RISC Family Users Guide

CYPRESS SEMICONDUCTOR

RISC
CY7C600 FAMILY USERS GUIDE
CYPRESS SEMICONDUCTOR

CY7C600 RISC FAMILY USERS GUIDE

CYPRESS SEMICONDUCTOR

3901 North First St., San Jose, CA 95134 (408) 943-2600

Telex: 821032 CYPRESS SNJ UD, TWX: 910 997 0753, FAX (408) 943-2741

Published June 3, 1988

Table of Contents

List of Figures

List of Tables

List of Tables

CHAPTER 1

Introduction

This chapter provides an overview of the basic concepts and advantages of RISC computer architectures in general and a brief summary of the specific features of the RISC computer implemented in Cypress's CY7C600 family.

Scalable Processor Architecture

The Cypress CY7C600 family implements a RISC architecture called SPARC™. SPARC stands for Scalable Processor ARChitecture. It is applicable to large high performance as well as small machines. The term "scalable" refers to the size of the smallest lines on a chip. As lines become smaller, chips get faster. However, some chip designs do not shrink well (they do not scale properly) because the architecture is too complicated. Because of its simplicity, the CY7C600 scales well. Consequently, CY7C600 systems will become faster as better semiconductor techniques are perfected. SPARC is an open computer architecture. We believe that the intelligent and aggressive nature of the SPARC design will make it an industry standard.The design specification is published, and other vendors are also producing SPARC microprocessors.

What is RISC?

RISC, an acronym for Reduced Instruction Set Computer, is a style of computer architecture emphasizing simplicity and efficiency. RISC designs begin with a necessary and sufficient instruction set. Typically, a few simple operations account for almost all computations. RISC machines are about two to five times faster than machines with traditional complex instruction set architectures. Also, RISC machines simpler designs are easier to implement, resulting in shorter design cycles.

RISC architectures are a response to the evolution from assembly language to high-level languages. Assembly language programs occasionally employ elaborate machine instructions, whereas high-level language compilers rarely do. For example, most C compilers use only about 30% of the available Motorola 68020 instructions. Studies show that approximately 80% of a typical programs computations require only about 20% of a processor's instruction set.

RISC is to hardware what the UNIX® operating system is to software. The UNIX system proves that operating systems can be both simple and useful. Hardware studies suggest the same conclusion. As advances in semiconductor technology reduce the cost of processing and memory, overly complex instruction sets become a performance liability. The designers of RISC machines strive for hardware simplicity, with close cooperation between machine architecture and compiler design. At each step, computer architects must ask to what extent does a feature improve or degrade performance and is it worth the cost of implementation? Each additional feature, no matter how useful it is in an isolated instance, makes all others perform more slowly by its mere presence.

The goal of RISC architecture is to maximize the effective speed of a design by performing infrequent functions in software, including in hardware only features that yield a net performance gain. Performance gains are measured by conducting detailed studies of large high-level language programs. RISC improves performance by providing the building blocks from which high-level functions can be synthesized without the overhead of general but complex instructions.

RISC Architecture

The following characteristics are characteristic of RISC architectures, including the CY7C600 design:

Single-cycle execution

Most instructions are executed in a single machine cycle.

Hardwired control with no microcode

Microcode adds a level of complexity and raises the number of cycles per instruction.

Load/Store, register-to-register design

All computational instructions involve registers. Memory accesses are made with only load and store instructions.

Simple fixed-format instructions with few addressing modes

All instructions are one word long (typically 32 bits) and have few addressing modes.

Pipelining

The instruction set design allows for the processing of several instructions at the same time.

High-performance memory

RISC machines have at least 32 general-purpose registers (the CY7C600 has 136) and large cache memories.

Migration of functions to software

Only those features that measurably improve performance are implemented in hardware. Programs contain sequences of simple instructions for executing complex functions rather than the complex instructions themselves.

Simple, Efficient Instruction Pipeline Visible to Compliers

For example, branches take effect after execution of the following instruction, permitting a fetch of the next instruction during execution of the current instruction.

The real keys to enhanced performance are single-cycle execution and keeping the cycle time as short as possible. Many characteristics of RISC architectures, such as load/store and register-to-register design, facilitate single-cycle execution. Simple fixed-format instructions, on the other hand, permit shorter cycles by reducing decoding time.

Note that some of these features, particularly pipelining and high-performance memories, have been used in supercomputer designs for many years. The difference is that in RISC architectures these ideas are integrated into a processor with a simple instruction set and no microcode.

Moving functionality from run time to compile time also enhances performance. Functions calculated at compile time do not require further calculating each time the program runs. Furthermore, optimizing compilers can rearrange pipelined instruction sequences and arrange register-to-register operations to reuse computational results.

A new set of design criteria has emerged:

> Instructions should be simple unless there is a good reason for complexity. To be worthwhile, a new instruction that increases cycle time by 10% must reduce the total number of cycles executed by at least 10%.
>
> Microcode is generally no faster than sequences of hardwired instructions. Moving software into microcode does not make it better, it just makes it easier to modify.
>
> Fixed-format instructions and pipelined execution are more important than program size. As memory gets cheaper and faster, the space/time tradeoff resolves in favor of time. Reducing space no longer decreases time.
>
> Compiler technology should use simple instructions to generate more complex instructions. Instead of substituting a complicated microcoded instruction for several simple instructions, which compilers did in the 1970s, optimizing compilers can form sequences of simple, fast instructions out of complex high-level code. Operands can be kept in registers to increase speed even further.

RISC's Speed Advantage

Using any given benchmark, the performance (P) of a particular computer is inversely proportional to the product of the benchmark's instruction count (I) the average number of clock cycles per instruction (C) and the inverse of the clock speed (S). Let's assume that a RISC

$$P = \frac{1}{I \times C \times \frac{1}{S}}$$

machine runs at the same clock speed as a corresponding traditional machine; S is identical. The number of clock cycles per instruction (I) is approximately 1.3 to 1.7 for RISC machines, and between 4 and 10 for traditional machines. This makes the instruction execution rate of RISC machines about 3 to 6 times faster than traditional machines. But, because traditional machines have more powerful instructions, RISC machines must execute more instructions for the same program, typically about 10% to 30% more. Since RISC machines execute 10% to 30% more instructions 3 to 6 times more quickly, they are about 2 to 5 times faster than traditional machines for executing typical large programs.

Compiled programs on RISC machines are somewhat larger than compiled programs on traditional machines, because several simple instructions replace one complex instruction resulting in decreased code density. All SPARC instructions are 32 bits wide, whereas some instructions on traditional machines are narrower. But the number of instructions actually executed may not be as great as the increased program size would indicate. A windowed register file, for example, often simplify call/return sequences so that context switches become less expensive.

CY7C600 Architecture

The SPARC CPU is composed of a CY7C601 Integer Unit (IU) that performs basic processing and a CY7C608 Floating-Point Controller (FPC) interface to the Texas Instruments' SN74ACT8847 Floating–Point Processor that performs floating-point calculations. The CY7C608/SN74ACT8847 combination act as a SPARC compatible Floating–Point Unit (FPU). CY7C600-based computers typically have a memory management unit (MMU), a large virtual-address cache for instructions and data, and are organized around a 32-bit data and instruction bus.

The integer and floating-point units operate concurrently. The FPU performs floating-point calculations with a set number of floating-point arithmetic units. The CY7C600 architecture also specifies an interface for the connection of an additional coprocessor.

Instruction Categories

The CY7C600 architecture has about 50 integer instructions, a few more than earlier RISC designs, but less than half the number of Motorola 68000 integer instructions. CY7C600 instructions fall into five basic categories:

Load and store instructions (the only way to access memory)

These instructions use two registers or a register and a constant to calculate the memory address involved. Half-word accesses must be aligned on 2-byte boundaries, word accesses on 4-byte boundaries, and double-word accesses on 8-byte boundaries. These alignment restrictions greatly speed up memory access.

Arithmetic/logical/shift instructions

These instructions compute a result that is a function of two source operands and place the result in a register. They perform arithmetic, logical, or shift operations.

Floating–Point and Coprocessor nstructions

These include floating-point calculations, operations on floating-point registers, and instructions involving the optional coprocessor. Floating-point operations execute concurrently with IU instructions and with other floating-point operations when necessary. This architectural feature hides floating-point concurrency from the programmer.

Control-transfer instructions

These include jumps, calls, traps, and branches. Control transfers are usually delayed until after execution of the next instruction, so that the pipeline is not emptied every time a control transfer occurs. Thus, compilers can be optimized for delayed branching.

Read/write control register instructions

These include instructions to read and write the contents of various control registers. Generally the source or destination is implied by the instruction.

Artificial Intelligence Instructions

These include the tagged arithmetic instructions Tagged Add and Tagged Subtract. Tagged instructions are useful for implementing artificial intelligence languages such as LISP, because tags can automatically indicate to software interpreters the data type of arithmetic operands.

Multiprocessing Instructions

These inclue two instructions for implementing semaphores in memory: Atomic Load/Store Unsigned Byte which loads a byte from memory then sets the location to all "1's and SWAP which exchanges the contents of a register and a memory location. Both of these instructions are "atomic" or uninterruptable.

Register Windows

A unique feature that contributes to the high performance of the CY7C600 design is its overlapping register windows. Results left in registers by a calling routine automatically become available operands for the called routine, obviating the need for load and store instructions to main memory.

According to the architectural specification, there may be between 2 and 32 register windows, each window having 24 working registers, plus 8 global registers. The first implementation has 8 register windows with 24 registers each (but count only 16 since 8 overlap), plus 8 global registers, for a total of 136 registers. Recent research suggests that register windows and tagged arithmetic, found in CY7C600 systems, but not in other commercial RISC machines, are sufficient to provide excellent performance for expert system development requiring AI languages such as LISP and Smalltalk.

Traps and Interrupts

The CY7C600 design supports a full set of traps and interrupts. They are handled by a table that supports 128 hardware and 128 software traps. Even though floating-point instructions can execute concurrently with integer instructions, floating-point traps are precise because the FPU supplies (from the table) the address of the instruction that failed.

Protection

Some CY7C600 instructions are privileged and can only be executed while the processor is in supervisor mode. This instruction execution protection ensures that user programs cannot accidentally alter the state of the machine with respect to its peripherals and vice versa.

The CY7C600 design also provides memory protection, which is essential for smooth multitasking operation. Memory protection makes it impossible for user programs to corrupt the system, other user programs, or themselves.

An Open Architecture

Advantages of Open Architecture

The CY7C600 design is the first open RISC architecture, and one of the few open CPU architectures. Standard products are more beneficial than proprietary ones, because standards allow users to acquire the most cost-effective hardware and software in a competitive multi-vendor marketplace. Integrated circuits come from several competing semiconductor vendors, while software is supplied by systems vendors. This advantage is lost when users are limited by a processor that has proprietary hardware and software.

RISC architectures, and the CY7C600 design in particular, are easy to implement because they are relatively simple. Since they have short design cycles, RISC machines can absorb new technologies almost immediately, unlike more complicated computer architectures.

CY7C600 systems were designed to support:

The C programming language and the UNIX operating system,

Numerical applications (using FORTRAN), and

Artificial intelligence and expert system applications using Lisp and Prolog

Supporting C is relatively easy; most modern hardware architectures are able to do so. The one essential feature is byte addressability. However, numerical applications require fast floating-point and artificial intelligence applications require large address spaces and interchangeability of data types.

The floating-point processor, with pipelined floating-point operation capabilities, achieves the high performance needed for numerical applications. Floating-point coprocessors are generally not part of RISC machines, but they are available for microprocessors such as the Motorola 68020 and the Intel 80386, and for CY7C600 systems as well.

For artificial intelligence and expert system applications, CY7C600 systems offer tagged instructions and word alignment. Because languages such as Lisp and Prolog are often interpreted, word alignment makes it easier for interpreters to manipulate and interchange integers and different types of pointers. In the tagged instructions, the two low-order bits of an operand specify the type of operand. If an operand is an integer, most of the time it is added to (or subtracted from) a register. If an operand is a pointer, most of the time a memory reference is involved. Language interpreters can leave operands in the appropriate registers, greatly improving the performance of exploratory programming environments.

The CY7C600 architecture does not dictate a memory management unit (MMU), although a high performance unit has been specified for the SPARC architecture. We expect the same processor to be used in different types of machines. For example, a single-user machine for embedded applications does not need an MMU. By contrast, a multitasking machine used for timesharing, such as a traditional UNIX box, needs a paging MMU. Furthermore, a multiprocessor such as a vector machine or hypercube requires specialized memory management facilities. The CY7C600 architecture can be implemented with a different MMU configuration for each of these purposes, without affecting user programs.

CY7C600 Machines and Other RISC Machines

The CY7C600 design has more similarities to Berkeley's RISC-II architecture than to any other RISC architecture. Like the RISC-II architecture, it uses register windows in order to reduce the number of load/store instructions. The CY7C600 architecture allows 32 register windows, but the initial implementation has 8 windows. The tagged instructions are derived from SOAR, the "Smalltalk On A RISC" processor developed at Berkeley after implementing RISC-II.

CY7C600 systems are designed for optimal floating-point performance, and support single-, double-, and extended-precision operands and operations, as specified by the ANSI/IEEE 754 floating-point standard. High floating-point performance results from concurrency of the IU and FPU. The integer unit loads and stores floating-point operands, while the floating-point unit performs calculations. If an error (such as a floating-point exception) occurs, the floating-point unit specifies precisely where the trap took place; execution is expediently resumed at the discretion of the integer unit. Furthermore, the floating-point unit has an internal instruction queue; it can operate while the integer unit is processing unrelated functions.

CY7C600 systems deliver very high levels of performance. The flexibility of the architecture makes future systems capable of delivering performance many times greater than the performance of the initial implementation. Moreover, the openness of the architecture makes it possible to absorb technological advances almost as soon as they occur.

CY7C600 Product Family

Since the CY7C600 has been designed to offer a complete solution for the implementation of high performance computers and controllers, the family consists of several members including an Integer Unit, a Floating-Point Controller, a Cache Tag RAM, a Cache Data RAM, and a Memory Management Unit.

The SPARC processor family consists of a CY7C601 Integer Unit (IU) to perform all non-floating-point operations and a CY7C608 Floating-Point Controller (FPC) which interfaces to a standard Floating-Point Unit to perform floating-point arithmetic concurrent with the IU. Support is also provided for a second generic coprocessor interface. The IU communicates with external memory via a 32-bit address bus and a 32-bit data/instruction bus. In typical data processing applications, the IU and FPU are combined with a high performance CY7C603 Memory Management Unit and a cache memory implemented with CY7C153 Cache RAMs and the CY7C181 Cache Tag RAM. In many dedicated controller applications the IU can function by itself with high speed local memory only.

CY7C601 Integer Unit

The IU is the basic processing engine which executes all of the instruction set except for floating-point operations. The CY7C601 IU contains a large 136 x 32 triple port register file which is divided into 8 windows. Each window contains 24 working registers and has access to the same 8 global registers. A current window pointer (CWP) field in the Processor State Register keeps track of which window is currently active. The CWP is decremented when the processor calls a subroutine and is incremented when the processor returns.

The registers in each windows are divided into ins, outs, and locals. Each window shares its ins and outs with adjacent windows. The outs of the previous window are the ins of the current window, and the outs of the current window are the ins of the next window. The globals are equally available to all windows and the locals are unique to each window. The windows are joined together in a circular stack where the outs of the last window are the ins of the first window.

The IU supports a multitasking operating system by providing user and supervisor modes. Some instructions are privileged and can only be executed while the processor is in supervisor mode. Changing from user to supervisor mode requires taking a hardware interrupt or executing a trap instruction.

The IU supports both asynchronous traps (interrupts) and synchronous traps (error conditions and trap instructions). Traps transfer control to an offset within a table. The base address of the table is specified by a Trap Base Register and the offset is a function of the trap type. Traps are taken before the current instruction causes any changes visible to the programmer and can therefore be considered to occur between instructions.

CY7C603 Memory Management Unit

The CY7C603 Memory Management Unit provides hardware support for a demand-paged virtual memory environment for the CY7C601 processor. The CY7C603 conforms to the

standard SPARC architecture definition for memory management. Page size is fixed at 4K bytes. The MMU translates 32-bit virtual addresses from the processor into 36-bit physical addresses.

High speed address look-up is provided by an on-chip translation lookaside buffer. Each entry contains the virtual to physical mapping of a 4K byte page. If a virtual address finds a match in one of the TLB entries, the physical address translation contained in that entry will be delivered to the outputs of the MMU. If the virtual address from the processor has no corresponding entry in the MMU, the MMU will automatically perform address translation for the virtual address using on-chip hardware to access a main memory resident three-level page table. Each "matched" TLB entry is checked for protection violation automatically and violations are reported to the Integer Unit as memory exceptions.

CY7C608 Floating-Point Controller

The CY7C608 Floating-Point Controller in combination with a Texas Instruments 8847 Floating-Point Processor form a SPARC compatible Floating-Point Unit or FPU. The FPU and CY7C601 IU operate concurrently. The FPU recognizes floating-point instructions and places them in a queue while the IU continues to execute non-floating-point instructions. If the FPU encounters an instruction which will not fit in its queue, the FPU holds the IU until the instruction can be stored.

The FPU contains its own set of registers on which it operates. The contents of these registers are transferred to and from external memory under control of the IU via floating-point load/store instructions. Processor interlock hardware hides floating-point concurrency from the compiler or assembly language programmer. A program containing floating-point computations generates the same results as if instructions were executed sequentially.

SN74ACT8847 Floating-Point Processor

The SN74ACT8847 form Texas Instruments combines a multiplier and an arithmetic logic unit ina single microprogrammable VLSI device. The 8847 is capable of operating at ther same clock rate as the Cypress IU and FPC and provides on the order of 4 to 4.5 Megaflops of double precision Linpack floating-point performance when operated at 33 Mhz with these devices. The 8847 is fully compatible with the IEEE standard for binary floating-point arithmetic, STD 754-1985. The Floating-Point Processor performs both single and double precision operations, including division and square root.

CY7C181 Cache Tag RAM

The CY7C181 Cache Tag RAM provides storage for 4096 cache address tag entries. These entries can be directly written or read by the processor. In normal operation twelve low order bits of address from the processor are used to select one of the tag entries in the CY7C181 and its 20-bit contents are compared on chip with the 20 high order processor address bits to determine if the cache contains the required data or instruction. This cache hit/miss comparison is then qualified by various built-in protection checks and the result is output. Pipelined accesses are supported via on-chip registers which capture both address and data from the processor.

The CY7C181 also contains the logic required in a system to implement the byte and half-word write capabilities provided in the SPARC instruction set. Cache tag update is also simplified by an automatic tag update on miss feature which eliminates the need for processor accesses during tag update.

CY7C153 Cache Data RAM

The CY7C153 static RAM is designed to interface easily to and provide maximum performance for the CY7C600 processor. The RAM has registered address inputs and latched data inputs and outputs as well as a self-timed write pulse which greatly simplify the design of cache memories for the CY7C601 Integer Unit. The device has a single clock that controls loading of the address register, data input latches, data output latches, pipeline control latch, and chip enable register. The chip enable is clocked into a register and pipelined through a control register to condition the output enable. This pipelined design allows a cache that works as an extension of the internal instruction pipeline of the CY7C601 Integer Unit thereby maximizing performance. The write enable is edge-activated and self timed thereby eliminating the need for the user to generate accurate write pulses in external logic. A separate asynchronous output enable is provided to disable outputs during a write or to allow other devices access to the bus.

CHAPTER 2

CY7C601 Architecture

CYThe CY7C601 is the Integer processing Unit (IU) of the SPARC 32-bit RISC architecture. The SPARC architecture makes possible the creation of a processor which can execute instructions at a rate approaching one instruction per processor clock. The SPARC Instruction Set Architecture consists of Integer, Floating-Point and coprocessor extensions. Implementation of the SPARC architecture is a variable left to the silicon designer.

In the Cypress implementation, the partitioning is into three specific elements, the Integer Unit, the Floating-point coprocessor, and a second coprocessor provision for future extensions into the arena of concurrent processing. The CY7C601 integer unit supports a tightly-coupled floating-point coprocessor (FPU) which operates concurrently and a second implementation-definable coprocessor also with the capability to operate concurrently with both the Integer Unit and the Floating-Point Coprocessor.

Overview

The SPARC architecture is used to produce a 32-bit Reduced Instruction Set Computer (RISC). The CY7C601 is the Integer Unit (IU) used to perform basic processing. The SPARC architecture provides in addition to an integer instruction set, a rich floating-point instruction set which may be implemented independently of the IU. It also provides instruction set support for an optional coprocessor. The details of the coprocessor are implementation-specific but both the generic coprocessor and the floating-point coprocessor may coexist with the CY7C601 IU and operate concurrently. The complete register set for the SPARC architecture is shown in *Figure 2–1*. In this implementation, the registers shown shaded are part of the CY7C601 IU, the lightly shaded registers are part of the floating-point coprocessor and the darkly shaded registers belong to any implementation specific coprocessor.

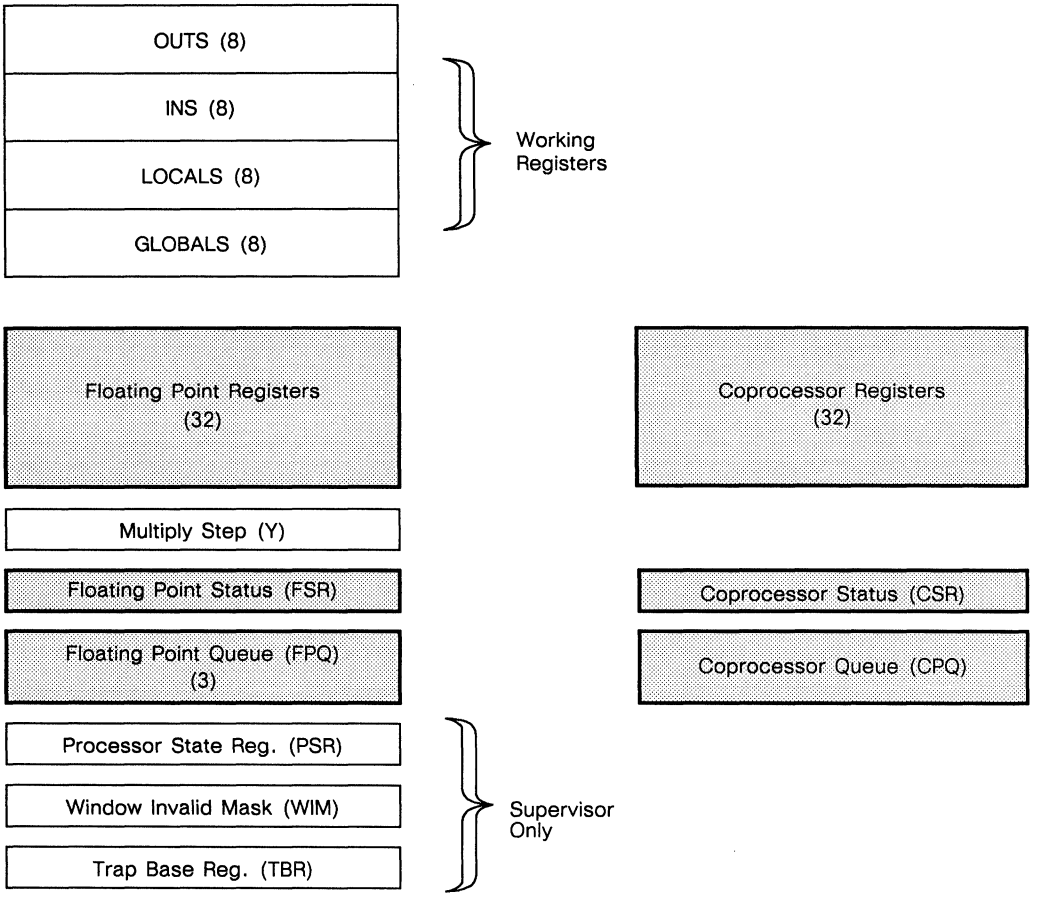


Figure 2–1. SPARC Architecture Register Model

An example of a computer system that uses the SPARC architecture is organized around a 32-bit virtual address bus and a 32-bit instruction/data bus. Its storage subsystem consists of a memory management unit (MMU) and a large cache for both instructions and data. The cache is virtual-address-based. Depending on the storage subsystem's interpretation of the processor's address space identifier (asi) bits, I/O registers are either addressed directly, bypassing the MMU, or they are mapped by the MMU into virtual addresses. Alternately, the entire physical memory space may exist in the virtual address space and no cache or memory management provisions may be necessary.As a controller, a SPARC system may include only the Integer Unit (with optional Floating–Point Unit), fast static RAM main memory and I/O devices.

Integer Unit, Floating-Point Unit, and Coprocessor

The IU is the basic processing engine of the SPARC architecture. It executes all the instruction except floating-point operate instructions and coprocessor instructions. A block diagram of the IU appears in *Figure 2–2*.

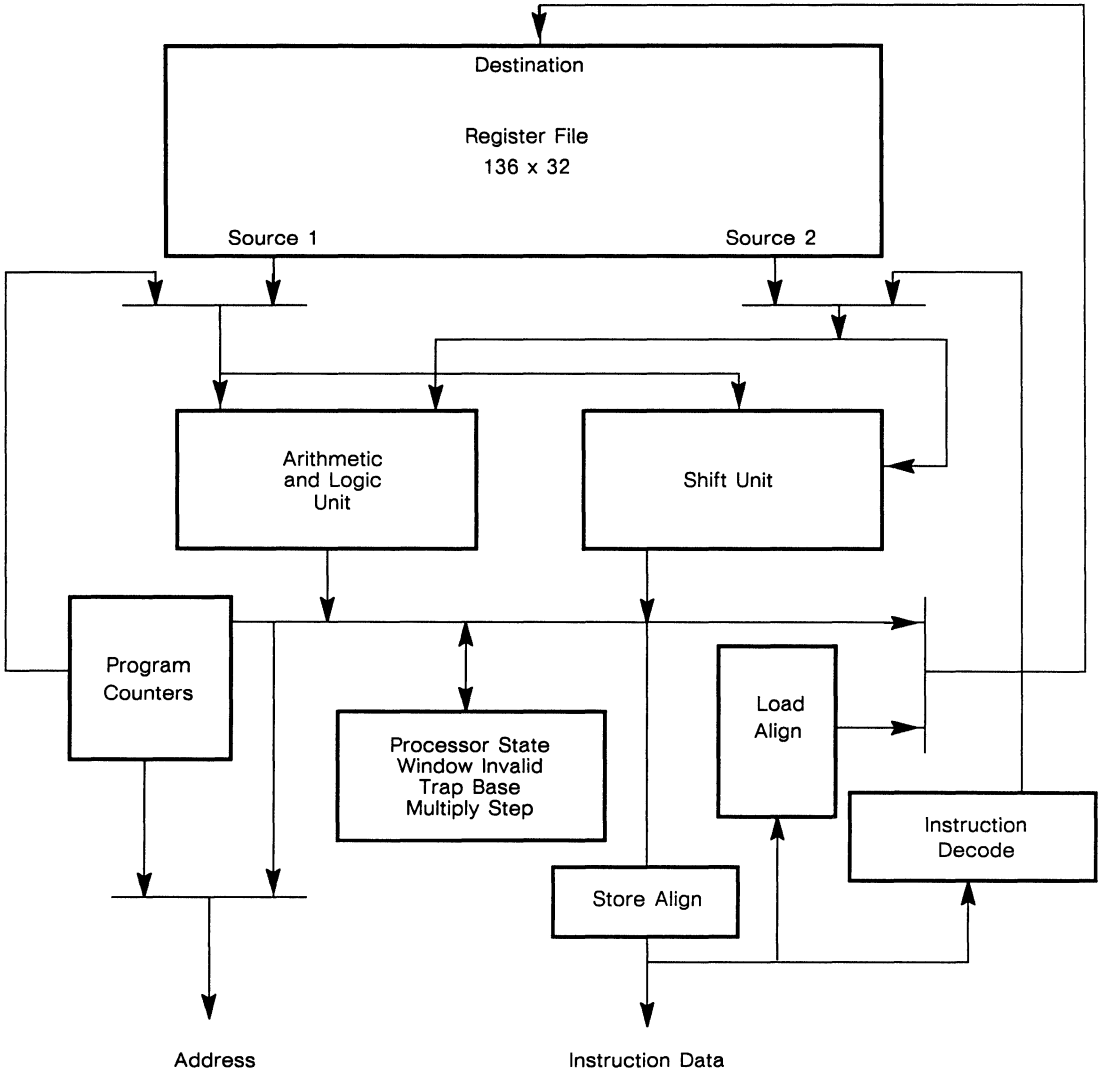


Figure 2–2. CY7C601 Block Diagram

The FPU and the IU operate concurrently. The FPU recognizes floating-point operate instructions and places them into a queue. Meanwhile, the IU continues to execute instructions. Floating-point operate instructions are executed from the queue when the specified floating-point registers are free and the required FPU is available. If the FPU encounters a floating-point operate instruction that doesn't fit in the queue, the IU stalls until the required FPU resource becomes available.

Floating-point load/store instructions are used to move data between the FPU and memory. The IU generates a memory address and the FPU either sources or sinks the data. Note that floating-point loads and stores are not floating-point operate instructions.

The architecture hides floating-point concurrency from the programmer, so the processor hardware provides the appropriate register interlocks. A program including floating-point computations generates the same results as if all instructions were executed sequentially.

The architecture supports an optional coprocessor. Like the FPU, the coprocessor recognizes coprocessor arithmetic instructions, and executes them concurrently with instructions executed by the IU.

Likewise, coprocessor load/store instructions are used to move data between the coprocessor and memory. For each floating-point load/store instruction, there is an analogous coprocessor load/store instruction.

The physical interface between both of the coprocessors and the IU is provided through a unique set of coprocessor control interfaces and a common interface to the system data bus and the virtual address bus. These control interfaces provide the synchronization and error handling that enable all three processors to operate concurrently. The common interface to the virtual address bus and data bus allows the IU to provide all addresses for floating-point and generic coprocessor Loads and Stores. and a Floating-Point Unit (FPU) to perform floating-point calculations concurrently with the IU.

Figure 2–2 is a block diagram of the CY7C601 Integer Unit. The processor is organized around the Arithmetic/Logic and Shift Units. Both of these are two operand units and accept 32-bit information from source 1 or 2 of the register file, the program counters or the instruction decoders. Results from these units may be passed to the Register File, Address bus Instruction/Data Bus, Program Counters and to or effect the Status Registers. Note that there is no direct connection from the data bus to either the Arithmetic/Logic or Shift Units. All data is transferred in or out of the processor with Load and Store Operations. Instructions are decoded and incoming information aligned as necessary directly from the Instruction/Data Bus. The Instruction Decoder contains a 4 stage pipeline for fetch, Decode, Execute and Write Operations.

The SPARC architecture supports a multitasking operating system by providing user and supervisor modes. Some instructions are privileged, and can only be executed while the processor is in supervisor mode. Changing from user to supervisor mode requires taking a hardware trap, or using a trap instruction.

Registers

The integer unit has two types of registers associated with it; working registers r registers and control/status registers. Working registers are used for normal operations, and control/status registers keep track of and control the state of the IU. The FPU has 32 working registers (called f registers, and two control/status registers: the Floating–point State Register (FSR), and the Floating–point Queue (FQ).

Integer Unit Registers

All r registers are 32 bits wide. They are divided into 8 global registers and a number of blocks called windows. Each window contains 24 r registers. The number of windows (NWINDOWS) ranges from 2 to 32 depending on the implementation. The CY7C601 contains 8 windows (a total of 136 registers) Windows are contiguously numbered from 0 to 7. At most NWINDOWS –1 windows are available to user code since one window must be available for trap handlers.

The windows are addressed by the CWP, a field of the Processor State Register (PSR). The CWP is incremented by a RESTORE or RETT instruction and decremented by a SAVE instruction. The active window is defined as the window currently pointed to by the CWP. The Window Invalid Mask (WIM) is a register which, under software control, detects the occurrence of IU register file overflows and underflows.

The registers in each window are divided into ins, outs, and locals. Note that the globals, while not really part of any particular window, can be addressed when any window is active. When any particular window is active, the registers are addressed as follows:

RP	Round To
0	Extended
1	Single
2	Double
3	(Unused)

Table 2–1. Extended Rounding Precision

Each window shares its ins and outs with adjacent windows. The outs from a previous window (CWP +1) are the ins of the current window, and the outs from the current window are the ins for the next window (CWP –1). The globals are equally available from all windows, and the locals are unique to each window.

The register addresses overlap such that, given a register with address o where $8 \leq o \leq 15$, o refers to exactly the same register as (o + 16) after the CWP is decremented by 1 modulo NWINDOWS (points to the next window). Likewise, given a register with address i where $24 \leq i \leq 31$, i refers to exactly the same register as address (i – 16) after the CWP is incremented by 1 modulo NWINDOWS (points to the previous window).

The windows are joined together in a circular stack, where the highest numbered window is adjacent to the lowest. If NWINDOWS = 8, the outs of window 7 are the ins of window 0. *Figure 2–3* and *Figure 2–4* show the relationships.

Because the processor logically provides new locals and outs after every procedure call, register local values need not be saved and restored across calls. The overlap registers also minimize the overhead of passing and returning values. They can be used as follows:

In preparation for a procedure call, a routine generally moves the parameters into its out registers. After the CALL, the CWP is decremented with the SAVE instruction, what was the next window becomes the active window, and the parameters are directly accessible by the callee, since the caller's outs are the callee's ins.

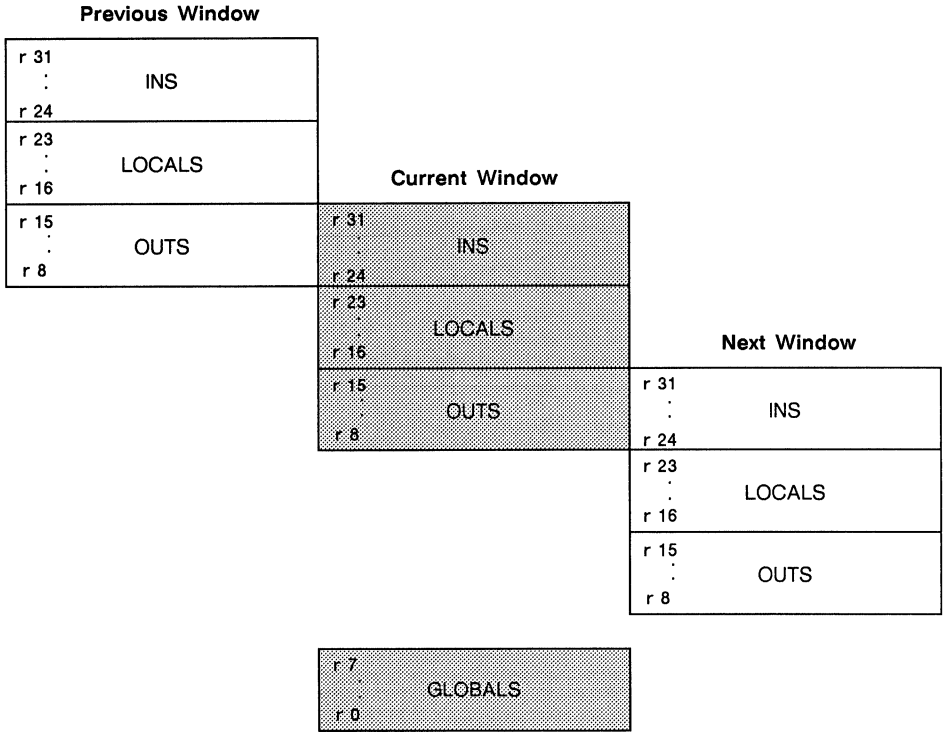


Figure 2–3. Working Register Set

Likewise, in preparing for a procedure return, a routine generally moves its result(s) into its in registers. After the CWP is incremented via the RESTORE instruction, what was the previous window becomes the active window, and the return values are accessible by the returnee, because the returner's ins are the returnee's outs. Note that the terms ins and outs are defined relative to calling, not returning.

Since any implementation has only a finite number of windows, the register file becomes full after the number of procedure calls exceeds the number of returns by NWINDOWS 1. A subsequent call causes the operating system to move one or more (in and local sets of) windows from the register file into memory. The SAVE instruction automatically checks for the window overflow condition.

Similarly, the register file can become empty when the number of procedure returns exceeds the number of calls by NWINDOWS –1. A subsequent return causes one or more previously saved windows to be moved from memory into the register file. The RESTORE instruction automatically checks for the window underflow condition. The architecture works best with efficient window overflow and window underflow handlers.

By software convention, you can provide additional locals (and consequently, fewer ins and outs). For example, software can assume that the boundary is actually between r[26] and r[27], providing 6 outs, 10 locals, and 6 ins.

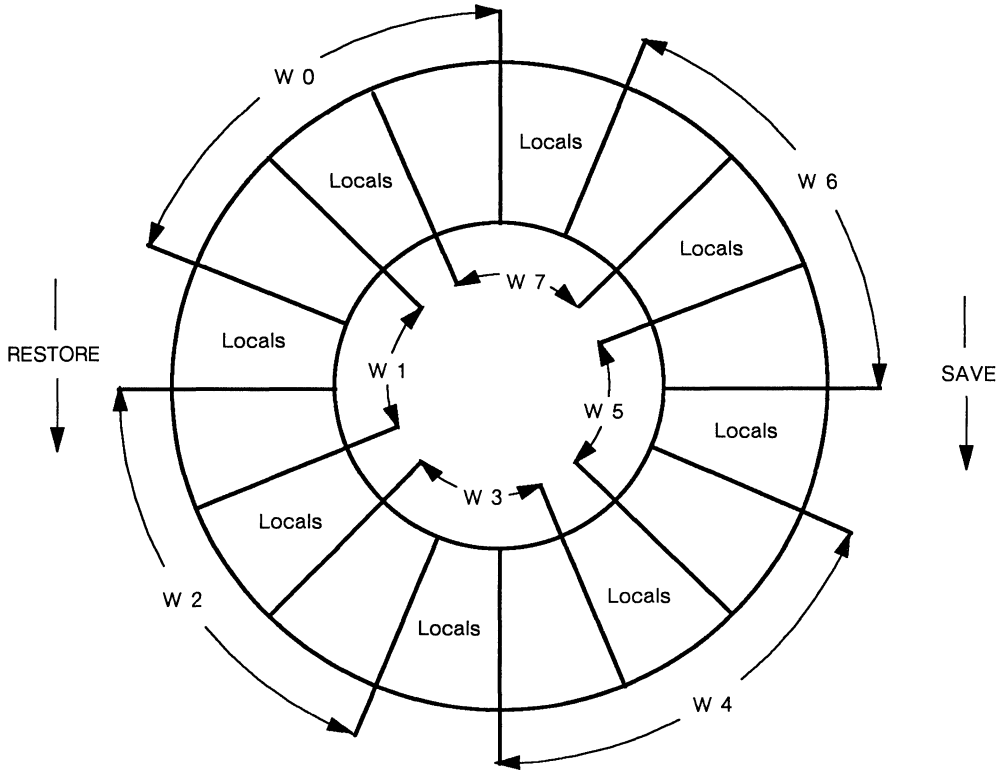


Figure 2–4. Overlapping Register Windows

In *Figure 2–4*, NWINDOWS = 8. It does not show the 8 globals. If the procedure corresponding to the window labeled w0 does a procedure call (executes a SAVE instruction), a window overflow trap will occur. The overflow trap handler uses the locals of w7:

CWP=0	active window = 0
CWP+1 = 1	previous window = 1
CWP–1 = 7	next window = 7
WIM=10000000	trap window = 7

Special r Registers

The utilization of two r registers is partially fixed by the instruction set: If global register r[0] is addressed as a source operand (rs1 or rs2 = 0), the operand value 0 is returned. If r[0] is addressed as a destination operand (rd = 0), no register is modified. The CALL instruction writes

its own address into out register r[15]. Also note that traps save the program counters (PC and nPC) into two locals of the next window.

Integer Unit Control/Status Registers

The IU's control/status registers are all 32-bit read/write registers unless specified otherwise. They include the program counters (PC and nPC), the Processor State Register (PSR), the Window Invalid Mask register (WIM), the Trap Base Register (TBR), and the multiply-step (Y) register. Control/status registers contain two types of fields, mode and status. Mode fields are set by the programmer; they appear in UPPER CASE (for example, PIL). Status fields appear in lower case (for example, ver).

Integer Program Counters (PC and nPC)

The Program Counter (PC) contains the address of the instruction currently being executed by the IU, and the nPC holds the address of the next instruction to be executed (assuming a trap does not occur). In delayed control transfers, the instruction that immediately follows a control transfer may be executed before control is transferred to the target. The nPC is necessary to implement this feature.

Processor State Register (PSR)

This 32-bit register contains various fields describing the state of the IU. It can be modified by the SAVE, RESTORE, Ticc and RETT instructions, or by instructions that modify the condition codes. The (privileged) instructions RDPSR and WRPSR read and write it directly. The PSR provides the following fields:

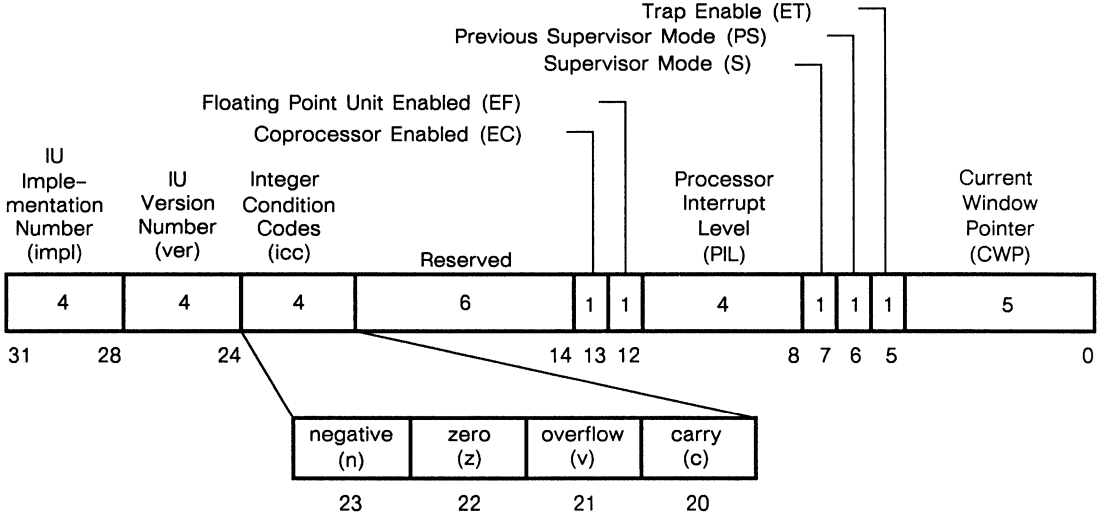


Figure 2-5. Processor State Register

Implementation (impl) Bits 31 through 28 identify the implementation number of the processor. This field is 0001 for the CY7C601. The WRPSR instruction does not modify this field.

Version (ver) Bits 27 through 24 contain a constant: the meaning of this constant depends on the value of the impl field. This field is 0000 for the initial version of the CY7C601.The WRPSR instruction does not modify this field.

Integer Condition Codes (icc) Bits 23 through 20 contains the integer unit's condition codes. 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 and Ticc instructions base their control transfer on these bits, which are defined as follows:

Negative (n) Bit 23 indicates whether the ALU result was negative for the last instruction that modified the icc field. 1 = negative, 0 = not negative.

Zero (z) Bit 22 indicates whether the ALU result was zero for the last instruction that modified the icc field. 1 = result was zero, and 0 = result was nonzero.

Overflow (v) If bit 21 is 1, it indicates that an arithmetic overflow occurred during the last instruction that modified the icc field. If bit 21 is 0, this indicates that an arithmetic overflow did not occur. Logical instructions that modify the icc field always set the overflow bit to 0.

Carry (c) If bit 20 is 1, it indicates that either an arithmetic carry out of bit 31 occurred as the result of the last addition that modified the icc, or that a borrow into bit 31 occurred as the result of the last subtraction that modified the icc .If bit 20 is 0, this indicates that a carry did not occur. Logical instructions that modify the icc field always set the carry bit to 0.

Reserved Bits 19 through 14 are reserved. This field should only be written to 0 by the WRPSR instruction.

Enable Coprocessor (EC) This bit determines whether the coprocessor is enabled or disabled. 1 = enabled, 0 = disabled.

Enable FPU (EF) This bit determines whether the FPU is enabled or disabled. 1 = enabled, 0 = disabled.

If the FPU is either disabled, or enabled and not present, an FPop, FBfcc, or floating-point load/store instruction causes an fp disabled trap. Similarly, if the coprocessor is either disabled, or enabled and not present, a CPop, CBccc, or coprocessor load/store instruction causes a cp disabled trap.

When the FPU (or CP) is disabled, it retains its state until it is reenabled or reset. When disabled, the FPU can continue to execute instructions in its queue. The CP can also, if it has a queue.

When the FPU is present, software can use the EF bit to determine whether a particular process uses the FPU. If a process does not use the FPU, the FPU's registers need not be saved and restored across context switches. Also, if the FPU is not present, (as indicated by the bp FPU present signal), the fp disabled trap can be used to emulate the floating-point instruction set. (This also applies to the coprocessor.)

Processor Interrupt Level (PIL) Bits 11 through 8 identify the processor interrupt level. The processor only accepts interrupts whose interrupt level is greater than the value in PIL. Bit 11 is the MSB and bit 8 is the LSB.

Supervisor (S) Bit 7 determines whether the processor is in supervisor mode: when S = 1, the processor is in supervisor mode. Note that because the instructions to write the PSR are only available in supervisor mode, supervisor mode can only be entered by a software or hardware trap.

Previous Supervisor (PS) Bit 6 contains the value of the S bit at the time of the most recent trap.

Enable Traps (ET) Bit 5 is the Trap Enable bit. When ET = 1, traps are enabled. When ET = 0, traps are disabled, and all asynchronous traps are ignored. Synchronous traps and floating-point/coprocessor traps cause the IU to halt and enter error mode. (See Appendix C for a definition of error mode.)

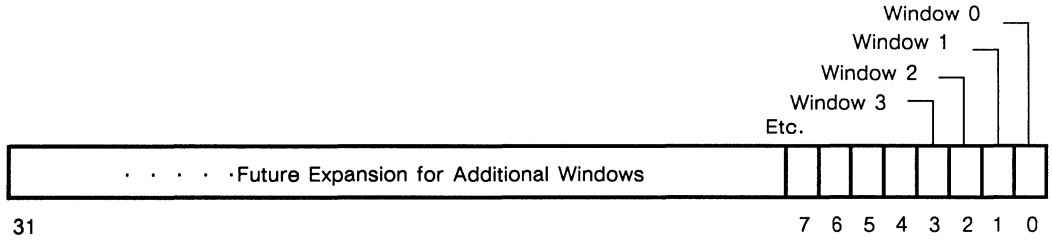
If traps are enabled (ET=1), some care must be taken when you disable them (ET=0). Since the "RDPSR, WRPSR" instruction sequence is interruptible, it may not be appropriate in some situations. Here are two alternatives: 1) generate a "trap instruction" trap instead (this disables traps); or 2) use the "RDPSR, WRPSR" sequence and write the interrupt trap handlers so that before they return to the supervisor, they restore the PSR to the value it had when the interrupt handler was entered. Note that the PS bit cannot be restored. In alternative (1), the trap handler should verify that it was called from the supervisor state before returning to the supervisor.

Current Window Pointer (CWP) Bits 4 through 0 comprise the Current Window Pointer, which points to the current active r register window. It is decremented by traps and the SAVE instruction, and incremented by RESTORE and RETT instructions.

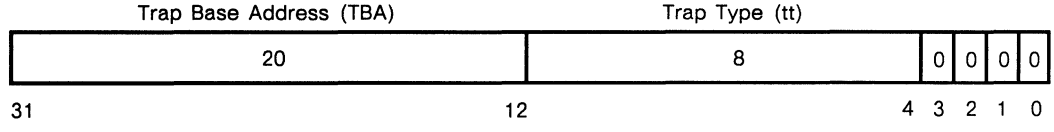
Window Invalid Mask Register (WIM)

This register is used to determine whether a window overflow or window underflow trap should be generated by a SAVE, RESTORE, or RETT instruction. Each bit in the WIM register corresponds to a window. For example, bit 0 corresponds to window 0 (CWP = 0), bit 1 corresponds to window 1 (CWP = 1), and so on. If a SAVE, RESTORE, or RETT would cause the CWP to point to a window whose corresponding WIM bit equals 1, it causes a window overflow (SAVE) or window underflow (RESTORE, RETT) trap.

This register can be read by the RDWIM instruction, and written by the WRWIM instruction. Bits corresponding to unimplemented windows read as zeroes and values written to unimplemented bits are ignored. The WIM provides the following fields:



WINDOW INVALID MASK



TRAP BASE REGISTER

Figure 2–6. Window Invalid & Trap Base Registers

Trap Base Register (TBR)

The trap base register contains three fields that generate the address of the trap handler when a trap occurs. These are:

Trap Base Address (TBA) Bits 31 through 12 comprise the Trap Base Address (TBA), which is controlled by software. It contains the most-significant 20 bits of the trap table address. (Note that the reset trap is an exception; it traps to address 0). The TBA field can be written by the WRTBR instruction.

Trap Type (tt) Bits 11 through 4 comprise the Trap Type (tt) field. This is an 8-bit field that is written by the processor at the time of a trap, and retains its value until the next trap. It provides an offset into the trap table. The WRTBR instruction does not affect the tt field. Bits 3 through 0 are zeroes. The WRTBR instruction does not affect this field.

Y Register

The multiply step instruction (MULScc) uses the 32-bit Y register to create 64-bit products. An example algorithm is described in Appendix B. This register can be read and written using the RDY and WRY instructions.

Floating-Point Registers

The floating-point unit has 32 working registers called f registers, a Floating-Point State Register (FSR) that contains mode and status information about the FPU, and a Floating-Point Queue

(FQ) that holds one or more 64-bit instruction/address pairs. Software uses the FQ to recover from floating-point exceptions.

Floating-Point f registers

The 32-bit f registers are numbered from f[0] to f[31]. These can be read and written by floating-point operate (FPop and FPcmp) instructions, or by load/store single/double floating-point instructions (LDF, LDDF, STF, STDF). They are addressable at all times.

A single f register can hold one single-precision operand. Double-precision operands require an f register pair, where the double-e datum occupies an even-numbered register, and the double-f datum occupies the following odd-numbered register. Extended-precision operands require an f register quad, with extended-e, extended-f, extended-f low, and extended-u in register addresses 0, 1, 2, and 3 modulo 4, respectively. Thus, the f register file can hold 8 extended, 16 double, or 32 single-precision operands.

Floating-Point State Register (FSR)

The FSR register fields contain FPU mode and status information. The fields are:

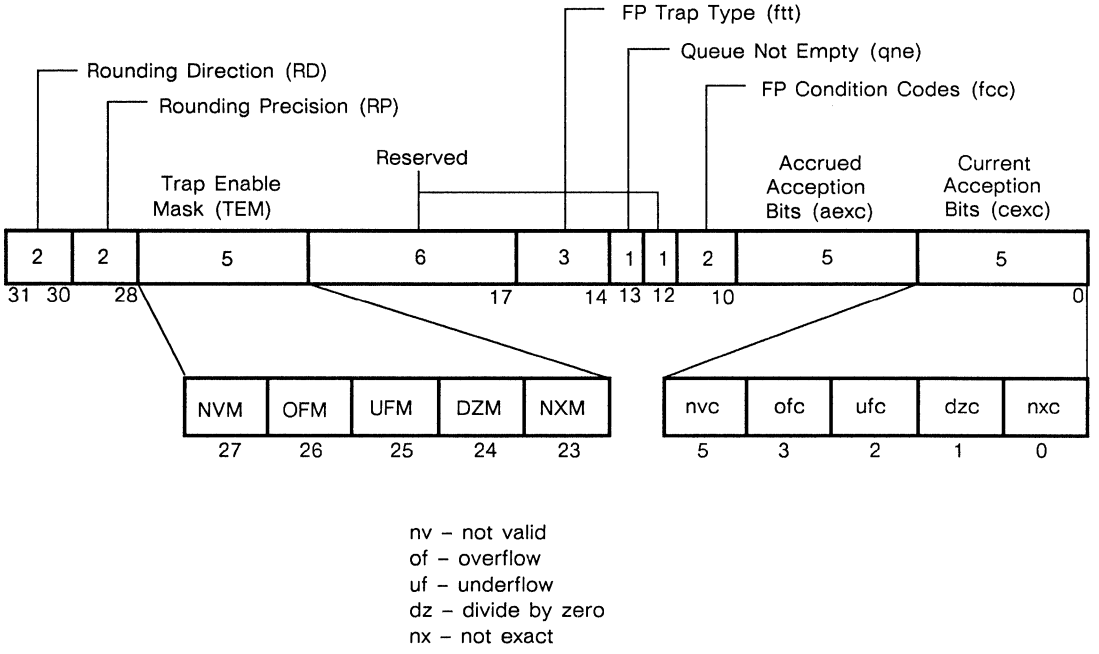


Figure 2-7. Floating-Point State Register

Rounding Direction (RD) Bits 31 and 30 select the rounding direction for floating-point results, according to the ANSI/IEEE 754-1985 Standard:

RD	Round Toward
0	Nearest (even if a tie)
1	0
2	+ 1
3	- 1

Table 2–2. Rounding Direction (RD)

Extended Rounding Precision (RP) Bits 28 and 29 determine the precision to which extended results are rounded, according to the ANSI/IEEE 754–1985 Standard:

RP	Round To
0	Extended
1	Single
2	Double
3	(Unused)

Table 2–3. Extended Rounding Precision

Trap Enable Mask (TEM) Bits 27 to 23 are enable bits for each of the five floating-point exceptions that can be indicated in the current exception field (cexc). (See definition of cexc below.) If a floating-point operate instruction generates one or more exceptions and the TEM bit corresponding to one or more of the exceptions is set (1), an fp exception trap is caused. A reset (0) TEM bit prevents that exception type from generating a trap. (See below.) The TEM field may be read and written by the STFSR and LDFSR instructions.

Abrupt Underflow (AU) Bit 22, when set to 1, causes denormalized floating-point operands and/or results to be rounded to zero. The definition of AU mode is implementation-dependent and is not defined by the ANSI/IEEE 754–1985 Standard.

Reserved Bits 21 through 17 and bit 12 are reserved. When read by an STFSR instruction, this field delivers all zeroes. This field should only be written to zero by the LDFSR instruction.

Floating-Point Trap Type (ftt) Bits 16 through 14 identify fp exception traps. After a floating-point exception trap occurs, the ftt field encodes the type of exception. ftt remains valid until the next FPop instruction completes. (Note that the exception-causing FPop and its address are in the first entry of the Floating-point Queue. The ftt field can be read by the STFSR instruction. An LDFSR instruction does not affect ftt. This field encodes the exception types as follows:

ftt	Trap Type
0	None
1	IEEE Exception
2	Unfinished FPop
3	Unimplemented FPop
4	Sequence Error

Table 2–4. Floating-Point Trap Types

An **IEEE exception** indicates that an ANSI/IEEE 754–1985 exception occurred for the FPop identified by the front entry of the FQ. The exception type(s) is indicated in the cexc field. If the IEEE exception results in a fp exception trap (as determined by the TEM) then the destination f register, fcc, and aexc fields remain unchanged. However, if the IEEE exception does not result in a trap, then the f register, fcc, and aexc fields are updated to their new values

An **unfinished FPop** indicates that an implementation's FPU was unable to generate correct results or exceptions, as defined by the ANSI/IEEE 754–1985 Standard. In this case, the cexc field is undefined. (However, the aexc and fcc fields, and the destination f register are not affected by the exception.)

An **unimplemented FPop** indicates that an implementation's FPU decoded an FPop that it did not implement. In this case, the cexc field is undefined. (However, the aexc and fcc fields, and the destination f register are not affected by the exception.)

In the case of an unfinished FPop or unimplemented FPop, the software should emulate or reexecute the instructions in the FQ, and update the FSR and destination f register(s) .

A **sequence error** indicates that an FPop or a load floating–point instruction is fetched while the FPU is in FPU exception mode, waiting for the FQ to be emptied by software.

Queue Not Empty (qne) Bit 13 indicates whether the Floating–point Queue (FQ) is empty after an fp exception trap or after a Store Double Floating–point Queue (STDFQ) instruction is executed. If qne = 0, the queue is empty; if qne = 1, the queue is not empty. The qne bit can be read by the STFSR instruction. The LDFSR instruction does not affect qne. However, executing successive STDFQ instructions will (eventually) cause the FQ to become empty (qne = 0).

Floating–point Condition Codes (fcc) Bits 11 and 10 contain the FPU condition codes. These bits are updated by floating–point compare instructions (FCMP and FCMPE) and are read and written by the STFSR and LDFSR instructions, respectively. Note that fcc is updated even if FCMPE generates an IEEE exception trap.

In the following table, fs1 and fs2 correspond to the values in the f registers specified by an instruction's rs1 and rs2 fields. The question mark (?) indicates an unordered relation, which is true if either fs1 or fs2 is a signaling or quiet NaN (see the section Processor Data Types).

The FBfcc instruction bases its control transfer on this field, which is interpreted as follows:

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (Unordered)

Table 2–5. Floating–Point Condition Codes

Accrued Exception Bits (aexc) Bits 9 through 5 accumulate IEEE floating–point exceptions while fp exception traps are disabled. After an FPop completes, the TEM and cexc fields are logically and 'd together. If the result is nonzero, an FP exception trap is generated; otherwise,

the new cexc field is or 'd into the aexc field. Thus, while traps are masked, exceptions are accumulated in the aexc field. The aexc field is read and written by the STFSR and LDFSR instructions.

Current Exception Bits (cexc) Bits 4 through 0 indicate one or more IEEE exceptions that were generated by the most recently executed FPop instruction. The absence of an exception causes the corresponding bit to be cleared. The cexc field is read and written by the STFSR and LDFSR instructions.

The cexc bits are not defined following an FPop that causes an unimplemented FPop or unfinished FPop fp exception trap. Following an FPop that does not generate an fp exception trap or that generates an IEEE exception trap, the cexc bits are set as follows:

nvc = 1 indicates an invalid operation: an operand is improper for the operation to be performed. For example, 0/0, and $\infty - \infty$ are invalid.

ofc = 1 indicates overflow: the rounded result would be larger in magnitude than the largest normalized number in the specified format.

ufc = 1 indicates underflow: the rounded result is inexact, and would be smaller in magnitude than the smallest normalized number in the indicated format.

dzc = 1 indicates division-by-zero: X/0, where X is subnormal or normalized. Note that 0/0 does not set the dzc bit.

nxc = 1 indicates an inexact result: The rounded result differs from the infinitely precise correct result.

nxc = 1 indicates inexact: The rounded result differs from the infinitely precise correct result.

The following illustration summarizes the handling of IEEE exception traps. Note that the aexc and ftt fields can normally only be cleared by software.

1. FPop generates an IEEE exception or exceptions
2. cexc is loaded with the IEEE exceptions generated
3. if cexc and TEM are equal to zero then:

aexc is loaded with aexc or cexc, the result is stored in the f registers, and fcc is updated accordingly.

if cexc or TEM are not equal to zero then:

ftt is loaded with the IEEE exception type and a fp exception trap is taken

Since the operating system must be capable of simulating the entire FPU in order to properly handle the unimplemented FPop and unfinished FPop floating-point exceptions, a user process always "sees" a fully implemented FSR as defined above. In other words, a user process always "sees" cexc, aexc, and TEM fields that conform to the ANSI/IEEE 754-1985 Standard.

Floating-Point Queue (FQ)

The Floating-point Queue keeps track of FPops that are pending completion by the FPU when an fp exception trap occurs. When an fp exception trap occurs, the first entry in the queue gives the

address of the FPop that caused the exception and the instruction itself. Any remaining entries in the queue contain FPop instructions (and their addresses) that had not finished when the exception occurred.

Addressing

Addressing conventions follow the big endian convention as shown in *Figure 2–7*. Addressing bytes for load and store byte instructions is such that increasing the address means decreasing the significance of the of the byte within the word. The most significant byte (MSB) of a word is accessed when address bits <1:0> = 0 and the least significant byte within the word is accessed when address bits <1:0> = 3.

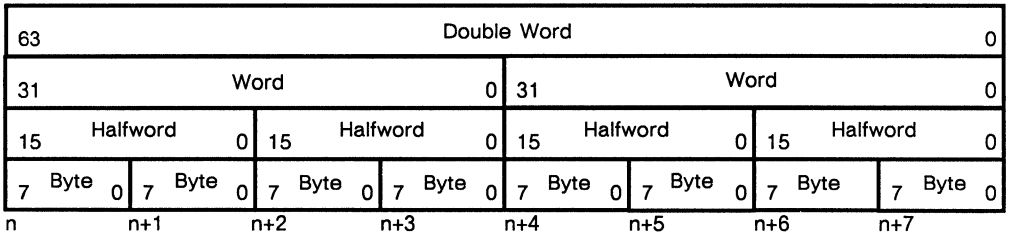


Figure 2–8. Address Conventions

Halfwords are addressed as follows. The least significant Halfword is addressed when address bit <1> = 1. The most significant Halfword is addressed when address bit <1> = 0. All words are located on word boundaries, address bits <1:0> = 0. When addressing Doublewords or word pairs, the least significant word is addressed when address bit <2> = 1 and the most significant word when address bit <2> = 0. The address of a Doubleword, Word or Halfword is the address of its most significant Byte.

A Doubleword datum is located at a doubleword address which is evenly divisible by 8. A Word datum is located on a word address which is evenly divisible by 4. A halfword datum is located on a halfword boundary, and is evenly divisible by 2. Attempting to access Halfwords, Words or Doublewords that are mis-aligned will result in a "memory address not aligned" trap.

Address generation is provided for two cases, generating a memory address for a load or store operation and generating an address for the program counter.

The three cases for Load and store operations are shown in the top of *Figure 2–8*. The first is simply the sum of two 32 values contained in the register file. The second is the sum of a 32-bit register value and a 13-bit sign extended immediate value from the Load or Store instruction. The third case is special in that it allows the addressing of the upper and lower 4K bytes of memory through the use of a single sign extended immediate value.

Normally the program counter is incremented by 4 on each instruction fetch (PC=PC+4 because instructions are located on word boundaries), pointing to the next instruction to be fetched. In a branch or call operation, the branch or call is relative to this next program counter location or PC + 4. In the case of the Call instruction, the program counter is updated with the sum of PC +

4 and a 30 bit absolute displacement. This allows control transfers to any location in the virtual memory space on a 4-byte boundary. The Branch operation sums the PC + 4 value with a 22-bit sign extended displacement, allowing branching within a 8-megabyte range of the PC + 4 location on 4-byte boundaries.

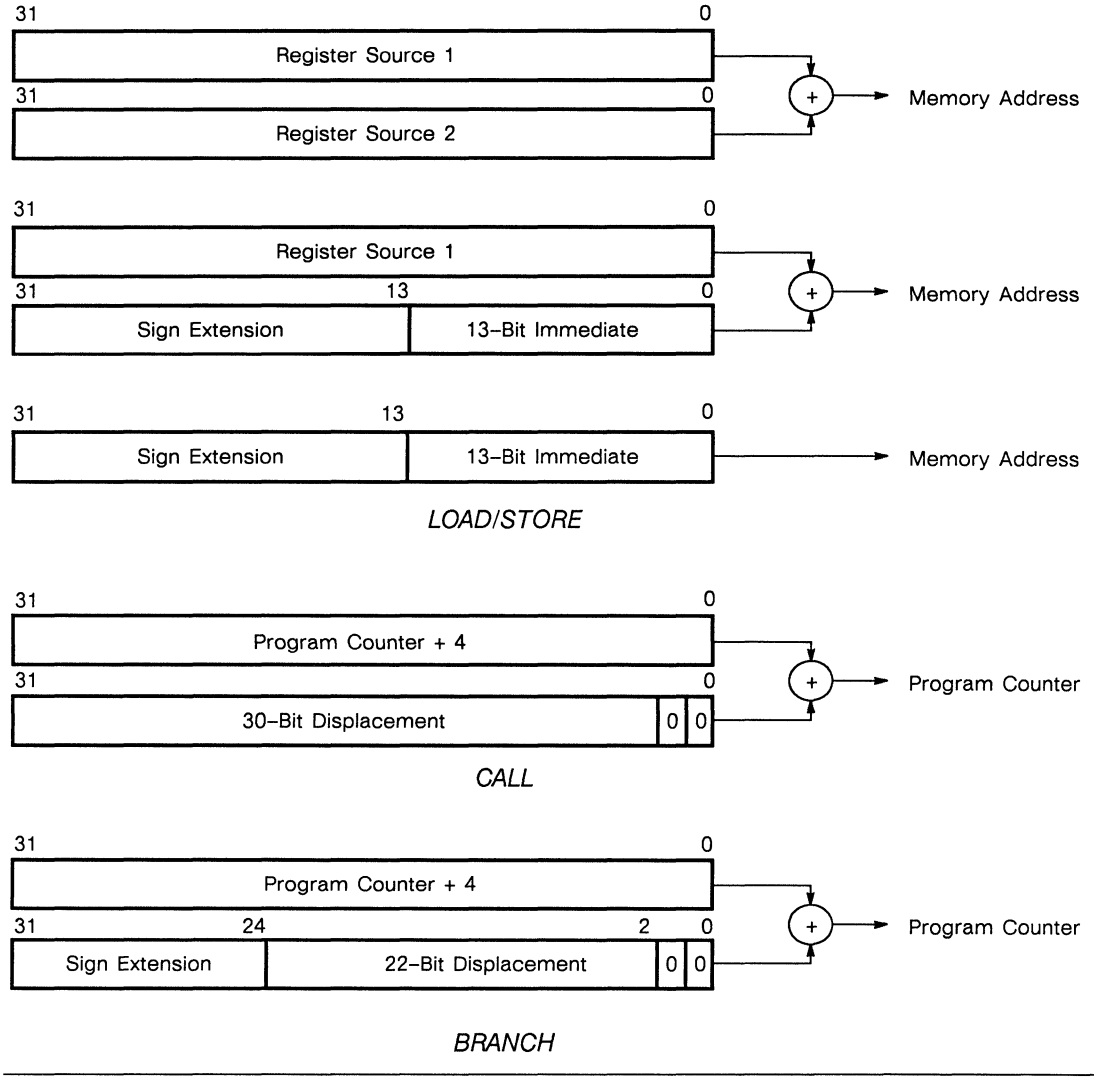


Figure 2–9. Addressing Modes

Instruction Set

Instructions fall into six basic categories: Load and store, Arithmetic/logical and shift, Control-transfer, Read/write control register, Floating-point operate and Coprocessor operate.

Instruction Format

Figure 2–9 shows each instruction format, with its fields and bit positions. It also lists the types of instructions that use that format.

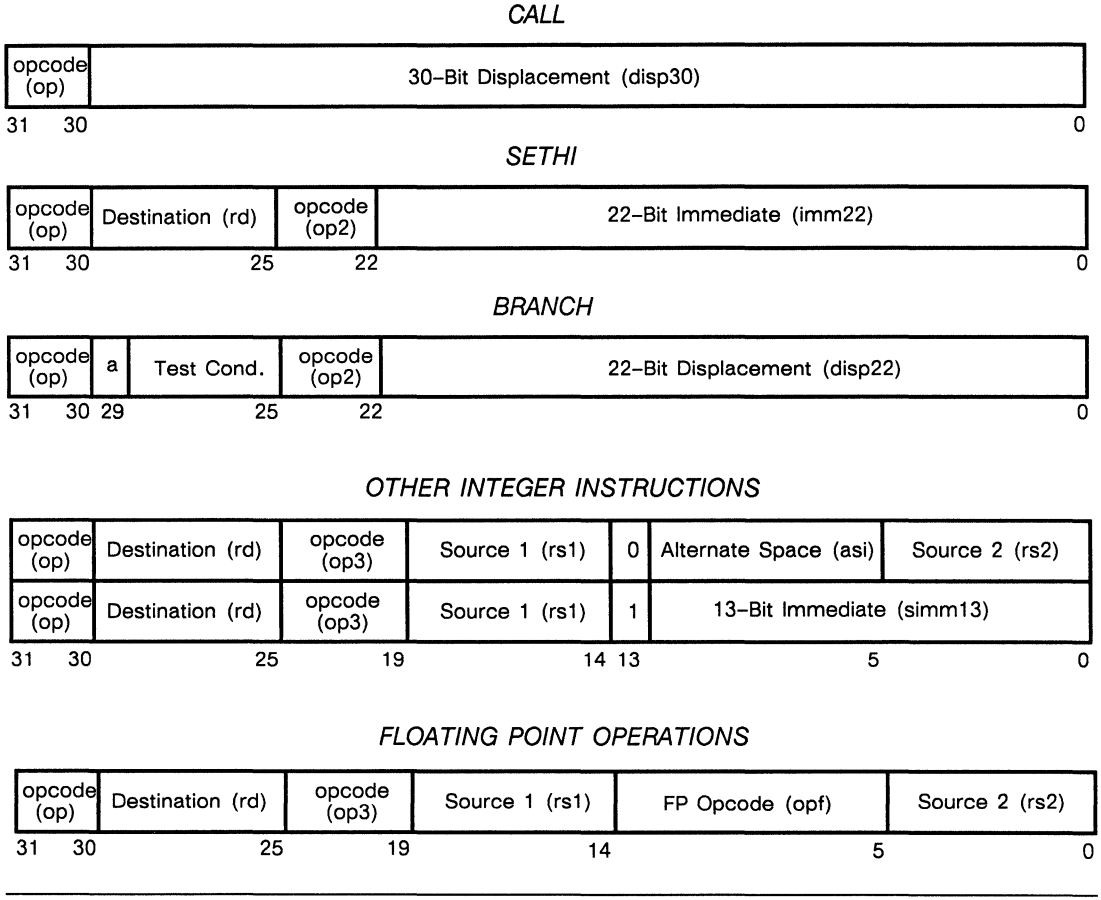


Figure 2–10. Instruction Format Summary

The fields in these instructions have the following meanings:

op This field places the instruction into one of the 3 major formats:

op Value	Instruction
0	Call
1	Bicc, FBfcc,CBccc, SETHI
2 or 3	other

Table 2–6. Use of op Field

op2 This field, bits 24 through 22 of format 2 instructions, selects the instruction as follows:

op2 Value	Instruction
0	UNIMPlemented
2	Bicc
4	SETHI
6	FBfcc
7	CBccc

Table 2–7. Use of op2 Field

rd For store instructions, this register selects an r register, or an f register (or an f register pair) to be the source. For all other instructions, this field selects an r register, or an f register (or an f register pair) to be the destination. Reading r[0] produces the result 0, and writing it causes the result to be discarded.

a The "a" bit means "annul" in format 2 instructions. This bit changes the behavior of the instruction encountered immediately after a control transfer, as described later in this chapter.

cond This field selects the condition code for format 2 instructions.

imm22 This field is a 22-bit constant value used by the SETHI instruction.

disp22 and **disp30** These fields are 30-bit and 22-bit sign-extended word displacements, for PC-relative calls and branches, respectively.

op3 The op3 field selects one of the format 3 opcodes.

i The i bit selects the type of the second ALU operand for non-FPop instructions. If i = 0, the second operand is r[rs2]. If i = 1, the second operand is sign-extended simm13.

asi This 8-bit field is the address space identifier generated by load/store alternate instructions. See discussion below.

rs1 This 5-bit field selects the first source operand from either the r registers or the f registers.

rs2 This 5-bit field selects the second source operand from either the r registers or the f registers.

simm13 This field is a sign-extended 13-bit immediate value used as the second ALU operand when i = 1.

opf This 9-bit field identifies a floating-point operate (FPop) instruction or a coprocessor operate (CPop) instruction. Note that it uses the synonym opc for coprocessor operate instructions (see the coprocessor operate instructions in Appendix B). A table in Appendix F shows the relationship between the opf field and FPop instructions.

The following sections describe each instruction category briefly; for a complete description of the instruction set, see Appendix B.

Load and Store Instructions

Load and store instructions are the only instructions that access data memory. They use two IU registers or an IU register and a signed immediate value to calculate the memory address. The instruction's destination field specifies either an IU register, FPU register, or coprocessor register; this register supplies the data for a store, or receives the data from a load. They generate a 32-bit byte address. In addition to the address, the processor always generates an address space identifier, or asi.

Address Space Identifier

The address space identifier generated by the processor is made available to the external system to distinguish up to 256 address spaces. These spaces can include system control registers, main memory, etc.

The SPARC architecture defines four address spaces and their asi values; these appear in *Table 2–8*. They indicate to the external system whether the processor is in user or supervisor mode (as indicated by the PSR), and whether the access is an instruction or a data reference.

Address Space Identifier (ASI)	Address Space
00001000	User Instruction
00001010	User Data
00001001	Supervisor Instruction
00001011	Supervisor Data

Table 2–8. ASI Bit Assignments

Load/store instructions normally generate an asi of either 10 or 11 for the data access, depending on whether the processor is in user or supervisor mode. However, the load from alternate space and store into alternate space instructions use the asi field supplied by the instruction itself. Note that the load/store alternate instructions are privileged; they can only be executed in supervisor mode.

Arithmetic/Logical/Shift

These instructions (with one exception) compute a result that is a function of two source operands; they either write the result into a destination register (r[rd]) or discard it. They perform arithmetic, tagged arithmetic, logical, or shift operations. The exception is a specialized instruction used to create 32-bit constants in two instructions. One of the operands is always r[rs1]. The other operand depends on the i bit in the instruction: if i = 0, the operand is r[rs2], but if i = 1, the operand is the sign-extended constant sign extend(simm13).

Register 0

Reading r[0] produces the value zero. If the destination field indicates a write into r[0], no r register is modified and the result is discarded. Most of these instructions have dual versions which modify the integer condition codes (icc) as a side effect. r[0] can be used to implement a register-to-register move in one of several ways: ADD with 0, OR with 0, etc. Subtract and set condition codes (SUBcc) can be used as an integer COMPARE instruction.

Shift

Shift instructions can be used to shift the contents of a register left or right, by a distance specified by the instruction or by an IU register. Shift instructions shift an r register left or right by a constant or variable amount, as described in Appendix B. None of the shift instructions changes the condition codes.

SETHI

The "set high 22 bits of r" (SETHI) instruction writes a 22-bit constant from the instruction into the high-order bits of the destination register. It clears the low-order 10 bits, and does not change the condition codes. SETHI can be used to construct a 32-bit constant using two instructions: SETHI followd by an OR Immediate.

Tagged Arithmetic

The tagged arithmetic instructions assume that the least-significant two bits of the operands are tags and set a condition code bit if they are not zero.The tagged add and subtract instructions (TADDcc, TSUBcc, TADDccTV and TSUBccTV) operate on tagged data where the tag is the low-order two bits of the data. If either of the instruction's two operands has a nonzero tag, the overflow bit of the PSR is set. The "trap on overflow" versions, TADDccTV and TSUBccTV, in addition to writing the condition codes, also cause an overflow trap.One possible model for tagging is to use 0 to tag integers and 3 for pointers to doublewords, i.e. list cells. If trapping overhead is insignificant, then TADDccTV or TSUBccTV is faster than the non-trapping versions, which would need to be followed by 'branch on overflow' instructions.

For example p contains a tagged pointer to a list cell, i.e. has 3 in its low-order two bits. Since the load and store instructions execute successfully only with properly aligned addresses, a load or store word with an address specifier of "p – 3" or "p + 1" will succeed, accessing the first or second word of the list cell, respectively; if, on the other hand, p contains a tag value other than 3, they will trap.

Control-Transfer Instructions

Control-transfer instructions include jumps, calls, traps, and branches. Control transfer is usually delayed so that the instruction immediately following the control transfer is executed before control actually transfers to the target address. The instruction following the control-transfer instruction is called a delay instruction. The delay instruction is always fetched, even when the control transfer is an unconditional branch. However, a bit in the control-transfer instruction can cause the delay instruction to be annulled (i.e. to have no effect) if the branch is not taken (or in one case, if the branch is taken).

There are five types of control transfer instructions:

1. Conditional branch (Bicc, FBfcc, CBccc)
2. Jump and Link (JMPL)
3. Call (CALL)
4. Trap (Ticc)
5. Return from trap (RETT)

Each of these can be further categorized according to whether it is:

1. PC-relative or register-indirect, or
2. delayed or non-delayed.

The following matrix shows these characteristics:

Instructions	Addressing Mode	Delayed	Annul Bit
Conditional Branch	Program Counter Relative	Yes	Yes
Call	Program Counter Relative	Yes	Yes
Jump	Register Indirect	Yes	No
Return	Register Indirect	Yes	No
Trap	Register Indirect	No	No

Table 2–9. Instruction Categories

The following paragraphs describe each of the characteristics:

PC-relative A PC-relative control transfer computes its target address by adding the (shifted) sign-extended immediate displacement to the program counter (PC).

Register-indirect A register-indirect instruction computes its target address as either "r[rs1] + r[rs2]" if i = 0, or "r[rs1] + sign ext(simm13)" if i = 1.

Delayed A control transfer instruction is delayed if it transfers control to the target address after a one-instruction delay. Delayed control transfers are described in the next section.

Delayed Control Transfer

Traditional architectures usually execute the target of a control transfer instruction immediately after the control-transfer instruction. This architecture delays by one instruction the execution of the target of a delayed control-transfer instruction. The instruction encountered immediately after a delayed control transfer is called the delay instruction.

PC and nPC

In general, the PC points to the instruction being executed by the IU, and the nPC points to the instruction to be executed next. Most instructions complete by copying the contents of the nPC into the PC, then either increment nPC by 4, or, if the instruction implies a control transfer, write the computed target address into nPC. The PC now points to the instruction that will be executed next, and the nPC points to the instruction that will be executed after the next one; in other words, two instructions hence.

Delay Instruction

The instruction pointed to by the nPC when a delayed control-transfer instruction is encountered is called the delay instruction. Normally, this is the next sequential instruction in the code space. However, if the instruction that preceded the delayed control transfer was itself a delayed control transfer, the address of the delay instruction is the target of the (first) control-transfer instruction, since that is where the nPC will point. This behavior is explained further in the section Back-to-Back Delayed Control Transfers below.

The following example shows the order of execution for a simple (not back-to-back) delayed control transfer. The order of execution is 8, 12, 16, 40. If the delayed control transfer-instruction were not taken, the order would be 8, 12, 16, 20.

PC	nPC	Instruction
8	12	Non-control transfer
12	16	Control transfer (target = 40)
16	40	Non-control transfer (delay instruction)
		Transfers control to 40
40	44	...

Table 2–10. Delay Instruction Example

Annul Bit

The a (annul) bit changes the behavior of the delay instruction. This bit is only available on conditional branch instructions (Bicc, FBfcc and CBccc). If a is set on a conditional branch (except BA, FBA and CBA) and the branch is not taken, the delay instruction is "annulled" (not executed). An annulled instruction has no effect on the state of the IU nor can a trap occur during an annulled instruction. If the branch is taken, the *a* bit is ignored and the delay instruction is executed. For example:

PC	nPC	Instruction	Action
8	12	Non-control transfer	Executed
12	16	Bicc (a=1) 40	Not Taken
16	40	...	Executed
20	24	...	Executed

Table 2–11. Effect of Annul Bit (a=1)

BA, FBA and CBA instructions are a special case; if the a bit is set in these instructions the delay instruction is not executed if the branch is taken, but it is executed if the branch is not taken. The following display shows the effect of the a bit on the delay instruction after various kinds of branches:

PC	nPC	Instruction	Action
8	12	Non-control transfer	Executed
12	16	Bicc (a=0) 40	Not Taken
16	40	...	Executed
40	44	...	Executed

Table 2–12. Effect of Annul Bit (a=0)

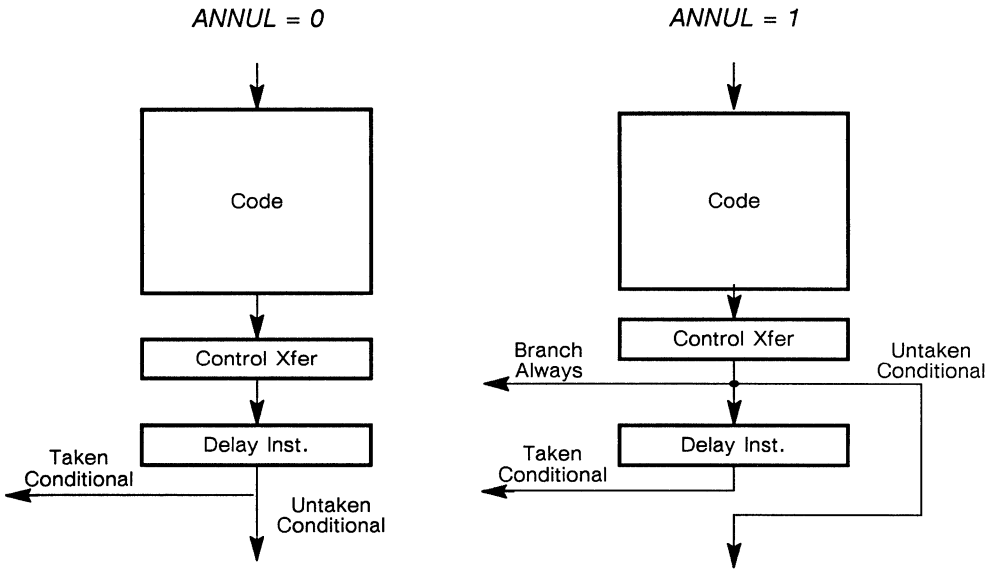


Figure 2–11. Delayed Control Transfer

The annul bit increases the likelihood that a compiler or optimizer can place a useful instruction in the delay slot after a branch. Refer to the following table:

Address	Instruction	Target
L	Non Control Transfer	
L'	...	
...	...	
...	...	
...	Bicc	L
D	NOP	
...	...	

Table 2–13. Code Optimizer Use of Annul Bit

If the Bicc has a = 0, a code optimizer may be able to move a non-control-transfer instruction from within the loop into location D. If the Bicc has a = 1, then the compiler can copy the non-control-transfer instruction at location L into location D, and change the branch to Bicc L'.

The annul bit can also be used to optimize "if-then-else" statements. Since the conditional branch instructions provide both true and false tests for all the conditions, an optimizer can arrange the code so that a non-control-transfer instruction from either the "else" branch or the "then" branch can be moved into the delay position after the branch instruction. For example:

Address	Instruction
	Bicc(cond. a=1) THEN
Delay:	Then Phrase 1
...	Else Phrase 1
...	Else Phrase 2
...	Goto
...	...
THEN	Then Phrase 2
...	Then Phrase 3

Address	Instruction
	Bicc(cond. a=1) ELSE
Delay:	Else Phrase 1
...	Then Phrase 1
...	Then Phrase 2
...	Goto
...	...
THEN	Else Phrase 2
...	Else Phrase 3

Table 2–14. Using Annul with IF-Then-Else Statements

When set in a branch always instruction (BA, BFA), the annul bit implements a "traditional," non-delayed branch instruction. This can also be used to dynamically replace unimplemented instructions with branches to software emulation routines as this requires less overhead than a trap.

Calls and Returns

Branch and call instructions use PC-relative displacements. The jump and link (JMPL) instruction uses a register-indirect displacement: it computes its target address as either the sum of two registers, or the sum of a register and a 13-bit signed immediate. The branch instruction provides a displacement of 8 Mbytes, while the call instruction's 30-bit word displacement allows a transfer to an arbitrary address.

A procedure that requires a register window is invoked by executing both a CALL (or a JMPL) and a SAVE instruction. A procedure that does not need a register window, a so-called "leaf" routine, is invoked by executing only a CALL (or a JMPL). Leaf routines can use only the \fIout\fP registers.

The CALL instruction stores PC, which points to the CALL itself, into register r[15] (an \fIout\fR register). JMPL stores PC, which points to the JMPL instruction, into the specified \fIr register\fP. These instructions then cause a transfer of control to a target that can be arbitrarily distant.

The SAVE instruction is similar to an ADD instruction, except that it also decrements the CWP by one, causing the active window to become the previous window, thereby "saving" the caller's window. Also, the source registers for the addition are from the previous window while the result is written into the new window. A procedure that uses a register window returns by executing both a RESTORE and a JMPL instruction. A leaf procedure returns by executing a JMPL only. The JMPL instruction typically returns to the instruction following the CALL's or JMPL's delay instruction; in other words, the typical return address is 8 plus the address saved by the CALL.

The RESTORE instruction, also like an ADD instruction, increments the CWP by one, causing the previous window to become the active window, thereby "restoring" the caller's window. Also, the source registers for the addition are from the current window while the result is written into the previous window.

Both SAVE and RESTORE compare the new CWP against the Window Invalid Mask (WIM) to check for window overflow or underflow. The SAVE and RESTORE instructions can be used to atomically update the CWP while establishing a new memory stack pointer in an \fIr register\fP.

Trap Instruction

The Ticc instruction evaluates the condition codes specified by its condition field, and if the result is true, it causes a trap with no delay instruction. If the condition codes evaluate to false, it executes as a NOP. A taken Ticc identifies the software trap by writing "trap number + 128" into the tt field of the TBR. The processor enters supervisor mode, disables traps, decrements the CWP, and saves PC and nPC into the locals r[17] and r[18] (respectively) of the new window. Ticc can be used to implement kernel calls, breakpointing, and tracing. It can also be used for run-time checks, such as out-of-range array indices, integer overflow, etc.

Delayed Control Transfer Couples

When a delayed control transfer is encountered immediately after another delayed control transfer, this creates what is called a delayed control-transfer couple, which the processor handles differently from a simple control transfer. The following tables show, first, a sequence of instructions that includes a delayed control-transfer couple, and second, a table that illustrates the order of execution depending on the nature of the control-transfer instructions. In the following tables, 'delayed control-transfer instruction' is abbreviated to 'DCTI'. Note that a "non-DCTI" may be either a non-control-transfer instruction, or a control-transfer instruction which is not delayed (i.e. a Ticc).

Address	Instruction	Target
8:	Non DCTI	
12:	DCTI	40
16:	DCTI	60
20:	Non DCTI	
24:	...	
...	...	
40:	Non DCTI	
44:	...	
...	...	
60:	Non DCTI	
64:	...	
...	...	

Table 2–15. Sequence of Delayed Control Transfer Couples

Where the annul bit is not indicated, it may be either 0 or 1. When the first instruction of a delayed control-transfer couple is a conditional branch, the transfer of control is undefined (case 6). If such a couple is executed, the location where execution continues is within the same address space but otherwise undefined. This sequence does not change any other aspect of the processor state.

Case	DCTI at Location 12	DCTI at Location 16	Order of Execution
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	B*A(a=1)	any CTI	12,40,44,...(16 annulled)
6	B*cc	DCTI	Not Supported

Definitions:

B*A----------BA,FBA, or CBA
B*cc----------Bicc,FBicc, or CBicc (except B*A)
DCTI Uncond.---CALL,JMPL,RETT, or B*A(a=0)
DCTI Taken-----CALL,JMPL,RETT,B*cc taken, or B*A(a=0)

Table 2–16. Execution of Delayed Control Transfer Couples

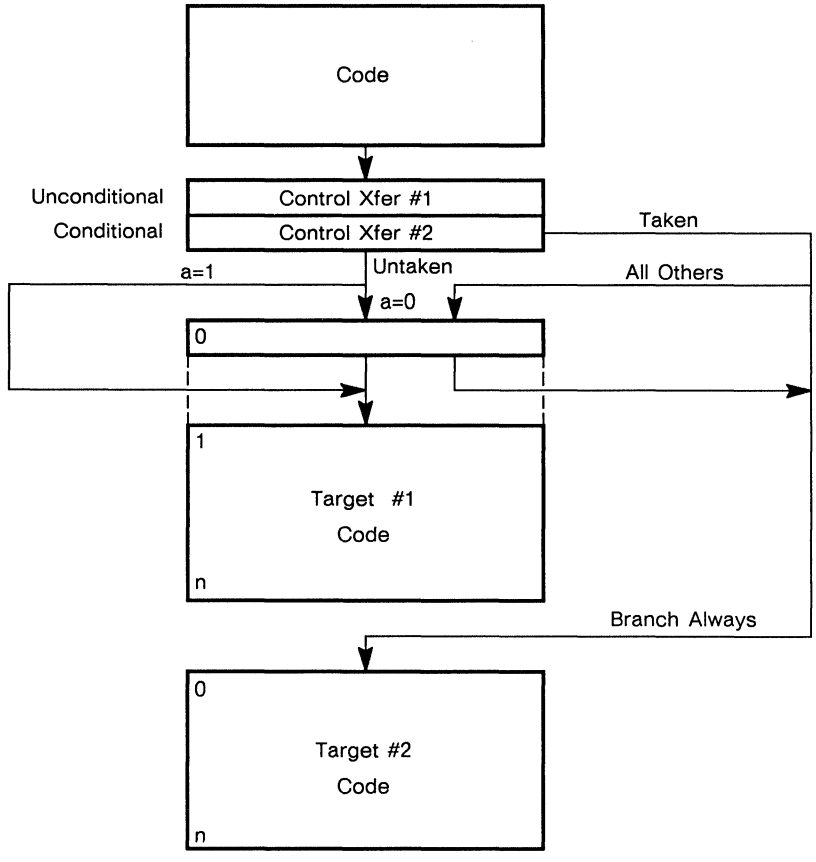


Figure 2–12. Back to Back Delayed Control Transfers

Read/Write Control Registers

The SPARC architecture provides instructions to read and write the contents of the various control registers. For reads and writes, the source and destination (respectively) are implied by the instruction itself. Case 1 of the above table includes the "JMPL, RETT" couple. RETT must always be preceded by a JMPL instruction. (If it is not, the location where execution continues is not necessarily within the address space implied by the PS bit of the PSR.) Trap handlers complete execution by executing the "JMPL, RETT" couple. These instructions read or write the contents of the programmer-visible control registers. This category includes instructions to read and write the PSR, the WIM, the TBR, the Y register, the FSR, and the CSR. These instructions are all privileged (available in supervisor state only), except those that read and write the Y register, the FSR, and the CSR.

Floating-point and Coprocessor Operate Instructions

Floating-point operate instructions perform all floating-point calculations. These are register-to-register instructions that use the floating-point registers. Like arithmetic/logical/shift instructions, these also compute some result that is a function of two source operands. However, they always write the result into a destination register.

Floating-point operate instructions execute concurrently with IU instructions and possibly with other floating-point instructions. A particular floating-point operate instruction is specified by a sub-field of the FPop instructions.

Coprocessor arithmetic instructions are defined by the implemented coprocessor, if any. They are specified by the CPop instruction. The architecture supports 1024 distinct coprocessor arithmetic instructions.

Floating-point loads and stores are NOT floating-point operate instructions (FPops), and coprocessor loads and stores are NOT coprocessor operate instructions. Floating-point and coprocessor loads and stores fall in the category "loads and stores".

Because the IU and the FPU can execute instructions concurrently, when a floating-point exception occurs, the program counter usually does not contain the address of the floating-point instruction that caused the exception. However, the first element of the floating-point queue points to the instruction that caused the exception, and the remaining elements point to floating-point operate instructions that have not yet completed. These can be re-executed or emulated.

Likewise, if the coprocessor executes instructions concurrently with the IU, the coprocessor can support a queue that, at the time of a coprocessor exception, will contain the instruction that generated the exception and remaining, unexecuted coprocessor instructions.

Condition Codes

Branch and Trap on Integer Condition Codes in *Table 2–13* are identical in terms of the operational condition being tested for either a Branch or Trap instruction.

A Bicc (Branch on Integer Condition Code) instruction (except BA and BN) evaluate the integer condition code (icc) according to the condition field. If the condition code is true, the branch is taken and the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 * sign ext (disp22))." If the condition code is false, the branch is not taken. If the branch is not taken and the *a* (annul) field is set, the delay instruction is annulled (not executed). If the branch is taken, the annul field is ignored.

Name	Operation	Cycles
LDSB(LDSBA*)	Load Signed Byte (from Alternate Space)	2
LDSH(LDSHA*)	Load Signed Halfword (from Alternate Space)	2
LDUB(LDUBA*)	Load Unsigned Byte (from Alternate Space)	2
LDUH(LDUHA*)	Load Unsigned Halfword (from Alternate Space)	2
LD(LDA*)	Load Word (from Alternate Space)	2
LDD(LDDA*)	Load Doubleword (from Alternate Space)	3
LDF	Load Floating Point	2
LDDF	Load Double Floating Point	3
LDFSR	Load Floating Point State Register	2
LDC	Load Coprocessor	2
LDDC	Load Double Coprocessor	3
LDCSR	Load Coprocessor State Register	2
STB(STBA*)	Store Byte (into Alternate Space)	3
STH(STHA*)	Store Halfword (into Alternate Space)	3
ST(STA*)	Store Word (into Alternate Space)	3
STD(STDA*)	Store Doubleword (into Alternate Space)	4
STF	Store Floating Point	3
STDF	Store Double Floating Point	4
STFSR	Store Floating Point State Register	3
STDFQ*	Store Double Floating Point Queue	4
STC	Store Coprocessor	3
STDC	Store Double Coprocessor	4
STCSR	Store Coprocessor State Register	3
STDCQ*	Store Double Coprocessor Queue	4
LDSTUB(LDSTUBA*)	Atomic Load/Store Unsigned Byte (in Alternate Space)	4
SWAP(SWAPA*)	Swap r Register with Memory (in Alternate Space)	4
ADD(ADDcc)	Add (and modify icc)	1
ADDX(ADDXcc)	Add with Carry (and modify icc)	1
TADDcc(TADDccTV)	Tagged Add and modify icc (and Trap on overflow)	1
SUB(SUBcc)	Subtract (and modify icc)	1

Table 2–17. SPARC Instruction Set

The exception BA (Branch Always) causes a branch to occur irrespective of the icc field. If the annul field is set, the delay instruction is annulled. If the annul field is not set, the delay instruction is executed. BN (Branch Never) acts like a "NOP" except that, if the annul field is set, the delay instruction is annulled. If the annul field is not set, the delay instruction is executed.

Except for BA, all Bicc instructions with the annul field set annul the delay instruction when the branch is not taken. BA, however, with the annul field set does the reverse, the delay instruction is annulled even though the branch is taken. The delay instruction other than a BA should not be a delayed control-transfer instruction.

The Ticc (Trap on Integer Condition Code) instructions evaluate the integer condition codes (icc) according to the condition field. If the condition code is true and there are no higher priority traps pending, then a Trap instruction is generated. If the condition code is false, then a Trap does not occur.

Name	Operation	Cycles
SUBX(SUBXcc)	Subtract with Carry (and modify icc)	1
TSUBcc(TSUBccTV)	Tagged Subtract and modify icc (and Trap on overflow)	1
MULScc	Multiply Step and modify icc	1
AND(ANDcc)	And (and modify icc)	1
ANDN(ANDNcc)	And Not (and modify icc)	1
OR(ORcc)	Inclusive Or (and modify icc)	1
ORN(ORNcc)	Inclusive Or Not (and modify icc)	1
XOR(XORcc)	Exclusive Or (and modify icc)	1
XNOR(XNORcc)	Exclusive Nor (and modify icc)	1
SLL	Shift Left Logical	1
SRL	Shift Right Logical	1
SRA	Shift Right Arithmetic	1
SETHI	Set High 22 Bits of r Register	1
SAVE	Save caller's window	1
RESTORE	Restore caller's window	1
Bicc	Branch on integer condition codes	1**
FBicc	Branch on floating point condition codes	1**
CBccc	Branch on coprocessor condition codes	1**
CALL	Call	1**
JMPL	Jump and Link	2**
RETT	Return from Trap	2**
Ticc	Trap on integer condition codes	1 (4 if Taken)
RDY	Read Y Register	1
RDPSR	Read Processor State Register	1
RDWIM	Read Window Invalid Mask	1
RDTBR	Read Trap Base Register	1
WRY	Write Y Register	1
WRPSR*	Write Processor State Register	1
WRWIM*	Write Window Invalid Mask	1
WRTBR*	Write Trap Base Register	1
UNIMP	Unimplemented Instruction	1
IFLUSH	Instruction Cache Flush	1
FPop	Floating Point Unit Operations	1 to Launch
CPop	Coprocessor Operations	1 to Launch

* privileged instruction
** assumes delay slot is filled with a useful instruction

Table 2–17. SPARC Instruction Set (Continued)

When a Trap is generated the tt field of the Trap Base Register (TBR) is 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.

An FBfcc instruction (except FBA and FBN) evaluates the floating-point condition codes (fcc) according to the condition field. If the condition code, shown in *Table 2–14* is true the branch is taken and the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 * sign ext (disp22))." If the condition code is false, the branch is not taken.

Cond	Test
0000	Never
0001	Equal
0010	Less than or equal
0011	Less than
0100	Less than or equal, unsigned
0101	Carry set (greater than or equal, unsigned)
0110	Negative
0111	Overflow set
1000	Always
1001	Not equal
1010	Greater than
1011	Greater than or equal
1100	Greater than, unsigned
1101	Carry clear (less than, unsigned)
1110	Positive
1111	Overflow clear

Table 2–18. Bicc and Ticc Condition Codes

Cond	Test
0000	Never
0001	Not equal
0010	Less than or greater than
0011	Un-ordered or less than
0100	Less than
0101	Un-ordered or greater than
0110	Greater than
0111	Un-ordered
1000	Always
1001	Equal
1010	Un-ordered or equal
1011	Greater than or equal
1100	Un-ordered or greater than or equal
1101	Less than or equal
1110	Un-ordered or less than or equal
1111	ordered

Table 2–19. FBfcc Condition Codes

If the branch is not taken and the a (annul) field is set, the delay instruction is not executed (annulled). If the branch is taken, the annul field is ignored and the delay instruction is executed.

FBN (Branch Never) acts like a "NOP", except that if the annul field is one, the delay instruction is not executed (annulled). If the annul field is zero, the delay instruction is executed.

FBA (Branch Always) causes a transfer of control, irrespective of the value of the condition code bits. If the annul field is one, the delay instruction is not executed (annulled). If the annul field is zero, the delay instruction is executed.

An FBfcc instruction generates an fp disabled trap (and does not branch on annul) if the PSR's (Program Status Register's) EF (Enable Floating-point) bit is reset or if the FPU is not present.

Except for FBA, all FBfcc instructions with a=1 annul the delay instruction when the branch is not taken. However, FBA with a=1 does the reverse: it annuls the delay instruction even though the branch is taken. Note that the instruction executed immediately before an FBfcc must not be a floating-point instruction.

A CBccc instruction (except CBA and CBN) evaluates the coprocessor condition codes (supplied by the coprocessor on CCC[1:0]) according to the cond field. If the condition code is true the branch is taken and the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 * sign ext (disp22))." If the condition code is false, the branch is not taken and the instruction acts like a "NOP."

If the branch is not taken and the a (annul) field is set, the delay instruction is not executed (annulled). If the branch is taken, the annul field is ignored and the delay instruction is executed.

CBN (Branch Never) acts like a "NOP", except that if the annul field is one, the delay instruction is not executed (annulled). If the annul field is zero, the delay instruction is executed.

CBA (Branch Always) causes a transfer of control, irrespective of the value of the condition code bits. If the annul field is one, the delay instruction is not executed (annulled). If the annul field is zero, the delay instruction is executed.

A CBccc instruction generates a cp disabled trap (and does not branch or annul) if the PSR's EC bit is reset or if no coprocessor is present.

Except for CBA, all CBccc instructions with a=1 annul the delay instruction when the branch is not taken. However, CBA with a=1 does the reverse: it annuls the delay instruction even though the branch is taken. A CBccc instruction must be immediately preceded by a non-coprocessor instruction.

Processor Pipeline

The IU uses a four-stage instruction pipeline. A basic single-cycle instruction enters the pipeline and completes four cycles later. During these four cycles, three more instruction may enter the pipeline. This way, after a 4-cycle delay required to fill the pipeline, one single-cycle instruction enters the pipeline, and one (single-cycle) instruction exits the pipeline (completes) every cycle. For example, in a stream of 10 single-cycle instructions, the first enters the pipeline during T1 (the clock cycle when the first instruction is fetched), and the last exits the pipeline after T14.A single-cycle instruction does not really complete in one cycle; it actually completes in four cycles. They are called "single-cycle" instructions because a series of single-cycle instructions will complete one-per-cycle after an initial four-cycle delay.A single-cycle instruction normally goes through the following four stages of the pipeline in four clock cycles:1. Fetch (F) The processor fetches the instruction and places it in the instruction register.2. Decode (D) The processor decodes the instruction, and reads the operands out from the register file.3. Execute (E) The processor executes the instruction and saves the results in (the processor's) temporary registers.4. Write (W) The processor updates the destination register, provided no traps or exceptions are raised during the execution of the instruction.

Opcode	'Cond'	CCC[1:0] Test
CBN	'0000'	Never
CB123	'0001'	1 or 2 or 3
CB12	'0010'	1 or 2
CB13	'0011'	1 or 3
CB1	'0100'	1
CB23	'0101'	2 or 3
CB2	'0110'	2
CB3	'0111'	3
CBA	'1000'	Always
CB0	'1001'	0
CB03	'1010'	0 or 3
CB02	'1011'	0 or 2
CB023	'1100'	0 or 2 or 3
CB01	'1101'	0 or 1
CB013	'1110'	0 or 1 or 3
CB012	'1111'	0 or 1 or 2

Table 2–20. CBccc Condition Codes

Floating-point operations FPops take 1 cycle in the integer unit, plus additional cycles in the FPU. For the number of cycles each takes in the FPU, refer to the CY7C608 Floating-point controller and TMS8847 chip specifications.

Processor Data Types

The architecture defines nine data types; these appear in *Figure 2–12*. The integer types include byte, unsigned byte, halfword, unsigned halfword, word and unsigned word. The ANSI/IEEE 754–1985 floating-point types include single, double, and extended. A byte is 8 bits wide, a halfword is 16 bits, a word is 32 bits, a double is 64 bits, and an extended is 128 bits.

The floating-point double type includes two subfields: 1) the double-e, which contains the sign, exponent, and high-order fraction, and 2) the double-f, which includes the low-order fraction. The floating-point extended type includes 4 subfields: 1) the extended-e, which contains the sign and exponent, 2) the extended-f, which contains the integer part of the mantissa, and the high-order part of the fraction, 3) the extended-f-low, which contains the low-order fraction, and 4) the extended-u which is unused.

Table 2–17 shows the single-precision floating-point format containing one sign bit, eight bits of biased exponent and 23 bits of fraction. The double-precision format is shown in *Table 2–18*. Double-precision numbers are represented with one sign bit, eleven bits of biased exponent and 52 bits of fraction.

The extended-precision format is shown in *Table 2–19*. Extended-precision numbers are represented with one sign bit, fifteen bits of biased exponent and 63 bits of fraction.

The architecture does not define or create results with $0 < e < 32767$, $j = 0$.

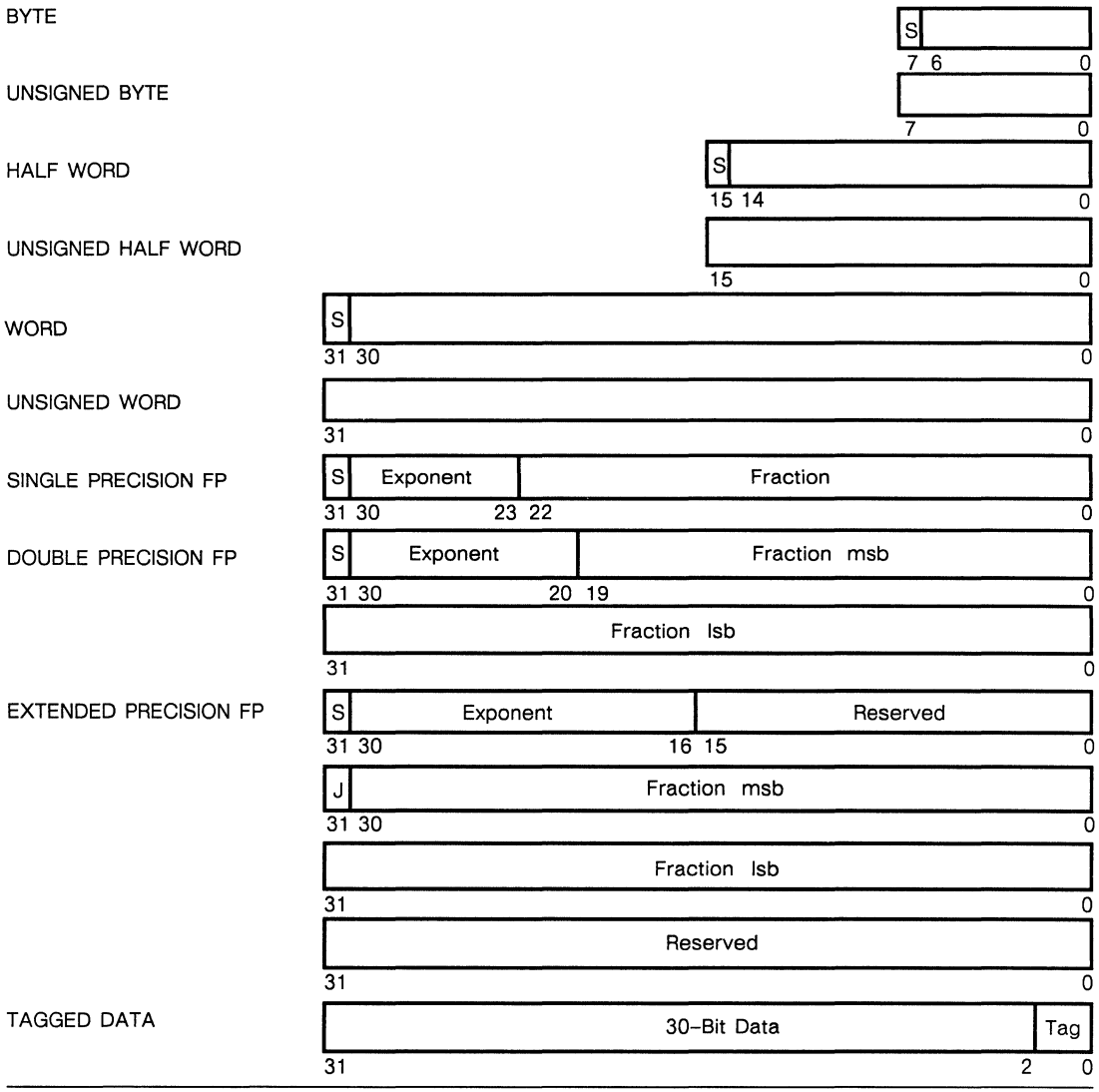


Figure 2–13. Processor Data Types

Traps

SPARC supports three types of traps: synchronous, floating-point/coprocessor and asynchronous (asynchronous traps are also called interrupts).

Synchronous traps are caused by an instruction, and occur before the instruction is completed.

Subfield	Address
Double-e	n
Double-f	n + 4
Extended-e	n
Extended-f	n + 4
Extended-f-low	n + 8
Extended-u	n + 12

Table 2–21. Arrangement of Double and Extended Data Types in Memory

s = sign (1) e = biased exponent (8) f = fraction (23)	
normalized number (0 < e < 255): subnormal number (e = 0): zero (e = 0):	(–1) s * 2 e–127 * 1.f (–1) s * 2 –126 * 0.f (–1) s * 0
signaling NaN: quiet NaN: infinity:	s = u ; e = 255 (max); f = .0 uuu uu (at least one bit must be nonzero) s = u ; e = 255 (max); f = .1 uuu uu s = u ; e = 255 (max); f = .000 00 (all zeroes)

Table 2–22. Single-Precision Floating-Point Format

s = sign (1) e = biased exponent (11) f–msb f–lsb = f = fraction (52)	
normalized number (0 < e < 2047): subnormalized number (e = 0): zero (e = 0):	(–1) s * 2e–1023 * 1.f (–1) s * 2–1022 * 0.f (–1) s * 0
signaling NaN: quiet NaN: infinity:	s = u ; e = 2047 (max); f = .0 uuu uu (at least one bit must be non-zero) s = u ; e = 2047 (max); f = .1 uuu uu s = u ; e = 2047 (max); f = .000 00 (all zeros)

Table 2–23. Double-Precision Floating-Point Format

Floating-point/coprocessor traps are caused by a floating-point operate (FPop) or coprocessor (CPop) instruction, and occur before the instruction is completed. However, due to the concurrent operation of the IU and the FPU, other non-floating-point instructions may have executed in the meantime.

Asynchronous traps occur when an external event interrupts the processor; they are not related to any particular instruction and occur between the execution of instructions.

s = sign (1) e = biased exponent (15) j = integer part (1) f-msb f-lsb = f = fraction (63)	
normalized number (0 < e < 32767; j = 1): subnormal number (e = 0; j = 0): zero (s = 0; e = 0):	(-1) s * 2 e-16383 * j.f (-1) s * 2 -16383 * j.f (-1) s * 0
signaling NaN: quiet NaN: infinity:	s = u; e = 32767 (max); j = u; f = .0 uuu uu (at least one bit must be nonzero) s = u; e = 32767 (max); j = u; f = .1 uuu uu s = u; e = 32767 (max); j = u; f = .000 00 (all zeroes)

Table 2–24. Extended-Precision Floating-Point Format

Synchronous and floating-point/coprocessor traps are generally taken before the instruction changes any processor or system state visible to a programmer; they happen "between" instructions. Instructions which access memory twice (double loads and stores and atomic instructions) are the only exceptions.

Traps transfer control to an offset within a table. The base address is specified in the trap base register (TBR), and the offset depends on the type of trap. Reset traps, however, cause the processor to transfer control to address 0. Because the program counters are not updated until after an instruction completes, the trap hardware captures both program counters and guarantees that the PC points to either the instruction that caused a synchronous trap, or to the instruction that was about to execute when a floating-point/coprocessor or asynchronous trap occurred. For floating-point/coprocessor traps, the instruction that caused the trap is in the floating-point queue (FQ) or the coprocessor queue (CP), and the PC will usually not point to it.

Trap	Priority	Trap Type (tt)	Synchronous or Asynchronous
Reset	1	–	Async
Instruction Access Exception	2	1	Sync
Illegal Instruction	3	2	
Privileged Instruction	4	3	
Floating Point Disabled	5	4	
Window Overflow	6	5	
Window Underflow	7	6	
Memory Address Not Aligned	8	7	
Floating Point Execption	9	8	
Data Access Execption	10	9	
Tag Overflow	11	10	Sync
Trap Instructions (Ticc)	12	128–255	Sync
Interrupt Level 15	13	31	Async
Interrupt Level 14	14	30	
Interrupt Level 13	15	29	
Interrupt Level 12	16	28	
Interrupt Level 11	17	27	
Interrupt Level 10	18	26	
Interrupt Level 9	19	25	
Interrupt Level 8	20	24	
Interrupt Level 7	21	23	
Interrupt Level 6	22	22	
Interrupt Level 5	23	21	
Interrupt Level 4	24	20	
Interrupt Level 3	25	19	
Interrupt Level 2	26	18	
Interrupt Level 1	27	17	Async

Table 2–25. Trap Type and Priority Assignment

Traps are described in detail in the chapter Traps, Exceptions, and Error Handling.

CHAPTER 3
Pipeline Operation

The CY7C600 achieves its very high performance by executing instructions at a rate approaching one instruction per clock cycle. The one instruction per clock is achieved by separating the execution of each instruction into four pipelined stages and executing all four stages in parallel. This chapter describes the operation of this pipelined method of instruction execution.

Pipeline Stages

The IU uses a four-stage instruction pipeline. A basic single-cycle instruction enters the pipeline and completes four cycles later. During these four cycles, three more instruction may enter the pipeline. This way, after a 4-cycle delay required to fill the pipeline, one single-cycle instruction enters the pipeline, and one (single-cycle) instruction exits the pipeline (completes) every cycle. For example, in a stream of 5 single-cycle instructions, the first enters the pipeline during T1 (the clock cycle when the first instruction is fetched), and the last exits the pipeline after T8 as shown in *Figure 3–1*.

Single Cycle Instructions

A single-cycle instruction does not really complete in one cycle; it actually completes in four cycles. They are called "single-cycle" instructions because the processor ill complete one instruction per cycle after an initial four-cycle delay.

A single-cycle instruction normally goes through the following four stages of the pipeline in four clock cycles:

1. **Fetch (F)**—The processor outputs the instruction address, fetches the instruction, and places it in the instruction register.
2. **Decode (D)**—The processor decodes the instruction, reads the operands from the register file, and computes the next instruction address.
3. **Execute (E)**—The processor executes the instruction and saves the results in (the processor's) temporary registers. Pending traps are prioritized and taken during this stage.
4. **Write (W)**—The processor updates the destination register, provided no traps or exceptions are raised during the execution of the instruction.

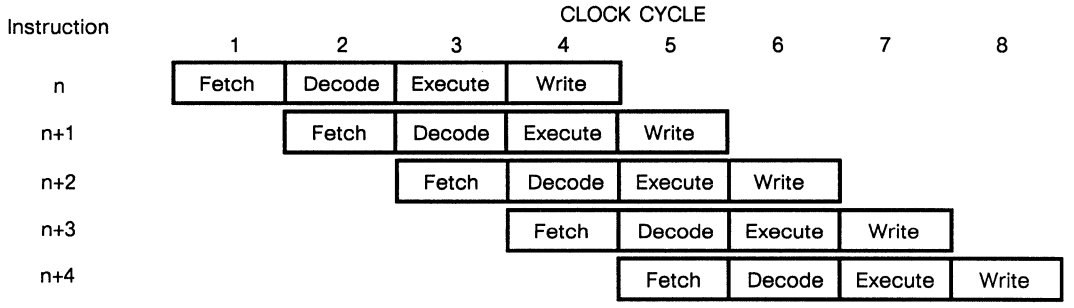


Figure 3–1. Pipeline with All Single Cycle Instructions

Multi-Cycle Instructions

Some instructions take extra cycles to complete. For example, a double-cycle instruction has one extra cycle, and a three-cycle instruction has two extra cycles. These extra cycles delay the pipeline long enough to complete, adding 1, 2, 3, or 4 extra cycles. For example, if a double-cycle instruction occurs in the above-mentioned 5-instruction stream, the last instruction in the stream will complete on T9. A single-cycle instruction moves through the pipeline as shown in *Figure 3–1*. Multiple-cycle instructions delay the entire pipeline, so that if the first instruction has one extra cycle, it holds up the pipeline as shown in *Figure 3–2*. Typically, instructions contain delay cycles because the processor needs to use the bus for something other than for fetching the next instruction (a data load or store to memory for example) and therefore for example in *Figure 3–2*. it cannot fetch instruction n+3 during T4. Instead, it fetches instruction n+3 during the last extra cycle (T5 in this case).

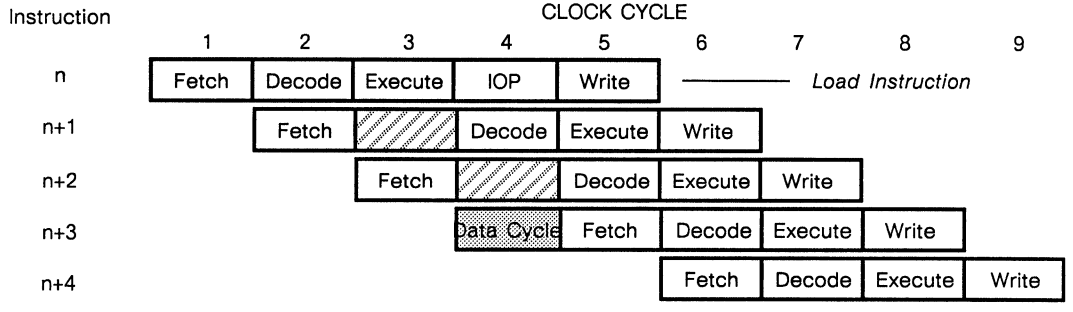


Figure 3–2. Pipeline with One Double Cycle Instruction (Load)

Internally Generated Opcodes

The extra cycles of multi-cycle instructions are actually generated by internal opcodes (IOPs) which are inserted in the pipeline as needed as shown in *Figure 3–3*. So that external instructions may continue to be fetched while internal opcodes are injected into the pipeline, a two-stage

prefetch buffer is used ahead of the pipeline. Only two stages are required because during any multi-cycle instruction a maximum of two extra cycles are available for fetching instructions. The buffers are used only as necessary to keep the pipeline full and fully utilize the external data bus bandwidth.

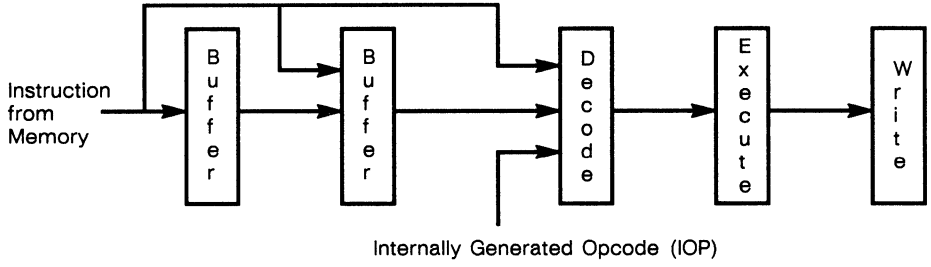


Figure 3–3. Processor Instruction Pipeline

Multi-cycle instructions may use up to three internally-generated opcodes to complete execution. A summary of internal opcode generation is included in *Table 3–1*.

Instruction	Number of Internal Opcodes
Single Loads	1
Double Loads	2
Single Stores	2
Double Stores	3
Atomic Load/Store	3
Jump	1
Return from Trap	1

Table 3–1. Internally Generated Opcodes

An example of a three cycle instruction is shown in *Figure 3–4*. This is a single word store instruction which contains two internal opcodes; one to allow external hardware time to determine whether or not the location to be written is in the cache, and the second to perform the actual data bus write cycle. The final write phase of the instruction is used to update internal processor status.

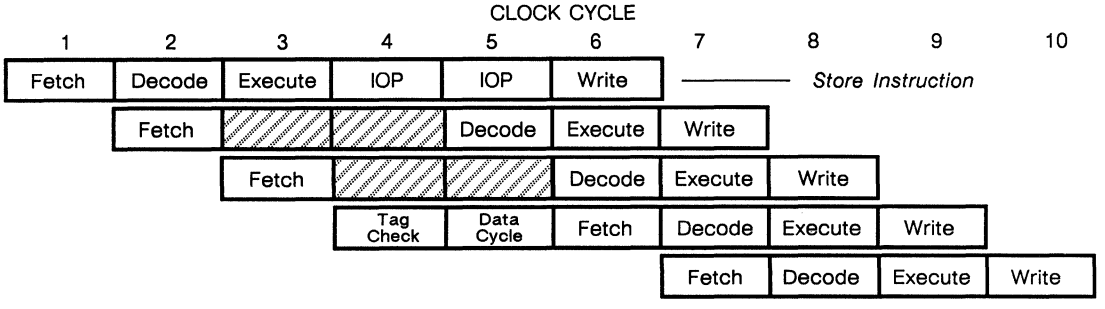


Figure 3–4. Pipeline with One Triple Cycle Instruction (Store)

Register Load Interlocks

Because of the pipelined execution of instructions, it is possible that an instruction may try to use the contents of a particular register which is in the process of being updated by a previous instruction. Special bypass paths in the pipeline of the CY7C601 make the correct data available to following instructions for all internal register to register operations but cannot solve the problem for loads to the registers from external memory. For this case register load interlock hardware prevents an instruction following a Load instruction from reading the register being loaded until the load is complete. Thus when a Load is followed by an instruction which reads the contents of that register the CY7C601 automatically generates a 1-cycle delay as shown in *Figure 3–5*.

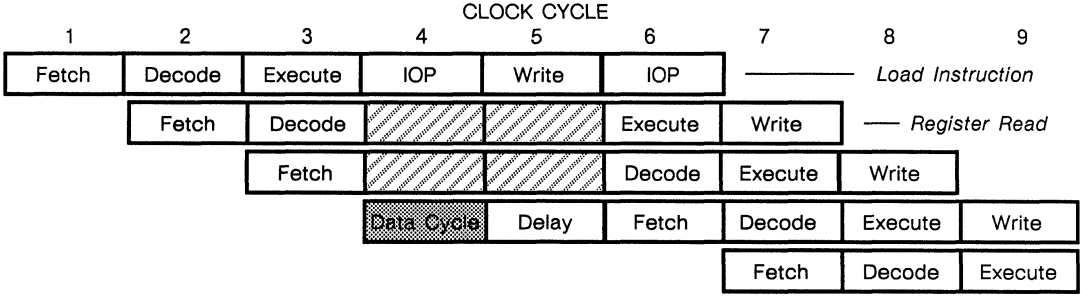


Figure 3–5. Pipeline with Hardware Interlock

In general, to maximize performance, compilers and assembly language programmers should avoid Loads followed immediately by instructions using the register contents.

Delayed Branch Instructions

The delayed branch instruction mechanism of the CY7C600 processor allows branches (whether taken or untaken) to occur without causing any delays in the pipeline as shown in *Figure 3–6*. If

the compiler or programmer cannot place an appropriate instruction in the delay instruction slot after the branch, the instruction fetched is annulled as shown in *Figure 3–7.*

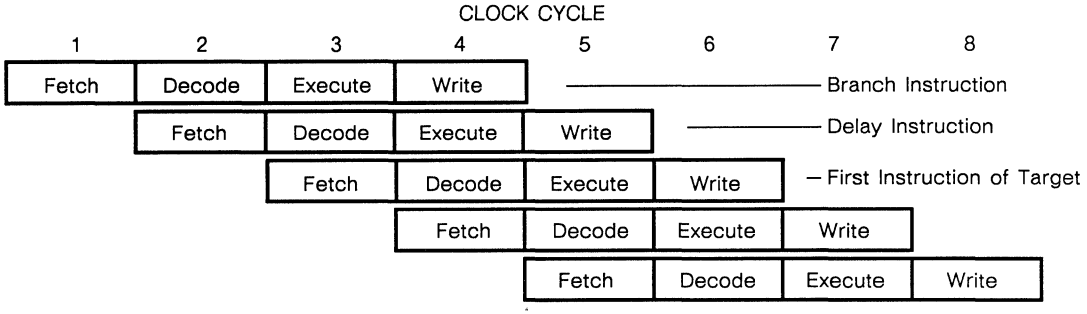


Figure 3–6. Pipeline During Branch Instruction

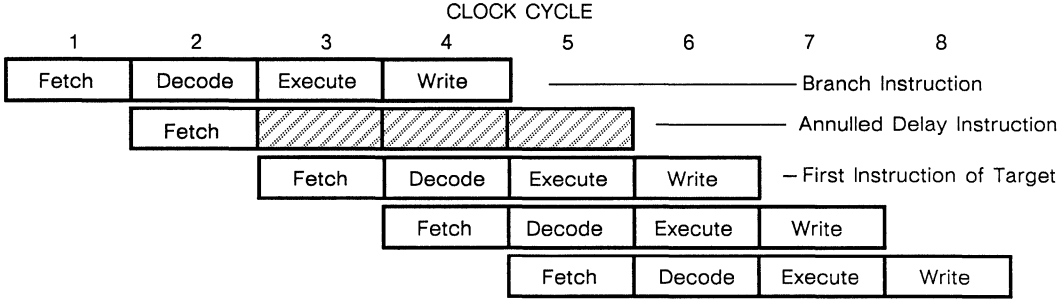


Figure 3–7. Branch with Annulled Delay Instruction

Pipeline Freezes

Whenever the processor pipeline is frozen as the result of an externally generated hold input such as MHOLD or BHOLD, the pipeline stalls in the execute phase of the instruction that caused the hold as shown in *Figure 3–8.*

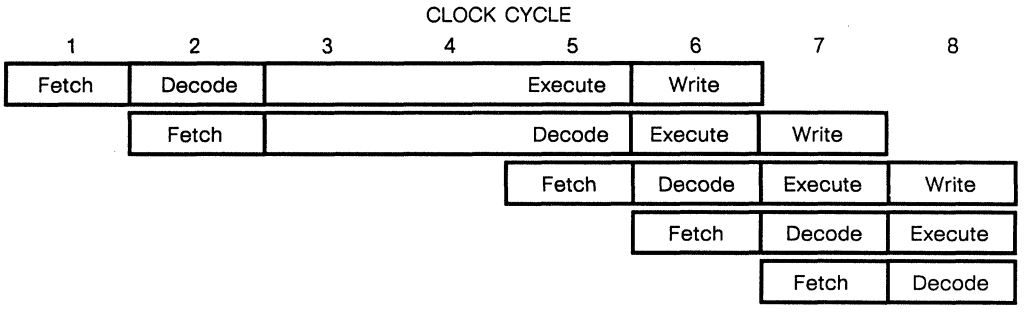


Figure 3–8. Pipeline Frozen by External Hold

Traps

Pending Traps whether synchronous or asynchronous are prioritized and taken during the execute phase of each instruction as shown in *Figure 3–9*. Instructions in the pipeline after detection of the trap are annulled and the first instruction of the trap target routine is executed in the fourth cycle following detection.

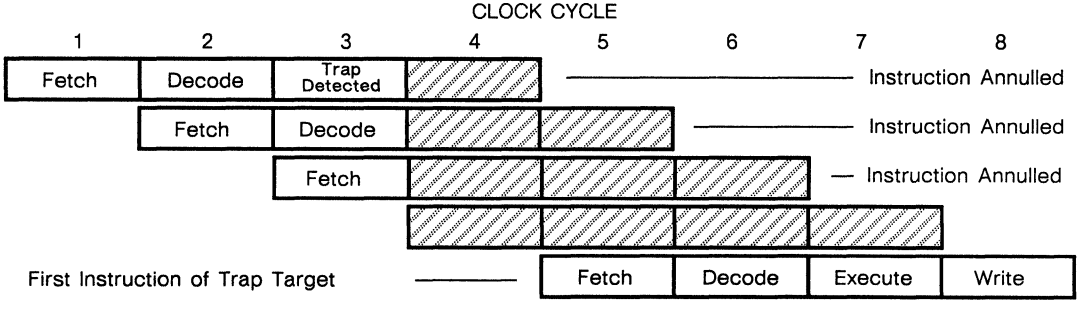


Figure 3–9. Pipeline Operation for Taken Trap

CHAPTER 4

Interrupts, Traps, and Exceptions

The CY7C601 Integer Unit generates traps in response to both internal and external events. These traps switch control from the instruction stream to an address in a trap table. The only exception is the reset trap which transfers program control to virtual address 0.

Trap Categories

Traps fall into two categories: synchronous and asynchronous. Synchronous traps are caused by hardware or by the Trap on Integer Condition Code (Ticc) instructions; they occur during the instruction that caused them. Asynchronous traps are caused by interrupt requests on inputs IRL[3:0]; they are synchronized by the IU and serviced after the current instruction has completed.

Synchronous Traps

The CY7C601 IU generates synchronous traps in response to internal conditions, external signals, or Trap (Ticc) instructions. These traps are taken immediately and the instruction that caused the trap is aborted BEFORE it changes any state in the processor. Synchronous traps may be caused by external hardware such as memory system. *Table 4–1* lists the synchronous traps that occur in response to external signals:

Trap	Initiating Signal	Condition
Data Access Exception	$\overline{\text{MEXC}}$	Memory error during data access
Instruction Access Exception	$\overline{\text{MEXC}}$	Memory error during instruction access
Floating Point Exception	$\overline{\text{FEXC}}$	Floating point unit error
Coprocessor Exception	$\overline{\text{CEXC}}$	Coprocessor unit error

Table 4–1. Externally Generated Synchronous Exception Traps

Asynchronous Traps

Asynchronous traps occur in response to the Interrupt Request (IRL[3:0]) inputs. These traps wait for the currently executing instruction to complete before they are processed. The trap type (tt) value for the trap is determined by the value of IRL[3:0], and the contents of the trap table at the offset specified in *Table 4–2*. For the interrupt to be taken, IRL[3:0] must be greater than the value in the Processor Interrupt Level (PIL) field of the Processor State Register (PSR).

The IRL[3:0] inputs provide 15 levels of interrupts. IRL[3:0]=0000 signifies no interrupts while IRL[3:0]=1111 signifies a non-maskable interrupt. All other IRL combinations represent interrupt requests which can be masked by the PIL field in the PSR.

IRL[3:0] are synchronized by 2 levels of D-type flip flops in the IU. The outputs of the two levels must agree before the interrupt can be processed. If the outputs of the two synchronizing levels disagree, the interrupt request will be ignored. This logic filters out transients on the IRL lines.

Trap Addressing

The Trap Base Register (TBR) generates the exact address of a trap handling routine. When a trap (other than some varieties of reset trap) occurs, the hardware writes a value into the trap type (tt) field of the TBR. This uniquely identifies the trap and serves as an offset into the table whose starting address is given by the TBA field of the TBR.

For most traps, updating the TBR is accomplished by simply writing the number from the following tt assignment table into the tt field of the TBR. However, for Ticc instructions, the tt value is calculated by the IU before it is written into the tt field in the TBR.

Trap Types and Priorities

Each type of trap is assigned a priority; when multiple trap requests occur, the highest priority trap is taken, and lower priority traps are ignored. To ensure IU recognition, lower priority trap requests must either persist or be repeated.

The following table shows the trap types, priorities, and assignments.

Trap	Priority	Trap Type (tt)	Synchronous or Asynchronous
Reset	1	–	Async
Instruction Access	2	1	Sync
Illegal Instruction	3	2	
Privileged Instruction	4	3	
Floating-Point Disabled	5	4	
Window Overflow	6	5	
Window Underflow	7	6	
Memory Address Not Aligned	8	7	
Floating-Point Exception	9	8	
Data Access Exception	10	9	
Tag Overflow	11	10	Sync
Trap Instructions (Ticc)	12	128–255	Sync
Interrupt Level 15	13	31	Async
Interrupt Level 14	14	30	
Interrupt Level 13	15	29	
Interrupt Level 12	16	28	
Interrupt Level 11	17	27	
Interrupt Level 10	18	26	
Interrupt Level 9	19	25	
Interrupt Level 8	20	24	
Interrupt Level 7	21	23	
Interrupt Level 6	22	22	
Interrupt Level 5	23	21	
Interrupt Level 4	24	20	
Interrupt Level 3	25	19	
Interrupt Level 2	26	18	
Interrupt Level 1	27	17	Async

Table 4–2. Trap Type and Priority Assignments

Trap Operation

A trap causes the following action:

- It disables traps (ET=0).
- It copies the S field of the PSR into the PS field and then sets the S field to 1.
- It decrements the CWP by 1, modulo the number of implemented windows.
- It saves the PC and nPC into r[17] and r[18], respectively, of the new window.
- It sets the tt field of the TBR to the appropriate value.

If the trap is not a reset, it writes the PC with the contents of TBR, and the nPC with the contents of TBR + 4. If the trap is a reset, it loads the PC with 0 and the nPC with 4.

Unlike many other processors, the SPARC architecture does not automatically save the PSR into memory during a trap. Instead, it saves the volatile S field into the PSR itself and the remaining fields are either altered in a reversible manner (ET and CWP), or should not be altered in a trap handler until the PSR has been saved into memory. The last two instructions of a trap handler should be JMPL followed by RETT. This restores the PC, the nPC and the S bit of the PSR.

Because the FPU and IU operate concurrently, the address that is saved from the PC as a result of a floating-point exception may not be the address of the FPop that caused the exception. If a floating-point exception occurs, the first element in the FQ points to the FPop that caused the exception, and the remaining elements point to FPops that have been started by the FPU but have not yet completed. These can be re-executed or emulated. For additional information on trap handlers, see Appendix C.

Interrupt Detection

As long as ET = 1, the IU checks for interrupts. It compares the external interrupt level (IRL[0:3]) against the PIL field of the PSR, and if IRL[0:3] is greater than the PIL, or if IRL[0:3] is 15 (unmaskable), then a trap occurs at the level requested by IRL[0:3].

Interrupt Response Timing

The Integer Unit samples the IRL[0:3] inputs at the rising edge of every clock. In order to properly synchronize these asynchronous inputs, the lines must be active for two consecutive clock edges to be accepted as valid inputs by the processor. Once the IRL input has been accepted, it is prioritized and the appropriate trap is taken during the next execute stage of the instruction pipeline. Best case interrupt response occurs when the interrupt is applied one clock plus one setup time before the execute phase of any instruction as shown in *Figure 4–1*. In this case the first instruction of the interrupt service routine is fetched during the fourth clock following the application of IRL.

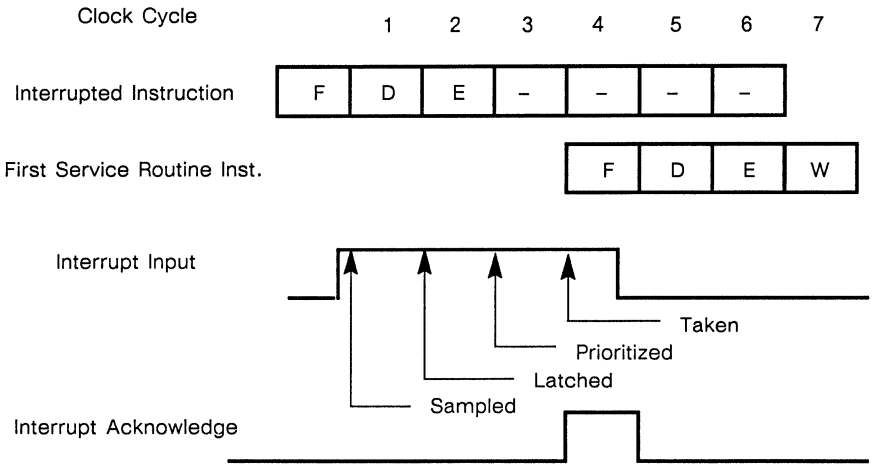


Figure 4–1. Best Case Interrupt Response Timing

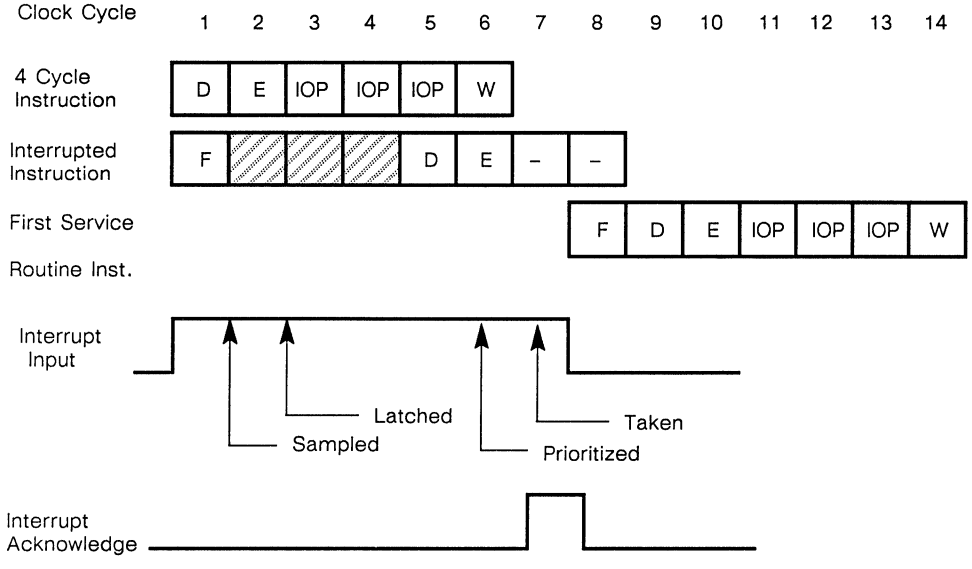


Figure 4–2. Worst Case Interrupt Response Timing

The worst case interrupt response occurs when the detection of the IRL input just misses the execute stage of a four-cycle instruction such as store double or atomic load/store as shown in *Figure 4–2*. In this case, the interrupt input must wait an additional 3 cycles for the next pipeline execute phase to occur. Also, the IRL input just misses the sampling clock edge resulting in an additional clock delay. As a result, the first instruction of the service routine is fetched in the eighth clock following the application of IRL.

Both the above best and worst case interrupt timing assumes that the processor is not stopped via the application of an external hold signal, and the IRL input is not blocked by the occurrence of a synchronous (internal) trap.

Interrupt Acknowledge

As shown in *Figure 4–1* and *Figure 4–2*, the Interrupt Acknowledge output pin goes active when an interrupt is taken, not when it is first detected and latched. This delay creates the possibility of ambiguity regarding which acknowledge is associated with which applied interrupt if IRL is not held stable until INTACK is received.

Floating-Point/Coprocessor Exception Traps

Floating-point/coprocessor exception traps are considered a separate class of traps because they are both synchronous and asynchronous. They are asynchronous because they occur sometime after the floating-point or coprocessor instruction that caused the exception. However, they are also synchronous because a floating-point or coprocessor instruction must be encountered in the instruction stream before the trap is taken.

When the FPU or Coprocessor recognizes an exception condition, it enters an "exception pending mode" state, and remains in this state until the IU takes the fp exception trap. When the IU takes the exception trap, the FPU leaves "exception pending" state, and enters "exception mode" state. The FPU or coprocessor remains in the exception mode state until the floating-point or coprocessor queue has been emptied by execution of one or more STDFQ or STDCQ instructions.

The PC that corresponds to a floating-point or coprocessor exception always points to a floating-point or coprocessor instruction since these exceptions are not recognized by the IU until the next floating-point or coprocessor instruction is encountered following the instruction which generated the exception. Therefore, the exception itself is always due to a previously executed floating-point or coprocessor instruction. The instruction which actually caused the trap and the value of the PC from which it was fetched are available in the floating-point (or coprocessor) queue.

Trap Descriptions

The following paragraphs describe the various traps, and the conditions that cause them.

Reset

A reset trap occurs when the IU leaves reset mode and enters execute mode. This is controlled by the reset input signal. The IU enters reset mode when the reset input signal is active, and enters execute mode when the reset input signal goes inactive. Except in one situation, reset does not change the value of the tt field of the TBR; the exception is when a return from trap instruction is executed while traps are not enabled and the processor is not in supervisor mode

(see description of return from trap instruction in Appendix B). Also, a reset trap causes the IU to begin execution at location 0, regardless of the value of the TBR.

Reset traps set the PSR S bit to 1 and the ET bit to 0. All other PSR fields, and all other registers retain their values from the last execute mode, except that on power-up they are undefined. An instruction access exception trap occurs when the memory exception input pin is active for a memory address used in an instruction fetch.

Illegal Instruction

This trap occurs:

1) when the UNIMP instruction is encountered

2) when an unimplemented instruction (FPops and CPops excluded) is encountered

3) when an instruction is fetched which, if executed, would result in an illegal processor state:

   a. Writing the Current Window Pointer (CWP) with a value greater than the number of implemented windows (7 for the CY7C608)

   b. Executing an Alternate Space instruction which has its i bit set to 1

   c. Executing an RETT instruction with ET=1 while in supervisor mode

   d. Executing an IFLUSH instruction with the input pin Flush=0

Unimplemented floating-point and unimplemented coprocessor instructions do not generate illegal instruction traps. They generate floating-point exceptions and coprocessor exception traps, respectively.

Privileged Instruction

This trap occurs when a privileged instruction is encountered while the S bit in the PSR = 0.

Floating-Point Disabled

This trap occurs when a FPop, FBfcc, or a floating-point load or store is encountered while either the EF bit in the PSR = 0 or no FPU is present as indicated by the processors Floating-Point Present (FP) input pin.

Coprocessor Disabled

This trap occurs when a CPop, CBccc, or a coprocessor load or store instruction is decoded while either the EC bit in the PSR = 0 or no coprocessor is present as indicated by the processors Coprocessor Present (CP) input pin.

Window Overflow

This trap occurs when a SAVE instruction would, if executed, cause the CWP to point to a window marked invalid in the WIM.

Window Underflow

This trap occurs when a RESTORE instruction would, if executed, cause the CWP to point to a window marked invalid in the WIM.

Memory Address not Aligned

This trap occurs when a load, store or JMPL instruction would, if executed, generate a memory address or a new PC value that is not properly aligned.

Floating–Point Exception

This trap occurs when the FPU is in exception pending state and a floating-point instruction (FP operate, floating-point load/store, FBfcc) is encountered in the instruction stream. The type of exception is encoded in the tt field of the FSR as described in the Registers section of chapter 2.

Coprocessor Exception

This trap occurs when the CP is in exception pending state and a coprocessor instruction (CP operate, coprocessor load/store, CBccc) is encountered in the instruction stream.

Data Access Exception

This trap occurs when the memory exception input pin is active for a memory address that corresponds to a data movement by a load or store instruction.

Tag Overflow

This trap occurs when a TADDccTV or TSUBccTV instruction is executed which causes the overflow bit of the integer condition codes to be set.

Trap Instruction

This trap occurs when a taken Ticc instruction is executed.

Interrupt Level

External interrupts are controlled by the value of IRL[3:0]. A value of 0 indicates that no interrupt is requested. Level 1 is the lowest priority interrupt and 15 is the highest. Interrupt level 15 cannot be masked by the Processor Interrupt Level (PIL) field of the PSR. When ET = 1, an external interrupt is recognized if IRL[3:0] = 15 or IRL[3:0] > PIL. When ET = 0 or (IRL[3:0] $\neq$ 15 and IRL[3:0] < PIL), no external interrupt is recognized.

CHAPTER 5

Pin Description

The Integer Unit's external signals fall into three categories: 1) memory subsystem interface signals, 2) floating-point unit/coprocessor interface signals, and 3) miscellaneous I/O signals. These are described in the following sections. Paragraphs after the tables describe each signal. Signals that are active LOW are marked with an overscore; all others are active HIGH. For example, $\overline{WE}$ is active LOW, while RD is active HIGH. The signals described in this chapter are summarized in *Figure 5–1* and in *Table 5–3* at the end of the chapter.

Memory Subsystem Interface Signals

The memory interface signals consist of 40 bits of address (32 bits of address and an 8-bit address space identifier), 32 bits of bidirectional data lines, 2 bits to identify the size (byte, halfword, word, or double word) of data bus transactions, and various control signals.

A[31:0] Address Bus

These 32 bits are the addresses of instructions or data and they are sent out "unlatched" by the Integer Unit. Assertion of the MAO signal during a cache miss (which is initiated by pulling one of the MHOLD inputs low) will force the Integer Unit to put the previous (missed) address on the address bus. A[31:0] pins are tristated if the $\overline{AOE}$ or $\overline{TOE}$ signal is deasserted (HIGH).

ASI[7:0] Address Space Identifier

These 8 bits are the Address Space Identifier for an instruction or data access to the memory. ASI[7:0] are sent out "unlatched" by the Integer Unit. The value on these pins during any given cycle is the Address Space Identifier corresponding to the memory address on the A[31:0] pins at that cycle. Assertion of the MAO signal during a cache miss (which is initiated by pulling one of the MHOLD inputs low) will force the Integer Unit to put the previous Address Space Identifier on the ASI[7:0] pins. ASI[7:0] pins are three-stated if the $\overline{AOE}$ or $\overline{TOE}$ signal is deasserted (HIGH). Normally, the encoding of the ASI bits is as shown in *Table 5–1*.

Address Space Identifier (ASI)	Address Space
Bit 7 0	
00001000	User Instruction
00001010	User Data
00001001	Supervisor Instruction
00001011	Supervisor Data

Table 5–1. ASI Bit Assignment

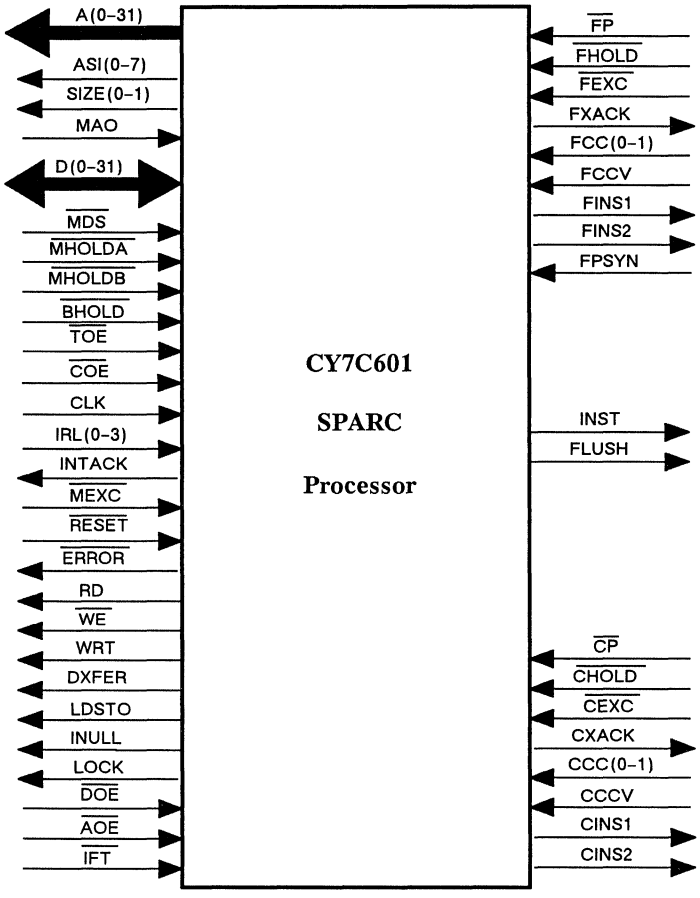


Figure 5–1. Integer Unit Pinout

D[31:0] Data Bus

D[31:0] is the bi-directional data bus to and from the Integer Unit. The data bus is driven by the Integer Unit during the execution of integer store instructions and the store cycle of atomic load store instructions. Similarly, the data bus is driven by the Floating-Point Unit only during the execution of floating-point store instructions.

The store data is sent out unlatched and must be latched externally before it is used. Once latched, store data is valid during the second data cycle of a store single access, the second and third data cycle of a store double access, and the third data cycle of an atomic load store access. The alignment for load and store instructions is done inside the processor. 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. D(31) corresponds to the most significant bit of the least significant byte of the 32-bit word. If a double word, word, or half word load or store instruction generates an improperly aligned address, a memory address not aligned trap will occur. Instructions and operands are always expected to be fetched from a 32-bit wide memory.

SIZE[1:0] Bus Transaction Size

These bits specify the data size associated with a data or instruction fetch. Size bits are sent out "unlatched" by the Integer Unit. The value on these pins at any given cycle is the data size corresponding to the memory address on the A[31:0] pins at that cycle. SIZE[1:0] remains valid on the bus during all data cycles of loads, stores, load doubles, store doubles and atomic load stores. Since all instructions are 32-bits long, SIZE[1:0] is set to "10" during all instruction fetch cycles. Encoding of the SIZE[1:0] bits is shown in *Table 5–2*.

Size 1	Size 0	Data Transfer Type
0	0	Byte
0	1	Halfword
1	0	Word
1	1	Word (Load/Store Double)

Table 5–2. Size Bit Assignment

$\overline{\text{MHOLD}}$ (A/B) Memory Holds

The processor pipeline will be frozen while $\overline{\text{MHOLDA}}$ is asserted and the IU outputs will revert to and maintain the value they had at the rising edge of the clock in the cycle before $\overline{\text{MHOLDA}}$ was asserted. $\overline{\text{MHOLDA}}$ is used to freeze the clock to both the Integer and Floating-Point Units during a cache miss (for systems with cache), or when slow memory is accessed. This signal must be presented to the processor chip at the beginning of each processor clock cycle and be stable during the high time of the processor clock. Either $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$ can be used for stopping the processor during a cache miss or memory exception.

$\overline{\text{MHOLDB}}$ has the same definition as $\overline{\text{MHOLDA}}$. The processor hardware uses the logical "OR" of all hold signals (i.e., $\overline{\text{MHOLDA}}$, $\overline{\text{MHOLDB}}$ and $\overline{\text{BHOLD}}$) to generate a final hold signal for freezing the processor pipeline. All HOLD signals are latched (transparent latch with clock high) in the IU before they are used.

$\overline{\text{BHOLD}}$ Bus Hold

$\overline{\text{BHOLD}}$ is asserted by the I/O controller when an external bus master requests the data bus. Assertion of this signal will freeze the processor pipeline. External logic should guarantee that after deassertion of $\overline{\text{BHOLD}}$, the data at all inputs to the chip is the same as what it was before $\overline{\text{BHOLD}}$ was asserted. Since the IU processes the $\overline{\text{BHOLD}}$ input through a transparent latch before it is used, this signal must be presented to the processor chip at the beginning of each processor clock cycle. $\overline{\text{BHOLD}}$ should be used only for bus access requests by an external device since the $\overline{\text{MDS}}$ and $\overline{\text{MEXC}}$ signals are not recognized while this input is active.

$\overline{\text{MDS}}$ Memory Data Strobe

Assertion of this signal enables the clock input to the on-chip Instruction Register (during an instruction fetch) or to the Load Result Register (during a data fetch). In a system with cache, $\overline{\text{MDS}}$ is used to signal the processor when the missed data (cache miss) is ready on the bus. In a system with slow memories, $\overline{\text{MDS}}$ is used to signal the processor when the read data is available on the bus.

$\overline{\text{MDS}}$ must be asserted only while the processor is frozen by either the $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$ input signals. The IU samples the $\overline{\text{MDS}}$ signal via an on-chip transparent latch before it is used. The $\overline{\text{MDS}}$ signal is also used for strobing memory exceptions. In other words, $\overline{\text{MDS}}$ should be asserted whenever $\overline{\text{MEXC}}$ is asserted (see $\overline{\text{MEXC}}$ definition).

$\overline{\text{MEXC}}$ Memory Exception

This signal is asserted by the memory (or cache) controller to initiate an instruction (or data) exception trap. $\overline{\text{MEXC}}$ is latched in the processor at the rising edge of CLK and is used in the following cycle. If $\overline{\text{MEXC}}$ is asserted during an instruction fetch cycle an instruction access exception is generated. If $\overline{\text{MEXC}}$ is asserted during a data fetch cycle a data access exception trap is generated.

The $\overline{\text{MEXC}}$ signal is used during ($\overline{\text{MHOLD}}$) in conjunction with the $\overline{\text{MDS}}$ signal to indicate to the IU that the memory system was unable to supply valid instruction or data. If $\overline{\text{MDS}}$ is applied without $\overline{\text{MEXC}}$, the IU accepts the contents of the data bus as valid information. When $\overline{\text{MDS}}$ is applied with $\overline{\text{MEXC}}$ an exception trap is generated and the contents of the data bus is ignored by the IU. (i.e., $\overline{\text{MHOLD}}$ and $\overline{\text{MDS}}$ must be LOW when $\overline{\text{MEXC}}$ is asserted). $\overline{\text{MEXC}}$ must be de-asserted in the same clock cycle in which $\overline{\text{MHOLD}}$ is released.

MAO Memory Address Output

This signal is used during a MHOLD by the Integer Unit to select between the current memory access parameters and the previous (missed) memory parameters (i.e. the value of those parameters at the second rising edge of CLK before MHOLD was applied). A logic HIGH value at this pin during a cache miss causes the Integer Unit to put A[31:0], ASI[7:0], SIZE[1:0], RD, $\overline{\text{WE}}$, WRT, LDSTO, LOCK, and DXFER values corresponding to the missed memory address on the bus. Normally, MAO should be kept at a LOW level, selecting the access parameters for the current access. MAO should not be used during store cycle misses because the $\overline{\text{WE}}$ output would be lost.

$\overline{\text{AOE}}$ Address Output Enable

De-assertion of this signal three-states all output drivers associated with A[31:0] and ASI[7:0] outputs. $\overline{\text{AOE}}$ is connected directly to the output drivers of the address and ASI signals and must be asserted during normal operations. This signal should be de-asserted only when the bus is granted to another bus master (i.e., when either $\overline{\text{BHOLD}}$, $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$ is asserted).

$\overline{\text{DOE}}$ Data Output Enable

De-assertion of this signal three-states all output drivers of the data D[31:0] bus. $\overline{\text{DOE}}$ is connected directly to the data bus output drivers and must be asserted during normal operations. This signal should be de-asserted only when the bus is granted to another bus master (i.e., when either $\overline{\text{BHOLD}}$, $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$ is asserted).

$\overline{COE}$ Control Output Enable

De-assertion of this signal three-states all output drivers associated with SIZE[1:0], RD, $\overline{WE}$, WRT, LOCK, LDSTO and DXFER outputs. $\overline{COE}$ is connected directly to the output drivers and must be asserted during normal operations. This signal should be de-asserted only when the bus is granted to another bus master (i.e., when either $\overline{BHOLD}$, $\overline{MHOLDA}$ or $\overline{MHOLDB}$ is asserted).

RD Read

This signal specifies whether the current memory access is a read or write operation. It is sent out "unlatched" by the Integer Unit and must be latched externally before it is used. RD is set to "0" only during data cycles of store instructions including the store cycles of atomic load store instructions. This signal when used in conjunction with SIZE[1:0], ASI[7:0], and LDSTO, can be used to check access rights of bus transactions. In addition, the RD signal may be used to turn off the output drivers of data RAMs during a store operation. For atomic load store instructions the RD signal is "1" during the first data cycle (read cycle) and "0" during the second and third data cycles (write cycle).

$\overline{WE}$ Write Enable

This signal is asserted by the Integer Unit during the second data cycle of store single instructions, the second and third data cycles of store double instructions, and the third data cycle of atomic load store instructions. The $\overline{WE}$ signal is sent out "unlatched" and must be latched externally before it is used. The $\overline{WE}$ signal may be externally qualified by HOLD signals (i.e., $\overline{MHOLDA}$ and $\overline{MHOLDB}$) to avoid writing into the memory during memory exceptions.

WRT Advanced Write

This signal is asserted (set to "1") by the processor during the first data cycle of single or double integer store instructions, the first data cycle of single or double floating-point store instructions, and the second data cycle of atomic load/store instructions. WRT is sent out "unlatched" and must be latched externally before it is used.

LDSTO Load/Store

This signal is asserted by the Integer Unit during the data cycles (which include the load cycle and both cycles of the store operation) of atomic load store operations. LDSTO is sent out "unlatched" by the Integer Unit and must be latched externally before it is used.

LOCK

This signal is set to "1" when the processor needs the bus for multiple cycle transactions such as atomic load store, double loads and double stores. LOCK is sent "unlatched" and should be latched externally before it is used. The bus may not be granted to another bus master as long as LOCK signal is asserted (i.e., $\overline{BHOLD}$ should not be asserted in the following processor clock cycle when LOCK=1).

DXFER Data Transfer

This signal is asserted by the processor at the beginning of all bus data transfer cycles. DXFER is "unlatched" and DXFER = 1 indicates a data cycle.

INULL Integer Unit Nullify Cycle

Assertion of INULL indicates that the current memory access (whose address is held in an external latch) is to be nullified by the processor. INULL is intended to be used to disable cache misses (in systems with cache) and to disable memory exception generation for the current memory access (i.e., $\overline{MDS}$ and $\overline{MEXC}$ should not be asserted for a memory access when INULL=1).

INULL is a latched output and is active during the same cycle as the address which it nullifies. INULL is asserted under the following conditions: 1. During the second cycle of a store instruction. 2. Whenever the IU address is invalid due to an external or internal exception. If a Floating-Point Unit or coprocessor unit is present in the system INULL should be or'ed with the FNULL and CNULL signals from these units.

$\overline{IFT}$ Instruction Cache Flush Trap

The state of this pin determines whether or not execution of the IFLUSH instruction generates a trap. If $\overline{IFT}$ =1, then IFLUSH executes like a NOP with no side effects. If $\overline{IFT}$ =0, then IFLUSH causes an Unimplemented Instruction Trap.

Floating-Point/Coprocessor Interface Signals

The floating-point/coprocessor unit interface is a dedicated group of connections between the IU and the FPU/Coprocessor. Note that no external circuits are required between the IU and the FPU/Coprocessor; all traces should connect directly. The interface consists of the following signals, described below.

$\overline{FP}$ Floating-Point Unit Present

This signal indicates whether or not a Floating-Point Unit exists in the system. The $\overline{FP}$ signal is normally pulled up to VDD by a resistor. It is grounded when the FPU chip is present. The Integer Unit generates a floating-point disable trap if $\overline{FP}$ = 1 during the execution of a floating-point instruction, FBfcc instruction, or floating-point load and store instructions.

$\overline{CP}$ Coprocessor Unit Present

This signal indicates whether or not a Coprocessor exists in the system. The $\overline{CP}$ signal is normally pulled up to VDD by a resistor. It is grounded when the Coprocessor chip is present. The Integer Unit generates a Coprocessor Disable Trap if $\overline{CP}$ = 1 during the execution of a coprocessor instruction, CBccc instruction, or coprocessor load and store instructions.

FCC[1:0] Floating-Point Condition Codes

These bits are taken as the current condition code bits of the FPU. They are considered valid if FCCV=1. During the execution of the FBfcc instruction, the processor uses these bits to determine whether the branch should be taken or not. FCC[1:0] are latched by the processor before they are used.

CCC[1:0] Coprocessor Condition Codes

These bits are taken as the current condition code bits of the Coprocessor. They are considered valid if CCCV=1. During the execution of the CBccc instruction, the processor uses these bits to determine whether the branch should be taken or not. CCC[1:0] are latched by the processor before they are used.

FCCV Floating-Point Condition Codes Valid

This signal should be asserted only when the FCC[1:0] bits are valid. The Floating-Point Unit de-asserts FCCV if pending floating-point compare instructions exist in the floating-point queue. FCCV is re-asserted when the compare instruction is completed and the floating-point condition codes FCC[1:0] are valid. The Integer Unit will enter a wait state if FCCV is de-asserted (i.e., FCCV = "0"). The FCCV signal is latched (transparent latch) in the IU before it is used.

CCCV Coprocessor Condition Codes Valid

This signal should be asserted only when the CCC[1:0] bits are valid. The Coprocessor de-asserts CCCV if pending coprocessor compare instructions exist in the coprocessor queue. CCCV is re-asserted when the compare instruction is completed and the coprocessor condition codes CCC[1:0] are valid. The Integer Unit will enter a wait state if CCCV is de-asserted (i.e., CCCV = "0"). The CCCV signal is latched (transparent latch) in the IU before it is used.

$\overline{\text{FHOLD}}$ Floating-Point Hold

This signal is asserted by the Floating-Point Unit if a situation arises in which the FPU cannot continue execution. The Floating-Point Unit checks all dependencies in the Decode stage of the instruction and asserts $\overline{\text{FHOLD}}$ (if necessary) in the next cycle. This signal is used by the Integer Unit to freeze the instruction instruction pipeline in the same cycle. The FPU must eventually deassert $\overline{\text{FHOLD}}$ in order to unfreeze the Integer Unit's pipeline. The $\overline{\text{FHOLD}}$ signal is latched (transparent latch) in the IU before it is used.

$\overline{\text{CHOLD}}$ Coprocessor Hold

This signal is asserted by the Coprocessor if a situation arises in which the Coprocessor cannot continue execution. The Coprocessor checks all dependencies in the Decode stage of the instruction and asserts $\overline{\text{CHOLD}}$ (if necessary) in the next cycle. This signal is used by the Integer Unit to freeze the instruction pipeline in the same cycle. The Coprocessor must eventually deassert $\overline{\text{CHOLD}}$ in order to unfreeze the Integer Unit's pipeline. The $\overline{\text{CHOLD}}$ signal is latched (transparent latch) in the IU before it is used.

$\overline{\text{FEXC}}$ Floating-Point Exception

Assertion of this signal indicates that a floating-point exception has occurred. $\overline{\text{FEXC}}$ must remain asserted until the Integer Unit takes the trap and acknowledges the FPU via FXACK signal. floating-point exceptions are taken only during the execution of floating-point instructions, FBfcc instruction and floating-point load and store instructions. $\overline{\text{FEXC}}$ is latched in the Integer Unit before it is used. The FPU should deassert $\overline{\text{FHOLD}}$ if it detects an exception while $\overline{\text{FHOLD}}$ is asserted. In this case $\overline{\text{FEXC}}$ should be asserted a cycle before $\overline{\text{FHOLD}}$ is deasserted.

$\overline{\text{CEXC}}$ Coprocessor Exception

This signal is asserted by the Integer Unit whenever a new instruction is being fetched. It is used by the FPU or coprocessor to latch the instruction on the D[31:0] bus into the FPU or coprocessor instruction buffer. The FPU (or coprocessor) needs two instruction buffers (D1 and D2) to save the last two fetched instructions. When INST is asserted a new instruction enters into the D1 buffer and the old instruction in D1 enters into the D2 buffer.

INST Instruction Fetch

This signal is asserted by the Integer Unit whenever a new instruction is being fetched. It is used by the FPU or coprocessor to latch the instruction on the D[31:0] bus into the FPU or coprocessor instruction buffer. The FPU (or coprocessor) needs two instruction buffers (D1 and D2) to save the last two fetched instructions. When INST is asserted a new instruction enters into the D1 buffer and the old instruction in D1 enters into the D2 buffer.

FLUSH Floating-Point/Coprocessor Instruction Flush

This signal is asserted by the Integer Unit and is used by the FPU or Coprocessor to flush the instructions in its instruction registers. This may happen when a trap is taken by the Integer Unit. Instructions that have entered into the floating-point (or coprocessor) queue may continue their execution if FLUSH is raised as a result of a trap or exception other than floating-point (or coprocessor) exceptions.

FINS1 Floating-Point Instruction in Buffer 1

This signal is asserted by the Integer Unit during the Decode stage of an FPU instruction if the instruction is in the D1 buffer of the FPU chip. The FPU uses this signal to latch the instruction in D1 buffer into its Execute stage instruction register.

FINS2 Floating-Point Instruction in Buffer 2

This signal is asserted by the Integer Unit during the Decode stage of an FPU instruction if the instruction is in the D2 buffer of the FPU chip. The FPU uses this signal to latch the instruction in D2 buffer into its Execute stage instruction register.

CINS1 Coprocessor Instruction in Buffer 1

This signal is asserted by the Integer Unit during the Decode stage of a coprocessor instruction if the instruction is in the D1 buffer of the coprocessor chip. The coprocessor uses this signal to latch the instruction in D1 buffer into its Execute stage instruction register.

CINS2 Coprocessor Instruction in Buffer 2

This signal is asserted by the Integer Unit during the Decode stage of a coprocessor instruction if the instruction is in the D2 buffer of the coprocessor chip. The coprocessor uses this signal to latch the instruction in D2 buffer into its Execute stage instruction register.

FXACK Floating-Point Exception Acknowledge

This signal is asserted by the Integer Unit in order to acknowledge to the FPU that the current $\overline{\text{FEXC}}$ trap is taken. The FPU must deassert $\overline{\text{FEXC}}$ after it receives an asserted level of FXACK signal so that the next floating-point instruction does not cause a "repeated" floating-point exception trap.

CXACK Coprocessor Exception Acknowledge

This signal is asserted by the Integer Unit in order to acknowledge to the Coprocessor that the current $\overline{\text{CEXC}}$ trap is taken. The Coprocessor must deassert $\overline{\text{CEXC}}$ after it receives an asserted level of CXACK signal so that the next coprocessor instruction does not cause a "repeated" coprocessor exception trap.

Miscellaneous I/O Signals

These signals are used by the IU to control external events or to receive input from external events. This interface consists of the signals discussed below.

IRL[3:0] Interrupt Request Level

The data on these pins defines the External Interrupt Level. IRL[3:0]=0000 indicates that no external interrupts are pending. The Integer Unit uses two on-chip synchronizing latches to sample these signals. A given interrupt level must remain valid for at least two consecutive cycles to be recognized by the Integer Unit. IRL[3:0]=1111 signifies an non-maskable interrupt. All other interrupt levels are maskable by the PIL field of the Processor State Register (PSR). External interrupts should be latched and prioritized by the external logic before they are passed to the Integer Unit. The external interrupt latches should keep the interrupts pending until they are taken (and acknowledged) by the Integer Unit. External interrupts can be acknowledged by software or by the Interrupt Acknowledge (INTACK) output.

INTACK Interrupt Acknowledge

This signal is asserted by the Integer Unit when an external interrupt is taken.

$\overline{\text{RESET}}$ Integer Unit Reset

Assertion of this pin will reset the Integer Unit. The $\overline{\text{RESET}}$ signal must be asserted for a minimum of eight processor clock cycles. After a reset, the integer unit will start fetching from address 0. The $\overline{\text{RESET}}$ signal is latched by the Integer Unit before it is used.

$\overline{\text{ERROR}}$ Error State

This signal is asserted by the Integer Unit when a trap is encountered while traps are disabled via the ET bit in the PSR. In this situation the Integer Unit saves the PC and nPC registers, sets the tt value in the TBR, enters into an error state, asserts the $\overline{\text{ERROR}}$ signal and then halts. The only way to restart the processor trapped in the error state, is to trigger a reset by asserting the $\overline{\text{RESET}}$ signal.

$\overline{\text{TOE}}$ Test Mode Output Enable

This signal is used to tristate all output drivers of the processor chip. It is used to isolate the chip from the rest of the system for debugging purposes.

FPSYN Floating-Point Synonym Mode

This pin is a mode pin which is used to allow execution of additional instructions in future designs. It should be normally kept deasserted (FPSYN=0) to disable the execution of these instructions.

CLK Clock

CLK is a 50% duty-cycle clock used for clocking the IU's pipeline registers. It is HIGH during the first half of the processor cycle, and LOW during the second half. The rising edge of CLK defines the beginning of each pipeline stage in the IU chip.

Pin Name	Description	Input/Output	Active
A(0–31)	Address	3–State Output	
ASI(0–7)	Address Space Identifier	3–State Output	
D(0–31)	Data	3–State BiDir.	
MEXC	Memory Exception	Input	LOW
MHOLDA	Memory Hold A	Input	LOW
MHOLDB	Memory Hold B	Input	LOW
BHOLD	Bus Hold	Input	LOW
AOE	Address Output Enable	Input	LOW
DOE	Data Output Enable	Input	LOW
COE	Control Output Enable	Input	LOW
MDS	Memory Data Strobe	Input	LOW
MAO	Memory Address Output Sel.	Input	
IFT	Instruction Flush Trap	Input	LOW
SIZE(0–1)	Bus Transaction Size	3–State Output	
RD	Read	3–State Output	HIGH
WE	Write	3–State Output	LOW
LDSTO	Load/Store Operation	3–State Output	HIGH
INULL	Null Cycle	3–State Output	HIGH
LOCK	Multi–Cycle Bus Lock	3–State Output	HIGH
DFETCH	Data Fetch Cycle	3–State Output	HIGH
WRT	Advanced Write	3–State Output	HIGH
FP	FPU Present	Input w Pullup	LOW
FCC(0–1)	FPU Condition Codes	Input	
FCCV	Condition Codes Valid	Input	HIGH
FHOLD	FPU Hold	Input	LOW
FEXC	FPU Exception	Input	LOW
CP	Coprocessor Present	Input w Pullup	LOW
CCC(0–1)	CP Condition Codes	Input	
CCCV	Condition Codes Valid	Input	HIGH
CHOLD	CP Hold	Input	LOW
CEXC	CP Exception	Input	LOW
INST	Instruction Fetch Cycle	3–State Output	HIGH
FLUSH	Flush FP/CP Instruction	3–State Output	HIGH
FINS1	FP Instruction Stage 1	3–State Output	HIGH
FINS2	FP Instruction Stage 2	3–State Output	HIGH
FXACK	FP Exception Acknowledge	3–State Output	HIGH
CINS1	CP Instruction Stage 1	3–State Output	HIGH
CINS2	CP Instruction Stage 2	3–State Output	HIGH
CXACK	CP Exception Acknowledge	3–State Output	HIGH
IRL(0–3)	Interrupt Request Lines	Input	
INTACK	Interrupt Acknowledge	3–State Output	HIGH
RESET	System Reset	Input	LOW
ERROR	IU Error Mode	3–State Output	LOW
TOE	Test Mode Output Enable	Input	HIGH
FPSYN	FPU Synonym Mode	Input	
CLK	System Clock	Input	
VSSO	Output Driver Ground	Ground	
VCCO	Output Driver Power	Power	
VSSI	Main Internal Ground	Ground	
VCCI	Main Internal Power	Power	
VSST	Input Circuit Ground	Ground	
VCCT	Input Circuit Power	Power	

Table 5–3. Pinout Summary

CYPRESS SEMICONDUCTOR

CHAPTER 6

Bus Operation

This section describes standard bus operations. These operations include instruction fetch, load integer, load double integer, load floating-point, load double floating-point, store integer, store double integer, store floating-point, store double floating-point, atomic load/store unsigned byte, and floating-point operations (FPop).

Instruction Fetch

The processor sends out address bits A[31:0] at the beginning of the fetch cycle that must be latched externally. Data is expected at the end of the fetch cycle and it is latched from the data bus into the on-chip instruction register. If the external hardware cannot provide the instruction during the fetch cycle, wait states must be used, as described in the Memory Interface chapter. The first clock cycle in *Figure 6–1* shows an instruction fetch. Since all instruction fetches are single-cycle operations, the pipe line runs smoothly with no delays. Under some conditions, the processor is unable to fetch an instruction, usually because a prior multi-cycle instruction needs to use the bus. When this occurs, the processor asserts INULL to indicate that the current fetch cycle should be nullified.

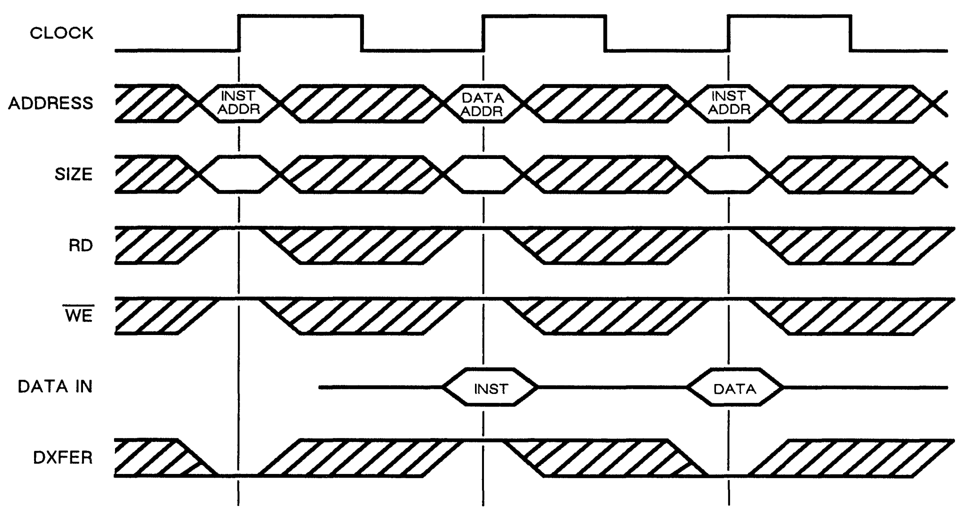


Figure 6–1. Instruction Fetch Followed by Load Cycle

Load Transactions

The second clock cycle in *Figure 6–1* shows the timing for a load single integer instruction. Load single is a two cycle operation: The first cycle is a regular instruction fetch while the second cycle is consumed in loading the required information from memory. Since the second cycle requires usage of the bus, instruction fetch has to be suspended for one clock. In other words, Load single causes a one-clock bus delay.

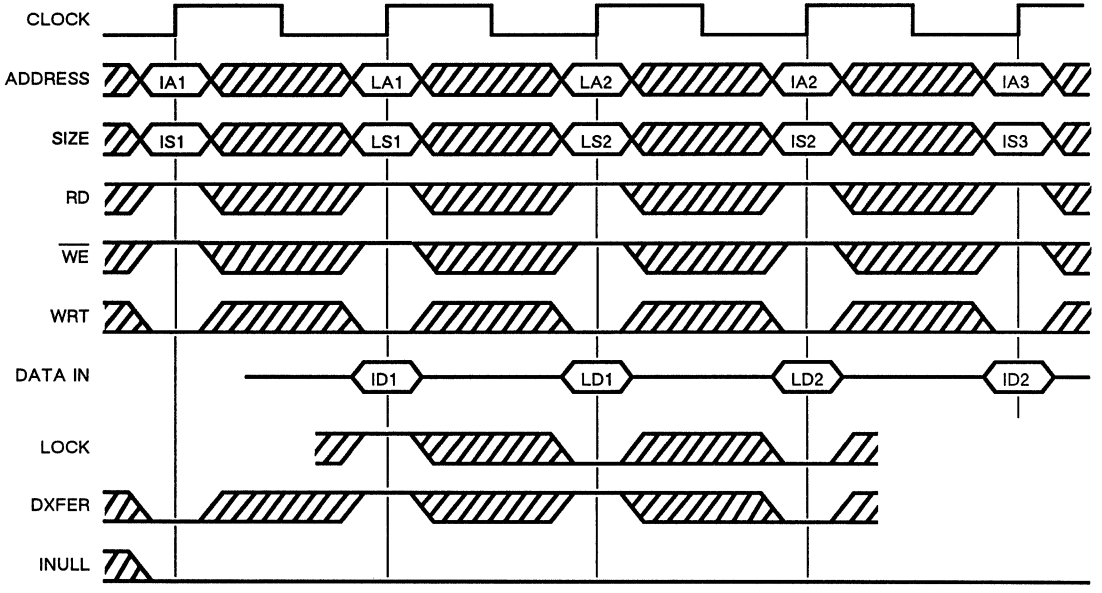


Figure 6–2. Load Double Word

Figure 6–2 shows the timing for a load double integer instruction. LA, LS, and LD labels indicate the address, size and data information respectively for the Load operation while IA, IS and ID indicate cyles associated with instruction fetches. The instruction consumes three clock cycles and causes a two-clock bus delay. The first cycle fetches the load double instruction, the second cycle loads in the first word, and the third cycle loads in the second word. The address of the second load is equal to the address of the first load + 4, and the size bits = 11 (indicating a double access). Load single floating-point instructions and load double floating-point instructions are similar to their integer counter-parts except that floating-point control signals are used.

Store Transactions

The second access in *Figure 6–3* shows a store single integer instruction (the first access is a load single integer). LA, LS, and LD labels indicate the address, size and data information respectively for the Load operation while SA, SS and SD indicate cyles associated with the Store operation. IA and IS indicate the beginning of the next instruction fetch. A store single operation requires three clock cycles and causes a 2-clock bus delay. During the first cycle, the store instruction is fetched. Since this is a normal instruction fetch cycle, it does not incur a bus delay.

During the second cycle, the address of the store is driven onto the bus where it remains well into the third cycle. Store data is generated at the middle of cycle 2 and removed at the middle of cycle 3. Actual memory update occurs in cycle 3. The store address is delivered early in cycle 2 so that it can be checked for possible cache miss or memory exception. Also note that INULL goes HIGH during the cycle in which the data to be stored is placed on the bus. This is to indicate that the address outputs of this cycle should not be used to generate a cache miss and resultant MHOLD. Tags should have been checked and any miss generated in the previous cycle. This arrangement allows external cache logic an extra cycle to determine whether or not data is present in the cache and to apply MHOLD if it is not.

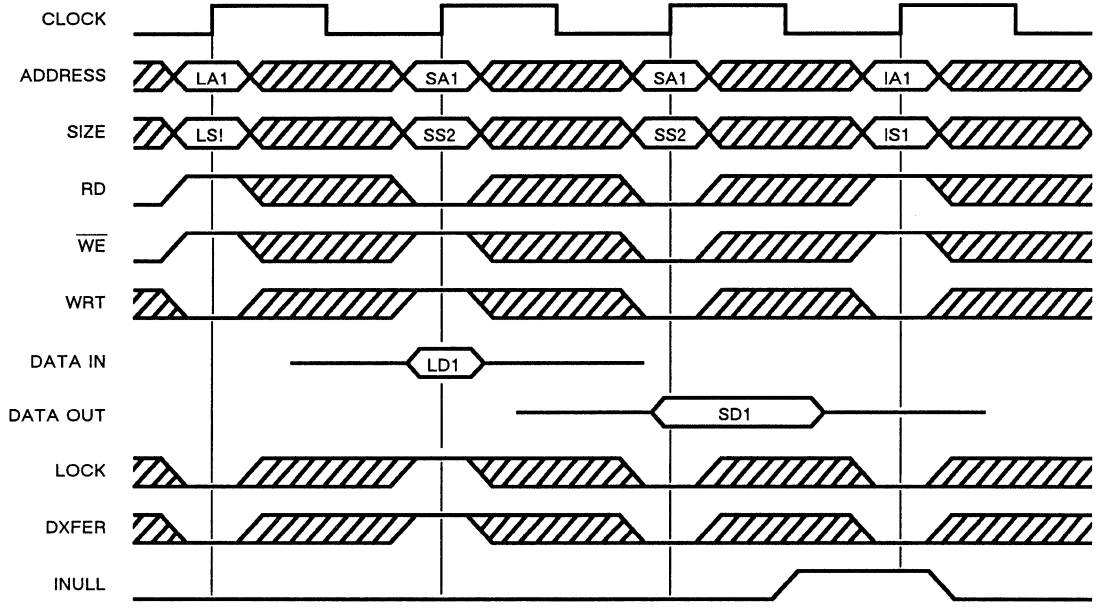


Figure 6–3. Load Followed by Store

Figure 6–4 shows the timing for a store double integer instruction. The timing is very similar to the store single integer timing except an extra cycle is needed to store the second word. In other words, store double integer is a 4 cycle operation which causes a 3-clock bus delay. Note that the address of the second store is equal to the first address + 4, and that the size bits are set to 1, 1 to indicate a double access. The timings for a single and double floating-point store instructions are similar to the integer stores, except that they generate FINS1 and FINS2 during the decode cycle of the floating-point instruction, as well as other floating-point signals. Note that INULL operates exactly as it does in the case of a store single word instruction (indicating that tags should not be checked during the first data store cycle). INULL does not however remain active for the second data store cycle. Since the data and instructions of SPARC processors are fully aligned, the second data cycles of Load and Store Double operations will not generate a miss if the first data cycle did not unless the cache line size is less than two words.

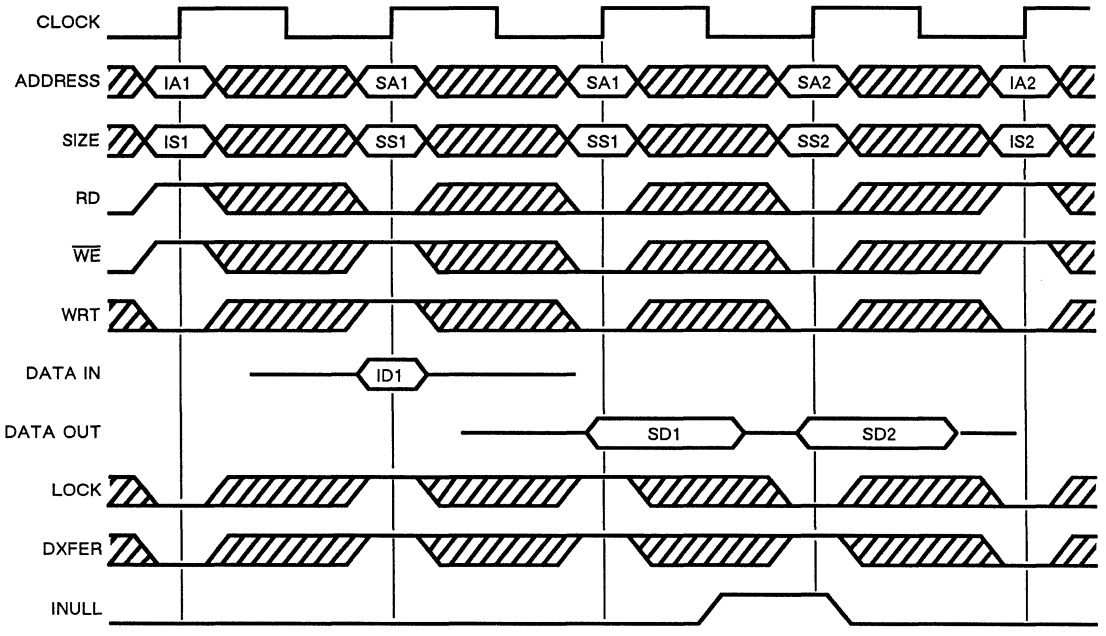


Figure 6–4. Store Double Word

Data Bus Contents

The following diagrams illustrate the behavior of the data bus lines D(0–31) during Load and Store operations. *Figure 6–5* shows the relationship between the data transferred during word, halfword, and byte operations and the pins of the data bus. The processor automatically aligns byte (and halfword) transfers as shown by the examples in *Figure 6–6* and *Figure 6–7.*

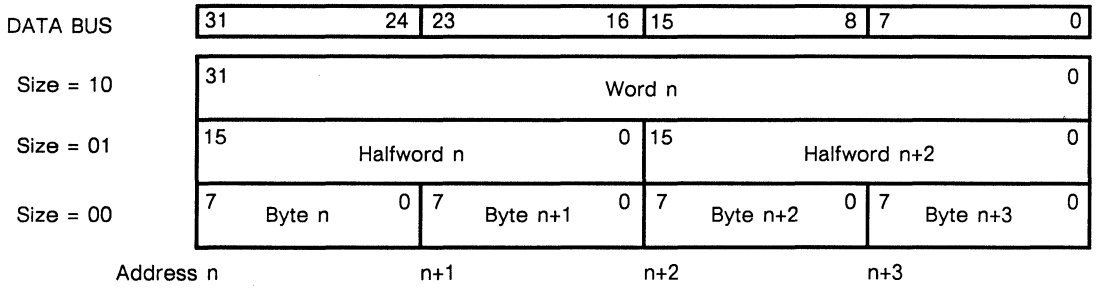


Figure 6–5. Data Bus Contents During Load/Store

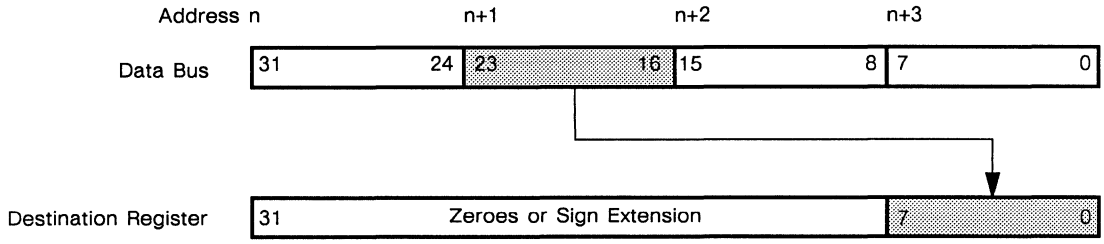


Figure 6–6. Byte Load Example (From Address n+1)

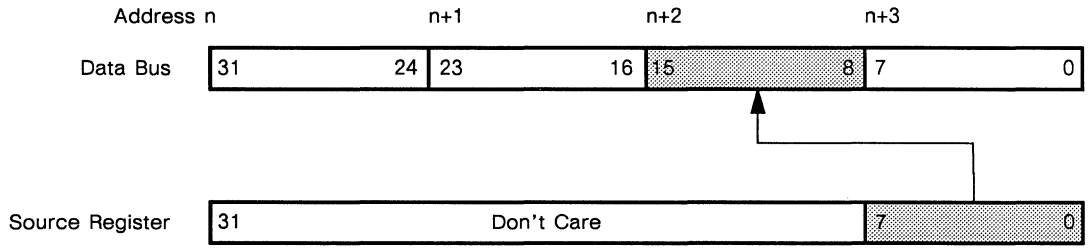


Figure 6–7. Byte Store Example (T0 Address n+2)

Atomic Transactions

Atomic transactions consist of two or more steps which are indivisible; once the sequence is started, it cannot be interrupted. To ensure that it has the bus for the second transaction, the IU asserts LOCK for as long as necessary. *Figure 6–8* shows the timing of an atomic load and store unsigned word instruction (SWAP). It takes 3 extra cycles, and is described in Appendix B.

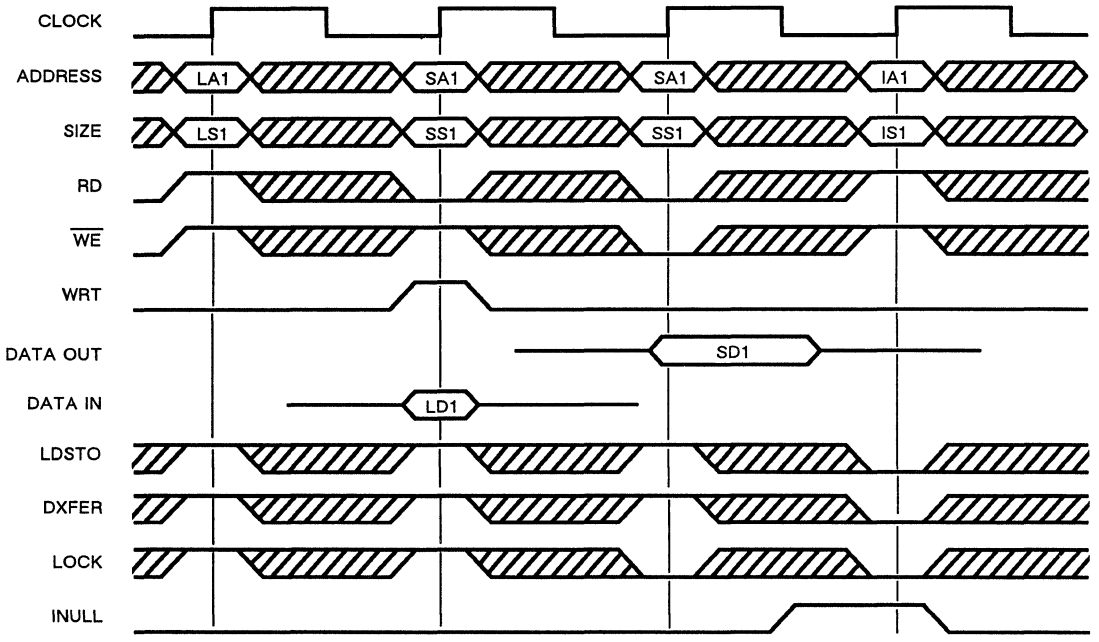


Figure 6–8. Atomic Load/Store Word (SWAP)

Floating-Point Operations

When the IU decodes an FPop, it signals the FPU using the FINS1 and FINS2 signals. Upon detecting active FINS1 or FINS2 signals, the floating-point controller starts the decode and execution of the fetched floating-point instruction. If the floating-point controller detects a condition which requires it to delay execution of the current instruction, it stops the IU by asserting $\overline{\text{FHOLD}}$. This can happen under the following conditions:

1. When a Store Floating-point State Register (STFSR) instruction starts execution and FPops are pending in the floating-point queue. In this case, the FPU controller detects the condition and asserts $\overline{\text{FHOLD}}$ because the STFSR instruction cannot proceed until all pending FPops have completed execution.
2. When there is either a resource or an operand dependency between the present FPop and one or more of the previously fetched instructions.
3. When a branch on floating-point condition instruction (FBfcc) starts execution while the floating-point conditions are not ready. This occurs when one of previously fetched instructions is a floating-point compare (FCMP) which the FPU has not yet completed.

Bus Arbitration

Because the Integer Unit behaves like a bus slave, bus arbitration must be performed externally using the $\overline{\text{BHOLD}}$ and LOCK signals as shown in *Figure 6–9*. The CY7C601 IU asserts LOCK

when it needs to retain the bus. External hardware requesting access to the bus should assert $\overline{\text{BHOLD}}$ when LOCK is inactive. When $\overline{\text{BHOLD}}$ is asserted, the processor's pipeline is held until it is de-asserted. The signals $\overline{\text{DOE}}$, $\overline{\text{AOE}}$, and $\overline{\text{COE}}$ can be used to turn off the output drivers of the data bus, the address bus, and the other control signals, thus allowing external sources to drive the bus.

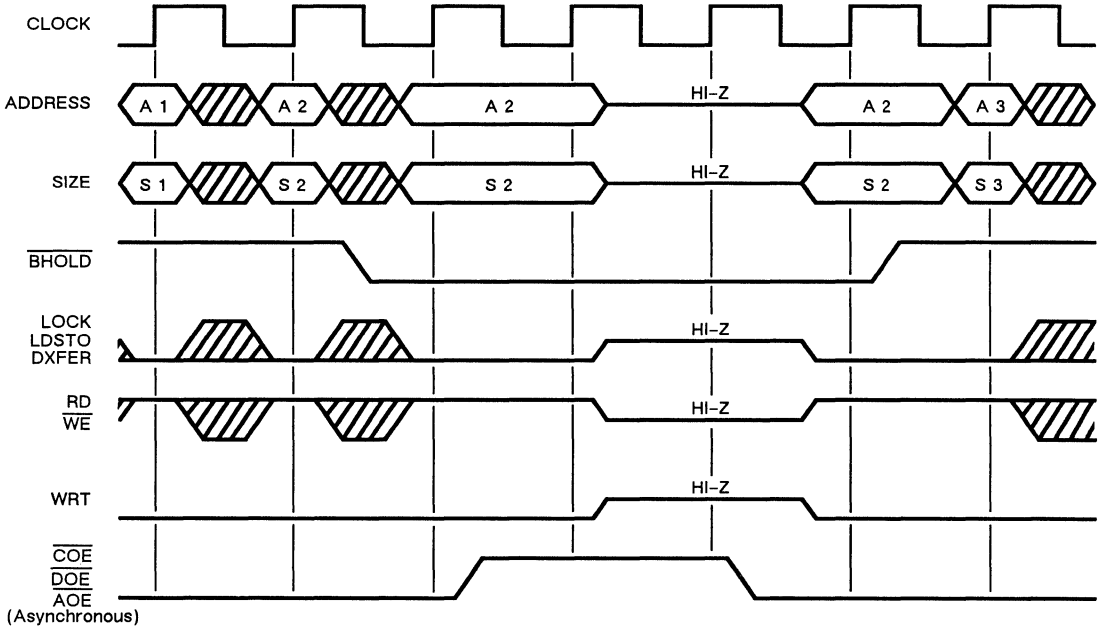


Figure 6–9. Bus Arbitration

Reset Operation

The processor requires two clock cycles to synchronize the reset input and six cycles to initialize the state of the machine for a total of eight clock cycles. Since Reset is synchronized internally, Clock must be applied to the processor for Reset to be recognized. Reset performs the following operations:

1. Initializes internal processor logic
2. Disables Traps by setting ET=0
3. Sets Supervisor Mode (S=1)
4. Sets trap type (tt) in Trap Base register only when reset occurs as the result of a return from trap instruction (RETT)
5. Sets Program Counter to location 0 (PC=0, nPC=4)

All other PSR fields and registers retain their previous value which on power up is undefined. There is no distinction between "warm" and "cold" reset in this processor. Reset (and Clock) should always be applied to the CY7C601 before power is applied to guarantee that chip reaches a valid state on power up.

CYPRESS SEMICONDUCTOR

CHAPTER 7

Memory Interface

The CY7C601 Processor has a 32-bit address bus, and can directly address 4 gigabytes of memory. The address bus, data bus and all memory interface signals (except INULL) are output "unlatched" in the previous cycle.

Memory Wait States

The IU provides several signals that can be used to insert wait states into the execution pipeline. These are $\overline{\text{MHOLDA}}$, $\overline{\text{MHOLDB}}$ and $\overline{\text{BHOLD}}$. Asserting any of these signals freezes the pipeline until the signal is deasserted. $\overline{\text{BHOLD}}$ should be used only for bus requests. While frozen, the IU does not fetch instructions or execute previously fetched instructions. The only state that changes is the contents of those latches used to capture external asynchronous events such as external interrupts, memory exceptions, and input control signals from the floating-point unit or coprocessor.

Cache Memory Systems

A cached memory system may use the lower address bits (for example A[15:2] for an 32-bit wide 64K byte cache) to address the cache RAMs, and the high address bits to compare the tags. For every cache access, the cache miss logic must send a hit or miss indication to the processor in the next cycle. If the cache hits, no wait state is inserted, and the memory access completes in one cycle.

If the cache misses during a read, the cache logic must stop the processor by asserting $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$ in the next cycle as shown in *Figure 7–1*. Asserting $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$ during a load operation causes the processor to halt with the address of the next instruction (the instruction following the instruction which caused the hold) on the address bus. In general, the memory system needs the address which caused the hold in order to retrieve correct data from memory. The MAO input provides this address by forcing the processor to output the previous rather than current address. The new data must be given to the processor while $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$ is asserted. The $\overline{\text{MDS}}$ signal is used to strobe the data into the processor since the internal processor clock is stopped while in hold.

Normally, the processor expects to receive a new instruction every cycle. If the memory is not fast enough to provide one instruction per cycle, then wait states must be inserted by asserting $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$ before the end of the cycle in which the processor needs the data.

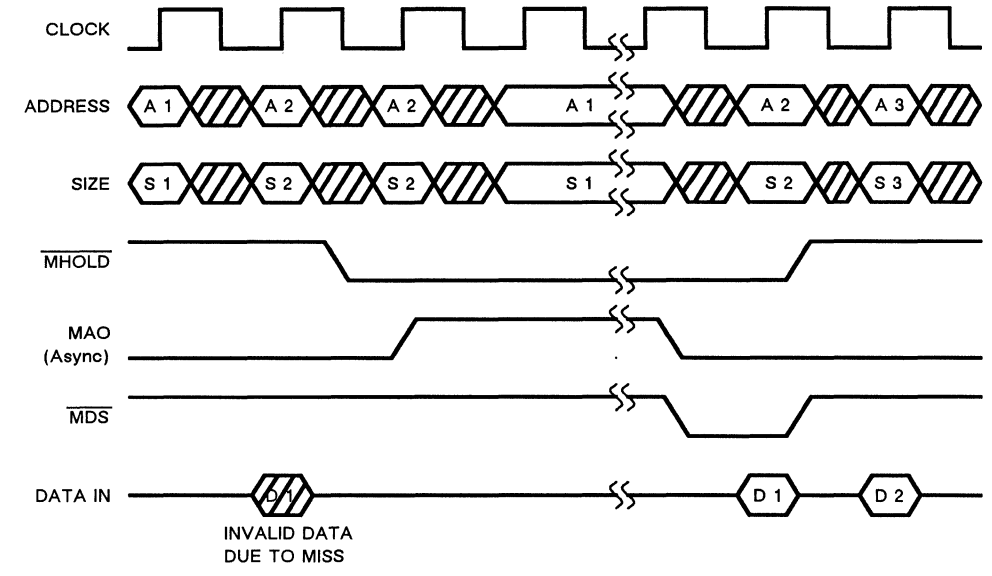


Figure 7–1. Load with Cache Miss

When a cache miss and resultant hold occurs during the first (tag check) cycle of a store operation the processor stops with the next address on the address bus just as it does during a load operation. However, since the processor outputs the same address twice during stores (once for tag check and once for the actual write operation), the next address is the same as the previous one as shown in *Figure 7–2*. As a result, the use of the MAO input is not required for misses during stores.

While the processor is stopped the data to be written to the memory system is available on the data bus as shown. The INULL signal goes active during the second cycle of the store to indicate to the memory system that a cache miss should not be generated for this cycle.

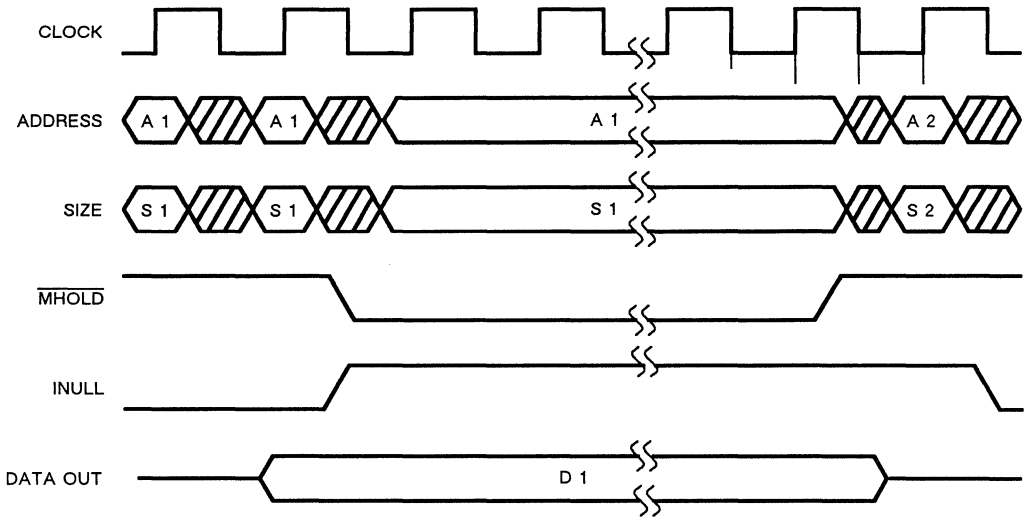


Figure 7–2. Store with Cache Miss

Memory Exceptions

The processor must receive all memory exceptions in the cycle following the event that caused the exception (in the same manner as a cache miss). When a memory exception occurs, the processor should be stopped by asserting $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$, and while in the hold state, $\overline{\text{MEXC}}$ should be asserted as shown in *Figure 7–3*.

The $\overline{\text{MDS}}$ strobe is needed to strobe the $\overline{\text{MEXC}}$ signal into the processor since the internal processor clock is stopped while in hold. An $\overline{\text{MDS}}$ strobe without $\overline{\text{MEXC}}$ (as is the case with a cache miss) causes the processor to accept the current contents of the data bus replacing the previously loaded invalid data, and to continue as if nothing had happened. An $\overline{\text{MDS}}$ strobe with $\overline{\text{MEXC}}$ applied causes the processor to ignore the current data bus contents and take a memory exception trap. $\overline{\text{MEXC}}$ must be deasserted in the same cycle in which $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$ is released.

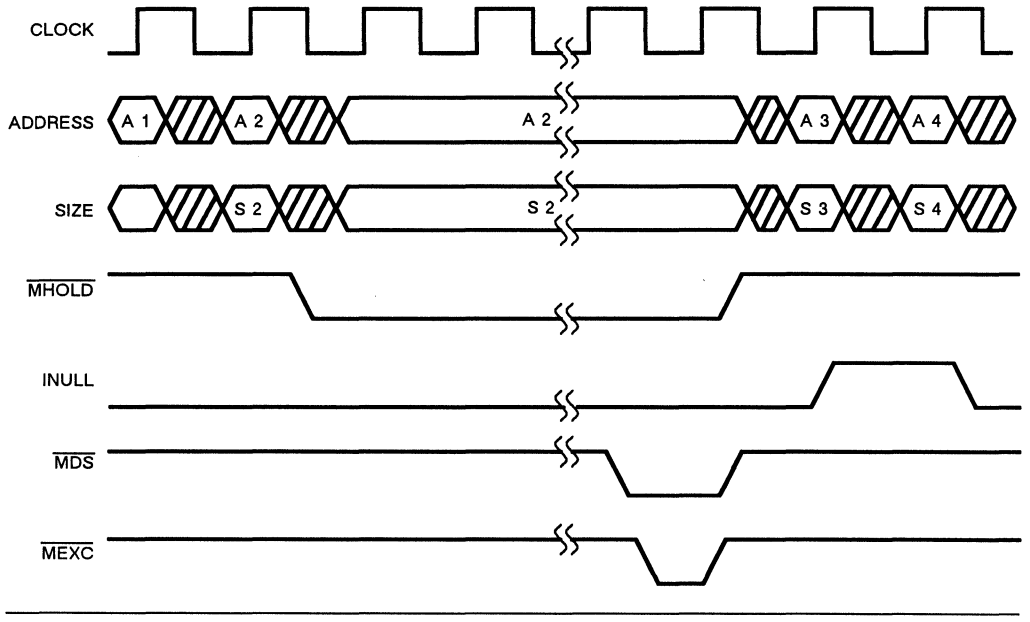


Figure 7–3. Load with Memory Exception

CHAPTER 8

CY7C601 Coprocessor Interface

The Integer Unit is the basic processing engine which executes all of the Integer instruction. The CY7C601 IU is implemented such that up to two Coprocessor Instruction Set extensions are possible. The most obvious use of these capabilities is for floating-point operations. These extensions, however, are generic in nature. Within the bounds of the SPARC Instruction Set architecture format, either coprocessor extension may be used for anything.

Concept of a Coprocessor

The concept of a coprocessor is that it is an extension of the basic IU instruction set. It operates from a common instruction stream with both the IU and any other coprocessors It interferes minimally with IU operations. In most cases, the coprocessor function should be able to be incorporated into the integer unit. Although this may be desirable in some cases, providing the capability of externally attaching a coprocessor or extending the architecture for a specific case provides both flexibility for the designer and at the same time does not burden all users with functions they may not need in this application.

Concurrency

Coprocessors operate concurrently with both the IU and any other coprocessors in the immediate system. This is necessary to maximize system performance. At the same time, it creates a problem in keeping track of the multiple operations in progress. For this reason, some method of keeping track of these simutaneous operations or synchronizing them is necessary.

Synchronization

Synchronization is achieved in hardware by sharing both the address and data buses, overtly requiring that the coprocessor test for data dependencies and transferring data to from memory with Load/Store operations where addresses are only generated by the IU.

Coprocessor Interface

The CY7C601 coprocessor interface consists of the normal address, data clock, reset and control signals and two special pair sets of coprocessor interface signals. These special interface signals provide synchronization and minimal status information transfer between coprocessor and integer unit. The configuration in *Figure 8–1* shows the basic systems connection between the CY7C601 IU and two possible coprocessors. *Table 8–1* shows the coprocessor interface signals, quantity and their description.

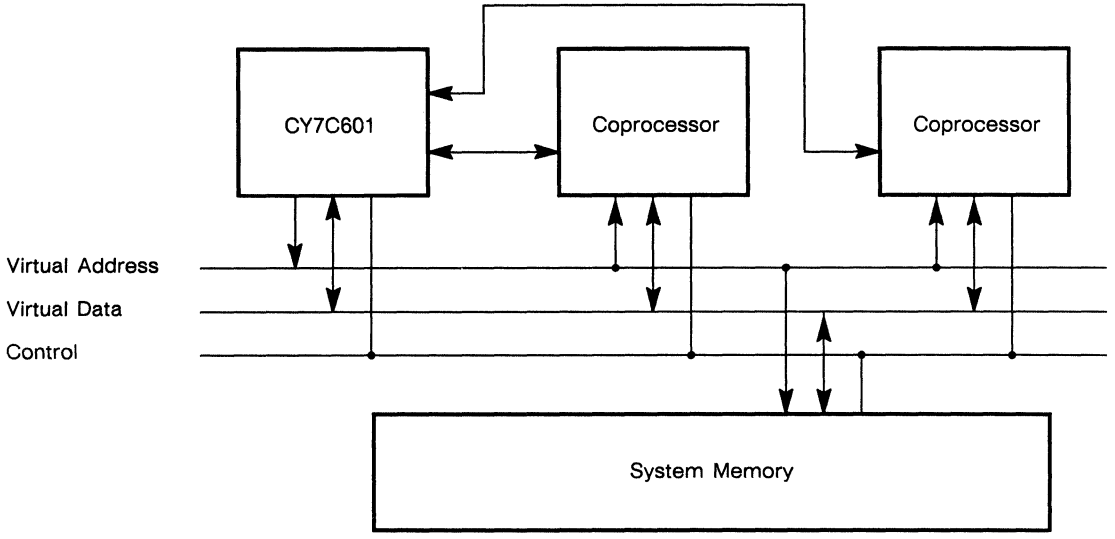


Figure 8–1. Coprocessor System Connection

Generic Signal	Generic Description	Quantity
CCV	Condition Code Valid	2
CC0	Condition Code 0	2
CC1	Condition Code 1	2
HOLD	Coprocessor Hold	2
EXC	Exception	2
P	Coprocessor Present	2
XACK	Exception Acknowledge	2
INS1	Instruction Stage 1	2
INS2	Instruction Stage 2	2
INST	Instruction Fetch Cycle	1
FLUSH	Flush Processor Instruction (s)	1

Table 8–1. Generic Coprocessor Interface Signals

Nine of the eleven coprocessor interface signals are duplicated so that a unique signal exists for each of the two coprocessors. The last two, **INST** and **FLUSH** are common to both coprocessors.

CCV Condition code valid is asserted to indicate that condition codes **CC0** and **CC1** are valid. The coprocessor deasserts **CCV** if pending compare instructions exist in the queue. **CCV** is then reasserted when the compare instruction is completed and condition the condition codes are again valid.

CC0, CC1 These are the current condition codes resulting from a coprocessor operation. They are only valid when **CCV** is **HIGH**. During execution of a **BCC** instruction (branch on condition code) the IU uses **CC0 & CC1** to determine whether the branch instruction should be taken. **CC0 & CC1** are latched by the IU before they are used.

HOLD Hold is asserted by the coprocessor if execution cannot be continued. The coprocessor checks all dependencies in the decode stage of the pipeline. **HOLD** is asserted in the next cycle if the instruction being checked has an unresolved dependency. **HOLD** is deasserted when the dependency is resolved and operation continues. This signal is latched by the IU before it is used. **HOLD** freezes the IU and other coprocessor pipeline in the cycle it is asserted and latched.

EXC Exception is asserted by the coprocessor to signal the IU that a coprocessor processing exception has occurred. It causes the IU to Trap and process the exception. Exception must remain asserted until the IU takes the Trap and issues an **XACK** signal in acknowledgment.

P Present indicates the presence of the coprocessor. The Present signal is normally pulled up to Vcc through a resistor. It is tied to Vss when the coprocessor is present. The IU generates a "Disable Trap" whenever a coprocessor instruction is executed if the **P** signal is deasserted.

XACK Exception acknowledge is asserted by the integer unit to acknowledge the coprocessor Exception has been recognized by taking the "Disable Trap". The coprocessor must immediately deassert the **EXC** signal that caused the "Disable Trap" so that the IU does not take a second Trap in response to the same **EXC**.

INS1 Instruction Stage 1 is asserted by the IU during the decode operation if the instruction being decoded is a coprocessor instruction and the instruction is located in the coprocessors instruction/address decode 1 register pair. This causes the coprocessor to move the instruction and its address into the execution phase and passes it to the queue for actual execution.

INS2 Instruction Stage 2 is asserted by the IU during the decode operation if the instruction being decoded is a coprocessor instruction and the instruction is located in the coprocessors instruction/address decode 2 register pair. This causes the coprocessor to move the instruction and its address into the execution phase and passes it to the queue for actual execution.

INST Instruction is asserted by the IU whenever an instruction is being passed from the memory to the IU for execution. The coprocessors use this information to latch both the instruction being fetched and the address of the instruction. Each coprocessor has two instruction and address decode registers see *Figure 8–6*. When **INST** is asserted, the first instruction/address pair is loaded into instruction decode register 1 and address decode register 1. If **INST** occurs again before the instruction initially fetched is passed to the coprocessor queue, the old instruction is passed to instruction decode register 2 and address decode register 2 while the new instruction is loaded into instruction decode register 1 and address decode register 1.

FLUSH Flush is asserted by the integer unit and causes the coprocessors to flush their internal instructions that have not been passed to the queue. This happens whenever the IU takes a branch or trap altering the instruction flow with instructions pending in the

decode registers of a coprocessor. Instructions that have been passed to the coprocessor queue will continue to execute whenever the Flush is caused by a Trap, Branch or exception taken by another device.

Table 8–1 shows only the generic names and descriptions of the coprocessor interface signals. Because the specific implementation of the CY7C601 integer unit assumes that coprocessor 1 will normally be a floating-point coprocessor, while coprocessor 2 may be a generic coprocessor implemented by the designer or some future device offered by Cypress, the signals are specifically named to reflect this use. See *Table 8–2* for the actual signal names for the coprocessor interface on the CY7C601.

CY7C601 Signal	CY7C601 Description	Quantity
FCCV	FP Condition Code Valid	1
FCC0	FP Condition Code 0	1
FCC1	FP Condition Code 1	1
FHOLD	FP Coprocessor Hold	1
FEXC	FP Exception	1
FP	FP Coprocessor Present	1
FXACK	FP Exception Acknowledge	1
FINS1	FP Instruction Stage 1	1
FINS2	FP Instruction Stage 2	1
CCCV	CP Condition Code Valid	1
CCC0	CP Condition Code 0	1
CCC1	CP Condition Code 1	1
CHOLD	CP Coprocessor Hold	1
CEXC	CP Exception	1
CP	CP Coprocessor Present	1
CXACK	CP Exception Acknowledge	1
CINS1	CP Instruction Stage 1	1
CINS2	CP Instruction Stage 2	1
INST	Instruction Fetch Cycle	1
FLUSH	Flush Processor Instruction (s)	1

Table 8–2. CY7C601 Coprocessor Interface Signals

An example of the interconnections between the Integer Unit and a Floating–Point Unit consisting of a CY7C608 Floating–Point Controller and a Texas Instruments' SN74ACT8847 Floating–Point Processor is shown in *Figure 8–2*. The three chips interconnect without additional intervening logic. The signal lines between the FPC and FPP are explained in detail in the respective data sheets for these parts. There are additional connections to the Floating–Point Processor not shown in this diagram which are required for a fully–functiomal system. See the SN74ACT8847 data sheet for details.

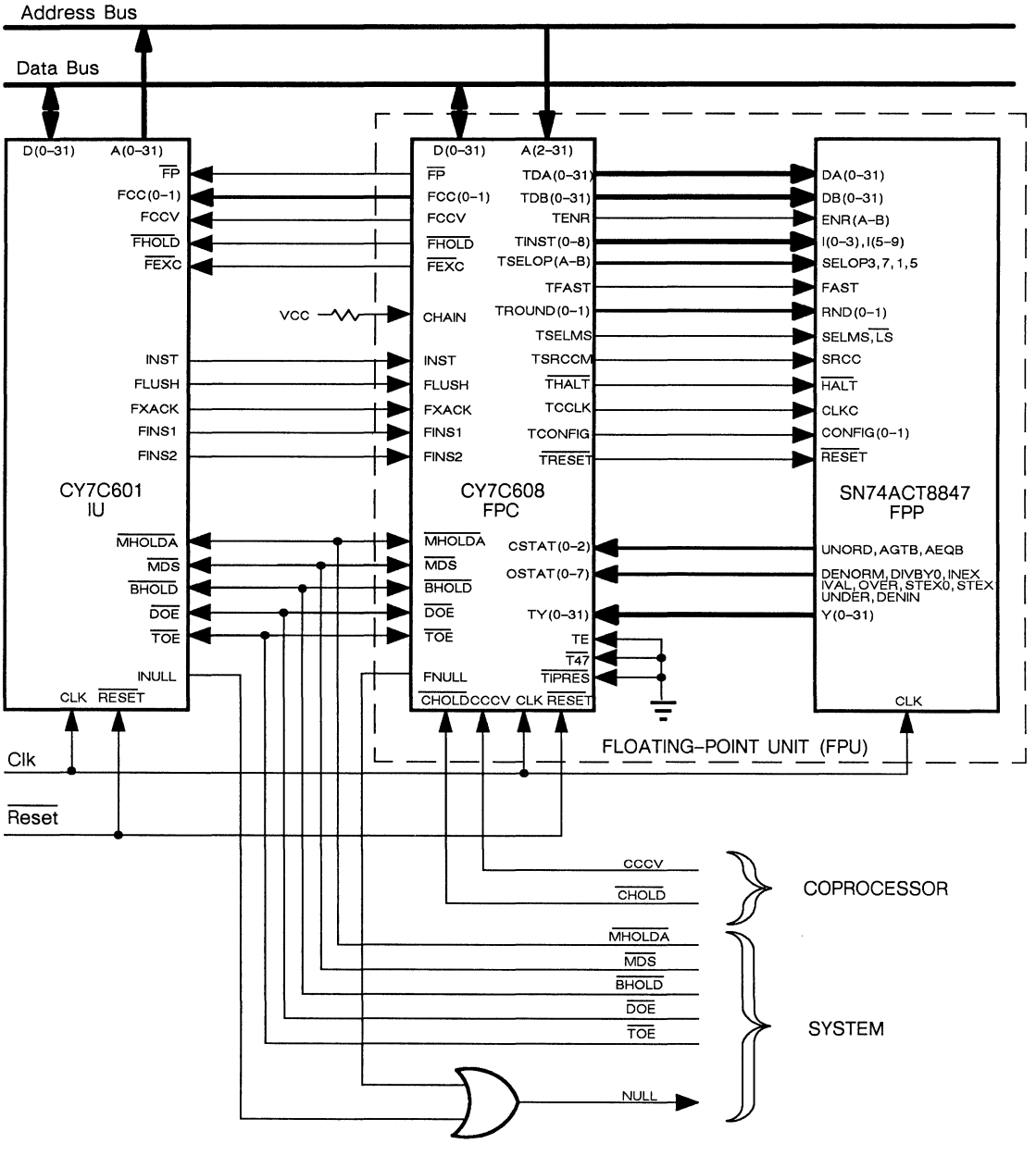


Figure 8–2. Floating–Point Controller System Connections

Coprocessor Operations

With respect to the Virtual bus, there are three basic operations that occur, Instruction Fetches, Data Load Operations and Data Store Operations. As indicated in *Figure 8–1*, all coprocessors are connected in parallel with the IU with additional control and synchronization accomplished through the unique coprocessor interface control signals. In addition, all processors are clocked in parallel with the same clock signal and share the reset signal. Since all instructions occur only on word boundaries, and all coprocessor load and store operation are also restricted to word aligned, the connection to the address bus by the coprocessor is only 30 bits. The two least significant bits are ignored.

The IU always generates the address for all bus transfers associated with IU or either coprocessor. A fetch operation consists of the IU operating in a normal manner and the coprocessors recognizing, from **INST** being asserted on the rising edge of the clock, that both the address present on the bus and the subsequent information fetched on the next clock rising edge are the address of an instruction and the instruction respectively. Recognizing this, the coprocessors attached to the bus load both the instruction and its address into on of their internal decode stages. All instructions are loaded into all processors in parallel on every occurrence of an asserted **INST**.

Load operations occur in two types, single word Loads (32 bits) and double word Loads (64 bits). As stated earlier, the IU generates the address or addresses that the data will be loaded from. A coprocessor Load loads data from a single 32 bit word aligned field in memory. In the case of a double word coprocessor Load, the information comes from two word aligned sequential 32 bit fields in memory. As shown in *Figure 8–4*, the IU generates the address for the load operation, and the data is loaded into the appropriate coprocessor only. This is accomplished by having the coprocessor execute the same load operation in sync with the IU. The IU accomplishes the generation of the address, and the coprocessor accepts the data from the bus and loads it into the appropriate register. In the case of a double load operation, two sequential load operations occur in sequential bus cycles.

Load operations are differentiated between the three potential processors on the virtual bus by instruction, there is a separate opcode for loading each of the two coprocessors and the IU.

In a similar manner, store operations are accomplished. The IU supplies the address, and at the same time the appropriate coprocessor supplies the data for the store operation. In the case of a double store, two sequential cycles are used to transfer 64 bits of data. Store operations are also differentiated among the processors on the virtual bus by instruction, there is a separate opcode for storing data from each of the two coprocessors and the IU.

Figure 8–6 shows the working and addressable registers of the coprocessor. The 32 bit by 32 word register file is the working register file for any coprocessor. All operands and results of coprocessor operations originate and return there. Coprocessor Load and Store operations are used to transfer data between this register file and memory. The Status register contains the current status of the coprocessor and may be interrogated with a store operation as well as tested where appropriate with the Bcc (branch on condition codes) by the IU. **CC0** and **CC1** are brought out as signals on the coprocessor as well as being present in the status register. This allows the IU to directly test these condition codes during a Bcc operation. As appropriate, the status register may be set with a Load operation.

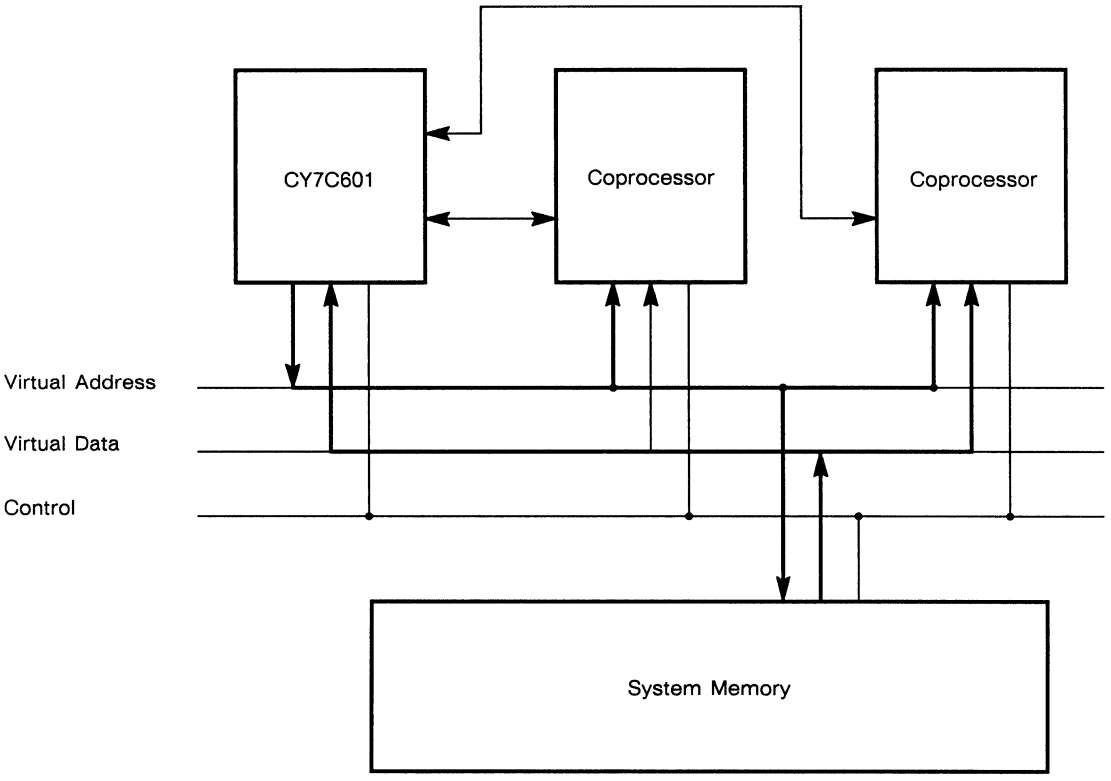


Figure 8–3. Instruction Fetch

Coprocessor Compare Operations

Coprocessor compares (CMP) and branch on coprocessor condition (Bcc) instructions interlock on the coprocessor condition codes. The SPARC architecture specifies that at least one non-Coprocessor instruction must occur between an CMP and an Bcc (branch on condition code) that expects valid condition codes. Violation of this may result in using old condition codes for branch condition evaluation and the result is undefined. When a Coprocessor compare is executed, the Coprocessor deasserts the CCV signal and keeps it deasserted until the compare instruction is completed and the Coprocessor condition codes are ready. Both IU and coprocessor wait during this interval while the coprocessor unit is performing the compare operation.

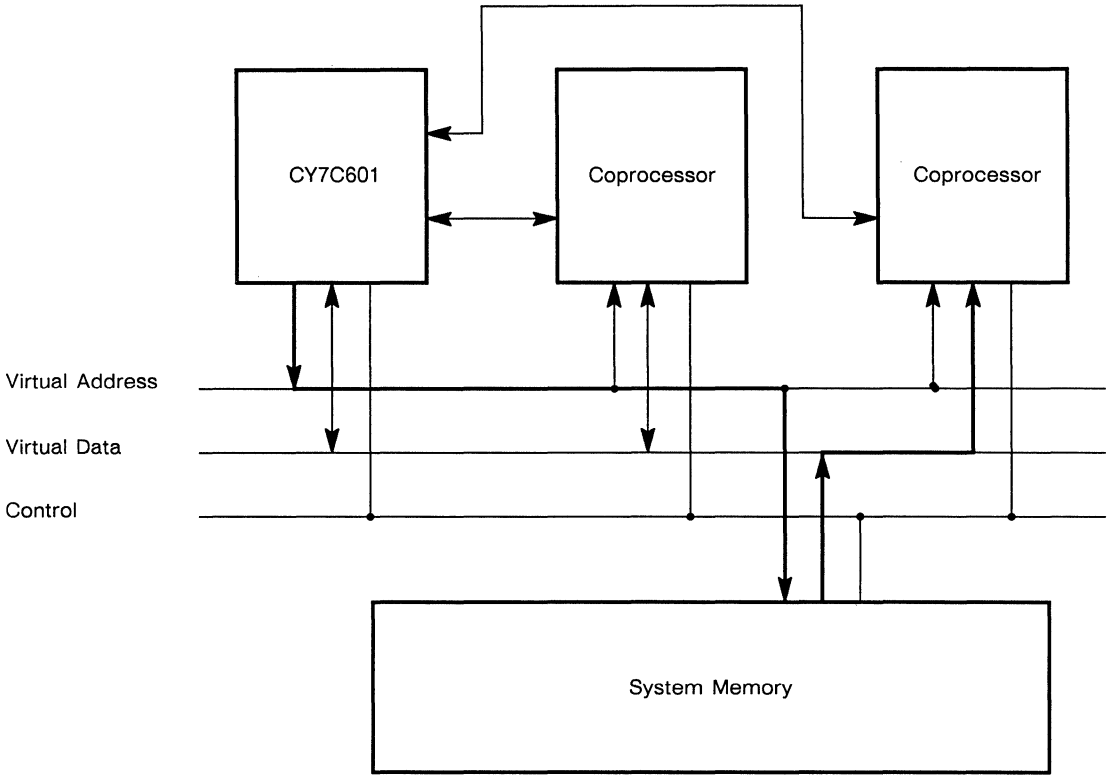


Figure 8–4. Load Operation

Unimplemented Coprocessor Instructions

This trap is generated when a coprocessor operation instruction defined in the architecture but not implemented on the current hardware is encountered. The trapped instruction can be emulated in software.

Unfinished Coprocessor instruction

This trap is signaled when an coprocessor operation cannot complete execution because the data has exceeded the capabilities of the coprocessor and/or has generated an inappropriate result.

Once a coprocessor operation has been started, the instruction and its address are available in the queue. Because trap handlers may look at both the PC or nPC, and the queue, an instruction's address is not entered into the coprocessor queue until it has "fallen off" the integer unit's PC queue. When an trap or exception is taken, the IU asserts the FLUSH signal. This causes the coprocessor to abort any instructions that have not yet entered the queue.

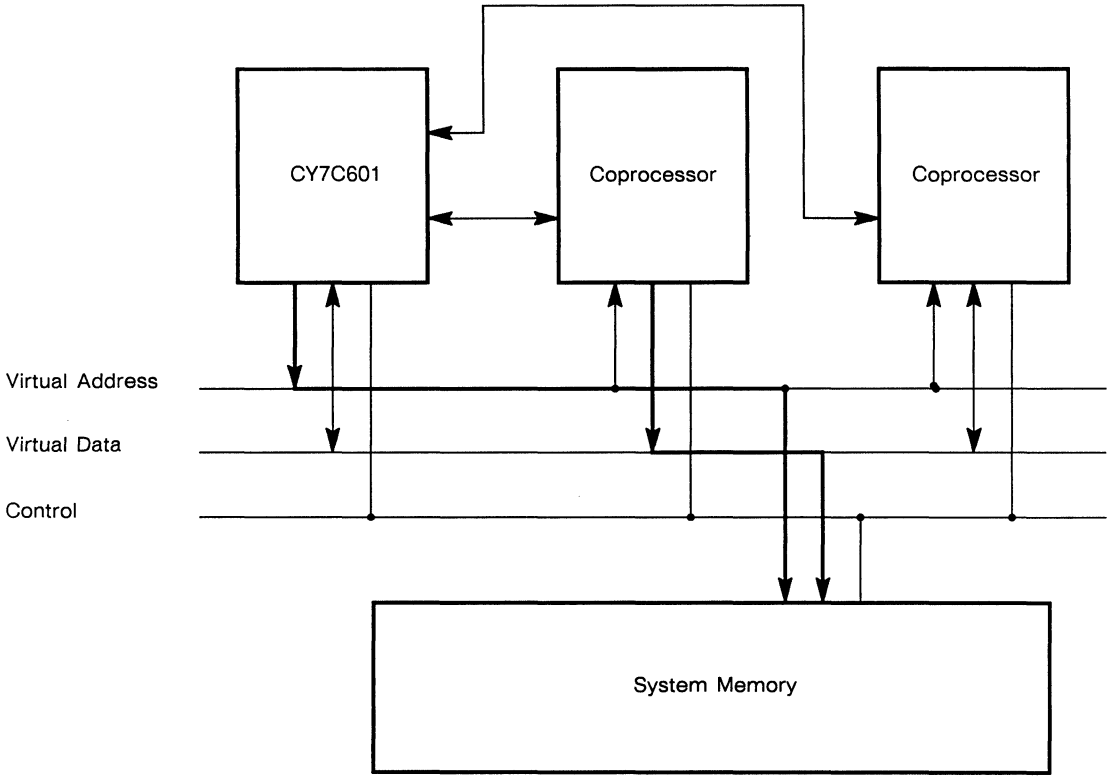


Figure 8–5. Store Operation

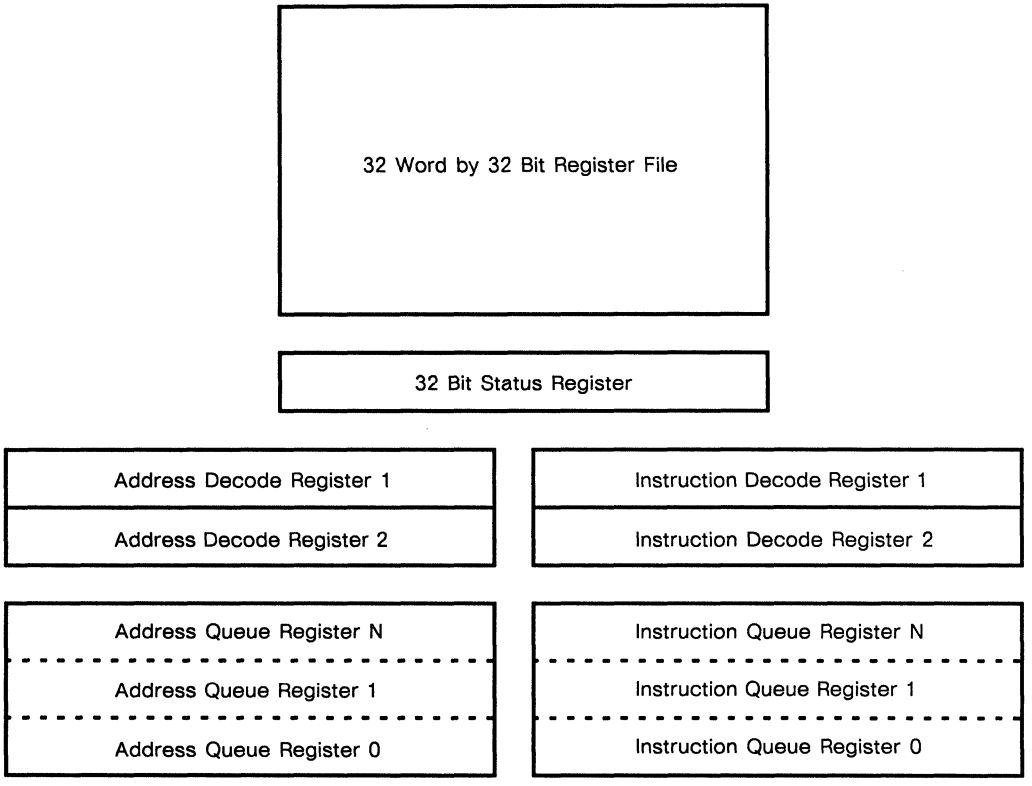


Figure 8–6. Coprocessor Register Model

CHAPTER 9
CY7C601 Design Examples

Clocking The CY7C601 IU

Clock Parameters

The CY7C601 Integer Unit can be driven at a maximum clock rate of 33 MHz. At 33 MHz, the high portion of the clock signal, measuring from 2.1 V of the rising edge to 2.1 V of the falling edge, must be a minimum of 13 ns. In addition, the low portion of the clock, measuring from 0.8 V of the falling edge to 0.8 V of the rising edge, must also be a minimum of 13 ns wide. At 25 MHz, the clock high and low requirements are relaxed to 18 ns and 18 nS respectively. Rise and fall time of the clock is specified at 1 volt per ns.

Clock Generation

In many designs, system clocking is supplied by hybrid crystal oscillators. Although most crystal oscillators are capable of generating clean clock signals, clock symmetry can vary from 40% to 60% depending on the vendor. In a 33 MHz CY7C601-based system, the user must select a crystal oscillator which satisfies the 13 nS clock high and low requirements of the IU. If the selected crystal oscillator cannot meet the clocking requirements, a clock signal with 13 ns min minimum. high time can be generated by dividing the output of the oscillator with a D-type flip flop (74AS74). The oscillator frequency in this case has to be two times the system clocking frequency (i.e., 66 MHz). The schematic of the clock generation circuit is shown in *Figure 9–1.* The jumper in the schematic allows the user to choose between direct oscillator output and the half frequency flip flop output.

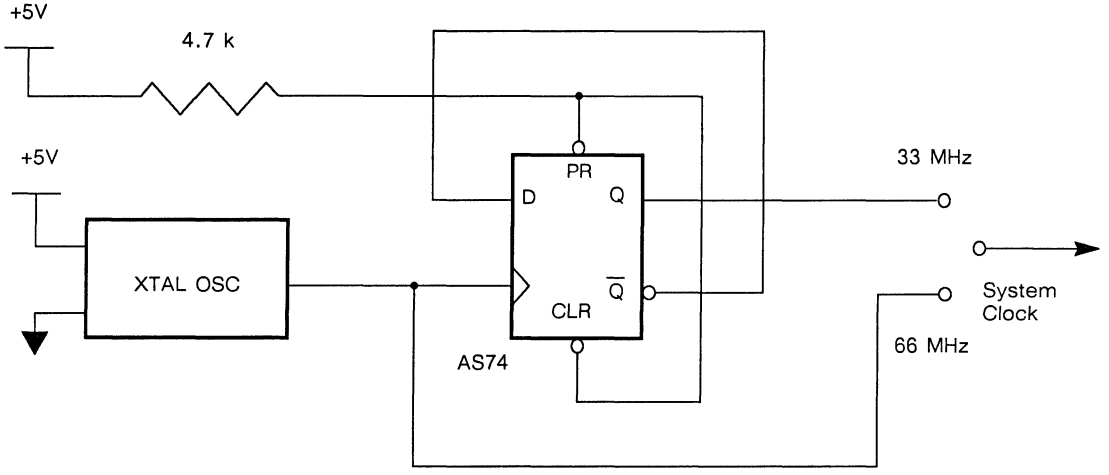


Figure 9–1. Sample Clock Generation Circuit

Clock Distribution

In a CY7C601-based system, the clock signal is used by various logic blocks for synchronization and information latching purposes.

These logic blocks include:

Address latches	Data latches	Wait state logic
Cache RAMs (CY7C153)	Peripheral access control logic	

The heavy demand for the clock signal makes clock buffering an necessity. Since multiple versions of the clock are required, the skew between clocks must be kept to a minimum. This is especially important in a 33 MHz system. The best approach to reduce clock skew between buffers is to select an integrated circuit with multiple buffers on-chip. In addition, the driving capability of the buffer should be weighed against its loading. One type of buffer featuring small inter-buffer skew and good driving power is the 74AS18xx series from Texas Instruments (e.g. 74AS1804) and they have been used successfully in CY7C601-based systems.

Once a buffered set of clock signals is available, they must be carefully distributed to maintain even loadings on all clock lines. Major load components are trace capacitance, socket capacitance, and input capacitance. Failure to maintain even loading will result in increased skew between clock signals. Another area of concern in clock distribution is signal reflection. A clean clock signal can be achieved only if each clock line is properly terminated. Common termination methods include parallel termination, series termination, AC termination, and diode termination.

Parallel Termination

Parallel termination involves placing a pull-up and a pull-down resistor at the end of the trace. If the Thevenin equivalent of the two resistors is equal to the characteristic impedance of the trace, no signal reflection will occur. With multi-layer printed circuit boards, a frequently used set of values is 220 ohms for the pull-up resistor and 330 ohms for the pull-down resistor. This gives an equivalent resistance of 132 ohms.

The advantages of parallel termination include no change in rise and fall times of the signal and the ability to support distributed loadings along the trace. The main disadvantage of this type of termination is the constant dissipation of power in the two resistors. *Figure 9–2* shows a trace with parallel termination. Note that the termination is at the load.

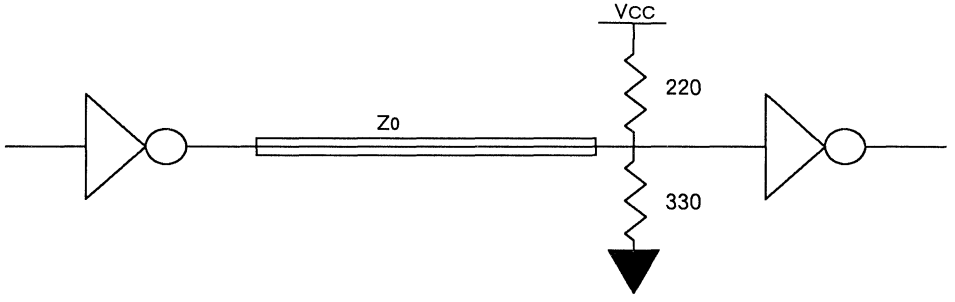


Figure 9–2. Parallel Termination

Series Termination

Series termination is accomplished by placing a resistor in series with the trace at the driving device. Series termination is also known as series damping because it "damps" out ringing caused by signal reflection. If the value of the series resistor plus the impedance of the driver equals the characteristic impedance of the trace, the reflection coefficient at the driver end will be zero and any reflected signal from the load end of the trace is absorbed. Typical resistor values are in the range of 16 to 68 ohms.

Series termination causes no DC power dissipation but will increase the propagation delay of the signal traveling down a trace. In addition, no distributed loads can be attached along the line and all loadings must be attached to the end of the trace. *Figure 9–3* shows a trace with series termination. Note that the series terminating resistor should be placed as close as possible to the driver.

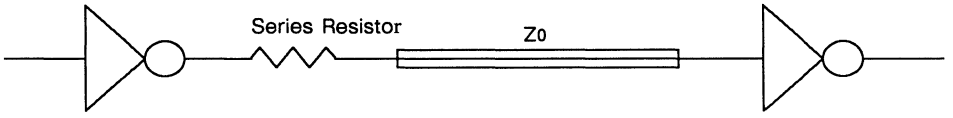


Figure 9–3. Series Termination

AC Termination

An AC termination is constructed by grounding the signal trace at the load end via a capacitor and a resistor connected in series. If the impedance of the capacitor is sufficiently small (less than 5 ohms) and the resistor's value is approximately equal to the characteristic impedance of the trace, the reflection coefficient will be close to zero and no signal will be reflected back to the driver. No DC power will be dissipated in the resistor because the capacitor blocks the DC path to ground.

AC termination is useful in applications with very high frequency signals. In order to reduce load on the driver, the value of the capacitor must be small. On the other hand, the impedance of the capacitor must be sufficiently small that the combined impedance of the capacitor and the resistor will approximate the characteristic impedance of the trace. Since Xc = 1/(2*pi*f*C), if C is small (e.g. 10 pF) then Xc will be small only if the frequency (f) is very high. *Figure 9–4* shows a trace with AC termination. Note that the termination is at the load.

Diode Termination

Diode termination is sometimes known as "active termination." A typical diode termination is shown in *Figure 9–5*. The signal seen by the load will be limited between (Vcc + diode drop) and one diode drop below ground. This method of termination is useful in applications where the trace or line impedance is not well defined.

Clock Stretching

If additional time is required for the CY7C601 IU to access a slow device, external logic must hold the IU by asserting the Memory Hold signal. Memory Hold is sampled by the IU at the

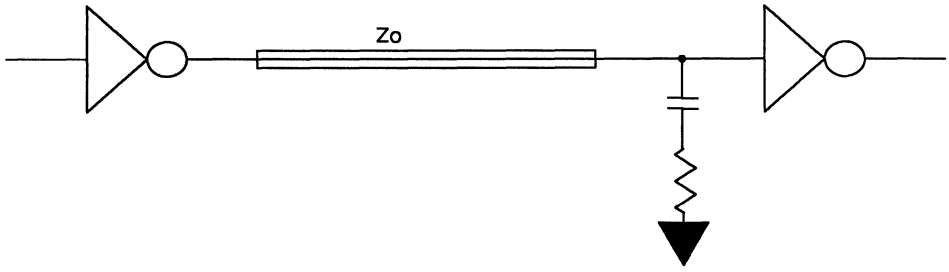


Figure 9–4. AC Termination

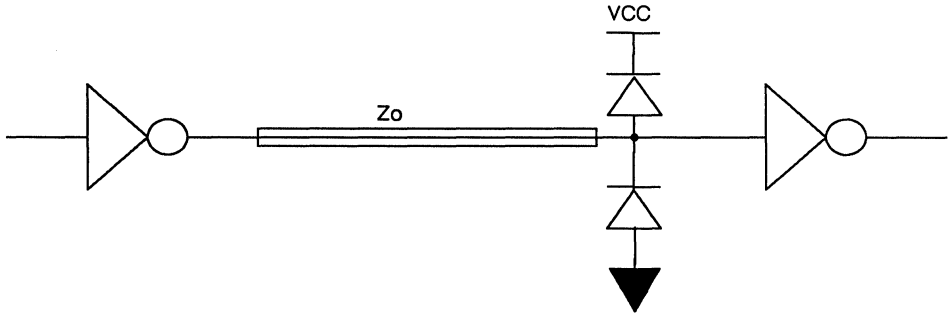


Figure 9–5. Diode Termination

falling edge of the clock in the following cycle. Because the processor typically generates a new address every clock, the address of the slow device would have been replaced by the address of the next access by the time MHOLD is recognized. In order to recover the lost address, the user can either:

Build a 32-bit address latch to store the previous access location.

Activate the MAO signal which causes the IU to return the previous access status (including the address) on the bus.

Once the address has been recovered, access of the slow device can proceed. When the required data is available, it must be strobed into the IU by pulsing the MDS signal for one clock.

The procedures described in the preceeding paragraphs can be avoided if the access of the slow device is completed within one clock cycle. This goal is attainable by using a technique called "clock stretching" which extends the low portion of the IU clock according to the time needed to complete an access. *Figure 9–6* illustrates a simple clock stretcher which inserts 2 wait states into an access.

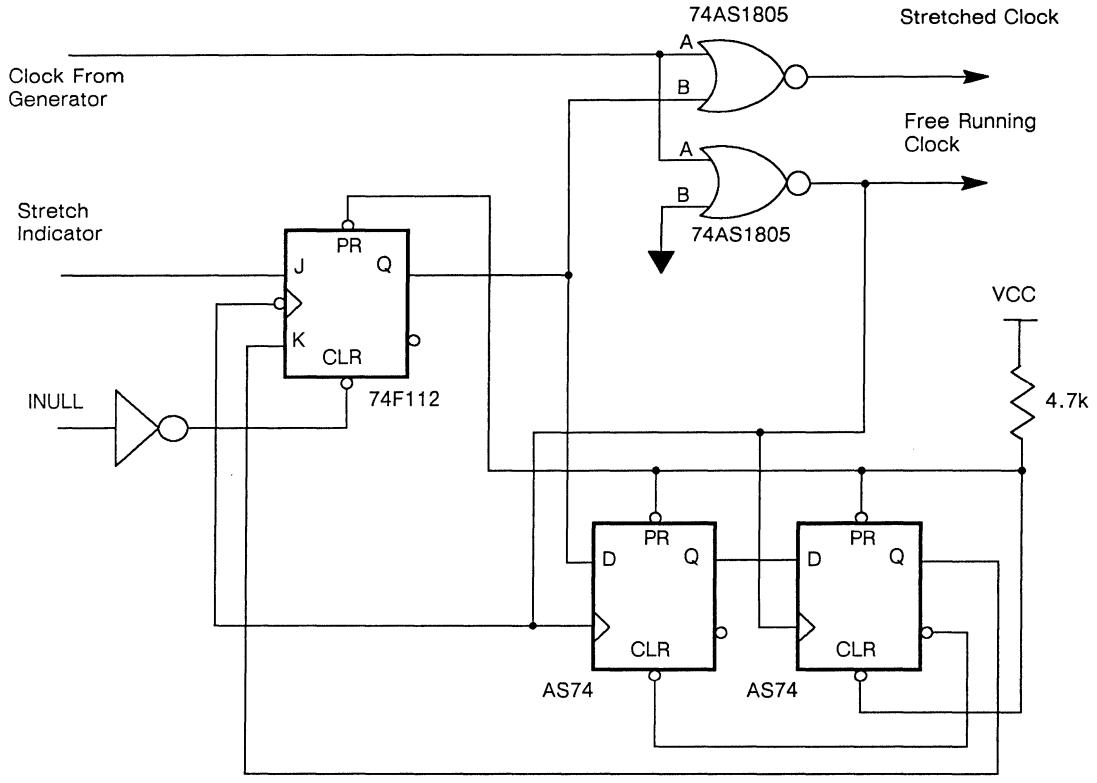


Figure 9–6. Clock Stretcher

The output of the clock generator is delivered to the A inputs of two 74AS1805 NOR buffers. With its B input tied to ground, the first NOR buffer functions as an inverter. Its output is a free running clock. The B input of the second buffer is connected to the output of a 74F112 flip flop clocked by the output of the first buffer. The output of this buffer is the "stretched clock."

At the beginning of an access, an address from the IU is decoded to determine whether extra wait states are needed. If additional cycles are required, the decoder will generate an active HIGH stretch indicator which is connected to the J input of the 74F112. The K input of the flip flop is LOW. At the falling edge of the free running clock, the output of the flip flop goes HIGH and force the output of the second NOR buffer to stay LOW.

With the clock successfully held, a timer is activated to count down the two wait states. In the following example, two 74AS74 D-type flip flops clocked by the free running clock are used. The D input of flop flop 1 (FF1) is connected to the output of the J-K flip flop. At the next clock rise, output of FF1 goes HIGH and one clock later, the Q output of FF2 goes HIGH. Since Q from FF2 is connected to the K input of the 74F112, the output of the J-K flip flop will toggle and go

LOW at the next clock fall and release the "stretched clock". At the same time, FF1 is cleared. The timing diagram of this clock stretcher is shown in *Figure 9–7.*

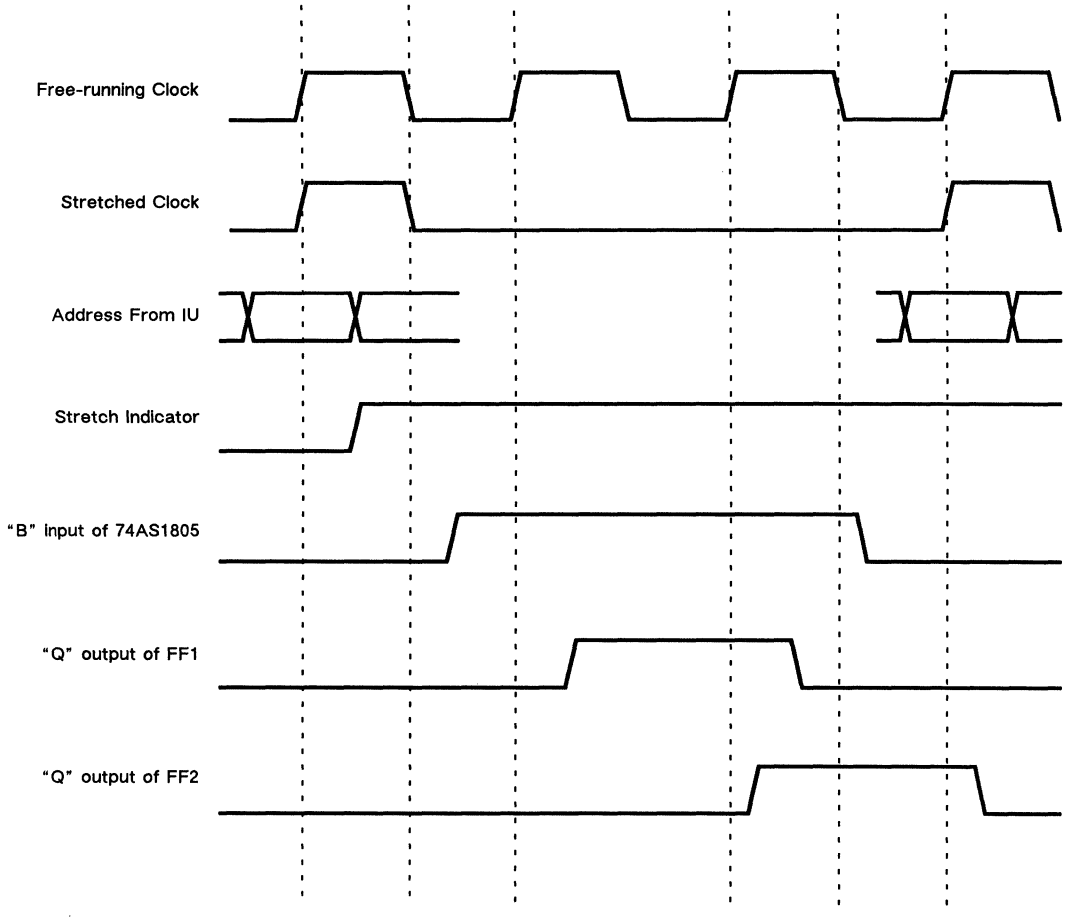


Figure 9–7. Clock Stretcher Timing

Figure 9–6 describes a clock stretcher with a fixed number of wait states. In a typical system, the number of wait states required by each device will be different. A programmable clock stretcher supporting 1 to 16 wait states can be constructed by substituting the dual D flip flop with a 74AS163 binary counter.

The IU generally performs one access per clock. However, a store cycle occupies two clocks with address checking occurring during the first clock and the actual writing during the second clock. If address checking can be completed in one regular clock period, then only the second clock should be stretched.

Under certain conditions such as access exceptions, the IU and the Floating-Point Controller will void an access by asserting the INULL and FNULL signals respectively. Clock stretching should be canceled when the access cycle is nullified. One approach to stop the clock stretcher logic is to clear the 74F112 flip flop when any one of the null signals is asserted.

Cache Design in a CY7C601 System

Cache memory is high speed memory located between the processor and main memory and is used for temporary information storage. In order to hold down system cost, the main memory section in most systems is implemented with slower and less expensive dynamic RAMs. As a result, one or more wait states are needed to access main memory. The cache memory, on the other hand, can return data to the processor with no wait states. As long as the information requested by the processor is found in the cache memory, processor references proceed without delay. Main memory is accessed only when the required data is not found in the cache. In a typical system, over 80% of the processor data are supplied by the cache.

The success of cache memory is based on the Principle of Locality. Two main aspects are stated in the Principle of Locality, temporal locality and spatial locality.

> Temporal locality states that data will be re-used: Information currently in use will be used again in the near future. An example of temporal locality is the do-loop statement.
>
> Spatial locality specifies that related data are usually stored together: Information which will be used in the near future are likely to be found near current data.
>
> The sequential behavior of most instruction streams is a good example of spatial locality. The cache memory puts the Principle of Locality to practice by maintaining a set of windows into the main memory with each window containing a set of recently used data and their immediate neighbors.

Building Blocks of a Cache System

Figure 9–8 shows a typical cache system block diagram. The major components are:

> Cache Tag Memory—This is a high speed memory which holds the directory of the cache. Each group of data in the cache memory is identified by a cache tag entry containing the high order address bits of that group. Tag entries are selected by low order address bits from the processor and the content of the selected entry is delivered to the tag comparison logic.
>
> Tag Comparator—The tag comparator compares the identifier stored in the cache tag memory with the high order address bits from the processor. A match indicates that a copy of the data referenced by the processor is found in the cache memory. The comparator output is delivered to the cache controller.
>
> Cache Memory—This is a fast static memory containing data recently referenced by the processor.
>
> Cache Controller—The cache controller regulates data transfer between the main memory and the processor. It is also responsible for updating the cache memory and the cache tag memory during a cache miss.

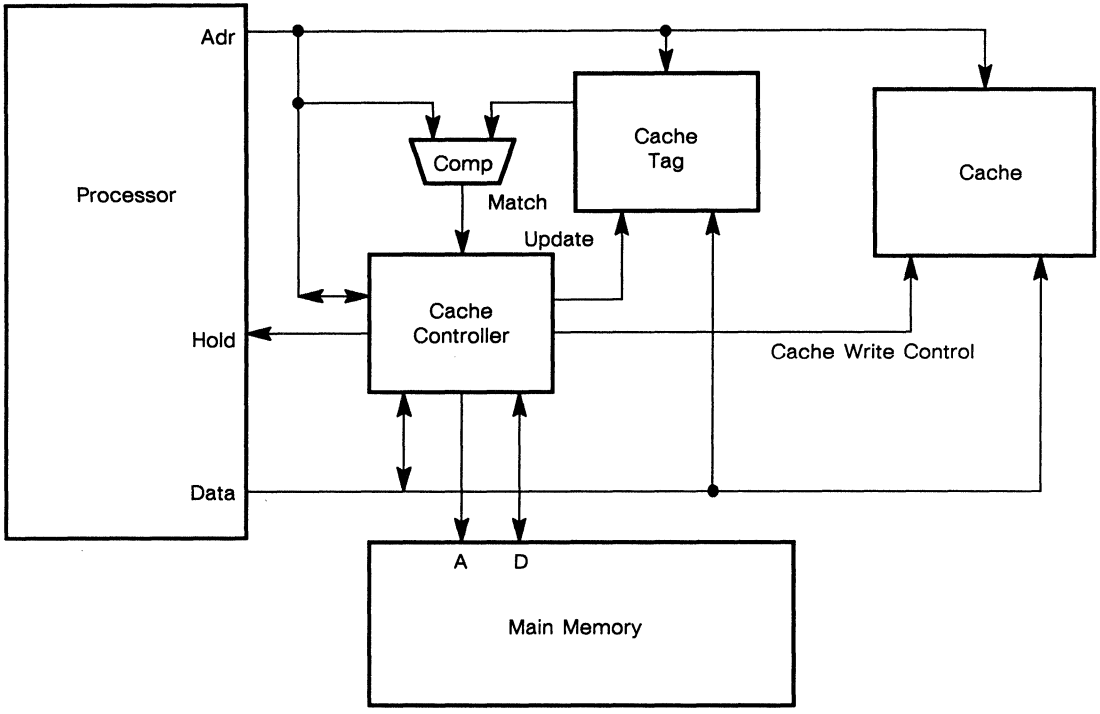


Figure 9–8. Cache Block Diagram

Cache Memory Operations

When the processor reads data from memory, the lower address bits are used to select an entry in the tag memory. The content of the selected tag entry is compared against the upper address bits from the processor. If a match is detected, data from the cache is returned to the processor. If the tag comparator signals a miss, then data requested by the processor is not found in the cache and the cache controller will be activated to transfer information from main memory to the processor and the cache. At the completion of the transfers, the directory in the tag memory is updated to reflect the change in the cache contents. This sequence of operations ensures that the cache is continuously updated with the most current information.

During processor write cycles, cache updates are dictated by the write policy. If a write-through policy is implemented, data is written into the cache and the main memory in case of a match. In this manner, main memory always contains the latest data. If a write-back policy is chosen, only the cache memory will be modified during the write access. Main memory is updated whenever a cache line (a group of cache data associated with one cache tag entry) is replaced. Although this policy enhances system performance by reducing bus traffic between the cache and main

memory, it introduces stale data in main memory which causes data inconsistencies in a multiprocessor system. Complicated "ownership protocols" are needed to resolve this problem.

If the processor write causes a tag miss, the cache is updated according to a second write policy. With a No-write-allocate policy, the write data is stored into main memory without updating the cache. If a Write-allocate policy is selected, the cache is updated after the data is stored into main memory.

Cypress Cache Support Chips

Cypress Semiconductor provides two cache support devices to simplify high speed cache memory design in CY7C601-based systems. The first device (CY7C181) is a sophisticated 4K-entry cache tag RAM with on-chip tag comparator and byte write generation logic. The second device (CY7C153) is a fast 32K by 8 cache RAM.

The CY7C181 Cache Tag RAM

The CY7C181 Cache Tag RAM provides storage for 4096 tag entries accessible by the processor. Each entry contains a 20-bit tag and 5 status bits: one valid bit, one dirty bit, and three protection bits. Twelve low order address lines from the processor are used to select one of the tag entries and its tag field is compared with 20 high order address bits by an on-chip comparator. The comparison result is available as an output to the cache controller. Interface to the CY7C601 IU is simplified by the inclusion of on-chip address and data latches as well as IU handshake signals such as Memory Hold, Memory Data Strobe, and Memory Exception.

Cache operations are greatly enhanced by the CY7C181: Read accesses benefit from the fast tag access and comparison features while the cache write logic is simplified by the provision of byte write signals.

The Cache Tag RAM reduces the amount of glue logic in the cache design by accepting a number of cache related signals from the system. These signals include the non-cacheable signal from the address decoder, the NULL signals from the IU and the FPC, the Address Space Identifiers (ASI) as well as the read and write signals from the IU. The block diagram of the cache tag RAM is shown in *Figure 9–9*.

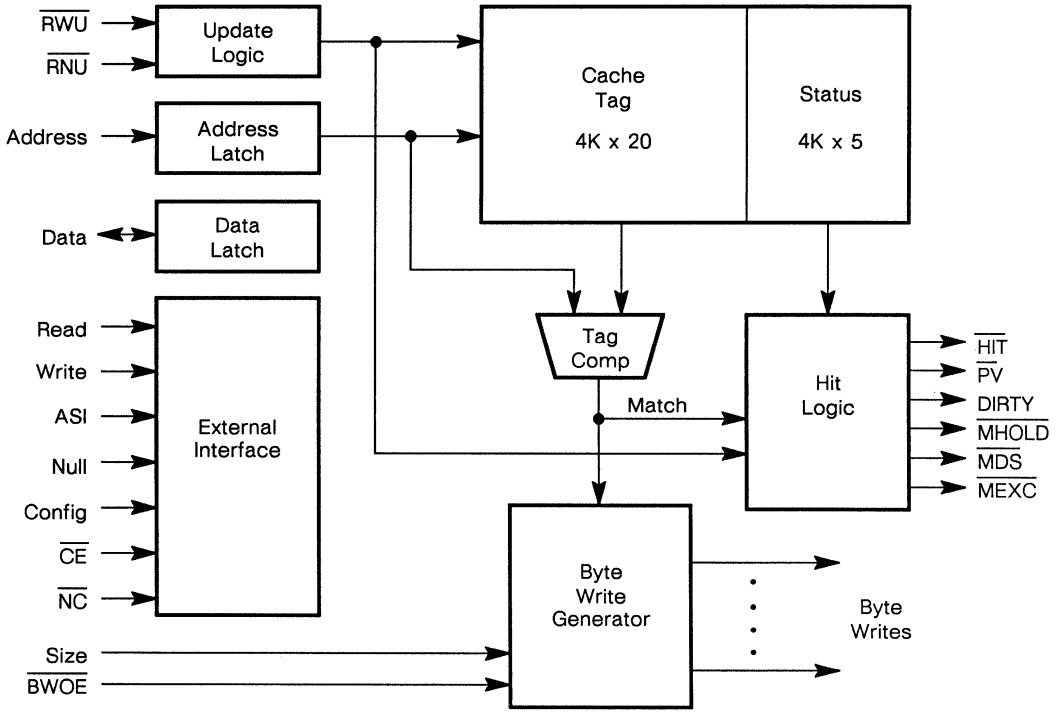


Figure 9–9. CY7C181 Cache Tag Ram Block Diagram

Processing an IU Cache Memory Access

During an IU access, address from the processor is latched on the rising edge of the system clock. The contents of the selected tag entry are processed by on-chip logic and the match signal is generated only when all of the following conditions are met:

The address is valid (The NULL signals are inactive).

The address Tag Comparator indicates a match.

The tag entry is valid (the valid bit of the entry is set).

The Non-cacheable (NC) input is inactive.

If a match is successful in a read access, the CY7C181 starts a new checking sequence in the next clock. If a match is successful in a write access, the CY7C181 generates the appropriate byte write signals to the cache RAMs in the next clock according to the SIZE information supplied by the IU. The two-clock write cycle is implemented to match the two-clock store sequence of the CY7C601 IU.

If a miss is detected, the Cache Tag RAM will suspend operations in the IU by asserting the Memory Hold signal. While the processor is held in a continuous wait state, the cache controller will get the requested data and update cache memory and IU. After all transfers are completed, the cache controller will update the tag storage and remove the CY7C181 from the hold state by issuing a Release with Update signal to the cache tag RAM. In situations where tag update is not necessary (such as a non-cacheable access), the CY7C181 can be removed from the hold state by a second release signal. The second release signal does not cause an update (Release No Update).

Access Protection

The CY7C181 Cache Tag RAM issues a protection violation to the processor whenever the access level specified by the three protection bits in the selected tag entry is not matched by the current access status. Current IU access status such as supervisor/user, instruction/data, read/write and so on are conveyed by the ASI code as well as the read/write signals. For example, if the 3 protection bits indicate that the entry is open to read data accesses in both supervisor and user modes, and the current access is a user write, the CY7C181 will issue a protection violation to the processor. If a protection violation is detected in a write cycle, all byte write signals will be de-asserted to prevent corrupting memory data.

Caching Policy

The CY7C181 understands the difference between write-through and write-back caching policies. A strap input pin, WP, sampled during reset, informs the Cache Tag RAM which policy is selected. The setting of the dirty bit in each tag entry during write cycles is controlled by the caching policy.

Tag Invalidation

Each tag entry can be individually invalidated by writing a zero into its associated valid bit. If the entire tag storage has to be invalidated, the user has two alternatives:

Software Invalidation—Writing to a special location in the CY7C181 will invalidate all tag entries.

Hardware Invalidation—Asserting the Cache Tag Invalidate input pin for one clock cycle will invalidate all tag entries.

Byte Write Generation

Since the CY7C601 IU is capable of accessing bytes, half words (two bytes), and words (four bytes), individual byte write signals are needed to enable only the bytes specified by the processor during write operations. Four byte write signals are required to support one bank of 32-bit wide cache memory.

The CY7C181 supports up to two cache memory banks by providing eight byte write signals. The byte write outputs are controlled by the IU SIZE code, the cache memory configuration (1 or 2 memory banks, supplied via the CONFIG input), and the three lowest address bits from the processor. For STORE single operations, the appropriate byte write signal(s) are asserted in the second clock of the cycle if a match is detected with no protection violation. For STORE double operations, byte writes are generated in both the second and third clocks. The byte writes are connected to the write inputs of the CY7C153 cache RAMs and they can be tri-stated by the cache controller via the Byte Write Enable input (Byte Write Output Enable).

Cascading the CY7C181

Multiple CY7C181s can be used together to support either a larger cache or a smaller line size. Each CY7C181 is equipped with a chip enable signal. De-assertion of chip select suspends on-chip functions and place all outputs in a tri-state condition. An external address decoder is required to select one of the Cache Tag RAMs for a particular access. *Figure 9–10* shows a cache system with four CY7C181 devices.

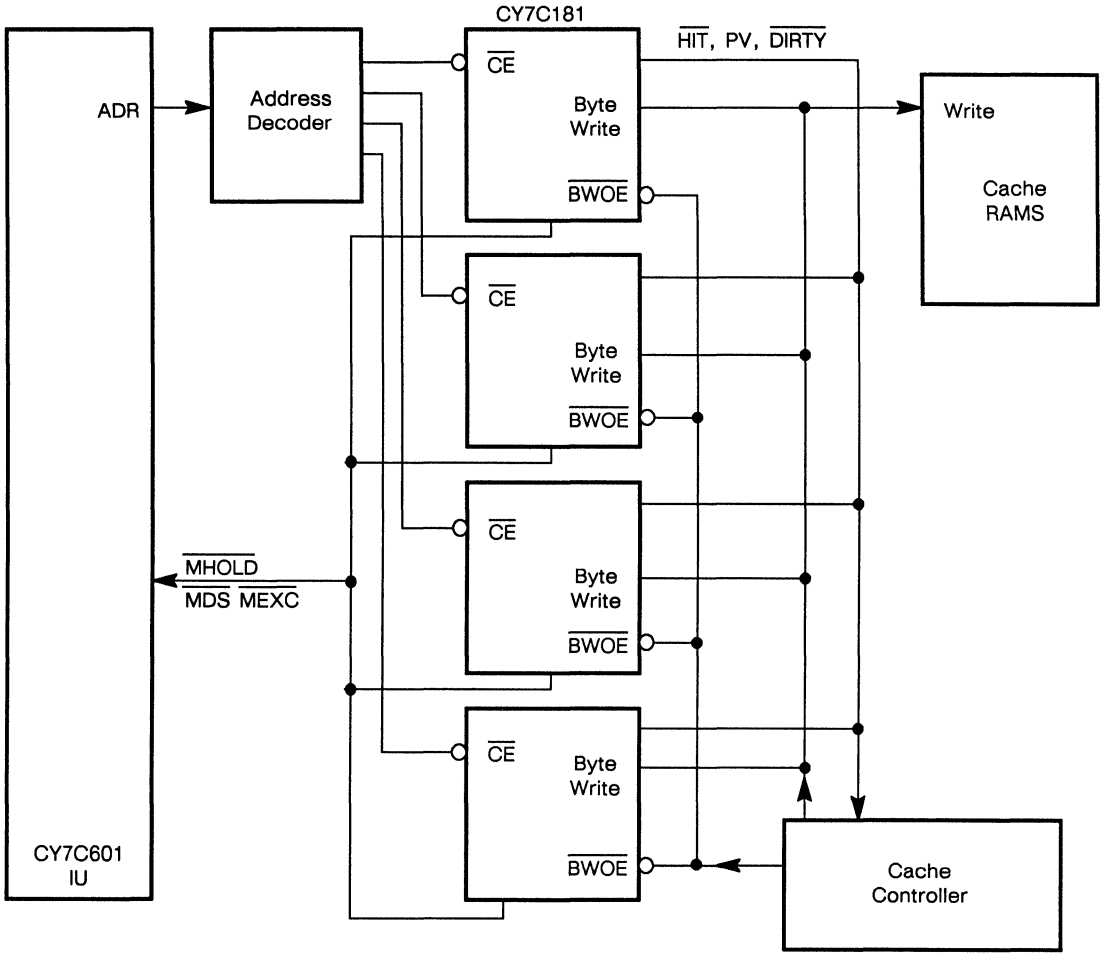


Figure 9–10. Cache System with 4 Cascaded CY7C181's

The CY7C153 Cache RAM

The CY7C153 cache RAM is a common I/O static RAM organized as 16K by 4 bits. Major features include input address and data latches, output data latch, self-timed write, and centered power pins. Address access time is approximately 17 nS.

The on-chip address latch captures IU addresses at the rising edge of the clock and uses them for the rest of the cycle. The data latch is transparent and it is open during the high portion of the clock. An output data latch is incorporated on-chip to satisfy the data hold time requirements of the IU.

Write operations are internally self-timed and initiated by the high to low transition of the write input. This feature eliminates complex off-chip write pulse generation which is difficult to accomplish at 33 MHz. The block diagram of the CY7C153 cache RAM is shown in *Figure 9–11.*

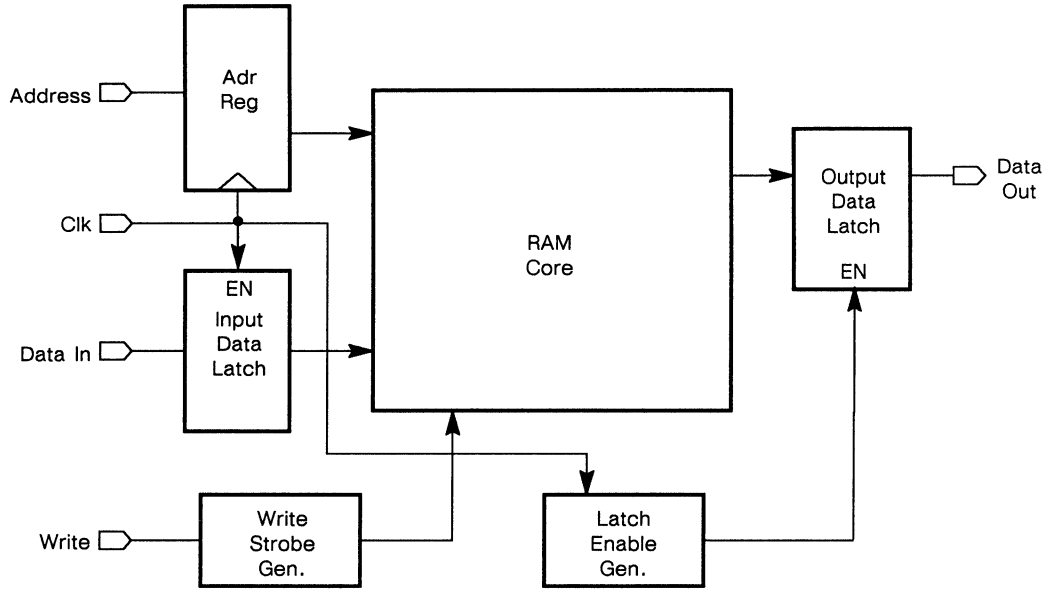


Figure 9–11. CY7C153 Registered Cache RAM

Functional Description of a Sample Cache

The block diagram of a CY7C601-based direct-mapped cache system is shown in *Figure 9–12* and *Figure 9–13.* Major function blocks include:

The CY7C601 Integer Unit

The CY7C181 Cache Tag RAM

256K-byte Cache RAMs (Eight CY7C153s)

The Cache Controller

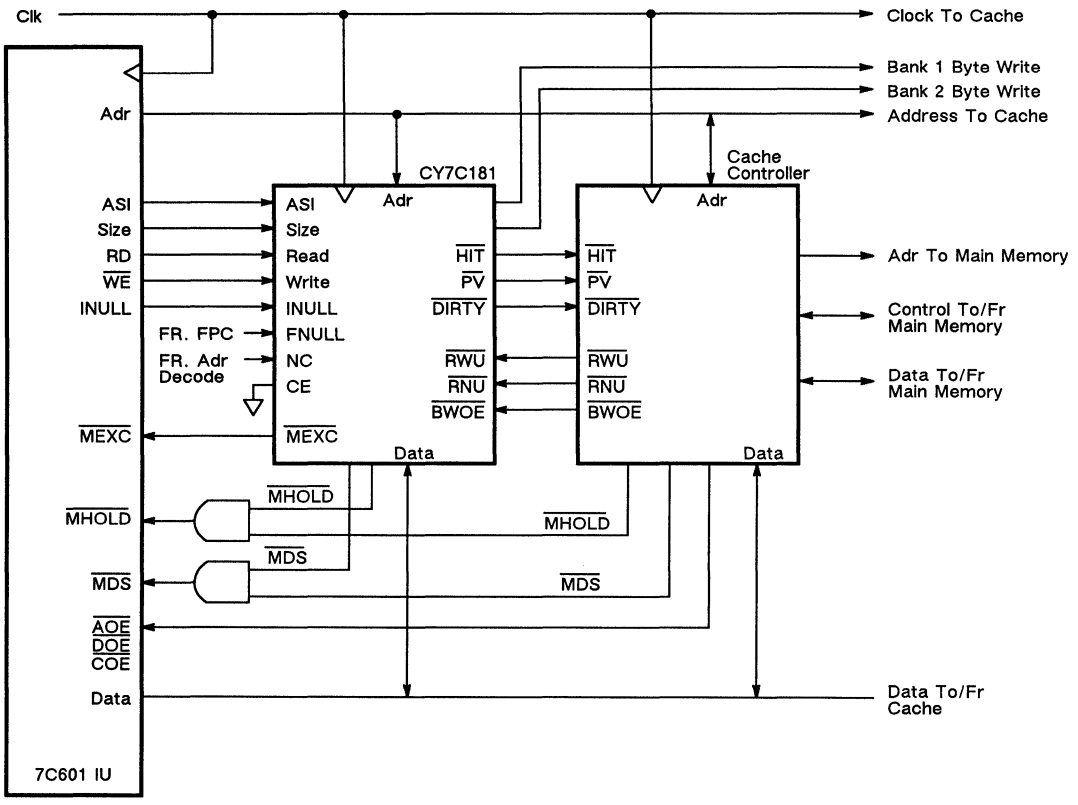


Figure 9–12. IU, Cache Tag and Cache Controller

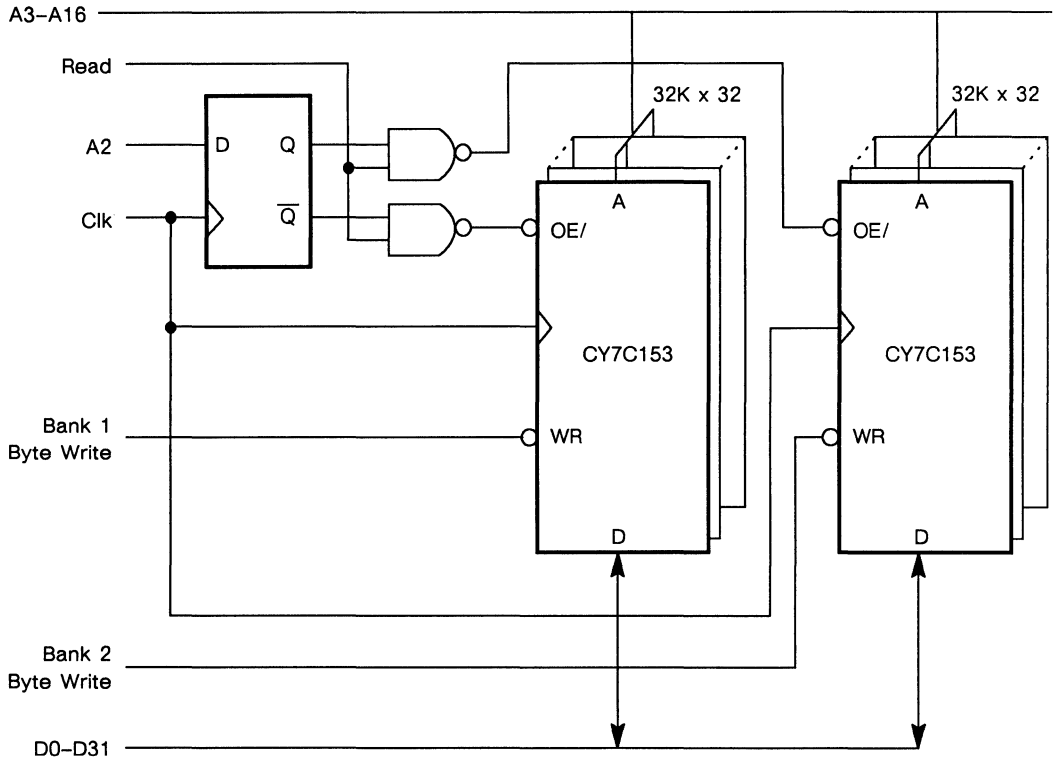


Figure 9–13. 256K-Byte Cache Memory

Main memory

Address from the IU is latched by the Cache Tag RAM and the cache data RAMs at the rising edge of the clock. A single bank of four CY7C153s provide a 128K–byte cache. With a 4K–entry Cache Tag, line size is 32 bytes. The cache data RAM may also be divided into two 32-bit wide banks with address line A2 serving as the bank selector. Data is stored alternately between the 2 cache banks. With a 4K-entry Cache Tag, line size is 64 bytes. A write through cache policy is implemented.

Read Accesses

During a read access, if the cache tag RAM indicates a hit, data from the chosen cache bank will be returned to the processor. If an address miss is detected, the cache tag RAM will hold the IU in a continuous wait (by asserting the Memory Hold signal) so that the cache controller can transfer data from main memory into the cache RAMs. In order to gain access to the cache data

RAMs, the cache controller first tri-states the address, data, and control lines from the IU (via All Output Enable, Data Output Enable, and Control Output Enable) and the byte write signals from the CY7C181 (via Byte Write Output Enable). Once the above signals are tri-stated, the cache controller outputs its own address and control signals to transfer the new cache line from main memory into the CY7C153s. At the end of the data transfers, the controller delivers the missed data to the IU and loads it by pulsing the Memory Data Strobe signal. With the new data in place, the cache tag is now ready to be modified to reflect the new cache line. The cache controller updates the tag and releases it from the hold state by:

Placing the "missed" address on the address line

Placing the new protection bits at the PI inputs of the cache tag RAM

Pulsing the RWU/ (release with update) input for one clock period

If the access is declared as non-cacheable by the address decoder, the CY7C181 will generate a miss and hold the IU (via Memory Hold) so that the access can proceed at its own speed. After the requested data is delivered to the IU (via Memory Data Strobe), the CY7C181 should be released from the hold state but with no tag update because the access is not cacheable. The cache controller accomplishes this function by asserting the Release No Update input of the cache tag RAM.

If a protection violation is found, the access will be canceled. A hit will be generated by the CY7C181 to prevent activation of the cache controller. At the same time, a memory exception is delivered to the IU via the Memory Hold, Memory Data Strobe, and Memory Exception signals.

Write Accesses

Write cycles are multi-clock operations: A STORE single cycle occupies 2 clocks and a STORE double cycle occupies 3 clocks. Because all STORE double accesses are aligned on 8-byte boundaries, both memory locations will fall inside the same cache line.

If the address is matched during the first clock, the first write will occur during the second clock and the second write will occur during the third clock (for STORE double). Since the IU is capable of 8-bit, 16-bit, and full 32-bit writes, individual byte write signals are needed. The CY7C181 generates 2 sets (4 byte writes per set) of byte writes based on the SIZE indicators from the processor and the number of cache banks in the system to support systems with single or dual cache banks.

If an address miss is detected, the cache tag RAM will hold the IU in a continuous wait (by asserting the Memory Hold signal) so that the cache controller can write the data from the IU into main memory and cache RAMs. All byte writes from the CY7C181 are de-asserted because cache update is performed by the cache controller. After the new data is stored, the cache tag is ready to be modified. The cache controller updates the tag and releases it from the hold state by presenting the "missed" address and the new protection bits to the CY7C181 and then pulses the Release With Update input for one clock period. With a STORE single cycle, this is the end of the access. However, an additional write is needed to complete a STORE double cycle.

Since both addresses belong to the same cache line and the cache line has been updated, the second address will always result in a hit. The appropriate byte write signals will be generated by the CY7C181 to complete the second store. The cache controller may hold the IU in a continuous wait until the data is written into main memory. This additional delay can be eliminated by

incorporating a write buffer into the cache controller. The write buffer stores the address and data designated for main memory into registers and releases the cache controller immediately. Actual main memory writing is overlapped with the following few processor cycles which will probably be read accesses for data already stored in the cache.

If the access is declared as non-cacheable by the address decoder, the CY7C181 will generate a miss and hold the IU (via Memory Hold) so that the write cycle can proceed at its own speed. In addition, all byte writes will be de-asserted. After the requested data is written into the designated device, the CY7C181 should be released from the hold state with no tag update because the access is not cacheable. The Release No Update signal will be asserted by the cache controller to accomplish this function. For a STORE double cycle, the above sequence will be repeated for the second address.

If a protection violation is found, the access will be canceled. A hit will be generated by the CY7C181 to prevent activation of the cache controller. At the same time, all byte writes will be de-asserted and a memory exception will be delivered to the IU via the Memory Hold, Memory Data Strobe, and Memory Exception signals.

Summary

The CY7C181 cache tag RAM and the CY7C153 cache RAMs are specifically designed to simplify CY7C601-based cache systems. A high performance 128 or 256 K-byte direct-mapped cache can be built with one CY7C181, four or eight CY7C153s, a cache controller, and a small amount of interface logic.

CYPRESS SEMICONDUCTOR

CY7C601 Electrical & Timing

Maximum Ratings

(Above which the useful life may be impaired)

Storage Temperature . . . . . -65°C to +150°C

Ambient Temperature with Power Applied. . . . . -55°C to +125°C

Supply Voltage to Ground Potential[4]. . . -0.5V to +7.0V

DC Voltage Applied to Outputs in High-Z State . . . . . -0.5V to +7.0V

DC Input Voltage . . . . . -3.0V to +7.0V

Output Low Sink Current 30mA

Operating Range

Range	Ambient Temperature	Vcc
Commercial	0°C to +70°C	5V ± 10%
Military [3]	-55°C to +125°C	5V ± 10%

Electrical Characteristics Over the Operating Range[1]

Parameters	Description	Test Conditions	Min.	Max.	Units
V_{OH}	Output HIGH Voltage	Vcc=Min., I_{OH} = -2.0 mA	2.4		V
V_{OL}	Output LOW Voltage	Vcc=Min., I_{OL}= 8.0 mA		0.5	V
V_{IH}	Input HIGH Voltage		2.1	Vcc	V
V_{IL}	Input LOW Voltage		-3.0	0.8	V
I_{IH}	Input HIGH Current	Vcc=Max., Vin=Vcc		10	uA
I_{IL}	Input LOW Current	Vcc=Max., Vin=Vss		-10	uA
I_{OH}	Output HIGH Current	Vcc=Min., V_{OH}=2.4 V	-2.0		mA
I_{OL}	Output LOW Current	Vcc=Min., V_{OL}=0.5 V	8.0		mA
I_{OZ}	Output Leakage Current	Vcc=Max., Vss ≤ Vout ≤ Vcc	-40	40	uA
I_{SC}	Output Short Circuit Current	Vcc=Max., Vout = 0 V	-30	-180	mA
I_{CCQ}	Quiescent Supply Current	Vss ≤ Vin ≤ V_{IL} or V_{IH} ≤ Vin ≤ Vcc		400	mA
I_{CC}	Supply Current	Vcc=Max., f = 33MHz		600	mA

Capacitance [2]

Parameters	Description	Test Conditions	Max.	Units
C_{IN}	Input Capacitance	Vcc = 5.0 V, Ta = 25°C, f = 1 MHz	10	pF
C_{OUT}	Output Capacitance	Vcc = 5.0 V, Ta = 25°C, f = 1 MHz	12	pF
C_{IO}	I/O Bus Capacitance	Vcc = 5.0 V, Ta = 25°C, f = 1 MHz	15	pF

Notes:
1. See last page of this document for Group A subgroup testing information
2. Tested initally and after any design or process changes that may affect these parameters.
3. Ambient Temperature is the 'instant on' case temperature.
4. All power and ground pins must be connected to the other pins of same type before any power is applied to the part.

AC Test Loads and Waveforms

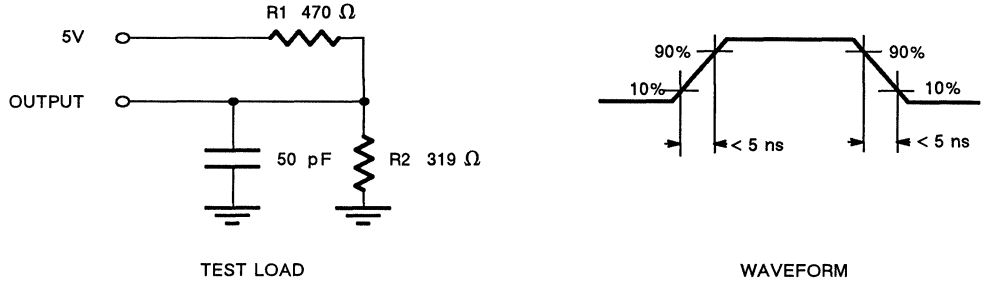


Figure 10–1. AC Test Loads

Switching Waveforms

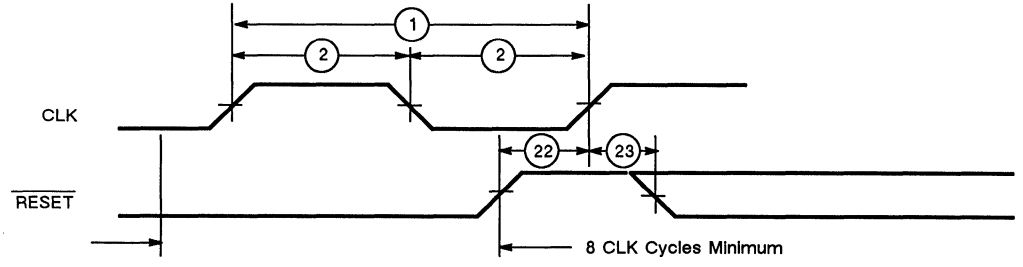


Figure 10–2. Clock & Reset Timing

Reset needs to be synchronized with CLK only if the processor must be guaranteed to be in step with other devices in the system (i.e. other processors)

Switching Characteristics Over the Operating Range [2,3]

Parameter		Description	CY7C601-33		CY7C601-25		Units
			Min.	Max.	Min.	Max.	
1	t_{CY}	Clock cycle	30	100	40	100	ns
2	t_{CHL}	Clock high and low	13		18		ns
3	t_{CRF}	Clock rise and fall		1		1	V/ns
4	t_{AD}	Address/Control output delay		26		35	ns
5	t_{AH}	Address/Control output hold	7		7		ns
6	t_{DOD}	Data output delay		15		20	ns
7	t_{DOH}	Data output hold	4		4		ns
8	t_{MAD}	MAO to Address/Control output delay		14		19	ns
9	t_{MAH}	MAO to Address/Control output hold	2		2		ns
10	t_{DIS}	Data input setup	3		5		ns
11	t_{DIH}	Data input hold	5		7		ns
12	t_{MES}	Memory Exception input setup	15		20		ns
13	t_{MEH}	Memory Exception input hold	1		2		ns
14	t_{HS}	Hold input setup	5		7		ns
15	t_{HH}	Hold input hold	5		7		ns
16	t_{OE}	Output enable delay		16		18	ns
17	t_{OD}	Output disable delay		16		18	ns
18	t_{TOE}	Output enable delay for All Outputs		19		21	ns
19	t_{TOD}	Output disable delay for All Outputs		19		21	ns
20	t_{SSD}	Synchronous Signal output delay [1]		15		20	ns
21	t_{SSH}	Synchronous Signal output hold [1]	3		3		ns
22	t_{RS}	Reset input setup	10		15		ns
23	t_{RH}	Reset input hold	3		3		ns
24	t_{HOD}	Hold to Address/Control output delay		16		24	ns
25	t_{HOH}	Hold to Address/Control output hold	0		0		ns
26	t_{FD}	FPU/Coprocessor Signal output delay		18		27	ns
27	t_{FH}	FPU/Coprocessor Signal output hold	4		4		ns
28	t_{FIS}	FPU/Coprocessor Signal input setup	8		10		ns
29	t_{FIH}	FPU/Coprocessor Signal input hold	3		4		ns

Switching Characteristics Continued

Parameter		Description	CY7C601-33		CY7C601-25		Units
			Min.	Max.	Min.	Max.	
30	t_{DXD}	DXFER output delay		23		30	ns
31	t_{DXH}	DXFER output hold	2		2		ns
32	t_{HDXD}	Hold to DXFER output delay		15		20	ns
33	t_{HDXH}	Hold to DXFER output hold	0		0		ns

Notes:
1. Includes INULL, INST, FLUSH, FXACK, CXACK, INTACK, and ERROR signals
2. Test conditions assume signal transition times of 5 ns or less, a timing reference level of 1.5V, input levels of 0 to 3.0V, and output loading of 50pf capacitance.
3. See the last page of this specification for Group A subgroup testing information

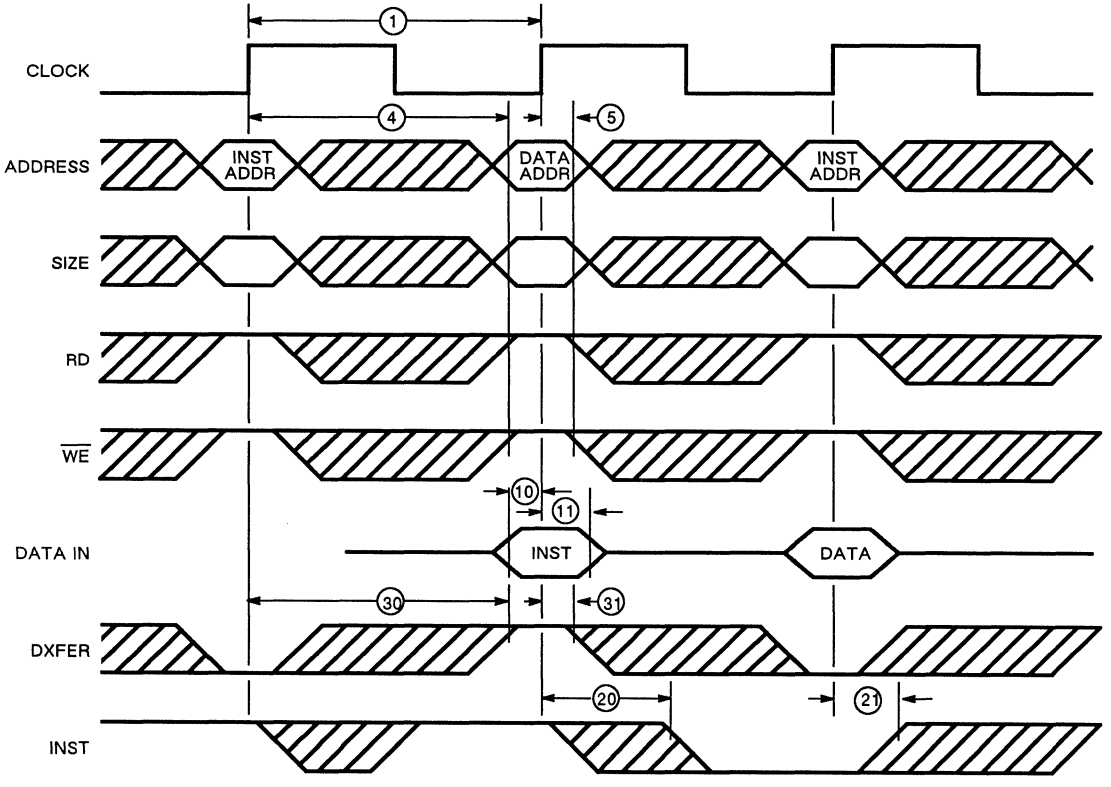


Figure 10-3. Load Timing

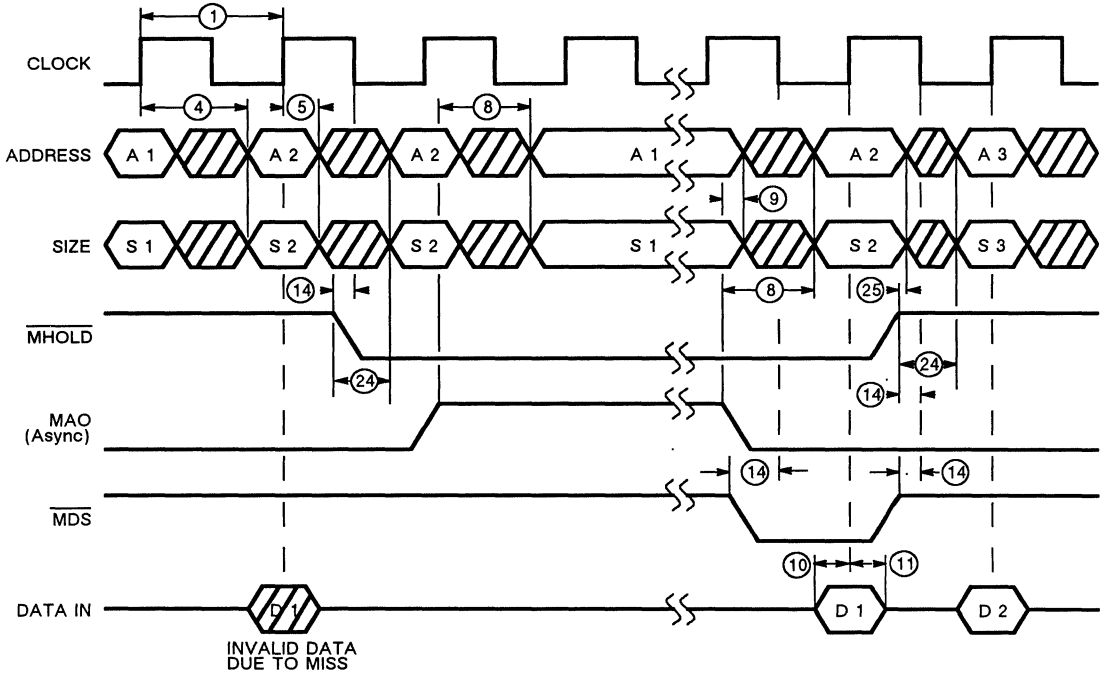


Figure 10–4. Load with Miss Timing

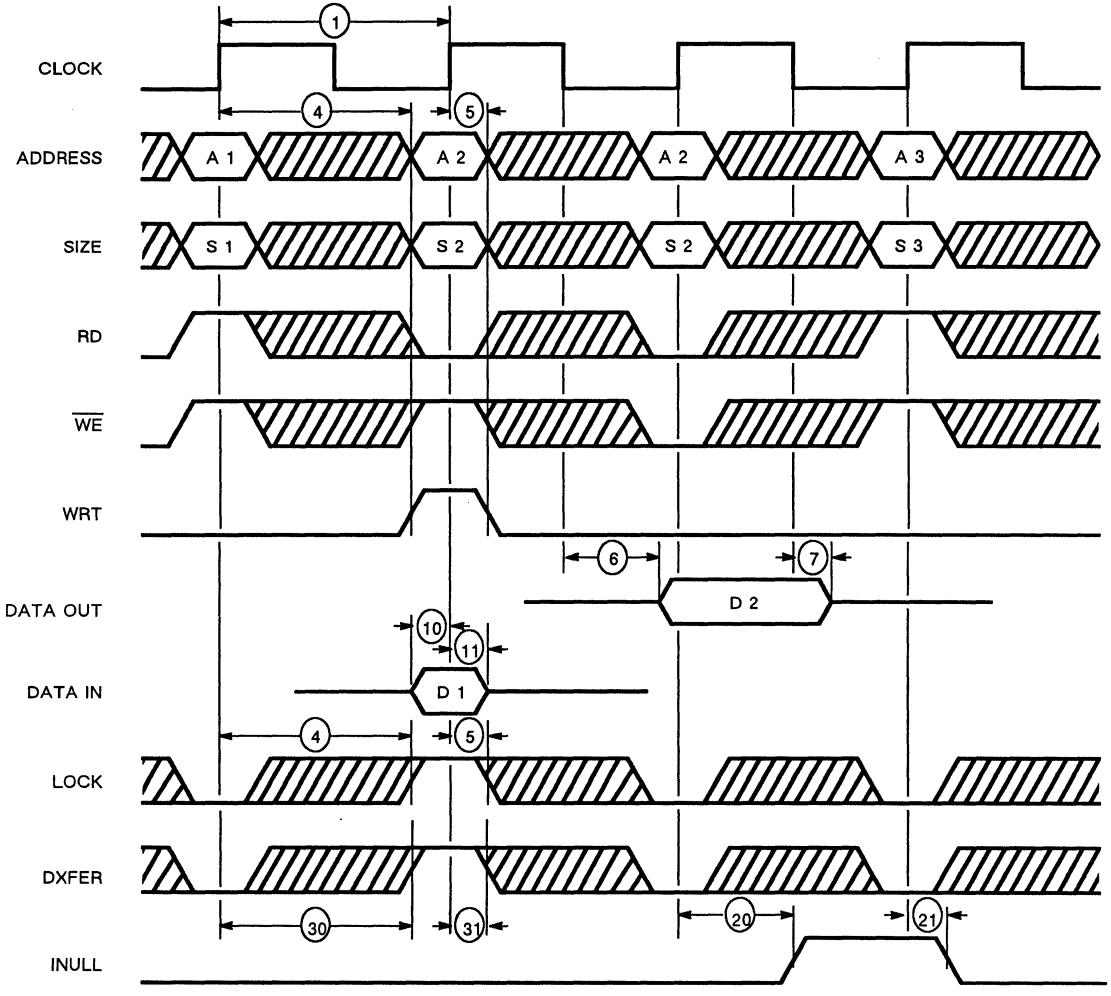


Figure 10–5. Load followed by Store Timing

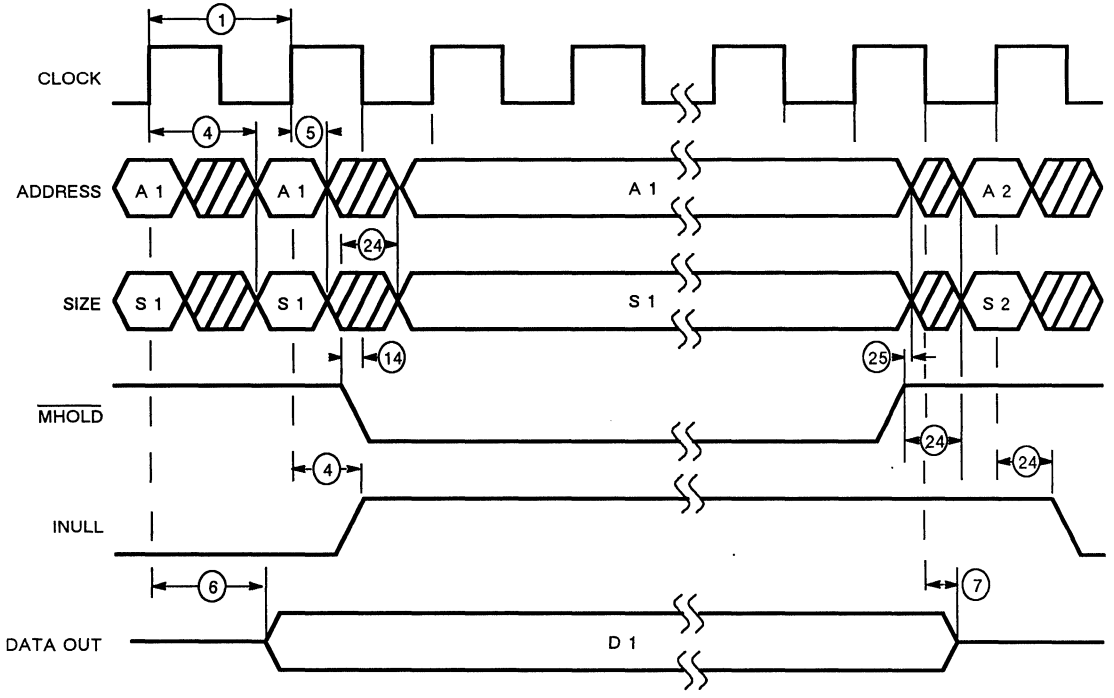


Figure 10–6. Store with Miss Timing

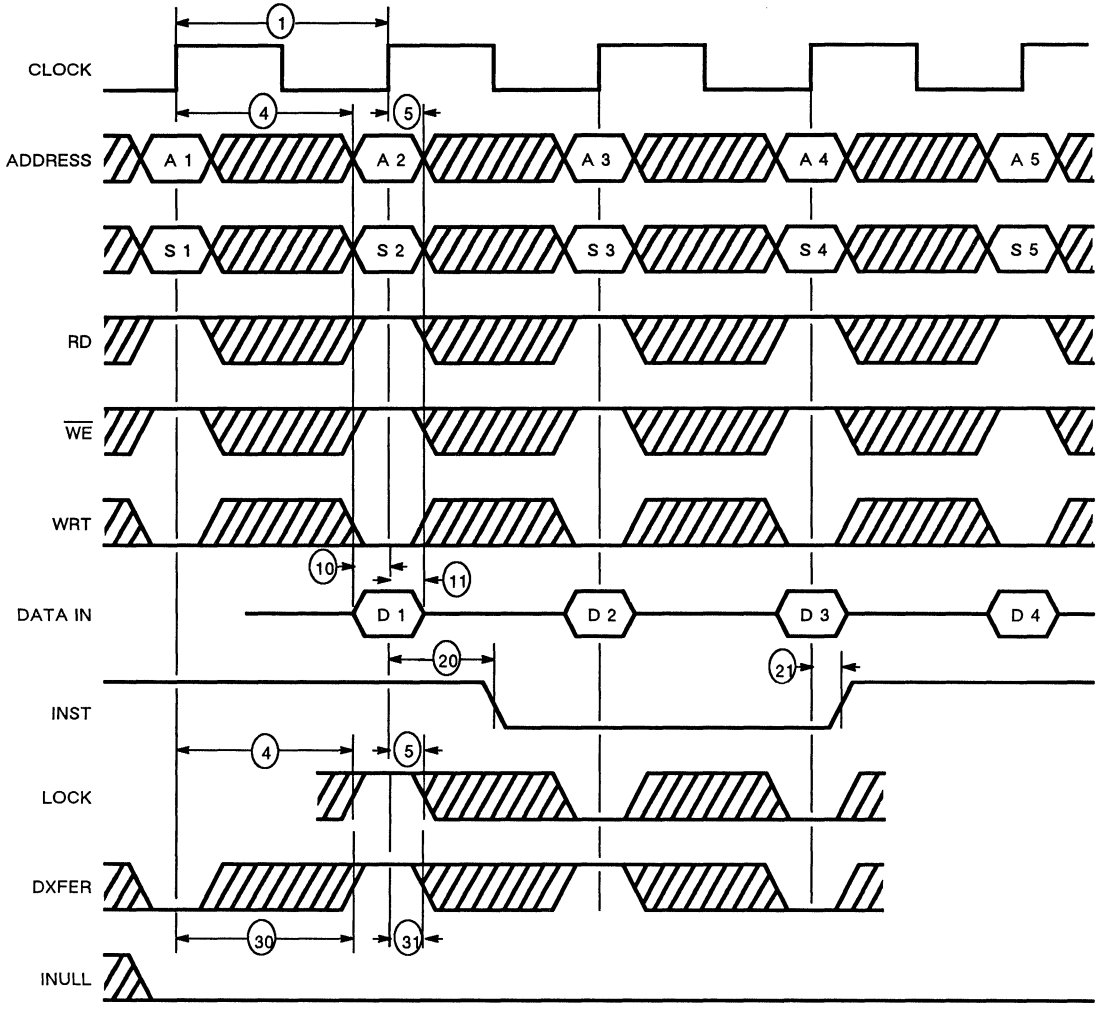


Figure 10–7. Load Double Timing

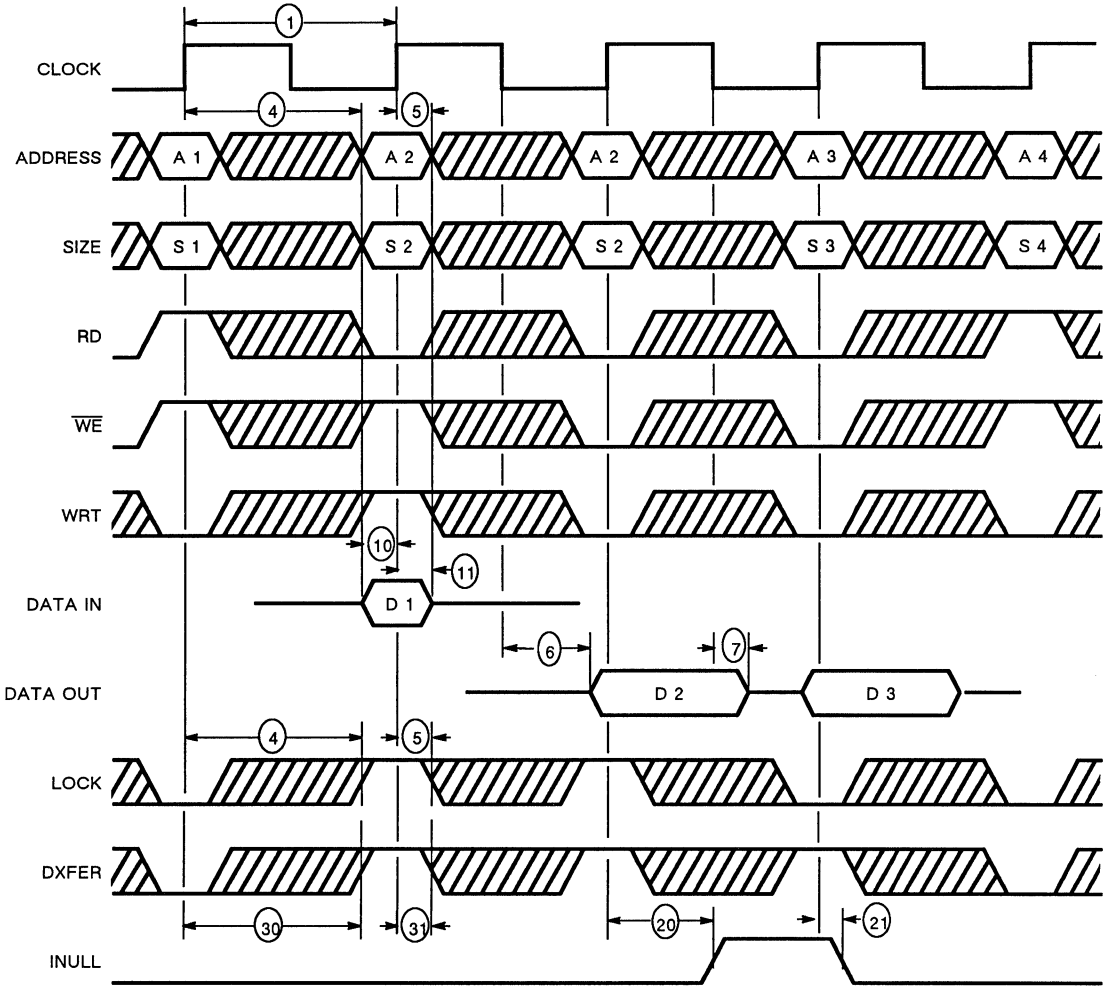


Figure 10–8. Store Double Timing

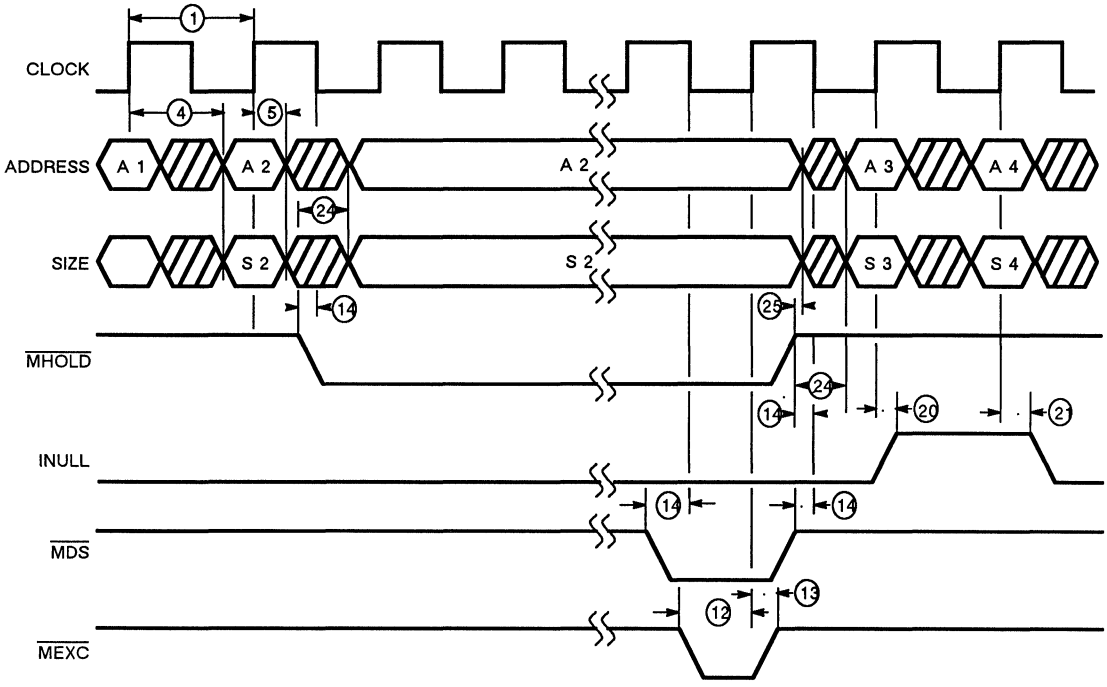


Figure 10–9. Read With Memory Exception Timing

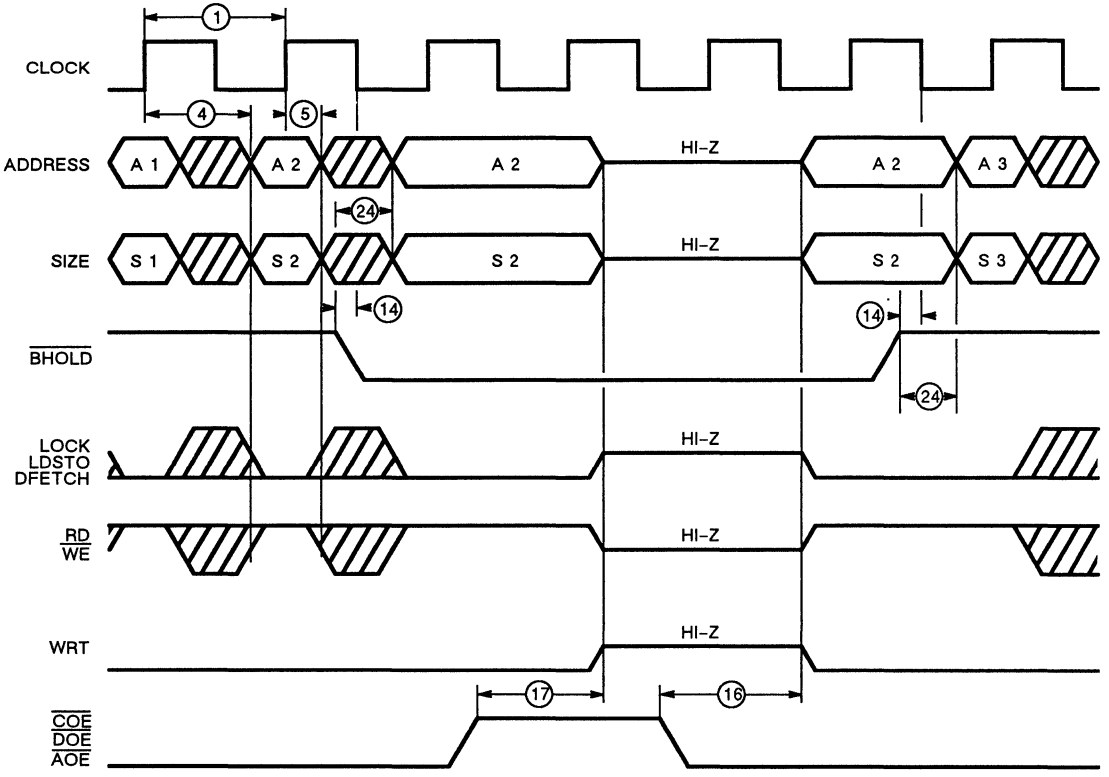


Figure 10–10. Bus Arbitration Timing

Military Specifications
Group A Subgroup Testing

DC Characteristics

Parameters	Subgroups
V_{OH}	1,2,3
V_{OL}	1,2,3
V_{IH}	1,2,3
V_{IL}	1,2,3
I_{IH}	1,2,3
I_{IL}	1,2,3

Parameters	Subgroups
I_{OH}	1,2,3
I_{OL}	1,2,3
I_{OZ}	1,2,3
I_{SC}	1,2,3
I_{CCQ}	1,2,3
I_{CC}	1,2,3

Switching Characteristics

Parameter		Subgroups
1	t_{CY}	7,8,9,10,11
2	t_{CHL}	7,8,9,10,11
4	t_{AD}	7,8,9,10,11
5	t_{AH}	7,8,9,10,11
6	t_{DOD}	7,8,9,10,11
7	t_{DOH}	7,8,9,10,11
8	t_{MAD}	7,8,9,10,11
9	t_{MAH}	7,8,9,10,11
10	t_{DIS}	7,8,9,10,11
11	t_{DIH}	7,8,9,10,11
12	t_{MES}	7,8,9,10,11
13	t_{MEH}	7,8,9,10,11
14	t_{HS}	7,8,9,10,11
15	t_{HH}	7,8,9,10,11
16	t_{OE}	7,8,9,10,11

Parameter		Subgroups
17	t_{OD}	7,8,9,10,11
18	t_{TOE}	7,8,9,10,11
19	t_{TOD}	7,8,9,10,11
20	t_{SSD}	7,8,9,10,11
21	t_{SSH}	7,8,9,10,11
22	t_{RS}	7,8,9,10,11
23	t_{RH}	7,8,9,10,11
24	t_{HOD}	7,8,9,10,11
26	t_{FD}	7,8,9,10,11
27	t_{FH}	7,8,9,10,11
28	t_{FIS}	7,8,9,10,11
29	t_{FIH}	7,8,9,10,11
30	t_{DXD}	7,8,9,10,11
31	t_{DXH}	7,8,9,10,11
32	t_{HDXD}	7,8,9,10,11

CHAPTER 11

Ordering & Mechanical Information

The CY7C601 SPARC Integer Processing Unit is available in two packages, a Plastic Multilayer type and a Ceramic Multilayer type. The Pinout and physical characteristics of both units are identical, and are shown in this chapter. Both packages have tuned 50 ohm characteristic impedance to provide the lowest noise and best interface characteristics possible.

In Table 12–1, there are three separate power systems shown. V_{CCO} and V_{SSO} provide power for the output drivers only. VCCI and VSSI provide power for the chip internals while VCCT and VSST provide power for the device input receivers. This separation of power allows offchip decoupling and provides the ability to manage power and noise in the system.

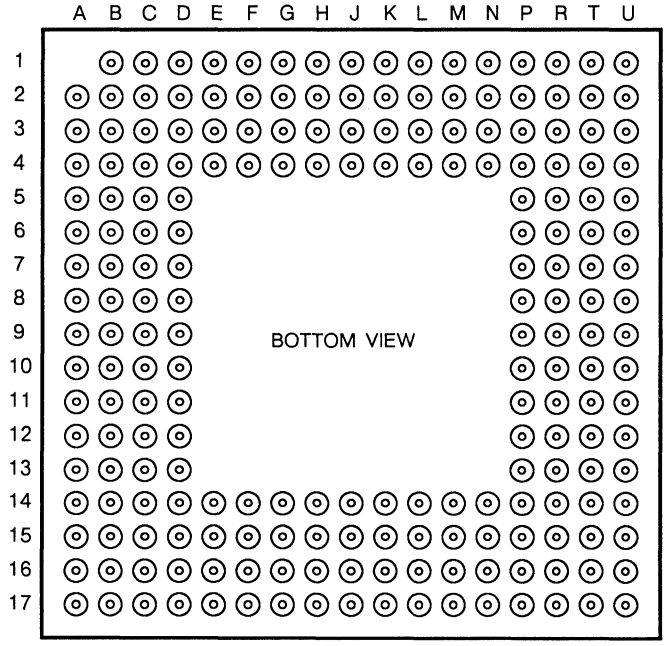


Figure 11–1. Pin Grid Array Package

Pin Name	Description	Input/Output	Active
A(0–31)	Address	3–State Output	
ASI(0–7)	Address Space Identifier	3–State Output	
D(0–31)	Data	3–State BiDir.	
MEXC	Memory Exception	Input	Low
MHOLDA	Memory Hold A	Input	Low
MHOLDB	Memory Hold B	Input	Low
BHOLD	Bus Hold	Input	Low
AOE	Address Output Enable	Input	Low
DOE	Data Output Enable	Input	Low
COE	Control Output Enable	Input	Low
MDS	Memory Data Strobe	Input	Low
MAO	Memory Address Output Sel.	Input	
IFT	Instruction Flush Trap	Input	Low
SIZE(0–1)	Bus Transaction Size	3–State Output	
RD	Read	3–State Output	High
WE	Write	Output	Low
LDSTO	Load/Store Operation	Output	High
INULL	Null Cycle	Output	High
LOCK	Multi–Cycle Bus Lock	Output	High
DFETCH	Data Fetch Cycle	Output	High
WRT	Advanced Write	Output	High
FP	FPU Present	Input w Pullup	Low
FCC(0–1)	FPU Condition Codes	Input	
FCCV	Condition Codes Valid	Input	High
FHOLD	FPU Hold	Input	Low
FEXC	FPU Exception	Input	Low
CP	Coprocessor Present	Input w Pullup	Low
CCC(0–1)	CP Condition Codes	Input	
CCCV	Condition Codes Valid	Input	High
CHOLD	CP Hold	Input	Low
CEXC	CP Exception	Input	Low
INST	Instruction Fetch Cycle	Output	High
FLUSH	Flush FP/CP Instruction	Output	High
FINS1	FP Instruction Stage 1	Output	High
FINS2	FP Instruction Stage 2	Output	High
FXACK	FP Exception Acknowledge	Output	High
CINS1	CP Instruction Stage 1	Output	High
CINS2	CP Instruction Stage 2	Output	High
CXACK	CP Exception Acknowledge	Output	High
IRL(0–3)	Interrupt Request Lines	Input	Low
INTACK	Interrupt Acknlwledge	Output	High
RESET	System Reset	Input	Low
ERROR	IU Error Mode	Output	Low
TOE	Test Mode Output Enable	Input	High
FPSYN	FPU Synonym Mode	Input	
CLK	System Clock	Input	
VSSO	Output Driver Ground	Ground	
VCCO	Output Driver Power	Power	
VSSI	Main Internal Ground	Ground	
VCCI	Main Internal Power	Power	
VSST	Input Circuit Ground	Ground	
VCCT	Input Circuit Power	Power	

Table 11–1. Pin Function

Pin Name	Pin Number
A0	K2
A1	K1
A2	L3
A3	L1
A4	L2
A5	M2
A6	N2
A7	M1
A8	M3
A9	P1
A10	P2
A11	N1
A12	N3
A13	R3
A14	R2
A15	R4
A16	T4
A17	T5
A18	R6
A19	T6
A20	U5
A21	U6
A22	U7
A23	T7
A24	U8
A25	T8
A26	U9
A27	R8
A28	T9
A29	R9
A30	T10
A31	U11
D0	R10
D1	T11
D2	U12
D3	T12
D4	U13
D5	T13
D6	T14
D7	R13
D8	U14
D9	U15
D10	R15
D11	P15
D12	N15
D13	M15
D14	M16
D15	N16
D16	L15
D17	M17
D18	L16
D19	L17
D20	K16
D21	K17
D22	J16
D23	J17
D24	H17
D25	H15
D26	G17
D27	H16
D28	G16
D29	F16
D30	F15
D31	G15

Pin Name	Pin Number
ASI0	F3
ASI1	F2
ASI2	G3
ASI3	G2
ASI4	G1
ASI5	H2
ASI6	H1
ASI7	J1
SIZE0	E2
SIZE1	D2
$\overline{\text{MEXC}}$	D8
MHOLDA	C8
MHOLDB	B8
BHOLD	A7
AOE	P3
COE	C2
DOE	N17
MDS	B7
MAO	E3
IFT	C14
RD	A4
WE	B4
LDSTO	C5
INULL	B5
LOCK	D4
DXFER	D3
WRT	E4
FP	C7
FCC0	A11
FCC1	B11
FCCV	C10
FHOLD	A8
FEXC	A5
CP	B6
CCC0	A12
CCC1	B13
CCCV	B10
CHOLD	C9
CEXC	A6
INST	C6
FLUSH	B14
FINS1	E17
FINS2	D16
FXACK	D11
CINS1	D15
CINS2	C17
CXACK	C13
IRL0	A10
IRL1	C11
IRL2	D10
IRL3	B12
INTACK	A13
RESET	A9
ERROR	B15
TOE	C15
FPSYN	C12
CLK	K3

Pin Name	Pin Number
VSSO	B16 H4 T16 B17 J2 T17 C3 K14 U16 C4 N14 U17 D6 P4
	D14 P6 F1 P11 F4 P14 F14 R5 F17 R14
VCCO	A15 L4 A16 M14 A17 N4 D1 P8 D12 P12 D17 P16 E1 P17 G4 R16 K4 R17 K15
VSSI	A3 J3 U2 A14 L14 U10 B2 M4 B3 P5 B9 P7 C1 R1 C16 R11 D13 T1 E15 T15 H14 U1
VCCI	A2 R12 B1 T2 D7 T3 E14 U3 E16 U4 G14 H3 J15 P10 R7
VSST	D9 J4 J14 P9
VCCT	D5 P13

Table 11-2. Pin Connections

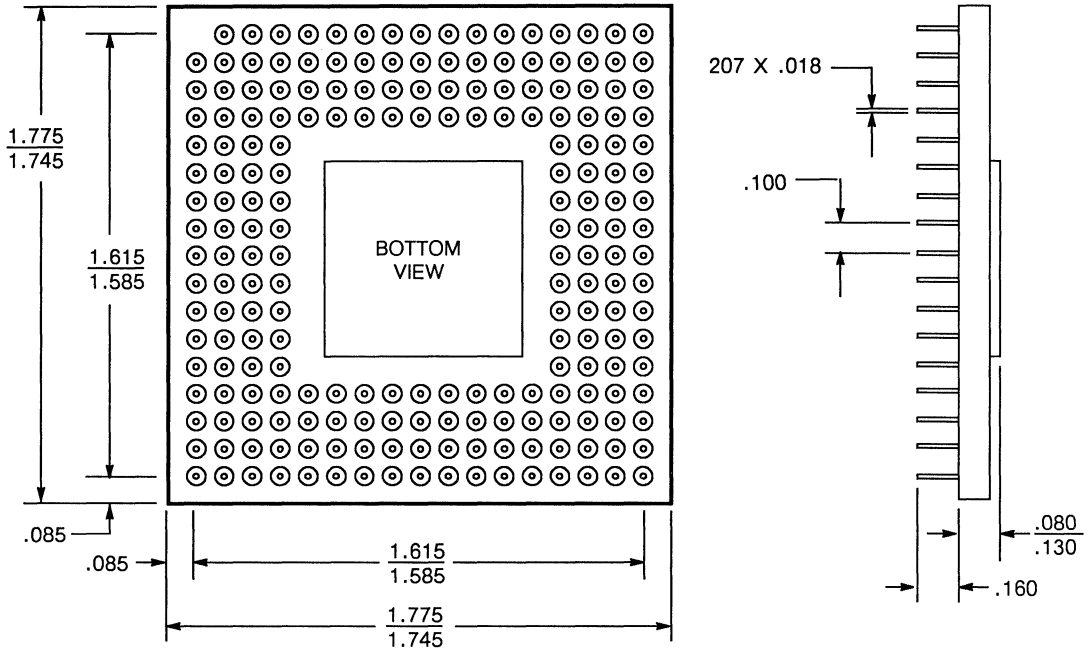


Figure 11–2. Package Outline

Clock Frequency (MHz)	Odering Code	Package Type	Operating Range
25	CY7C601–25GC	G208	Commercial
33	CY7C601–33GC	G208	Commercial
25	CY7C601–25PC	P208	Commercial
33	CY7C601–33PC	P208	Commercial
25	CY7C601–25GMB	G208	Military

Table 11–3. Ordering Information

CYPRESS SEMICONDUCTOR

CHAPTER 12

Programming Considerations

This section of the Users Manual will be published in the Second Edition, January 1989.

CHAPTER 13

Development Environment

This section of the Users Manual will be published in the Second Edition, January 1989.

APPENDIX A
Assembly Language Syntax

This appendix supports *Appendix B, Instruction Descriptions*. Every instruction description in *Appendix B* includes a table that describes the suggested assembly language format for that instruction. This appendix describes the notation used in the assembly language syntax descriptions.

Understanding the use of type fonts is crucial to understanding the syntax descriptions in *Appendix B*. Items in `typewriter font` are literals, to be entered exactly as they appear. Items in *italic font* are metasymbols which are to be replaced by numeric or symbolic values when actual SPARC assembly-language code is written. For example, "*asi*" would be replaced by a number in the range of 0 to 255 (the value of the asi bits in the binary instruction), or by a symbol which had been bound to such a number.

Subscripts on metasymbols further identify the placement of the operand in the generated binary instruction. For example, reg_{rs2} is a *reg* (i.e. register name) whose binary value will end up in the *rs2* field of the resulting instruction.

Register Names

reg

A *reg* is an Integer Unit register. It can have a value of:

%0 through %31	all integer registers
%g0 through %g7	global registers—same as %0 through %7
%o0 through %o7	out registers—same as %8 through %15
%l0 through %l7	local registers—same as %16 through %23
%i0 through %i7	in registers—same as %24 through %31

Subscripts further identify the placement of the operand in the binary instruction as one of:

reg_{rs1}	—*rs1* field
reg_{rs2}	—*rs2* field
reg_{rd}	—*rd* field

freg

A *freg* is a floating-point register. It can have a value from %f0 through %f31. Subscripts further identify the placement of the operand in the binary instruction as one of:

$freg_{rs1}$	—*rs*1 field
$freg_{rs2}$	—*rs*2 field
$freg_{rd}$	—*rd* field

creg

A *creg* is a coprocessor register. It can have a value from %c0 through %c31. Subscripts further identify the placement of the operand in the binary instruction as one of:

$creg_{rs1}$	—*rs*1 field
$creg_{rs2}$	—*rs*2 field
$creg_{rd}$	—*rd* field

Special Symbol Names

Certain special symbols need to be written exactly as they appear in the syntax table. These appear in `typewriter font`, and include a percent sign (`%`), also in typewriter font. The percent sign is part of the symbol name; it must appear as part of the literal value.

The symbol names are:

`%psr`	Processor State Register
`%wim`	Window Invalid Mask register
`%tbr`	Trap Base Register
`%y`	Y register
`%fsr`	Floating-point State Register
`%csr`	Coprocessor State Register
`%fq`	Floating-point Queue
`%cq`	Coprocessor Queue
`%hi`	Unary operator that extracts high 22 bits of its operand
`%lo`	Unary operation that extracts low 10 bits of its operand

Values

Some instructions use operands comprising values as follows:

simm13—A signed immediate constant that fits in 13 bits

const22—A constant that fits in 22 bits

asi—An alternate address space identifier (0 to 255)

Label

A sequence of characters, comprised of alphabetic letters (a–z, A–Z [upper and lower case distinct]), underscore (_), dollar sign ($), period (.), and decimal digits (0–9), which does not begin with a decimal digit.

Some instructions offer a choice of operands. These are grouped as follows:

regaddr:

reg_{rs1}
$reg_{rs1} + reg_{rs2}$

address:

- reg_{rs1}
- $reg_{rs1} + reg_{rs2}$
- $reg_{rs1} + simm13$
- $reg_{rs1} - simm13$
- $simm13$
- $simm13 + reg_{rs1}$

reg_or_imm

- reg_{rs2}
- $simm13$

APPENDIX B
Instruction Definitions

This appendix describes the SPARC architecture's instruction set. A more detailed, algorithmic definition of the instruction set appears in *Appendix C.*

Related instructions are grouped into subsections. Each subsection consists of five parts:

- A **table** of the opcodes defined in the subsection with the values of the field(s) which uniquely identify the instruction(s).
- An illustration of the applicable instruction **format(s)**.
- A table of the suggested **assembly language syntax.** (The syntax notation is described in Appendix A.)
- A **description** of the salient features, restrictions, and trap conditions.
- A list of the synchronous or floating-point/coprocessor traps which can occur as a consequence of executing the instruction(s).

This section does not include any timing information (in either cycles or absolute time) since timing is strictly implementation-dependent.

The following table lists all the instructions:

Opcode	Name
LDSB (LDSBA*)	Load Signed Byte (from Alternate space)
LDSH (LDSHA*)	Load Signed Halfword (from Alternate space)
LDUB (LDUBA*)	Load Unsigned Byte (from Alternate space)
LDUH (LDUHA*)	Load Unsigned Halfword (from Alternate space)
LD (LDA*)	Load Word (from Alternate space)
LDD (LDDA*)	Load Doubleword (from Alternate space)
LDF	Load Floating-point
LDDF	Load Double Floating-point
LDFSR	Load Floating-point State Register
LDC	Load Coprocessor
LDDC	Load Double Coprocessor
LDCSR	Load Coprocessor State Register
STB (STBA*)	Store Byte (into Alternate space)
STH (STHA*)	Store Halfword (into Alternate space)
ST (STA*)	Store Word (into Alternate space)
STD (STDA*)	Store Doubleword (into Alternate space)
STF	Store Floating-point
STDF	Store Double Floating-point
STFSR	Store Floating-point State Register
STDFQ*	Store Double Floating-point Queue

*privileged instruction

Table B–1. Instruction Set

Opcode	Name
STC	Store Coprocessor
STDC	Store Double Coprocessor
STCSR	Store Coprocessor State Register
STDCQ*	StoreDouble Coprocessor Queue
LDSTUB (LDSTUBA*)	Atomic Load-Store Unsigned Byte (in Alternate space)
SWAP (SWAPA*)	Swap r Register with Memory (in Alternate space)
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
AND (ANDcc)	And (and modify icc)
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)
SLL	Shift Left Logical
SRL	Shift Right Logical
SRA	Shift Right Arithmetic
SETHI	Set High 22 bits of *r register*
SAVE	Save caller's window
RESTORE	Restore caller's window
Bicc	Branch on integer condition codes
FBfcc	Branch on floating-point condition codes
CBccc	Branch on coprocessor condition codes
CALL	Call
JMPL	Jump and Link
RETT*	Return from Trap
Ticc	Trap on integer condition codes
RDY	Read Y register
RDPSR*	Read Processor State Register
RDWIM*	Read Window Invalid Mask register
RDTBR*	Read Trap Base Register
WRY	Write Y register
WRPSR*	Write Processor State Register
WRWIM*	Write Window Invalid Mask register
WRTBR*	Write Trap Base Register
UNIMP	Unimplemented instruction
IFLUSH	Instruction cache Flush
FPop	Floating-point Operate: FiTO(s,d,x), F(s,d,x)TOi, FsTOd, FsTOx, FdTOs, FdTOx, FxTOs, FxTOd, FMOVs, FNEGs, FABSs, FSQRT(s,d,x), FADD(s,d,x), FSUB(s,d,x), FMUL(s,d,x), FDIV(s,d,x), FCMP(s,d,x), FCMPE(s,d,x)
CPop	Coprocessor operate

*privileged instruction

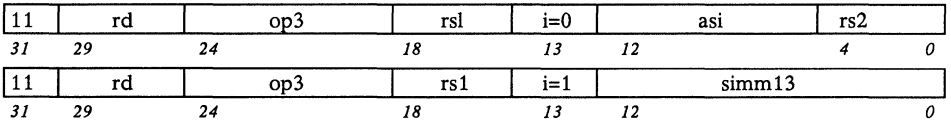
Table B–1. InstructionSet (continued)

Load Integer Instructions

Opcode	op3	Operation
LDSB	001001	Load Signed Byte
LDSBA*	011001	Load Signed Byte from Alternate space
LDSH	001010	Load Signed Halfword
LDSHA*	011010	Load Signed Halfword from Alternate space
LDUB	000001	Load Unsigned Byte
LDUBA*	010001	Load Unsigned Byte from Alternate space
LDUH	000010	Load Unsigned Halfword
LDUHA*	010010	Load Unsigned Halfword from Alternate space
LD	000000	Load Word
LDA*	010000	Load Word from Alternate space
LDD	000011	Load Doubleword
LDDA*	010011	Load Doubleword from Alternate space

*privileged instruction

Format (3):



Assembly Language Syntax	
ldsb	*[address], reg_{rd}*
ldsba	*[regaddr] asi, reg_{rd}*
ldsh	*[address], reg_{rd}*
ldsha	*[regaddr] asi, reg_{rd}*
ldub	*[address], reg_{rd}*
lduba	*[regaddr] asi, reg_{rd}*
lduh	*[address], reg_{rd}*
lduha	*[regaddr] asi, reg_{rd}*
ld	*[address], reg_{rd}*
lda	*[regaddr] asi, reg_{rd}*
ldd	*[address], reg_{rd}*
ldda	*[regaddr] asi, reg_{rd}*

Description

The load single integer instructions move either a byte, halfword, or word from memory into the *r register* defined by the *rd* field. A fetched byte or halfword is right-justified in *rd* and may be either zero-filled or sign-extended.

The load double integer instructions (LDD, LDDA) move a doubleword 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 at the effective memory address + 4 is moved into the odd *r register*. The least significant bit of the *rd* field is ignored. (Note that a load double with rd = 0 modifies only r[1].)

The effective address for a load instruction is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one. Instructions which load from an alternate address space must have zero in the *i* field and the address space identifier to be used for the

load in the *asi* field. Otherwise the address space indicates either a user or system data space access, according to the S bit of the PSR.

LD and LDA cause a mem_address_not_aligned trap if the effective address is not word-aligned; LDUH, LDSH, LDUHA, and LDSHA trap if the address is not halfword-aligned; and LDD and LDDA trap if the address is not doubleword-aligned.

If a load single instruction traps, the destination register remains unchanged.

If a load double instruction is trapped with a data access exception during the effective address memory access, the destination registers remain unchanged. However a specific implementation might cause a data_access_exception trap during the effective address + 4 memory access, but not during the effective address access. Thus, the *even* destination *r register* can be changed in this case. (Note that this cannot happen across a page boundary because of the doubleword-alignment restriction.)

Implementation Note

On effective address +4 accesses, the system should limit data_access_exceptions to non-restartable errors, such as uncorrectable memory errors.

Programming Note

The execution time of a load integer instruction may increase if the next instruction uses the register specified by the *rd* field of the load instruction as a source operand (*rs1* or *rs2*). In the case of load doubleword instructions, this applies to both destination registers. Whether the time increase occurs or not is implementation-dependent.

Programming Note

When *i* = 1 and *rs1* = 0, any location in the lowest or highest 4K bytes of an address space can be accessed without using a register.

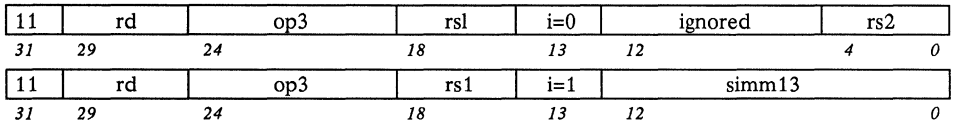
Traps

illegal_instruction (load alternate space with i = 1)
privileged_instruction (load alternate space only)
mem_address_not_aligned (excluding LDSB, LDSBA, LDUB, and LDUBA)
data_access_exception

Load Floating-Point Instructions

opcode	op3	operation
LDF	100000	Load Floating-point register
LDDF	100011	Load Double Floating-point register
LDFSR	100001	Load Floating-point State Register

Format (3):



Assembly Language Syntax	
ld	*[address], freg$_{rd}$*
ldd	*[regaddr], freg$_{rd}$*
ld	*[address],* %fsr

Description

The load single floating-point instruction (LDF) moves a word from memory into the *f register* identified by the *rd* field.

The load double floating-point instruction (LDDF) moves a doubleword from memory into an *f register pair*. The most significant word at the effective memory address is moved into the even *f register*. The least significant word at the effective memory address +4 is moved into the odd *f register*. The least significant bit of the *rd* field is ignored.

The load floating-point state register instruction (LDFSR) waits for all FPops that have not finished execution to complete and then loads a word from memory into the FSR.

The effective address for the load instruction is either "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or r[*rs1*]+ sign_ext(simm13)" if the *i* field is one.

LDF and LDFSR cause a mem_address_not_aligned trap if the effective address is not word-aligned; and LDDF traps if the address is not doubleword-aligned. A load floating-point instruction causes an fp_disabled trap if the EF field of the PSR is 0 or if no FPU is present.

If a load single floating-point instruction is trapped with a data access exception, the destination *f register* either remains unchanged or is set to an implementation-defined constant value.

If a load double floating-point instruction is trapped with a data access exception, either the destination *f registers* remain unchanged or one or both are set to an implementation-defined constant value.

Programming Note

The execution time of a load floating-point instruction may increase if the next instruction uses the register specified by the *rd* field of the load instruction as a source operand (*rs1* or *rs2*). In the case of load double floating-point instructions, this applies to both destination registers. Whether the time increases or not is implementation-dependent.

Programming Note

When *i* = 1 and *rs1* = 0, any location in the lowest or highest 4K bytes of an address space can be accessed without using a register.

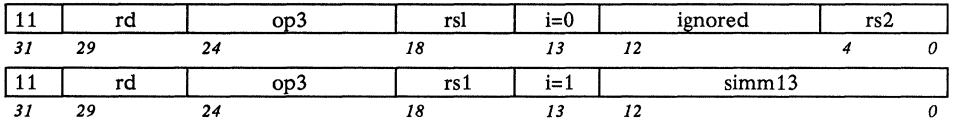
Traps

fp_disabled
fp_exception
mem_address_not_aligned
data_access_exception

Load Coprocessor Instructions

opcode	op3	operation
LDC	110000	Load Coprocessor register
LDDC	110011	Load Double Coprocessor register
LDCSR	110001	Load Coprocessor State Register

Format (3):



Assembly Language Syntax	
ld	*[address], creg$_{rd}$*
ldd	*[address], creg$_{rd}$*
ld	*[address],* %csr

Description

The load single coprocessor instruction (LDC) moves a word from memory into a coprocessor register. The load double coprocessor instruction (LDDC) moves a doubleword from memory into a coprocessor register pair. The load coprocessor state register instruction (LDCSR) moves a word from memory into the Coprocessor State Register. The semantics of these instructions depend on the implementation of the attached coprocessor.

The effective address for the load instruction is either "r[*rs1*] + r[*rs2*]" if the *i* field is zero, or "r[*rs1*] + sign_ext(simm13)" if the *i* field is one.

LDC and LDCSR cause a mem_address_not_aligned trap if the effective address is not word-aligned; and LDDC traps if the address is not doubleword-aligned. A load coprocessor instruction causes a cp_disabled trap if the EC field of the PSR is 0 or if no coprocessor is present.

If a load coprocessor instruction traps, the state of the coprocessor depends on its implementation.

Implementation Note

On effective address +4 accesses, the system should limit data_access_exceptions to non-restartable errors, such as uncorrectable memory errors.

Programming Note

The execution time of a load coprocessor instruction may increase if the next instruction uses the register specified by the *rd* field of the load instruction as a source operand (*rs1* or *rs2*). In the case of load double coprocessor instructions, this applies to both destination registers. Whether the time increases or not is implementation-dependent.

Programming Note

When *i* = 1 and *rs1* = 0, any location in the lowest or highest 4K bytes of an address space can be accessed without using a register.

Traps

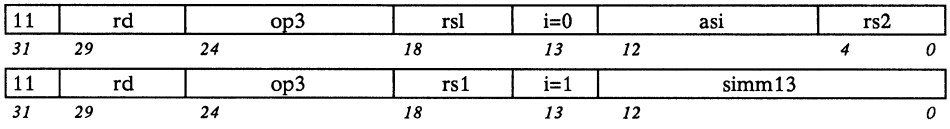
cp_disabled
cp_exception
mem_address_not_aligned
data_access_exception

Store Integer Instructions

opcode	op3	operation
STB	000101	Store Byte
STBA*	010101	Store Byte into Alternate space
STH	000110	Store Halfword
STHA*	010110	Store Halfword into Alternate space
ST	000100	Store Word
STA*	010100	Store Word into Alternate space
STD	000111	Store Doubleword
STDA*	010111	Store Doubleword into Alternate space

*privileged instruction

Format (3):



Assembly Language Syntax		
stb	*reg$_{rd}$, [address]*	*synonyms:*stub, stsb
stba	*reg$_{rd}$, [regaddr] asi*	*synonyms:*stuba, stsba
sth	*reg$_{rd}$, [address]*	*synonyms:*stuh, stsba
stha	*reg$_{rd}$, [regaddr] asi*	*synonyms:*stuha, stsha
st	*reg$_{rd}$, [address]*	
sta	*reg$_{rd}$, [regaddr] asi*	
std	*reg$_{rd}$, [address]*	
stda	*reg$_{rd}$, [regaddr] asi*	

Description

The store single integer instructions move the word, the least significant halfword, or the least significant byte from the *r register* specified by the *rd* field into memory.

The store double integer instructions (STD, STA) move a doubleword from an *r register pair* into memory. The most significant word in the even *r register* is written into memory at the effective address and the least significant word in the following odd *r register* is written into memory at the effective address +4.

The effective address for a store instruction is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one. Instructions which store to an alternate address space must have zero in the *i* field and the address space identifier to be used for the store in the

asi field. Otherwise the address space indicates either a user or system data space access, according to the S bit in the PSR.

ST and STA cause a mem_address_not_aligned trap if the effective address is not word-aligned; STH and STHA trap if the address is not halfword-aligned; and STD and STDA trap if the address is not doubleword-aligned.

If a store single instruction traps, memory remains unchanged. However, in the case of a store double, an implementation might cause a data_access_exception trap during the effective address +4 memory access, but not during the effective address access. Thus, data at the effective memory address can be changed in this case. (Note that this cannot happen across a page boundary because of the doubleword-alignment restriction.)

Implementation Note

On effective address +4 accesses, the system should limit data_access_exceptions to non-restartable errors, such as uncorrectable memory errors.

Programming Note

When *i* = 1 and *rs1* = 0, any location in the lowest or highest 4K bytes of memory can be written without using a register.

Traps

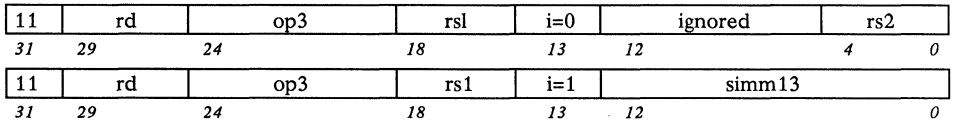
illegal_instruction (store alternate with i = 1)
privileged_instruction (store alternate only)
mem_address_not_aligned (excluding STB and STBA)
data_access_exception

Store Floating-Point Instructions

opcode	op3	operation
STF	100100	Store Floating-point
STDF	100111	Store Double Floating-point
STFSR	100101	Store Floating-point State Register
STDFQ*	100110	Store Double Floating-point Queue

*privileged instruction

Format (3):



Assembly Language Syntax	
st	*freg$_{rd}$, [address]*
std	*freg$_{rd}$, [address]*
st	%fsr, *[address]*
std	%fq, *[address]*

Description

The store single floating-point instruction (STF) moves the contents of the *f register* specified by the *rd* field into memory.

The store double floating-point instruction (STDF) moves a doubleword from an *f register pair* into memory. The most significant word in the even *f register* is written into memory at the effective address and the least significant word in the odd *f register* is written into memory at the effective address +4.

The store floating-point queue instruction (STDFQ) stores the front entry of the Floating-point Queue (FQ) into memory. The address part of the front entry is stored into memory at the effective address, and the instruction part of the front entry at the effective address +4. If the FPU is in exception_mode, the queue is then advanced to the next entry, or it becomes empty (as indicated by the *qne* bit in the FSR).

The store floating-point state register instruction (STFSR) waits for all FPops that have not finished execution to complete and then writes the FSR into memory.

The effective address for a store instruction is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one.

STF and STFSR cause a mem_address_not_aligned trap if the address is not word-aligned and STDF and STDFQ trap if the address is not doubleword-aligned. A store floating-point instruction causes an fp_disabled trap if the EF field of the PSR is 0 or if the FPU is not present.

If a store single floating-point instruction traps, memory remains unchanged. However, in the case of a store double, an implementation may cause a data_access_exception trap during the effective address + 4 memory access, but not during the effective address access. Data at the effective memory address can be changed in this case. (Note that this cannot happen across a page boundary because of the doubleword-alignment restriction.)

Implementation Note

On effective address +4 accesses, the system should limit data_access_exceptions to non-restartable errors, such as uncorrectable memory errors.

Traps

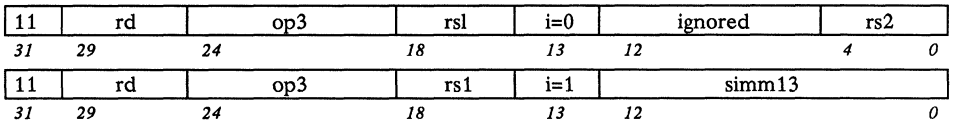
fp_disabled
fp_exception
privileged_instruction (STDFQ only)
mem_address_not_aligned
data_access_exception

Store Coprocessor Instructions

opcode	op3	operation
STC	110100	Store Coprocessor
STDC	110111	Store Double Coprocessor
STCSR	110101	Store Coprocessor State Register
STDCQ*	110110	Store Double Coprocessor Queue

*privileged instruction

Format (3):



Assembly Language Syntax	
st	$creg_{rd}$, *[address]*
std	$creg_{rd}$, *[address]*
st	%csr, *[address]*
std	%cq, *[address]*

Description

The store single coprocessor instruction (STC) moves the contents of a coprocessor register into memory. The store double coprocessor instruction (STDC) moves the contents of a coprocessor register pair into memory. The store coprocessor state register instruction (STCSR) moves the contents of the coprocessor state register into memory. The store double coprocessor queue instruction (STDCQ) moves the front entry of the coprocessor queue into memory. The semantics of these instructions depend on the implementation of the attached coprocessor, if any.

The effective address for a store instruction is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one.

STC and STCSR cause a mem_address_not_aligned trap if the address is not word-aligned and STDC and STDCQ trap if the address is not doubleword-aligned. A store coprocessor instruction causes a cp_disabled trap if the EC field of the PSR is 0 or if no coprocessor is present.

If a store single coprocessor instruction traps, memory remains unchanged. However, in the case of a store double, an implementation might cause a data_access_exception trap during the effective address +4 memory access, but not during the effective address access. Thus, data at the effective memory address can be changed in this case. (Note that this cannot happen across a page boundary because of the doubleword-alignment restriction.)

Implementation Note

On effective address +4 accesses, the system should limit data_access_exceptions to non-restartable errors, such as uncorrectable memory errors.

Traps

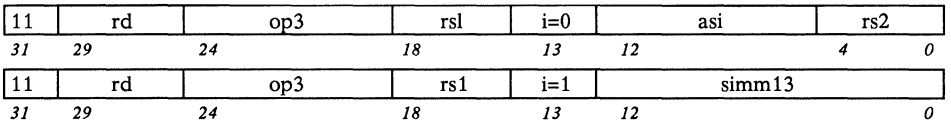
cp_disabled
cp_exception
privileged_instruction (STDCQ only)
mem_address_not_aligned
data_access_exception

Atomic Load-Store Unsigned Byte Instructions

opcode	op3	operation
LDSTUB	001101	Atomic Load-Store Unsigned Byte
LDSTUBA*	011101	Atomic Load-Store Unsigned Byte into Alternate space

*privileged instruction

Format (3):



Assembly Language Syntax	
ldstub	*[address], reg$_{rd}$*
ldstuba	*[regaddr] asi, reg$_{rd}$*

Description

The atomic load-store instructions move a byte from memory into an *r register* identified by the *rd* field and then rewrite the same byte in memory to all ones without allowing intervening asynchronous traps. In a multiprocessor system, two or more processors executing atomic load-store instructions addressing the same byte simultaneously are guaranteed to execute them in some serial order.

The effective address of an atomic load-store is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one. LDSTUBA must have zero in the *i* field, or an illegal_instruction trap occurs. The address space identifier used for the memory accesses is taken from the *asi* field. For LDSTUB, the address space indicates either a user or system data space access, according to the S bit in the PSR.

If an atomic load-store instruction traps, memory remains unchanged. However, an implementation may cause a data_access_exception trap during the store memory access, but not during the load access. In this case, the destination register can be changed.

Implementation Note

The system should limit data_access_exceptions on the store access to non-restartable errors, such as protection violation or uncorrectable memory errors.

Programming Note

When *i* = 1 and *rs1* = 0, any location in the lowest or highest 4K bytes of memory can be accessed without using a register.

Traps

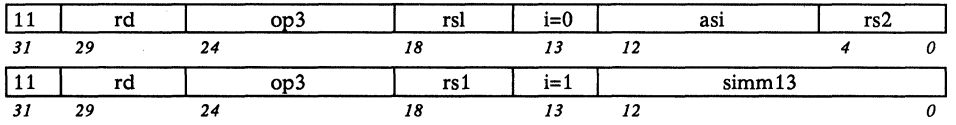
illegal_instruction (LDSTUBA with i = 1 only)
privileged_instruction (LDSTUBA only)
data_access_exception

SWAP r Register with Memory

opcode	op3	operation
SWAP	001111	SWAP *r register* with memory
SWAPA*	011111	SWAP *r register* with Alternate space memory

*privileged instruction

Format (3):



Assembly Language Syntax	
swap	[*source*], reg_{rd}
swapa	[*regsource*] asi, reg_{rd}

Description

The swap instructions exchange the *r register* identified by the *rd* field with the contents of the addressed memory location. This is performed atomically without allowing asynchronous traps. In a multiprocessor system, two or more processors issuing swap instructions simultaneously are guaranteed to get results corresponding to the executing the instructions serially, in some order.

The effective address of the swap instruction is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one. SWAPA must have zero in the *i* field or an illegal_instruction trap occurs. The address space identifier used for the memory accesses is taken from the *asi* field. For SWAP, the address space indicates either a user or a system data space access, according to the S bit in the PSR.

These instructions cause a mem_address_not_aligned trap if the effective address is not word-aligned.

If a swap instruction traps, memory remains unchanged.

Programming Note

When *i* = 1 and *rs1* = 0, any location in the lowest or highest 4K bytes of memory can be written without using a register.

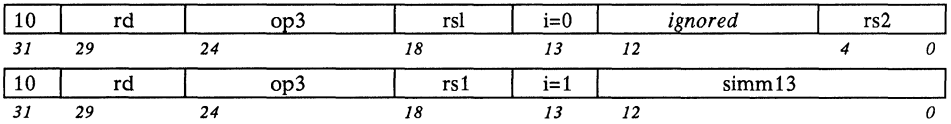
Traps

illegal instruction (i = 1 and SWAPA only)
privileged_instruction (SWAPA only)
data_access_exception

Add Instructions

opcode	op3	operation
ADD	000000	Add
ADDcc	010000	Add and modify icc
ADDX	001000	Add with Carry
ADDXcc	011000	Add with Carry and modify icc

Format (3):



Assembly Language Syntax	
add	*reg$_{rsl}$, reg_or_imm, reg$_{rd}$*
addcc	*reg$_{rsl}$, reg_or_imm, reg$_{rd}$*
addx	*reg$_{rsl}$, reg_or_imm, reg$_{rd}$*
addxcc	*reg$_{rsl}$, reg_or_imm, reg$_{rd}$*

Description

ADD and ADDcc compute either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one, and place the result in the *r register* specified in the *rd* field.

ADDX and ADDXcc add the PSR's carry (c) bit also; that is, they compute "r[rs1] + r[rs2] + c" or "r[rs1] + sign_ext(simm13) + c" and place the result in the *r register* specified in the *rd* field.

ADDcc and ADDXcc modify **all** the integer condition codes.

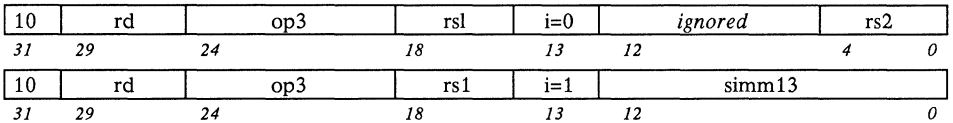
Traps

(none)

Tagged Add Instructions

opcode	op3	operation
TADDcc	100000	Tagged Add and modify icc
TADDccTV	100010	Tagged Add, modify icc and Trap on Overflow

Format (3):



Assembly Language Syntax	
taddcc	*reg$_{rsl}$, reg_or_imm, reg$_{rd}$*
taddcctv	*reg$_{rsl}$, reg_or_imm, reg$_{rd}$*

Description

These instructions compute either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one. An overflow condition exists if bit 1 or bit 0 of either operand is not zero, or if the addition generates an arithmetic overflow.

If a TADDccTV causes an overflow condition, a tag_overflow trap is generated and the destination register and condition codes remain unchanged. If a 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.

If a TADDcc causes an overflow condition, the overflow bit *(v)* of the PSR is set; if it does not cause an overflow, it 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.

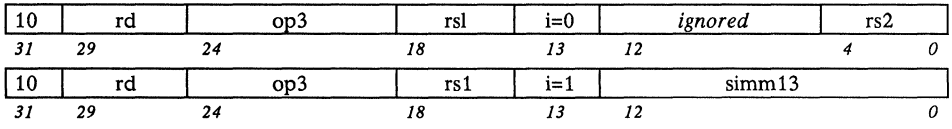
Traps

tag_overflow (TADDccTV only)

Subtract Instructions

opcode	op3	operation
ADD	000000	Add
ADDcc	010000	Add and modify icc
ADDX	001000	Add with Carry
ADDXcc	011000	Add with Carry and modify icc

Format (3):



Assembly Language Syntax	
sub	*reg$_{rsl}$, reg_or_imm, reg$_{rd}$*
subcc	*reg$_{rsl}$, reg_or_imm, reg$_{rd}$*
subx	*reg$_{rsl}$, reg_or_imm, reg$_{rd}$*
subxcc	*reg$_{rsl}$, reg_or_imm, reg$_{rd}$*

Description

These instructions compute either "r[rs1] - r[rs2]" if the *i* field is zero, or r[rs1] - sign_ext(simm13)" if the *i* field is one, and place the result in the *r register* specified in the *rd* field.

SUBX and SUBXcc ("SUBtract eXtended") also subtract the PSR's carry *(c)* bit; that is, they compute "r[rs1] - r[rs2] - c" or "r[rs1] - sign_ext(simm13) - *c*" and place the result in the *r register* specified in the *rd* field.

SUBcc and SUBXcc modify **all** the integer condition codes.

Programming Note

A SUBcc with *rd* = 0 can be used for signed and unsigned integer compare.

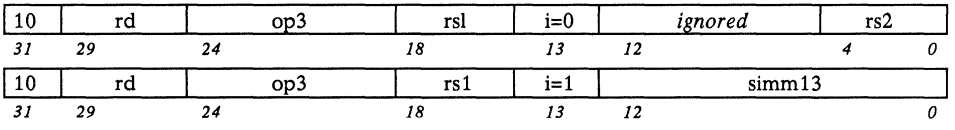
Traps

(none)

Tagged Subtract Instructions

opcode	op3	operation
TSUBcc	100001	Tagged Subtract and modify icc
TSUBccTV	100011	Tagged Subtract, modify icc and Trap on Overflow

Format (3):



Assembly Language Syntax
tsubcc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*
tsubcctv *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description

These instructions compute either "r[rs1] - r[rs2]" if the *i* field is zero, or "r[rs1] - sign_ext(simm13)" if the *i* field is one. An overflow condition exists if bit 1 or bit 0 of either operand is not zero, or if the subtraction generates an arithmetic overflow.

If a TSUBccTV causes an overflow condition, a tag_overflow trap is generated and the destination register and condition codes remain unchanged. If a TSUBccTV does not cause an overflow condition, the integer condition codes are updated (in particular, the overflow bit *(v)* is set to 0) and the result of the traction is written into the *r register* specified by the *rd* field.

If a TSUBcc causes an overflow condition, the overflow bit *(v)* of the PSR is set; if it does not cause an overflow, it is cleared. In either case, the remaining integer condition codes are also updated and the result of the subtraction is written into the *r register* specified by the *rd* field.

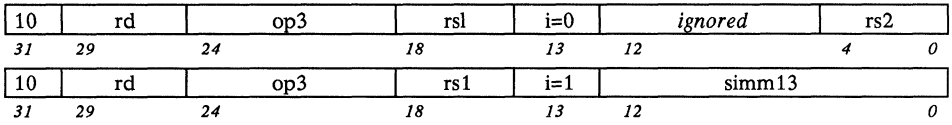
Traps

tag_overflow (TSUBccTV only)

Multiply Step Instruction

opcode	op3	operation
MULScc	100100	Multiply Step and modify icc

Format (3):



Assembly Language Syntax	
mulscc	reg_{rsl}, *reg_or_imm*, reg_{rd}

Description

The multiply step instruction can be used to generate the 64-bit product of two signed or unsigned words (See Appendix E). MULScc works as follows:

1. 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) is computed.
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(simm13)" 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).

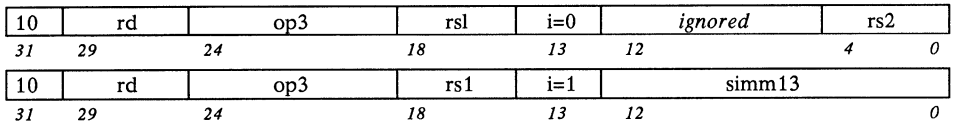
Traps

(none)

Logical Instructions

opcode	op3	operation
AND	000001	And
ANDcc	010001	And and modify icc
ANDN	000101	And Not
ANDNcc	010101	And Not and modify icc
OR	000010	Inclusive Or
ORcc	010010	Inclusive Or and modify icc
ORN	000110	Inclusive Or Not
ORNcc	010110	Inclusive Or Not and modify icc
XOR	010011	Exclusive Or and modify icc
XNOR	000111	Exclusive Nor
XNOR	010111	Exclusive Nor and modify icc

Format (3):



Assembly Language Syntax	
and	reg_{rsl}, *reg_or_imm*, reg_{rd}
andcc	reg_{rsl}, *reg_or_imm*, reg_{rd}
andn	reg_{rsl}, *reg_or_imm*, reg_{rd}
andncc	reg_{rsl}, *reg_or_imm*, reg_{rd}
or	reg_{rsl}, *reg_or_imm*, reg_{rd}
orcc	reg_{rsl}, *reg_or_imm*, reg_{rd}
orn	reg_{rsl}, *reg_or_imm*, reg_{rd}
orncc	reg_{rsl}, *reg_or_imm*, reg_{rd}
xor	reg_{rsl}, *reg_or_imm*, reg_{rd}
xorcc	reg_{rsl}, *reg_or_imm*, reg_{rd}
xnor	reg_{rsl}, *reg_or_imm*, reg_{rd}
xnorcc	reg_{rsl}, reg_or_imm, reg_{rd}

Description

These instructions implement the bitwise logical operations. They compute either "r[rs1] **op** r[rs2]" if the *i* field is zero, or "r[rs1] **op** sign_ext(simm13)" if the *i* field is one (**op** = and, and not, or, or not, xor, xnor).

ANDcc, ANDNcc, ORcc, ORNcc, XORcc and XNORcc modify **all** the integer condition codes as described in the chapter *Registers*.

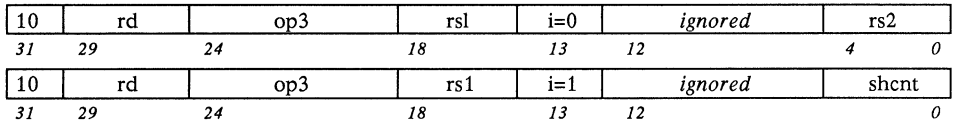
Traps

(none)

Shift Instructions

opcode	op3	operation
SLL	100101	Shift Left Logical
SRL	100110	Shift Right Logical
SRA	100111	Shift Right Arithmetic

Format (3):



Assembly Language Syntax	
sll	reg_{rsl}, *reg_or_imm*, reg_{rd}
srl	reg_{rsl}, *reg_or_imm*, reg_{rd}
sra	reg_{rsl}, *reg_or_imm*, reg_{rd}

Description

The shift count for these instructions is the least significant five bits of either "[rs2]" if the *i* field is zero, or "simm13" if the *i* field is one. (The least significant five bits of "simm13" is called "shcnt" in the above format.)

SLL shifts the value of "r[rs1]" left by the number of bits implied by the shift count.

SRL and SRA shift the value of "r[rs1]" right by the number of bits implied by the shift count.

SLL and SRL replace vacated positions with zeroes, whereas SRA fills vacated positions with the most significant bit of "r[rs1]." No shift occurs when the shift count is zero.

All of these instructions place the shifted result in the *r register* specified in the *rd* field.

These instructions do **not** modify the condition codes.

Programming Note

"Arithmetic left shift by 1 (and calculate overflow)" can be implemented with an ADDcc instruction.

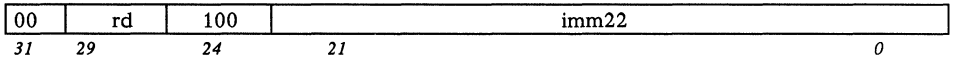
Traps

(none)

SETHI Instruction

opcode	op	op2	operation
SETHI	00	100	Set High

Format (2):



Assembly Language Syntax	
sethi	*const22*, reg_{rd}
sethi	%hi *(value)*, reg_{rd}

Description

SETHI zeroes the least significant 10 bits of "r[rd]" and replaces its high-order 22 bits with *imm22*.

The condition codes are not affected.

Programming Note

It is suggested that sethi 0, %0 be used as the preferred NOP, since it will not cause an increase in execution time if it follows a load instruction.

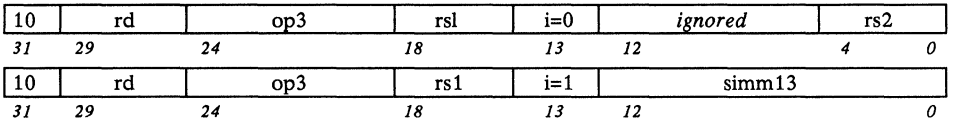
Traps

(none)

SAVE and RESTORE Instructions

opcode	op3	operation
SAVE	111100	Save caller's window
RESTORE	111101	Restore caller's window

Format (3):



Assembly Language Syntax	
save	reg_{rsl}, *reg_or_imm*, reg_{rd}
restore	reg_{rsl}, *reg_or_imm*, reg_{rd}

Description

The SAVE instruction subtracts one from the CWP (modulo the number of implemented windows) and compares this value, the "new_CWP," against the Window Invalid Mask (WIM) register. If the WIM bit corresponding to the new_CWP is set, "(WIM and 2^{new_CWP}) = 1," then a window_overflow trap is generated. If the WIM bit corresponding to the new_CWP is reset, then a window_overflow trap is not generated and new_CWP is written into CWP. This causes the **active** window to become the **previous** window, thereby saving the caller's window.

The RESTORE instruction adds one to the CWP (modulo the number of implemented windows) and compares this value, the "new_CWP," against the Window Invalid Mask (WIM) register. If the WIM bit corresponding to the new_CWP is set, "(WIM and 2^{new_CWP}) = 1," then a window_underflow trap is generated. If the WIM bit corresponding to the new_CWP is reset, then a window_underflow trap is not generated and new_CWP is written into CWP. This causes the **previous** window to become the **active** window, thereby restoring the caller's window.

Furthermore, if an overflow or underflow trap is **not** generated, SAVE and RESTORE behave like normal ADD instructions, except that the operands "r[rs1]" or "r[rs2]" **are** read from the **old** window (i.e., the window addressed by the original CWP) and the result is written into "r[rd]" of the **new** window (i.e., the window addressed by new_CWP).

Note that CWP arithmetic is performed modulo the number of implemented windows (NWINDOWS).

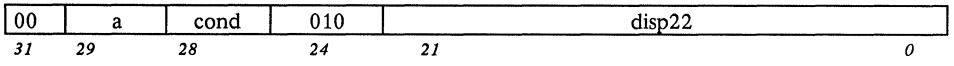
Traps

window_overflow (SAVE only)
window_underflow (RESTORE only)

Branch on Integer Condition 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 (C or 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

Format (2):



Assembly Language Syntax		
`ba{,a}`	*label*	
`bn{,a}`	*label*	
`bne{,a}`	*label*	*synonym:* `bnz`
`be{,a}`	*label*	*synonym:* `bz`
`bg{,a}`	*label*	
`ble{,a}`	*label*	
`bge{,a}`	*label*	
`bl{,a}`	*label*	
`bgu{,a}`	*label*	
`bleu{,a}`	*label*	
`bcc{,a}`	*label*	*synonym:* `bgeu`
`bcs{,a}`	*label*	*synonym:* `blu`
`bpos{,a}`	*label*	
`bneg{,a}`	*label*	
`bvc{,a}`	*label*	
`bvs{,a}`	*label*	

NOTE: *To set the "annul" bit for Bicc instructions, append an (optional)* `",a"` *to the opcode. For example, use "*`bgu,a`* label". The preceding table indicates that the* `,a"` *is optional by enclosing it in braces ({}).*

Description

A Bicc instruction (except BA and BN) evaluates the integer condition codes *(icc)* according to the *cond* field. If the condition codes evaluate to true the branch is taken and the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 * sign_ext (disp22))." If the condition codes evaluate to false, the branch is not taken. If the branch is not taken and the *a*

(annul) field is set, the delay instruction is not executed (annulled). If the branch is taken, the annul field is ignored. (Annulment, delay instructions, and delayed control transfers are described further in the chapter *Instructions*.)

BN (Branch Never) acts like a "NOP." except that, if the annul field is one, the delay instruction is not executed (annulled). If the annul field is zero, the delay instruction is executed.

BA (Branch Always) causes a transfer of control, irrespective of the value of the condition code bits. If the annul field is one, the delay instruction is not executed (annulled). If the annul field is zero, the delay instruction is executed.

NOTE: *Except for BA, all Bicc instructions with a=1 annul the delay instruction when the branch is not taken. However, BA with a=1 does the reverse: it annuls the delay instruction even though the branch is taken.*

NOTE: *The delay instruction of a Bicc, other than a BA, should not be a delayed control-transfer instruction.*

Programming Note

An untaken branch takes as much or more time than a taken branch. The additional time it takes is implementation-dependent.

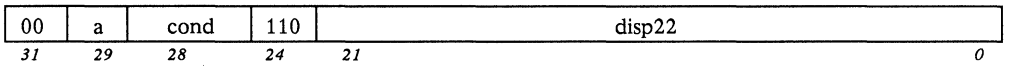
Traps

(none)

Floating-Point Branch on Condition Instructions

opcode	cond	operation	fcc test
FBA	1000	Branch Always	1
FBN	0000	Branch Never	0
FBU	0111	Branch on Unordered	U
FBG	0110	Branch on Greater	G
FBUG	0101	Branch on Unordered or Greater	G or U
FBL	0100	Branch on Less	L
FBUL	0011	Branch on Unordered or Less	L or U
FBLG	0010	Branch on Less or Greater	L or G
FBNE	0001	Branch on Not Equal	L or G or U
FBE	1001	Branch on Equal	E
FBUE	1010	Branch on Unordered or Equal	E or U
FBGE	1011	Branch on Greater or Equal	E or G
FBUGE	1100	Branch on Unordered or Greater or Equal	E or G or U
FBLE	1101	Branch on Less or Equal	E or L
FBULE	1110	Branch on Unordered or Less or Equal	E or L or U
FBO	1111	Branch on Ordered	E or L or G

Format (2):



Assembly Language Syntax		
`fba{,a}`	*label*	
`fbn{,a}`	*label*	
`fbu{,a}`	*label*	
`fbg{,a}`	*label*	
`fbug{,a}`	*label*	
`fbl{,a}`	*label*	
`fbul{,a}`	*label*	
`fblg{,a}`	*label*	
`fbne{,a}`	*label*	*synonym:* `fbnz`
`fbe{,a}`	*label*	*synonym:* `fbz`
`fbue{,a}`	*label*	
`fbge{,a}`	*label*	
`fbuge{,a}`	*label*	
`fble{,a}`	*label*	
`fbule{,a}`	*label*	
`fbo{,a}`	*label*	

NOTE: *To set the "annul" bit for FBfcc instructions, append an (optional) ",a" to the opcode. For example, use "*`fbl,a`* *label". The preceding table indicates that the* `",a"` *is optional by enclosing it in braces ({}).*

Description

An FBfcc instruction (except FBA and FBN) evaluates the floating-point condition codes *(fcc)* according to the *cond* field. If the condition codes evaluate to true the branch is taken and the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 * sign_ext (disp22))." If the condition codes evaluate to false, the branch is not taken. If the branch is not taken and the *a* (annul) field is set, the delay instruction is not executed (annulled).If the branch is taken, the annul field is ignored and the delay instruction is executed. (Annulment, delay instructions, and delayed control transfers are described further in the chapter *Instructions*.)

FBN (Branch Never) acts like a "NOP", except that if the annul field is one, the delay instruction is not executed (annulled). If the annul field is zero, the delay instruction is executed.

FBA (Branch Always) causes a transfer of control, irrespective of the value of the condition code bits. If the annul field is one, the delay instruction is not executed (annulled). If the annul field is zero, the delay instruction is executed.

An FBfcc instruction generates an fp_disabled trap (and does not branch on annul) if the PSR's EF bit is reset or if the FPU is not present.

NOTE: *Except for FBA, all FBfcc instructions with a=1 annul the delay instruction when the branch is not taken. However, FBA with a=1 does the reverse: it annuls the delay instruction even though the branch is taken.*

NOTE: *The instruction executed immediately before an FBfcc must not be a floating-point instruction.*

Programming Note

An untaken branch takes as much or more time than a taken branch. The additional time it takes is implementation-dependent.

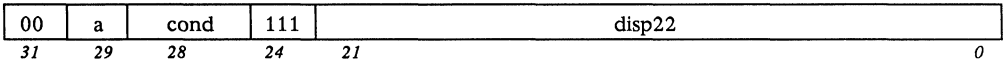
Traps

fp_disabled
fp_exception

Coprocessor Branch on Condition Instructions

opcode	cond	bp_CP_cc[1:0] test
CBA	1000	Always
CBN	0000	Never
CB3	0111	3
CB2	0110	2
CB23	0101	2 or 3
CB1	0100	1
CB13	0011	1 or 3
CB12	0010	1 or 2
CB123	0001	1 or 2 or 3
CB0	1001	0
CB03	1010	0 or 3
CB02	1011	0 or 2
CB023	1100	0 or 2 or 3
CB01	1101	0 or 1
CB013	1110	0 or 1 or 3
CB012	1111	0 or 1 or 2

Format (2):



Assembly Language Syntax	
`cba{,a}`	*label*
`cbn{,a}`	*label*
`cb3{,a}`	*label*
`cb2{,a}`	*label*
`cb23{,a}`	*label*
`cb1{,a}`	*label*
`cb13{,a}`	*label*
`cb12{,a}`	*label*
`cb123{,a}`	*label*
`cb0{,a}`	*label*
`cb03{,a}`	*label*
`cb02{,a}`	*label*
`cb023{,a}`	*label*
`cb01{,a}`	*label*
`cb013{,a}`	*label*
`cb012{,a}`	*label*

NOTE: *To set the "annul" bit for CBccc instructions, append an (optional) ",a" to the opcode. For example, use "*`cb12,a`* *label". The preceding table indicates that the "*`,a`*" is optional by enclosing it in braces ({}).*

Description

A CBccc instruction (except CBA and CBN) evaluates the coprocessor condition codes (supplied by the coprocessor on bp_CP_cc[1:0]) according to the *cond* field. If the condition codes evaluate to true the branch is taken and the instruction causes a PC-relative, delayed control transfer to the address "PC + (4 * sign_ext (disp22))." If the condition codes evaluate to false, the branch is not taken and the instruction acts like a "NOP."

If the branch is not taken and the *a* (annul) field is set, the delay instruction is not executed (annulled). If the branch is taken, the annul field is ignored and the delay instruction is executed. (Annulment, delay instructions, and delayed control transfers are described further in the chapter *Instructions*.)

CBN (Branch Never) acts like a "NOP", except that if the annul field is one, the delay instruction is not executed (annulled). If the annul field is zero, the delay instruction is executed.

CBA (Branch Always) causes a transfer of control, irrespective of the value of the condition code bits. If the annul field is one, the delay instruction is not executed (annulled). If the annul field is zero, the delay instruction is executed.

A CBccc instruction generates a cp_disabled trap (and does not branch or annul) if the PSR's EC bit is reset or if no coprocessor is present.

NOTE: *Except for CBA, all CBccc instructions with a=1 annul the delay instruction when the branch is not taken. However, CBA with a=1 does the reverse: it annuls the delay instruction even though the branch is taken.*

NOTE: *A CBccc instruction must be immediately preceded by a non-coprocessor instruction.*

Programming Note

An untaken branch takes as much or more time than a taken branch. The additional time it takes is implementation-dependent.

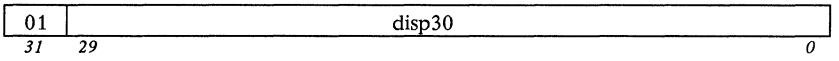
Traps

cp_disabled
cp_exception

CALL Instruction

opcode	op	operation
CALL	01	Call

Format (1):



Assembly Language Syntax	
`call`	*label*

Description

The CALL instruction causes an unconditional, delayed, PC-relative control transfer to address "PC + (4 * disp30)". Since the word displacement (*disp30*) 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 *out* register r[15].

The PC-relative displacement is formed by appending two low-order zeros to the instruction's 30-bit word displacement field.

Programming Note

A JMPL instruction with *rd* = 15 can be used as a register-indirect CALL.

Programming Note

The execution time of a CALL instruction may increase if the next instruction uses r[15] as a source operand. Whether this happens is implementation-dependent.

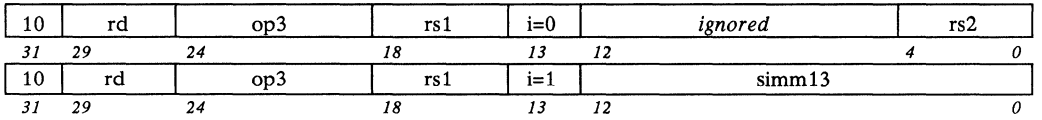
Traps

(none)

Jump and Link Instruction

opcode	op3	operation
JMPL	111000	Jump and Link

Format (3):



Assembly Language Syntax	
jmpl	*address*, reg_{rd}

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] + sign_ext(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 the *rd* field.

If either of the low-order two bits of the jump address is nonzero, a mem_address_not_aligned trap occurs.

Programming Note

JMPL with *rd* = 0 can be used to return from a subroutine. The typical return address is "r[31]+8", if the subroutine was entered by a CALL instruction.

Programming Note

JMPL with *rd* = 15 can be used as a register-indirect CALL.

Programming Note

The execution time of a JMPL instruction may increase if the next instruction uses r[rd] as a source operand. Whether this happens is implementation-dependent.

Traps

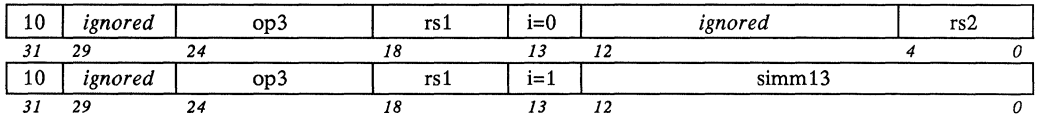
mem_address_not_aligned

Return from Trap Instruction

opcode	op3	operation
RETT*	111001	Return from Trap

*privileged instruction

Format (3):



Assembly Language Syntax	
`rett`	*address*

Description

The RETT instruction adds one to the CWP (modulo the number of implemented windows) and compares this value, the "new_CWP," against the Window Invalid Mask (WIM) register. If the WIM bit indexed by the new_CWP is set, "(WIM and 2^{new_CWP}) = 1," then a window_underflow trap is generated. If the WIM bit indexed by the new_CWP is reset, then a window_underflow trap is not generated and new_CWP is written into CWP. This causes the *previous* window to become the *active* window, thereby restoring the window that existed at the time of the trap.

If a window_underflow trap is not generated, RETT causes a delayed control transfer to the target address. The target 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. Furthermore, RETT restores the S field of the PSR from the PS field, and sets the ET field to one.

If traps are enabled (ET=1), an illegal_instruction trap occurs. If traps are disabled (ET=0) and the processor is not in supervisor mode (S=0), or if a window_underflow condition is detected, or if either of the low-order two bits of the target address is nonzero, a reset trap occurs. If a reset trap occurs, the *tt* field of the TBR encodes the trap condition: privileged_instruction, window_underflow, or mem_address_not_aligned.

NOTE: *The instruction executed immediately before a RETT must be a JMPL instruction. (See discussion in the chapter "Instructions".)*

Programming Note

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

```
jmpl        %l7, %0          ! old PC
rett        %l8              ! old nPC
```

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

```
jmpl        %l8, %0          ! old nPC
rett        %l8 + 4          ! old nPC + 4
```

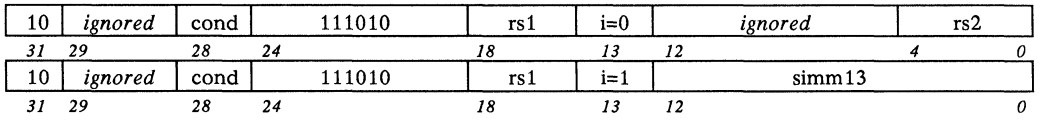
Traps

illegal_instruction
reset (privileged_instruction)
reset (mem_address_not_aligned)
reset (window_underflow)

Trap on Integer Condition Instruction

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 (C or 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

Format (3):



Assembly Language Syntax		
ta	*address*	
tn	*address*	
tne	*address*	*synonym:* tnz
te	*address*	*synonym:* tz
tg	*address*	
tle	*address*	
tge	*address*	
tl	*address*	
tgu	*address*	
tleu	*address*	
tcc	*address*	*synonym:* tgeu
tcs	*address*	*synonym:* tlu
tpos	*address*	
tneg	*address*	
tvc	*address*	
tvs	*address*	

Description

A Ticc instruction evaluates the integer condition codes (*icc*) according to the *cond* field. If the condition codes evaluate to true and there are no higher priority traps pending, then a trap_instruction trap is generated. If the condition codes evaluate to false, a trap_instruction trap does not occur.

If a trap_instruction trap is generated, the *tt* field of the Trap Base Register (TBR) is 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.

See the chapter *Traps, Exceptions and Error Handling* for the complete definition of a trap.

Traps

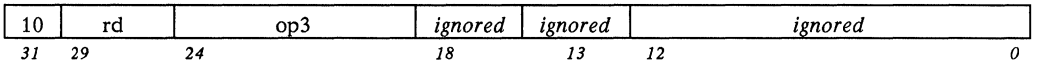
trap_instruction

Read State Register Instructions

opcode	op3	operation
RDY	101000	Read Y register
RDPSR*	101001	Read Processor State Register
RDWIM*	101010	Read Window Invalid Mask register
RDTBR*	101011	Read Trap Base Register

*privileged instruction

Format (3):



Assembly Language Syntax	
rd	%y, reg_{rd}
rd	%psr, reg_{rd}
rd	%wim, reg_{rd}
rd	%tbr, reg_{rd}

Description

These instructions read the specified IU state registers into the *r* register specified in the *rd* field.

Programming Note

The execution time of any of these instructions may increase if the next instruction uses the register specified by the *rd* field of this instruction as a source operand. Whether it does or not is implementation-dependent.

Traps

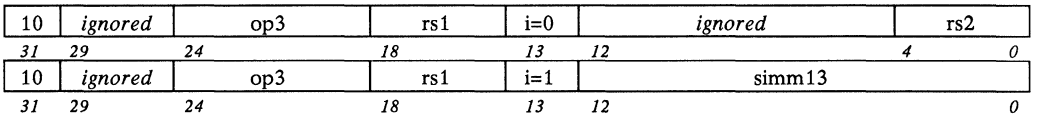
privileged_instruction (RDPSR, RDWIM and RDTBR only)

Write State Register Instructions

opcode	op3	operation
WRY	110000	Write Y register
WRPSR*	110001	Write Processor State Register
WRWIM*	110010	Write Window Invalid Mask register
WRTBR*	110011	Write Trap Base Register

*privileged instruction

Format (3):



Assembly Language Syntax	
wr	*reg$_{rs1}$, reg_or_imm,* %y
wr	*reg$_{rs1}$, reg_or_imm,* %psr
wr	*reg$_{rs1}$, reg_or_imm,* %wim
wr	*reg$_{rs1}$, reg_or_imm,* %tbr

Description

These instructions write either "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 writeable subfields of the specified IU state register.

WRPSR does not write the PSR and causes an illegal_instruction trap if the result would cause the CWP field of the PSR to point to an unimplemented window.

These instructions are *delayed-write* instructions:

1. If any of the three instructions after a WRPSR uses any field of the PSR that is changed by the WRPSR, the value of that field is unpredictable. (Note that any instruction which references a non-global register implicitly uses the CWP.)
2. If a WRPSR instruction is updating the PSR's PIL to a new value and is simultaneously setting ET to 1, this can result in an interrupt trap at a level equal to the old value of the PIL.

Programming Note

Two WRPSR instructions should be used when enabling traps and changing the value of the PIL. The first WRPSR should specify ET=0 with the new PIL value, and the second WRPSR should specify ET=1 and the new PIL value.

3. If any of the three instructions after a WRWIM is a SAVE, RESTORE or RETT, the occurrence of window_overflow and window_underflow traps is unpredictable.
4. If any of the three instructions that follow a WRY is a MULScc or RDY, the value of Y used is unpredictable.
5. If any of the three instructions that follow a WRTBR causes a trap, the trap base address (TBA) used may be either the old or the new value.
6. If any of the three instructions after a write state register instruction reads the modified state register, the value read is unpredictable.
7. If any of the three instructions after a write state register instruction is trapped, a subsequent read state register instruction in the trap handler will get the register's new value.

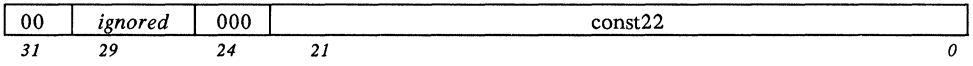
Traps

privileged_instruction (WRPSR, WRWIM and WRTBR only)
illegal_instruction (WRPSR only)

Unimplemented Instruction

opcode	op	op2	operation
UNIMP	00	000	Unimplemented

Format (2):



Assembly Language Syntax	
unimp	*const22*

Description

The UNIMP instruction causes an illegal_instruction trap. The const22 value is ignored.

Programming Note

This instruction can be used as part of the protocol for calling a function that is expected to return an aggregate value, such as a C-language structure. See *Appendix D* for an example.

- An UNIMP instruction is placed after (not in) the delay slot after the CALL instruction in the calling function.
- If the callee function is expecting to return a structure, it will find the size of the structure that the caller expects to be returned as the *const22* operand of the UNIMP instruction. The callee can check the opcode to make sure it is indeed UNIMP.

- If the function is not going to return a structure, upon returning it attempts to execute the UNIMP instruction rather than skipping over it as it should. This causes the program to terminate. This behavior adds some run-time type checking to an interface that cannot be checked properly at compile time.

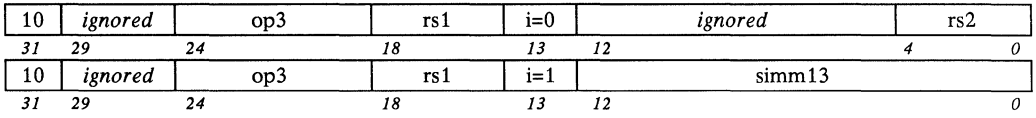
Traps

illegal_instruction

Instruction Cache Flush Instruction

opcode	op3	operation
IFLUSH	111011	Instruction cache Flush

Format (3):



Assembly Language Syntax
`iflush` *address*

Description

The IFLUSH instruction causes a word to be flushed from an instruction cache that may be internal to the processor. The address of the word to be flushed is either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign_ext(simm13)" if the *i* field is one.

Implementation Note:

If there is no instruction cache internal to the processor, IFLUSH acts as a "NOP." If there is an internal instruction cache, IFLUSH flushes the addressed word from the cache. If there is an external instruction cache, IFLUSH causes an illegal_instruction trap. The presence of an external instruction cache is determined by the `bp_I_cache_present` signal.

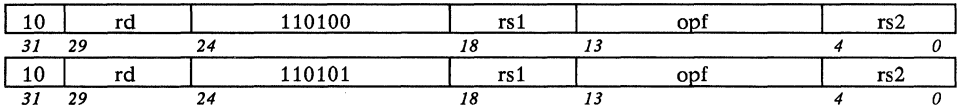
Traps

illegal_instruction

Floating-Point Operate (FPop) Instructions

opcode	op3	operation
FPop1	110100	Floating-point operate
FPop2	110101	Floating-point operate

Format (3):



The Floating-Point Operate (FPop) instructions are encoded using two type 3 instruction formats called FPop1 and FPop2. The floating-point operations themselves are encoded by the *opf* field. (Note that the load/store floating-point instructions are not "FPop" instructions.)

All FPop instructions take all operands from and return all results to *f registers* and/or the FSR. They perform operations on ANSI/IEEE 754-1985 single, double, and extended formats.

NOTE: *All FPops which deal with extended format data are not currently implemented in floating-point hardware and are emulated in software. Extended data FPops currently generate a floating point exception trap when executed..*

All multiple-precision floating-point instructions (including load/store floating-point) assume that operands are located in register *pairs* (for double precision) or *quadruples* (for extended precision). The following table indicates the alignment assumptions. Note that single-precision operands can be in any *f register*.

Operand	f Register Address
Double-e	0 mod 2
Double-f	1 mod 2
Extended-e	0 mod 4
Extended-f	1 mod 4
Extended-f-low	2 mod 4
Extended-u	3 mod 4

Table B–2. Arrangement of Double and Extended Operands in f registers

According to this convention, the least significant bit of an *f register* address is ignored by double-precision FPops and the least significant two bits of an *f register* address are ignored by extended-precision FPops.

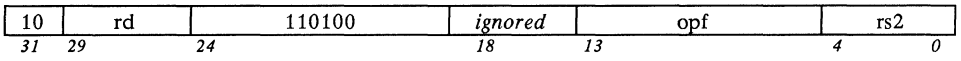
A program including floating-point computations generates the same results as if all instructions were executed sequentially (assuming it runs to completion). Note that floating-point loads and stores are not floating-point operate instructions.

Results are written (or traps are caused) in the order that FPops are encountered in the instruction stream. The section *Instructions* in chapter 2 explains this in more detail. An FPop instruction causes an fp_disabled trap if the EF field of the PSR is 0 or if no FPU is present.

Convert Integer to Floating-Point Instructions

opcode	opf	operation
FiTOs	011000100	Convert Integer to Single
FiTOd	011001000	Convert Integer to Double
FiTOx	011001100	Convert Integer to Extended

Format (3):



Assembly Language Syntax	
fitos	$freg_{rs2}$, $freg_{rd}$
fitod	$freg_{rs2}$, $freg_{rd}$
fitox	$freg_{rs2}$, $freg_{rd}$

Description

These instructions convert the 32-bit integer argument in the *f register* specified by rs2 into a floating-point number in the destination format according to the ANSI/IEEE 754-1985 specification. They place the result in the destination *f register(s)* specified by rd.

For FiTOs and FiTOx with single-precision rounding, rounding is performed according to the rounding direction (RD) field of the FSR.

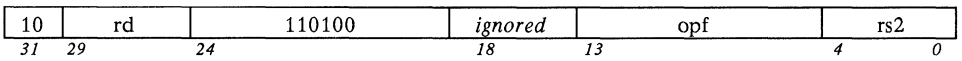
Traps

fp_disabled
fp_exception (NX) (FiTOs and FiTOx when RP=single)

Convert Floating-Point to Integer

opcode	opf	operation
FsTOi	011010001	Convert Single to Integer
FdTOi	011010010	Convert Double to Integer
FxTOi	011010011	Convert Extended to Integer

Format (3):



Assembly Language Syntax	
fstoi	$freg_{rs2}$, $freg_{rd}$
fdtoi	$freq_{rs2}$, $freg_{rd}$
fxtoi	$freg_{rs2}$, $freg_{rd}$

Description

These instructions convert the floating-point source argument in the *f register* or *f registers* specified by rs2 to a 32-bit integer (in the *f register* specified by the *rd* field) according to the ANSI/IEEE 754-1985 specification.

The floating-point argument is rounded toward zero and the *rd* field of the FSR is ignored.

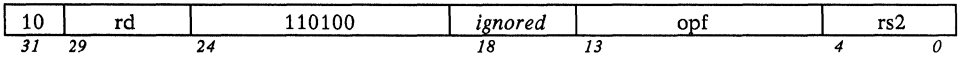
Traps

fp_disabled
fp_exception (NV, NX)

Convert Between Floating-Point Formats Instructions

opcode	opf	operation
FsTOd	011001001	Convert Single to Double
FsTOx	011001101	Convert Single to Extended
FdTOs	011000110	Convert Double to Single
FdTOx	011001110	Convert Double to Extended
FxTOs	011000111	Convert Extended to Single
FxTOd	011001011	Convert Extended to Double

Format (3):



Assembly Language Syntax	
`fstod`	*fregrs2, fregrd*
`fstox`	*fregrs2, fregrd*
`fdtox`	*fregrs2, fregrd*
`fdtox`	*fregrs2, fregrd*
`fxtod`	*fregrs2, fregrd*
`fxtos`	*fregrs2, fregrd*

Description

These instructions convert the floating-point source argument in the *f register* or *f registers* specified by rs2 to a floating-point number in the destination format according to the ANSI/IEEE 754-1985 specification. They place the result in the *f register* or *f registers* specified by rd.

Rounding is performed according to the rounding direction (RD) field of the FSR. In the case of FdTOx, the outcome is also a function of the rounding precision (RP) field.

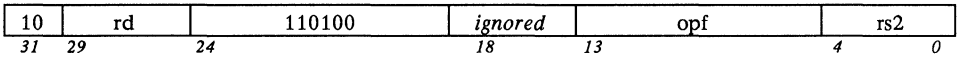
Traps

fp_disabled
fp_exception (OF, UF, NV, NX)

Floating-Point Move Instructions

opcode	opf	operation
FMOVs	000000001	Move
FNEGs	000000101	Negate
FABSs	000001001	Absolute Value

Format (3):



Assembly Language Syntax	
fmovs	$freg_{rs2}$, $freg_{rd}$
fnegs	$freg_{rs2}$, $freg_{rd}$
fabss	$freg_{rs2}$, $freg_{rd}$

Description

FMOVs moves a word from f[rs2] to f[rd]. Multiple FMOVs's are required to transfer a multiple-precision number between *f registers.*

FNEGs complements the sign bit, and FABs clears it.

These instructions do not round.

Programming Note

FNEGs or FABSs instructions can also operate on the high-order words (the word that contains the sign bit) of *double* and *extended* operands. Thus an FNEGs instruction and an FMOVs instruction would be used to negate a *double* and put the results in a different pair of *f registers.*

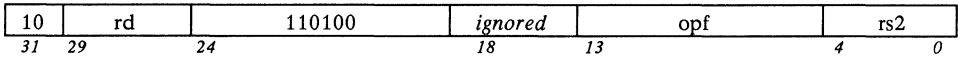
Traps

fp_disabled

Floating-Point Square Root Instructions

opcode	opf	operation
FSQRTs	000101001	Square Root Single
FSQRTd	000101010	Square Root Double
FSQRTx	000101011	Square Root Extended

Format (3):



Assembly Language Syntax	
fsqrts	$freg_{rs2}$, $freg_{rd}$
fsqrtd	$freg_{rs2}$, $freg_{rd}$
fsqrtx	$freg_{rs2}$, $freg_{rd}$

Description

These instructions generate the square root of the floating-point source argument in the *f register* or *f registers* specified by rs2 according to the ANSI/IEEE 754-1985 specification. They place the result in the destination *f register* or *f registers* specified by the *rd* field.

Rounding is performed according to the rounding direction (RD) field of the FSR. In the case of FSQRTx, the outcome is also a function of the rounding precision (RP) field.

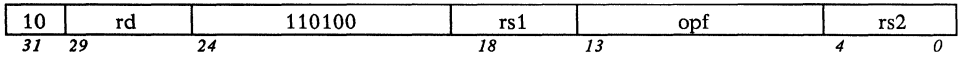
Traps

fp_disabled
fp_exception (NV, NX)

Floating-Point Add and Subtract Instructions

opcode	opf	operation
FADDs	001000001	Add Single
FADDd	001000010	Add Double
FADDx	001000011	Add Extended
FSUBs	001000101	Subtract Single
FSUBd	001000110	Subtract Double
FSUBx	001000111	Subtract Extended

Format (3):



Assembly Language Syntax	
fadds	*fregrs1, fregrs2, fregrd*
faddd	*fregrs1, fregrs2, fregrd*
faddx	*fregrs1, fregrs2, fregrd*
fsubs	*fregrs1, fregrs2, fregrd*
fsubd	*fregrs1, fregrs2, fregrd*
fsubx	*fregrs1, fregrs2, fregrd*

Description

These instructions add or subtract their operands according to the ANSI/IEEE 754-1985 specification, and place the result in the *f register* or *f registers* specified in the *rd* field. The subtract instructions subtract the floating-point value specified by *rs2* from the one specified by *rs1*.

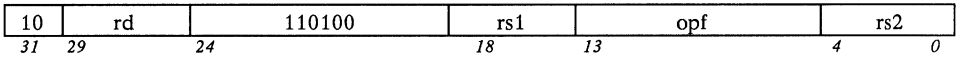
Traps

fp_disabled
fp_exception (OF, UF, NX)

Floating-Point Multiply and Divide Instructions

opcode	opf	operation
FMULs	001001001	Multiply Single
FMULd	001001010	Multiply Double
FMULx	001001011	Multiply Extended
FDIVs	001001101	Divide Single
FDIVd	001001110	Divide Double
FDIVx	001001111	Divide Extended

Format (3):



Assembly Language Syntax	
fmuls	*freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$
fmuld	*freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$
fmulx	*freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$
fdivs	*freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$
fdivd	*freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$
fdivx	*freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$

Description

These instructions multiply or divide their operands according to the ANSI/IEEE 754-1985 specification, and place the result in the *f register* or *f registers* specified in the *rd* field. The divide instructions divide the floating-point value specified by *rs1* by the one specified by *rs2*.

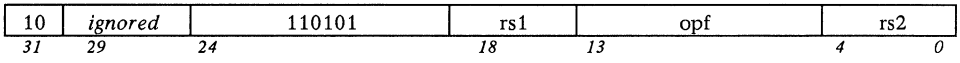
Traps

fp_disabled
fp_exception (OF, UF, DZ (FDIV only), NV, NX)

Floating-Point Compare Instructions

opcode	opf	operation
FCMPs	001010001	Compare Single
FCMPd	001010010	Compare Double
FCMPx	001010011	Compare Extended
FCMPEs	001010101	Compare Single and Exception if Unordered
FCMPEd	001010110	Compare Double and Exception if Unordered
FCMPEx	001010111	Compare Extended and Exception if Unordered

Format (3):



Assembly Language Syntax	
fcmps	*fregrs1, fregrs2*
fcmpd	*fregrs1, fregrs2*
fcmpx	*fregrs1, fregrs2*
fcmpes	*fregrs1, fregrs2*
fcmped	*fregrs1, fregrs2*
fcmpex	*fregrs1, fregrs2*

Description

These instructions compare their operands according to the ANSI/IEEE 754-1985 specification. The floating-point condition codes in the FSR are set as follows:

NOTE: *This table is a duplicate of Table 3–5 in the chapter "Registers".*

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

Table B–3. Floating-Point Condition Codes (fcc)

In this table, *fs1* refers to the value specified by the *rs1* field and *fs2* refers to the value specified by the *rs2* field of the compare instruction.

The "Compare and Cause Exception if Unordered" instructions (FCMPE) cause an invalid exception (NV) if either of the operands is a signaling or quiet NaN. FCMP also causes an invalid exception if either operand is a signaling NaN.

NOTE: *A non-floating point instruction must be executed between an FCMP and a subsequent FBfcc.*

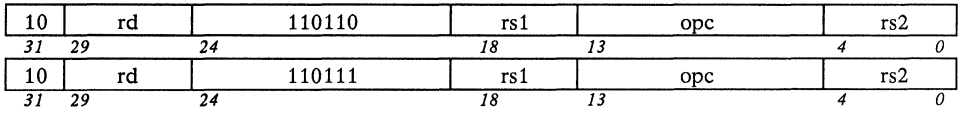
Traps

fp_disabled
fp_exception (NV)

Coprocessor Operate Instructions

opcode	op3	operation
CPop1	110110	Coprocessor Operate
CPop2	110111	Coprocessor Operate

Format (3):



NOTE: *The assembly language syntax for these instructions is unspecified.*

The Coprocessor Operate (CPop) instructions are encoded via two type 3 instruction formats called CPop1 and CPop2. The coprocessor operations themselves are encoded by the *opc* field and are coprocessor-dependent. (Note that the load/store coprocessor instructions are not "CPop" instructions.)

All CPop instructions take all operands from and return all results to coprocessor registers. The data types supported by the coprocessor are coprocessor-dependent. Operand alignment is coprocessor-dependent.

A CPop instruction causes a cp_disabled trap if the EC field of the PSR is 0 or if no coprocessor is present.

Whether a CPop generates a cp_exception trap is coprocessor-dependent.

APPENDIX C

ISP Descriptions

This appendix provides a description of the SPARC architecture using the Instruction-Set Processor (ISP) description language. It includes register definitions, instruction fields, processor states, instruction dispatch, traps, and instruction descriptions.

The instruction interpreter defines the ordering of events. Except for a few cases (which are documented), the interpreter together with the instruction and register definitions provide a supplemental description of the processor.

Note that the use of a particular variable in the notation does not necessarily imply that its related signal is present in an implementation, or visible to the programmer.

The instruction description language is a modified version of Bell and Newell's ISP instruction description language, which was created to accurately describe computer instruction sets. While the semantics are somewhat intuitive, the following guidelines provide important details:

- The only data type is the *bit vector*. Variables are defined as bit vectors of particular widths, declared as **variable<n:m>**. Variable subfields can be defined, also with the **<n:m>** notation. The value of a vector is a number in a base indicated by its subscript. The default base is decimal. Arrays of vectors are declared as **array[n:m]**.
- The notation ← indicates variable assignment, and := indicates a macro definition.
- When a bit vector is assigned to another of greater length, the operand is right-justified in the destination vector and the high-order positions are zero-filled. The macro **zero_extend** is sometimes used to make this clear. Conversely, the macro **sign_extend** causes the high-order positions of the result to be filled with the highest-order (sign) bit of its operand.
- The semicolon ';' separates statements. Parentheses '()' group statements and expressions that could otherwise be interpreted ambiguously.
- All statements are generally executed "simultaneously." However, if the term **next** appears, it indicates that the statement or statements which follow the **next** are executed after
  those that appear before the **next**. Thus, all statements between **next** phrases are executed concurrently. More precisely, this means that all expressions on the right hand sides of assignments located between **next's** are evaluated first, after which the variables on the left hand sides are updated. (This convention emulates synchronous, clocked hardware.)

For example, if A=0 and B=0, execution of the following two statements,

A ← B+1;

B ← A+1;

results in A=1 and B=1. However,

```
A ← B+1;
next;
B ← A+1;
```

results in A=1 and B=2.

- The symbol □ designates concatenation of vectors. A comma ',' on the left side of an assignment separates quantities that are concatenated for the purpose of assignment. For example, if the 2-bit vector T2 equals 3, and X, Y, and Z are 1-bit vectors, then:

  ```
  X, Y, Z ← 0□T2
  ```

  results in X=0, Y=1, and Z=1.

- The operators '+' and '–' perform two's complement arithmetic.
- The phrase **fork,** used only in the instruction interpreter for the FPop instructions, indicates that the associated routine may be executed concurrently with *all* other subsequent statements. There is no notation for rejoining: after the forked routine executes its last statement, it terminates.
- The major difference between the notation used here and the 1971 version of ISP is that the notation here uses the more common:

  ```
  if cond then S1 else S2
  ```

  whereas Bell and Newell used the following:

  ```
  (cond → S1, ¬ cond → S2)
  ```

- The macros memory_read and memory_write, are implementation-dependent. These routines define the interface without referring to implementation-specific signals:

  ```
  load_data ← memory_read(addr_space, address)
  memory_write(addr_space, address, byte_mask, store_data)
  ```

Memory_read returns the word in memory specified by both the address and the address space identifier.

Memory_write writes all or part of the word store_data into the word specified by the given address. If there is an exception, memory_write does not change the state of the external system or the MMU. Byte_mask is a 4-bit value that indicates which of the four bytes in store_data are to be written into the addressed word.

Register Definitions

```
PSR<31:0>;                          {Processor State Register}
    impl                    := PSR<31:28>;
    ver                     := PSR<27:24>;
    icc                     := PSR<23:20>;
        N                       := PSR<23>;
        Z                       := PSR<22>;
        V                       := PSR<21>;
        C                       := PSR<20>;
```

```
        reserved                    := PSR<19:14>;
        EC                          := PSR<13>;
        EF                          := PSR<12>;
        PIL                         := PSR<11:8>;
        S                           := PSR<7>;
        PS                          := PSR<6>;
        ET                          := PSR<5>;
        CWP                         := PSR<4:0>;

TBR<31:0>;                                  {Trap Base Register}
        TBA                         := TBR<31:12>;
        tt                          := TBR<11:4>;
        zero                        := TBR<3:0>;

FSR<31:0>;                                  {Floating-Point State Register}
        RD                          := FSR<31:30>;
        RP                          := FSR<29:28>;
        TEM                         := FSR<27:23>;
            NVM                         := FSR<27>;
            OFM                         := FSR<26>;
            UFM                         := FSR<25>;
            DZM                         := FSR<24>;
            NXM                         := FSR<23>;
            AU                          := FSR<22>;
        reserved                    := FSR<21:17>;
        ftt                         := FSR<16:14>;
        qne                         := FSR<13>;
        reserved                    := FSR<12>;
        fcc                         := FSR<11:10>;
        aexc                        := FSR<9:5>;
            nva     := FSR<9>;
            ofa     := FSR<8>;
            ufa     := FSR<7>;
            dza     := FSR<6>;
            nxa     := FSR<5>;
        cexc                        := FSR<4:0>;
            nvc     := FSR<4>;
            ofc     := FSR<3>;
            ufc     := FSR<2>;
            dzc     := FSR<1>;
            nxc     := FSR<0>;

CSR<31:0>;                                  {CP State Register}
WIM<31:0>;                                  {Window Invalid Mask Register}
Y<31:0>;                                    {Y Register}
PC<31:0>;                                   {Program Counter}
```

```
nPC<31:0>;                                  {Next Program Counter}
FQ<63:0>;                                   {Floating-Point Queue}
CQ<63:0>;                                   {Coprocessor Queue}
G[1:7]<31:0>;                               {Global Registers}
R[0:(16*NWINDOWS)-1]<31:0>;                 {Windowed Registers}
f[0:31]<31:0>;                              {Floating-Point Registers}

r[n] := if (n = 0)
      then 0
      else if (1 ≤ n ≤ 7)
          then G[n]                                        {globals}
          else R[(n-8) + (CWP*16)] ; {windowed registers}
```

System Interface Definitions

```
bp_IRL<3:0>;
bp_reset_in;
pb_error;
pb_retain_bus;
bp_FPU_present;
bp_CP_present;
bp_I_cache_present;
bp_CP_exception;
bp_CP_cc <1:0>;
bp_memory_exception;
```

Instruction Fields

The numbers in braces are the widths of the fields in bits.

```
instruction<31:0> ;

      op          {2}  := instruction<31:30>;
      op2         {3}  := instruction<24:22>;
      op3         {6}  := instruction<24:19>;
      opf         {9}  := instruction<13:5>;
      opc         {9}  := instruction<13:5>;
      asi         {8}  := instruction<12:5>;
      i           {1}  := instruction<13>;
      rd          {5}  := instruction<29:25>;
      a           {1}  := instruction<29>;
      cond        {4}  := instruction<28:25>;
      rs1         {5}  := instruction<18:14>;
      rs2         {5}  := instruction<4:0>;
      simm13     {13}  := instruction<12:0>;
      shcnt       {5}  := instruction<4:0>;
      disp30     {30}  := instruction<29:0>;
      disp22     {22}  := instruction<21:0>;
```

Processor States and Instruction Fetch

The IU can be in one of three states: execute_mode, reset_mode, or error_mode.

The FPU can be in one of five states: reset_mode, error_mode, fpu_execute_mode, fpu_exception_pending_mode, or fpu_exception_mode. The FPU's reset_mode and error_mode correspond to the IU's reset and error modes. The remaining FPU states are described in Section C.6.

The processor (that is the IU and FPU) is in reset_mode when bp_reset_in is asserted. The processor remains in reset_mode until bp_reset_in is deasserted, at which point the IU enters execute_mode and the FPU enters fpu_execute_mode.

When bp_reset_in is deasserted, the first instruction address is 0, with ASI=9 (supervisor instruction).

The processor enters error_mode from any state exept reset_mode if a synchronous trap is generated while traps are disabled. (See the chapter *Traps, Exceptions, and Error Handling*).

5.) The processor remains in error_mode until bp_reset_in is asserted, at which time it enters reset_mode.

Implementation Note

The external system should assert bp_reset_in whenever pb_error is detected.

The following ISP code defines the three IU states. In execute_mode, the IU fetches and dispatches instructions.

```
while (reset_mode) (
      if (bp_reset_in = 0) then (
           reset_mode ← 0;
           execute_mode ← 1;
           trap ← 1;
           reset ← 1
      )
);
addr_space := S=0 then 8 else 9;
while (execute_mode) (
      check_interrupts;              { see Section C.8}
      next;
      { the following code emulates the delayed nature of the write state register
        instructions.}
      PSR ← PSR′; PSR′ ← PSR′′; PSR′′ ← PSR′′′; PSR′′′ ← PSR′′′′;
      TBR ← TBR′; TBR′ ← TBR′′; TBR′′ ← TBR′′′; TBR′′′ ← TBR′′′′;
      WIM ← WIM′; WIM′ ← WIM′′; WIM′′ ← WIM′′′; WIM′′′ ← WIM′′′′;
      Y ← Y′; Y′ ← Y′′; Y′′ ← Y′′′; Y′′′ ← Y′′′′;
      next;
         if (trap = 1) then
                  execute_trap;                          { see Section C.8}
         next;
         instruction ← memory_read(addr_space, PC);
         next;
         if (bp_memory_exception = 1) then (
                  trap ← 1;
                  instruction_access_exception ← 1
```

```
    ) else (
            if (annul = 0) then (
                dispatch_instruction { see Section C.5 }
            ) else (
                annul ← 0;
                PC ← nPC;
                nPC ← nPC + 4
            )
    )
);
while (error_mode) (
    if (bp_reset_in = 1) then
            error mode ← 0
            reset mode ← 1
            pb_error ← 0
    )
);
```

Instruction Dispatch

The "dispatch_instruction" macro determines if the fetched instruction is an FPop or CPop. If it is an FPop, it is executed by the "execute_FPU_instruction" macro (Section C.6) as soon as the FPU can accept another instruction. If the fetched instruction is a CPop, it is executed by the "execute_CP_instruction" macro (Section C.7) as soon as the CP can accept another instruction.

If the instruction is neither an FPop or a CPop, it is executed by the "execute_IU_instruction" macro, which includes all the macro definitions in Section C.9 (except for FPop and CPop).

Unused bit patterns in the *op, op2, op3, opf,* and *i* fields of instructions cause illegal_instruction traps. Other fields that are defined to be *unused* are ignored and do not cause traps.

The macro 'floating-point_instr' returns a 1 if the instruction is a floating-point instruction. Similarly, the macro 'coprocessor_instr' returns a 1 if the instruction is a coprocessor instruction.

```
unimplemented_IU_instr := (
    if ( ( (op=00₂) and (op2=000₂) )            [UNIMP instruction}
            or
         ( ((op=11₂) or (op=10₂)) and (op3=unassigned) )
            or
         ( (i = 1) and
            (LDSBA or LDSHA or LDUBA or LDUHA or LDA or
         LDDA or STDA or LDSTUBA or SWAPA
            STBA or STHA or STA
            )
         ) then 1 else 0
```

```
);
floating_point_instr := (
      if (LDF or LDDF or LDFSR or
            STF or STDF or STFSR or STDFQ or
            FPop1 or FPop2 or FBfcc) then 1 else 0
);
coprocessor_instr: = (
      if (LDC or LDDC or LDCSR or
          STC or STDC or STCSR or STDCQ or CPop1 or CPop2 or CBccc) then 1 else 0
);
dispatch_instruction := (
      if (unimpl_IU_instr = 1) then (
            trap ← 1;
            illegal_instruction ← 1
      );
      if (floating-point_instr = 1) then (
            if (EF = 0) then (
                  trap ← 1;
                  fp_disabled ← 1
            ) else (
                  if ( fpu_exception_pending_mode = 1 ) then (
                        fpu_exception_pending_mode ← 0;
                        fpu_exception_mode ← 1;
                        trap ← 1
            );
            while ( (fp_not_ready = 1) and (trap = 0) )
                  check_interrupts;
      )
);
if (coprocessor_instr = 1) then (
      if (EC = 0) then (
            trap ← 1;
            cp_disabled ← 1
      ) else (
            check_CP_exception;
            next;
            while ( (cp_not_ready = 1) and (trap = 0) ) (
```

```
                    check_interrupts;
        )
);
next;
if (trap = 0) then
        if (FPop1 or FPop2) then fork execute_FPU_instruction
        else if (CPop1 or CPop2) then fork execute_CP_instruction
        else  execute_IU_instruction
        );
)
execute_IU_instruction := (
        { do routine for specific instruction, defined below }
        next;
        if (trap = 0 and
             not (CALL or RETT or JMPL or Bicc or FBfcc or CBccc or Ticc) ) then (
             PC ← nPC;
             nPC ← nPC + 4
        )
);
execute_FPU_instruction := (
        if (FPU_exception_mode)_ then (
             ftt <- sequence_error;
             FPU_exception_mode <- 0;              {see following discussion}
             FPU_exception_pending_mode <- 1
        ) else (
             enqueue_FQ(instruction, PC)
             { execute description defined below }
        )
);
```

Floating-Point Instruction Execution

The FPU can execute floating-point operate (FPop) instructions concurrently with other FPops and with non-floating-point instructions.To do this, it maintains a Floating-point Queue (FQ) of FPop instructions pending completion, and can force the IU to wait until resource and data dependencies have been resolved.

The architecture ensures that a program containing FPops generates the same numerical results as if there were no concurrency.

After the FPU begins to execute an FPop, the IU continues to fetch and execute instructions until one of five "hold" conditions occurs. Any one of these causes the IU to stop fetching instructions until the condition is no longer true:

1) If, for a load floating-point register instruction, the destination *f register* is the source or destination register of an executing FPop, the IU waits until the executing FPop no longer requires the register.
2) If, for a store floating-point register instruction, the source *f register* is the destination register of an executing FPop, the IU waits until executing FPop no longer require the register.
3) A load or store floating-point state register instruction (LDFSR, STFSR) causes the IU to wait until all executing and pending FPops have completed.
4) A branch on floating-point condition (FBfcc) instruction causes the IU to wait until any executing or pending floating-point compare instructions (FCMP, FCMPE) have finished.
5) When the IU encounters an FPop, it stops fetching instructions until the FPop has been accepted by the FPU.

Floating-Point Queue (FQ)

The floating-point queue (FQ) has at least one entry for each of the FPU's arithmetic units that can execute in parallel with other arithmetic units. The depth of the queue is implementation-dependent.

Each entry in the queue (for the purposes of the definition in this appendix) contains 1) the FPop instruction itself, 2) the PC from which the FPop was fetched, 3) an indication of the arithmetic unit executing it, 4) a completion status bit that indicates whether the operation finished properly, and 5) a temporary result, including any exceptions or condition codes generated by the instruction. Parts (1) and (2) of the front entry are visible to the programmer using the STDFQ instruction; the other parts and the other entries are invisible to the programmer.

(Note that load floating-point, store floating-point, and FBfcc instructions are never entered in the queue.)

For the purposes of the definition in this appendix, when an arithmetic unit finishes, it deposits its computed result, any exceptions or conditions it may have generated, and a completion status bit, into the reserved location in the queue. As FPops complete, each entry moves toward the front of the queue (if it is not already there).

The FPU can stop executing an FPop in one of four ways: 1) completed without exception (normal), 2) IEEE_exception, 3) unfinished_FPop, or 4) unimplemented_FPop. The following paragraphs describe each:

Normal Completion

If the FPop represented by the front entry in the queue caused no unmasked exceptions, the FPU 1) writes the result into the *f register(s)* specified by the *rd* field of the instruction (if any), 2) updates the FSR's *cexc* and *fcc* fields, 3) removes the entry from the queue, and 4) advances the queue.

IEEE_Exception

If the FPop pointed to by the front entry in the queue caused an IEEE_exception trap, the FPU updates the FSR's *cexc* and *ftt* fields to identify the exception, and does **not** write the result into

the *f register(s)* specified by the *rd* field of the instruction, **nor** does it remove the entry from the queue. However, if an IEEE_exception does not result in a fp_exception trap, all results are written, including the destination *f register, cexc, aexc,* and *fcc*.

Unimplemented_FPop or Unfinished_FPop

If the FPop pointed to by the front entry in the queue is not implemented, or if the arithmetic unit was unable to complete it according to the ANSI/IEEE 754–1985 specification (for example, a multiply unit may not be able to postnormalize a denormalized result or handle a NaN), the FPU updates the *ftt* field of the FSR to identify the exception, and does **not** write the result into the *f register(s)* specified by the *rd* field of the instruction, **nor** does it remove the entry from the queue. The front entry in the queue identifies the FPop that generated the floating-point exception trap.

FQ_Front_Done

The implementation-dependent macro 'FQ_front_done' returns a 1 if an arithmetic unit has finished processing the FPop at the front of the FQ. The implementation-dependent macro 'stop_FPU' stops all current processing of FQ entries.

FPU States

The FPU can be in any of three modes: FPU_execute_mode, FPU_exception_pending_mode, or FPU_exception_mode. In FPU_execute_mode, it executes floating-point instructions.

The FPU enters the FPU_exception_pending_mode state when an FPop instruction causes an IEEE_exception, unfinished_FPop exception, unimplemented_FPop exception, or a sequence_error. The FPU remains in FPU_exception_pending_mode until the IU fetches another floating-point instruction, at which time a fp_exception trap is caused and the FPU enters the FPU_exception_mode state.

In FPU_exception_mode, the FPU executes only store floating point instructions. If an FPop or a load floating point instruction is fetched while the unit is in FPU_exception_mode, the *ftt* field of the FSR will be updated to indicate "sequence_error", and the FPU will enter FPU_exception_pending_mode. The instruction that caused the sequence_error is not entered into the FQ.

The FPU returns to FPU_execute_mode after the FQ has been emptied via STDFQ instructions, that is, *qne* is 0.

```
while (FPU_execute_mode) (
      if (FQ_front_done = 1) then (
            if (fp_unimplemented = 1) then (    {not implemented}
                  fp_exception ← 1; ftt ← unimplemented_FPop;
            );
            if (FQ_c = 0) then (            {not finished}
                  fp_exception ← 1; ftt ← unfinished_FPop;
            ) else (                                {executed and finished}
                  cexc ← texc;
                  next;
```

```
                if ( cexc and TEM ≠ 0) then ( {floating-point trap}
                    fp_exception ← 1; ftt ← IEEE_Exception;
                ) else      {no floating-point trap}
                    aexc ← aexc or cexc;
                    if (FQ_single_result = 1) then
                        f[rd] ← result;
                    if (FQ_double_result = 1) then
                        f[rdE], f[rdO] ← result;
                    if (FQ_extended_result = 1) then
                        f[rdEE], f[rdEO], f[rdOE] ← result;
                    if (FQ_compare = 1) then
                        fcc ← tfcc;
                    equeue_FQ;
                )
            )
            next;
            if (fp_exception = 1) then (
                FPU_execute_mode ← 0;
                FPU_exception_pending_mode ← 1
            )
        )
)
```

Coprocessor Instruction Execution

The CP can execute coprocessor operate (CPop) instructions concurrently with integer instructions and other CPops. Although the instruction set includes a "store CP double queue" instruction, the existence of the queue and the type of concurrency available in the coprocessor is dependent on the coprocessor itself.

The FPU leaves FPU_exception_mode and enters FPU_execute_mode after the FQ has been emptied (via execution of STDFQ instructions.)

```
execute_CP_instruction := ( {not specified} ) ;
```

Traps

```
execute_trap := (
        select_trap;
        ET ← 0;                                    {ignore asynchronous traps}
        PS ← S;
        annul ← 0;
```

```
        CWP ← (CWP - 1) mod NWINDOWS;{point to next window}
        r[17] ← PC;                                  {preserve program counters}
        r[18] ← nPC;
        next;
        S ← 1;                                       {set supervisor mode}
        if (reset_trap = 0) then (
            PC ← TBR;
            nPC ← TBR + 4
        ) else (
            reset_trap ← 0;
            PC ← 0;
            nPC ← 4
        )
);
select_trap := (
        if (ET = 0 or reset_trap = 1) then
            error_mode ← 1
        else if (instruction_access_exception = 1) then
            tt ← 00000001₂
        else if (illegal_instruction = 1) then
            tt ← 00000010₂
        else if (privileged_instruction = 1) then
            tt ← 00000011₂
        else if (fp_disabled = 1) then
            tt ← 00000100₂
        else if (cp_disabled = 1) then
            tt ← 00100100₂
        else if (window_overflow = 1) then
            tt ← 00000101₂
        else if (window_underflow = 1) then
            tt ← 00000110₂
        else if (mem_address_not_aligned = 1) then
            tt ← 00000111₂
        else if (fp_exception = 1) then
            tt ← 00001000₂;
        else if (cp_exception = 1) then
            tt ← 00101000₂;
```

```
        else if (data_access_exception = 1) then
                tt ← 00001001₂
        else if (tag_overflow = 1) then
                tt ← 00001010₂
        else if (trap_instruction = 1) then
                tt ← 1₂□ticc_trap_type
        else if (interrupt_level > 0) then
                tt ← 0001₂□interrupt_level
        next;
        trap ← 0;           {since the tt field has been set, reset the trap signal}
        reset_trap ← 0;
        instruction_access_exception ← 0;
        illegal_instruction ← 0;
        privileged_instruction ← 0;
        fp_disabled ← 0;
        cp_disabled ← 0;
        window_overflow ← 0;
        window_underflow ← 0;
        mem_address_not_aligned ← 0;
        fp_exception ← 0;
        cp_exception ← 0;
        data_access_exception ← 0;
        tag_overflow ← 0;
        trap_instruction ← 0;
        interrupt_level ← 0
);
check_interrupts := (
        if (bp_reset_in = 1) then (
                reset_mode ← 1
        ) else if (ET = 1 and (bp_IRL = 15 or bp_IRL > PIL)) then (
                trap ← 1;
                interrupt_level ← bp_IRL
        );
);
```

Instruction Definitions

This section contains the ISP definitions of the SPARC architecture instructions. These complement the instruction descriptions in *Appendix B, Instruction Descriptions*.

Load Instructions

```
if ( (LDF or LDDF or LDFSR) then (
      if (EF = 0 or bp_FPU_present = 0) then (
            trap ← 1; fp_disabled ← 1
      ) else if (FPU_exception_mode = 1) then (
            ftt ← sequence_error;
            FPU_exception_mode ← 0 ;
            FPU_exception_pending_mode ← 1 ;
) ;
if ( (LDC or LDDC or LDCSR) and (EC = 0 or bp_CP_present = 0) ) then (
      trap ← 1; cp_disabled ← 1 ) ;
next;
if (trap = 0) then (
      if (LDD or LD or LDSH or LDUH or LDSB or LDUB
          or LDDF or LDF or LDFSR or LDDC or LDC or LDCSR) then (
            address ← r[rs1] + (if i=0 then r[rs2] else sign_extend(simm13));
            addr_space ← (if (S = 0) then 10 else 11)
      ) else if (LDDA or LDA or LDSHA or LDUHA or LDSBA or LDUBA) then (
            if (S = 0) then (
                  trap ← 1; privileged_instruction ← 1
            )
            address ← r[rs1] + r[rs2];
            addr_space ← asi
      );
);
next;
if (trap = 0) then (
      if ( ((LDD or LDDA or LDDF or LDDC) and address<2:0> ≠ 0) or
           ((LD or LDA or LDF or LDFSR or LDC or LDCSR) and address<1:0> ≠ 0) or
           ((LDSH or LDSHA or LDUH or LDUHA) and address<0> ≠ 0) ) then (
           trap ← 1; mem_addr_not_aligned ← 1
      )
);
next;
if (trap = 0) then (
      data ← memory_read(addr_space, address);
      MAE ← bp_memory_exception;
      next;
```

```
        if (MAE = 1) then (
                trap ← 1; data_access_exception ← 1
        ) else (
                if (LDSB or LDSBA or LDUB or LDUBA) then (
                        if      (address<1:0> = 0) byte ← data<31:24>
                        else if (address<1:0> = 1) byte ← data<23:16>
                        else if (address<1:0> = 2) byte ← data<15:8>
                        else if (address<1:0> = 3) byte ← data<7:0>;
                        next;
                        if (LDSB or LDSBA) then
                                word0 ← sign_extend_byte(byte)
                        else
                                word0 ← zero_extend_byte(byte)
                ) else if (LDSH or LDSHA or LDUH or LDUHA) then (
                        if      (address<1:0> = 0) halfword ← data<31:16>
                        else if (address<1:0> = 2) halfword ← data<15:0>;
                        next;
                        if (LDSH or LDSHA) then
                                word0 ← sign_extend_halfword(halfword)
                        else
                                word0 ← zero_extend_halfword(halfword)
                ) else
                        word0 ← data
        )
);
next;
if (trap = 0) then (
        if ( rd ≠ 0 and (LD or LDA or LDSH or or LDSHA or LDUHA or LDUH or LDSB or LDSBA
             or LDUB or LDUBA) ) then
                r[rd] ← word0
        else if ( ((rd and 11110₂) ≠ 0) and (LDD or LDDA) ) then
                r[rd and 11110₂] ← word0
        else if (LDF) then
                f[rd] ← word0
        else if (LDFSR) then (
                wait_for_FAUs_to_complete; {implementation-defined}
                FSR ← word0 )
```

```
      else if (LDC) then
            c[rd] ← word0
      else if (LDCSR) then
            CSR ← word0
);
next;
if (trap = 0 and (LDD or LDDA or LDDF or LDDC)) then (
      word1 ← memory_read(addr_space, address + 4);
      MAE ← bp_memory_exception;
      next;
      if (MAE = 1) then (
            trap ← 1; data_access_exception ← 1 )
      else if (LDD or LDDA) then
            r[rd or 1] ← word1
      else if (LDDF) then
            f[rd or 1] ← word1
      else if (LDDC) then
            c[rd or 1] ← word1
);
```

Store Instructions

```
if ((STF or STDF or STFSR or STDFQ) and (EF = 0 or bp_FPU_present = 0) ) then (
      trap ← 1; fp_disabled ← 1 ) ;
if ((STC or STDC or STCSR or STDCQ) and (EC = 0 or bp_CP_present = 0) ) then (
      trap ← 1; cp_disabled ← 1 ) ;
if (trap = 0) then (
      if (STD or ST or STH or STB or STF or STDF or STFSR or STDFQ or STCSR or STC or
      STDC or STDCQ) then (
            address ← r[rs1] + (if i=0 then r[rs2] else sign_extend(simm13));
            addr_space ← (if S=0 then 10 else 11)
      ) else if (STDA or STA or STHA or STBA) then (
            if (S = 0) then (
                  trap ← 1; privileged_instruction ← 1
            ) else (
                  address ← r[rs1] + r[rs2];
                  addr_space ← asi;
            )
      );
```

```
);
next;
if (trap = 0) then (
      if (STD or STDA or STDF or STDFQ or STDC or STDCQ) then (
            if (address<2:0> ≠ 0) then
                  trap ← 1; mem_addr_not_aligned ← 1 )
      else if (ST or STA or STF or STFSR or STC or STCSR) the (
            if (address<1:0> ≠ 0) then
                  trap ← 1; mem_addr_not_aligned ← 1 )
      else if (STH or STHA) then (
            if (address<0> ≠ 0) then (
                  trap ← 1; mem_addr_not_aligned ← 1 )
      );
);
next;
if (trap = 0) then (
      if (STDF) then (
            byte_mask ← 1111₂; data0 ← f[rd and 1110₂] )
      else if (STDFQ) then (
            byte_mask ← 1111₂; data0 ← FQ.ADDR )
      else if (STDC) then (
            byte_mask ← 1111₂; data0 ← c[rd and 1110₂] )
      else if (STDCQ) then (
            byte_mask ← 1111₂; data0 ← CQ.ADDR )
      else if (STD or STDA) then (
            byte_mask ← 1111₂; data0 ← r[rd and 1110₂] )
      else if (ST or STA) then )
            byte_mask = 1111₂; data0 = r[rd])
      else if (STH or STHA) then (
            if (address<1:0> = 0) then (
                  byte_mask ← 1100₂; data0 ← shift_left_logical(r[rd], 16) )
            else if (address<1:0> = 2) then (
                  byte_mask ← 0011₂; data0 ← r[rd] ) )
      else if (STB or STBA) then (
            if (address<1:0> = 0) then (
                  byte_mask ← 1000₂; data0 ← shift_left_logical(r[rd], 24) )
            else if (address<1:0> = 1) then (
```

```
                byte_mask ← 0100₂; data0 ← shift_left_logical(r[rd], 16) )
           else if (address<1:0> = 2) then (
                byte_mask ← 0010₂; data0 ← shift_left_logical(r[rd], 8) )
           else if (address<1:0> = 3) then (
                byte_mask ← 0001₂; data0 ← r[rd] )
      );
);
next;
if (trap = 0) then (
      memory_write(addr_space, address, byte_mask, data0);
      MAE ← bp_memory_exception
      next;
      if (MAE = 1) then (
           trap ← 1; data_access_exception ← 1
      )
);
next;
if (trap = 0) then (
      if (STD or STDA) then data1 ← r[rd or 1]
      else if (STDF) then data1 ← f[rd or 1]
           else if (STDFQ) then (
                  data1 <- FQ.INSTR;
                  dequeue_FQ;
           next;
           if (qne = 0) then (
                  FPU_exception_mode ← 0 ;
                  FPU_execute_mode ← 1
           )
      )
      else if (STDC) then data1 ← c[rd or 1]
      else if (STDCQ) then data1 ← CQ.INSTR
      next;
      memory_write(addr_space, address + 4, 1111₂, data1);
      MAE ← bp_memory_exception;
      next;
      if (MAE = 1) then (
           trap ← 1; data_access_exception ← 1
      )
);
```

Atomic Load-Store Unsigned Byte Instructions

```
if (LDSTUB) then (
      address ← r[rs1] + (if i=0 then r[rs2] else sign_extend(simm13));
      addr_space ← (if (S = 0) then 10 else 11)
) else if (LDSTUBA) then (
      if (S = 0) then (
            trap ← 1; privileged_instruction ← 1
      )
      address ← r[rs1] + r[rs2];
      addr_space ← asi
);
next;
if (trap = 0) then (
      pb_retain_bus ← 1;
      next;
      data ← memory_read(addr_space, address);
      MAE ← bp_memory_exception;
      next;
      if (MAE = 1) then (
            trap ← 1; data_access_exception ← 1
      ) else (
            if      (address<1:0> = 0) word ← zero_extend_byte(data<31:24>)
            else if (address<1:0> = 1) word ← zero_extend_byte(data<23:16>)
            else if (address<1:0> = 2) word ← zero_extend_byte(data<15:8>)
            else if (address<1:0> = 3) word ← zero_extend_byte(data<7:0>);
            next;
            if (rd ≠ 0) then r[rd] ← word
      )
);
next;
if (trap = 0) then (
            if      (address<1:0> = 0) then ( byte_mask ← 1000₂)
            else if (address<1:0> = 1) then ( byte_mask ← 0100₂)
            else if (address<1:0> = 2) then ( byte_mask ← 0010₂)
            else if (address<1:0> = 3) then ( byte_mask ← 0001₂)
      ;
      next;
```

```
      memory_write(addr_space, address, byte_mask, FFFFFFFF16);
      MAE ← bp_memory_exception;
      next;
      pb_retain_bus ← 0;
      if (MAE = 1) then (
           trap ← 1; data_access_exception ← 1
      )
);
```

Swap r Register with Memory Instructions

```
if (SWAP) then (
      address ← r[rs1] + (if i=0 then r[rs2] else sign_extend(simm13));
      addr_space ← (if (S = 0) then 10 else 11)
) else if (SWAPA) then (
      if (S = 0) then (
           trap ← 1; privileged_instruction ← 1
      )
      address ← r[rs1] + r[rs2];
      addr_space ← asi
);
next;
if (trap = 0) then (
      temp ← r[rd];
      pb_retain_bus ← 1;
      next;
      word ← memory_read(addr_space, address);
      MAE ← bp_memory_exception;
      next;
      if (MAE = 1) then (
           trap ← 1;  data_access_exception ← 1
      ) else (
           if (rd ≠ 0) then r[rd] ← word
      )
);
next;
if (trap = 0) then (
      memory_write(addr_space, address, 1111₂, temp);
      MAE ← bp_memory_exception;
```

```
      next;
      pb_retain_bus ← 0;
      if (MAE = 1) then (
            trap ← 1; data_access_exception ← 1
      )
);
```

Add Instructions

```
operand2 := if i=0 then r[rs2] else sign_extend(simm13);

if (ADD or ADDcc) then
      result ← r[rs1] + operand2;
else if (ADDX or ADDXcc) then
      result ← r[rs1] + operand2 + C;
next;
if (rd ≠ 0) then
      r[rd] ← result;
if (ADDcc or ADDXcc) then (
      N ← result<31>;
      Z ← if result=0 then 1 else 0;
      V ← (r[rs1]<31> and operand2<31> and not result<31>) or
           (not r[rs1]<31> and not operand2<31> and result<31>);
      C ← (r[rs1]<31> and operand2<31>) or
           (not result<31> and (r[rs1]<31> or operand2<31>))
);
```

Tagged Add Instructions

```
operand2 := if i=0 then r[rs2] else sign_extend(simm13);

result ← r[rs1] + operand2;
next;
temp_v ← (r[rs1]<31> and operand2<31> and not result<31>) or
     (not r[rs1]<31> and not operand2<31> and result<31>) or
     (r[rs1]<1:0> ≠ 0 or operand2<1:0> ≠ 0);
next;
if (TADDccTV and temp_v = 1) then (
      trap ← 1; tag_overflow ← 1
) else (
      N ← result<31>;
      Z ← if result=0 then 1 else 0;
      V ← temp_v;
```

```
      C ← (r[rs1]<31> and operand2<31>) or
           (not result<31> and (r[rs1]<31> or operand2<31>));
      if (rd ≠ 0) then
           r[rd] ← result;
);
```

Subtract Instructions

```
operand2 := if i=0 then r[rs2] else sign_extend(simm13);

if (SUB or SUBcc) then
      result ← r[rs1] - operand2;
else if (SUBX or SUBXcc) then
      result ← r[rs1] - operand2 - C;
next;
if (rd ≠ 0) then
      r[rd] ← result;
if (SUBcc or SUBXcc) then (
      N ← result<31>;
      Z ← if result=0 then 1 else 0;
      V ← (r[rs1]<31> and not operand2<31> and not result<31>) or
           (not r[rs1]<31> and operand2<31> and result<31>);
      C ← (not r[rs1]<31> and operand2<31>) or
           (result<31> and (not r[rs1]<31> or operand2<31>))
);
```

Tagged Subtract Instructions

```
operand2 := if i=0 then r[rs2] else sign_extend(simm13);

result ← r[rs1] - operand2;
next;
temp_v ← (r[rs1]<31> and not operand2<31> and not result<31>) or
     (not r[rs1]<31> and operand2<31> and result<31>) or
     (r[rs1]<1:0> ≠ 0 or operand2<1:0> ≠ 0);
next;
if (TSUBccTV and temp_v = 1) then (
      trap ← 1; tag_overflow ← 1
) else (
      N ← result<31>;
      Z ← if result=0 then 1 else 0;
      V ← temp_v;
      C ← (not r[rs1]<31> and operand2<31>) or
           (result<31> and (not r[rs1]<31> or operand2<31>));
```

```
      if (rd ≠ 0) then
           r[rd] ← result;
);
```

Multiply Step Instruction

```
operand1 := (N xor V)□r[rs1]<31:1>;
operand2 := (
      if (Y<0> = 0) then 0
      else if (i = 0) then r[rs2] else sign_extend(simm13)
);
result ← operand1 + operand2;
Y ← r[rs1]<0>□Y<31:1>;
next;
if (rd ≠ 0) then
      r[rd] ← result;
N ← result<31>;
Z ← if result=0 then 1 else 0;
V ← (operand1<31> and operand2<31> and not result<31>) or
     (not operand1<31> and not operand2<31> and result<31>);
C ← (operand1<31> and operand2<31>) or
     (not result<31> and (operand1<31> or operand2<31>))
```

Logical Instructions

```
operand2 := if i=0 then r[rs2] else sign_extend(simm13);

if (AND or ANDcc) then result ← r[rs1] and operand2
else if (ANDN or ANDNcc) then result ← r[rs1] and not operand2
else if (OR or ORcc) then result ← r[rs1] or operand2
else if (ORN or ORNcc) then result ← r[rs1] or not operand2
else if (XOR or XORcc) then result ← r[rs1] xor operand2
else if (XNOR or XNORcc) then result ← r[rs1] xor not operand2;
next;
if (rd ≠ 0) then r[rd] ← result;
if (ANDcc or ANDNcc or ORcc or ORNcc or XORcc or XNORcc) then (
      N ← result<31>;
      Z ← if result=0 then 1 else 0;
      V ← 0;
      C ← 0
);
```

Shift Instructions

```
shift_count := if i=0 then r[rs2]<4:0> else shcnt;

if (SLL and rd ≠ 0) then
      r[rd] ← shift_left_logical(r[rs1], shift_count)
else if (SRL and rd ≠ 0) then
      r[rd] ← shift_right_logical(r[rs1], shift_count)
else if (SRA and rd ≠ 0) then
      r[rd] ← shift_right_arithmetic(r[rs1], shift_count)
```

SETHI Instruction

```
if (rd ≠ 0) then (
      r[rd]<31:10> ← imm22;
      r[rd]<9:0> ← 0
)
```

SAVE and RESTORE Instructions

```
operand2 := if i=0 then r[rs2] else sign_extend(simm13);

if (SAVE) then (
      new_cwp ← (CWP - 1) mod NWINDOWS;
      next;
      if ((WIM and 2^new_cwp) ≠ 0) then (
            trap ← 1; window_overflow ← 1
      ) else (
            result ← r[rs1] + operand2;          {operands from old window}
            CWP ← new_cwp
      )
) else if (RESTORE) then (
      new_cwp ← (CWP + 1) mod NWINDOWS;
      next;
      if ((WIM and 2^new_cwp) ≠ 0) then (
            trap ← 1; window_underflow ← 1
      ) else (
            result ← r[rs1] + operand2;          {operands from old window}
            CWP ← new_cwp
      )
);
next;
```

```
if (trap = 0 and rd ≠ 0) then
      r[rd] ← result                                  {destination in new window}
```

Branch on Integer Condition Instructions

```
eval_icc := (
      if (BNE and (Z = 0)) then 1 else 0;
      if (BE and (Z = 1)) then 1 else 0;
      if (BG and ((Z or (N xor V)) = 0)) then 1 else 0;
      if (BLE and ((Z or (N xor V)) = 1)) then 1 else 0;
      if (BGE and ((N xor V) = 0)) then 1 else 0;
      if (BL and ((N xor V) = 1)) then 1 else 0;
      if (BGU and (C = 0 and Z = 0)) then 1 else 0;
      if (BLEU and (C = 1 or Z = 1)) then 1 else 0;
      if (BCC and (C = 0)) then 1 else 0;
      if (BCS and (C = 1)) then 1 else 0;
      if (BPOS and (N = 0)) then 1 else 0;
      if (BNEG and (N = 1)) then 1 else 0;
      if (BVC and (V = 0)) then 1 else 0;
      if (BVS and (V = 1)) then 1 else 0;
      if (BA) then 1;
      if (BN) then 0
);
PC ← nPC;
if (eval_icc) = 1 then (
      nPC ← PC + sign_extend(disp22□00₂);
      if (BA and a = 1) then
            annul ← 1
) else (
      nPC ← nPC - 4;
      if (a = 1) then
            annul ← 1
)
```

Floating-Point Branch on Condition Instructions

```
E := if fcc=0 then 1 else 0;
L := if fcc=1 then 1 else 0;
G := if fcc=2 then 1 else 0;
U := if fcc=3 then 1 else 0;
```

```
eval_fcc := (
      if (FBU and U)) then 1 else 0;
      if (FBG and G) then 1 else 0;
      if (FBUG and (G or U) then 1 else 0;
      if (FBL and L) then 1 else 0;
      if (FBUL and (L or U)) then 1 else 0;
      if (FBLG and (L or G)) then 1 else 0;
      if (FBNE and (L or G or U)) then 1 else 0;
      if (FBE and E)) then 1 else 0;
      if (FBUE and (E or U)) then 1 else 0;
      if (FBGE and (E or G)) then 1 else 0;
      if (FBUGE and (E or G or U)) then 1 else 0;
      if (FBLE and (E or L)) then 1 else 0;
      if (FBULE and (E or L or U)) then 1 else 0;
      if (FBO and (E or L or G)) then 1 else 0;
      if (FBA) then 1;
      if (FBN) then 0
);

PC ← nPC;
if (eval_fcc = 1) then (
      nPC ← PC + sign_extend(disp22□00₂);
      if (FBA and (a = 1)) then
            annul ← 1
) else (
      nPC ← nPC + 4;
      if (a = 1) then
            annul ← 1
)
```

Coprocessor Branch on Condition Instructions

```
C0 := if bp_CP_cc<1:0>=0 then 1 else 0;
C1 := if bp_CP_cc<1:0>=1 then 1 else 0;
C2 := if bp_CP_cc<1:0>=2 then 1 else 0;
C3 := if bp_CP_cc<1:0>=3 then 1 else 0;
eval_bp_CP_cc := (
      if (CB3 and C3)) then 1 else 0;
      if (CB2 and C2) then 1 else 0;
```

```
      if (CB23 and (C2 or C3) then 1 else 0;
      if (CB1 and C1) then 1 else 0;
      if (CB13 and (C1 or C3)) then 1 else 0;
      if (CB12 and (C1 or C2)) then 1 else 0;
      if (CB123 and (C1 or C2 or C3)) then 1 else 0;
      if (CB0 and C0)) then 1 else 0;
      if (CB03 and (C0 or C3)) then 1 else 0;
      if (CB02 and (C0 or C2)) then 1 else 0;
      if (CB023 and (C0 or C2 or C3)) then 1 else 0;
      if (CB01 and (C0 or C1)) then 1 else 0;
      if (CB013 and (C0 or C1 or C3)) then 1 else 0;
      if (CB012 and (C0 or C1 or C2)) then 1 else 0;
      if (CBA) then 1;
      if (CBN) then 0
);
PC ← nPC;
if (eval_bp_CP_cc = 1) then (
      nPC ← PC + sign_extend(disp22□00₂);
      if (CBA and (a = 1)) then
            annul ← 1
) else (
      nPC ← nPC + 4;
      if (a = 1) then
            annul ← 1
)
```

CALL Instruction

```
r[15] ← PC;
PC ← nPC;
nPC ← PC + disp30□00₂
```

Jump and Link Instruction

```
jump_address ← r[rs1] + (if i=0 then r[rs2] else sign_ext(simm13));
next;
if (jump_address<1:0> ≠ 0) then (
      trap ← 1;
      mem_address_not_aligned ← 1
) else (
```

```
      if (rd ≠0) then r[rd] ← PC;
      PC ← nPC;
      nPC ← jump_address
)
```

Return from Trap Instruction

```
new_cwp ← (CWP + 1) mod NWINDOWS;
address ← r[rs1] + (if i=0 then r[rs2] else sign_extend(simm13));
next;
if (ET) then (
      trap ← 1;
      illegal_instruction ← 1
) else if (S = 0) then (
      trap ← 1;
      privileged_instruction ← 1
) else if ((WIM and 2^new_cwp) ≠ 0) then (
      trap ← 1;
      window_underflow ← 1
) else if (address<1:0> ≠ 0) then (
      trap ← 1;
      mem_address_not_aligned ← 1
) else (
      ET ← 1;
      PC ← nPC;
      nPC ← address;
      CWP ← new_cwp;
      S ← pS
)
```

Trap on Integer Condition Instructions

```
trap_eval_icc := (
      if (TNE and (Z = 0)) then 1 else 0;
      if (TE and (Z = 1)) then 1 else 0;
      if (TG and ((Z or (N xor V)) = 0)) then 1 else 0;
      if (TLE and ((Z or (N xor V)) = 1)) then 1 else 0;
      if (TGE and ((N xor V) = 0)) then 1 else 0;
      if (TL and ((N xor V) = 1)) then 1 else 0;
      if (TGU and (C = 0 and Z = 0)) then 1 else 0;
```

```
      if (TLEU and (C = 1 or Z = 1)) then 1 else 0;
      if (TCC and (C = 0)) then 1 else 0;
      if (TCS and (C = 1)) then 1 else 0;
      if (TPOS and (N = 0)) then 1 else 0;
      if (TNEG and (N = 1)) then 1 else 0;
      if (TVC and (V = 0)) then 1 else 0;
      if (TVS and (V = 1)) then 1 else 0;
      if (TA) then 1;
      if (TN) then 0
);

trap_number := r[rs1] + (if i=0 then r[rs2] else sign_ext(simm13));
if (Ticc) then (
      if (trap_eval_icc = 1) then (
            trap ← 1;
            trap_instruction ← 1;
            ticc_trap_type ← trap_number <6:0>
      ) else (
            PC ← nPC;
            nPC ← nPC + 4
      )
);
```

Read State Register Instructions

```
if ((RDPSR or RDWIM or RDTBR) and S = 0) then (
      trap ← 1;
      privileged_instruction ← 1
) else if (rd ≠ 0) then (
      if (RDY) then
            r[rd] ← Y
      else if (RDPSR) then
            r[rd] ← PSR
      else if (RDWIM) then
            r[rd] ← WIM
      else if (RDTBR) then
            r[rd] ← TBR;
);
```

Write State Register Instructions

```
operand2 := if i=0 then r[rs2] else sign_extend(simm13);
result := r[rs1] xor operand2;
if (WRY) then
      Y' ← result
else if (WRPSR) then (
      if (result<4:0> ≥ NWINDOWS) then (
           trap ← 1;
           illegal_instruction ← 1
      ) else if (S = 0) then (
           trap ← 1;
           privileged_instruction ← 1
      ) else
           PSR' ← result
) else if (WRWIM) then (
      if (S = 0) then (
           trap ← 1;
           privileged_instruction ← 1
      ) else
           WIM' ← result
) else if (WRTBR) then
      if (S = 0) then (
           trap ← 1;
           privileged_instruction ← 1
      ) else
           TBR' ← result
);
```

Unimplemented Instruction

```
trap ← 1;
illegal_instruction ← 1
```

Instruction Cache Flush Instruction

```
address := r[rs1] + (if i=0 then r[rs2] else sign_extend(simm13));

if (IU_cache_present) then
      flush_IU_cache_word(address)                {implementation-dependent}
else if (bp_I_cache_present) then (
      trap ← 1;
```

```
        illegal_instruction ← 1
)
```

Floating-Point Operate Instructions

The multiple precision FPops use the following notation to indicate *f register* alignment:

double precision

```
        rs1E := rs1<4:1>□02; rs1O := rs1<4:1>□12;
        rs2E := rs2<4:1>□02; rs2O := rs2<4:1>□12;
        rdE := rd<4:1>□02; rdO := rd<4:1>□12
```

extended precision

```
        rs1EE := rs1<4:2>□002; rs1EO := rs1<4:2>□012; rs1OE := rs1<4:2>□102;
        rs2EE := rs2<4:2>□002; rs2EO := rs2<4:2>□012; rs2OE := rs2<4:2>□102;
        rdEE := rd<4:2>□002; rdEO := rd<4:2>□012; rdOE := rd<4:2>□102
```

Most of the floating-point routines defined below (or not defined since they are implementation-dependent) return: (1) a single, double, or extended *result*; (2) a 5-bit exception vector (*texc*) similar to the *cexc* field of the FSR, or a 2-bit condition code vector (*tfcc*) identical to the *fcc* field of the FSR; and (3) a completion status bit (*c*) which indicates whether the arithmetic unit was able to complete the operation.

Convert Integer to Floating-Point Instructions

```
if (FiTOs) then
        result, texc, c ← cvt_integer_to_single(f[rs2])
else if (FiTOd) then
        result, texc, c ← cvt_integer_to_double(f[rs2])
else if (FiTOx) then
        result, texc, c ← cvt_integer_to_extended(f[rs2])
```

Convert Floating-Point to Integer

```
if (FsTOi) then
        result, texc, c ← cvt_single_to_integer(f[rs2])
else if (FdTOi) then
        result, texc, c ← cvt_double_to_integer(f[rs2E]□f[rs2O])
else if (FxTOi) then
        result, texc, c ← cvt_extended_to_integer(f[rs2EE]□f[rs2EO]□f[rs2OE]);
```

Convert Between Floating-Point Formats Instructions

```
if (FsTOd) then
        result, texc, c ← cvt_single_to_double(f[rs2])
else if (FsTOx) then
        result, texc, c ← cvt_single_to_extended(f[rs2])
```

```
else if (FdTOs) then
      result, texc, c ← cvt_double_to_single(f[rs2E]□f[rs2O])
else if (FdTOx) then
      result, texc, c ← cvt_double_to_extended(f[rs2E]□f[rs2O])
else if (FxTOs) then
      result, texc, c ← cvt_extended_to_single(f[rs2E]□f[rs2O]□f[rs2OE])
else if (FxTOd) then
      result, texc, c ← cvt_extended_to_double(f[rs2EE]□f[rs2EO]□f[rs2OE])
```

Floating-Point Move Instructions

```
if (FMOVs) then
      result ← f[rs2]
else if (FNEGs) then
      result ← f[rs2] xor 80000000₁₆
else if (FABSs) then
      result ← f[rs2] and 7FFFFFFF₁₆;
texc ← 0;
C ← 1
```

Floating-Point Square Root Instructions

```
if (FSQRTs) then
      result, texc, c ← sqrt_single(f[rs2])
else if (FSQRTd) then
      result, texc, c ← sqrt_double(f[rs2E]□f[rs2O])
else if (FSQRTx) then
      result, texc, c ← sqrt_extended(f[rs2EE]□f[rs2EO]□f[rs2OE])
```

Floating-Point Add and Subtract Instructions

```
if (FADDs) then
      result, texc, c ← add_single(f[rs1], f[rs2])
else if (FSUBs) then
      result, texc, c ← sub_single(f[rs1], f[rs2])
else if (FADDd) then
      result, texc, c ← add_double(f[rs1E]□f[rs1O], f[rs2E]□f[rs2O])
else if (FSUBd) then
      result, texc, c ← sub_double(f[rs1E]□f[rs1O], f[rs2E]□f[rs2O])
else if (FADDx) then
      result, texc, c ← add_extended(f[rs1EE]□f[rs1EO]□f[rs1OE],
           f[rs2EE]□f[rs2EO]□f[rs2OE])
```

```
else if (FSUBx) then
      result, texc, c ← sub_extended(f[rs1EE]□f[rs1EO]□f[rs1OE],
           f[rs2EE]□f[rs2EO]□f[rs2OE])
```

Floating-Point Multiply and Divide Instructions

```
if (FMULs) then
      result, texc, c ← mul_single(f[rs1], f[rs2])
else if (FDIVs) then
      result, texc, c ← div_single(f[rs1], f[rs2])
else if (FMULd) then
      result, texc, c ← mul_double(f[rs1E]□f[rs1O], f[rs2E]□f[rs2O])
else if (FDIVd) then
      result, texc, c ← div_double(f[rs1E]□f[rs1O], f[rs2E]□f[rs2O])
else if (FMULx) then
      result, texc, c ← mul_extended(f[rs1EE]□f[rs1EO]□f[rs1OE],
           f[rs2EE]□f[rs2EO]□f[rs2OE])
else if (FDIVx) then
      result, texc, c ← div_extended(f[rs1EE]□f[rs1EO]□f[rs1OE],
           f[rs2EE]□f[rs2EO]□f[rs2OE])
```

Floating-Point Compare Instructions

```
if (FCMPs) then
      tfcc, texc, c ← compare_single(f[rs1], f[rs2])
else if (FCMPd) then
      tfcc, texc, c ← compare_double(f[rs1E]□f[rs1O], f[rs2E]□f[rs2O])
else if (FCMPx) then
      tfcc, texc, c ← compare_extended(f[rs1EE]□f[rs1EO]□f[rs1OE],
           f[rs2EE]□f[rs2EO]□f[rs2OE])
else if (FCMPEs) then
      tfcc, texc, c ← compare_e_single(f[rs1], f[rs2]);
else if (FCMPEd) then
      tfcc, texc, c ← compare_e_double(f[rs1E]□f[rs1O], f[rs2E]□f[rs2O])
else if (FCMPEx) then
      tfcc, texc, c ← compare_e_extended(f[rs1EE]□f[rs1EO]□f[rs1OE],
           f[rs2EE]□f[rs2EO]□f[rs2OE])
```

APPENDIX D

Software Considerations

This appendix describes how software can use the SPARC architecture effectively. It describes assumptions that compilers may make about the resources available, and how compilers can use them. It does not discuss how the operating system may use the architecture.

Registers

How to use registers is typically a very important resource allocation problem for compilers. The SPARC architecture provides windowed registers (*in, out, local*), global registers, and floating-point registers.

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. When a routine is called, the caller's *outs* become the callee's *ins*.

One of the caller's *out* registers is used as the stack pointer, SP. It points to an area in which the system can store *r16* through *r31* when the register file overflows. **It is essential that this register have the correct value when the corresponding underflow trap occurs so that the register window can be reloaded.** It is also important that this register be kept up to date with register window changes, and that the overhead for doing calls be kept as small as possible. Since SP is in one of the caller's *out* registers, it can be used by the callee as its FP, and the callee can use the SAVE instruction to set its own SP from its FP.

Up to six parameters* may be passed by placing them in the *out* registers; additional parameters are passed in the memory stack. When the callee is entered, the parameters passed in registers are now in its corresponding *ins*. One of the other two *in/out* registers is used as the caller's old SP, which is also the current routine's frame pointer, FP (see below). The other is used to pass the subroutine's return address. With the exception of SP, *out* registers may be used as temporaries between subroutine calls.

If a routine is passed more than six parameters, the remainder are passed on the memory stack. If, on the other hand, it is passed fewer than six parameters, it may use the other parameter registers as if they were *locals*. If a register parameter has its address taken, it must be stored on the memory stack, and used from there for the lifetime of the pointer (or for the extent of the procedure, if the compiler cannot figure this out). A function returns its value by writing it into its *ins* (which are the caller's *outs*).

---

*Six is more than adequate, since the overwhelming majority of procedures in system code—at least 97% measured statically, according to the studies cited by Weicker (Weicker, R.P., Dhrystone: A Synthetic Systems Programming Benchmark, *CACM* 27:10, October 1984)—take fewer than six parameters. The average number of parameters, measured statically or dynamically, is no greater than 2.1 in any of these studies.

Local Registers

The *locals* are used for automatic variables and most temporaries. The compiler may also copy parameters out of the memory stack into the *locals* and use them from there. If an automatic variable has its address taken, it must be stored in the memory stack for the lifetime of the pointer (or for the extent of the procedure, if the compiler cannot figure this out).

Global Registers

Unlike the *ins*, *locals*, and *outs*, the *globals* are not part of any register window, but are a single set of registers with global scope, like the registers of a more traditional architecture. This means that if they are used on a per-procedure basis, they must be saved and restored.

The *global* registers can be used for temporaries and for global variables or pointers, either visible to the user or maintained as part of the program's execution environment. For instance, one could by convention address all global scalars by offsets from register *r7*. This would allow 2^{13} bytes of global scalars, and would enable access to these variables faster than if they were only accessible via absolute addresses. This is because absolute addresses longer than 13 bits require a SETHI instruction.

Floating-Point Registers

There are thirty-two 32-bit floating-point registers. They are accessed differently from the other registers and cannot be moved to or from anything but memory. Like the global registers, they must be managed by software. Compilers probably will **not** pass parameters in them, but will use them for user variables and compiler temporaries. Across a procedure call, either the caller saves the live floating-point registers, or the callee saves the ones it uses and subsequently restores them.

	r31	(i7)	return address
	r30	(FP)	frame pointer
	r29	(i5)	incoming param reg 5
in	r28	(i4)	incoming param reg 4
	r27	(i3)	incoming param reg 3
	r26	(i2)	incoming param reg 2
	r25	(i1)	incoming param reg 1
	r24	(i0)	incoming param reg 0
	r23	(l7)	local 7
	r22	(l6)	local 6
	r21	(l5)	local 5
local	r20	(l4)	local 4
	r19	(l3)	local 3
	r18	(l2)	local 2
	r17	(l1)	local 1
	r16	(l0)	local 0
	r15	(o7)	temp
	r14	(SP)	stack pointer
	r13	(o5)	outgoing param reg 5
out	r12	(o4)	outgoing param reg 4
	r11	(o3)	outgoing param reg 3
	r10	(o2)	outgoing param reg 2
	r9	(o1)	outgoing param reg 1
	r8	(o0)	outgoing param reg 0
	r7	(g7)	global 7
	r6	(g6)	global 6
	r5	(g5)	global 5
global	r4	(g4)	global 4
	r3	(g3)	global 3
	r2	(g2)	global 2
	r1	(g1)	global 1
	r0	(g0)	0
	f31		floating-point value
floating point	:		:
	f0		floating-point value

Figure D-1. Registers as Seen by a Procedure

The Memory Stack

Parameters beyond the sixth are passed on the stack. Parameters which must be addressable are stored in the stack. Space is reserved on the stack for passing a one-word hidden parameter. This is used when the caller is expecting to be returned a C language *struct* by value; it gives the address of stack space allocated by the caller for that purpose (see Function Returning Aggregate Values). Space is reserved on the stack for keeping the procedure's *in* and *local* registers, should the register stack overflow. Automatic variables which must be addressable are kept there, as are some compiler-generated temporaries. These include automatic arrays and automatic records. Space is reserved on the stack for saving floating-point registers across calls. Space on the stack may be dynamically allocated using the `alloca` function from the C library. Automatic variables on the stack are addressed relative to FP, while temporaries and outgoing parameters are addressed relative to SP. When a procedure is active, its stack frame appears as in Figure D-2.

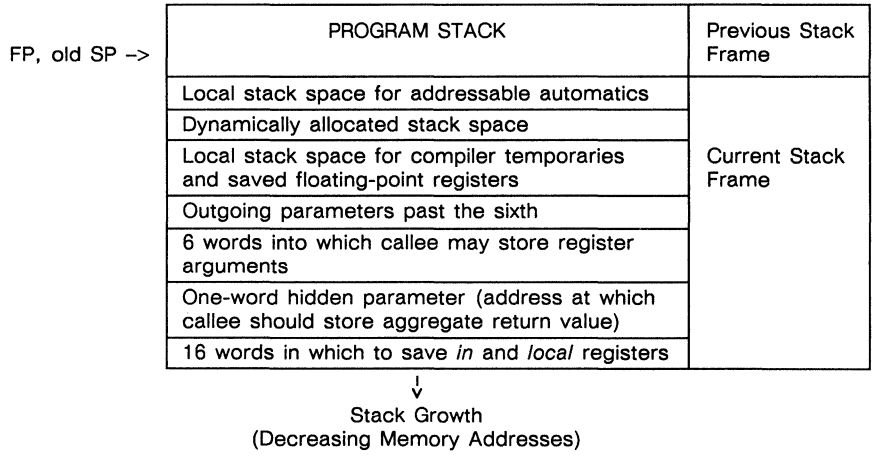


Figure D-2. The User Stack Frame

Example Code

In the following example we assume the following pseudo-instructions are provided by the assembler:

pseudo-instruction	equivalent instruction
ret	jmp %i7 + 8
retl	jmp %o7 + 8
mov *reg_or_imm, reg*	or %g0, *reg_or_imm, reg*

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

```
! CALLER:
!       int i;                                  /* in register %l7 */
!       i = sum3( 1, 2, 3 );
        ...
        mov   1, %o0
        mov   2, %o1
        call  sum3
        mov   3, %o2                            ! last parameter in delay slot
        mov   %o0, %l7
        ...

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

sum3:
        save  %sp, -(16*4), %sp                 ! setup new sp
        add   %i0, %i1, %l7                     ! compute sum in local
        add   %l7, %i2, %l7
        ret
        restore                     %l7, 0, %o0 ! move result into output reg, restore
```

Since "sum3" does not call any subroutines (i.e. it is a "leaf" routine) it can be recoded as:

```
sum3:
        add   %o0, %o1, %o3         ! use %o3 as a local
        retl                                    ! can't use ret; use retl
        add   %o2, %o3, %o0
```

Functions Returning Aggregate Values

Some programming languages, including C, some dialects of Pascal, and Modula-2, allow the user to define a function returning an aggregate value, such as a C **struct** 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. Reentrancy 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 the section *The Memory Stack* in this appendix. 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 function value 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 is the low-order twelve bits of the size in bytes of the aggregate value expected. When an aggregate-returning function is about to return 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, then the hidden parameter is assumed to be valid, and the function returns control to the location following the unimplemented instruction. Otherwise, the hidden parameter is assumed *not* to be valid, and no value can be returned. Conversely, if a scalar-returning function is called when an aggregate value is expected, the function returns as usual, executing the UNIMP instruction and causing a trap.

APPENDIX E

Example Integer Multiplication and Division Routines

This appendix contains routines a SPARC architecture system might use to perform integer multiplication and division.

In these examples, it is assumed that the assembler provides the following pseudo-instructions:

Pseudo Instruction	Equivalent Instruction
`nop`	`sethi 0,%g0`
`jmp`	`jmpl` *address*, `%g0`
`ret`	`jmp %i7 +8`
`retl`	`jmp %o7 +8`
`mov` *reg_or_imm, neg*	`or %g0,` *reg_or_imm, reg*
`tst` *reg*	`subcc` *reg*, `%g0, %g0`
`neg` *reg*	`sub %g0,` *reg, reg*
`cmp` *reg, reg_or_imm*	`subcc` *reg, reg_or_imm*, `%g0`
`inc` *reg*	`add` *reg*, `1,` *reg*
`incc` *reg*	`addcc` *reg*, `1,` *reg*
`dec` *reg*	`sub` *reg*, `1,` *reg*
`deccc` *reg*	`subcc` *reg*, `1,` *reg*

It is also assumed that the assembler recognizes " / * . . . * / "-style comments, and " ! " as the beginning of a comment which extends to the end of the current line.

Signed Multiplication

```
/*
 * Procedure to perform a 32-bit by 32-bit multiply.
 * Pass the multiplicand in %i0, and the multiplier in %i1.
 * The least significant 32 bits of the result are returned in %i0,
 * and the most significant in %i1.
 *
 * This code has an optimization built-in for short (less than 13-bit)
 * multiplies. Short multiplies require 26 or 27 instruction cycles,
 * and long ones require 47 to 51 instruction cycles. For two positive numbers
 * (the most common case) a long multiply takes 47 instruction cycles.
 *
 * This code indicates that overflow has occurred by leaving the Z condition
 * code clear. The following call sequence would be used if you wish
 * to deal with overflow:
 *
 *       call      .mul
 *       nop                          ! (or set up last parameter here)
 *       bnz       overflow_code      ! (or tnz to overflow handler)
 *
 * 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
        andncc      %o0, 0xfff, %g0      ! mask out lower 12 bits
        be          mul_shortway         ! can do it the short way
        andcc       %g0, %g0, %o4        ! zero the partial product and clear N and
                                         ! V conditions
        !
        ! long multiply
        !
        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
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        mulscc      %o4, %o1, %o4
        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
        !
        ! If %o0 (multiplier) was negative, the result is:
        !     (%o0 * %o1) + %o1 * (2**32)
        ! We fix that here.
        !
        tst         %o0
        rd          %y, %o0
        bge         1f
        tst         %o0                  ! for when we check for overflow

        sub         %o4, %o1, %o4        ! bit 33 and up of the product are in
                                         ! %o4, so we don't have to shift %o1
        !
        ! We haven't overflowed if:
        !     low-or         der bits are positive and high-order bits are 0
```

```
        !    low-order bits are negative and high-order bits are -1
        !
        ! If you are not interested in detecting overflow,
        ! replace the following few instructions with:
        !
        !         1:  retl
        !             mov           %o4, %o1
        !
1:
        bge       2f                ! if low-order bits were positive.
        addcc     %o4, %g0, %o1     ! return most sig. bits of prod and set
                                    ! Z appropriately (for positive product)
        retl                        ! leaf-routine return
        subcc     %o4, -1, %g0      ! set Z if high order bits are -1 (for negative
                                    ! product)
2:
        retl                        ! leaf-routine return
        nop
        !
        ! short multiply
        !
mul_shortway:
        mulscc    %o4, %o1, %o4     ! first iteration of 13
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4     ! 12th iteration
        mulscc    %o4, %g0, %o4     ! last iteration only shifts

        rd        %y, %o5
        sll       %o4, 12, %o0      ! left shift middle bits by 12 bits
        srl       %o5, 20, %o5      ! right shift low bits by 20 bits
        !
        ! We haven't overflowed if:
        !    low-order bits are positive and high-order bits are 0
        !    low-order bits are negative and high-order bits are -1
        !
        ! If you are not interested in detecting overflow,
        ! replace the following code with:
        !
        !         or        %o5, %o4, %o0
        !         retl
        !         mov       %o4, %o1
        !
        orcc      %o5, %o0, %o0     ! merge for true product
        bge       3f                ! if low-order bits were positive.
        sra       %o4, 20, %o1      ! right shift high bits by 20 bits
                                    ! and put into  %o1
        retl                        ! leaf-routine return
        subcc     %o1, -1, %g0      ! set Z if high order bits are -1 (for
                                    ! negative product)
```

```
3:
      retl                           ! leaf-routine return
      addcc     %o1, %g0, %g0        ! set Z if high order bits are 0
```

Unsigned Multiplication

```
/*
 * Procedure to perform a 32 by 32 unsigned multiply.
 * Pass the multiplier in %o0, and the multiplicand in %o1.
 * The least significant 32 bits of the result will be returned in %o0,
 * and the most significant in %o1.
 *
 * This code has an optimization built-in for short (less than 13 bit)
 * multiplies. Short multiplies require 25 instruction cycles, and long ones
 * require 46 or 48 instruction cycles.
 *
 * This code indicates that overflow has occurred, by leaving the Z condition
 * code clear. The following call sequence would be used if you wish to
 * deal with overflow:
 *
 *          call      .umul
 *          nop                      ! (or set up last parameter here)
 *          bnz       overflow_code  ! (or tnz to overflow handler)
 *
 * 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
.umul:
      or        %o0, %o1, %o4     ! logical or of multiplier and multiplicand
      mov       %o0, %y           ! multiplier to Y register
      andncc    %o4, 0xfff, %o5   ! mask out lower 12 bits
      be        mul_shortway      ! can do it the short way
      andcc     %g0, %g0, %o4     ! zero the partial product and clear N and
                                  ! V conditions
      !
      ! long multiply
      !
      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
      mulscc    %o4, %o1, %o4
      mulscc    %o4, %o1, %o4
      mulscc    %o4, %o1, %o4
      mulscc    %o4, %o1, %o4
      mulscc    %o4, %o1, %o4
      mulscc    %o4, %o1, %o4
      mulscc    %o4, %o1, %o4
      mulscc    %o4, %o1, %o4
      mulscc    %o4, %o1, %o4
      mulscc    %o4, %o1, %o4
      mulscc    %o4, %o1, %o4
      mulscc    %o4, %o1, %o4
      mulscc    %o4, %o1, %o4
```

```
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        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 shifty-add approach, if both numbers are positive,
         * you get the correct result. With 32-bit twos-complement numbers,
         * -x can be represented as ((2 - (x/(2**32)) mod 2) * 2**32. To avoid
         * a lot of 2**32's, we can 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 multiply.s)
         * 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
        bge       1f
        nop
        add       %o4, %o0, %o4
1:
        rd        %y, %o0            ! return least sig. bits of prod
        retl                         ! leaf-routine return
        addcc     %o4, %g0, %o1      ! delay slot; return high bits and set
                                     ! zero bit appropriately
        !
        ! short multiply
        !
mul_shortway:
        mulscc    %o4, %o1, %o4      ! first iteration of 13
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
```

```
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4
        mulscc    %o4, %o1, %o4       ! 12th iteration
        mulscc    %o4, %g0, %o4       ! last iteration only shifts

        rd        %y, %o5
        sll       %o4, 12, %o4        ! left shift partial product by 12 bits
        srl       %o5, 20, %o5        ! right shift product by 20 bits
        or        %o5, %o4, %o0       ! merge for true product
        !
        ! The delay instruction (addcc) moves zero into %o1,
        ! sets the zero condition code, and clears the other conditions.
        ! This is the equivalent result to a long umultiply which doesn't overflow.
        !
        retl                          ! leaf-routine return
        addcc     %g0, %g0, %o1
```

Division

Integer division implemented in software or microcode is usually done by a method such as the non-restoring algorithm, which provides one digit of quotient per step. A W-by-W digit division, of radix-B digits, is most easily achieved using 2*W-digit arithmetic.

Program 1

A binary-radix, 16-digit version of this method is illustrated by the C language function in Program 1, which performs an unsigned division, producing the quotient in Q and the remainder in R.

```
#include <stdio.h>
#include <assert.h>

#define W 16     /* maximum number of bits in the dividend & divisor */

unsigned short
divide( dividend, divisor )
    unsigned short dividend, divisor;
{
    long R;      /* partial remainder -- need 2*W bits */
    unsigned short Q;  /* partial quotient */
    int iter;

    R = dividend;
    Q = 0;
    for ( iter = W; iter >= 0; iter -= 1 ){
      assert( Q*divisor+R == dividend );
      if (R >= 0){
          R -= divisor <<iter;
          Q  += 1<<iter;
      } else {
          R += divisor <<iter;
          Q  -= 1<<iter;
      }
   }
   if ( R < 0 ){
```

```
    R += divisor;
    Q -= 1;
   }
   return Q;
}
```

Program 2

In the simple form shown above, this method has two drawbacks:

- It requires a 2*W-digit accumulator
- It always requires W steps.

Both these problems may be overcome by estimating the quotient before the actual division is carried out. This can cut the time required for a division from O(W) to $O(\log_B (\text{quotient}))$. Program 2 illustrates how this estimate may be used to reduce the number of divide steps required and the size of the accumulator.

```
#include <stdio.h>
#include <assert.h>

#define W 32 /* maximum number of bits in a divisor or dividend */

#define Big_value (unsigned)(1<<(W-2)) /* 2 ^ (W-1) */

int
estimate_log_quotient( dividend, divisor )
    unsigned dividend, divisor;
{
    unsigned log_quotient;

    for (log_quotient = 0; log_quotient < W; log_quotient += 1 )
      if ( ( divisor <<log_quotient) > Big_value )
          break;
      else if ( (divisor <<log_quotient) >= dividend )
          break;

    return log_quotient;
}

unsigned
divide( dividend, divisor )
    unsigned dividend, divisor;
{
    int  R;      /* remainder */
    unsigned Q;  /* quotient  */

    int iter;
    R = dividend;
    Q = 0;
    for ( iter = estimate_log_quotient( dividend, divisor); iter >= 0; iter -= 1 ){
      assert( Q*divisor+R == dividend );
      if (R >= 0){
          R -= divisor <<iter;
          Q  += 1<<iter;
```

```
        } else {
            R += divisor <<iter;
            Q  -= 1<<iter;
        }
    }
    if ( R < 0 ){
        R += divisor;
        Q -= 1;
    }
    return Q;
}
```

Program 3

Another way of reducing the number of division steps required is to choose a larger base, B'. This is only feasible if the cost of the radix-B' inner loop does not exceed the cost of the radix-B inner loop by more than $\log_B (B')$. When $B' = B^N$ for some integer N, a radix-B' inner loop can easily be constructed from the radix-B inner loop by arranging an N-high, B-ary decision tree. Programs 3 and 4 illustrate how this can be done. Program 3 uses N-level recursion to show the principle, but the overhead of recursion in this example far outweighs the loop overhead saved by reducing the number of steps required. Program 4 shows how run-time recursion can be eliminated if N is fixed at two.

```
#include <stdio.h>
#include <assert.h>

#define W 32 /* bits in a word */

int B,          /* number base of division (must be a power of 2) */
    N;          /* log2(B)*/
#define WB  (W/N)    /* base B digits in a word */
#define Big_value (unsigned)(B<<(WB-2)) /* B ^ (WB-1) */

int Q,    /* partial quotient */
    R,    /* partial remainder */
    V;    /* multiple of the divisor */
int
estimate_log_quotient( dividend, divisor )
    unsigned dividend, divisor;
{
    unsigned log_quotient;

    for (log_quotient = 0; log_quotient < WB; log_quotient += 1 )
      if ( ( divisor <<log_quotient*N) > Big_value )
          break;
      else if ( (divisor <<log_quotient*N) >= dividend )
          break;

    return log_quotient;
}


int
compute_digit( level, quotient_digit)
      int level, quotient_digit;
```

```
{
      if (R >= 0){
           R -= V << level;
           quotient_digit += 1<<level;
      } else {
           R += V << level;
           quotient_digit -= 1<<level;
      }
      if (level > 0)
           return compute_digit( level-1, quotient_digit );
      else
                return quotient_digit;
}

unsigned
divide( dividend, divisor )
    unsigned dividend, divisor;
{

    int iter;

    B = (1<<(N));
    R = dividend;
    Q = 0;
    for ( iter = estimate_log_quotient( dividend, divisor); iter >= 0; iter -= 1 ){
      assert( Q*divisor+R == dividend );
      V = divisor << (iter*N);
      Q += compute_digit( N-1, 0) << (iter*N);
    }
    if ( R < 0 ){
      R += divisor;
      Q -= 1;
    }
    return Q;
}
```

Program 4

```
#include <stdio.h>
#include <assert.h>

#define W 32     /* bits in a word */

#define B 4      /* number base of division (must be a power of 2) */
#define N 2      /* log2(B)*/
#define WB  (W/N)     /* base B digits in a word */
#define Big_value (unsigned)(B<<(WB-2)) /* B ^ WB-1 */

int
estimate_log_quotient( dividend, divisor )
    unsigned dividend, divisor;
{
    unsigned log_quotient;

    for (log_quotient = 0; log_quotient < WB; log_quotient += 1 )
      if ( ( divisor <<log_quotient*N) > Big_value )
```

```
            break;
        else if ( (divisor <<log_quotient*N) >= dividend )
            break;

    return log_quotient;
}

int
unsigned
divide( dividend, divisor )
    unsigned dividend, divisor;
{

int Q,    /* partial quotient */
    R,    /* partial remainder */
    V;    /* multiple of the divisor */
    int iter;

    R = dividend;
    Q = 0;
    for ( iter = estimate_log_quotient( dividend, divisor); iter >= 0; iter -= 1 ){
      assert( Q*divisor+R == dividend );
      V = divisor << (iter*N);
      /* N-deep, B-wide decision tree */
      if ( R >= 0 ) {
           R -= V<<1;
           if ( R >= 0 ){
                R -= V;
                Q += 3 <<(N*iter);
      } else {
                R += V;
                Q += 1 <<(N*iter);
                }
      } else {
           R += V<<1;
           if ( R >= 0 ){
                R -= V;
                Q -= 1 <<(N*iter);
           } else {
                R += V;
                Q -= 3 <<(N*iter);
                }
      }
    }
    if ( R < 0 ){
      R += divisor;
      Q -= 1;
    }
    return Q;
}
```

Program 5

At the risk of losing even more clarity, we can optimize away several of the bookkeeping operations, as shown in Program 5.

```
#include <stdio.h>
#include <assert.h>

#define W 32     /* bits in a word */

#define B 4      /* number base of division (must be a power of 2) */
#define N 2      /* log2 (B)*/
#define WB  (W/N)     /* base B digits in a word */
#define Big_value (unsigned)(B<<(WB-2)) /* B ^ WB-1 */

int
unsigned
divide( dividend, divisor )
    unsigned dividend, divisor;
{

    int Q,    /* partial quotient */
        R,    /* partial remainder */
        V;    /* multiple of the divisor */
    int iter;

    R = dividend;
    Q = 0;
    V = divisor;
    for ( iter = 0; V <= Big_value && V <= dividend; iter += 1 )
      V <<= N;

    for ( V <<= (N-1); iter >= 0; iter -= 1 ){
      Q <<= N;
      assert( Q*(1<<(iter*N))*divisor+R == dividend );
      /* N-deep, B-wide decision tree */
      if ( R >= 0 ){
            R -= V;
            V >>= 1;
            if ( R >= 0 ){
                  R -= V;
                  V >>= 1;
                  Q += 3 ;
            } else {
                  R += V;
                  V >>= 1;
                  Q += 1 ;
            }
    } else {
            R += V;
            V >>= 1;
            if ( R >= 0 ){
                  R -= V;
                  V >>= 1;
                  Q -= 1;
            } else {
                  R += V;
                  V >>= 1;
                  Q -= 3;
            }
      }
    }
```

```
    if ( R < 0 ){
      R += divisor;
      Q -= 1;
    }
    return Q;
}
```

Program 6

Program 6 is, essentially, the method we recommend for SPARC. The depth of the decision tree—two in the preceding examples—is controlled by the constant N, and is currently set to three, based on empirical evidence. The decision tree is not explicitly coded, but defined by the recursive m4 macro DEVELOP_QUOTIENT_BITS. Other differences include:

- Handling of signed and unsigned operands
- More care is taken to avoid overflow for very large quotients or divisors
- Special tests are made for division by zero and zero quotient
- The routine is conditionally compiled for either division or remaindering.

```
/*
 * Divison/Remainder
 *
 * Input is:
 *     dividend -- the thing being divided
 *     divisor  -- how many ways to divide it
 * Important parameters:
 *     N -- how many bits per iteration we try to get
 *          as our current guess: define(N, 3)
 *     WORDSIZE -- how many bits altogether we're talking about:
 *          obviously: define(WORDSIZE, 32)
 * A derived constant:
 *     TOPBITS -- how many bits are in the top "decade" of a number:
 *          define(TOPBITS, eval( WORDSIZE - N*((WORDSIZE-1)/N) ) )
 * Important variables are:
 *     Q -- the partial quotient under development -- initally 0
 *     R -- the remainder so far -- initially == the dividend
 *     ITER -- number of iterations of the main division loop which will
 *          be required. Equal to CEIL( lg2(quotient)/N )
 *          Note that this is log_base_(2^N) of the quotient.
 *     V -- the current comparand -- initially divisor*2^(ITER*N-1)
 * Cost:
 *     current estimate for non-large dividend is
 *          CEIL( lg2(quotient) / N ) x ( 10 + 7N/2 ) + C
 *     a large dividend is one greater than 2^(31-TOPBITS) and takes a
 *     different path, as the upper bits of the quotient must be developed
 *     one bit at a time.
 *     This uses the m4 and cpp macro preprocessors.
 */

#include "sw_trap.h"

define(dividend, `%i0')
define(divisor, `%i1')
define(Q, `%i2')
define(R, `%i3')
```

```
define(ITER,    `%l0')
define(V, `%l1')
define(SIGN,    `%l2')
define(T, `%l3')     ! working variable
define(SC,`%l4')

/*
 * This is the recursive definition of how we develop quotient digits.
 * It takes three important parameters:
 *     $1 -- the current depth, 1<=$1<=N
 *     $2 -- the current accumulation of quotient bits
 *     N  -- max depth
 * We add a new bit to $2 and either recurse or insert the bits in the quotient.
 * Dynamic input:
 *     R -- current remainder
 *     Q -- current quotient
 *     V -- current comparand
 *     cc -- set on current value of R
 * Dynamic output:
 *     R', Q', V', cc'
 */

define(DEVELOP_QUOTIENT_BITS,
       !depth $1, accumulated bits $2
       bl   L.$1.eval(2^N+$2)
       srl  V,1,V
       ! remainder is positive
       subcc R,V,R
       ifelse( $1, N,
       `     b    9f
             add  Q, ($2*2+1), Q
       ',`   DEVELOP_QUOTIENT_BITS( incr($1), `eval(2*$2+1)')
       ')
L.$1.eval(2^N+$2):                  ! remainder is negative
       addcc R,V,R
       ifelse( $1, N,
       `     b    9f
             add  Q, ($2*2-1), Q
       ',`   DEVELOP_QUOTIENT_BITS( incr($1), `eval(2*$2-1)')
       ')
       ifelse( $1, 1, `9:')
')
ifelse( ANSWER, `quotient', `
       .global   .div, .udiv
.udiv:      ! UNSIGNED DIVIDE
       save %sp,-64,%sp
       b    divide
       mov  0,SIGN                  ! result always positive

.div: ! SIGNED DIVIDE
       save %sp,-64,%sp
       orcc divisor,dividend,%g0     ! are either dividend or divisor negative
       bge  divide                  ! if not, skip this junk
       xor  divisor,dividend,SIGN     ! record sign of result in sign of SIGN
       tst  divisor
       bge  2f
       tst  dividend
       !    divisor < 0
```

```
      bge  divide
      neg  divisor
      2:
      !    dividend < 0
      neg  dividend
      !    FALL THROUGH
',`
      .global   .rem, .urem
.urem:      ! UNSIGNED REMAINDER
      save %sp,-64,%sp              ! do this for debugging
      b    divide
      mov  0,SIGN                   ! result always positive

.rem: ! SIGNED REMAINDER
      save %sp,-64,%sp              ! do this for debugging
      orcc divisor,dividend,%g0      ! are either dividend or divisor negative
      bge  divide                   ! if not, skip this junk
      mov  dividend,SIGN            ! record sign of result in sign of SIGN
      tst  divisor
      bge  2f
      tst  dividend
      !    divisor < 0
      bge  divide
      neg  divisor
      2:
      !    dividend < 0
      neg  dividend
      !    FALL THROUGH
')

divide:
!     compute size of quotient, scale comparand
      orcc divisor,%g0,V  ! movcc   divisor,V
      te   ST_DIV0 ! if divisor = 0
      mov  dividend,R
      mov  0,Q
      sethi %hi(1<<(WORDSIZE-TOPBITS-1)),T
      cmp  R,T
      blu  not_really_big
      mov  0,ITER
      !
      ! Here, the dividend is >= 2^(31-N) or so. We must be careful here, as
      ! our usual N-at-a-shot divide step will cause overflow and havoc. The
      ! total number of bits in the result here is N*ITER+SC, where SC <= N.
      ! Compute ITER, in an unorthodox manner: know we need to Shift V into
      !     the top decade: so don't even bother to compare to R.
      1:
           cmp  V,T
           bgeu 3f
           mov  1,SC
           sll  V,N,V
           b    1b
           inc  ITER
      !
      ! Now compute SC
      !
      2:   addcc V,V,V
           bcc  not_too_big ! bcc    not_too_big
```

```
            add  SC,1,SC
                 !
                 ! We're here if the divisor overflowed when Shifting.
                 ! This means that R has the high-order bit set.
                 ! Restore V and subtract from R.
                 sll  T,TOPBITS,T ! high order bit
                 srl  V,1,V ! rest of V
                 add  V,T,V
                 b    do_single_div
                 dec  SC
       not_too_big:
       3:   cmp  V,R
            blu  2b
            nop
            be   do_single_div
            nop
       ! V > R: went too far: back up 1 step
       !    srl  V,1,V
       !    dec  SC
       ! do single-bit divide steps
       !
       ! We have to be careful here. We know that R >= V, so we can do the
       ! first divide step without thinking. BUT, the others are conditional,
       ! and are only done if R >= 0. Because both R and V may have the high-
       ! order bit set in the first step, just falling into the regular
       ! division loop will mess up the first time around.
       ! So we unroll slightly...
       do_single_div:
            deccc SC
            bl   end_regular_divide
            nop
            sub  R,V,R
            mov  1,Q
            b    end_single_divloop
            nop
       single_divloop:
            sll  Q,1,Q
            bl   1f
            srl  V,1,V
            ! R >= 0
            sub  R,V,R
            b    2f
            inc  Q
       1:   ! R < 0
            add  R,V,R
            dec  Q
       2:
       end_single_divloop:
            deccc SC
            bge  single_divloop
            tst  R
            b    end_regular_divide
            nop

  not_really_big:
  1:
       sll  V,N,V
       cmp  V,R
```

```
      bleu 1b
      inccc ITER
      be   got_result
      dec  ITER
do_regular_divide:

      ! Do the main division iteration
      tst  R
      ! Fall through into divide loop
divloop:
      sll  Q,N,Q
      DEVELOP_QUOTIENT_BITS( 1, 0 )
end_regular_divide:
      deccc ITER
      bge  divloop
      tst  R
      bge  got_result
      nop
      ! non-restoring fixup here
ifelse( ANSWER, `quotient',
`     dec  Q
',`   add  R,divisor,R
')

got_result:
      tst  SIGN
      bge  1f
      restore
      ! answer < 0
      retl          ! leaf-routine return
ifelse( ANSWER, `quotient',
`     neg  %o2,%o0  ! quotient  <- -Q
',`   neg  %o3,%o0  ! remainder <- -R
')
1:    retl          ! leaf-routine return
ifelse( ANSWER, `quotient',
`     mov  %o2,%o0  ! quotient  <-  Q
',`   mov  %o3,%o0  ! remainder <-  R
')
```

APPENDIX F

Opcodes and Condition Codes

This appendix lists the opcodes and condition codes.

op	Instruction
01	CALL

Table F–1. Format 1 Opcodes

op2	Instruction
000	UNIMP
001	*Unimplemented*
010	Bicc
011	*Unimplemented*
100	SETHI
101	*Unimplemented*
110	FBfcc
111	CBccc

Table F–2. Format 2 Opcodes (op = 00)

op3	Instruction
000000	ADD
000001	AND
000010	OR
000011	XOR
000100	SUB
000101	ANDN
000110	ORN
000111	XNOR
001000	ADDX
001001	*Unimplemented*
001010	*Unimplemented*
001011	*Unimplemented*
001100	SUBX
001101	*Unimplemented*
001110	*Unimplemented*
001111	*Unimplemented*
010000	ADDcc
010001	ANDcc
010010	ORcc
010011	XORcc
010100	SUBcc
010101	ANDNcc
010110	ORNcc
010111	XNORcc
011000	ADDXcc
011001	*Unimplemented*
011010	*Unimplemented*
011011	*Unimplemented*
011100	SUBXcc
011101	*Unimplemented*
011110	*Unimplemented*
011111	*Unimplemented*

Table F–3. Format 3 Opcodes (op = 10, op3 = 0nnnnn)

op3	Instruction
100000	TADDcc
100001	TSUBcc
100010	TADDccTV
100011	TSUBccTV
100100	MULScc
100101	SLL
100110	SRL
100111	SRA
101000	RDY
101001	RDPSR
101010	RDWIM
101011	RDTBR
101100	*Unimplemented*
101101	*Unimplemented*
101110	*Unimplemented*
101111	*Unimplemented*
110000	WRY
110001	WRPSR
110010	WRWIM
110011	WRTBR
110100	FPop1
110101	FPop2
110110	CPop1
110111	CPop2
111000	JMPL
111001	RETT
111010	Ticc
111011	IFLUSH
111100	SAVE
111101	RESTORE
111110	*Unimplemented*
111111	*Unimplemented*

Table F–4. Format 3 Opcodes (op = 10, op3 = 1nnnnn)

op3	Instruction
000000	LD
000001	LDUB
000010	LDU
000011	LDD
000100	ST
000101	STB
000110	ST
000111	STD
001000	*Unimplemented*
001001	LDSB
001010	LDSH
001011	*Unimplemented*
001100	*Unimplemented*
001101	LDSTUB
001110	*Unimplemented*
001111	SWAP
010000	LDA
010001	LDUBA
010010	LDUHA
010011	LDDA
010100	STA
010101	STBA
010110	STHA
010111	STDA
011000	*Unimplemented*
011001	LDSBA
011010	LDSHA
011011	*Unimplemented*
011100	*Unimplemented*
011101	LDSTUBA
011110	*Unimplemented*
011111	SWAPA

Table F–5. Format 3 Opcodes (op = 11, op3 = 0nnnnn)

op3	Instruction
100000	LDF
100001	LDFSR
100010	*Unimplemented*
100011	LDDF
100100	STF
100101	STFSR
100110	STDFQ
100111	STDF
101000—101111	*Unimplemented*
110000	LDC
110001	LDCSR
110010	*Unimplemented*
110011	LDDC
110100	STC
110101	STCSR
110110	STDCQ
110111	STDC
111000—111111	*Unimplemented*

Table F–6. Format 3 Opcodes (op = 11, op3 = 1n0nnn)

opf	Instruction
000000001	FMOVs
000000101	FNEGs
000001001	FABSs
000101001	FSQRTs
000101010	FSQRTd
000101011	FSQRTx
001000001	FADDs
001000010	FADDd
001000011	FADDx
001000101	FSUBs
001000110	FSUBd
001000111	FSUBx
001001001	FMULs
001001010	FMULd
001001011	FMULx
001001101	FDIVs
001001110	FDIVd
001001111	FDIVx
011000001	FsTOiR
011000010	FdTOiR
011000011	FxTOiR
011000100	FiTOs
011000110	FdTOs
011000111	FxTOs
011001000	FiTOd
011001001	FsTOd
011001011	FxTOd
011001100	FiTOx
011001101	FsTOx
011001110	FdTOx
011010001	FsTOi
011010010	FdTOi
011010011	FxTOi

Table F–7. FPop Opcodes (op = 11, op3 = 110100)

opf	Instruction
001010001	FCMPs
001010010	FCMPd
001010011	FCMPx
001010101	FCMPEs
001010110	FCMPEd
001010111	FCMPEx

Table F–8. FPop Opcodes (op = 11, op3 = 110101)

cond	Test
0000	Never
0001	Equal
0010	Less than or equal
0011	Less than
0100	Less than or equal, unsigned
0101	Carry set (less than, unsigned)
0110	Negative
0111	Overflow set
1000	Always
1001	Not equal
1010	Greater than
1011	Greater than or equal
1100	Greater than, unsigned
1101	Carry clear (greater than or equal, unsigned)
1110	Positive
1111	Overflow clear

Table F–9. Bicc and Ticc Conditon Codes

cond	Test
0000	Never
0001	Not equal
0010	Less than or greater than
0011	Unordered or less than
0100	Less than
0101	Unordered or greater than
0110	Greater than
0111	Unordered
1000	Always
1001	Equal
1010	Unordered or equal
1011	Greater than or equal
1100	Unordered or greater than or equal
1101	Less than or equal
1110	Unordered or less than or equal
1111	Ordered

Table F–10. FBfcc Condition Codes

opcode	cond	by_CP_cc[1:0] test
CBN	0000	Never
CB123	0001	1 or 2 or 3
CB12	0010	1 or 2
CB13	0011	1 or 3
CB1	0100	1
CB23	0101	2 or 3
CB2	0110	2
CB3	0111	3
CBA	1000	Always
CB0	1001	0
CB03	1010	0 or 3
CB02	1011	0 or 2
CB023	1100	0 or 2 or 3
CB01	1101	0 or 1
CB013	1110	0 or 1 or 3
CB012	1111	0 or 1 or 2

Table F–11. CBccc Condition Codes

CYPRESS SEMICONDUCTOR CORPORATION
3901 NORTH FIRST STREET
SAN JOSE, CALIFORNIA 95134
TELEPHONE (408) 943-2600