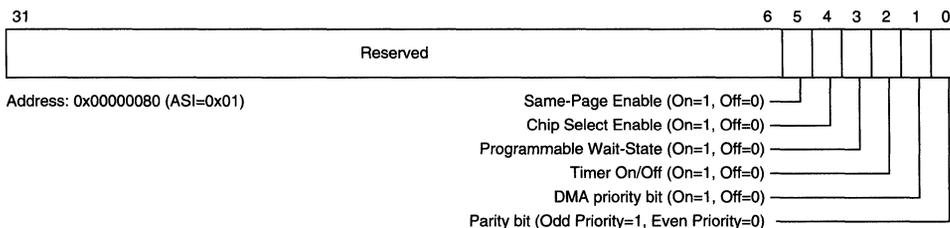
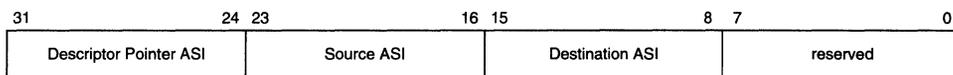


Figure B5-3. System Support Control Register

5.2.2 DP/Source/Destination ASI Register



Address: 0x00000180 (DMA0) (ASI = 0x01)
 0x000001A0 (DMA1)

Figure B5-4. DP/Source/Destination ASI Register

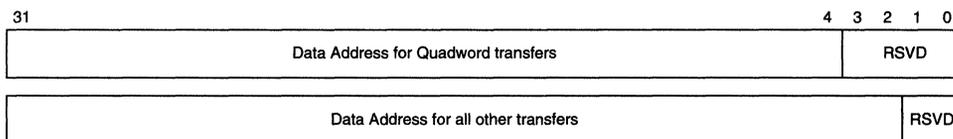
Bits 31-24: Description Pointer ASI (DP ASI)—ASI of the Descriptor Pointer, a register used in buffer-chaining mode. It points to the next element of the linked list whose elements describe the source and destination of the DMA transfer.

Bits 23-16: Source ASI—ASI of the Current Source Address Register, which is described below.

Bits 15-8: Destination ASI (Dest ASI)—ASI of the Current Destination Address Register, which is described below.

Bits 7-0: Reserved

5.2.3 Current Source Address Register

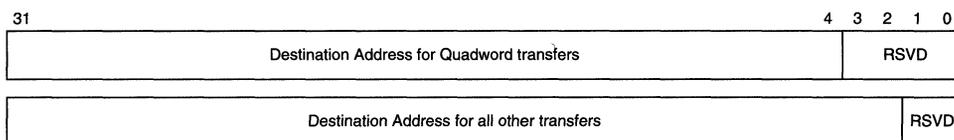


Address: 0x00000184 (DMA0) (ASI=0x01)
 0x000001A4 (DMA1)

Figure B5-5. Current Source Address Register

The Current Source Address Register is used to address memory accesses in flyby mode, and to hold the source data address in flowthru mode. It contains one 30-bit (31:2) word-aligned address. For byte, halfword, and word transfers, all 30 bits (31:2) are used; for quadword transfers, only 28 bits (31:4) are used. Bits beyond the current address field are ignored. The CSA Register value is updated after a transfer in the read phase has been done, and points to the next location to be transferred. Note that in flyby mode, a DMA transfer has just one Read/Write phase; in flowthru mode, a DMA transfer has one read phase, one write phase, and an intervening idle clock cycle.

5.2.4 Current Destination Address Register



Address: 0x00000188 (DMA0) (ASI=0x01)
0x000001A8 (DMA1)

Figure B5-6. Current Destination Address Register

The Current Destination Address Register is not used in flyby mode; it holds the destination data address in flowthru mode. It contains one 30-bit (31:2) word-aligned address. For byte, halfword, and word transfers, all 30 bits (31:2) are used; for quadword transfers, only 28 bits (31:4) are used. Bits beyond the current address field are ignored. The CDA Register value is updated after a transfer in the write phase has been done.

5.2.5 Current Byte Count Register



Address: 0x0000018C (DMA0) (ASI=0x01)
0x000001AC (DMA1)

Figure B5-7. Current Byte Count (CBC) Register

The CBC register indicates the number of bytes of data still left to be transmitted. The value of the data should be programmed to be one less than the actual number of bytes to be transmitted. For example, to transfer two words, this register should be loaded with the value "7". The value will be decremented at the beginning of the DMA transfer cycle by the number of bytes involved in the transfer, regardless of the unit in terms of which the transfer is specified (half-word, word, etc.). The Byte Count Register is updated only in the Read phase, not in the Write phase; it is updated at the beginning of the transfer.

5.2.6 Descriptor Pointer Register

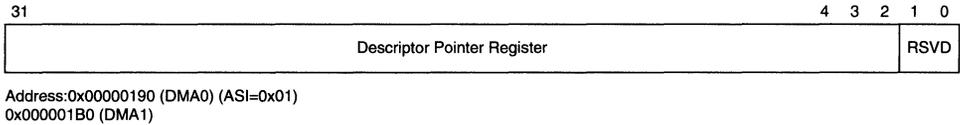


Figure B5-8. Descriptor Pointer (DP) Register

Used in Chaining Mode, the DP Register points to the next element of the linked list. Successive elements of the list describes the source and destination of successive buffers to be transferred.

5.2.7 Channel Control Register

Bits 31 to 16 are reserved, ignored on a Write, and Read as zero. The entire register is reset to zero. Note that the two channel control registers are not identical: the HPC and SW bits in the channel 0 register are global, while the same bits in the channel 1 register are reserved, and read as undefined.

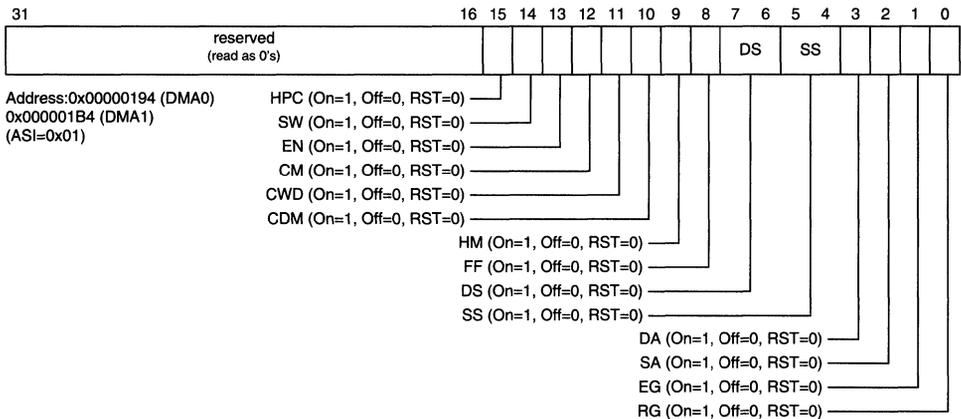


Figure B5-9. Channel Control Register

The Channel Priority Switch Mode bit “SW” and the High Priority Channel bit “HPC” of the channel 0 Control Register determine the priority setup of the DMA Controller. These two global bits should be programmed only when both channels are disabled.

Bits 31-16 Reserved

- Bit 15: High Priority Channel (HPC)—0 if channel 0 has high priority; 1 if channel 1 has high priority. (The HPC should be programmed to specify the channel that has high priority at the outset; if SW=1, it will be updated to show the current high-priority channel as the DMA transfer progresses. Note that this bit exists only in the channel 0 control register; the corresponding bit in the channel 1 control register is reserved, and read as undefined.
- Bit 14: Channel Priority Switch Mode (SW)—0 if fixed, 1 if switchable. (If 0, the HCP is fixed, and specifies a prechosen higher priority channel; if switchable, the HCP will be updated to whichever channel is not currently being serviced.) Note that this bit exists only in the channel 0 control register; the corresponding bit in the channel 1 control register is reserved, and read as undefined.
- Bit 13: Enable [Start] DMA (EN)—0 if disable channel, 1 if enable. (The DMA channel can be enabled by writing 1 to this field, and is reset by the hardware when the channel enters the disabled state. In Internal Request mode (see RG field), a 1 here means Start DMA; in External Request mode, a 1 here means Accept External DMA request.)
- Bit 12: Chaining Mode (CM)—0 if reprogramming, 1 if buffer chaining.
- Bit 11: Chaining Wait Mode (CWM)—0 if Chaining Wait Function disable, 1 if enable. (Decides whether next chaining descriptor is to be read.)
- Bit 10: Chaining Debug Mode (CDM)—0 if assert -EOP only after the whole Chaining transfer, 1 if assert -EOP after each buffer transfer
- Bit 9: Transfer/Handshake Mode (HM)—0 if Single Transfer, 1 if Demand Transfer. (Applies only to external request; for internal program request, DMAC supports block transfer mode only.)
- Bit 8: Flyby/Flowthru (FF)—0 if Flyby (single address), 1 if Flowthru (Dual Address).
- Bits 7-6: Destination Size (DS)—00 if word, 01 if byte, 10 if halfword, 11 if quadword.
- Bits 5-4: Source Size (SS)—00 if word, 01 if byte, 10 if halfword, 11 if quadword.
- Bit 3: Destination Addressing (DA)—0 if increment, 1 if hold.
- Bit 2: Source Addressing (SA)—0 if increment, 1 if hold.
- Bit 1: External Control Option (EC)—0 if source request, 1 if destination request.
- Bit 0: Request Generation (RG)—RG=0 if internal request, 1 if external request.

5.2.8 Channel Status Register

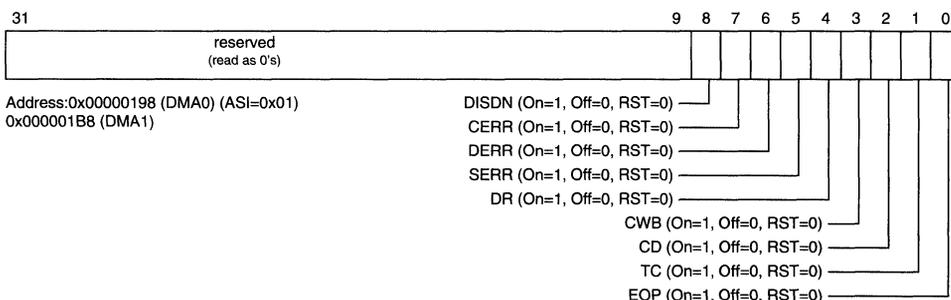


Figure B5-10. Channel Status Register

- Bits 31-9: This register is shown as having only 9 bits because these bits are reserved, ignored on a Write, and Read as zero. The entire register is reset to zero.
- Bit 8: Disable Done (DISDN)—the user can disable the DMA channel by writing 0 to the Enable bit of the Control Register. This bit will be set when the channel has been effectively software-disabled.
- Bit 7: Chaining Error on DMA Transfer (CERR)
- Bit 6: Destination Error on DMA Transfer (DERR)
- Bit 5: Source Error on DMA Transfer (SERR)
- Bit 4: DMA Request presented (DR)—A DMA request is pending.
- Bit 3: Chaining Wait (CWB)—If the Chaining Wait Mode in the Control Register has been set, this status bit will be set after each buffer has been transferred. The Chaining Descriptor fetch will not be executed. After the program redoes the setup for this channel, and clears this status bit, the DMA will proceed with the new register setup.
- Bit 2: Chaining Done (CD)—The whole chain of data buffers have been successfully transferred; set up in chaining mode.
- Bit 1: Terminal Count (TC)—A data buffer has been successfully transferred. It will be set when termination of transfer is reached for nonchaining mode and chaining debug mode.
- Bit 0: End of Process, external (EOP)—Channel transfer stop due to external –EOP signal.

5.2.9 Channel Initialization

The DMA Control has two transfer modes: 1) Single Buffer Transfer Mode, and 2) Buffer Chaining Mode. Each mode has its own programming requirements.

To initialize the DMA Channel for Single Buffer Transfer Mode, the user must program these registers:

- ASI Register
- Current Source Address Register
- Current Destination Address Register
- Current Byte Count Register
- Channel Control Register

After programming these registers, the user writes the start (enable) bit of the Channel Control Register to enable the Channel.

To initialize the DMA Channel for Buffer Chaining Mode, the user must program the registers listed above for Single Buffer Mode transfers, and in addition must program the Descriptor Pointer (DP) Register. The DP points to the next element of the chaining list for the buffers to be transferred. After the channel finishes transferring each block, it will spend five data access cycles to set up the DP/ Source/ Destination ASI Register, the Current Source Address Register, the Current Destination Address Register, and the Current Byte Count Register. The last chaining cycle is used to get the pointer and put it in the Descriptor Pointer Register.

When TC happens, the DMA will load the chaining information pointed to by the DP, and the DMA process continues. An external –EOP will disable the channel.

In chaining mode, whether block or demand transfers are being carried out, a channel that has reached TC will load the chaining block descriptor, and the DMA Controller will see if a request from the high priority channel is outstanding. If it is, the DMAC will suspend the next transfer of the present sequence, and release the bus to the high priority channel. For example: assume that priority switching mode is in effect; channel 0, the original high priority channel, is in chaining mode; and channel 1 is in reprogramming mode. If both channels get –DREQ asserted, channel 0 will be serviced first. WHEN TC is reached, DMAC will load the information for the next transfer block; the outstanding request from channel 1 will be noted, and—because channel 1 is the high priority channel—its request will be serviced now.

5.2.10 Buffer Chaining Data Structure

- PSDASI (Descriptor, Source, and Destination ASI)
- SA (Source Address)
- DA (Destination Address)
- BC (Byte Count)

- NPTR (Next Buffer Descriptor Pointer); a NULL pointer, 0000, indicates the end of the block buffer list.

5.2.11 DMA Initialization

DMA operations can be initiated by either software request or hardware request. A software request is made by clearing the Request Generation bit and setting the DMA Enable bit. A hardware request is made by setting the Request Generation bit and the DMA Enable bit, and then causing the assertion of an external -DREQ .

When the CPU clears the Request Generation bit and sets the DMA Enable bit, the software-initiated DMA starts immediately. A hardware request is started only when -DREQ is asserted while the DMA Enable bit is set. -DREQ is edge-sensitive for Single Transfer Mode, level-sensitive for Demand Transfer Mode. For Demand Mode to complete a whole buffer block, -DREQ must be asserted until -EOP is asserted. -EOP can be asserted by the DMA Controller or an external device.

5.2.12 Basic DMA Timing

1. For a single transfer, the DMAC will sample -DREQ for the next DMA request after -DACK is asserted. That is, DMAC will try to detect the edge that signals such a request; an edge asserted between that which caused the last transfer and the assertion of -DACK will be ignored. Even if an edge is detected before the DMAC releases the bus, the DMAC will still release the bus and then request it again.
2. -DACK will toggle during the read or write cycle to enable the peripheral device. Ready (from BIU) will be used to deassert the -DACK .
3. -DACK is used for handshaking with a peripheral device to deassert the -DREQ for single transfer mode. -EOP(TC) is used for handshaking with a peripheral device to deassert the -DREQ for demand transfer mode.
4. TC will be used to enable the reloading of the address/count to the current registers to initialize the set up for a buffer chaining transfer. External -EOP will disable the DMAC channel in chaining mode, and leave the state of the channel as it was.

5.2.13 Error Conditions

Memory Access Exceptions:

- Source Transfer Exception
- Destination Transfer Exception
- Chaining Exception

When an Error condition occurs, the relevant bits in the Status Register will be set up, and -EOP will be asserted.

When a memory-exception occurs, -EOP will be asserted one cycle later. This -EOP can be used as input to the interrupt controller. The -EOP due to a memory exception can be deasserted by clearing the status bit of the corresponding exception.

For quad-word transfers, if an exception occurs during the read phase, DMA will still finish all four reads, but will not go into the write phase. If an exception occurs during the write phase, DMA will complete all four writes.

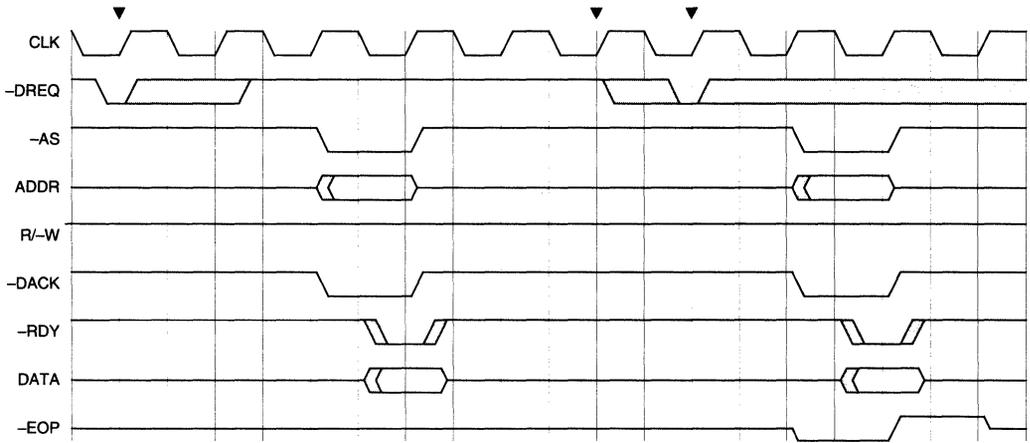
For transfers other than quad-word, the DMA will stop immediately after the exception occurs.

5.3 External Interface

5.3.1 Transfer Protocols

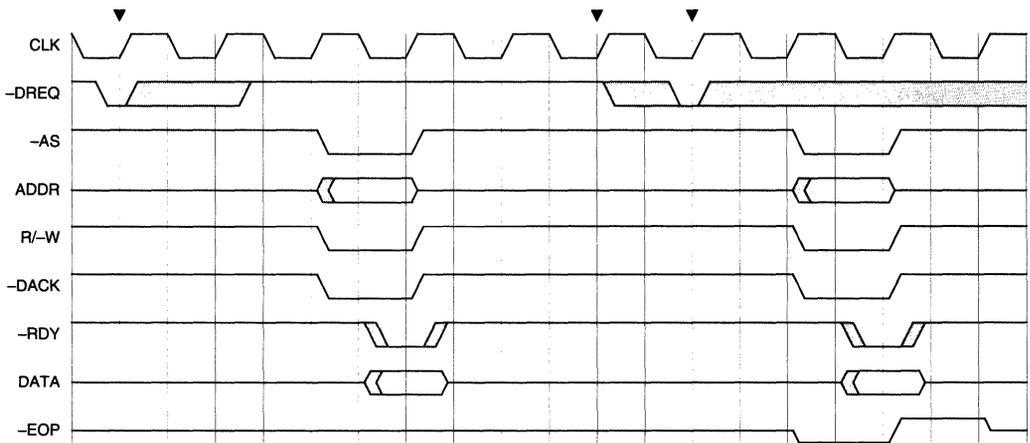
Single Transfer Mode

In the Single Transfer Mode, one data entry transfer from source to destination is performed by the DMAC at a time. The -DREQ input is arbitrated according to the channel priority decisions made by the user. The channel with the DMA request will signal the BIU for bus service. After a DMA data entity has been transferred, control of the bus will be released. Transfers continue in this manner until the Byte Count expires, or until external -EOP is found active. Since the -DREQ is edge-sensitive for single transfers, a -DREQ pulse will cause only one transfer, no matter what its length. The channel will request the bus for each DMA transfer. Bus control is released between each transfer and the next. The DMAC will sample the next -DREQ edge for a DMA transfer request after -DACK is asserted. A new request edge coming before -DACK has been asserted will be ignored. A timing diagram for single transfer mode is given below in Figure B5-11. This diagram shows two consecutive DMA transfers. A sample High and then Low of -DREQ constitutes an edge request for a transfer. The last block transfer is accompanied by -EOP . -R/W is asserted High in flyby mode for a destination transfer—that is, one where data will flow from memory—and asserted Low for a source transfer, where data will flow to memory. In Figure B5-13 below, showing a quadword transfer taking four data cycles. The last DMA transfer is accompanied by EOP .



▼ Is the sensing edge for -DREQ

Figure B5-11. Single Transfer, Edge-Sensitive, Flyby (R/-W high)



▼ Is the sensing edge for -DREQ

Figure B5-12. Single Transfer, Edge-Sensitive, Flyby (R/-W low)

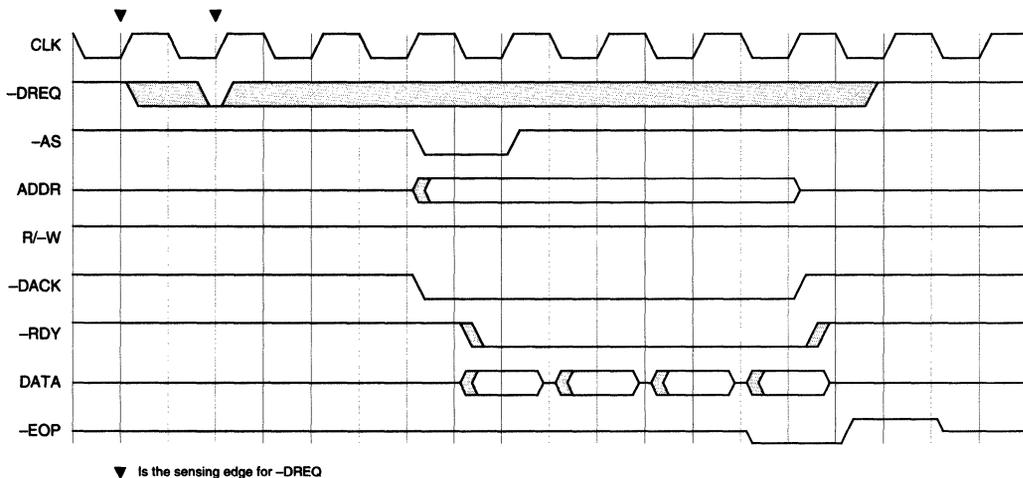


Figure B5-13. Single Transfer, Edge-Sensitive, Flyby, Quadword (R/-W high)

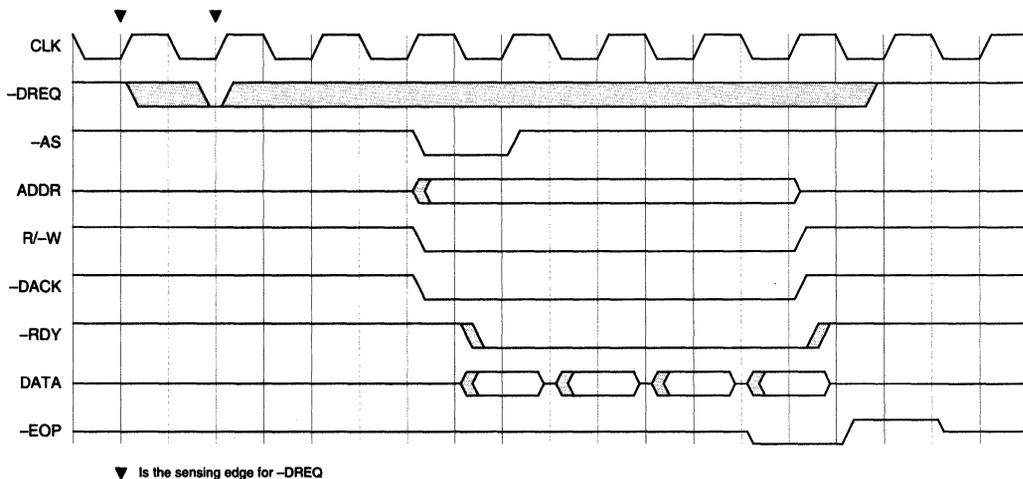


Figure B5-14. Single Transfer, Edge-Sensitive, Flyby, Quadword (R/-W low)

Block Transfer Mode

Block transfer is initiated by software request. In this mode, the CPU starts the DMA action by setting the Start bit of the control register. The transaction will continue until the Terminal Count (TC) happens, or until -EOP is asserted by the external device.

Block transfer mode can be used for either flowthru or flyby transactions. For flyby transactions, the DMAC will assert and then deassert the -DACK for each transferred datum.

A timing diagram for software-initiated block transfer is shown in Figure B5-15 below. The timing is the same as that for demand transfer mode, except that the request is set by software. The transfer will begin two cycles after the channel control register has been written.

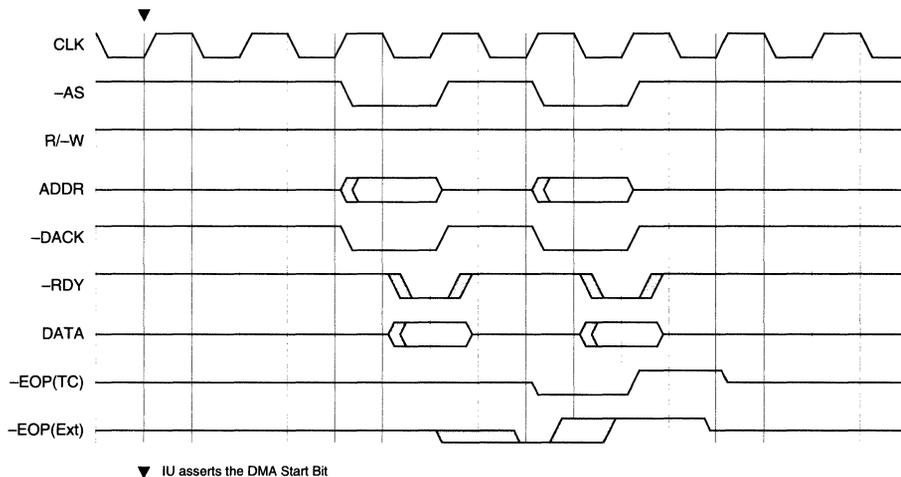


Figure B5-15. Block Transfer, Flyby (R/-W high)

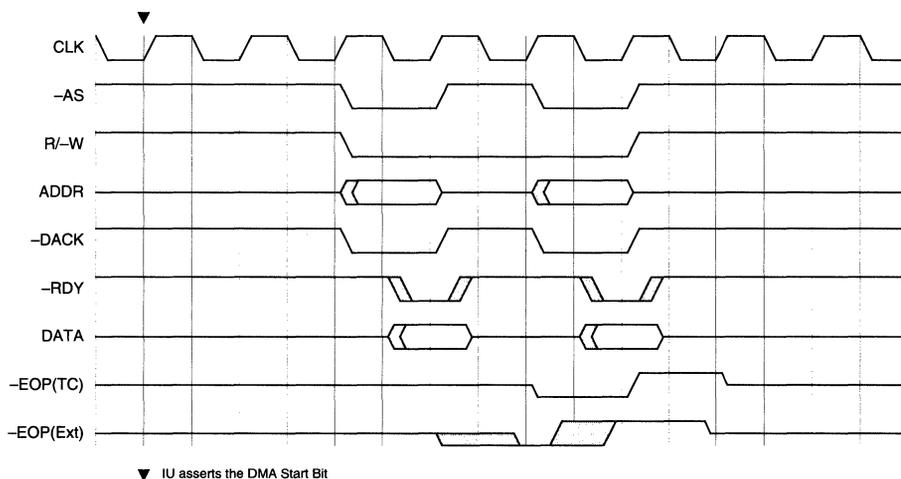


Figure B5-16. Block Transfer, Flyby (R/-W low)

Demand Transfer Mode

Demand Transfer Mode provides flexible handshaking procedures during the DMA process. A Demand Transfer is initiated by an external level-sensitive DMA request (-DREQ). The next request will be sampled after the preceding transfer request has been completed. The process continues until (a) the external device deasserts the -DREQ, (b) the byte count (TC) expires, or (c) an external -EOP is encountered. A timing diagram for demand transfer is shown below in Figure B5-17. When a request for a demand transfer is made, the DMAC will look at the -DREQ to see if any request is pending.

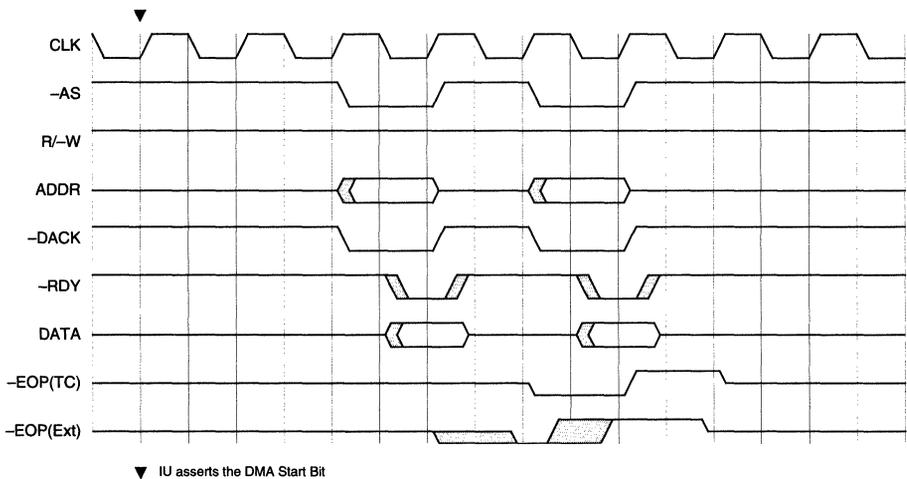
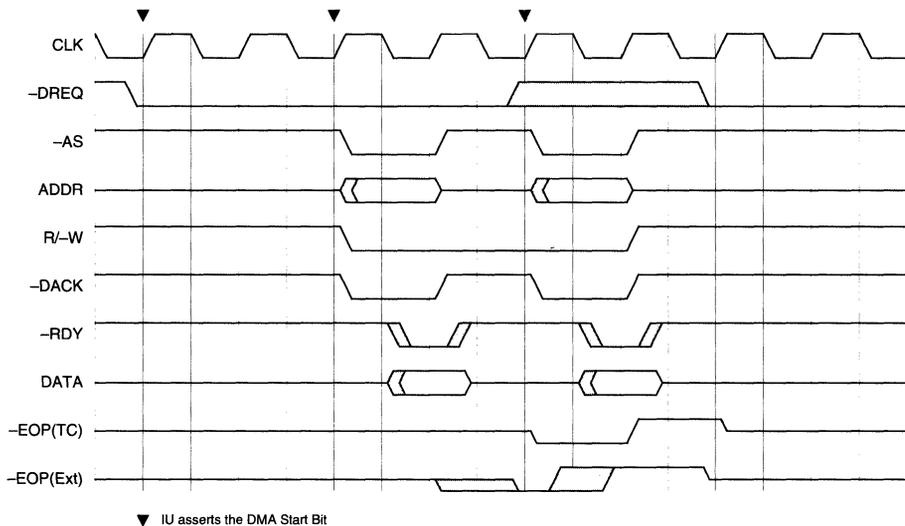


Figure B5-17. Demand Transfer, Flyby (R/-W high)



▼ IU asserts the DMA Start Bit

Figure B5-18. Demand Transfer, Flyby (R/-W low)

Transfer Addressing

- Flyby**—Flyby mode is in effect when the source and destination have the same width, and flyby mode is enabled. -DACK is used to acknowledge the external DMA request, and to access the requestor's data. One bus cycle is needed for a byte, half-word, or word transfer; four bus cycles are needed for a quad-word flyby transfer. A single address is needed for this type of bus operation. The R/-W will signal the direction of data flow; for $\text{R/-W}="1"$, the data flow is from the memory counterpart to the requesting device, and for $\text{R/-W}="0"$ it is from the requesting device to the memory counterpart.
- Flowthru**—For this bus operation, a read sequence is used to obtain the data from the source, and a write sequence is used to send the data to the destination. During read, the data will be assembled and put in a Temporary Register. During write, the data in the Temporary Register will be disassembled and sent to the destination. The DMA Controller will toggle the -DACK during the read or write session, depending on whether the External Control Option (EC) is set to Source or Destination Request. Whichever type of Request is specified by the EC, the other address is optional; for example, if $\text{EC}=0$ (Source Request), the provision of a destination address is unnecessary. The programmer can use the -DACK to enable a read or write to the external device whether the DMA request is internal or external.

Source/Destination Data Length

The source and destination data length can be byte, half-word, word, or quad-word. For flyby transfer, the source and destination data length must be the same. For flowthru mode, if the source and destination data lengths differ, the DMAC will automatically assemble the data during read to the bigger of the two sizes, and disassemble the data to the size of the destination during write. The assembly/disassembly applies only to the byte, half-word, and word sizes.

To take advantage of the burst transfer supported by BIU, the DMAC offers quad-word transfer. Quad-word transfer requires that both source and destination size be quad-word, and both source and destination addresses have to be aligned on quad-word boundaries. The DMAC will assert the quad-word address, and indicate to the BIU that quad-word transfer is needed; BIU will then decide when to proceed with burst-mode transfer.

For consistency with the memory mapping seen by the IU, address (31:2) is used as the byte address for byte transfers, as the halfword address for halfword transfers, and as the word address for either word or quad-word transfers.

Program/DMA Interaction

The -EOP issued by the DMAC can be used as an input to an interrupt controller.

A chaining wait mechanism is supported, enabling synchronization between the program and DMA buffer chaining. This chaining wait function provides a way for the user to modify the channel setup and/or modify the chaining descriptors while a chained DMA activity is in progress. The user can set the chaining wait function bit in the Control Register to enable this function. When this bit is set, and a buffer block has been transferred, the chaining wait bit in the Status Register will be set, and the corresponding DMA channel will go to chaining wait state, which is equivalent to the disabled state. The chaining wait bit set in the Control Register will block the loading of the next descriptor. The user can reprogram the channel, and then reset the chaining wait in the Status Register to restart the transfer. After the block has been transferred, -EOP will be issued as an input to the interrupt controller. The interrupt service routine may modify the channel setup registers and/or the chaining descriptors, and then clear the chaining wait bit in the Status Register. After the chaining wait bit in the Status Register has been cleared, the DMAC will start the DMA transfer using the modified channel setup.

-EOP will be asserted on these conditions:

Single buffer mode:TC (byte count expires)

Error on abnormal read/write transfer.

Chaining mode: If only the chaining mode bit is set,
and the whole chain transfer is completed

Chaining wait function set in Control Register and the TC (byte count expires)

Error on abnormal read/write transfer

If chaining debug mode is set in the control register, -EOP will be asserted at the end of each transferred block.

Note: to use chaining wait, the user must set both chaining mode (CM) and chaining wait mode (CWM) in the control register. To use chaining debug, the user must set both CM and Chaining Debug Mode (CDM) in the control register.

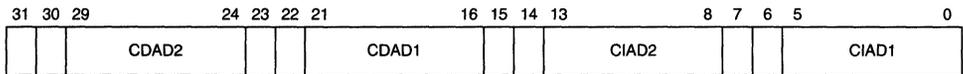
-EOP can be used to interrupt the CPU, and the interrupt will be serviced based on the content of the Channel Status Register.

Memory Exception

Memory Exception (MEXC) is asserted by BIU to signal that an error condition was generated during transfer. The DMA channel will stop the transfer immediately, set up the relevant bit (Source/Destination/Chaining error) in the DMA channel Status Register, and assert the -EOP. The -EOP will be deasserted when the memory exception status bit is cleared by the program. For quad-word transfer (intended for burst mode), the DMA will finish all four read or write cycles before stopping and setting up the relevant bit in the Status Register.

This flag affects only the number of windows and the impl/ver fields in the PSR. The MB86932's cache and TLB can be enabled even if the flag is set to 1. Since the cache and TLB are disabled after Reset, the MB86932 will emulate the MB86930 correctly as long as the program being executed does not set the bits that enable the cache and the TLB.

Context Compare Register



Address: 0x000FF20 to (DASI=0x01)

Figure B6-1. Context Compare Register

- Bit 31: mask comparison of DASI[7:0] with dasid2[7:0]
- Bit 30: Context_DA_Mask_2(CDAM2)
- Bits 29-24: Context_DA_Description_2 (CDAD2)
- Bit 23: mask comparison of DASI[7:0] with dasid1[7:0]
- Bit 22: Context_DA_Mask_1 (CDAM1)
- Bits 21-16: Context_DA_Description_1 (CDAD1)
- Bit 15: mask comparison of ISUPER with isuperd2
- Bit 14: Context_IA_Mask_2 (CIAM2)
- Bits 13-8: Context_IA_Description_2 (CIAD2)
- Bit 7: mask comparison of ISUPER with isuperd1
- Bit 6: Context_IA_Mask_1 (CIAM1)
- Bits 5-0: Context_IA_Description_1 (CIAD1)

Context IA/DA Mask Flags:

Each of the four Context Description fields is associated with a Mask field. If a Mask field is set to 1, its associated Context field is *not* compared. These flags are to be set when the ICE/Monitor logic does not recognize the concept of "context"; in such cases, a break will be invoked whenever the IA/DA matches the break-point address, regardless of context. Within the break routine, a check can be made to see if the context is the one for which the break was defined. This masking condition governs all the cases listed below.

6.2.2 Logic of Context Comparison

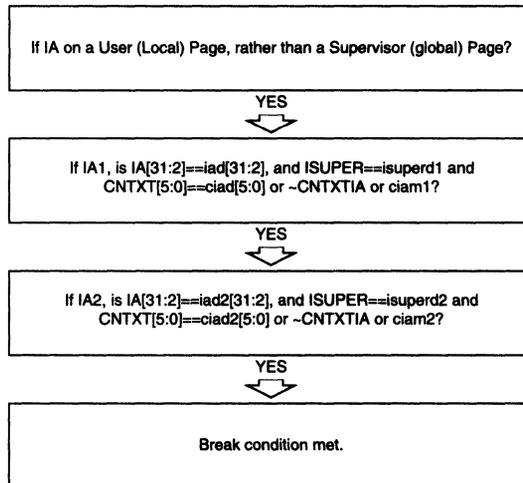


Figure B6-2. On an Instruction Address (IA) Match Break:

If no page in the TLB matches (that is, the IA's page is not found in there), the value of the CNTXTIA signal is undefined. This may cause an incorrect IA break request, but this will do no harm; the memory exception trap invoked by the TLB will cancel that incorrect break request.

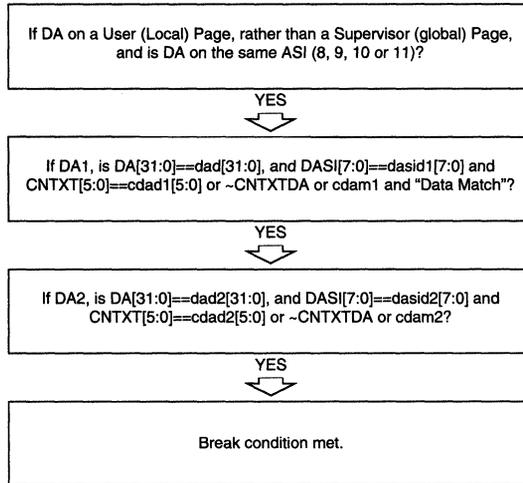


Figure B6-3. On a Data Address (DA) and Data Data (DD) Match Break:

If no page in the TLB matches (that is, the DA's page is not found in there), the value of the CNTXTDA signal is undefined. This may cause an incorrect DA break request, but this will do no harm; the TLB-miss or instruction-access exception trap invoked by the TLB will cancel that incorrect break request.

CHAPTER B7



MB86932 External Interface

7.1 SIGNAL DESCRIPTIONS†

Symbol	Type	Description
-RESET	I	SYSTEM RESET: Asserting reset for at least 4 processor cycles after the clock has stabilized, causes the MB86932 to be initialized.
XTAL1, (CLK_IN) XTAL2	I/O O G(Q) I (Q)	EXTERNAL OSCILLATOR: The crystal inputs determine execution rate and timing of the MB86932 processor. Connecting a crystal to these pins forms a complete crystal oscillator circuit. The crystal oscillator frequency is the same as the processor operating frequency. When driving the processor with an external clock, XTAL2 pin should be left floating.
CLKOUT1	O G(Q) I (Q)	CLOCK OUTPUT 1: This is an output signal against which MB86932 bus transactions can be referenced. The CLKOUT1 frequency is the same as the frequency applied to XTAL1 and is the same as the processor operating frequency. CLKOUT1 is in phase with CLK_IN.
CLKOUT2	O G(Q) I (Q)	CLOCK OUTPUT 2: This is an output signal against which MB86932 bus transactions can be referenced. The CLKOUT2 frequency is the same as the frequency applied to XTAL1 and is the same as the processor operating frequency. CLKOUT2 is out of phase with CLK_IN.
-LOCK	O S(L) G(Z) I (1)	BUS LOCK: This is a control signal asserted by the processor to indicate to the system that the current bus transaction requires more than one transfer on the bus. The Atomic Load Store instruction for example requires contiguous bus transactions which cause the assertion of the bus lock signal. The bus may not be granted to another bus owner as long as -LOCK is active. -LOCK is asserted with the assertion of -AS and remains active until -READY is asserted at the end of the locked transaction.

7.1 SIGNAL DESCRIPTIONS[†] (Continued)

Symbol	Type	Description
-BREQ	I S(L)	BUS REQUEST: Asserted by another device on the bus to indicate that it wants ownership of the bus. The request must be answered with a bus grant (-BGRNT) from the MB86932 before the device can proceed by driving the bus. Once the bus has been granted, the device has ownership of the bus until it de-asserts -BREQ. The user should ensure that devices on the bus cannot monopolize the bus to the exclusion of the CPU. Inputs to -BREQ while -RESET is active are valid and cause Bus Grant to be asserted.
-BGRNT	O S(L) G(O) I(Q)	BUS GRANT: Asserted by the CPU in response to a request from a device wanting ownership of the bus. The CPU grants the bus to other devices only after all transfers for the current transaction are completed. All bus drivers are three-stated with the assertion of the bus grant signal.
-ERROR	O S(L) G(Q) I(Q)	ERROR SIGNAL: Asserted by the CPU to indicate that it has halted in an error state as a result of encountering a synchronous trap while traps are disabled. In this situation the CPU saves the PC and nPC registers, sets the tt value in the TBR, enters into an error state and asserts the -ERROR signal. The system can monitor the -ERROR pin and initiate a reset under the error condition. This pin is high on reset.
-MEXC	I S(L)	MEMORY EXCEPTION: Asserted by the memory system to indicate a memory error on either a data or instruction access. Assertion of this signal initiates either a data or instruction access exception trap in the IU. The current bus access is invalidated by asserting the -MEXC in the same cycle as the -READY signal. The IU ignores the contents of the data bus in cycles where -MEXC is asserted.
IRL <3:0>	I A(L)	INTERRUPT REQUEST BUS: The value on these pins defines the external interrupt level. IRL<3:0>=1111 forces a non-maskable interrupt. IRL value of 0000 indicates no pending interrupts. All other values indicate maskable interrupts as enabled in the PIL field of the processor status register (PSR). Interrupts should be latched and prioritized by external logic and should be held pending until acknowledged by the processor. An interrupt controller is available on the MB86940.
-TIMER_OVF	O S(L) G(Q) I(Q)	TIMER UNDERFLOW: Asserted by the processor to indicate that the internal 16-bit timer has underflowed. This signal can be used to initiate a DRAM refresh cycle or a one cycle periodic waveform. On reset, the timer is turned off and -TIMER_OVF is high.
-SAME_PAGE	O S(L) G(1) I(1)	SAME-PAGE DETECT: The -SAME_PAGE is used to take advantage of fast consecutive accesses within Fast Page Mode DRAM page boundaries. This signal is an output asserted by the processor when the current address is within the same page as the previous memory access. The -SAME_PAGE signal is asserted with -AS and remains active for one processor cycle. -SAME_PAGE is never asserted in the first transaction following a transaction by another device on the bus. The page size is specified by writing the SAME-PAGE MASK register.
-CS0, -CS1, -CS2, -CS3, -CS4, -CS5	O S(L) G(1) I(1)	CHIP SELECTS: These outputs are asserted when the value on the address bus matches the address range in one of the corresponding ADDRESS RANGE registers. The signals are used to decode the current address into one of six address ranges. Address ranges should not overlap. Each address range has a corresponding wait specifier which is used to automatically assert the -READY signal after a user defined number of processor clock cycles. This allows a variety of memory and I/O devices with different access times to be connected to the MB86933 without the need for additional logic.

7.1 SIGNAL DESCRIPTIONS[†] (Continued)

Symbol	Type	Description																														
ADR <31:2>	O S(L) G(Z) I(1)	ADDRESS BUS: The 30-bit ADDRESS BUS (A31-A2) is an output which identifies the data or instruction address of a 32-bit word. Reads are always one word in size while byte, half-word, or word transaction sizes for writes is identified by separate byte-enable signals (–BE0-3). The address bus is valid for the duration of the bus transaction.																														
ASI <7:0>	O S(L) G(Z) I(1)	<p>ADDRESS SPACE IDENTIFIERS: The ADDRESS SPACE IDENTIFIERS are outputs which indicate to which of 256 available spaces the current ADDRESS BUS value corresponds. The ASI values are defined as follows:</p> <table border="1" data-bbox="565 494 1027 916"> <thead> <tr> <th>ASI <7:0></th> <th>ADDRESS SPACE</th> </tr> </thead> <tbody> <tr><td>0x1</td><td>Control Register</td></tr> <tr><td>0x2</td><td>Instruction Cache Lock</td></tr> <tr><td>0x3</td><td>Data Cache Lock</td></tr> <tr><td>0x4 - 0x7</td><td>Application Definable</td></tr> <tr><td>0x8</td><td>User Instruction Space</td></tr> <tr><td>0x9</td><td>Supervisor Instruction Space</td></tr> <tr><td>0xA</td><td>User Data Space</td></tr> <tr><td>0xB</td><td>Supervisor Data Space</td></tr> <tr><td>0xC</td><td>Instruction Cache Tag RAM</td></tr> <tr><td>0xD</td><td>Instruction Cache Data RAM</td></tr> <tr><td>0xE</td><td>Data Cache Tag RAM</td></tr> <tr><td>0xF</td><td>Data Cache Data RAM</td></tr> <tr><td>0x10 - 0xFD</td><td>Application Definable</td></tr> <tr><td>0xFE - 0xFF</td><td>Reserved for Debug Hardware</td></tr> </tbody> </table> <p>The ASI values specified as “application definable” can be used by supervisor mode instructions such as Load Alternate and Store Alternate. The ASI value is available in the same cycle in which the corresponding address value is asserted on the address bus. The ASI pins are valid for the duration of the bus transaction. ASI values 0x8, 0x9, 0xA, and 0xB are cacheable.</p>	ASI <7:0>	ADDRESS SPACE	0x1	Control Register	0x2	Instruction Cache Lock	0x3	Data Cache Lock	0x4 - 0x7	Application Definable	0x8	User Instruction Space	0x9	Supervisor Instruction Space	0xA	User Data Space	0xB	Supervisor Data Space	0xC	Instruction Cache Tag RAM	0xD	Instruction Cache Data RAM	0xE	Data Cache Tag RAM	0xF	Data Cache Data RAM	0x10 - 0xFD	Application Definable	0xFE - 0xFF	Reserved for Debug Hardware
ASI <7:0>	ADDRESS SPACE																															
0x1	Control Register																															
0x2	Instruction Cache Lock																															
0x3	Data Cache Lock																															
0x4 - 0x7	Application Definable																															
0x8	User Instruction Space																															
0x9	Supervisor Instruction Space																															
0xA	User Data Space																															
0xB	Supervisor Data Space																															
0xC	Instruction Cache Tag RAM																															
0xD	Instruction Cache Data RAM																															
0xE	Data Cache Tag RAM																															
0xF	Data Cache Data RAM																															
0x10 - 0xFD	Application Definable																															
0xFE - 0xFF	Reserved for Debug Hardware																															
–BMODE8	I S(L)	<p>8-BIT BOOT MODE: This signal is sampled during reset and causes read accesses, memory mapped to –CS0, to assume 8-bit ROM memory. The MB86932 generates four sequential fetches to assemble a complete instruction or data word before continuing. Bytes are fetched in sequence (0,1,2,3) as encoded by –BE[2] and –BE[3] (00, 01, 02, 03). Writes to –CS0 are unaffected by boot mode selection and if left unconnected, a weak pull-up on this pin (and –BMODE16 pin) causes the processor to default to 32-bit mode.</p> <p>Note: BMODE8 and BMODE16 should not be asserted at the same time.</p>																														
–BMODE16	I S(L)	<p>16-BIT BOOT MODE: This signal is sampled during reset and causes read accesses, memory mapped to –CS0, to assume 16-bit ROM memory. The MB86932 generates two sequential fetches to assemble a complete instruction or data word before continuing. Half words are fetched in sequence (0,1) as encoded by –BE[2]. Writes to –CS0 are unaffected by boot mode selection. If left unconnected, a weak pull-up on this pin (and –BMODE8 pin) causes the processor to default to 32-bit mode.</p> <p>Note: BMODE8 and BMODE16 should not be asserted at the same time.</p>																														

7.1 SIGNAL DESCRIPTIONS† (Continued)

Symbol	Type	Description																																																																								
-BE3-0	O S(L) G(Z) I(O)	<p>BYTE ENABLES (O): These pins indicate whether the current store transaction is a byte, half-word or word transaction. -BE0-3 signals are available in the same cycle in which the corresponding address value is asserted on the address bus and is valid for the duration of the bus transaction. This bus should be used only to qualify store transactions. For load transactions all sub-word requests are read (and replaced in the cache) as words and then the appropriate byte or half-word is extracted by the integer unit.</p> <p>Possible values for -BE3-0 are as follows:</p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td></td> <td style="text-align: center;">31</td> <td style="text-align: center;">24</td> <td style="text-align: center;">23</td> <td style="text-align: center;">16</td> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td></td> <td colspan="3" style="text-align: center;">Byte 0</td> <td colspan="2" style="text-align: center;">Byte 1</td> <td colspan="2" style="text-align: center;">Byte 2</td> <td style="text-align: center;">Byte 3</td> </tr> <tr> <td>Byte Writes</td> <td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td> </tr> <tr> <td>Half-Word Writes</td> <td colspan="4">1</td><td colspan="4">0</td><td colspan="4">1</td> </tr> <tr> <td>Word Writes</td> <td colspan="8">0</td><td colspan="2">0</td><td colspan="2">0</td> </tr> </table> <p>BE<2:3> are also used in 8 and 16-bit ROM accesses as follows:</p> <table style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Bus Mode</th> <th>Byte</th> <th>BE<2:3></th> </tr> </thead> <tbody> <tr> <td rowspan="4">8-bit</td> <td>0</td> <td>00</td> </tr> <tr> <td>1</td> <td>01</td> </tr> <tr> <td>2</td> <td>10</td> </tr> <tr> <td>3</td> <td>11</td> </tr> <tr> <td rowspan="2">16-bit</td> <td>0 & 1</td> <td>00</td> </tr> <tr> <td>2 & 3</td> <td>10</td> </tr> </tbody> </table>		31	24	23	16	15	8	7	0		Byte 0			Byte 1		Byte 2		Byte 3	Byte Writes	1	1	1	0	1	1	0	1	1	1	Half-Word Writes	1				0				1				Word Writes	0								0		0		Bus Mode	Byte	BE<2:3>	8-bit	0	00	1	01	2	10	3	11	16-bit	0 & 1	00	2 & 3	10
	31	24	23	16	15	8	7	0																																																																		
	Byte 0			Byte 1		Byte 2		Byte 3																																																																		
Byte Writes	1	1	1	0	1	1	0	1	1	1																																																																
Half-Word Writes	1				0				1																																																																	
Word Writes	0								0		0																																																															
Bus Mode	Byte	BE<2:3>																																																																								
8-bit	0	00																																																																								
	1	01																																																																								
	2	10																																																																								
	3	11																																																																								
16-bit	0 & 1	00																																																																								
	2 & 3	10																																																																								
D <31:0>	I/O S(L) G(Z) I(Z)	<p>DATA BUS: The bus interface has 32 bidirectional data pins (D31-D0) to transfer data in thirty-two bit quantities. D(31) corresponds to the most significant bit of the least significant byte of the 32-bit word. A double word is aligned on an 8-byte boundary, a word is aligned on a 4-byte boundary, and a half-word is aligned on a 2-byte boundary. If a load or store of any of these quantities is not properly aligned, a Not Aligned Trap will occur in the processor.</p> <p>In write bus cycles, the point at which data is driven onto the bus depends on the type of the preceding cycle. If the preceding cycle was a write, data is driven in the cycle immediately following the cycle in which -READY was asserted. If the preceding cycle was a read, data is driven one cycle after the cycle in which -READY was asserted to minimize bus contention between the processor and the system.</p> <p>Pins D[7:0] are used when the 8-bit boot mode is enabled and D[15:0] are used when 16-bit mode is enabled.</p>																																																																								
-AS	O S(L) G(Z) I(1)	<p>ADDRESS STROBE: A control signal asserted by the MB86930 or other bus master to indicate the start of a new bus transaction. A bus transaction begins with the assertion of -AS and ends with the assertion of -READY. -AS remains asserted for 1 clock cycle. During cycles in which neither the processor nor another bus master is driving the bus the bus is idle, and -AS remains de-asserted.</p>																																																																								
RD/-WR	O S(L) G(Z) I(1)	<p>READ/BUS TRANSACTION: This signal specifies whether the current bus transaction is a read or a write operation. When -AS is asserted and RD/-WR is low, then the current transaction is a write. With -AS asserted and RD/-WR high, the current transaction is a read. RD/-WR remains active for the duration of the bus transaction and is de-asserted with the assertion of -READY.</p>																																																																								

7.1 SIGNAL DESCRIPTIONS† (Continued)

Symbol	Type	Description
–READY	I S(L)	READY: This is a control signal asserted by the external memory system to indicate that the current bus transaction is being completed and that it is ready to start with the next bus transaction in the following cycle. In case of a fetch from memory, the processor will strobe the value on the data bus at the rising edge of CLK_IN following the assertion of –READY. For the case of a write, the memory system will assert –READY when the appropriate access time has been met. In most cases, no additional logic is required to generate the –READY signal. On-chip circuitry can be programmed to assert –READY based on the address of the current transaction. The external system can override the internal ready generator to terminate the current bus cycle early. Up to 6 address ranges each with different transaction times can be programmed.
–DREQ0-1	A(L) I	DMA REQUEST: Indicates that an external device is requesting a DMA transfer. This signal is edge sensitive for single transfers and level sensitive for demand transfer. –DREQ0 corresponds to DMA channel 0, while –DREQ1 corresponds to DMA channel 1.
–DACK0-1	O	DMA ACKNOWLEDGE: This signal is asserted when an external device asserts –DREQ and the processor accesses the external device. –DACK1 corresponds to DMA channel 0, while –DACK1 corresponds to DMA channel 1.
–EOP0-1	I/O	END OF PROCESS: This signal is asserted by the external device when it wants to terminate a DMA transfer. Alternately, the processor drives this signal when the byte count reaches zero. –EOP0 corresponds to DMA channel 0, while –EOP1 corresponds to DMA channel 1. A pull-up holds –EOP0-1 high when it is not being driven.
–PBREQ	O	PROCESSOR BUS REQUEST: This signal is asserted by the processor to indicate to an external bus arbiter that it needs to regain control of the bus. This provides a handshake between the arbiter and the processor to allow the bus to be allocated based on demand.
–BMREQ	O	BURST MODE REQUEST: This signal is asserted by the processor to indicate to the external system that the processor's burst mode is enabled and the current transaction can be a burst. If the external system supports burst mode, it asserts –BMACK concurrently with –RDY to begin the burst mode transfer.
–BMACK	I	BURST MODE ACKNOWLEDGE: This signal is asserted by the system to indicate that it can support burst mode for the address currently on the bus. The system asserts –BMACK in response to the processor asserting –BMREQ.
CLK_ECB	I	EXTERNAL CLOCK BYPASS: Tying this signal high causes the CLK_IN signal to bypass the Phases Lock Loop (PLL). This signal is used for testing of the chip.
PARITY<3:0>	I/O	PARITY: When enabled, this signal provides even or odd parity checking for data bus accesses.
EMU_SD <3:0>	I/O	EMULATOR STATUS/DATA BITS: Bi-directional pins used by a hardware emulator to control and monitor MB86930 execution. These pins should be left unconnected.
EMU_D<3:0>	I/O	EMULATOR DATA BITS: Bi-directional pins used by a hardware emulator to control and monitor MB86930 execution. These pins should be left unconnected.
EMU_BRK	I	EMULATOR BREAK REQUEST LINE: Input used by a hardware emulator to request a trap when emulation is enabled. This pin should be left unconnected.

7.1 SIGNAL DESCRIPTIONS† (Continued)

Symbol	Type	Description
-EMU_ENB	I	EMULATOR ENABLE: Tied low while the MB86930 is being reset to enable hardware emulator mode on the chip. This pin should be left unconnected.
TCK	I	TEST CLOCK: JTAG compatible test clock input.
TMS	I	TEST MODE: JTAG compatible test mode select pin. Test is enabled when -TMS is low.
TDI	I	TEST DATA IN: JTAG compatible test data input.
TDO	O	TEST DATA OUT: JTAG compatible test data output.
-TRST	I	TEST RESET: Asynchronous reset for JTAG logic. If not using JTAG, this signal must be pulled low.

†. In the following descriptions, signal names preceded by a minus sign (-) indicate an active low state. Dual function pins have two names separated by a slash (/).

- Notes:**
- I = Input Only Pin
 - O = Output Only Pin
 - I/O = Either Input or Output Pin
 - = Pins "must be" connected as described
 - A(L) = Asynchronous: Inputs may be asynchronous to CLKOUT.
 - S(L) = Synchronous: Inputs must meet setup and hold times relative to CLK_IN. Outputs are Synchronous to CLK_IN
 - G(...) = While the bus is granted to another bus master (-BGRNT=asserted), the pin is
 - G(1) is driven to V_{CC}
 - G(0) is driven to V_{SS}
 - G(Z) floats
 - G(Q) is a valid output
 - I(...) = While the bus is between bus cycles (or being reset) and is not granted to another bus master, the pin is
 - I(1) is driven to V_{CC}
 - I(0) is driven to V_{SS}
 - I(Z) floats
 - I(Q) is a valid output

CHAPTER B8

MB86932 JTAG

8.1 MB86932 JTAG Pin List

The MB 86932 JTAG cells are arranged in a shift register configuration (see Figure B8-8). When shifting in a JTAG pattern through TDI, the LSB should correspond to the JTAG cell value for `-TIMER_OVF` pin whereas, the MSB of the pattern should correspond to the `CLK_ENB` pin's JTAG cell. As far as JTAG output through TDO is concerned, the first bit out corresponds to `-TIMER_OVF` JTAG cell value and the last output bit corresponds to the `CLK_ENB` JTAG cell value. Table B8-1 lists the order of all of the JTAG cells.

Table B8-1: JTAG Pin Order

Order	JTAG Cell	JTAG Cell Type	Function
1	<code>-TIMER_OVF</code>	output	Timer Overflow pin
2	<code>XTAL1</code>	input	Crystal input
3	<code>-TEST</code>	input	Factory test pin
4	<code>PARITY<2></code>	in/out	
5	<code>PARITY<3></code>	in/out	
6	<code>EMU_BRK</code>	input	Emulator break input

Table B8-1: JTAG Pin Order (Continued)

Order	JTAG Cell	JTAG Cell Type	Function
7	icediojo [†]	output	Bidirectional control for EMU_D/EMU_SD buses icediojo = 1: EMU_D and EMU_SD buses are input icediojo = 0: EMU_D and EMU_SD buses are output
8	EMU_SD_i<3>	input	Input bit 3 of EMU_SD<3:0> bus
9	EMU_SD_o<3>	output	Output bit 3 of EMU_SD<3:0> bus
:	:	:	:
14	EMU_SD_i<0>	input	Input bit 0 of EMU_SD<3:0> bus
15	EMU_SD_o<0>	output	Output bit 0 of EMU_SD<3:0> bus
13	EMU_D_i<3>	input	Input bit 3 of EMU_D<3:0> bus
14	EMU_D_o<3>	output	Output bit 3 of EMU_D<3:0> bus
:	:	:	:
22	EMU_D_i<0>	input	Input bit 0 of EMU_D<3:0> bus
23	EMU_D_o<0>	output	Output bit 0 of EMU_D<3:0> bus
24	iceenblio [†]	output	Bidirectional control signal for –EMU_ENB pin iceenblio = 1: –EMU_ENB pin is an input iceenblio = 0: –EMU_ENB pin is an output
25	–EMU_EN_i	input	Input bit of –EMU_ENB pin
26	–EMU_EN_o	output	Output bit of –EMU_ENB pin
27	dbusiojo [†]	output	Bidirectional control signal for D<31:0>, Parity <3:0> dbusiojo = 1: D<31:0>, Parity <3:0> are inputs dbusiojo = 0: D<31:0>, Parity <3:0> are outputs
28	D_i<31>	input	Input bit 31 of D<31:0> bus
29	D_o<31>	output	Output bit 31 of D<31:0> bus
:	:	:	:
54	D_i<18>	input	Input bit 18 of <31:0> bus
55	D_o<18>	output	Output bit 18 of D<31:0> bus
56	–BMODE16	input	
57	D_i<17>	input	Input bit 17 of D<31:0> bus
58	D_o<17>	output	Output bit 17 of D<31:0> bus
59	D_i<16>	input	Input bit 16 of D<31:0> bus
60	D_o<16>	output	Output bit 16 of D<31:0> bus
61	D_i<15>	input	Input bit 15 of D<31:0> bus
62	D_o<15>	output	Output bit 15 of D<31:0> bus
63	–BMODE8	input	
64	D_i<14>	input	Input bit 14 of D<31:0> bus

Table B8-1: JTAG Pin Order (Continued)

Order	JTAG Cell	JTAG Cell Type	Function
65	D_o<14>	output	Output bit 14 of D<31:0> bus
:	:	:	:
80	D_i<6>	input	Input bit 6 of <31:0> bus
81	D_o<6>	output	Output bit 6 of D<31:0> bus
82	-BMREQ	output	
83	D_i<5>	input	Input bit 5 of D<31:0> bus
84	D_o<5>	output	Output bit 5 of D<31:0> bus
:	:	:	:
93	D_i<0>	input	Input bit 0 of <31:0> bus
94	D_o<0>	output	Output bit 0 of D<31:0> bus
95	-RESET	input	Chip reset pin
96	-BREQ	input	Bus request input
97	-MEXC	input	Memory exception input
98	-READY	input	External memory transaction complete signal
99	tstatejo [†]	output	Three-state control signal for ADR, ASI, -BE, -AS, RD/WR and -LOCK If tstatejo = 1: signals are three-stated. If tstatejo = 0: signals are outputs.
100	-BGRNT	output	Bus grant output signal
101	-ERROR	output	Error output signal
102	-LOCK	output	Bus lock output signal
103	-BMACK	input	
104	-RD/WR	output	Memory Read/Write output signal
105	-AS	output	Start of memory transaction output signal
106	-PBREQ	output	
107	-CS<0>	output	
108	-DREQ0	input	
109	-CS<1>	output	
110	-CS<2>	output	
111	-CS<3>	output	
112	-CS<4>	output	
113	-DREQ1	input	
114	-CS<5>	output	
115	-SAMEPAGE	output	

Table B8-1: JTAG Pin Order (Continued)

Order	JTAG Cell	JTAG Cell Type	Function
116	–DACK0	output	
117	BE<3>	output	
118	BE<2>	output	
119	BE<1>	output	
120	BE<0>	output	
121	ASI<0>	output	
122	ASI<1>	output	
123	ASI<2>	output	
124	ASI<3>	output	
125	–DACK1	output	
126	ASI<4>	output	
127	ASI<5>	output	
128	ASI<6>	output	
129	ASI<7>	output	
130	ADR<2>	output	
131	ADR<3>	output	
132	ADR<4>	output	
133	ADR<5>	output	
134	eopio0	output	Bidirectional control for –EOP0 pin eopio0 = 1: –EOP0 is input eopio0 = 0: –EOP0 is output
135	ADR<6>	output	
136	ADR<7>	output	
137	–EOP0_i	input	
138	–EOP0_o	output	
139	ADR<8>	output	
140	ADR<9>	output	
141	eopio1	output	Bidirectional control for –EOP1 pin eopio1 = 1: –EOP1 is input eopio1 = 0: –EOP1 is output
142	ADR<10>	output	
143	–EOP1_i	input	
144	–EOP1_o	output	
145	ADR<11>	output	
:	:	:	

Table B8-1: JTAG Pin Order (Continued)

Order	JTAG Cell	JTAG Cell Type	Function
150	ADR<16>	output	
151	PARITY_i<0>	input	
152	PARITY_o<0>	output	
153	ADR<17>	output	
154	ADR<18>	output	
155	ADR<19>	output	
156	ADR<20>	output	
157	PARITY_i<1>	input	
158	PARITY_o<1>	output	
159	ADR<21>	output	
:	:	:	
169	ADR<31>	output	
170	IRL<3>	input	
171	IRL<2>	input	
172	IRL<1>	input	
173	IRL<0>	input	
174	CLK_ENB	input	

†. These are internal I/O control signals. Therefore, there are no corresponding external pins.

1. The following pins are not three-stable: –SAME_PAGE, –CS<5:0>, –BGRNT, TIMER_OVF, –ERROR.

2. The following pins have no corresponding JTAG cells: CLKOUT1, CLKOUT2, XTAL2, –TRST, TCK, TMS, TDI, TDO.

Section 4



MB86933

Chapter C2: Programmer's Model

2.1 Program Modes	C2-1
2.2 Memory Organization	C2-2
2.3 Registers	C2-4
2.3.1 Register Windows	C2-4
2.3.2 Special Uses of the r Registers	C2-5
2.3.3 SPARC-Defined Special-Purpose Registers	C2-5
2.3.4 Memory-Mapped Control Registers	C2-8
2.4 Data Types	C2-8
2.5 Instructions	C2-9
2.6 Interrupts and Traps	C2-9

Chapter C3: Internal Architecture

3.1 Integer Unit	C3-2
3.1.1 I Block	C3-3
3.1.2 A Block	C3-8
3.1.3 E Block	C3-10
3.1.4 Programmer-Visible State and Processor State	C3-15
3.2 Bus Interface Unit	C3-16
3.2.1 Exception Handling	C3-16
3.2.2 Effect on the Pipeline	C3-16

Chapter C4: External Interface

4.1 Signals	C4-1
4.1.1 Processor Control and Status	C4-3
4.1.2 Memory Interface	C4-4
4.1.3 Bus Arbitration	C4-7
4.1.4 Peripheral Functions	C4-7
4.1.5 Test and Boundary-Scan	C4-7
4.2 Bus Operation	C4-8
4.2.1 Exception Handling	C4-8
4.2.2 Bus Cycles	C4-9

4.3 System Support Functions	C4-15
4.3.1 System-Configuration Registers	C4-15
4.3.2 Same-Page Detection	C4-18
4.3.3 Programmable Timer.....	C4-18
4.4 ROM Interface.....	C4-19
4.4.1 Purpose.....	C4-19
4.4.2 Features	C4-19
4.4.3 Bus Configuration on Reset	C4-20
4.4.4 System Interface	C4-20
4.4.5 PROM Address Space.....	C4-21
4.4.6 Load/Stores.....	C4-21
4.4.7 Memory Exception.....	C4-22
4.4.8 Bus Request	C4-22
4.4.9 Timing.....	C4-23
4.4.10 Store in 8/16 Bit.....	C4-24

Chapter C5: Programming Considerations

5.1 MB86933 Programming Information.....	C5-1
---	-------------

Chapter C6: System Design Considerations

6.1 Interfacing SRAM	C6-1
6.2 Interfacing Page-Mode DRAM	C6-3
6.3 In-Circuit Emulation	C6-5

Chapter C7: Instruction Set

7.1 MB86933 Instruction Set	C7-1
--	-------------

Chapter C8: MB86933 JTAG

8.1 MB86933 JTAG Information	C8-1
---	-------------



Overview of the MB86933

The MB86933 is functionally and architecturally similar to the MB86930 SPAR-Clite RISC processor. The MB86933 has the same integer unit as the MB86930, supports the same instruction set as the MB86930, and is system bus compatible with the MB86930.

Several MB86930 features and signals are not available on the MB86933, however, to reduce processor cost and package size. The MB86933 has no caches, no write buffer, no pre-fetch buffer, and has six register windows rather than eight. It has twenty-six Address Bus signals (ADR<27:2>) rather than thirty, has four Address Space Identifier signals (ASI<3:0>) rather than eight, and has no emulator-support signals. The MB86932 can be used for MB86933 in-circuit emulation, so MB86933 emulator-support signals are not necessary.

The MB86933 does support 8- and 16-bit ROMs as well as 32-bit ROMs - a feature not available on the MB86930. The processor reads two external signals, -BMODE8 and -BMODE16, during reset to identify the ROM size. This allows use of the smaller ROMs to reduce board space and component cost.

1.1 Organization and Content

This section is organized in the same way as section 1 of this manual which describes the MB86930 processor. In general, this section contains descriptions of the MB86933 processor that differ from the MB86930 processor. Descriptions that are the same for both processors are generally not repeated in this section, and the

reader is referred to the main section of the manual for these identical descriptions.

These MB86933 differences with respect to the MB86930 processor are summarized as follows:

- No instruction cache or data cache
- No write buffer or prefetch buffer
- Six register windows rather than eight
- ADR<31:28> not used
- ASI<7:4> not used
- EMU_SD<3:0> not used
- EMU_D<3:0> not used
- EMU_BRK not used
- -EMU_ENB not used
- No in-circuit emulation support
- -BMODE8 and -BMODE16 inputs added to support 8- and 16-bit ROMs, as well as 32-bit ROMs.

1.2 General Description

The MB86933 is a high-performance processor that is suitable for use in embedded control applications such as printers, scanners, robotic machinery, telecom switches and monitors, and I/O subsystems. It operates at clock speeds up to 20 MHz, executes SPARC instructions at a maximum rate of 18 MIPs, and is available in a 160-pin QFP package.

The processor consists of a Harvard (Aiken) architecture Integer Unit (IU) core and a Bus Interface Unit (BIU). These units are connected internally with separate instruction and data buses, and to external memory and I/O with separate 26-bit address and 32-bit data buses.

A register file in the IU is accessed through 6 register windows. An integer multiply unit (MU) within the IU speeds applications that require integer multiplication. The processor uses software to emulate floating-point instructions. The data path and other arrayed blocks are full-custom designs to optimize die area and speed. Random control blocks are standard-cell designs. All circuits are fully static.

The MB86933 provides a mechanism for code and data protection, but is optimized for embedded applications that do not require virtual-to-physical address translation. The MB86933 processor can be designed into a virtual-memory sys-

tem, however, by using external memory management logic for address translation.

1.3 Special Features

The following MB86933 features make the processor an ideal choice for a wide variety of low cost, high-performance embedded systems:

- **Fast Instruction Execution:** The instruction set is streamlined and hardwired for fast execution, with most instructions executing in a single cycle. At 20 MHz the MB86933 executes instructions at a peak rate of 20 MIPs and at a sustained rate of 18 MIPs. The Integer Unit (IU) features a 5-stage pipeline that has been designed to handle data interlocks, and an optimized branch handler for efficient control transfers.
- **Large Register Set:** An internal register file, consisting of eight global registers and 96 registers organized into six overlapping windows, speeds interrupt response time and context switches. The register file windows minimize accesses to memory during procedure linkages, and facilitate passing of parameters and assignment of variables.
- **System Support Functions:** Glue logic between the MB86933 and the system is minimized by programmable chip selects, programmable wait-state circuitry, and support for connection to fast page-mode DRAM. Multiple bus masters are supported through a simple handshake protocol.
- **Clock Generator:** A crystal can be connected directly to the on-chip oscillator, or an external clock source can be used. A phase-locked loop minimizes the skew between on- and off-chip clocks.
- **Enhanced Instruction Set:** The MB86933 incorporates a fast integer multiply instruction that executes in a fast 5, 3 or 2 cycles for 32-bit, 16-bit and 8-bit operands. An integer divide-step instruction cuts divide times by a factor of 5 to 10 over previous SPARC implementations. A scan instruction supports a single-cycle search for the most significant non-sign bit in a word.
- **Fully Static Circuit Design:** Its static design gives the MB86933 superior noise immunity. Future members of the SPARC^{lite} family will support a low-power mode in which the processor clock can be slowed or stopped for arbitrary periods of time to reduce operating current.
- **ROM Size Option Support:** Two external signals allow the processor to identify whether 8-, 16-, or 32-bit ROMs are in use. This feature allows use of smaller ROMs for a reduction in cost and in board space.

1.4 Programmer's Model

This section briefly introduces those aspects of the MB86933 processor architecture that are visible to software: the user and supervisor modes of program execu-

tion, the organization of the address space, the register set, the supported data types, the instruction set, and interrupts and traps. Each of these topics are discussed in more detail in following chapters.

1.4.1 Program Modes

The MB86933 architecture supports protection in multitasking environments by providing two mutually exclusive modes of program execution, *user mode* and *supervisor mode*. Certain instructions are privileged, and can only be executed when the processor is in supervisor mode. Any attempt to execute a privileged instruction in user mode causes a trap.

Typically, application programs run in user mode, while operating systems run in supervisor mode. Following reset, the processor is in supervisor mode. To enter user mode, software must clear a bit in the Processor State Register. The processor enters supervisor mode from user mode only when a hardware reset, an interrupt, or a trap occurs.

1.4.2 Memory Organization

The processor can directly address up to 4 Gigabytes of memory, organized into 16 address spaces of 256 Megabytes each. Every external access involves an 4-bit Address Space Identifier (ASI), as well as a 26-bit word address. The ASI selects one of the address spaces, and the 26-bit address selects a 32-bit word within that space.

Four of the address spaces are defined in the SPARC architecture: the User Instruction, Supervisor Instruction, User Data, and Supervisor Data spaces. The other address spaces are application-defined or reserved. The application-defined address spaces can be used for either data memory or for I/O. All I/O is memory-mapped.

The organization of the entire addressable range is illustrated in Figure C1-1.

Loads and stores are the only instructions that cause external accesses. Versions of these instructions exist for transferring bytes, half-words, words and double words between external memory (or I/O) and processor registers. The user instruction and data spaces are accessible in both user and supervisor modes. The remaining address spaces are accessible only in supervisor mode.

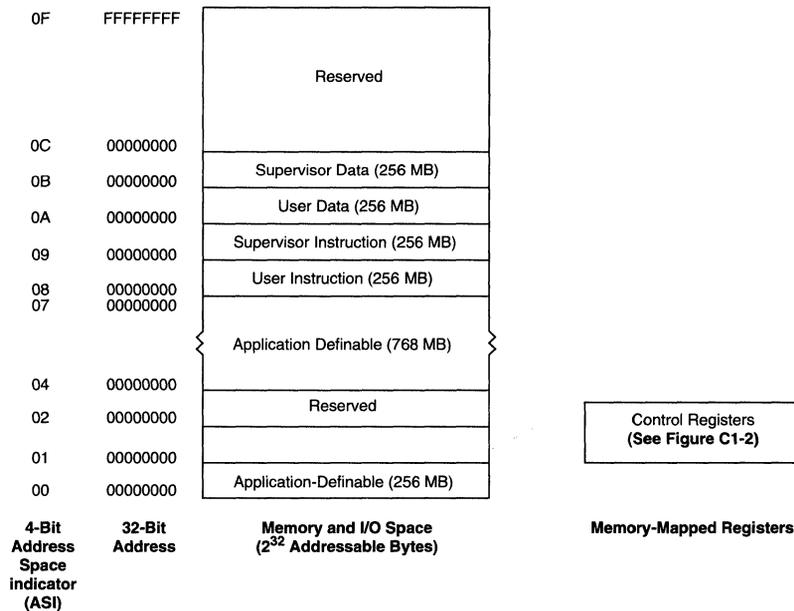


Figure C1-1. Address Space Organization

The MB86933 processor does not contain memory-management hardware. Virtual-addresses can be translated by software, or by an external memory-management unit.

Note that the MB86933 has no caches, no write buffer, no pre-fetch buffer, and has six register windows rather than eight. It has twenty-six Address Bus signals (ADR<27:2>) rather than thirty, four Address Space Identifier signals (ASI<3:0>) rather than eight, no emulator-support signals, and no memory management unit. These and other differences between the MB86933 and other SPARClite processors should be considered when porting code to the MB86933 from another SPARClite processor, and when porting code from the MB86933 to another SPARClite processor. Documentation for other SPARClite should be referenced to identify differences with the MB86933 that may affect ported code.

1.4.3 Registers

All registers are 32 bits wide. There are *general-purpose registers*, whose contents have no pre-assigned meaning, and *special-purpose registers* that contain control and status information or special data values. Some of the special-purpose registers are defined in the SPARC architecture; the rest are MB86933- specific regis-

ters. The non-SPARC special-purpose registers are memory-mapped. The general-purpose registers and the special-purpose Y Register are the only registers that can be accessed in user mode. The register set is illustrated in Figure C1-2.

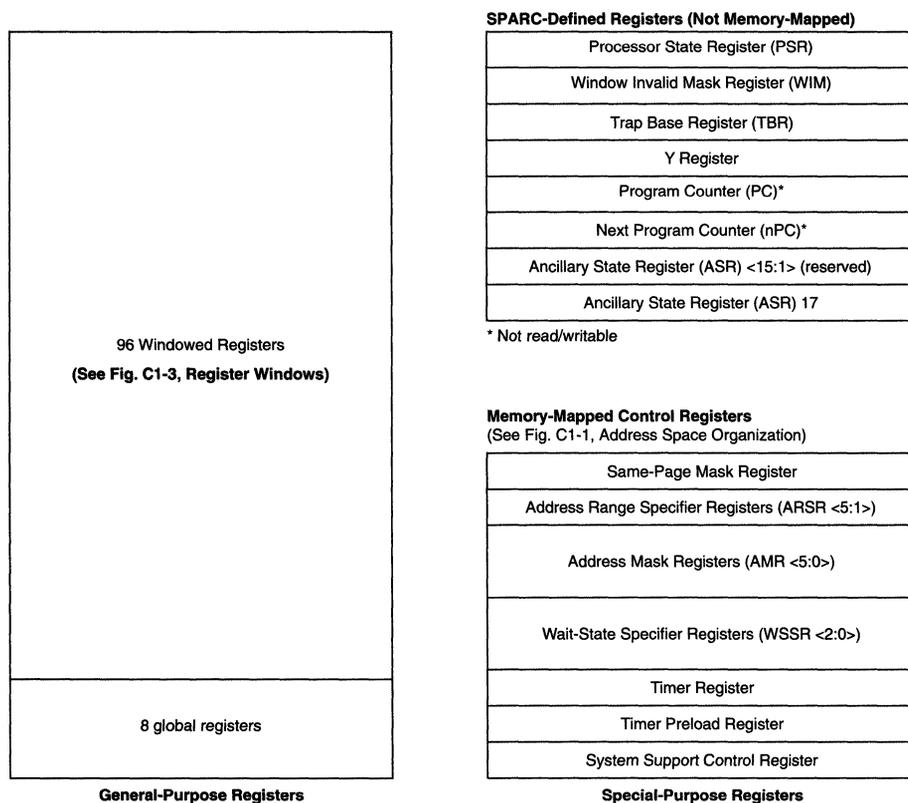


Figure C1-2. Register Set

General-Purpose Registers

The MB86933 contains 104 general-purpose registers; 8 of these are *global registers*; the other 96 registers are divided into 6 overlapping blocks, or *windows*. Each window contains 24 registers. Of these, 8 are *local* to the window, 8 are “*out*” reg-

isters shared with the adjacent window below, and 8 are “in” registers shared with the adjacent window above. This organization is illustrated in Figure C1-3.

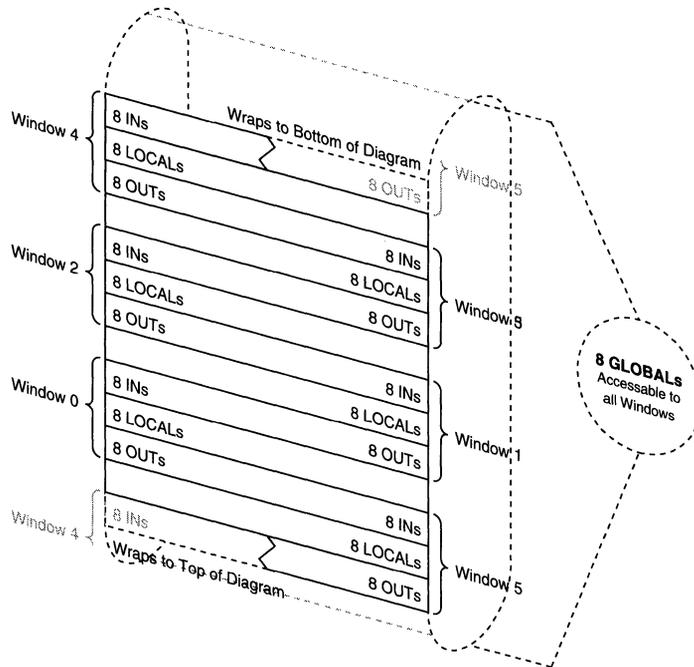


Figure C1-3. Register Windows

At any given time, 32 general-purpose registers can be accessed directly: the 8 global registers, and the 24 registers of the currently active window. The value in the Current Window Pointer (CWP) field of the Processor State Register (PSR) determines which window is active.

The overlap between adjacent windows makes it easy to pass parameters to a subroutine. Values to be passed are written to the “out” registers of the current window, which are the same as the “in” registers of the adjacent window. A SAVE instruction can then be used to decrement the Current Window Pointer, making the parameter values available to the subroutine without moving any data. A RESTORE instruction can be used to increment the CWP upon return from the subroutine. In effect, the general-purpose registers cache the top portion of the run-time stack.

The window overlap also speeds interrupt handling because interrupts automatically decrement the CWP, giving the interrupt routing its own window. The SPARC architecture requires a free window to be available to handle these traps.

Special-Purpose Registers

The special-purpose registers include the control and status registers defined by the SPARC architecture, and a collection of memory-mapped registers that control peripheral functions.

Special instructions exist for reading and writing each of the SPARC control and status registers except the Program Counter and the Next Program Counter. The Y Register can be read and written in user mode; the instructions that access the other SPARC-defined registers are privileged.

The memory-mapped registers can be read and written with the alternate-space load and alternate-space store instructions, which are also privileged.

The SPARC-defined registers, shown in Figure C1-2, are as follows:

- Processor State Register (PSR)—The primary processor control and status register. It contains *mode* fields that are set by the operating system to configure the processor, and *status* fields that are set by the processor to indicate the effects of instruction execution.
- Window Invalid Mask Register (WIM)—Used by software to detect the occurrence of register file underflows and overflows. It contains one mask bit for each register window. If an operation that normally increments or decrements the Current Window Pointer would cause the CWP to point to a window whose corresponding WIM bit equals 1, a trap occurs.
- Trap Base Register (TBR)—Contains three fields used by the processor to generate the address of the service routine when an interrupt or trap occurs.
- Y Register—Used in stepwise multiplication and division routines based on the MULSc and DIVSc instructions. Also used for integer multiply operations.
- Program Counter (PC)—Contains the word address of the instruction currently being executed by the Integer Unit. The PC cannot be directly read or written.
- Next Program Counter (nPC)—Contains the word address of the next instruction to be executed, assuming that no trap occurs. The nPC cannot be directly read or written.
- Ancillary State Registers (ASR[31:1])—The SPARC definition includes 31 Ancillary State Registers, 15 of which (ASR[15:1]) are reserved for future use. The remaining ASR's can be defined and used in any way by SPARC implementations. SPARClite defines the following ASR:

ASR17— Used to enable and disable single-vector trapping. (When this feature is enabled, all traps vector to a single location.) Single vector trapping provides a small memory alternative to the standard 1K word trap table.

The memory-mapped MB86933-specific registers, shown in Figure C1-2, are as follows:

- Same-Page Mask Register—Controls the operation of the same-page detection logic by specifying which bits of the current ASI and address are to be compared with those of the previous ASI and address.
- Address Range Specifier Registers (ARSR[5:1])—Control the assertion of the Chip-Select outputs (–CS[5:1]). –CS_n is asserted when the value on the address bus falls in the address range specified by ARSR_n. –CS₀ is asserted during accesses to the lowest address range in Supervisor Instruction Space.
- Address Mask Registers (AMR[5:0])—AMR_n controls the comparison of the current address with ARSR_n by specifying which bits are to be compared and which are “don’t cares.”
- Wait-State Specifier Registers (WSSR[2:0])—Determine for each address range the number of clock cycles between assertion of an address in that range on the address bus, and assertion of –READY signal by the processor. This makes it possible for memory and I/O devices with different access times to be connected to the processor without additional logic.
- Timer Register—Contains the current timer count.
- Timer Pre-Load Register—Contains the value that is loaded into the timer when the timer overflows.
- System Support Control Register—Allows selective enabling and disabling of same-page detection, chip-select, programmable wait-states, and the timer.

1.4.4 Data Types

The MB86933 supports the same data types as the MB86930 processor. Please refer to section 1.3.4 of the main section of this manual for a description of the data types.

1.4.5 Instructions

The MB86933 supports the same instructions as the MB86930 processor. Please refer to section 1.3.5 of the main section of this manual for a description of the instructions.

1.4.6 Interrupts and Traps

The MB86933 supports the same interrupts and traps as the MB86930 processor. Please refer to section 1.3.7 of the main section of this manual for a description of the interrupts and traps.

1.5 Internal Architecture

The internal architecture of the MB86933 is illustrated in Figure C1-4. The processor core consists of an Integer Unit that supports a superset of the SPARC integer instruction set. The Bus Interface Unit handles the interface between the processor and the system. A Clock Generator with built-in phase-locked loop simplifies system clock design.

Internally, the various functional units are connected by separate instruction and data buses. For connection with external memory and I/O, a unified address bus and a unified data bus are extended off-chip. The main functional units are discussed briefly in the following sections, and more fully in the *Internal Architecture* chapter.

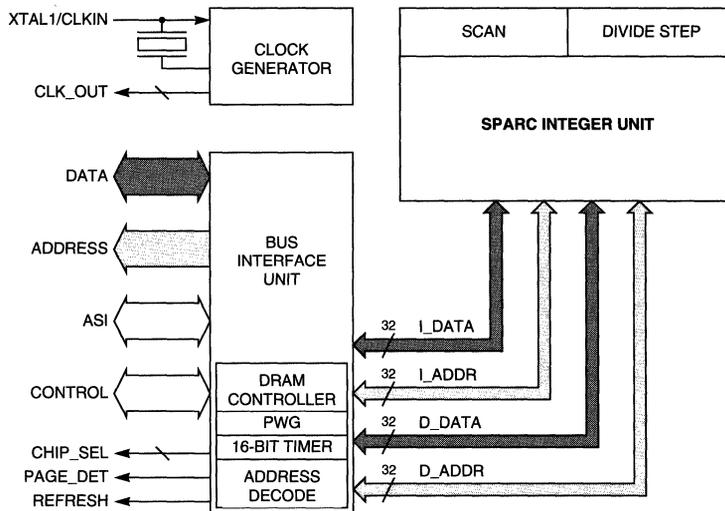


Figure C1-4. Internal Architecture (Block Diagram)

1.5.1 Integer Unit

The Integer Unit (IU) is a compact, fully custom implementation of the SPARC architecture. The IU is hard-wired for high performance. Its internal functional units are designed around a modular architecture and can be customized to meet different application requirements. In the MB86933, for example, this flexibility was used to provide direct hardware support for integer multiplication, and to extend the SPARC instruction set by supporting divide-step and scan instructions.

The IU implements a five-stage instruction pipeline to allow a sustained execution rate of nearly one instruction per cycle. The operation of the pipeline under ideal conditions is illustrated in Figure C1-5.

The pipeline consists of the following stages:

- Fetch (F)—One of the instruction memory spaces is addressed and returns an instruction.
- Decode (D)—The instruction is decoded; the register file is addressed and returns operands.
- Execute (E)—The ALU computes a result.
- Memory (M)—External memory is addressed (for load and store instructions only; this stage is idle for other instructions).
- Writeback (W)—The result (or loaded memory datum) is written into the register file.

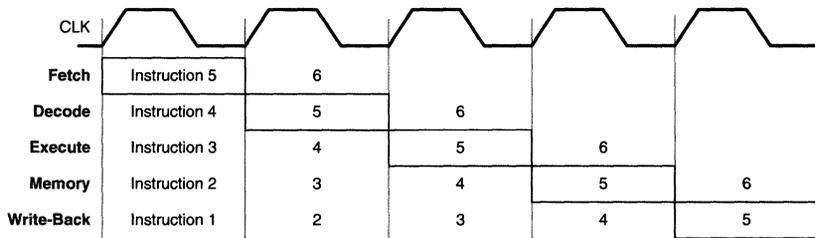


Figure C1-5. Instruction Pipeline

No instructions execute out-of-order; that is, if instruction A enters the pipeline before instruction B, then instruction A necessarily reaches the writeback stage before instruction B. Conditions that hold up the pipeline, and the effect of traps on pipeline operations, are discussed in the *Internal Architecture* chapter.

1.5.2 Bus Interface Unit

The Bus Interface Unit (BIU) contains the logic that allows the processor to communicate with the system.

1.6 External Interface

The processor's external interface consists of signals, bus operations, and system support functions. This section gives an overview; details are discussed more fully in the *External Interface* chapter. The *System Design Considerations* chapter discusses issues that are likely to arise in the design of MB86933-based system.

1.6.1 Signals

The processor's external signals, illustrated in Figure C1-6, can be grouped by function as follows:

- Processor Control and Status—Reset, error, and clock signals.
- Memory Interface—Data and address buses, ASI and byte-enables, chip-selects, and other control signals used to access external memory and memory-mapped devices.
- Bus Arbitration—Signals used by external devices in requesting, and by the processor in granting, control of the bus.
- Peripheral Functions—Interrupt-requests and timer overflow.
- Boundary-Scan—Test signals used for hardware verification.
- ROM Size—Used to identify ROM size.

1.6.2 Bus Operation

At any given time the Bus Interface Unit is handling requests for external memory and I/O operations, is arbitrating for bus access, or is idle. From the point of view of the external system, bus transactions are handled in fairly standard ways:

- Memory and I/O Operations—Read and write transactions are initiated with the BIU asserting the -AS signal. The RD/-WR output indicates the transaction type. The $\text{-BE}[3:0]$ outputs indicate the transaction width. The BIU drives the address and ASI signals, and either drives (during stores) or reads (during loads) the signals on the data bus. The transaction ends when the external system or programmable wait-state generator asserts -READY .

An atomic load-store is executed as a load followed immediately by a store, with no operation allowed between. The -LOCK output is asserted to indicate that the bus is being used for more than one consecutive memory operation.

- Arbitration—Any external device can request ownership of the bus by asserting the -BREQ signal. The BIU three-states its bus drivers and asserts -BGRNT to indicate that it is relinquishing control of the bus. Upon completion of its transaction the external device de-asserts -BREQ , and the BIU responds by de-asserting -BGRNT during the following cycle.

Chapter 4 of this addendum contains bus timing diagrams and a bus state diagram, further describes bus operations, and describes transactions that are interrupted by exceptions.

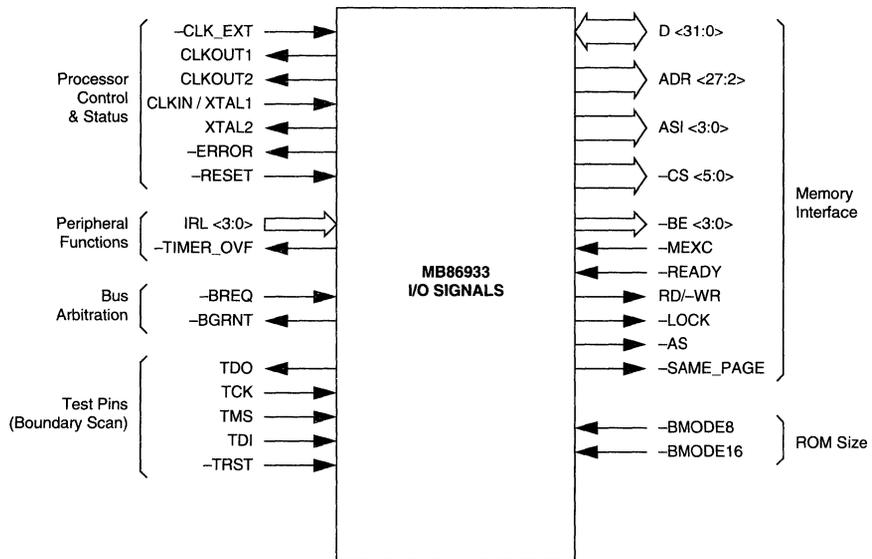


Figure C1-6. Input and Output Signals

1.6.3 System Support Functions

MB86933 system support is the same as MB86930 system support. Please refer to section 1.5.3 of the main section of this manual for a description of the system support functions.

1.7 Development-Support Tools

The MB86933 development-support tools are the same as the MB86930 development-support tools. Please refer to section 1.6 of the main section of this manual for a description of the development-support tools.

.....

Programmer's Model

This chapter describes the MB86933 processor resources that are available to software. It discusses the user and supervisor modes, the organization of the address space, the processor registers, the supported data types, the instruction set, and interrupts and traps. A separate section describes the internal state of the processor after reset.

The *Programming Considerations* chapter contains information about how to use these processor resources to best advantage.

2.1 Program Modes

The SPARC architecture provides two mutually exclusive modes of program execution, *user mode* and *supervisor mode*. The processor is in supervisor mode when the S bit of the Processor State Register (PSR) is 1, and in user mode when this bit is 0. Instructions which access either special-purpose registers or alternate memory spaces are privileged. The use of *privileged* instructions is restricted to supervisor mode.

Separate user and supervisor modes provides system protection in multitasking environments. System code runs in supervisor mode and has full access to processor resources, while application code runs in user mode and is prevented from having unwanted side effects. Embedded systems connected to a network can use a protection scheme based on the distinction between user and supervisor modes. In such a scheme, network service routines intended to have system-wide effects

run in supervisor mode. Routines intended to have only local effects, on the other hand, run in user mode.

In many embedded systems, however, this hierarchy is not required, and the processor can operate exclusively in supervisor mode. In this way, application code can directly manipulate the Current Window Pointer (in the PSR) and other processor control fields.

On reset, the processor is in supervisor mode. To enter user mode, software must clear the S bit in the PSR. The processor enters supervisor mode from user mode only when a hardware reset, an interrupt, or a trap occurs. A return from trap (RETT) instruction restores the value the S bit had before the trap was taken.

2.2 Memory Organization

The processor can directly address up to 4 Gb of memory, organized into 16 address spaces of 256 Mb each. These address spaces may or may not overlap in physical memory, depending on the system design. Every external access involves a 4-bit Address Space Identifier (ASI) as well as a 26-bit word address. The ASI selects one of the address spaces, and the address selects a word within that space (see Table C2-1).

Only the user instruction and data spaces are accessible in user mode. The other 254 address spaces can be accessed only in supervisor mode.

Table C2-1: ASI Address Space Map

ASI <3:0>	Address Space
0x0	Application Definable
0x1	Control Registers
0x2 - 0x3	Reserved
0x4 - 0x7	Application Definable
0x8	User Instruction Space
0x9	Supervisor Instruction Space
0xA	User Data Space
0xB	Supervisor Data Space
0xC - 0xF	Reserved

Note that the MB86933 has no caches, no write buffer, no pre-fetch buffer, and has six register windows rather than eight. It has twenty-six Address Bus signals (ADR<27:2>) rather than thirty, four Address Space Identifier signals (ASI<3:0>) rather than eight, no emulator-support signals, and no memory management unit. These and other differences between the MB86933 and other SPARClite processors should be considered when porting code to the MB86933 from another SPARClite processor, and when porting code from the MB86933 to another SPAR-

Clite processor. Documentation for other SPARClite should be referenced to identify differences with the MB86933 that may affect ported code.

Loads and stores are the only instructions that cause external accesses. Versions of these instructions exist for transferring bytes, half-words, words and double words between memory (or I/O) and processor registers. Addressing conventions for external accesses are “big-endian”:

- *Bytes*—Increasing the address decreases the significance of a byte within the word. That is, the most significant byte of a word—the “big end” of the word—is accessed when bits [1:0] of the address are both 0. The least significant byte is accessed when address bits [1:0] are both 1.
- *Halfwords*—The most significant halfword of a word is accessed when bit 1 of the address is 0, and the least significant halfword when address bit 1 is 1.
- *Doublewords*—The most significant word of a doubleword is accessed when bit 2 of the address is 0, and the least significant word is accessed when address bit 2 is 1.

The address of a halfword, word, or doubleword is the address of its most significant byte. The addressing conventions are illustrated in Figure C2-1.

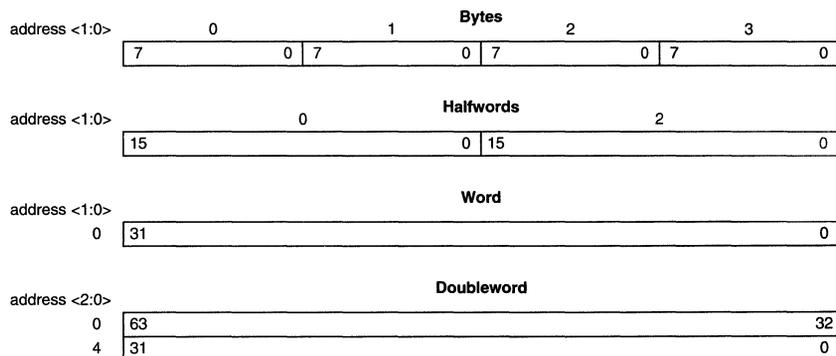


Figure C2-1. Addressing Conventions

Load and store operations require proper alignment of data in memory. An aligned doubleword address is divisible by 8, an aligned word address is divisible by 4, and an aligned half-word address is divisible by 2. If a load or store instruction generates an improperly aligned address, a `memory_address_not_aligned` trap occurs, and the access must be performed piecemeal under software control.

The processor does not contain memory-management hardware. Virtual-address translation can be handled by software or by an external memory-management unit.

2.3 Registers

There are two types of registers: the *general-purpose* or *r registers* whose contents have no pre-assigned meaning, and the *special-purpose registers* that contain control and status information, or special-purpose data. All registers are 32 bits wide. The register set is illustrated in Figure C1-2.

The general-purpose (*r*) registers can be accessed in user mode. There are 104 *r* registers. Eight are *global registers*; the other 96 registers are divided into six overlapping blocks called *windows*.

There are of two kinds of special-purpose registers: (1) registers that are defined by the SPARC architecture, and (2) memory-mapped registers that control peripheral functions. Special instructions exist for reading and writing each SPARC register except the Program Counter and the Next Program Counter. The memory-mapped registers can be read and written with the alternate-space load and store instructions. All instructions that access special-purpose registers are privileged except reads and writes to the SPARC-defined Y register.

2.3.1 Register Windows

The general-purpose register set is organized into a set of 8 global registers and a set of overlapping windows, as specified by the SPARC architecture. There are 6 windows in the MB86933. Each window contains 24 registers. Of these, 8 are *local* to the window, 8 are "*out*" registers shared with the adjacent window below, and 8 are "*in*" registers shared with the adjacent window above. This organization is illustrated in Figure C2-2.

Thirty-two general-purpose registers can be accessed directly at any time: the 8 global registers, and the 24 registers of the currently active window. The value in the Current Window Pointer (CWP) field of the Processor State Register (PSR) determines which window is active. (See Section 5.3 for register addressing conventions.)

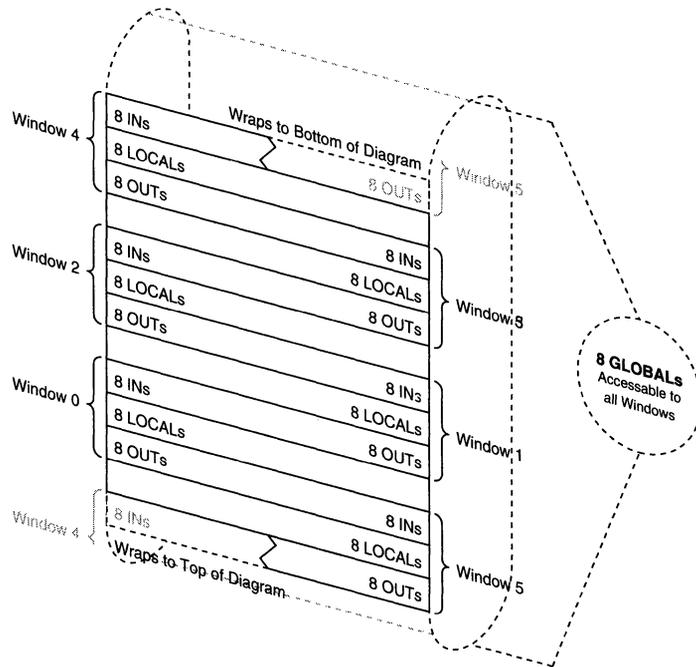


Figure C2-2. Register Windows

Register Addressing

Please refer to Section 2.3.1 of the main section of this manual for a description of MB86933 register addressing.

Performance Features

Please refer to Section 2.3.1 of the main section of this manual for a description of the MB86933 performance features.

2.3.2 Special Uses of the r Registers

Please refer to Section 2.3.2 of the main section of this manual for a description of MB86933 r register use.

2.3.3 SPARC-Defined Special-Purpose Registers

The registers discussed in this section are defined as part of the SPARC architecture.

Processor State Register (PSR)

The Processor State Register is the primary processor control and status register. It contains 11 mode and status fields that configure the processor and report processor status and exception results. The *mode* fields, shown in upper case in Figure C2-3, are set by the operating system to configure the processor. The *status* fields, shown in lower case, are set by the processor to indicate the effects of instruction execution.

Except for several fields described below, the PSR can be written and read directly with the privileged instructions WRPSR and RDPSR. The PSR can also be modified by the SAVE, RESTORE, Ticc, and RETT instructions, and by any instruction that modifies the condition codes.



Figure C2-3. Processor State Register

- Bits 31-28: **Implementation (impl)**—Identifies the implementation number of the processor as 0. The value in this field cannot be changed by a WRPSR instruction.
- Bits 27-24: **Version (ver)**—Identifies the processor version as 4, and is intended for factory use. It can be read, but not written.
- Bits 23-20: **Integer Condition Codes (icc)**—Contains the negative (n), zero (z), overflow (v), and carry (c) integer condition-code flags. These bits are modified by the WRPSR instruction, and by arithmetic and logical instructions whose names end with the letters *cc* (for example, ANDcc). The Bicc (Branch on integer condition codes) and Ticc (Trap on integer condition codes) instructions transfer program control based on the values of these bits. The integer condition code flags are defined as follows:
- n (Bit 23) Set to 1 if the ALU result was negative for the last instruction that modified the icc field; equal to 0 otherwise.
 - z (Bit 22) Set to 1 if the ALU result was zero for the last instruction that modified the icc field; equal to 0 otherwise.
 - v (Bit 21) If this bit equals 1, an arithmetic overflow occurred on the last instruction that modified the icc field; it equals 0 otherwise. Logical instructions that modify the icc field always reset the overflow bit to 0.
 - c (Bit 20) If this bit equals 1, either an arithmetic carry out of bit 31 occurred on the last addition that modified the icc, or a borrow out of bit 31 occurred as the result of the last subtraction that modified the icc. The carry bit equals 0 otherwise. Logical instructions that modify the icc field always reset the carry bit to 0.
- Bits 19-12: **Reserved**—This field is reserved. When using the WRPSR instruction, this field should always be written with 0s.
- Bits 11-8: **Processor Interrupt Level (PIL)**—Specifies the levels of interrupt that the processor will accept. The processor accepts only interrupts with level 15 (non-maskable interrupts), or

Next Program Counter, Ancillary State Registers,

Please refer to Section 2.3.3 of the main section of this manual for a description of these registers.

2.3.4 Memory-Mapped Control Registers

In addition to the registers defined by the SPARC architecture, the MB86933 provides a collection of memory-mapped registers that control peripheral functions. Figure 2-5 shows these registers and their locations in memory. The memory-mapped registers can be read and written with the alternate-space load and store instructions, which are privileged.

0x0000080	ASI=0x1	System Support Control Register
0x0000120	ASI=0x1	Same-Page Mask Register
0x0000124	ASI=0x1	Address Range Specifier Registers (ARSR <5:1>)
0x0000140	ASI=0x1	Address Mask Register (AMR <5:0>)
0x0000160	ASI=0x1	Wait-State Specifier Registers (WSSR <2:0>)
0x0000174	ASI=0x1	Timer Register
0x0000178	ASI=0x1	Timer Preload Register

Figure C2-5. Locations of Memory-Mapped Control Registers

Same-Page Mask Register, Address Range Specifier Register, Address Mask Register, Wait-State Specifier Register, System Control Support Register, Timer Register, Timer Preload Register

Please refer to Section 2.3.4 of the main section of this manual for a description of these registers.

2.4 Data Types

Please refer to Section 2.4 of the main section of this manual for a description of the data types.

2.5 Instructions

Please refer to Section 2.5 of the main section of this manual for a description of the instructions. Note that *modulo 8* in the description becomes *modulo 6* for the MB86933 processor.

2.6 Interrupts and Traps

Please refer to Section 2.7 of the main section of this manual for a description of the interrupts and traps. Note that *modulo 8* in the description becomes *modulo 6* for the MB86933 processor.

CHAPTER C3



Internal Architecture

The MB86933 internal architecture is illustrated in Figure C3-1. The processor consists of a Clock Generator, an Integer Unit, and a Bus Interface Unit. Internally, the various functional units are connected by separate instruction and data buses. A unified address bus and a unified data bus extend off-chip for connecting external memory and I/O.

This chapter discusses the individual functional units and gives an overview of the flow of data and control signals through the processor.

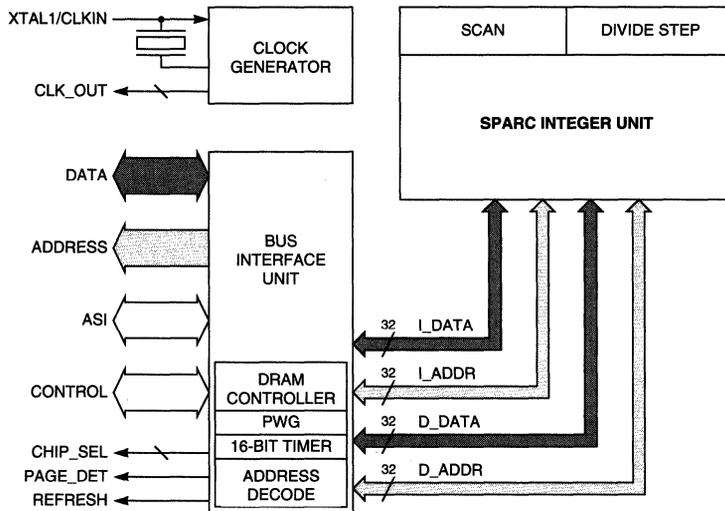


Figure C3-1. Internal Architecture (Block Diagram)

3.1 Integer Unit

The Integer Unit (IU) is a compact, full-custom implementation of the SPARC architecture. It is hard-wired for maximum performance; that is, it uses no micro-code. It contains three functional units:

- *Instruction Block*—Contains the instruction pipeline and decodes instructions into control signals for the other blocks.
- *Address Block*—Performs all instruction-address manipulations.
- *Execute Block*—Performs all data manipulations, and generates operand addresses for load and store instructions and effective addresses for some of the control transfer instructions.

The IU is based on a Harvard (Aiken) architecture, as shown in Figure C3-2. There are separate address buses for instructions and data. There are also two 32-bit data interfaces: the instruction data bus, and the data bus. These four buses allows the IU to retrieve data and instructions simultaneously from on-chip cache.

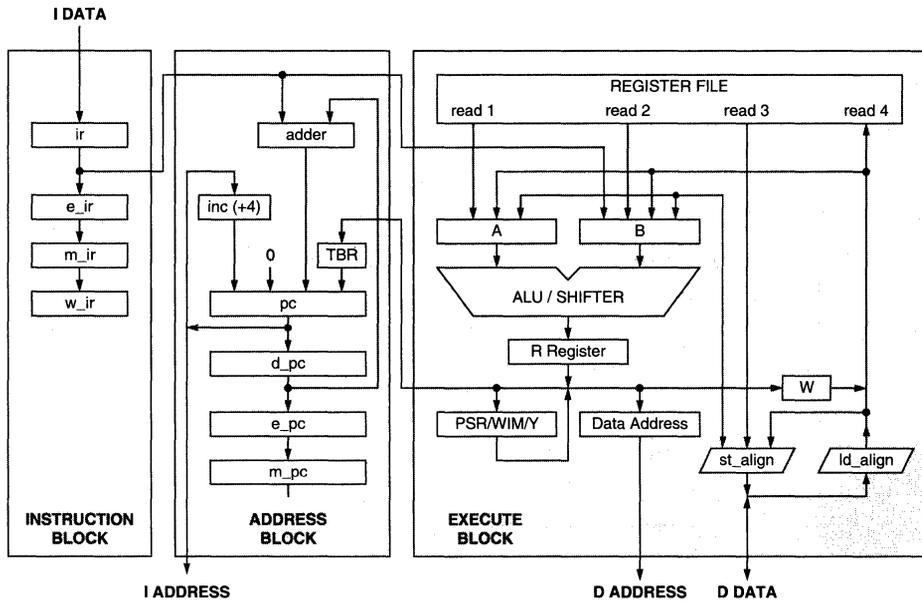


Figure C3-2. Integer Unit Data Path

3.1.1 I Block

The instruction block (I Block) contains the five-stage instruction pipeline and the logic that decodes instructions into control signals for the rest of the IU. The I block detects all bypass and interlock conditions.

The main interfaces to the I block are:

- The Instruction data bus from main memory.
- The Immediate data field that goes to the A block for computing PC relative control transfers and to the E block to be used as immediate data.
- Control signals to the A block and E block including the register file read and write addresses, register enable signals, multiplexer controls, and partly or fully decoded operation codes for the ALU/Shifter.
- Status signals back from the E block including possible trap conditions such as `memory_address_not_aligned` and `tag_overflow`.

Instruction Pipeline

The IU implements a five-stage instruction pipeline to allow a sustained execution rate of nearly one instruction per cycle. The operation of the pipeline under

ideal conditions is illustrated in Figure C3-3. The pipeline consists of the following stages:

1. Fetch (F)—One of the instruction memory spaces is addressed and returns an instruction. (The figure below assumes a hit in the instruction cache.)
2. Decode (D)—The instruction is decoded; the register file is addressed and returns operands.
3. Execute (E)—The ALU computes a result.
4. Memory (M)—External memory is addressed (for load and store instructions only; this stage is idle for other instructions).
5. Writeback (W)—The result (or loaded memory datum) is written into the register file.

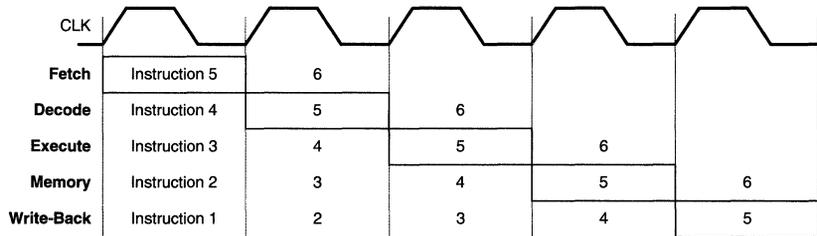


Figure C3-3. Instruction Pipeline

No instructions execute out-of order; that is, if instruction A enters the pipeline before instruction B, then instruction A necessarily reaches the writeback stage before instruction B.

The control logic for the instruction pipeline is illustrated in Figure C3-4. At each cycle a horizontal control word is available that is wider than 32 bits and controls every multiplexer, latch-enable, and unit op-code in the chip. The horizontal control word is composed of control signals that are active during the decode stage of instruction N, the execute stage of instruction N-1, the memory stage of instruction N-2 and the writeback stage of instruction N-3. Some control bits require no decoding and are simply hardwired from the appropriate bits in the instruction register. Because the SPARC instruction set is not completely orthogonal (not every instruction field has the same meaning in every instruction) most bits require some decoding based on a single instruction in the pipeline. Some control

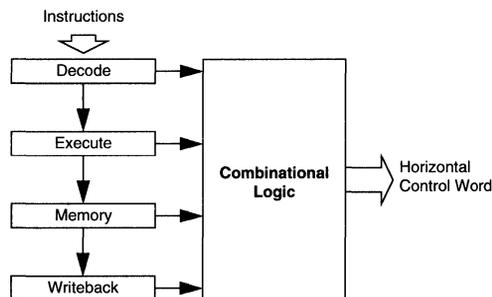


Figure C3-4. Instruction Pipeline Control Logic

bits require decoding using logic that looks at two instructions in the pipeline - when controlling multiplexers to select data bypass paths, for example.

Pipeline Hold

The IU does not complete one instruction on absolutely every cycle. During a load instruction, for example, external memory may be slow in returning the requested data. Because the IU does not execute or complete instructions out of order, the pipeline must be stopped until the requested data is returned. Only then can the instruction complete, and only then can the following instructions be executed.

There are also some hazards built into the IU data path that require interrupting the one-cycle-per-instruction sequence of the pipeline. For example, a double-word load cannot be performed in one cycle because there is not enough memory or register-file bandwidth to move the data through the datapath. Another example is a load to a register that is followed by an instruction that uses that register. Because the operand of the second instruction is required in the decode stage but is not available, this instruction must be delayed until the operand is available.

Conditions that hold up the processor pipeline are handled uniformly by the I Block control logic and are referred to as *hold conditions*. A complete list of possible hold conditions is given in Table C3-1.

The *interlock conditions* are:

- **Load/Use Instruction Pairs**—If a load instruction that has $rd=N$ as its destination register is followed by an instruction that uses $rs=N$ as one of its source operands, then the load must proceed through Writeback before the following instruction can enter the Execute stage.
- **CALL/Use %r15 Instruction Pairs**—Similarly, since the CALL instruction implicitly writes the current value of the PC into r15, it must proceed to

Writeback before any following instruction that uses r15 can enter the Execute stage.

Any time an interlock is detected, a NOP is inserted into the pipeline. The address block is signaled, so that the address of the instruction that causes the interlock is replicated in the address pipe. The NOP itself cannot cause a trap.

Table C3-1: Conditions That Cause a Pipeline Hold

Name	Description	Pipeline Stage	Instruction Affected
ihold	Processor is attempting to fetch an instruction that is not yet available.	Fetch	Any instruction
dhold	Data is not yet available	Memory	Loads and Stores
mhold	Multiplication in progress	Execute	Integer Multiplication
Interlock	An instruction in the pipeline must wait for some prior instruction to be completed (through Writeback).		Load/Use and CALL/Use r15 Instruction Pairs
Multicycle Instruction	An instruction which inherently requires more than one cycle is in the pipeline	Execute	Load and Store Double-word, Atomic Load/Store

The multicycle instructions are LDD, LDDA, STD, STDA, LDSTUB, LDSTUBA, SWAP, and SWAPA. When a multicycle instruction enters the Execute stage, it and the instruction in the `d_ir` register are frozen for an additional cycle. Although it is possible to detect a multicycle instruction while it is in the Decode stage (unlike interlocks, which cannot be detected without looking at two instructions, those in the `d_ir` and `e_ir` registers), the I Block allows it to progress to the Execute stage before a hold is generated and inserted. This simplifies control somewhat because there are fewer points at which the pipeline must be held.

Note that the maximum number of internally generated hold cycles an instruction can cause is two, as in the following case:

```
LDD [%r1+%r2], %0r4
ADD %r5, %r5, %r6
```

The LDD takes two cycles, and it generates an interlock because the next instruction uses the data loaded in the second data memory cycle of the LDD instruction.

When a hold condition occurs, combinational logic generates one or more *freeze signals* that prevent latches from being updated, and hence keep the pipeline from advancing. For some holds—`dhold`, for example—the entire pipeline is frozen, with freeze signals being generated for all stages in the pipeline. For other

holds—interlock conditions, for example—later stages in the pipeline must advance for the hold condition to be resolved. Thus only the earlier stages of the pipeline are frozen.

Trap Logic

The MB86933 supports precise traps. That is, when a trap occurs, the saved programmer-visible state of the processor reflects the completion of all instructions prior to the trapped instruction, and no following instructions including the trapped instruction. Thus, when an instruction causes a trap, one of two statements is true:

- No results from that instruction have been written into the programmer-visible registers (the register file or the PSR, TBR, WIM, or Y registers).
- Or, if data has been written into a programmer-visible register, the data contained in that register prior to being written by the trapped instruction is saved by the processor and can be restored when the trap is taken.

Table C3-2 shows the pipeline stages in which the various trap conditions are detected.

Table C3-2: Detection of Trap Conditions

Priority	Trap Type	Stage Detected	Trap
1			reset (hardware reset)
1	-	D	reset
2	1	F	instruction_access_exception
3	3	D	priv_instruction
4	2	D	illegal_instruction
5	4	D	fp_disabled
5	36	D	cp_disabled
6	5	D	window_overflow
7	6	D	window_underflow
8	7	E	mem_address_not_aligned
10	9	M	data_access_exception
11	10	E	tag_overflow
12	128-254	D	trap_instruction (Ticc)
13	255	F	instruction_breakpoint
13	255	M	data_breakpoint
14	31		interrupt_level_15
15	30		interrupt_level_14
.	.		.
.	.		.
28	17		interrupt_level_1

As shown in Table C3-2, the last stage in which a trap can be detected is the Memory stage (a data memory exception for a load or store). If a programmer-visible

register is updated prior to this stage, its original contents must be restored when and if the trap is taken.

Due to the pipelined operation of the IU, a trap condition for one instruction may actually be detected before a trap condition for a prior instruction. Thus, it is necessary to align the detected trap conditions so that all trap conditions for instruction N are considered together before any trap conditions resulting from instruction N+1 are considered.

The trap coder is illustrated in Figure C3-5. Its purpose is to align in time the (possibly several) trap sources for a single instruction to determine if a trap is to be taken or not and, if taken, to determine the highest priority trap and code its trap type.

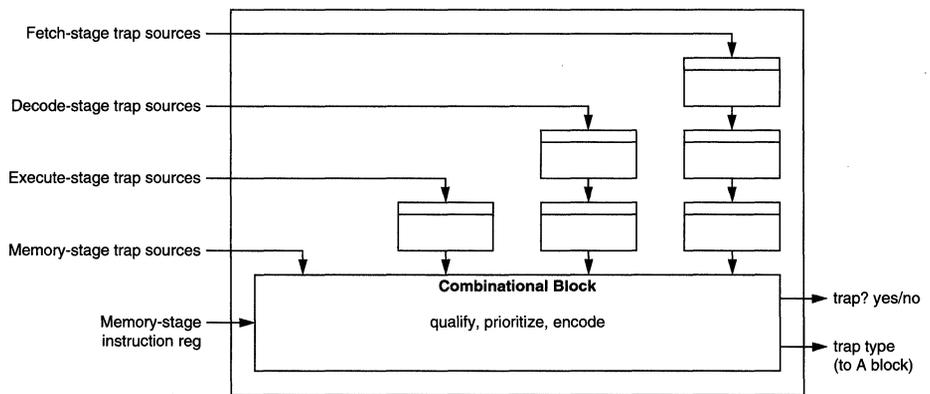


Figure C3-5. Trap Coder

When a trap is taken, the trap type field goes to the A Block where it is used immediately as a trap target address (when concatenated with the Trap Base Address) and is latched into the Trap Base Register.

3.1.2 A Block

The A Block contains the address pipeline. Along with the E Block, it is responsible for all instruction-address manipulations. The A Block executes the CALL and Bicc instructions. The A Block and E Block are used together to execute the JMPL, Ticc, and RETT instructions. In these cases, the A Block controls the update of the Program Counter. The A Block's main interface to the rest of the chip outside the IU is the instruction address bus.

The address pipeline is illustrated in Figure C3-6. The fetch-stage program counter (PC) addresses instruction memory via the instruction address bus. Because a CALL, JMPL, or trap may require that the address of an instruction be

written back to the register file, the address of every instruction tracks the instruction itself in the instruction pipeline so that it is available in the memory stage if it must be written back to the register file. These address pipeline registers are the decode, execute, and memory program counters. Each of these registers contains the address from which the instruction in the corresponding instruction register was fetched.

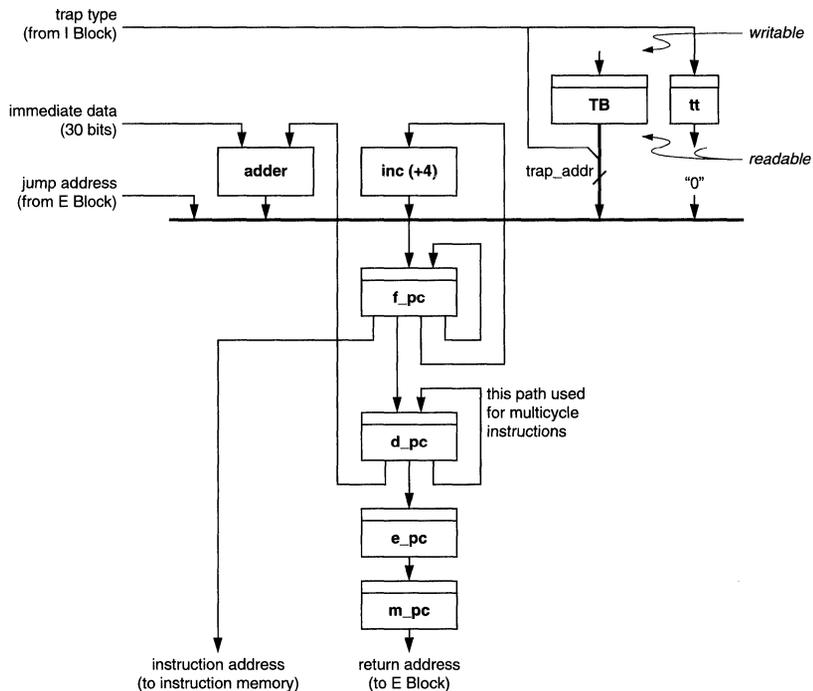


Figure C3-6. Address Pipeline

The PC has five possible sources:

1. +4 incremter, for normal, sequential instruction fetch.
2. The address adder, for PC-relative control transfer (Bicc or CALL instruction). The immediate data field contains offset information and comes from the I Block.
3. The jump address for a JMPL or RETT instruction. The jump address bus contains jump target information and comes from the E block by way of the register file and ALU.
4. The TBR, concatenated with the trap type (tt) or with zeroes (when Single-Vector Trapping is enabled), during a Ticc instruction execution or an interrupt or

trap. The trap type comes from the trap priority encoder, part of the I Block; when concatenated with TBR[31:12], it gives the target address for a trap.

5. Zeroes, concatenated with the trap type, for reset.

Note that "+4" is used to indicate that the (byte) address is incremented by 4 to fetch the next instruction. In reality, the two least significant bits of the address are not implemented in hardware because they are never used. Word alignment, for the case of a jump address coming from the E Block is verified in the E Block (and to some extent, the I Block).

The return address bus is written back to the register file in the case of a CALL, JMPL or Trap.

Several control signals come from the I block. These include:

- PC input-select signals that control the PC input multiplexer.
- The address adder control signal, which determines whether a 30-bit or a 22-bit immediate address field is added to the previous value of the PC (now found in the decode-stage PC).
- Pipeline freeze signals that can prevent the updating of registers in the pipeline when a hold condition is detected.

3.1.3 E Block

The E Block is responsible for all IU data manipulations. It generates operand addresses for load and store instructions, and effective addresses for some of the control transfer instructions.

As shown in Figure C3-7, the E Block contains the Store Align Unit (SAU), the Load Align Unit (LAU), the Register File (RF), and the Adder, Shift, and Logic Unit (ASLU). The E Block also contains the result bypass logic that determines which operands are driven into the ASLU, and the store bypass logic that determines what data is latched for stores.

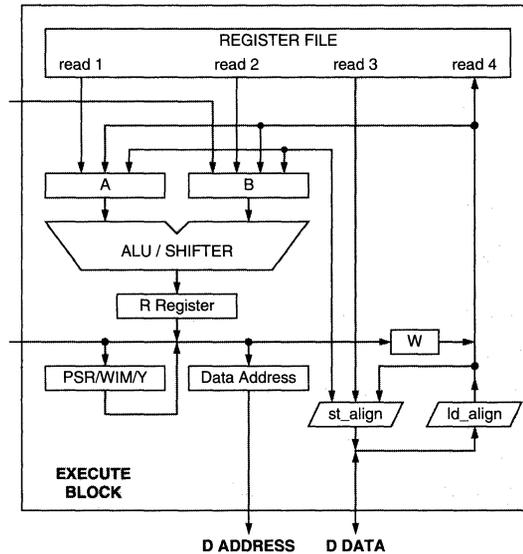


Figure C3-7. Execute Block

Adder, Shift, and Logic Unit (ASLU)

The ASLU incorporates an integer adder, a barrel shifter, a logic unit, and a scan unit. The integer adder calculates the results of the addition, subtraction, multiply-step, and divide-step instructions, and generates the carry, overflow, negative, and zero condition code values. It is used in load and store operations to calculate effective data addresses, and in register-indirect control transfers to calculate the new address to be placed in the PC register of the A Block. The integer adder also serves the multiplication unit by adding the “sum” and “carry” vectors during integer multiplications. The barrel shifter/logic unit executes the logic and shift instructions. The scan unit exists solely to support the scan instruction.

Results from the integer adder, the barrel shifter, the logic unit, and the scan unit are multiplexed into the R (Result) Register. Results from the integer adder are also made available to the Y Register.

Register File

The register file contains 104 registers of 32 bits each. The organization of these registers into windows is discussed in the *Programmer’s Model* chapter. The register file has one write port and three read ports. The write port is used for the instruction destination register (denoted *rd* in instruction descriptions). Two of the read ports are used for the two instruction source registers (*rs1* and *rs2*). The

remaining port is used for the data to be stored when a store or swap instruction is executed. In this way, even store instructions can be executed in a single cycle.

The register file also contains the address decoders for all four ports. Each address presented to the decoders consists of 8 bits derived from an instruction field, and the Current Window Pointer. These are physical addresses into the register file memory array.

Bypass Logic

As shown in Figure C3-7, the A and B operand registers have inputs that come from sources other than the register file or the immediate data bus. These inputs are results from previous instructions that have not yet written back to the register file. There are two such *bypass paths* in the E Block:

- *Result Bypass*—The result of an ALU operation in the R register is written back to the A or B operand register in the Memory stage of the following ALU operation.
- *Write Bypass*—The data in the W register is written to the A or B operand register, in the Writeback stage.

The result bypass path is selected when one instruction generates a result that can be used by the immediately following instruction. More precisely, if an instruction in the Decode stage of the pipeline has $rs1 = N$, and the instruction in the Execute stage has $rd = N$, the $rs1$ operand will not come from the register file, but directly from the R register in the ALU through the result bypass. Since an intervening SAVE or RESTORE instruction may have changed the Current Word Pointer, it is the *physical addresses* of the register source and destination that are compared, not the logical addresses (which depend on the CWP).

As an example, consider the instruction sequence:

```
add %r1,%r2,%r3          ; r1 + r2 -> r3
add %r3,%r4,%r5          ; r3 + r4 -> r5
```

The second add instruction takes its A source operand not from the register file, but directly from the result of the ALU through the result bypass.

The write bypass is selected when an instruction in the Decode stage has $rs1 = N$, and the instruction in the Memory stage has $rd = N$. In this case, the $rs1$ operand will not come from the register file, but from the W register through the write bypass. In the following instruction sequence, the third instruction uses the write bypass as its A source operand:

```

add %r1,%r2,%r3           ; r1 + r2 -> r3
add %r4,%r5,%r6           ; r4 + r5 -> r6
add %r3,%r7,%r8           ; r3 + r7 -> r8

```

If both bypass conditions apply, the result bypass takes precedence.

There is a third bypass path, called the *store bypass*, that is shown in Figure C3-7. The register file has a dedicated store port that is used for reading the rd register of a store instruction, which contains the data to be stored. The store port is read in the Execute stage of the store. When a store and the immediately preceding instruction access the same rd register, a bypass from the Writeback stage of the preceding instruction to the Memory stage of the store is needed. In the code sample below, the result of the first instruction becomes available to the Memory stage of the store by means of the store bypass path.

```

add %r1,%r2,%r3           ; r1 + r2 -> r3
st  %r3[%r4 + %r5]        ; r3 -> mem[r4 + r5]

```

Branch Evaluation Logic

The branch evaluation logic, which forms part of the E Block, evaluates branch conditions based on the current values of the integer condition codes of the PSR register. The icc bits n (negative), z (zero), c (carry) and v (overflow) form part of the branch evaluation block. The interpretation of these bits is discussed in the *Programmer's Model* chapter.

There are several ways that the icc bits can be modified. First, they can be written and read via the jump address bus by the instructions WRPSR and RDPSR.

Certain arithmetic instructions modify the icc bits as a side effect. When one of these instructions is executing, the new icc values are generated in the E Block during the Execute stage, latched at the end of this stage, and loaded into the PSR during the Memory stage.

Another path leads to the icc bits from the Writeback-stage copy of the PSR. When a trap occurs on an instruction that alters the icc bits, this path allows the pre-trap icc values to be restored to the PSR.

The combinational logic that performs the branch evaluation for the IU condition codes has as inputs:

- *Integer Condition Codes*—Directly from the ALU if the instruction in the Execute stage is one that can modify the icc, from the multiplication unit, or from the icc bits of the PSR if the instruction in the Execute stage is not one that can modify the icc.

- *The cond Field*—From the branch instruction in the Execute stage. (See the discussion of the Bicc instruction in the *Programmer's Model* chapter.)
- *Bicc Indicator*—A control signal that indicates whether the instruction in the Decode stage is a Bicc instruction. This signal remains valid into the Execute stage.

The output of the combinational logic is a single signal that, when active, causes the branch target address to be loaded into the PC during the Execute stage. Otherwise, PC+4 is loaded into the PC.

Load Align Unit (LAU) and Store Align Unit (SAU)

The LAU and SAU align data for loads and stores, respectively. Bytes and half-words to be loaded are right-justified in a 32-bit word, and either sign-extended or zero-extended on the left, depending on whether the load instruction specified signed or unsigned operation. The LAU performs the alignment and extension during Writeback.

Byte and halfword stores take their data from the least significant byte or halfword of the register specified in the instruction's rd field. The SAU performs the necessary alignment for writing the data to the byte or halfword memory address specified in the instruction.

Multiply Unit

The E Block contains hardware to perform integer multiplications. The Multiply Unit (MU) multiplies two 32-bit signed or unsigned integers to produce a 64-bit product. Some multiplication instructions modify the integer condition codes as a side effect; others do not. The multiplication instructions are discussed in the *Programmer's Model* chapter.

The multiply hardware implements a version of *Booth's algorithm*. Booth's algorithm is similar to a "shift and add" multiply algorithm in that it scans the multiplier from the least significant to the most significant bit and, based on the bit string encountered, iteratively adds the multiplicand to produce partial products. It is also similar in that the resulting partial product is right shifted to ready it for the following iteration of the algorithm.

Booth's algorithm differs from a "shift and add" algorithm in that it can also be used directly with a negative multiplier (whereas "shift and add" requires a positive multiplier). It also differs in that the hardware must provide for both addition and subtraction of the multiplicand. In particular, a 1-bit Booth's algorithm examines two multiplier bits per iteration, looks for a bit transition, and either adds the multiplicand, subtracts the multiplicand, or adds zero to the existing partial product to produce the new partial product. It "retires" one bit of the multiplier per iteration.

Table C3-3 shows the possible bit transitions encountered in the multiplier for a 1-bit Booth, and the value that is added to the multiplicand for each transition.

Table C3-3: Booth's Algorithm

Multiplier Bits		Add to Shifted Partial Product
Current	Previous	
0	0	+0
0	1	+multiplicand
1	0	-multiplicand
1	1	+0

This technique can be extended so that more than one bit is examined during a given iteration. In particular, the MU performs an 8-bit Booth's algorithm. It examines 9 bits of the multiplier at a time and, based on the eight transitions of these nine bits, determines what multiple of the multiplicand to add to the old partial product to produce the new partial product. The addition is performed in the ALSU.

The MU produces 8 bits of the final product and "retires" 8 bits of the multiplier per cycle, and therefore requires only 5 cycles to do a 32x32 bit multiply (producing a 64-bit result).

The execution of the instruction is controlled by a synchronous state machine that generates control signals for the multiply hardware. Since instructions do not execute out of order, the Integer Unit (IU) must be frozen during the multiply instructions that require more than 1 cycle. Conceptually, the multiply instruction goes through all of the pipeline stages (F,D,E,M,W), but its Execute stage is from 1 to 5 machine cycles long. During the Fetch and Decode stages, the multiply instruction progresses like other instructions.

3.1.4 Programmer-Visible State and Processor State

The SPARC Architecture defines the *programmer-visible state* of the processor as a collection of registers, and specifies the effects of instructions in terms of these registers. These definitions implicitly assume that every instruction completes before the next one begins. The MB86933 processor, however, is pipelined, so that normally four instructions begin execution before the first one completes. The actual *processor state* (excluding the register file) therefore encompasses more than the programmer-visible state. For most of the programmer-visible registers, there is a corresponding register in the processor associated with the Writeback stage of the pipeline. That is, instructions normally update the register file and programmer-visible state registers in the Writeback stage.

An instruction may update staged copies of the PSR before Writeback, making the new values available to following instructions sooner; but these staged copies are not user visible. The PSR associated with the Writeback stage can never be updated early; if an instruction traps, it will not have altered any state that can not be restored.

3.2 Bus Interface Unit

The Bus Interface Unit (BIU) contains the logic that allows the processor to communicate with the system. When the BIU performs a read, it returns the data to the IU.

The BIU also handles external requests for control of the bus. The external signals of the BIU and the relative timing of events in typical bus operations are discussed in the *External Interface* chapter that follows. That chapter also treats the various system-support features of the processor in detail.

3.2.1 Exception Handling

The external memory system can indicate an exception during a memory operation by asserting the -MEXC input. If -MEXC is asserted during an instruction fetch, the BIU indicates an instruction memory exception to the IU. If -MEXC is asserted during a data fetch, the BIU indicates a data access exception to the IU.

Any system that wants to recover from this error should store the address and data for the write causing the exception into a register. It should also have a status bit to indicate that the exception was caused during a write operation. It is the responsibility of the data access exception service routine to determine the cause of the exception, and to recover accordingly.

3.2.2 Effect on the Pipeline

The pipeline hold signals, ihold and dhold , are asserted if an instruction or data cannot be made available in the cycle that it is required by the pipeline. In general the following hierarchy rules apply to the bus interface unit:

- The bus cycle currently in progress will complete
- If there is a pending request for a load or store operation, it will be serviced
- If there is a pending request for an instruction, it will be fetched.

The pipeline is stalled during every external memory access if the external -Ready signal or the internal Ready signal is not asserted. (See the Wait-State Specifier Registers description in Section 2.3.4 of the main section of this manual for a description of the internal Ready signal).

CHAPTER C4



External Interface

The processor external interface consists of signals for bus operations and for system control. This chapter details the MB86933 signal set, describes basic bus timing, and describes the programmable wait-state generator, on-chip timer, and same-page detection logic. See the MB86933 Data Sheet for specific electrical and timing information.

The System Design Considerations chapter of this document discusses issues that are likely to arise in the design of SPARClite systems.

4.1 Signals

The processor's external signals are illustrated in Figure C1-6 of the *Overview* chapter, and are listed in Table C4-1. A dash at the beginning of a signal name, as in -RESET , indicates that the signal is active-low.

Table C4-1: Input and Output Signals

Symbol	Type	Symbol	Type	Symbol	Type	Symbol	Type
ADR <27:2>	O S(L) G(Z) I(1)	-BMODE16	I	-LOCK	O S(L) G(Z) I(1)	TDI	I
-AS	O S(L) G(Z) I(1)	CLKOUT1 CLKOUT2	O G(Q) I(Q)	-MEXC	I S(L)	TDO	O
ASI <3:0>	O S(L) G(Z) I(1)	CLK_ECB	I	-SAME_PAGE	O S(L) G(1) I(1)	-TIMER_OVF	O S(L) G(Q) I(Q)
-BE 3:0	O S(L) G(Z) I(0)	-CS0, -CS1 -CS2, -CS3 -CS4, -CS5	O S(L) G(1) I(1)	RD/-WR	O S(L) G(Z) I(1)	TMS	I
-BGRNT	O S(L) G(0) I(Q)	D <31:0>	I/O S(L) G(Z) I(Z)	-READY	I S(L)	-TRST	I
-BREQ	I S(L)	-ERROR	O S(L) G(Q) I(Q)	-RESET	I A(L)	XTAL1 (CLKIN) XTAL2	I O G(Q) I(Q)
-BMODE8	I	IRL <3:0>	I A(L)	TCK	I		

NOTE:

- I = Input Only Pin
- O = Output Only Pin
- I/O = Either Input or Output Pin
- = Pins "must be" connected as described
- S(L) = Synchronous: Inputs must meet setup and hold times relative to CLKIN Outputs are Synchronous to CLKIN
- A(L) = Asynchronous: Inputs may be asynchronous to CLKOUT.
- G(...) = While the bus is granted to another bus master (-BGRNT=asserted), the pin is
 - G(1) is driven to V_{CC}
 - G(0) is driven to V_{SS}
 - G(Z) floats
 - G(Q) is a valid output
- I(...) = While the bus is between bus cycles (or being reset) and is not granted to another bus master, the pin is
 - I(1) is driven to V_{CC}
 - I(0) is driven to V_{SS}
 - I(Z) floats
 - I(Q) is a valid output

The following sections describe the signal set in detail, arranged by functional group as follows:

- Processor Control and Status—Reset, error, and clock signals.
- Memory Interface—Data and address buses, ASI and byte-enables, chip selects, and other control signals used to access external memory and memory-mapped devices.
- Bus Arbitration—Signals used by external devices in requesting, and by the processor in granting, control of the bus.
- Peripheral Functions—Interrupt-requests and timer overflow.
- Boot ROM Size—Input signals used to identify the boot ROM size.
- Boundary-Scan—JTAG-compatible test signals used for board verification.

4.1.1 Processor Control and Status

Signal	Function
CLKOUT1 CLKOUT2	CLOCK OUTPUTS (O): MB86933 bus transactions can be referenced against these outputs. CLKOUT1 has the same frequency and phase as the internal oscillator, or the signal applied to CLKIN. CLKOUT2 is the same as CLKOUT1, but phase-shifted 180 degrees.
–ERROR	ERROR SIGNAL (O): Asserted by the CPU to indicate that it has halted in an error state as a result of encountering a synchronous trap while traps are disabled. In this situation, the CPU saves the Trap Type (tt) value in the Trap Base Register, enters into an error state and asserts the –ERROR signal. The system can monitor the –ERROR pin and initiate a reset to recover from the error condition.
–RESET	SYSTEM RESET (I): Resets the processor to a known internal state. –RESET should be asserted for at least 4 processor cycles after the clock has stabilized. The internal state of the processor immediately after reset is described in the <i>Programmer's Model</i> chapter.
XTAL1 (CLKIN) XTAL2	EXTERNAL OSCILLATOR (XTAL1, XTAL2): Determines the execution rate and timing of the processor. Connecting a crystal across these pins forms a complete crystal oscillator circuit. The processor operating frequency is the same as the crystal oscillator frequency. The processor can also be driven by an external clock. In this case, the clock signal is applied to XTAL1 (CLKIN); XTAL2 should be left unconnected. The processor operating frequency is the same as the external clock frequency.

4.1.2 Memory Interface

Signal	Function																				
ADR[27:2]	ADDRESS BUS (O): Specifies the data or instruction address of a 32-bit word. Reads are always one word in size while byte, half-word, or word transaction sizes for writes are identified by separate byte-enable signals (–BE3-0). The value on the address bus is valid for the duration of the bus transaction.																				
–AS	ADDRESS STROBE (O): Asserted by the MB86933 or other bus master to indicate the start of a new bus transaction. A bus transaction begins with the assertion of –AS and ends with the assertion of –READY. During cycles in which neither the processor nor another bus master is driving the bus, the bus is idle, and –AS remains de-asserted. See Table C4-1 for signal values while the bus is idle. The MB86933 asserts –AS for 1 clock cycle.																				
ASI[3:0]	<p>ADDRESS SPACE IDENTIFIERS (O): Indicates which of the 16 available address spaces the current bus transaction is accessing. The ASI values are defined as follows:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>ASI <3:0></th> <th>ADDRESS SPACE</th> </tr> </thead> <tbody> <tr> <td>0x0</td> <td>Application Definable</td> </tr> <tr> <td>0x1</td> <td>Control Registers</td> </tr> <tr> <td>0x2 - 0x3</td> <td>Reserved</td> </tr> <tr> <td>0x4 - 0x7</td> <td>Application Definable</td> </tr> <tr> <td>0x8</td> <td>User Instruction Space</td> </tr> <tr> <td>0x9</td> <td>Supervisor Instruction Space</td> </tr> <tr> <td>0xA</td> <td>User Data Space</td> </tr> <tr> <td>0xB</td> <td>Supervisor Data Space</td> </tr> <tr> <td>0xC - 0xF</td> <td>Reserved</td> </tr> </tbody> </table> <p>The ASI values specified as “application definable” can be used by privileged (supervisor mode) instructions such as load and store alternate. The ASI value is available in the same cycle in which the corresponding address value is asserted on the address bus. The values on the ASI pins are valid for the duration of the bus transaction.</p>	ASI <3:0>	ADDRESS SPACE	0x0	Application Definable	0x1	Control Registers	0x2 - 0x3	Reserved	0x4 - 0x7	Application Definable	0x8	User Instruction Space	0x9	Supervisor Instruction Space	0xA	User Data Space	0xB	Supervisor Data Space	0xC - 0xF	Reserved
ASI <3:0>	ADDRESS SPACE																				
0x0	Application Definable																				
0x1	Control Registers																				
0x2 - 0x3	Reserved																				
0x4 - 0x7	Application Definable																				
0x8	User Instruction Space																				
0x9	Supervisor Instruction Space																				
0xA	User Data Space																				
0xB	Supervisor Data Space																				
0xC - 0xF	Reserved																				

Signal	Function																																																																				
-BE3-0	<p>BYTE ENABLES (O): These pins indicate whether the current store transaction is a byte, half-word or word transaction. -BE3-0 signals are available in the same cycle in which the corresponding address value is asserted on the address bus and is valid for the duration of the bus transaction. This bus should be used only to qualify store transactions. For load transactions all sub-word requests are read (and replaced in the cache) as words and then the appropriate byte or half-word is extracted by the integer unit.</p> <p>Possible values for -BE3-0 are as follows:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td></td> <td style="text-align: center;">31</td> <td style="text-align: center;">24</td> <td style="text-align: center;">23</td> <td style="text-align: center;">16</td> <td style="text-align: center;">15</td> <td style="text-align: center;">8</td> <td style="text-align: center;">7</td> <td style="text-align: center;">0</td> </tr> <tr> <td></td> <td colspan="3" style="text-align: center;">Byte 0</td> <td colspan="2" style="text-align: center;">Byte 1</td> <td colspan="2" style="text-align: center;">Byte 2</td> <td style="text-align: center;">Byte 3</td> </tr> <tr> <td>Byte Writes</td> <td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td> </tr> <tr> <td>Half-Word Writes</td> <td>1</td><td>1</td><td>0</td><td>0</td><td></td><td>0</td><td>0</td><td>1</td><td>1</td><td></td> </tr> <tr> <td>Word Writes</td> <td></td><td></td><td></td><td></td><td>0</td><td>0</td><td>0</td><td>0</td><td></td><td></td> </tr> </table> <p>BE<2:3> are also used in 8 and 16-bit ROM accesses as follows:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Bus Mode</th> <th>Byte</th> <th>BE<2:3></th> </tr> </thead> <tbody> <tr> <td rowspan="4">8-bit</td> <td>0</td> <td>0 0</td> </tr> <tr> <td>1</td> <td>0 1</td> </tr> <tr> <td>2</td> <td>1 0</td> </tr> <tr> <td>3</td> <td>1 1</td> </tr> <tr> <td rowspan="2">16-bit</td> <td>0 & 1</td> <td>0 0</td> </tr> <tr> <td>2 & 3</td> <td>1 0</td> </tr> </tbody> </table>		31	24	23	16	15	8	7	0		Byte 0			Byte 1		Byte 2		Byte 3	Byte Writes	1	1	1	0	1	1	0	1	1	1	Half-Word Writes	1	1	0	0		0	0	1	1		Word Writes					0	0	0	0			Bus Mode	Byte	BE<2:3>	8-bit	0	0 0	1	0 1	2	1 0	3	1 1	16-bit	0 & 1	0 0	2 & 3	1 0
	31	24	23	16	15	8	7	0																																																													
	Byte 0			Byte 1		Byte 2		Byte 3																																																													
Byte Writes	1	1	1	0	1	1	0	1	1	1																																																											
Half-Word Writes	1	1	0	0		0	0	1	1																																																												
Word Writes					0	0	0	0																																																													
Bus Mode	Byte	BE<2:3>																																																																			
8-bit	0	0 0																																																																			
	1	0 1																																																																			
	2	1 0																																																																			
	3	1 1																																																																			
16-bit	0 & 1	0 0																																																																			
	2 & 3	1 0																																																																			
-BMODE8	<p>8-BIT BOOT MODE: This signal is sampled during reset and causes read accesses memory mapped to -CS0 to assume 8-bit ROM memory. The MB86933 generates four sequential fetches to assemble a complete instruction or data word before continuing. Bytes are fetched in sequence (0,1,2,3) as encoded by -BE[2] and -BE[3] (00, 01, 02, 03). Writes to -CS0 are unaffected by boot mode selection. If left unconnected, a weak pull-up on this pin (and -BMODE16 pin) causes the processor to default to 32-bit mode.</p> <p><i>Note:</i> At reset, -BMODE8 must not be asserted while -BMODE16 is asserted, or undefined operation may result.</p>																																																																				
-BMODE16	<p>16-BIT BOOT MODE: This signal is sampled during reset and causes read accesses memory mapped to -CS0 to assume 16-bit ROM memory. The MB86933 generates two sequential fetches to assemble a complete instruction or data word before continuing. Half words are fetched in sequence (0,1) as encoded by -BE[2]. Writes to -CS0 are unaffected by boot mode selection. If left unconnected, a weak pull-up on this pin (and -BMODE8 pin) causes the processor to default to 32-bit mode.</p> <p><i>Note:</i> At reset, -BMODE16 must not be asserted while -BMODE8 is asserted, or undefined operation may result.</p>																																																																				
-CS[5-0]	<p>CHIP SELECTS (O): One of these signals is asserted when the value on the address bus lies in the range specified by the corresponding Address Range Specifier Register. The -CS signals are used to decode the current address into one of eight address ranges. Address ranges should not overlap. Each address range has a corresponding wait-state specifier which is used to generate an internal -READY signal after a user-defined number of processor clock cycles. This allows a variety of memory and I/O devices with different access times to be connected to the MB86933 without the need for additional logic. CS0 is enabled at reset (See Chapter 2).</p>																																																																				

Signal	Function
D[31:0]	<p>DATA BUS (I/O): D31 corresponds to the most significant bit of Byte 0. D0 corresponds to the least significant bit of byte 3. A double word is aligned on an 8-byte boundary, a word is aligned on a 4-byte boundary, and a half-word is aligned on a 2-byte boundary. If a load or store of any of these quantities is not properly aligned, a mem_address_not_aligned Trap will occur in the processor.</p> <p>During write cycles, the point at which data is driven onto the bus depends on the type of the preceding cycle. If the preceding cycle was a write, data is driven in the cycle immediately following the cycle in which --READY was asserted. If the preceding cycle was a read, data is driven one cycle after the cycle in which --READY was asserted, in order to minimize bus contention between the processor and the system.</p>
--LOCK	<p>BUS LOCK (O): Asserted by the processor to indicate that the current bus transaction requires more than one transfer on the bus. The Atomic Load Store instruction, for example, requires contiguous bus transactions and so causes the BUS LOCK signal to be asserted. The bus will not be granted to another bus master as long as --LOCK is active. --LOCK is asserted with the assertion of --AS and remains active until --READY is asserted at the end of the locked transaction</p>
--MEXC	<p>MEMORY EXCEPTION (I): Asserted by the memory system to indicate a memory error on either a data or instruction access. Assertion of this signal initiates either a Data or Instruction Access Exception trap in the IU. The current bus access is invalidated by asserting the --MEXC in the same cycle as the --READY signal. The IU ignores the value on the data bus in cycles where --MEXC is asserted.</p>
RD/ --WR	<p>READ/WRITE BUS TRANSACTION (O): Specifies whether the current bus transaction is a read or a write operation. When --AS is asserted and RD/--WR is high, then the current transaction is a read. With --AS asserted and RD/--WR low, the current transaction is a write. RD/--WR remains active for the duration of the bus transaction and is de-asserted with the assertion of --READY.</p>
--READY	<p>READY (I): Asserted by the external memory system to indicate that the current bus transaction is being completed and that it is ready to start with the next bus transaction in the following cycle. In case of a fetch from memory, the processor will strobe the value on the data bus at the rising edge of CLKIN following the assertion of --READY. In the case of a write, the memory system will assert --READY when the appropriate access time has been met.</p> <p>In most cases, no external logic is required to generate the --READY signal. On-chip circuitry can be programmed to assert --READY internally, based on the address of the current transaction. The external system can override the internal ready generator to terminate the current bus cycle early. Up to 6 address ranges each with different transaction times can be programmed. (See the <i>System Support Functions</i> section, below.)</p>
--SAME_PAGE	<p>SAME-PAGE DETECT (O): Asserted when the address of the current memory access is within the same page as the previous memory access. --SAME_PAGE can be used to take advantage of fast consecutive accesses within page-mode DRAM page boundaries. --SAME_PAGE is asserted with --AS and remains active for one processor cycle. --SAME_PAGE is never asserted in the first transaction following a transaction by another device on the bus. The page size is specified by writing the Same-Page Mask Register. (See the <i>System Support Functions</i> section, below.)</p>

4.1.3 Bus Arbitration

Signal	Function
–BGRNT	BUS GRANT (O): Asserted by the CPU in response to a request from a device wanting ownership of the bus. The CPU grants the bus to other devices only after all transfers for the current transaction are completed. All bus drivers are three-stated with the assertion of the BUS GRANT signal.
–BREQ	BUS REQUEST (I): Asserted by another device on the bus to indicate that it wants ownership of the bus. The request must be answered with a bus grant (–BGRNT) from the MB86933 before the device can proceed by driving the bus. Once the bus has been granted, the device has ownership of the bus until it de-asserts –BREQ. The user should ensure that devices on the bus do not monopolize the bus to the exclusion of the CPU. The assertion of –BREQ is recognized by the processor even when –RESET is being asserted.

4.1.4 Peripheral Functions

Signal	Function
IRL[3:0]	INTERRUPT REQUEST BUS (I): The value on these pins defines the external interrupt level. IRL[3:0]=1111 forces a non-maskable interrupt. An IRL value of 0000 indicates no pending interrupts. All other values indicate maskable interrupts as enabled in the Processor Interrupt Level field of the Processor Status Register (PSR). Interrupts should be latched and prioritized by external logic and should be held pending until acknowledged by the processor. An interrupt controller is available on the MB86940 peripheral chip. IRL inputs are sampled by the processor in cycle 1, synchronized in the following cycle, and recognized by the processor in the third cycle.
–TIMER_OVF	TIMER OVERFLOW (O): Indicates that the processor's internal 16-bit timer has overflowed. This signal can be used to initiate a DRAM refresh cycle or a one-cycle periodic waveform. On reset, the timer is turned off and –TIMER_OVF is high.

4.1.5 Test and Boundary-Scan

Signal	Function
–CLK_ECB	EXTERNAL CLOCK BYPASS (I): When tied high, causes the CLKIN signal to bypass the on-chip phase-locked loop. This signal is intended primarily for testing the chip.
TCK	TEST CLOCK (I): JTAG compatible test clock input.
TDI [†]	TEST DATA IN (I): JTAG compatible test data input.
TDO [†]	TEST DATA OUT (O): JTAG compatible test data output.
TMS [†]	TEST MODE (I): JTAG compatible test mode select pin.
–TRST [†]	TEST RESET (I): Asynchronous reset for JTAG logic. If not using JTAG, this signal must be pulled low.

†. See appendix for more information

4.2 Bus Operation

The Bus Interface Unit handles requests for external memory and I/O operations, arbitrates for bus access, or is idle. Bus transactions are handled as follows:

- **Memory and I/O Operations**—Read and write transactions are initiated with the processor asserting the -AS signal. The RD/-WR output indicates the transaction type. The $\text{-BE}[3:0]$ outputs indicate the transaction width. The processor drives the address and ASI signals and either drives (during stores) or reads (during loads) the signals on the data bus. The transaction ends when -READY is asserted.

An atomic load-store is a load followed immediately by a store, with no operation between. The -LOCK output is asserted during atomic operations to indicate that the bus is being used for more than one consecutive memory operation.

- **Arbitration**—Any external device can request ownership of the bus by asserting the -BREQ signal. The processor three-states its bus drivers and asserts -BGRNT to indicate that it is relinquishing control of the bus. Upon completion of its transaction, the external device de-asserts -BREQ , and the processor responds by de-asserting -BGRNT the following cycle.

In any cycle the BIU can receive a request for accesses to instruction memory, to data memory, or to both. If it receives a request for both in the same cycle, it completes the data memory transaction first.

4.2.1 Exception Handling

The external memory system can indicate an exception during a memory operation. The BIU signals the appropriate data or instruction exception to the IU, which will trap accordingly.

Any system that must recover from this error should store the address and data of the write operation in hardware. If the system can generate both read and write exceptions, the system must also provide a status bit that indicates whether the exception was generated during a read or during a write operation. With access to this information, the data access exception service routine can determine the cause of the exception and recover accordingly.

4.2.2 Bus Cycles

This section describes the relative timing of events in representative bus transactions.

Load

A read transaction begins with the BIU asserting -AS to indicate a new bus transaction. The -AS signal is de-asserted after one cycle. At the same time, $\text{ADR}<27:2>$ and $\text{ASI}<3:0>$ bits are asserted with the location to be read. The BIU drives the $\text{RD}/\text{-WR}$ signal high to indicate a read transaction.

Note that the -BE lines indicate byte, halfword or word operations during load operations, although their use is optional. The processor loads a word regardless of the size of the data requested (byte, halfword, word).

The external memory system responds with the read data on pins $\text{D}<31:0>$. It also asserts the -READY signal when the data is ready (unless internal ready generation is selected). For slow memory, the -READY signal is delayed until data is valid.

A load double operation is treated as back-to-back reads.

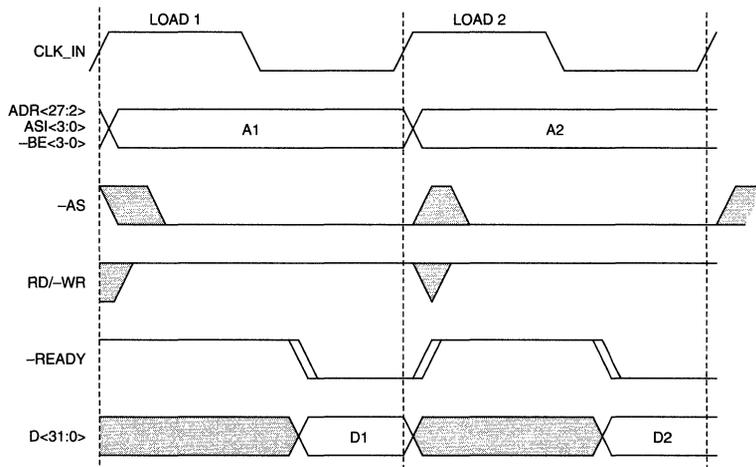


Figure C4-1. Load Timing

Load with Exception

If the external memory system sees a memory exception, it can terminate the current memory transaction by asserting the -MEXC and -READY signals. The data on the data bus is ignored by the MB86933.

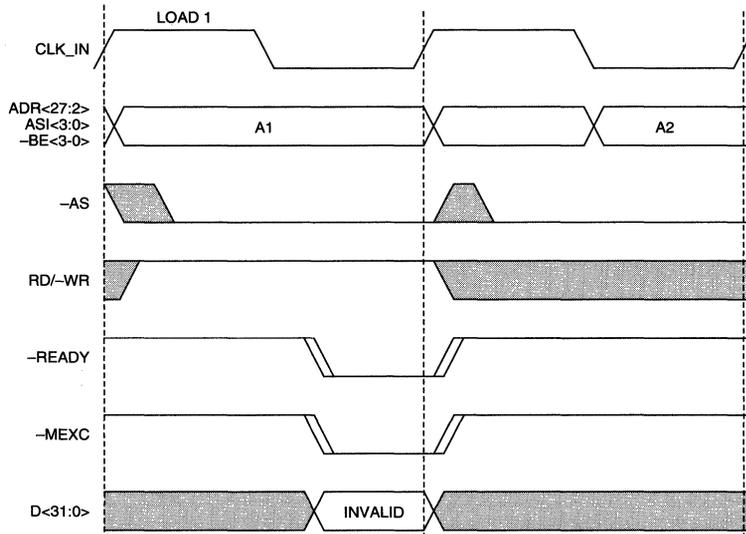


Figure C4-2. Load with Exception Timing

Store

A write transaction begins with the BIU asserting -AS , to indicate a new bus transaction. The -AS signal is de-asserted after one cycle. At the same time the $\text{ADR}\langle 27:2 \rangle$ and $\text{ASI}\langle 3:0 \rangle$ pins are driven with the location to be written, and the write data is asserted on $\text{D}\langle 31:0 \rangle$. The $\text{-BE}\langle 3:0 \rangle$ pins indicate byte, half-word or word transaction width. The BIU drives the $\text{RD}/\text{-WR}$ signal low to indicate a write transaction.

The external memory system responds by asserting the -READY signal when it has stored the data. There is always one idle bus cycle between the termination of a read cycle and the beginning of a write cycle to provide time for switching of the data bus drivers.

A store double operation is treated as back-to-back writes.

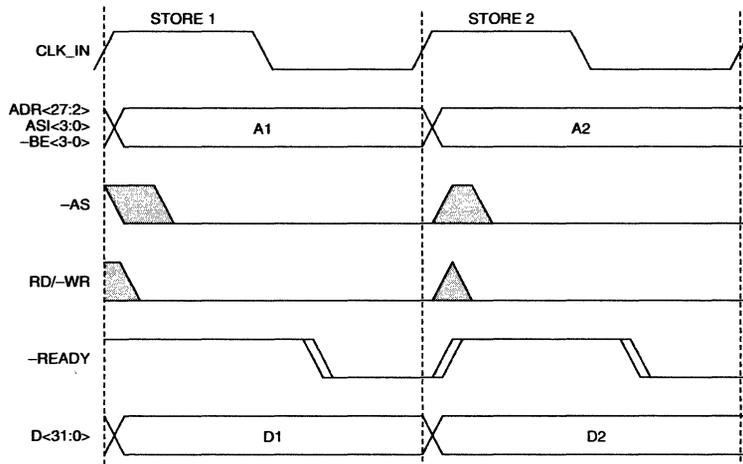


Figure C4-3. Store Timing

Store with Exception

If an access exception occurs during a write, the external memory system can terminate the current memory transaction by asserting the -MEXC and -READY signals. The external memory system is expected to ignore the data on the data bus in this situation.

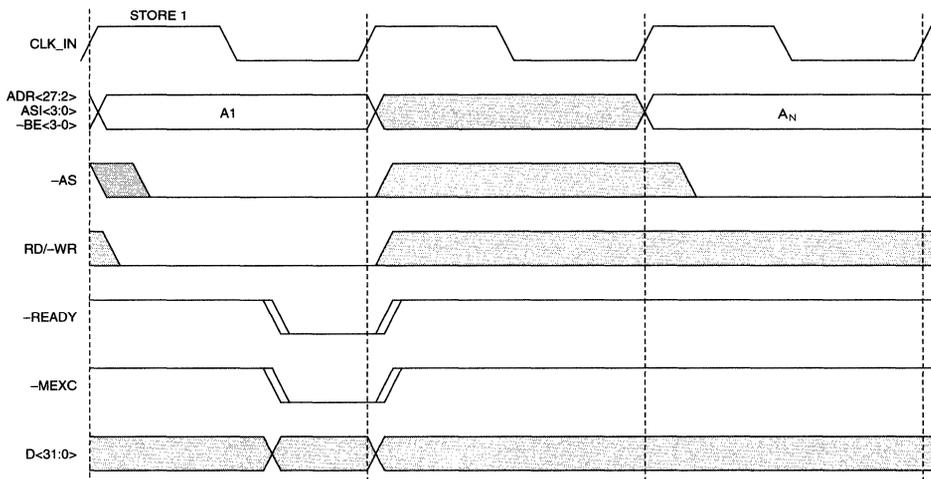


Figure C4-4. Store with Exception Timing

Atomic Load Store

An atomic load store executes as a load followed by a store, with no operation between. The -LOCK signal is asserted to indicate that the bus is being used for more than one external memory operation.

There is one cycle between the termination of the read and the beginning of the write to provide time for the switching of the data bus drivers.

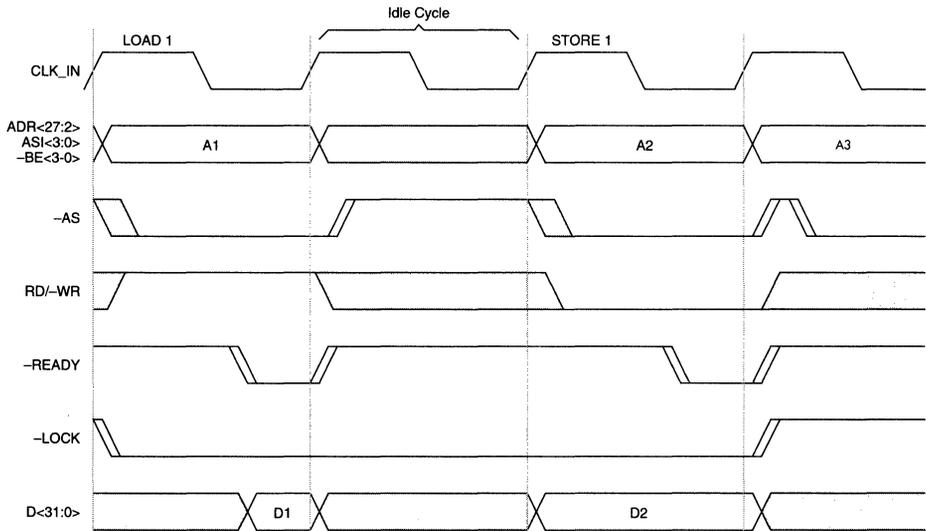


Figure C4-5. Atomic Load Store Timing

External Bus Request and Grant

Any external device can request ownership of the bus by asserting the -BREQ signal. The BIU asserts the -BGRNT signal to indicate that it is relinquishing control of the bus, and three-states all of its bus drivers. The external device can complete its transaction during the following cycle. Upon completion of its transaction, the external device de-asserts the -BREQ signal. The BIU responds by de-asserting the -BGRNT signal during the following cycle.

The MB86933 is the default owner of the bus.

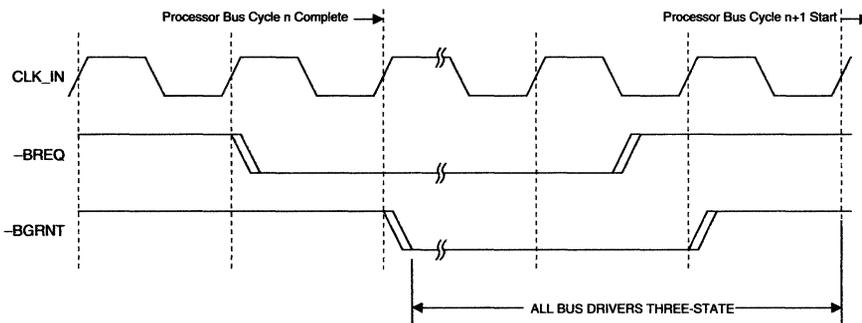


Figure C4-6. External Bus Request and Grant Timing

Processor Reset

The MB86933 is reset by asserting the -RESET signal for a minimum of 4 clock cycles (see Figure 4-7). Systems using an external crystal to clock the processor should assert -RESET for at least 4 cycles after the crystal has stabilized.

If the processor is reset following a halt in Error Mode and if power to the processor is not removed, after reset the tt field will contain the value of the Trap that caused the processor to halt.

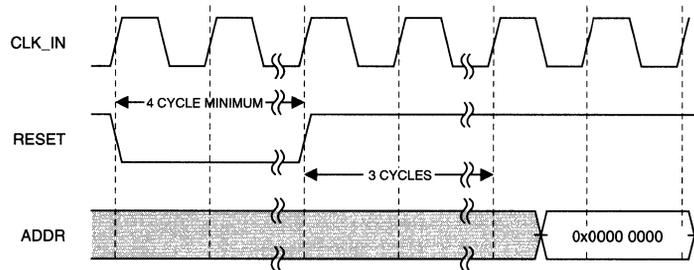


Figure C4-7. Reset Timing

4.3 System Support Functions

Built-in system support functions help to minimize the amount of glue logic required in the external system. The support includes programmable chip select logic, programmable wait-state generation, same-page detection logic and a timer for generating refresh requests. For a more detailed description of the programming of these registers refer to chapter 2.

The System Support Control Register turns the various system support features on and off.

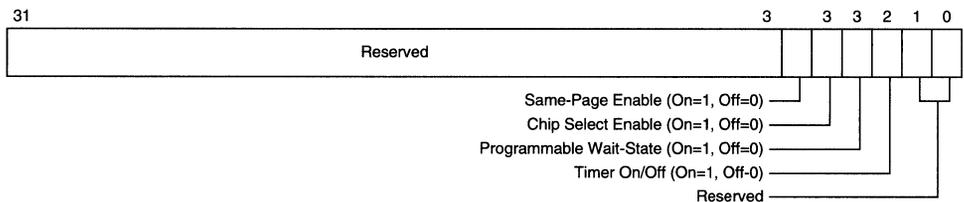


Figure C4-8. System Support Control Register

4.3.1 System-Configuration Registers

The system-configuration registers (Address Range Specifiers, Address Masks, and Programmable Wait-State Specifiers) allow software to define six different address ranges. When an address driven by the processor is in one of these ranges, the corresponding Chip-Select ($-\text{CS}$) pin is asserted. After a number of clock cycles determined by the corresponding Programmable Wait-State Specifier, the processor automatically generates an internal $-\text{READY}$ signal. This

makes it possible for memory and I/O devices with different access times to be connected to the processor without additional logic.

The contents of the Address Range Specifier Registers 1-5 (ARSR[5:0]) define five of the six address ranges. An additional address range is available, corresponding to -CS0 . For this address range, ADR is hardwired to 0, and ASI is hardwired to 0x9 (Supervisor Instruction Space). With Mask Register AMR0, -CS0 ranges 8K words. -CS0 is enabled at reset. -CS1 , -CS2 , -CS3 , -CS4 and -CS5 are disabled at reset.

Note that the MB86933 has no caches, no write buffer, no pre-fetch buffer, and has six register windows rather than eight. It has twenty-six Address Bus signals (ADR<27:2>) rather than thirty, four Address Space Identifier signals (ASI<3:0>) rather than eight, no emulator-support signals, and no memory management unit. These and other differences between the MB86933 and other SPARClite processors should be considered when porting code to the MB86933 from another SPARClite processor, and when porting code from the MB86933 to another SPARClite processor. Documentation for other SPARClite should be referenced to identify differences with the MB86933 that may affect ported code.

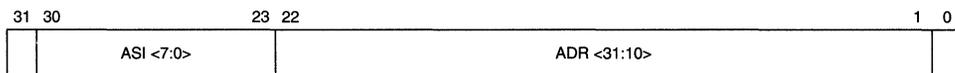


Figure C4-9. Address Range Specifier Register Format

An Address Mask Register is associated with each address range. Any address driven by the chip is compared with the value in all address range specifiers. Only those bits of the register are compared for which the corresponding mask bits are 0. If the specified bits of the current address match one of the address range specifiers, the corresponding chip-select (-CS) pins are asserted. When no bus transaction is being performed, all the -CS pins are high (inactive). The Address Mask Register corresponding to -CS0 is initialized to compare all bits except ADR<14:10>.

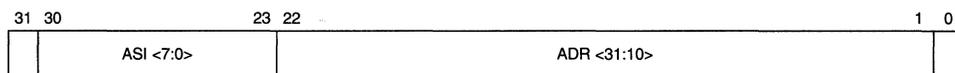
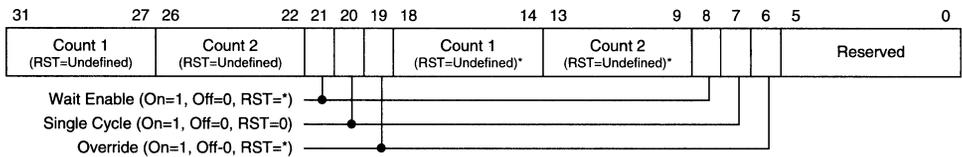


Figure C4-10. Address Mask Register Format

A Programmable Wait-State Specifier is associated with each address range. Three registers are used to specify the wait states for the six address ranges. Each register contains the wait-state specifiers for two address ranges.

When the address currently being driven by the processor matches the unmasked bits in one of the Address Range Specifiers, the corresponding wait-state specifier is selected. The format of Wait-State Specifier Registers is shown in Figure C4-11.



* See Table 2-3 in *MB86930 Chap 2 "Programmer's Model"*

Figure C4-11. Wait-State Specifier Registers

Bits 31-19: Wait-State Specifier—When an external access falls within an address range defined by an ARSR and AMR, the corresponding wait-state specifier determines when, and whether, the processor generates an internal --READY signal to terminate the access.

Count1 (Bits 31-27): The number of wait-states inserted before the internal --READY , under the following conditions: the Single Cycle bit equals 0 and the current access is not on the same page as the previous access. The number of wait-states is the value of this field +1 (i.e., 0=1 wait-state, 1=2 wait-states, etc.) The value of Count1 is undefined on reset.

Count2 (Bits 26-22): The number of wait-states inserted before the internal --READY , under the following conditions: the Single Cycle bit equals 0 and the current access is on the same page as the previous access. The number of wait-states is the value of this field +1 (i.e., 0=1 wait-state, 1=2 wait-states, etc.) The value of Count2 is undefined on reset.

Wait Enable (Bit 21): Enables and disables the wait-state generator for an individual address range. If the Wait Enable bit of a wait-state specifier equals 0, the internal --READY is not asserted when addresses in the corresponding range are accessed by the processor. If Wait Enable is 1, the single cycle bit must be 0. See Table 2-3 in *MB86930 Chap 2 "Programmer's Model"* for reset value.

Single Cycle (Bit 20): Specifies the timing of the internal --READY signal. If the Single Cycle bit equals 1 when an address in the appropriate range is accessed, the internal --READY is asserted in the same cycle. If the Single Cycle bit equals 0, and the current transaction is in the same page as the previous transaction, then Count2 is used as the number of cycles after which --READY is asserted internally. If the transaction is not in the same page, Count1 is used instead. If Single Cycle is enabled, the Wait Enable bit must be 0. See Table 2-3 in *MB86930 Chap 2 "Programmer's Model"* for reset value.

Override (Bit 19): Allows the system to terminate a memory transaction before the internally specified time. If the Override bit equals 1, and external hardware asserts the external --READY signal, then the wait-state generator will stop counting and will wait for the next transaction. This bit is cleared to 0 on reset.

Bits 18-6: Wait-State Specifier—The wait-state specifier for a second address range. This field is organized just like bits 31-19.

Bits 5-0: Reserved

The Count1 and Count2 fields of the Wait-State Specifier corresponding to -CS0 have all their bits set to 1 following reset. In this way, 32 wait-state cycles (the maximum number) are inserted into the processor's first instruction accesses. The override bit for -CS0 is enabled as well.

4.3.2 Same-Page Detection

The same-page detection logic determines whether the address of the current memory transaction is on the same page as the previous transaction. If it is, the processor asserts the -SAME_PAGE signal. The system can then take advantage of the fast consecutive accesses possible within fast-page mode DRAM page boundaries. The same-page detection logic consists of a mask register, a register to store the address and ASI bits of the previous transaction, and a comparator.

The Same-Page Mask Register specifies which bits of the current address and ASI must be compared with the previous address and ASI. Only those bits are compared for which the mask bit is 1.



Figure C4-12. Same-Page Mask Register

The -SAME_PAGE signal is never asserted for the first transaction following a transaction by another device on the bus. When using the internal wait-state generator, DRAM control logic should issue a bus request when initiating a refresh cycle so that the -SAME_PAGE logic is reset appropriately. The -SAME_PAGE feature is disabled at reset.

4.3.3 Programmable Timer

The 16-bit programmable timer causes the -TIMER_OVF output signal to be asserted at software-defined intervals. This signal can be used to initiate DRAM refresh cycles, or to control other periodic events in the external system.

The current timer count is stored in the Timer Register. When the timer overflows, it is loaded with the value in the Timer Preload Register. The contents of both of these registers are undefined following reset.

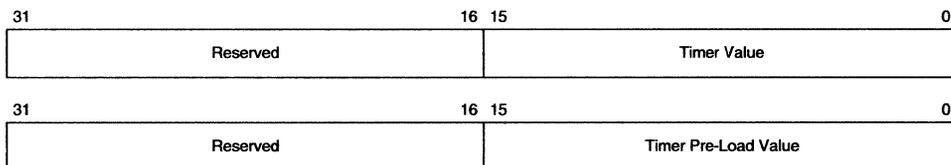


Figure C4-13. Timer and Timer Preload Registers

The timer can also be loaded by writing directly to the Timer Register. The timer can be turned off by writing a 0 to the Timer On/Off bit in the System Support Control register. The timer is clocked at the processor clock frequency.

4.4 ROM Interface

4.4.1 Purpose

The data bus of the MB86933 can be configured upon reset to 8- and 16-bit bus modes as well as the standard 32-bit mode. This flexibility accommodates those cases in which boot code resides in PROMs organized as blocks of bytes or half-words.

4.4.2 Features

Bus Configuration: the data bus configurations are fixed to specific segments of the bus:

- 8-bit mode: D[7:0]
- 16-bit mode: D[15:0]
- 32-bit mode: D[31:0]

4.4.3 Bus Configuration on Reset

Two external pins, -BMODE16 and -BMODE8 are used to determine the bus configuration. The two bus configuration pins have weak pull-ups, so that if unconnected, the bus configuration will default to a 32-bit bus.

(reserved): $\text{-BMODE16}=0, \text{-BMODE8}=0$

8-bit mode: $\text{-BMODE16}=1, \text{-BMODE8}=0$

16-bit mode: $\text{-BMODE16}=0, \text{-BMODE8}=1$

32-bit mode: $\text{-BMODE16}=1, \text{-BMODE8}=1$

4.4.4 System Interface

In order to minimize external "glue logic" required for interfacing to the 8- or 16-bit bus, the BE bits are encoded to reflect the two LSBs of a byte address or the LSB of a halfword address. Therefore, the $\text{ADR}[27:2]$ and selected -BE bits can be concatenated to form a complete address for a non-32 bit bus mode.

Table C4-2: System Interface BE Bits

Bus Mode	Byte	BE[0:3]
8-bit bus	0	0000
	1	0001
	2	0010
	3	0011
16-bit bus	0 & 1	0000
	2 & 3	0010

8-bit bus mode address= $\{\text{ADR}[27:2], \text{-BE}[2], \text{-BE}[3]\}$

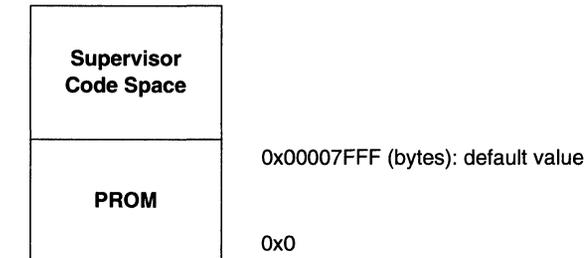
16-bit bus mode address= $\{\text{ADR}[27:2], \text{-BE}[2]\}$

$\text{-CS}[0]$, which is enabled on reset, and the internal -READY generation logic, can be used to minimize any glue logic required to interface to the PROM. On reset, the wait state generator, corresponding to $\text{-CS}[0]$ for internal -READY generation, is set to 32 cycles. Later on in the boot code, the wait state generator can be changed to a more appropriate value.

4.4.5 PROM Address Space

The PROM address space is defined by the $-CS[0]$ address-range specifier. On reset, the $-CS[0]$ address range defaults to 32K bytes (starting address=0x0), and the ASI is initialized to 0x9. The PROM address range can be changed later using the mask bit register associated with $-CS[0]$. An example of the supervisor address space (ASI=0x9) memory map is shown below:

Figure C4-14. Supervisor Address Space (ASI=0x9) Memory Map



Any memory access from the PROM address space, in a non-32 bit mode, will make the $-BE$ bit encodings reflect the LSBs of a byte/halfword address. Furthermore, the fetched bytes/halfwords will be assembled into a 32-bit word. On the other hand, any access from the non-PROM address range will result in a normal, 32-bit memory access.

4.4.6 Load/Stores

One of the functions of the boot code is to set the processor and system configuration. This might involve loading system parameters from PROM, loading data from memory mapped I/O, and storing data to non-PROM address space. All loads from the PROM address space behave the same way as instruction fetches, in that, for a non-32 bit bus mode $-BE$, bit encoding and word assembly are done. Loads from a non-PROM address space behave in the normal (32-bit) manner. In order to meet the $-BE$ AC timing, the $-BE$ bits on the MB86933 need to be all 0's for all types of loads—word, halfword, and byte—from the non-PROM address space. This requires a functional change from the current specification of the MB86930's $-BE$ bits, which reflect the byte information for loads. This change does not cause a problem, since the processor fetches a full 32-bit word on a load, and the IU selects the byte appropriately. As on the MB86930 $-BE$ bits should be ignored for 32-bit loads.

Since stores to the PROM will never occur, for all stores, regardless of address space, the $-BE$ bits will reflect the byte information of the store. Therefore, byte and halfword stores to the PROM address space becomes meaningless, since the

-BE[2] and -BE[3] bits no longer reflect the byte address. Furthermore, store word operations to the PROM address space will not result in a dis-assembly process for a non-32 bit bus mode. Since stores to PROM address space are not disabled, the user would have to qualify -CS[0] with the R/-W signal to use it as a PROM chip select signal. This will not be necessary if the user can be sure that a store to PROM space never occurs.

A summary of the -BE[0:3] bit behavior for loads from the PROM address space is shown below. For all load instructions (byte, halfword, word), a full 32-bit fetch occurs. For example, in the 8-bit bus mode, four bytes will be fetched for all loads, and the BE bits will sequence with the proper 2 LSBs of the byte address.

Table C4-3: Load -BE[0:3] Bit Behavior

Bus Mode	Operation	BE[0:3] in PROM space
8-bit bus	Loads (all)	0000=>0001=>0010=>0011
16-bit bus	Loads (all)	0000=>0010
32-bit bus	Loads (all)	0000

4.4.7 Memory Exception

Any memory exception that occurs during a fetch from the PROM address space in a non-32 bit bus mode will be held off until the entire word is fetched.

4.4.8 Bus Request

Any bus request happening during the non-32 bit bus mode fetch will not be recognized until the end of the complete 32-bit fetch operation.

4.4.9 Timing

Timing examples for the 8- and 16-bit bus modes with 1 wait-state memory are shown below. Note that -AS is asserted at the beginning for one cycle.

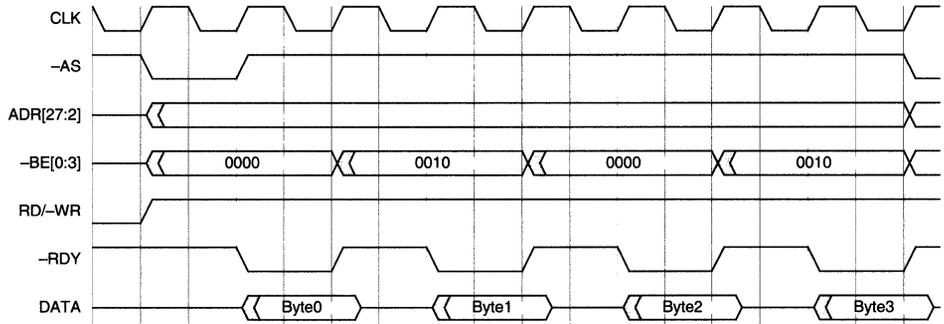


Figure C4-15. 8-bit Bus Mode Read (1 Wait State)

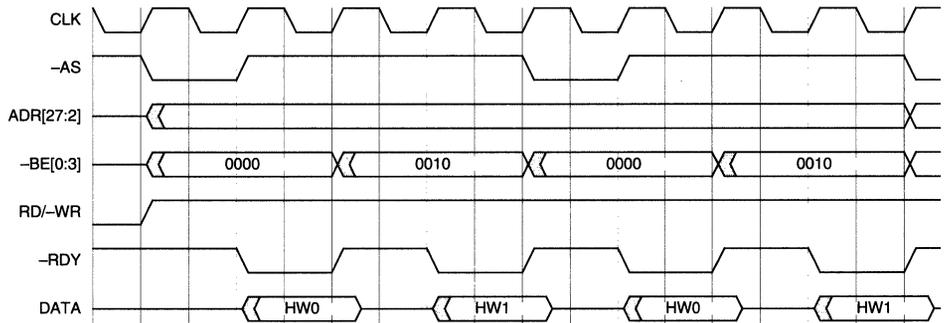


Figure C4-16. 16-bit Bus Mode Read (1 Wait State)

4.4.10 Store in 8/16 Bit

For all stores, regardless of address space, the -BE bits will reflect the byte information of the store. The following note may be useful for system designers.

Store Byte: All 4 bytes are the same in the whole word data (i.e., $D[31:24] = D[23:16] = D[15:8] = D[7:0]$).

Byte	$\text{-BE}\{3:0\}$
0	1110
1	1101
2	1011
3	0111

Store halfword: 2 half word are the same in the whole word data (i.e., $D[31:16] = D[15:0]$).

Half Word	$\text{-BE}\{3:0\}$
0	1100
1	0011

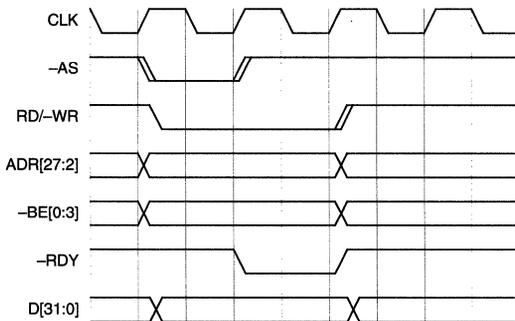


Figure C4-17. Store to 8/16-Bit Address Space

CHAPTER C5

Programming Considerations

5.1 MB86933 Programming Information

Chapter 5 of the main section of this manual contains programming information for the SPARClite processors that applies specifically to the MB86930 processor.

The MB86933, however, has no caches, has six register windows rather than eight, and differs from the MB86930 processor in other ways (see the Overview section of this addendum). Therefore, information given in Chapter 5 relating to features that are not supported by the MB86933 should be disregarded. The Chapter should be referenced only for programming information that is appropriate for the MB86933.

CHAPTER C6

System Design Considerations

This chapter describes SRAM and page-mode DRAM interfacing to the MB86933 processor, and MB86933 in-circuit emulation. Chapter 6 of this manual describes system design considerations for SPARC*lite* processors in more detail.

6.1 Interfacing SRAM

The address bus, data bus, and chip select signals of the SRAM can be connected directly to the address bus, data bus, and a chip select of the processor. The output enable signal can be generated by gating RD/*-*WR high and Chip select low to produce output enable low. Write enable for the SRAMs requires more consideration.

The processor data hold time for a write is specified as zero hold after the rising edge of the clock. RD/*-*WR hold time at the end of a write operation can be 0 after the rising edge of the clock, or can be held low if the next cycle is also a write. Thus an implementation cannot use RD/*-*WR directly as *-*WE for the SRAMs.

Figure C6-1 shows timing for an typical system using 2 cycle access SRAM operating at 20 MHz. Individual -WE signals are generated for each of the 4 bytes in the data word.

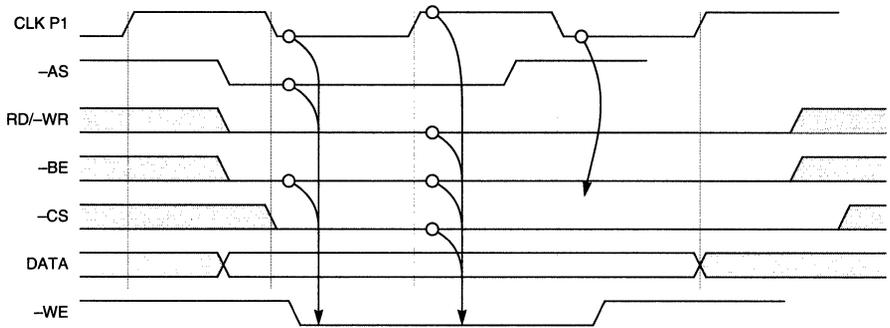


Figure C6-1. SRAM Interfacing Example

The SRAM is controlled with a PAL using the following equations:

```

!clkd = !clkp1;
!soe_ = rw & !scs_;
!swe3_ = !rw & !as_ & !be3_ & !clkp1
    # !rw & !as_ & !be3_ & !clkd
    # !rw & !scs_ & !swe3_ & clkp1
    # !rw & !scs_ & !swe3_ & clkd;

!swe2_ = !rw & !as_ & !be2_ & !clkp1
    # !rw & !as_ & !be2_ & !clkd
    # !rw & !scs_ & !swe2_ & clkp1
    # !rw & !scs_ & !swe2_ & clkd;

!swe1_ = !rw & !as_ & !be1_ & !clkp1
    # !rw & !as_ & !be1_ & !clkd
    # !rw & !scs_ & !swe1_ & clkp1
    # !rw & !scs_ & !swe1_ & clkd;

!swe0_ = !rw & !as_ & !be0_ & !clkp1
    # !rw & !as_ & !be0_ & !clkd
    # !rw & !scs_ & !swe0_ & clkp1
    # !rw & !scs_ & !swe0_ & clkd;

```

Clock low, -AS lo, -BE low, and $\text{RD}/\text{-WR}$ low cause -WE to be asserted. Clock high, -CS low, -BE low and $\text{RD}/\text{-WR}$ low cause -WE to stay low. When clock goes low again, -WE is negated. This way there is sufficient data hold time.

For this system, CLKOUT1 from the processor was used because it has better duty cycle control than an oscillator clock.

6.2 Interfacing Page-Mode DRAM

Interfacing Dynamic RAM requires a DRAM controller for generating RAS and CAS (Row Address Strobe and Column Address Strobe), and for handling refresh. The DRAM controller is typically implemented as a state machine. The DRAM controller and signal interfaces should be designed carefully to accommodate refresh operations and fast page mode access.

The programmable 16-bit timer provided in the MB86933 processor core can be used for timing the refresh interval. The timer output signal, -TIMER_OVF (Timer Overflow), goes low for a single clock cycle at the end of each timer interval. The timer interval is programmed in software, with the correct time interval depending on how the refresh operation is implemented.

The correct number of wait states can be generated by either the processor's internal wait-state generator, or the DRAM controller.

The processor supports fast “page mode” access to DRAM. When the current DRAM address is within the same page as the previous DRAM access, the `-SAME_PAGE` (Same-Page Detect) signal is asserted. This tells the DRAM controller that DRAM can be accessed using CAS only without selecting a new row of the DRAM, saving time. Page-mode accesses thus provide timing advantages comparable to the burst-mode accesses of some other processors.

To take advantage of page hits, RAS is asserted and left asserted to continuously select a row. CAS is asserted one access at a time to select a memory location in that row. Accesses need not be in consecutive locations. RAS can remain asserted as long as each access is in the same row, and CAS can be asserted once to access each memory location. RAS remains asserted between accesses.

The wait-state generator can be programmed to use a different (smaller) number of clock cycles for a “page hit” (when the current address is within the same page as the previous DRAM access).

When using the internal wait-state generator instead of the external `-READY` signal, the processor has no way of detecting a refresh operation that occurs during an access. One solution is to have the DRAM controller take control of the bus during refresh using `-BREQ` (Bus Request), thereby preventing the processor from requesting a memory access for the duration of the refresh operation. The disadvantage of this solution is that the processor is forced to remain idle. An alternative solution is to disable the internal wait-state generator and let the DRAM controller generate the `-READY` signal for all DRAM accesses.

Figure C6-2 is a simplified state diagram for a DRAM memory controller. Upon reset, the state machine starts in the RAS Precharge and Idle state, and remains in that state until a memory access or refresh request occurs.

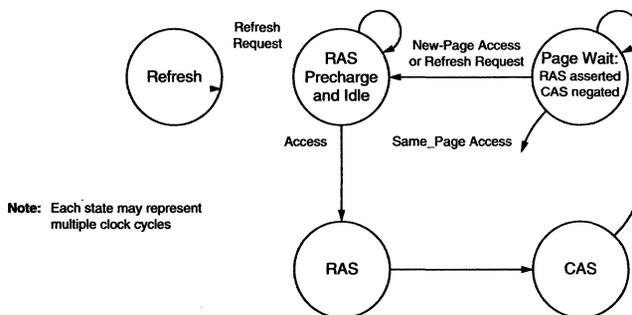


Figure C6-2. Simplified State Diagram for DRAM Controller

If a refresh request occurs, the state machine goes into the Refresh state. (In practice, this will actually be a number of sequential states.) When the refresh operation is complete, the state machine returns to the RAS Precharge and Idle state.

When the processor requests a DRAM memory access, the state machine enters the RAS state, in which the RAS signal is asserted to select the row. From there it goes to the CAS state, in which the CAS signal is asserted to select the column. At this point, data is clocked into the appropriate part, and the bus cycle ends.

From there the state machine enters the Page Wait state, in which the state machine waits for either another memory access, or a refresh request. In this state, RAS is asserted and CAS is negated. If there is a memory access to the same page of DRAM (as indicated by the `-SAME_PAGE` signal), the state machine goes directly to the CAS state, and CAS is asserted to select the memory location. If there is a memory access to a different page of DRAM or if a refresh request occurs, the state machine goes to the RAS Precharge and Idle state, then to the requested operation. The state machine waits with RAS asserted until one of these events occurs.

For more information, refer to SPARClite Application Note #1, which describes DRAM interfacing.

6.3 In-Circuit Emulation

The MB86932 processor supports all MB86933 functions and signals, and can be used for in-circuit emulation of the MB86933.

The MB86932 processor has ten pins that are used for in-circuit emulation: four emulator status/data bits, four emulator data bits, an emulator break request line, and an emulator enable pin.

To allow for compatibility with an in-circuit emulator, the system's reset circuit should be designed to allow the in-circuit emulator to take control of the `-RESET` signal. For example, a jumper in the `-RESET` input line close to the processor can be included, allowing the normal Reset circuit to be easily disconnected from the processor.

To simplify the task of emulating the processor, it is recommended that the processor's emulator pins be connected to a standard format connector. Access to these pins allow the emulator to take full control of the processor, as well as to trace processor activity. If this socket is included on production boards, an emulator can be used for board diagnostics and maintenance later in the product life cycle.

For more information contact Fujitsu Microelectronics Semiconductor Division or your emulator vendor.

CHAPTER C7

Instruction Set

7.1 MB86933 Instruction Set

The MB86933 processor supports the same instruction set as the MB86930 processor. Chapter 7 of the main section of this manual therefore fully describes the MB86933 instruction set.

Note that the MB86933 has six register windows rather than eight. Therefore, references to eight register windows in the description should be changed to six register windows for the MB86933, and *modulo 8* in the description should be changed to *modulo 6*.

Table C8-1: JTAG Pin Order

Order	JTAG Cell	JTAG Cell Type	Function
:			:
93	D_i<0>	input	Input bit 0 of <31:0> bus
94	D_o<0>	output	Output bit 0 of <31:0> bus
95	dbusiojo	output	D<31:0> bus bidirectional control signal dbusiojo = 1: D<31:0> bus is an input dbusiojo = 0: D<31:0> bus is an output
96	tstatejo	output	Three-state control signal If tstatejo=1 then the following pins are three-stated. ADR<27:2>, ASI<3:0>, -BE<3:0>, -AS, -RD/WR, -LOCK
97	-MEXC	input	Memory exception input
98	-READY	input	External memory transaction complete signal
99	-BREQ	input	Bus request input
100	-AS	output	Start of memory transaction output signal
101	-RD/WR	output	Memory Read/Write output signal
102	-LOCK	output	Bus lock output signal
103	-BGRNT	output	Bus grant output signal
104	-ERROR	output	Error output signal
105	-SAME_PAGE	output	Same-Page output signal
106	-CS<0>	output	LSB of chip select output signal
:			:
111	-CS<5>	output	MSB of chip select output signal
112	CLK_ENB	input	PLL control pin. CLK_ENB=1: PLL on CLK_ENB=0: PLL off
113	XTAL1	input	Crystal input
114	-TIMER_OVF	output	Timer Overflow pin
115	-BE<0>	output	Byte 0 enable output signal
:			:
118	-BE<3>	output	Byte 3 enable output signal
119	ASI<0>	output	LSB of ASI output pins
:			:
122	ASI<3>	output	MSB of ASI output pins
123	-RESET	input	Chip reset pin

Table C8-1: JTAG Pin Order

Order	JTAG Cell	JTAG Cell Type	Function
124	–BMODE8	input	8-bit Boot Mode
125	–BMODE16	input	16-bit Boot Mode

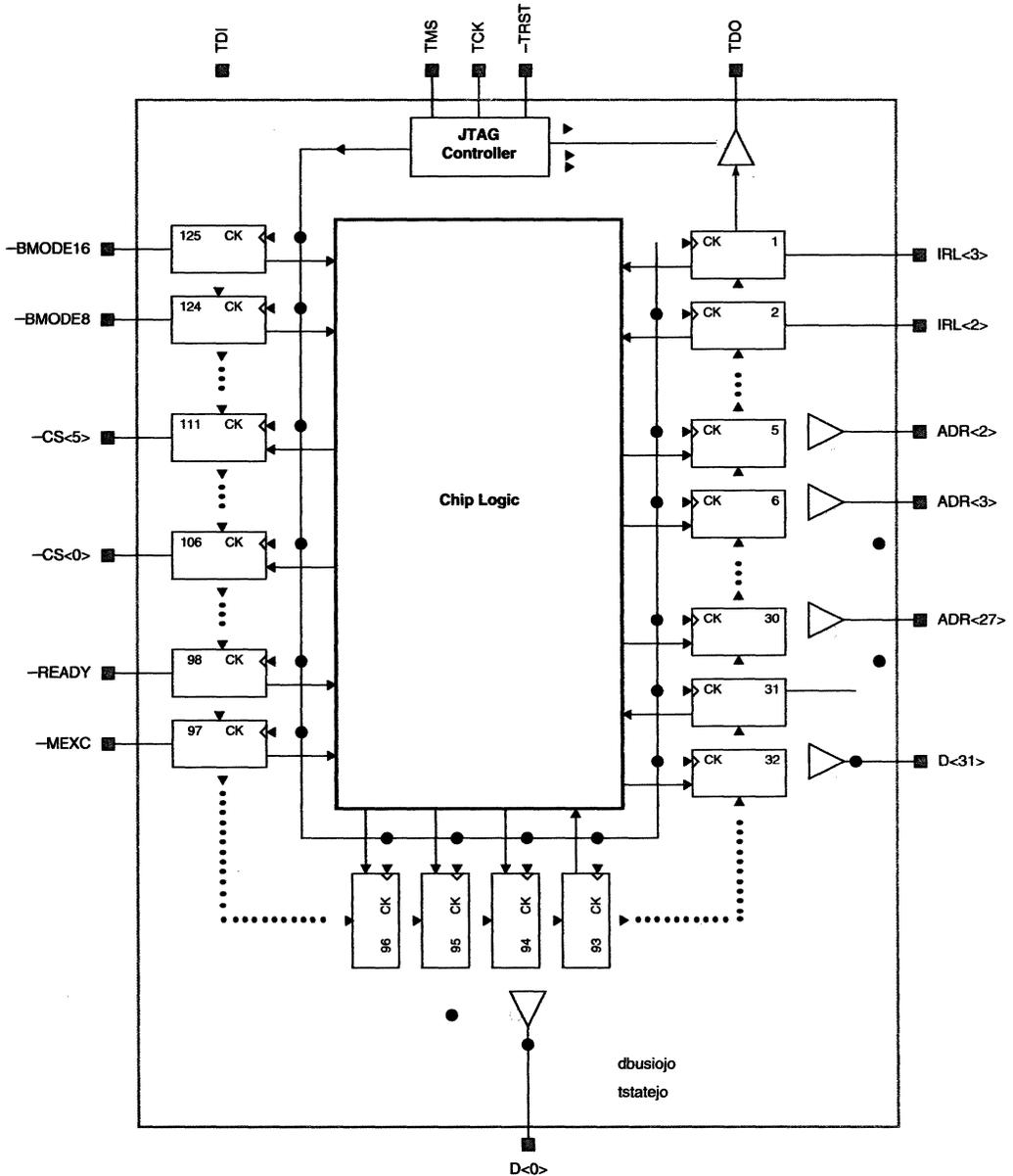


Figure C8-1. JTAG Cell Organization

FUJITSU MICROELECTRONICS, INC. SALES OFFICES

CALIFORNIA

2880 Lakeside Drive, Ste 250
Cupertino, CA 95014
(408) 996-1600

Century Center
2603 Main Street, #510
Irvine, CA 92714
(714) 724-8777

COLORADO

5445 DTC Parkway, P4
Englewood, CO 80111
(303) 740-8880

GEORGIA

3500 Parkway Lane, #210
Norcross, GA 30092
(404) 449-8539

ILLINOIS

One Pierce Place, #910
Itasca, IL 60143-2681
(708) 250-8580

MASSACHUSETTS

75 Wells Avenue, #5
Newton Center, MA 02159-3251
(617) 964-7080

MINNESOTA

3460 Washington Drive, #209
Eagan, MN 55122-1303
(612) 454-0323

NEW YORK

898 Veterans Memorial Hwy.
Building 2, Suite 310
Hauppauge, NY 11788
(516) 582-8700

OREGON

15220 N.W. Greenbrier Pkwy.,
#360
Beaverton, OR 97006
(503) 690-1909

TEXAS

14785 Preston Rd., #670
Dallas, TX 75240
(214) 233-9394

For further information outside the U.S., please contact:

ASIA

Fujitsu Microelectronics Pacific Asia Ltd.
616-617, Tower B, New Mandarin Plaza,
14 Science Museum Rd., Tsimshatsui East,
Kowloon, Hong Kong
Tel: 723-0393 • Fax: 721-6555

Fujitsu Limited
Semiconductor Marketing
Furukawa Sogo Building
6-1 Marunouchi, 2-chome
Chiyoda-ku, Tokyo 100, Japan
Tel: 03-3216-3211 • Fax: 03-3216-9771

Fujitsu Microelectronics Pacific Asia Ltd.
1906, No. 333 Keelung Rd., Sec. 1,
Taipei, 10548, Taiwan, R.O.C.
Tel: 02-7576548 • Fax: 02-7576571

Fujitsu Microelectronics PTE Ltd.
51 Bras Basah Rd.
Plaza by the Park
#06-04/07 Singapore 0718
Tel: 336-1600 • Fax: 336-1609

EUROPE

Fujitsu Mikroelektronik GmbH
Immeuble le Trident
3-5 voie Felix Eboué
94024 Creteil Cedex, France
Tel: 01-42078200 • Fax: 01-42077933

Fujitsu Mikroelektronik GmbH
Am Siebenstein 6-10
6072 Dreieich-Buchsschlag, Germany
Tel: 06103-6900 • Fax: 06103-690122

Fujitsu Mikroelektronik GmbH
Carl-Zeiss-Ring 11
8045 Ismaning, Germany
Tel: 089-9609440 • Fax: 089-96094422

Fujitsu Mikroelektronik GmbH
Am Joachimsberg 10-12
7033 Herrenberg, Germany
Tel: 07032-4085 • Fax: 07032-4088

Fujitsu Microelectronics Italia, S.R.L.
Centro Direzionale Milanofiori
Strada 4-Palazzo A/2
20094 Assago (Milano), Italy
Tel: 02-8246170/176 • Fax: 02-8246189

Fujitsu Mikroelektronik GmbH
Europalaan 26A
5623 LJ Eindhoven, The Netherlands
Tel: 040-447440 • Fax: 040-444158

Fujitsu Microelectronics Ltd.
Torggatan 8
17154 Solna, Sweden
Tel: 08-7646365 • 08-280345

Fujitsu Microelectronics Ltd.
Hargrave House
Belmont Road
Maidenhead
Berkshire SL6 6NE, United Kingdom
Tel: 0628-76100 • Fax: 0628-781484

Notes:

FUJITSU MICROELECTRONICS, INC.
3545 North First Street, San Jose, CA 95134-1804