Am29030™ and Am29035™ Microprocessors
User's Manual and Data Sheet

Advanced
Micro
Devices

Am29030™ and Am29035™ Microprocessors User's Manual and Data Sheet

ADVANCED MICRO DEVICES

This Manual was written and edited by Scott McMahon and Mike Johnson.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION AND OVERVIEW

THE Am29030™ AND Am29035™ MICROPROCESSORS

The Am29030 and Am29035 microprocessors are the first in a new series of 32-bit, streamlined instruction processors that employ submicron circuits to provide high performance even with low-cost system components. High circuit densities and a high degree of on-chip integration enable the Am29030 and Am29035 microprocessors to operate at high frequencies while providing a streamlined interface that simplifies system design.

The Am29030 and Am29035 microprocessors were designed expressly to meet the requirements of embedded applications such as laser beam printers, graphics processing, application program interface (API) accelerators, X terminals and servers, and scanners. Such applications make the following four demands on system design:

- High performance at low cost: A high-frequency processor must interface with low-cost memory without degrading processor performance. The system must provide excellent real-time response at low cost.
- Design flexibility: One basic design must establish an entire product line.
- Reduced time-to-market: A complete suite of development, debug, and benchmarking tools is critical for reducing the product development cycle.
- A rational, easy upgrade path: The processor family must provide bus-, pin-, and software-compatibility so that processor upgrades are transparent to both hardware and software. In addition, the processor's system interface must accommodate easy memory upgrades in convenient (0.5-MB) increments.

The Am29030 and Am29035 processors are, in fact, highly optimized for any embedded application that requires high performance at low cost. In addition to graphics and imaging applications, the processors are ideal for use in network applications such as bridges and node processors for fiber optic (FDDI) networks.

DESIGN PHILOSOPHY

The Am29030 and Am29035 processors are the result of a design philosophy that recognizes that processor performance must be considered in light of the processor's hardware and software environment. The key to maximizing performance lies in the realization that the processor is part of an integrated system, and is itself a collection of components that must be properly integrated.

Processor features must be considered not only on their own merits, but also in relation to other components of the system. A particular feature that—considered alone—increases one aspect of processor performance may actually decrease the performance of the total system, because of the burden that it places elsewhere in the system. As an illustration, consider the factors involved in the execution time of any processor task:

TASK TIME = (INSTRUCTIONS / TASK) * (CYCLES / INSTRUCTION) * (TIME / CYCLE)

To minimize the time taken, it is necessary to minimize the above product. This is not equivalent to minimizing all of the terms that contribute to the product; in fact, this is generally not possible due to the interaction of the terms.

As an example of the interaction of the above terms, consider the number of instructions required for a task. An attempt to minimize this number, a more or less traditional approach to processor architecture design, increases the average number of cycles required for the execution of an instruction, because of the increased number of operations performed by each instruction. In addition, cycle time is increased because of instruction-decode time.

A second example of the interaction in the above equation appears in an attempt to reduce the cycle time through the pipelining of operations. In theory, the cycle time can be made arbitrarily small by the definition of an arbitrarily large number of pipeline stages. In practice—at least in the case of general-purpose processors—pipelining rarely yields much of its potential benefit. This is due to situations where the pipeline cannot be kept fully occupied, such as when memory references and branches occur. In these situations, additional pipeline stages increase the number of cycles required for an operation, and thus affect the CYCLES / INSTRUCTION term.

OPTIMUM PERFORMANCE

Each of the terms in the above equation has some minimum bound for a given implementation technology and task. In general, this minimum bound cannot be approached without an offsetting increase in the other terms, making the overall product less-than-optimum. The question then arises, what combination of terms will yield an optimum product? There are several things to note when answering this question.

The first observation is that the number of operations underlying a given task is more or less fixed. Any single processor ultimately limits the time required for a task because it has a single execution unit and a single instruction stream. The operations that must be performed are reflected in the INSTRUCTIONS / TASK and CYCLES / INSTRUCTION terms. These operations may be performed by relatively few instructions, where each instruction takes multiple cycles to execute, or by a larger number of instructions, where each takes a single cycle to execute. In the first case, the instructions are complex; in the second, they are simple.

The point is that the trade-off between simple and complex instructions is not one-to-one. For example, reducing the number of cycles per instruction by a factor of three does not increase the number of instructions per task by the same factor. There are two reasons for this. The first is that, even when an instruction set supports complex operations, a large proportion of the instructions that are executed perform operations that could be performed as well by simple instructions. The second is that simple instructions expose more of the internal processor operation to an optimizing compiler. This allows the compiler to tailor the organization and sequence of operations to the task at hand, thereby reducing the total number of instructions executed.

PERFORMANCE LEVERAGE

Another important observation is that there is a tremendous amount of leverage in the TIME / CYCLE and CYCLES / INSTRUCTION terms. As they are made smaller, they have a proportionately greater effect on performance.

For example, a reduction of 10 ns in the cycle time of a processor operating with a 200-ns cycle time yields an increase of 5% in the processor's performance. The same improvement in a processor operating with a 50-ns cycle time yields a 20% increase in performance.

Correspondingly, a reduction of 0.2 in the number of cycles per instruction in a processor that averages 5 cycles per instruction yields a 4% increase in performance. However, the same reduction yields a 12.5% performance increase in a processor that averages 1.6 cycles per instruction.

CONCLUSION

The conclusion is that it is possible—and desirable—to yield somewhat in the number of instructions executed for a given task, and more than make up for the performance impact of this increase by reductions in the cycle time and in the number of cycles per instruction. For example, if both the cycle time and the number of cycles per instruction are reduced by a factor of three, while the number of instructions for a given task is allowed to grow by 50%, the resulting task time is reduced by a factor of six.

The Am29030 and Am29035 microprocessor architectures were designed with the above effects in mind. Maximum performance is obtained by the optimization of the product of the number of instructions per task, the number of cycles per instruction, and the cycle time, not by minimizing one factor at the expense of the others. This is accomplished by careful definition of all processor components. In particular:

1. The INSTRUCTION/TASK term is optimized by the definition of simple instructions. The processor provides an efficient instruction set and a large number of general-purpose registers to an optimizing, high-level language compiler. Most reductions in this term are accomplished by the compiler. The number of instructions for a given task may be greater than the number of instructions for processors with complex instruction sets. However, this increase is more than offset by other improvements in processor performance.
2. The CYCLES/INSTRUCTION term is optimized by the data-flow structure and performance-enhancing features of the processor. A large amount of processor hardware is dedicated to achieving an average instruction-execution rate that is close to single-cycle execution.
3. The TIME/CYCLE term is optimized by the implementation technology, the processor system interface, and judicious use of pipelining. The simplicity of the instruction set and processor features helps minimize the cycle time.

PURPOSE OF THIS MANUAL

This manual describes the technical features, programming interface, and complete instruction set of the Am29030 and Am29035 microprocessors.

INTENDED AUDIENCE

This manual is intended for computer hardware and software architects and system engineers who are designing or are considering designing systems based on the Am29030 and Am29035 microprocessors.

Am29030 and Am29035 MICROPROCESSORS USER'S MANUAL OVERVIEW

This manual contains information on the Am29030 and Am29035 microprocessors that is essential for system hardware and software architects and design engineers. Additional information is available in the form of data sheets, application notes, and other documentation that is provided with software products and hardware-development tools.

The information in this manual is organized into twelve chapters:

Chapter 1 introduces the features and performance aspects of the Am29030 and Am29035 microprocessors.

Chapter 2 describes the programmer's model of the Am29030 and Am29035 microprocessors, including the instruction set and register model.

Chapter 3 expands on the programmer's model, discussing different data formats and handling. Instructions that manipulate external data are also discussed.

Chapter 4 details the management of the run-time stack and defines the conventions that apply to procedure linkage and register usage.

Chapter 5 describes the internal pipelining and the effects of the pipeline on program behavior.

Chapter 6 describes the system protection features provided by the Am29030 and Am29035 microprocessors.

Chapter 7 describes the memory management features of the Am29030 and Am29035 microprocessors.

Chapter 8 provides a description of the interrupt and trap mechanism and details the handling of interrupts and traps.

Chapter 9 describes the operation of the instruction cache.

Chapter 10 details the system interface of the Am29030 and Am29035 processors.

Chapter 11 describes the software and hardware facilities for debugging and testing.

Chapter 12 provides a detailed description of the instruction set.

For those readers desiring only a brief overview of the Am29030 and Am29035 microprocessors, Chapter 1 identifies the outstanding features of the processors. This chapter addresses the basic software and hardware concerns. Chapters 2, 3, and 5 are recommended reading for all developers, both hardware and software.

For software architects and system programmers interested mainly in software-related issues, Chapters 4, 6, 7, 8 and Section 10.2 provide the necessary information. Chapter 9 describes software issues related to the instruction cache. Chapters 11 and 12 provide related information.

For hardware architects and systems hardware designers interested mainly in hardware-related issues, Chapters 10, 11 and Appendix A provide most of the required information; Chapters 5, 9, and 12 also provide related information.

29K FAMILY DOCUMENTATION

ORDER NO.	TITLE
10620	**29K User's Manual** Describes the Am29000™ microprocessor's technical features, programming interface, and complete instruction set.
11011	**29K Graphics Primitives Handbook** Describes a set of graphics functions written in C-callable assembly language. This is an excellent introduction/tutorial for graphics programming on the Am29000 microprocessor.
11426	**Fusion29K℠ Catalog** Provides information on more than 100 tools that speed a 29K Family embedded product to market. Includes products from over 50 expert suppliers of embedded development solutions. Design solution chapters include: laser printer and OCR solutions, graphics solutions, and networking solutions.
12990	**Fusion29K Newletter** Contains quarterly updates on developments in the 29K Family.
14779	**Am29050™ User's Manual** Describes the Am29050 microprocessor's technical features, programming interface, and complete instruction set.
15176	**29K Laser Printer Solutions Brochure** Reviews how the 29K Family of microprocessors fits into the laser printer marketplace. Includes a description of AMD's PCL and Postscript® Laser29K™ Low-Cost Raster Image Processor demonstration boards.
12175	**29K Family Data Book** A comprehensive collection of data sheets for the Am29000 microprocessor, Am29027™ arithmetic accelerator, High C 29K™ Cross Development Toolkit, and XRAY29K™ Source-Level Debugger. It also includes application notes to help shorten designers learning curves and hardware and software development time.
10626	**XRAY29K Data Sheet**
10957	**High C 29K Data Sheet**
13089	**Am29005™ Data Sheet**
14721	**EB29K Data Sheet**
15039	**Am29050 Data Sheet**
11539	**Host Interface (HIF) v2.0 Specification**

To order literature, contact your local AMD sales office or call: 800–2929–AMD Ext.3 (in the U.S.), or 800–531–5202 Ext. 55651 (in Canada), or direct dial from any location: 512–462–5651.

RELATED PUBLICATIONS

The IEEE Std. 1149.1–1990 (JTAG) may be ordered from:

IEEE Computer Society Press
Customer Service Center
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720–1264
USA

IEEE Catalogue No. SH13144
1–800–CS–BOOKS
714–821–4010 (FAX)

CHAPTER 1

FEATURES AND PERFORMANCE

This chapter provides an evaluation of the Am29030 and Am29035 microprocessors as an aid in considering a particular application. A detailed technical description of these microprocessors is contained in subsequent chapters. This chapter informally describes the features of the processors, concentrating on features which distinguish the Am29030 and Am29035 microprocessors from other available processors and how these features enhance system performance and cost-effectiveness. This chapter consists of the following sections:

- Distinctive Characteristics
- Key Features and Benefits
- Performance Overview
- Debugging and Testing

1.1 DISTINCTIVE CHARACTERISTICS

1.1.1 Am29030 Microprocessor

- Designed for printer, imaging, graphics, and other embedded applications
- Full 32-bit architecture
- CMOS technology/TTL-compatible
- 8 Kbyte, two-way-set-associative Instruction Cache
- Scalable Clocking™ Technology
- Operational frequencies of 25 and 33 MHz
- 26 million instructions per second sustained at a 33 MHz operating frequency
- 1.26 clock cycles per instruction average
- 4-GB virtual address space
- 192 general-purpose registers
- Three-address instruction architecture
- Streamlined system interface for simplified high-frequency operation
- Burst-mode and page-mode access support
- 8,16, or 32-bit ROM interface
- 64-entry Memory Management Unit on-chip
- Demand paging
- Fully pipelined
- On-chip Timer Facility
- Enhanced debugging support

- Master/slave chip output checking
- IEEE Std.1149.1-1990 (JTAG) compliant Standard Test Access Port and Boundary-Scan Architecture implementation

1.1.2 Am29035 Microprocessor

The Am29035 microprocessor is very similar to the Am29030 microprocessor, with the following exceptions:

- Operational frequency of 16 MHz
- 12 million instructions per second sustained at a 16 MHz operating frequency
- Programmable 16 or 32-bit data bus width
- 4 Kbyte, direct-mapped Instruction Cache

1.1.3 Feature Summary

The following table compares the features of the Am29030 and Am29035 microprocessors:

Feature	Am29035	Am29030
Input Clock (MHz)	16	25, 33
Cache Size	4K-bytes (Direct Mapped)	8K-bytes (2-Way Set Associative)
Scalable Clocking	Yes	Yes
Narrow Read	Yes	Yes
Programmable Bus Sizing	Yes	No
Package	144-pin QFP	145-pin PGA

1.2 KEY FEATURES AND BENEFITS

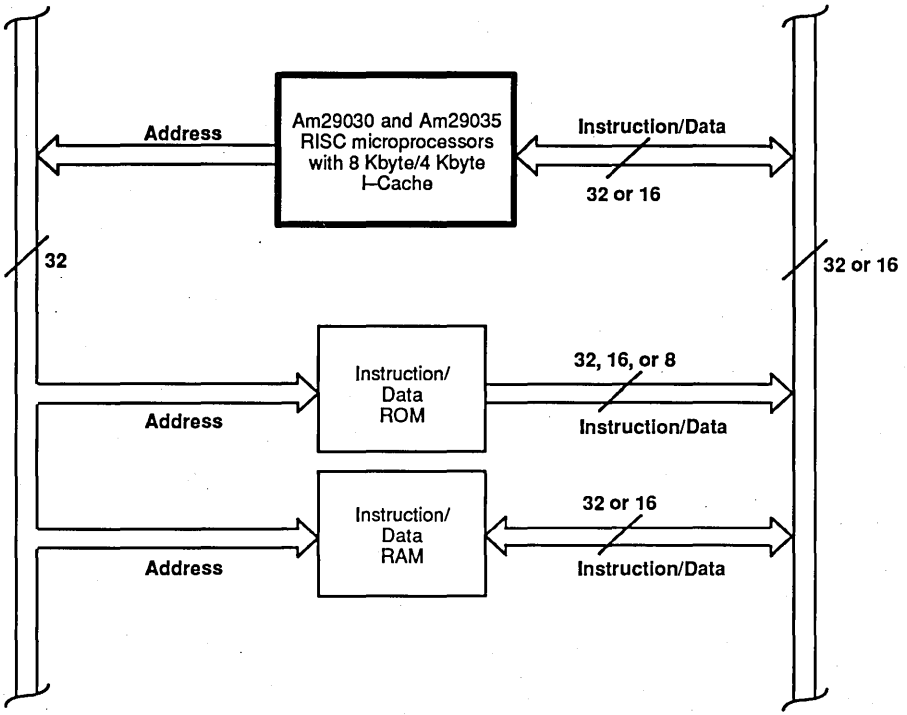
The Am29030 and Am29035 RISC microprocessors are high-performance, general-purpose, 32-bit microprocessors implemented in complementary metal-oxide semiconductor (CMOS) technology. They are targeted primarily at printer, imaging and graphics applications, using a flexible architecture, a high-bandwidth memory interface, and rapid execution of simple instructions which are common in embedded applications.

The Am29030 and Am29035 microprocessors also position the 29K architecture to enter the realm of submicron technology which is characterized by very high circuit densities and operating frequencies.

The Am29030 and Am29035 microprocessors are fully software-compatible with the Am29000, Am29005 and Am29050 microprocessors. They can be used in most existing Am29000 microprocessor applications without software modifications.

A representative system diagram for the Am29030 and Am29035 microprocessors is shown in Figure 1-1.

Figure 1-1 Simplified System Diagram



1.2.1 Large, On-Chip Instruction Cache

The use of submicron circuitry allows the integration of a large on-chip instruction cache. The Am29030 microprocessor has an 8K-byte two-way-set-associative instruction cache, and the Am29035 microprocessor has a 4K-byte direct-mapped instruction cache. A large instruction cache provides very high cache hit rates, which reduces the average number of cycles per instruction by minimizing the effect of memory latency. This is a key feature, since it allows designers to use low-cost memory that requires a simpler (and therefore less costly) memory design.

The large, on-chip instruction cache also plays a major role in providing a streamlined system interface, as described in Section 1.2.5

1.2.2 Scalable Clocking Technology

A feature unique to the Am29030 and Am29035 microprocessors is Scalable Clocking technology. Scalable Clocking technology is comprised of several features which aid in the design of low-cost high frequency designs.

A primary feature of Scalable Clocking technology is the ability of the processor to drive the memory system at the same speed as the processor or at half processor speed. This capability provides substantial benefits. First, the half-speed mode allows the use of slower, lower-cost memory without significant degradation of processor

performance. For example, a 33-MHz processor could be combined with a 20-MHz memory system with only a slight loss in performance. Another advantage is that system performance can be upgraded by simply replacing the processor with a higher-speed processor. For example, a processor may be replaced with a faster processor while utilizing the existing memory system, running at half-speed if necessary.

Another feature of Scalable Clocking technology is the processor runs at the frequency of the oscillator input. The Am29030 and Am29035 processors do not require a double-frequency oscillator to generate internal clocks. Relaxed duty cycle restrictions allow the processor to directly use oscillators with duty cycles of 30/70 to 70/30.

High frequency operation is further simplified through the use of a hardwired wait state which is enforced during the initial cycle of all simple data accesses and the initial cycle of a burst-mode access. The main benefit of this approach is that the address and data pins are not required to change state during the same cycle. This reduces electrical noise and therefore aids in high-frequency designs.

Finally, Scalable Clocking technology encompasses relaxed timing specifications. This reduces the cost and complexity of external system design.

1.2.3 Narrow Read Interface

Both the Am29030 and Am29035 microprocessors can be connected to 8-, 16-, and 32-bit memories. If the data sized accessed is larger than that supported by memory, the processor automatically generates the necessary sequencing to perform multiple reads.

This ability to perform narrow reads is particularly useful for a ROM interface. Using narrow reads, the processor can execute a bootstrap program from a small boot ROM. Such a bootstrap program would most likely download the application program into RAM. This not only allows the use of low-cost ROMs, it also conserves board space and allows easy revision of application code.

1.2.4 Programmable Bus Sizing

The Am29035 processor's Instruction/Data bus can be dynamically programmed to be either 16 or 32 bits wide for data transfers. This enables the Am29035 microprocessor to write either 16-bit or 32-bit devices. The processor automatically performs multiple 16-bit writes when writing more than 16 bits.

This unique feature, called Programmable Bus Sizing, provides a flexible interface to low-cost memory, as well a convenient, flexible upgrade path. For example, a system can start with a 16-bit memory design and can subsequently improve performance by migrating to a 32-bit memory design. Of particular advantage is the ability to add memory in half-megabyte increments. This provides significant cost savings for applications that do not require larger memory upgrades.

1.2.5 Streamlined System Interface

The high level of integration achieved in the Am29030 and Am29035 microprocessors allows a large on-chip instruction cache to be implemented, which in turn enables the system interface to be streamlined. The addition of the instruction cache reduces the external instruction bandwidth requirements, and therefore relaxes the requirements of the system interface.

The initial processors of the 29K family have a Branch Target Cache™ memory of 512 or 1 Kbyte. This structure was used due to the limited level of integration permitted by process technology at that time. Since the Branch Target Cache (BTC) memory caches the initial instruction sequence of non-sequential instruction fetches, only the initial access latency of the memory system is reduced and therefore the processor must provide sufficient instruction bandwidth. In the initial 29K processors, this meant having a separate instruction and data bus to allow concurrent instruction and data accesses.

Research[1] has demonstrated that, at cache sizes below approximately 4 Kbytes, a BTC is more cost effective than a conventional instruction cache. At larger cache sizes, a conventional cache provides performance superior to the BTC's and is able to maintain sufficient instruction bandwidth without a dedicated instruction bus.

The large on-chip instruction cache of the Am29030 and Am29035 microprocessors satisfy the instruction bandwidth requirements and therefore remove the need for separate instruction and data buses. The Am29030 and Am29035 microprocessors employ a streamlined, 2-bus external interface, which comprises an address bus and an instruction/data bus. This allows the use of lower-performance and lower-cost memory, provides a reduction in the memory-system parts count, and reduces the board area required for the memory system. In addition, the simplified design requirements reduce development costs.

1.2.6 Pin-, Bus-, and Software-Compatibility

Compatibility within a processor family is critical for achieving a rational, easy upgrade path. The Am29030 and Am29035 microprocessors provide compatibility on several levels. The processors are software-compatible with the existing members of the 29K family (the Am29000, Am29005, and Am29050 microprocessors). In addition, the processors are pin-, bus-, and software-compatible with future members of the Am29030 and Am29035 processors.

Pin- and bus-compatibility within the Am29030 and Am29035 processors is a unique feature that ensures a convenient upgrade path, without hardware or software redesign, for embedded applications.

1.2.7 Wide Range of Price/Performance Points

To reduce design costs and time-to-market, one basic system design may be used as the foundation for an entire product line. From this design, numerous implementations of the product at various levels of price and performance may be derived with minimum time, effort, and cost.

The Am29030 and Am29035 processors provide this capability through Scalable Clocking technology, the narrow read interface, programmable bus sizing, and hardware and software compatibility. Processors can be upgraded without hardware and software redesign and combined with high-performance or mid-performance memory. The narrow read interface accommodates numerous ROM configurations. In addition, programmable bus sizing allows Am29035 microprocessor-based systems to support memory upgrades in half-megabyte increments.

These new AMD processors provide a wide range of price/performance points for any system design.

1.2.8 Complete Development and Support Environment

A complete development and support environment is vital for reducing a product's time-to-market. Advanced Micro Devices has created a standard development environment for the 29K Family of processors. In addition, the Fusion29K℠ third-party support organization provides the most comprehensive customer/partner program in the embedded processor market.

Advanced Micro Devices offers a complete set of hardware and software tools for design, integration, debugging, and benchmarking. These tools, which are available now for the RISC family include the following:

- HighC29K optimizing C compiler with assembler, linker, ANSI library functions, and 29K architectural simulator
- XRAY29K source-level debugger
- Debug monitor
- EB29030 execution board

In addition, Advanced Micro Devices has developed a standard host interface (HIF) for OS services, and extensions for the UNIX® common object file format (COFF).

This support is augmented by an engineering hotline, an on-line bulletin board, and field application engineers.

1.3 PERFORMANCE OVERVIEW

The Am29030 and Am29035 microprocessors provide a significant margin of performance over other processors in their class, since the majority of processor features were defined for the maximum achievable performance at a reasonable cost. This section describes the features of the Am29030 and Am29035 microprocessors from the point of view of system performance.

1.3.1 Instruction Timing

The Am29030 and Am29035 microprocessors use an Arithmetic/Logic Unit, a Field Shift Unit, and a Prioritizer to execute most instructions. Each of these is organized to operate on 32-bit operands and provide a 32-bit result. All operations are performed in a single cycle.

The performance degradation of load and store operations is minimized in the Am29030 and Am29035 microprocessors by overlapping them with instruction execution, by taking advantage of pipelining, and by organizing the flow of external data into the processor so that the impact of external accesses is minimized.

1.3.2 Pipelining

Instruction operations are overlapped with instruction fetch, instruction decode, operand fetch, and result write-back to the Register File. Pipeline forwarding logic detects pipeline dependencies and routes data as required, avoiding delays that might arise from these dependencies.

Pipeline interlocks are implemented by processor hardware. Except for a few special cases, it is not necessary to rearrange programs to avoid pipeline dependencies, although this is sometimes desirable for performance.

1.3.3 Instruction Cache

The Am29030 microprocessor utilizes an 8K-byte, two-way-set-associative instruction cache to meet the instruction bandwidth requirements for high performance.

The Am29035 microprocessor utilizes a 4K-byte, direct-mapped instruction cache to meet the instruction bandwidth requirements for mid-range performance.

In both processors, the instruction cache stores the most recently fetched instructions. The instruction cache block size is four words (16 bytes). Either processor allows all or part of the instruction cache to be locked, allowing the processor to retain special instruction sequences. The Am29030 microprocessor allows either one or both columns of the cache to be locked, and the Am29035 microprocessor allows the entire cache to be locked.

1.3.4 Instruction Set Overview

The Am29030 and Am29035 microprocessors employ a three-address instruction set architecture. The compiler or assembly-language programmer is given complete freedom to allocate register usage. There are 192 general-purpose registers, allowing the retention of intermediate calculations and avoiding needless data destruction. Instruction operands may be contained in any of the general-purpose registers, and the results may be stored into any of the general-purpose registers.

The Am29030 and Am29035 instruction set contains 117 instructions which are divided into nine classes. These classes are integer arithmetic, compare, logical, shift, data movement, constant, floating-point, branch, and miscellaneous. The floating-point instructions are not executed directly, but are emulated by trap handlers.

All directly implemented instructions are capable of executing in one processor cycle, with the exception of interrupt returns, loads, and stores.

1.3.5 Data Formats

The Am29030 and Am29035 microprocessors define a word as 32 bits of data, a half-word as 16 bits, and a byte as 8 bits. The hardware provides direct support for word-integer (signed and unsigned), word-logical, word-boolean, half-word integer (signed and unsigned) and character data (signed and unsigned).

Word-boolean data is based on the value contained in the most significant bit of the word. The values TRUE and FALSE are represented by the MSB values 1 and 0 respectively.

Other data formats, such as character strings, are supported by instruction sequences. Floating-point formats (single and double precision) are defined for the processors; however, there is no direct hardware support for these formats in the Am29030 and Am29035 microprocessors.

1.3.6 Protection

The Am29030 and Am29035 microprocessors provide a variety of system protection features. The processors offer two mutually exclusive modes of execution, the User and Supervisor modes, which restrict or permit accesses to certain processor registers and external storage locations.

The Memory Management Unit (MMU) provides for memory protection through the use of six access permission bits. These bits restrict memory accesses to instruction

execution (user and/or supervisor execution) and type of data access (read, write or no access). The MMU may be used also to provide protection for the system via user-defined outputs.

The Register File may be configured to restrict accesses to Supervisor-mode programs on a bank-by-bank basis.

1.3.7 Memory Management

A 64-entry Translation Look-Aside Buffer (TLB) performs virtual-to-physical address translation, avoiding the cycle which would be required to transfer the virtual address to an external TLB. A number of enhancements improve the performance of address translation:

1. Pipelining—The operation of the TLB is pipelined with other processor operations.
2. Task Identifiers—Task Identifiers allow TLB entries to be matched to different processes, so that TLB invalidation is not required during task switches.
3. Least-Recently Used Hardware—This hardware allows immediate selection of a TLB entry to be replaced.
4. Software Reload—Software reload allows the operating system to use a page-mapping scheme which is best matched to its environment. One of Paged-segmented, one-level-page mapping, two-level-page mapping, or any other user-defined page-mapping scheme can be supported. Because Am29030 and Am29035 instructions execute at an average rate of nearly one instruction per cycle, software reload has performance approaching that of hardware TLB reload.

1.3.8 Interrupts and Traps

When an Am29030 or Am29035 microprocessor takes an interrupt or trap, it does not automatically save its current state information in memory. This lightweight interrupt and trap facility greatly improves the performance of temporary interruptions such as TLB reload or other simple operating-system calls which require no saving of state information.

In cases where the processor state must be saved, the saving and restoring of state information is under the control of software. The methods and data structures used to handle interrupts—and the amount of state saved—may be tailored to the needs of a particular system.

Interrupts and traps are dispatched through a 256-entry Vector Table which directs the processor to a routine that handles a given interrupt or trap. The Vector Table may be relocated in memory by the modification of a processor register. There may be multiple Vector Tables in the system, though only one is active at any given time.

The Vector Table is a table of pointers to the interrupt and trap handlers, requiring only 1 Kbyte of memory. This structure requires that the processors perform a vector fetch every time an interrupt or trap is taken. The vector fetch requires at least three cycles, in addition to the number of cycles required for the basic memory access.

1.4 DEBUGGING AND TESTING

The Am29030 and Am29035 microprocessors provide debugging and testing features at both the software and hardware levels.

Software debugging is facilitated by the instruction trace facility and instruction breakpoints. Instruction tracing is accomplished by forcing the processor to trap after

each instruction has been executed. Instruction breakpoints are implemented by the HALT instruction or by a software trap.

Software can access all tag/status, and instruction words in the instruction cache for testing.

The processors provide two additional features to assist system debugging and testing. The first feature, the Test/Development Interface, is composed of a group of pins that indicate the state of the processor and control the operation of the processor. The second feature is an IEEE Std. 1149.1-1990 (JTAG) compliant Standard Test Access Port and Boundary-Scan Architecture. The Test Access Port provides a scan interface for testing system hardware in a production environment, and contains extensions that allow a hardware-development system to control and observe the processor without interposing hardware between the processor and system.

REFERENCES

1 **Hill, M.D.** *Aspects of Cache Memory and Instruction Buffer Performance*, PhD Dissertation, University of California at Berkeley, CA, USA (1987)

CHAPTER 2

PROGRAMMING

This chapter focuses on programming the Am29030 and Am29035 microprocessors. First, this chapter presents an instruction set overview. It then describes the register model, emphasizing the general- and special-purpose registers. This chapter also describes certain special-purpose registers that deal directly with instruction execution. Finally, this chapter describes general considerations related to applications programming.

2.1 INSTRUCTION SET

The Am29030 and Am29035 microprocessors recognize 117 instructions. All instructions execute in a single cycle, except for IRET, IRETINV, LOADM, STOREM, and certain arithmetic instructions such as floating-point instructions.

Most instructions deal with general-purpose registers for operands and results; however, in most instructions, an 8-bit constant can be used in place of a register-based operand. Some instructions deal with special-purpose registers, TLB registers, and external devices and memories.

This section describes the nine instruction classes in the Am29030 and Am29035 microprocessors, and provides a brief summary of instruction operations. A detailed instruction specification is contained in Chapter 12. Section 12.1 describes the nomenclature used here.

If the processor attempts to execute an instruction which is not implemented, an Illegal Opcode trap occurs, unless the instruction is reserved for emulation (see Section 2.1.10). Reserved instructions are assigned individual traps.

2.1.1 Integer Arithmetic

The Integer Arithmetic instructions perform add, subtract, multiply, and divide operations on word-length integers. Certain instructions in this class cause traps if signed or unsigned overflow occurs during the execution of the instruction. There is support for multi-precision arithmetic on operands whose lengths are multiples of words. All instructions in this class set the ALU Status Register. The integer arithmetic instructions are shown in Table 2-1.

2.1.2 Compare

The Compare instructions test for various relationships between two values. For all Compare instructions except the CPBYTE instruction, the comparisons are performed on word-length signed or unsigned integers. There are two types of Compare instructions. The first type places a Boolean value reflecting the outcome of the compare into a general-purpose register. For the second type, assert instructions, instruction execution continues only if the comparison is true; otherwise a trap occurs. The assert instructions specify a vector for the trap (see Section 8.2).

Table 2-1 Integer Arithmetic Instructions

Mnemonic	Operation Description
ADD	DEST ← SRCA + SRCB
ADDS	DEST ← SRCA + SRCB IF signed overflow THEN Trap (Out of Range)
ADDU	DEST ← SRCA + SRCB IF unsigned overflow THEN Trap (Out of Range)
ADDC	DEST ← SRCA + SRCB + C
ADDCS	DEST ← SRCA + SRCB + C IF signed overflow THEN Trap (Out of Range)
ADDCU	DEST ← SRCA + SRCB + C IF unsigned overflow THEN Trap (Out of Range)
SUB	DEST ← SRCA − SRCB
SUBS	DEST ← SRCA − SRCB IF signed overflow THEN Trap (Out of Range)
SUBU	DEST ← SRCA − SRCB IF unsigned underflow THEN Trap (Out of Range)
SUBC	DEST ← SRCA − SRCB − 1 + C
SUBCS	DEST ← SRCA − SRCB − 1 + C IF signed overflow THEN Trap (Out of Range)
SUBCU	DEST ← SRCA − SRCB − 1 + C IF unsigned underflow THEN Trap (Out of Range)
SUBR	DEST ← SRCB − SRCA
SUBRS	DEST ← SRCB − SRCA IF signed overflow THEN Trap (Out of Range)
SUBRU	DEST ← SRCB − SRCA IF unsigned underflow THEN Trap (Out of Range)
SUBRC	DEST ← SRCB − SRCA − 1 + C
SUBRCS	DEST ← SRCB − SRCA − 1 + C IF signed overflow THEN Trap (Out of Range)
SUBRCU	DEST ← SRCB − SRCA − 1 + C IF unsigned underflow THEN Trap (Out of Range)
MULTIPLU	DEST ← SRCA · SRCB (unsigned)
MULTIPLY	DEST ← SRCA · SRCB (signed)
MUL	Perform one-bit step of a multiply operation (signed)
MULL	Complete a sequence of multiply steps
MULTM	DEST ← SRCA · SRCB (signed), most-significant bits
MULTMU	DEST ← SRCA · SRCB (unsigned), most-significant bits
MULU	Perform one-bit step of a multiply operation (unsigned)
DIVIDE	DEST ← (Q//SRCA)/SRCB (signed) Q ← Remainder
DIVIDU	DEST ← (Q//SRCA)/SRCB (unsigned) Q ← Remainder
DIV0	Initialize for a sequence of divide steps (unsigned)
DIV	Perform one-bit step of a divide operation (unsigned)
DIVL	Complete a sequence of divide steps (unsigned)
DIVREM	Generate remainder for divide operation (unsigned)

Table 2-2 Compare Instructions

Mnemonic	Operation Description
CPEQ	IF SRCA = SRCB THEN DEST ← TRUE ELSE DEST ← FALSE
CPNEQ	IF SRCA <> SRCB THEN DEST ← TRUE ELSE DEST ← FALSE
CPLT	IF SRCA < SRCB THEN DEST ← TRUE ELSE DEST ← FALSE
CPLTU	IF SRCA < SRCB (unsigned) THEN DEST ← TRUE ELSE DEST ← FALSE
CPLE	IF SRCA ≤ SRCB THEN DEST ← TRUE ELSE DEST ← FALSE
CPLEU	IF SRCA ≤ SRCB (unsigned) THEN DEST ← TRUE ELSE DEST ← FALSE
CPGT	IF SRCA > SRCB THEN DEST ← TRUE ELSE DEST ← FALSE
CPGTU	IF SRCA > SRCB (unsigned) THEN DEST ← TRUE ELSE DEST ← FALSE
CPGE	If SRCA ≥ SRCB THEN DEST ← TRUE ELSE DEST ← FALSE
CPGEU	IF SRCA ≥ SRCB (unsigned) THEN DEST ← TRUE ELSE DEST ← FALSE
CPBYTE	IF (SRCA.BYTE0 = SRCB.BYTE0) OR (SRCA.BYTE1 = SRCB.BYTE1) OR (SRCA.BYTE2 = SRCB.BYTE2) OR (SRCA.BYTE3 = SRCB.BYTE3) THEN DEST ← TRUE ELSE DEST ← FALSE
ASEQ	IF SRCA = SRCB THEN Continue ELSE Trap (VN)
ASNEQ	IF SRCA <> SRCB THEN Continue ELSE Trap (VN)
ASLT	IF SRCA < SRCB THEN Continue ELSE Trap (VN)
ASLTU	IF SRCA < SRCB (unsigned) THEN Continue ELSE Trap (VN)
ASLE	IF SRCA ≤ SRCB THEN Continue ELSE Trap (VN)
ASLEU	IF SRCA ≤ SRCB (unsigned) THEN Continue ELSE Trap (VN)
ASGT	IF SRCA > SRCB THEN Continue ELSE Trap (VN)
ASGTU	IF SRCA > SRCB (unsigned) THEN Continue ELSE Trap (VN)
ASGE	IF SRCA ≥ SRCB THEN Continue ELSE Trap (VN)
ASGEU	IF SRCA ≥ SRCB (unsigned) THEN Continue ELSE Trap (VN)

The assert instructions support run-time operand checking and operating-system calls. If the trap occurs in the User mode, and a trap number between 0 and 63 is specified by the instruction, a Protection Violation trap occurs. The Compare instructions are shown in Table 2-2.

2.1.3 Logical

The Logical instructions perform a set of bit-by-bit Boolean functions on word-length bit strings. All instructions in this class set the ALU Status Register. These instructions are shown in Table 2-3.

2.1.4 Shift

The Shift instructions (Table 2-4) perform arithmetic and logical shifts. All but the EXTRACT instruction operate on word-length data and produce a word-length result. The EXTRACT instruction operates on double-word data and produces a word-length result. If both parts of the double-word for the EXTRACT instruction are from the same source, the EXTRACT operation is equivalent to a rotate operation. For each operation, the shift count is a 5-bit integer, specifying a shift amount in the range of 0 to 31 bits.

2.1.5 Data Movement

The Data Movement instructions (Table 2-5) move bytes, half-words, and words between processor registers. In addition, they move data between general-purpose registers and external devices, and memories.

Table 2-3 Logical Instructions

Mnemonic	Operation Description
AND	DEST ← SRCA & SRCB
ANDN	DEST ← SRCA & ~ SRCB
NAND	DEST ← ~ (SRCA & SRCB)
OR	DEST ← SRCA | SRCB
NOR	DEST ← ~ (SRCA | SRCB)
XOR	DEST ← SRCA ^ SRCB
XNOR	DEST ← ~ (SRCA ^ SRCB)

Table 2-4 Shift Instructions

Mnemonic	Operation Description
SLL	DEST ← SRCA << SRCB (zero fill)
SRL	DEST ← SRCA >> SRCB (zero fill)
SRA	DEST ← SRCA >> SRCB (sign fill)
EXTRACT	DEST ← high-order word of (SRCA//SRCB << FC)

Table 2-5 Data Movement Instructions

Mnemonic	Operation Description
LOAD	DEST ← EXTERNAL WORD [SRCB]
LOADL	DEST ← EXTERNAL WORD [SRCB] assert LOCK output during access
LOADSET	DEST ← EXTERNAL WORD [SRCB] EXTERNAL WORD [SRCB] ← h'FFFFFFFF' assert $\overline{\text{LOCK}}$ output during access
LOADM	DEST.. DEST + COUNT ← EXTERNAL WORD [SRCB] .. EXTERNAL WORD [SRCB + COUNT · 4]
STORE	EXTERNAL WORD [SRCB] ← SRCA
STOREL	EXTERNAL WORD [SRCB] ← SRCA assert $\overline{\text{LOCK}}$ output during access
STOREM	EXTERNAL WORD [SRCB] .. EXTERNAL WORD [SRCB + COUNT · 4] ← SRCA .. SRCA + COUNT
EXBYTE	DEST ← SRCB, with low-order byte replaced by byte in SRCA selected by BP
EXHW	DEST ← SRCB, with low-order half-word replaced by half-word in SRCA selected by BP
EXHWS	DEST ← half-word in SRCA selected by BP, sign-extended to 32 bits
INBYTE	DEST ← SRCA, with byte selected by BP replaced by low-order byte of SRCB
INHW	DEST ← SRCA, with half-word selected by BP replaced by low-order half-word of SRCB
MFSR	DEST ← SPECIAL
MFTLB	DEST ← TLB [SRCA]
MTSR	SPDEST ← SRCB
MTSRIM	SPDEST ← 0I16
MTTLB	TLB [SRCA] ← SRCB

2.1.6 Constant

The Constant instructions (Table 2-6) provide the ability to place half-word and word constants into registers. Most instructions in the instruction set allow an 8-bit constant as an operand. The Constant instructions allow the construction of larger constants.

2.1.7 Floating-Point

The Floating-Point instructions (Table 2-7) provide operations on single-precision (32-bit) or double-precision (64-bit) floating-point data. They also provide conversions between single-precision, double-precision, and integer number representations. In the Am29030 and Am29035 processor implementations, these instructions cause traps to routines which perform the floating-point operations.

Table 2-6 Constant Instructions

Mnemonic	Operation Description
CONST	DEST ← 0I16
CONSTH	Replace high-order half-word of SRCA by I16
CONSTN	DEST ← 1I16

Table 2-7 Floating-Point Instructions

Mnemonic	Operation Description
FADD	DEST (single-precision) ← SRCA (single-precision) + SRCB (single-precision)
DADD	DEST (double-precision) ← SRCA (double-precision) + SRCB (double-precision)
FSUB	DEST (single-precision) ← SRCA (double-precision) − SRCB (single-precision)
DSUB	DEST (double-precision) ← SRCA (double-precision) − SRCB (double-precision)
FMUL	DEST (single-precision) ← SRCA (single-precision) · SRCB (single-precision)
FDMUL	DEST (double-precision) ← SRCA (single-precision) · SRCB (single-precision)
DMUL	DEST (double-precision) ← SRCA (double-precision) · SRCB (double-precision)
FDIV	DEST (single-precision) ← SRCA (single-precision / SRCB (single-precision)
DDIV	DEST (double-precision) ← SRCA (double-precision) / SRCB (double-precision)
FEQ	IF SRCA (single-precision) = SRCB (single-precision) THEN DEST ← TRUE ELSE DEST ← FALSE
DEQ	IF SRCA (double-precision) = SRCB (double-precision) THEN DEST ← TRUE ELSE DEST ← FALSE
FGE	IF SRCA (single-precision) >= SRCB (single-precision THEN DEST ← TRUE ELSE DEST ← FALSE
DGE	IF SRCA (double-precision) >= SRCB (double-precision THEN DEST ← TRUE ELSE DEST ← FALSE
FGT	IF SRCA (single-precision) > SRCB (single-precision) THEN DEST ← TRUE ELSE DEST ← FALSE

Floating-Point Instructions (continued)

Mnemonic	Operation Description
DGT	IF SRCA (double-precision) > SRCB (double-precision) THEN DEST ← TRUE ELSE DEST ← FALSE
SQRT	DEST (single-precision, double-precision) ← SQRT [SRCA (single-precision, double-precision)]
CONVERT	DEST (integer, single-precision, double-precision) ← SRCA (integer, single-precision, double-precision)
CLASS	DEST ← CLASS [SRCA (single-precision, double-precision)]

2.1.8 Branch

The Branch instructions (Table 2-8) control the execution flow of instructions. Branch target addresses may be absolute, relative to the Program Counter (with the offset given by a signed instruction constant), or contained in a general-purpose register. For conditional jumps, the outcome of the jump is based on a Boolean value in a general-purpose register. Procedure calls are unconditional, and save the return address in a general-purpose register. All branches have a delayed effect; the instruction sequence following the branch is executed regardless of the outcome of the branch.

Table 2-8 Branch Instructions

Mnemonic	Operation Description
CALL	DEST ← PC//00 + 8 PC ← TARGET Execute delay instruction
CALLI	DEST ← PC//00 + 8 PC ← SRCB Execute delay instruction
JMP	PC ← TARGET Execute delay instruction
JMPI	PC ← SRCB Execute delay instruction
JMPT	IF SRCA = TRUE THEN PC ← TARGET Execute delay instruction
JMPTI	IF SRCA = TRUE THEN PC ← SRCB Execute delay instruction
JMPF	IF SRCA = FALSE THEN PC ← TARGET Execute delay instruction
JMPFI	IF SRCA = FALSE THEN PC ← SRCB Execute delay instruction

JMPFDEC	IF SRCA = FALSE THEN
	SRCA ← SRCA − 1
	PC ← TARGET
	ELSE
	SRCA ← SRCA − 1
	Execute delay instruction

2.1.9 Miscellaneous

The Miscellaneous instructions (Table 2-9) perform various operations that cannot be grouped into other instruction classes. In certain cases, these are control functions available only to Supervisor-mode programs.

2.1.10 Reserved Instructions

Sixteen Am29030 and Am29035 microprocessor operation codes are reserved for instruction emulation. Each of these instructions causes a trap and sets the indirect pointers IPC, IPA, and IPB. The relevant operation codes, and the corresponding trap vectors, are as follows:

Operation Codes (Hexadecimal)	Trap Vector Numbers (Decimal)
D8–DD	24–29
E7–E9	39–41
F8	56
FA–FF	58–63

The reserved instructions are intended for future processor enhancements, and users desiring compatibility with future processor versions should not use them for any purpose.

2.2 REGISTER MODEL

The Am29030 and Am29035 microprocessors have three classes of registers that are accessible by instructions. These are the general-purpose registers, special-purpose registers and Translation Look-Aside Buffer (TLB) registers. Any operation available to the Am29030 and Am29035 microprocessors can be performed on the general-purpose registers, while special-purpose registers are accessed only by the instructions MTSR, MTSRIM, and MFSR, and the TLB registers are accessed only by the instructions MTTLB and MFTLB. This section describes the general-purpose and special-purpose registers. The TLB registers are discussed in Section 7.2.

Table 2-9 Miscellaneous Instructions

Mnemonic	Operation Description
CLZ	Determine number of leading zeros in a word
SETIP	Set IPA, IPB, and IPC with operand register numbers
EMULATE	Load IPA and IPB with operand register numbers, and Trap (VN)
INV	Reset all Valid bits in Branch Target Cache memory to zeros
IRET	Perform an interrupt return sequence
IRETINV	Perform an interrupt return sequence and reset all Valid bits in the Instruction Cache to zeros
HALT	Enter Halt mode

2.2.1 General-Purpose Registers

The Am29030 and Am29035 microprocessors incorporate 192 general-purpose registers. The organization of the general-purpose registers is diagrammed in Figure 2-1.

General-purpose registers hold the following types of operands for program use:

1. 32-bit addresses
2. 32-bit signed or unsigned integers
3. 32-bit branch-target addresses
4. 32-bit logical bit strings
5. 8-bit signed or unsigned characters
6. 16-bit signed or unsigned integers
7. Word-length Booleans
8. Single-precision floating-point numbers
9. Double-precision floating-point numbers (in two register locations)

Because a large number of general-purpose registers are provided, a large amount of frequently used data can be kept on-chip, where access time is fastest.

Am29030 and Am29035 microprocessor instructions can specify two general-purpose registers for source operands, and one general-purpose register for storing the instruction result. These registers are specified by three 8-bit instruction fields containing register numbers. A register may be specified directly by the instruction, or indirectly by one of three special-purpose registers.

2.2.1.1 REGISTER ADDRESSING

The general-purpose registers are partitioned into 64 global registers and 128 local registers, differentiated by the most-significant bit of the register number. The distinction between global and local registers is the result of register-addressing considerations.

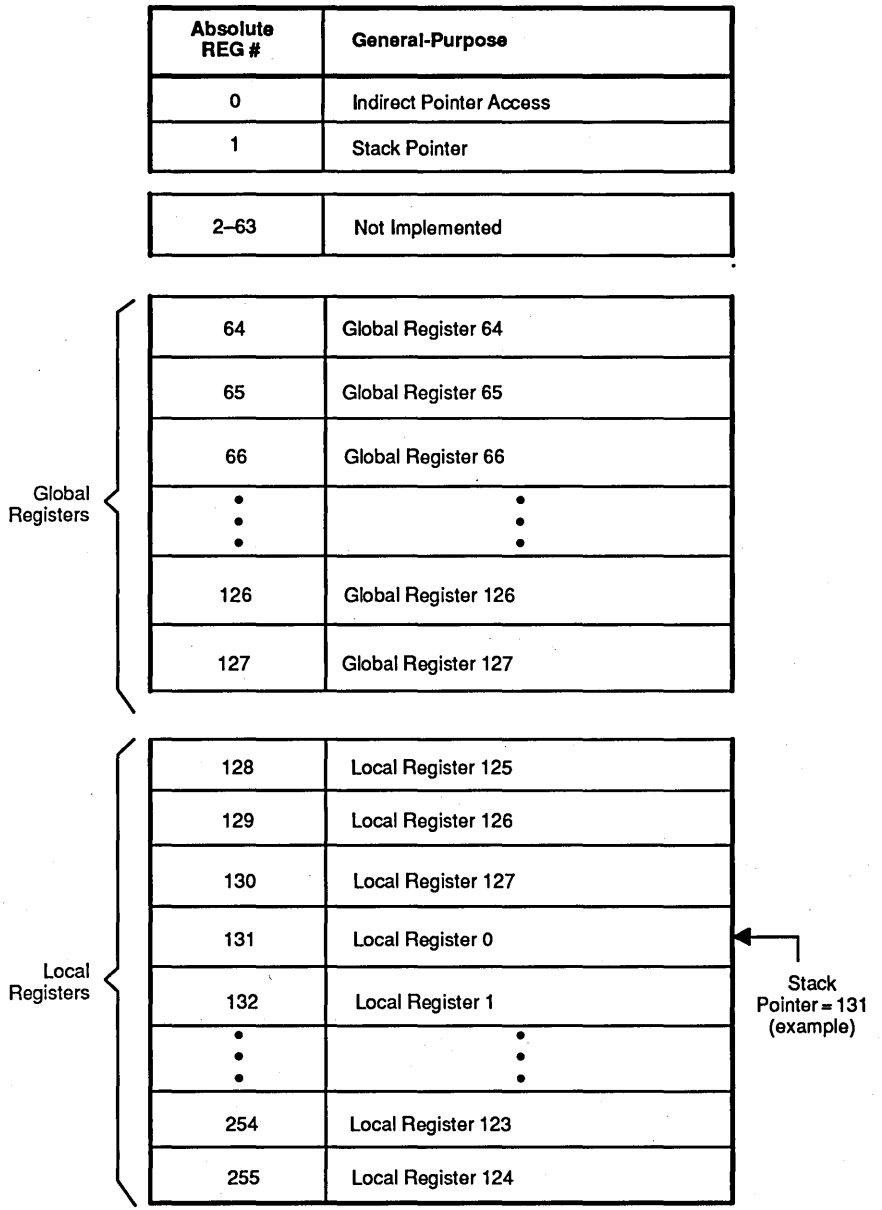
The following terminology is used to describe the addressing of general-purpose registers:

1. Register number—this is a software-level number for a general-purpose register. For example, this is the number contained in an instruction field. Register numbers range from 0 to 255.
2. Global-register number—this is a software-level number for a global register. Global-register numbers range from 0 to 127.
3. Local-register number—this is a software-level number for a local register. Local-register numbers range from 0 to 127.
4. Absolute-register number—this is a hardware-level number used to select a general-purpose register in the Register File. Absolute-register numbers range from 0 to 255.

2.2.1.2 GLOBAL REGISTERS

When the most-significant bit of a register number is 0, a global register is selected. The seven least-significant bits of the register number give the global-register number. For global registers, the absolute-register number is equivalent to the register number.

Figure 2-1 General-Purpose Register Organization



Global registers 2 through 63 are not implemented. An attempt to access these registers yields unpredictable results; however, they may be protected from User-mode access by the Register Bank Protect Register (see Section 6.2.1).

The register numbers associated with Global Registers 0 and 1 have special meaning. The number for Global Register 0 specifies that an indirect pointer is to be used as the source of the register number (see Section 2.3); there is an indirect pointer for each of the instruction operand/result registers. Global Register 1 contains the Stack Pointer, which is used in the addressing of local registers.

2.2.1.3 LOCAL REGISTERS

When the most-significant bit of a register number is 1, a local register is selected. The seven least-significant bits of the register number give the local-register number. For local registers, the absolute-register number is obtained by adding the local-register number to bits 8–2 of the Stack Pointer and truncating the result to seven bits; the most-significant bit of the original register number is unchanged (i.e., it remains a 1).

The Stack Pointer addition applied to local-register numbers provides a limited form of base-plus-offset addressing within the local registers. The Stack Pointer contains the 32-bit base address. This assists run-time storage management of variables for dynamically nested procedures (see Chapter 4).

2.2.1.4 LOCAL-REGISTER STACK POINTER

The Stack Pointer is a 32-bit register that may be an operand of an instruction as any other general-purpose register. However, a shadow copy of Global Register 1 is maintained by processor hardware for use in local-register addressing. This shadow copy is set only with the results of Arithmetic and Logical instructions. If the Stack Pointer is set with the result of any other instruction class, local registers cannot be accessed predictably until the Stack Pointer is set once again with an Arithmetic or Logical instruction.

A modification of the Stack Pointer has a delayed effect on the addressing of local registers, as discussed in Section 5.6.

2.2.2 Special-Purpose Registers

The Am29030 and Am29035 microprocessors contain 28 special-purpose registers. The organization of the special-purpose registers is shown in Figure 2-2.

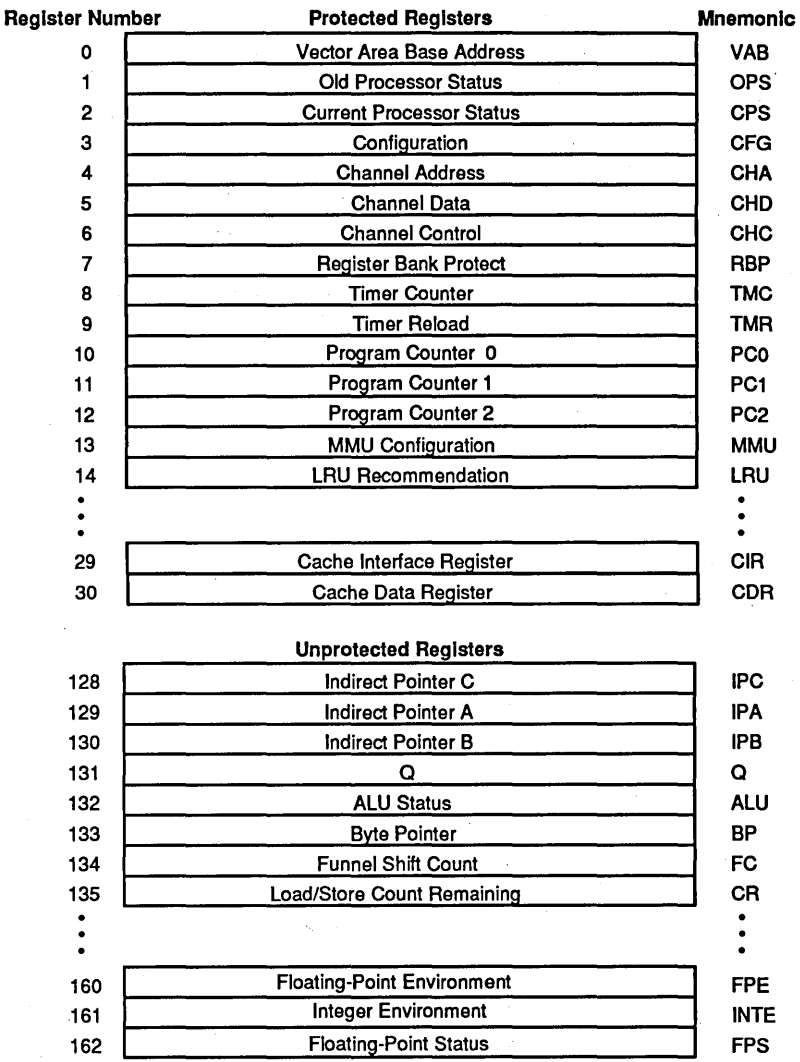
Special-purpose registers provide controls and data for certain processor operations. Some special-purpose registers are updated dynamically by the processor, independent of software controls. Because of this, a read of a special-purpose register following a write does not necessarily get the data that was written.

Some special-purpose registers have fields that are reserved for future processor implementations. When a special-purpose register is read, a bit in a reserved field is read as a 0. An attempt to write a reserved bit with a 1 has no effect; however, this should be avoided because of upward-compatibility considerations.

The special-purpose registers are accessed by explicit data movement only. Instructions that move data to or from a special-purpose register specify the special-purpose register by an 8-bit field containing a special-purpose register number. Register numbers are specified directly by instructions.

The special-purpose registers are partitioned into protected and unprotected registers. Special-purpose registers numbered 0–127 and 160–255 are protected (note that not all of these are implemented). Special-purpose registers numbered 128–159 are unprotected (again, not all are implemented).

Figure 2-2 Special-Purpose Registers



Protected special-purpose registers numbered 0–127 are accessible only by programs executing in the Supervisor mode. An attempted read or write of a special-purpose register by a User-mode program causes a protection violation trap to occur. Special-purpose registers numbered 160–255, though architecturally unprotected, are not accessible by programs in the User mode or the Supervisor mode. These register numbers are reserved for virtual registers in the arithmetic architecture, and any attempted access causes a Protection Violation trap.

The Floating-Point Environment Register, Integer Environment Register, and Floating-Point Status Register are not implemented in processor hardware. These registers are implemented via the virtual arithmetic interface provided on the Am29030 and Am29035 microprocessor.

An attempted read of an unimplemented special-purpose register yields an unpredictable value. An attempted write of an unimplemented, protected special-purpose register has an unpredictable effect on processor operation, unless the write causes a Protection Violation. An attempted write of an unimplemented, unprotected special-purpose register has no effect; however, this should be avoided because of upward-compatibility considerations.

2.3 ADDRESSING REGISTERS INDIRECTLY

Specifying Global Register 0 as an instruction operand register or result register causes an indirect access to the general-purpose registers. In this case, the absolute-register number is provided by an indirect pointer contained in a special-purpose register.

Each of the three possible registers for instruction execution has an associated 8-bit indirect pointer. Indirect register numbers can be selected independently for each of the three operands. Since the indirect pointers contain absolute-register numbers, the number in an indirect pointer is not added to the Stack Pointer when local registers are selected.

The indirect pointers are set by the Move To Special Register, SETIP, and EMULATE instructions and by floating-point, MULTIPLY, MULTM, MULTIPLU, MULTMU, DIVIDE, and DIVIDU instructions.

For a Move-To-Special-Register instruction, an indirect pointer is set with bits 9–2 of the 32-bit source operand. This provides consistency between the addressing of words in general-purpose registers and the addressing of words in external devices or memories. A modification of an indirect pointer using a Move To Special Register has a delayed effect on the addressing of general-purpose registers, as discussed in Section 5.6.

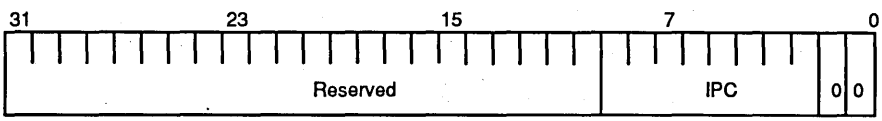
For the remaining instructions, all three indirect pointers are set simultaneously with the absolute-register numbers derived from the register numbers specified by the instruction. For any local registers selected by the instruction, the Stack-Pointer addition is applied to the register numbers before the indirect pointers are set.

Except when an indirect pointer is set by a Move-To-Special-Register instruction, register numbers stored into the indirect pointers are checked for bank-protection violations at the time that the indirect pointers are set.

2.3.1 Indirect Pointer C (IPC, Register 128)

This unprotected special-purpose register (Figure 2-3) provides the RC-operand register number (see Section 12.3) when an instruction RC field has the value zero (i.e., when Global Register 0 is specified).

Figure 2-3 Indirect Pointer C Register



Bits 31–10: Reserved.

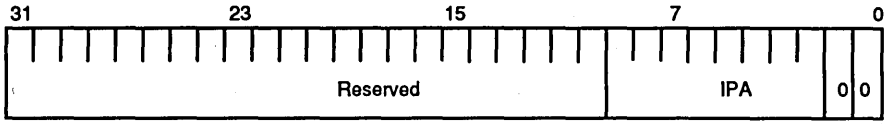
Bits 9–2: Indirect Pointer C (IPC)—The 8-bit IPC field contains an absolute-register number for a general-purpose register. This number directly selects a register (Stack-Pointer addition is not performed in the case of local registers).

Bits 1–0: Zeros—The IPC field is aligned for compatibility with word addresses.

2.3.2 Indirect Pointer A (IPA, Register 129)

This unprotected special-purpose register (Figure 2-4) provides the RA-operand register number (see Section 12.3) when an instruction RA field has the value zero (i.e., when Global Register 0 is specified).

Figure 2-4 Indirect Pointer A Register



Bits 31–10: Reserved.

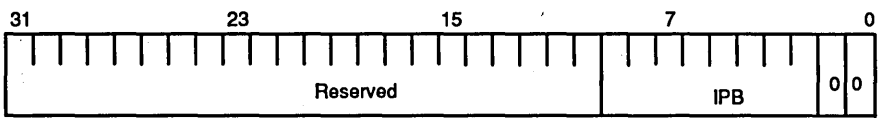
Bits 9–2: Indirect Pointer A (IPA)—The 8-bit IPA field contains an absolute-register number for either a general-purpose register or a local register. This number directly selects a register (Stack-Pointer addition is not performed in the case of local registers).

Bits 1–0: Zeros—The IPA field is aligned for compatibility with word addresses.

2.3.3 Indirect Pointer B (IPB, Register 130)

This unprotected special-purpose register (Figure 2-5) provides the RB-operand register number (see Section 12.3) when an instruction RB field has the value zero (i.e., when Global Register 0 is specified).

Figure 2-5 Indirect Pointer B Register



Bits 31–10: Reserved.

Bits 9–2: Indirect Pointer B (IPB)—The 8-bit IPB field contains an absolute-register number for a general-purpose register. This number directly selects a register (Stack-Pointer addition is not performed in the case of local registers).

Bits 1–0: Zeros—The IPB field is aligned for compatibility with word addresses.

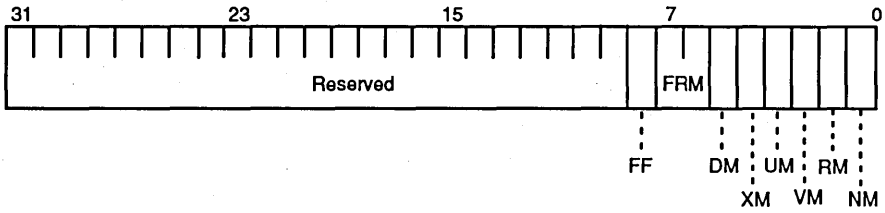
2.4 INSTRUCTION ENVIRONMENT

This section describes the special-purpose registers that affect the execution of floating-point and integer arithmetic instructions.

2.4.1 Floating-Point Environment (FPE, Register 160)

This unprotected special-purpose register (Figure 2-6) contains control bits that affect the execution of floating-point operations.

Figure 2-6 Floating-Point Environment Register



Bits 31–9: Reserved.

Bit 8: Fast Float Select (FF)—The FF bit being 1 enables fast floating-point operations, in which certain requirements of the IEEE floating-point specification are not met. This improves the performance of certain operations by sacrificing conformance to the IEEE specification.

Bits 7–6: Floating-Point Round Mode (FRM)—This field specifies the default mode used to round the results of floating-point operations, as follows:

FRM1–0	Round Mode
00	Round to nearest
01	Round to $-\infty$
10	Round to $+\infty$
11	Round to zero

Bit 5: Floating-Point Divide-By-Zero Mask (DM)—If the DM bit is 0, a Floating-Point Exception trap occurs when the divisor of a floating-point division operation is zero and the dividend is a non-zero, finite number. If the DM bit is 1, a Floating-Point Exception trap does not occur for divide-by-zero.

Bit 4: Floating-Point Inexact Result Mask (XM)—If the XM bit is 0, a Floating-Point Exception trap occurs when the result of a floating-point operation is not equal to the infinitely precise result. If the XM bit is 1, a Floating-Point Exception trap does not occur for an inexact result.

Bit 3: Floating-Point Underflow Mask (UM)—If the UM bit is 0, a Floating-Point Exception trap occurs when the result of a floating-point operation is too small to be expressed in the destination format. If the UM bit is 1, a Floating-Point Exception trap does not occur for underflow.

Bit 2: Floating-Point Overflow Mask (VM)—If the VM bit is 0, a Floating-Point Exception trap occurs when the result of a floating-point operation is too large to be expressed in the destination format. If the VM bit is 1, a Floating-Point Exception trap does not occur for overflow.

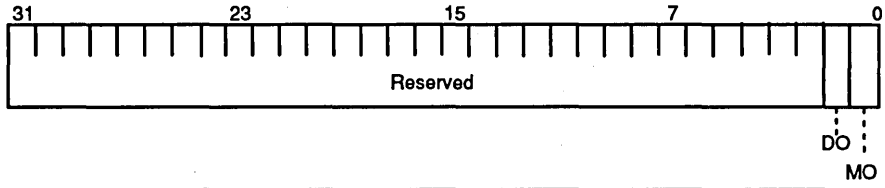
Bit 1: Floating-Point Reserved Operand Mask (RM)—If the RM bit is 0, a Floating-Point Exception trap occurs when one or more input operands to a floating-point operation is a reserved value, or when the result of a floating-point operation is a reserved value. If the RM bit is 1, a Floating-Point Exception trap does not occur for reserved operands.

Bit 0: Floating-Point Invalid Operation Mask (NM)—If the NM bit is 0, a Floating-Point Exception trap occurs when the input operands to a floating-point operation produce an indeterminate result (e.g., ∞ times 0). If the NM bit is 1, a Floating-Point Exception trap does not occur for invalid operations.

2.4.2 Integer Environment (INTE, Register 161)

This unprotected special-purpose register (Figure 2-7) contains control bits which affect the execution of integer multiplication and division operations.

Figure 2-7 Integer Environment Register



Bits 31–2: Reserved.

Bit 1: Integer Division Overflow Mask (DO)—If the DO bit is 0, an Out of Range trap occurs when overflow of a signed or unsigned 32-bit result occurs during a DIVIDE or DIVIDU instruction, respectively. If the DO bit is 1, an Out of Range trap does not occur for overflow during integer divide operations.

The DIVIDE and DIVIDU instructions always cause an Out of Range Trap upon division by zero, regardless of the value of the DO bit.

Bit 0: Integer Multiplication Overflow Exception Mask (MO)—If the MO bit is 0, an Out of Range trap occurs when overflow of a signed or unsigned 32-bit result occurs during a MULTIPLY or MULTIPLU instruction, respectively. If the MO bit is 1, an Out of Range trap does not occur for overflow during integer multiply operations.

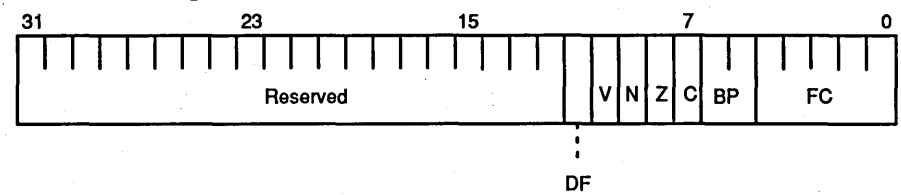
2.5 STATUS RESULTS OF INSTRUCTIONS

This section discusses the status information generated by arithmetic, logical and floating-point operations, and the special registers which contain this status information.

2.5.1 ALU Status (ALU, Register 132)

This unprotected special-purpose register (Figure 2-8) holds information about the outcome of Arithmetic/Logic Unit (ALU) operations as well as control for certain operations performed by the Execution Unit.

Figure 2-8 ALU Status Register



Bits 31–12: Reserved.

Bit 11: Divide Flag (DF)—The DF bit is used by the instructions that implement division. This bit is set at the end of the division instructions either to 1 or to the complement of the 33rd bit of the ALU. When a Divide Step instruction is executed, the DF bit determines whether an addition or subtraction operation is performed by the ALU.

Bit 10: Overflow (V)—The V bit indicates that the result of a signed, two's-complement ALU operation required more than 32 bits to represent the result correctly. The value of this bit is determined by exclusive-ORing the ALU carry-out with the carry-in to the most-significant bit for signed, two's-complement operations. This bit is not used for any special purpose in the processor and is provided for information only.

Bit 9: Negative (N)—The N bit is set with the value of the most-significant bit of the result of an arithmetic or logical operation. If two's-complement overflow occurs, the N bit does not reflect the true sign of the result. This bit is used in divide operations.

Bit 8: Zero (Z)—The Z bit indicates that the result of an arithmetic or logical operation is zero. This bit is not used for any special purpose in the processor, and is provided for information only.

Bit 7: Carry (C)—The C bit stores the carry-out of the ALU for arithmetic operations. It is used by the add-with-carry and subtract-with-carry instructions to generate the carry into the Arithmetic/Logic Unit.

Bits 6–5: Byte Pointer (BP)—The BP field holds a 2-bit pointer to a byte within a word. It is used by Insert Byte and Extract Byte instructions. The mapping of the pointer value to the byte position depends on the value of the Byte Order (BO) bit in the Configuration Register.

The most-significant bit of the BP field is used to determine the position of a half-word within a word for the Insert Half-Word, Extract Half-Word, and Extract Half-Word, Sign-Extended instructions. The mapping of the most-significant bit to the half-word position depends on the value of the BO bit in the Configuration Register.

The BP field is set by a Move To Special Register instruction with either the ALU Status Register or the Byte Pointer Register as the destination. It is also set by a load or store instruction if the Set Byte Pointer (SB) bit in the instruction is 1. A load or store sets the BP field with the complement of the Byte Order bit of the Configuration Register, for compatability with other 29K family processors.

Bits 4–0: Funnel Shift Count (FC)—The FC field contains a 5-bit shift count for the Funnel Shifter. The Funnel Shifter concatenates two source operands into a single 64-bit operand and extracts a 32-bit result from this 64-bit operand; the FC field specifies the number of bit positions from the most-significant bit of the 64-bit operand to the most-significant bit of the 32-bit result. The FC field is used by the EXTRACT instruction.

The FC field is set by a Move To Special Register instruction with either the ALU Status Register or the Funnel Shift Count Register as the destination.

2.5.2 Arithmetic Operation Status Results

The Arithmetic instructions modify the V, N, Z, and C bits. These bits are set according to the result of the operation performed by the instruction.

All instructions in the Arithmetic class—except for MULTIPLY, MULTM, DIVIDE, MULTIPLU MULTMU, and DIVIDU—perform an add. In the case of subtraction, the subtract is performed by adding the two's-complement or one's-complement of an operand to the other operand. The multiply step and divide step operations also

perform adds, again possibly complementing one of the operands before the operation is performed. In general, the status bits are based on the results of the add.

If two's-complement overflow occurs during the add, the V bit of the ALU Status Register is set; otherwise it is reset. Two's-complement overflow occurs when the carry-in to the most-significant bit of the intermediate result differs from the carry-out. When this occurs, the result cannot be represented by a signed word integer. Note that the V bit always is set in this manner, even when the result is unsigned.

The N bit of the ALU Status Register is set to the value of the most-significant bit of the result of the add. Note that the divide step and multiply step operations may shift the result after the operation is performed. In the cases where shifting occurs, the N bit may not agree with the result that is written into a general-purpose register, since the N bit is based only on the result of the add, not on the shift.

If the result of the add causes a zero word to be written to a general-purpose register, the Z bit of the ALU Status Register is set; otherwise, it is reset. The Z bit always reflects the result written into a general-purpose register; if shifting is performed by a multiply or divide step, the Z bit reflects the shifted value.

If there is a carry out of the add operation, the C bit is set; otherwise it is reset.

2.5.3 Logical Operation Status Results

The Logical instructions modify the N and Z bits. These bits are set according the result of the instruction. The V and C bits are meaningless in regard to the logical instructions, so they are not modified.

The N bit of the ALU Status Register is set to the value of the most-significant bit of the result of the logical operation.

If the result of the logical operation is a zero word, the Z bit of the ALU Status Register is set; otherwise, it is reset.

2.5.4 Floating-Point Status Results

The floating-point instructions check for a number of exceptional conditions, and report these exceptions by setting bits of the Floating-Point Status Register. The exceptional conditions also may cause traps, depending on the state of mask bits in the Floating-Point Environment Register. There are two groups of status bits in the Floating-Point Status Register: trap status bits and sticky status bits. When an exception is detected, the Am29030 and Am29035 microprocessors set the trap status bit and/or the sticky status bit associated with the exception, depending on the corresponding exception mask bit and on whether or not a trap occurs. The sticky status bit is set whenever the corresponding exception is masked, regardless of whether or not a trap occurs. A trap status bit is set whenever a trap occurs, regardless of the state of the corresponding mask bit.

A trap status bit is reset when a trap occurs and the indicated status does not apply to the trapping operation. A sticky status bit is reset only by software.

2.5.5 Floating-Point Status (FPS, Register 162)

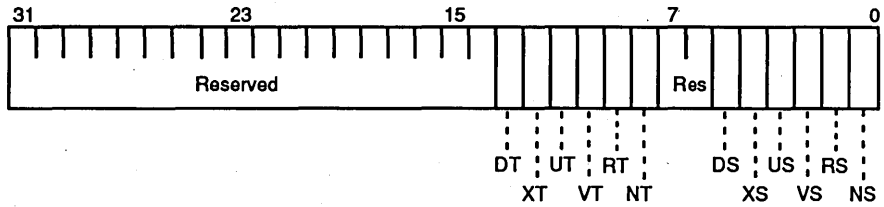
This unprotected special-purpose register (Figure 2-9) contains status bits indicating the outcome of floating-point operations.

The floating-point status bits are divided into two groups. The first group consists of the sticky status bits (DS, XS, US, VS, RS, and NS), which, once set, remain set until

explicitly cleared by a Move-to-Special-Register (MTSR) or Move-to-Special-Register-Immediate (MTSRIM) instruction. Only those sticky status bits corresponding to masked exceptions are updated. The update occurs at the end of instruction execution.

The second group consists of the trap status bits (DT, XT, UT, VT, RT, and NT), which report the status of an operation for which a Floating-Point Exception trap is taken. These bits are updated only by an operation which takes a trap as a result of an unmasked Floating-Point Exception; all other operations leave these bits unchanged. A trap status bit is updated regardless of the state of the corresponding exception mask in the Floating-Point Environment Register.

Figure 2-9 Floating-Point Status



Bits 31–14: Reserved.

Bit 13: Floating-Point Divide By Zero Trap (DT)—The DT bit is set when a Floating-Point Exception trap occurs, and the associated floating-point operation is a divide with a zero divisor and a non-zero, finite dividend. Otherwise, this bit is reset when a Floating-Point Exception trap occurs.

Bit 12: Floating-Point Inexact Result Trap (XT)—The XT bit is set when a Floating-Point Exception trap occurs, and the result of the associated floating-point operation is not equal to the infinitely-precise result. Otherwise, this bit is reset when a Floating-Point Exception trap occurs.

Bit 11: Floating-Point Underflow Trap (UT)—The UT bit is set when a Floating-Point Exception trap occurs, and the result of the associated floating-point operation is too small to be expressed in the destination format. Otherwise, this bit is reset when a Floating-Point Exception trap occurs.

Bit 10: Floating-Point Overflow Trap (VT)—The VT bit is set when a Floating-Point Exception trap occurs, and the result of the associated floating-point operation is too large to be expressed in the destination format. Otherwise, this bit is reset when a Floating-Point Exception trap occurs.

Bit 9: Floating-Point Reserved Operand Trap (RT)—The RT bit is set when a Floating-Point Exception trap occurs, and the result of the associated floating-point operation is a reserved value. Otherwise, this bit is reset when a Floating-Point Exception trap occurs.

Bit 8: Floating-Point Invalid Operation Trap (NT)—The NT bit is set when a Floating-Point Exception trap occurs and the input operands to the associated floating-point operation produce an indeterminate result. Otherwise, this bit is reset when a Floating-Point Exception trap occurs.

Bits 7–6: Reserved.

Bit 5: Floating-Point Divide By Zero Sticky (DS)—The DS bit is set when the DM bit of the Floating-Point Environment Register is 1, the divisor of a floating-point division operation is a zero, and the dividend is a non-zero, finite number.

Bit 4: Floating-Point Inexact Result Sticky (XS)—The XS bit is set when the XM bit of the Floating-Point Environment Register is 1, and the result of a floating-point operation is not equal to the infinitely precise result.

Bit 3: Floating-Point Underflow Sticky (US)—The US bit is set when the UM bit of the Floating-Point Environment Register is 1, and the result of a floating-point operation is too small to be expressed in the destination format.

Bit 2: Floating-Point Overflow Sticky (VS)—The VS bit is set when the VM bit of the Floating-Point Environment Register is 1, and the result of a floating-point operation is too large to be expressed in the destination format.

Bit 1: Floating-Point Reserved Operand Sticky (RS)—The RS bit is set when the RM bit of the Floating-Point Environment Register is 1, and either one or more input operands to a floating-point operation is a reserved value or the result of a floating-point operation is a reserved value.

Bit 0: Floating-Point Invalid Operation Sticky (NS)—The NS bit is set when the NM bit of the Floating-Point Environment Register is 1, and the input operands to a floating-point operation produce an indeterminate result.

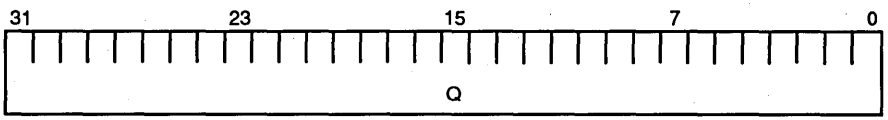
2.6 INTEGER MULTIPLICATION AND DIVISION

The Am29030 and Am29035 microprocessors do not directly support the instructions MULTIPLU, MULTMU, MULTIPLY, MULTM, DIVIDE, and DIVIDU. The processors are capable of performing these instructions as a sequence of multiply- or divide-steps, which are directly supported by hardware. A special register, Q, is used in conjunction with the SRCA and SRCB operands to execute the multiply- or divide-step. This section describes the Q register and discusses the general method for multiplication and division.

2.6.1 Q (Q, Register 131)

The Q Register is an unprotected special-purpose register (Figure 2-10).

Figure 2-10 Q Register



Bits 31–0: Quotient/Multiplier (Q)—During a sequence of divide steps, this field holds the low-order bits of the dividend; it contains the quotient at the end of the divide. During a sequence of multiply steps, this field holds the multiplier; it contains the low-order bits of the result at the end of the multiply.

For an integer divide instruction, the Q field contains the high-order bits of the dividend at the beginning of the instruction, and contains the remainder upon completion of the instruction.

2.6.2 Multiplication

The processor performs integer multiplication by a series of multiply step instructions. Note that when the product of a constant and a variable is to be computed, a more efficient sequence of shift and add instructions usually can be found.

If a program requires the multiplication of two integers, the required sequence of multiply steps may be executed in-line or executed in a multiply routine called as a procedure. It may be beneficial to precede a full multiply procedure with a routine to discover whether or not the number of multiply steps may be reduced. This reduction is possible when the operands do not use all of the available 32 bits of precision.

The following routine multiplies two 32-bit signed integers, giving a 64-bit result. Unsigned multiplication can be performed by substituting the MULU instruction for the MUL and MULL instructions:

```
; 32 bit * 32 bit –>64 bit signed multiply
; Input:    multiplicand in lr2, multiplier in lr3
; Output:   result most-significant word in gr96, result least-significant word in gr97

SMul64:
          mtsr    Q, lr3              ; put multiplier in the Q register
          mul     gr96, lr2, 0        ; perform initial multiply step
          .rep    30                  ; expand out 30 copies of the next instruction
                                      ;  in-line
          mul     gr96, lr2, gr96     ; total of 30 more multiply steps
          .endr
          mull    gr96, lr2,gr96      ; perform last sign correcting step
          mfsr    gr97, Q             ; get the least-significant result word
```

The following routine multiplies two 32-bit integers, returning a 32-bit result. It attempts to minimize the number of multiply-step instructions by checking the input operands. It is coded as a subroutine, with pointers to its operands passed in the indirect pointers IPC, IPA, and IPB. This allows the routine to operate on any combination of registers, rather than forcing the operands to be in fixed registers:

```
; 32 bit * 32 bit –> 32 bit signed or unsigned multiply called by:

;         call    tpc, MUL32                  ; call the multiply routine
;         setip   dst_reg, src1_reg, src2_reg ; passing pointers to the operand registers
;                                             ; in the delay slot


; Input:    operands in the registers pointed to by indirect-pointer registers IPA  and IPB
; Output:   result least-significant word in the register pointed to by IPC
; Used:     return address in tpc, special registers Q and FC
; Destroyed:        previous contents of registers tpc, Temp0 – Temp2
; Symbolic register names:
          .reg    Temp0, gr116
          .reg    Temp1, gr119
          .reg    Temp2, gr120
          .reg    tpc,  gr122
          .word   0x00200000            ; Debugger tag word

Mul32:
; need an instruction to separate SETIP (probably last instruction) from access of indirect
; pointers
          mtsrim  FC,8                  ; useful when if one operand is 8-bit
          or      Temp0, gr0, 0         ; copy value of IPA register

; next we'll check to see that the operand with the most leading zeros becomes the multiplier
          cpgtu   Temp1,gr0,gr0
          jmpf    Temp1,do8             ; the operands are already ordered correctly
          or      Temp1,Temp1,gr0       ; if we jump, Temp1 holds 0, so this copies
                                        ; the value of the IPB register
```

```
            const   Temp0,0                 ; we must swap the operands
            or      Temp0,Temp0,gr0
            or      Temp1,gr0,0
do8:
            cpleu   Temp2,Temp1,0x7f        ; less than 8 bits?
            jmpf    Temp2,do16              ; no, check for 16 bits
            mtsr    Q,Temp0
            mulu    Temp0,Temp1,0
            .rep    7                       ; expand out 7 copies of the next instruction
                                            ;  in-line
            mulu    Temp0,Temp1,Temp0       ; total of 7 more multiply steps
            .endr
; the top 24 bits of the result are in the lower 24 bits of Temp0, and the bottom 8 bits are in the
;  top of Q
            mfsr    Temp1,Q
            jmpi    tpc                     ; return to the calling routine
            extract gr0,Temp0,Temp1         ; extract  the result in the delay-slot of the
                                            ; jump
do16:
            const   Temp2,0x7fff            ; less than 16 bits?
            cplequ  Temp2,Temp0,Temp2
            jmpf    Temp2,do32              ; no, perform all 32 steps
            mulu    Temp0,Temp1,0           ; perform initial multiply-step
            .rep    15                      ; expand out 15 copies of next instruction
                                            ; in-line
            mulu    Temp0,Temp1,Temp0       ; total of 15 more multiply-steps
            .endr
; the top 16 bits of the result will be in the lower16 bits of Temp0,the bottom 16 bits in the top
; of Q
            mtsrim  FC,16                   ; extract on bit-16 boundary
            mfsr    Temp1,Q
            jmpi    tpc                     ; return to the calling routine
            extract gr0,Temp0,Temp1         ; extracting the result in the delay-slot of the
                                            ; jump
do32:
            mulu    temp0,Temp1,0           ; perform initial step
            .rep    31                      ; expand out 32 copies of the next instruction
                                            ; in-line
            mulu    Temp0,Temp1,Temp0       ; total of 31 more multiply steps
            .endr
            jmpi    tpc                     ; return to calling routine
            mfsr    gr0,Q                   ; copy the result to the return register in the
                                            ; delay slot
```

2.6.3 Division

The processors perform integer division by a series of divide step instructions. When the divisor is a power of 2, and the dividend is unsigned, the divide should be accomplished by a right shift.

If a program requires the division of two integers, the required sequence of divide steps may be executed in-line or executed in a divide routine called as a procedure. It may be beneficial to precede a full divide procedure with a routine to discover whether or not the number of divide steps may be reduced. This reduction is possible when the operands do not use all of the available 32 bits of precision.

The following routine divides a 64-bit, unsigned dividend by a 32-bit unsigned divisor:

```
; 64 bit / 32 bit → 32 bit unsigned divide
; Input:    most-significant dividend word in lr2, least-significant dividend word in lr3,
;           divisor in lr4
; Output:   quotient in gr96, remainder in gr97

UDiv64:
          mtsr      Q, lr3              ; put least-significant word of the dividend in
                                        ; the Q register
          div0      gr97, lr2           ; perform initial divide step

          .rep      31                  ; expand out 31 copies of the next
                                        ; instruction in-line
          div       gr97, gr97, lr4     ; total of 30 more divide steps
          .endr

          divl      gr97, gr97, lr4     ; perform last step
          divrem    gr97, gr97, lr4     ; compute remainder
          mfsr      gr96, Q             ; get the quotient
```

The following routine divides a 32-bit unsigned dividend by a 32-bit unsigned divisor:

```
; 32 bit / 32 bit → 32 bit unsigned divide
; Input:    dividend word in lr2, divisor in lr3
; Output:   quotient in gr96, remainder in gr97

UDiv32:
          mtsr      Q, lr2              ; put the dividend in the Q register
          div0      gr97, 0             ; perform initial divide step, zeroing out
                                        ; the upper bits of the dividend

          .rep      31                  ; expand out 31 copies of the next
                                        ; instruction in-line
          div       gr97, gr97, lr4     ; total of 30 more divide steps
          .endr

          divl      gr97, gr97, lr4     ; perform last step
          divrem    gr97, gr97, lr4     ; compute remainder
          mfsr      gr96, Q             ; get the quotient
```

The following routine divides a 32-bit signed dividend by a 32-bit signed divisor. It also traps division by zero. Because the divide-step instructions only operate on unsigned operands, extra code is required to perform sign checking and conversion:

```
; 32 bit / 32 bit signed divide, called by:

;         call      tpc, SDiv32         ; call the divide routine
;         setip     dst_reg, src1_reg, src2_reg
                                        ; passing pointers to the operand
                                        ; registers in the delay slot
; Input:    dividend and divisor in the registers pointed to by the indirect-pointer
;           registers IPA and IPB
; Output:   result quotient in the register pointed to by IPC, remainder left in Temp0
; Used:     return address in tpc, special register Q
; Destroyed: previous contents of registers tpc, Temp0 – Temp2
; Symbolic register names:
          .reg      Temp0, gr116
          .reg      Temp1, gr119
          .reg      Temp2, gr120
          .reg      tpc, gr122
          .word     0x00200000          ; Debugger tag word
```

```
SDiv32:
        const   Temp1, 0
        asneq   V_DIVBYZERO, Temp1, gr0
                                        ; check for divide by zero with an assert
        add     Temp0, gr0, 0           ; get dividend from indirect pointer
        jmpf    Temp0, pdividend        ; is it negative (jmpf is also "jmppos")
        add     Temp2, Temp1, gr0       ; get divisor from indirect pointer
        const   Temp1, 3                ; set negative result and remainder flags
        subr    Temp0, Temp0, 0         ; make dividend positive
pdividend:
        jmpf    Temp2, pdivisor         ; is divisor negative?
        mtsr    Q, Temp0                ; copy dividend to Q register in delay slot
                                        ; of the jump
        xor     Temp1, Temp1, 1         ; turn off negative result flag
        subr    Temp2, Temp2, 0         ; make divisor positive
pdivisor:
        div0    Temp0, 0                ; initialize

        .rep    31                      ; expand out 31 copies of the next
                                        ; instruction in-line
        div     Temp0, Temp0, Temp2     ; total of 30 more divide steps
        .endr

        divl    Temp0, Temp0, Temp2     ; perform last divide step
        divrem  Temp0, Temp0, Temp2     ; get positive remainder
        mfsr    Temp2, Q                ; get positive quotient
        sll     Temp1, Temp1, 30        ; copy negative remainder flag to test bit
        jmpf    Temp1, premainder       ; if it is not set, remainder is ok
        sll     Temp1, Temp1, 1         ; copy negative result flag to test bit
        subr    Temp0, Temp0, 0         ; negate remainder
premainder:
        jmpfi   Temp1, tpc              ; return to caller if result is positive
        add     gr0, Temp2, 0           ; copying quotient to the result register
                                        ; in the delay slot
        jmpi    tpc                     ; else return to caller,
        subr    gr0, Temp2, 0           ; negating the quotient in the delay slot
```

2.7 I NEED AN INSTRUCTION TO...

This section discusses topics of general concern in the implementation of applications programs.

2.7.1 Run-Time Checking

The assert instructions provide programs with an efficient means of comparing two values and causing a trap when a specified relation between the two values is not satisfied. The instructions assert that some specified relation is true and trap if the relation is not true. This allows run-time checking—such as checking that a computed array index is within the boundaries of the storage for an array—to be performed with a minimum performance penalty.

Assert instructions are available for comparing two signed or unsigned operands. The following relations are supported: equal-to, not-equal-to, less-than, less-than or equal-to, greater-than, and greater-than-or-equal-to.

The assert instructions specify a vector number for the trap. However, only vector numbers 64 through 255 (inclusive) may be specified by User-mode programs. If a

User-mode assert instruction causes a trap, and the vector number is between 0 and 63 inclusive, a Protection Violation trap occurs, instead of the specified trap.

Since the assert instructions allow the specification of the vector number, several traps may be defined in the system for different situations detected by the assert instructions.

2.7.2 Operating-System Calls

An applications program can request a service from the operating system by using the following instruction:

```
asneq System_Routine, gr1, gr1
```

This instruction always creates a trap, since it attempts to assert that the content of a register is not equal to itself (the register number used here is irrelevant, as long as the register is otherwise accessible).

The System_Routine vector number specified by the instruction invokes the execution of the operating system routine that provides the requested service. This vector number may have any value between 64 and 255, inclusive (vector numbers 0 through 63 are pre-defined or reserved). Thus, as many as 192 different operating-system routines may be invoked from the applications program.

In cases where the indirect pointers may be used, the EMULATE instruction allows two operand/result registers to be specified to the operating-system routine. The instruction is as follows:

```
emulate System_Routine, lr3, lr6
```

In this case, the System_Routine vector number performs the same function as in the previous example. Here, however, LR3 and LR6 are specified as operand registers and/or result registers (these particular registers are used only for illustration). The operating-system routine has access to these registers via the indirect pointers, which allows flexible communication.

2.7.3 Multiprecision Integer Operations

The processor allows the Carry (C) bit of the ALU Status Register to be used as an operand for add and subtract instructions. This provides for the addition and subtraction of operands which are greater than 32 bits in length. For example, the following code implements a 96-bit addition with signed overflow detection.

```
add     lr7, gr96, lr2
addc    lr8, gr97, lr3
addcs   lr9, gr98, lr4
```

Global registers GR96-GR98 contain the first operand, local registers LR2–LR4 contain the second operand, and local registers LR7–LR9 contain the result. The first two add instructions set the C bit, which is used by the second two instructions. If the addition causes a signed overflow, then an Out of Range trap occurs; overflow is detected by the final instruction.

2.7.4 Complementing a Boolean

To complement a Boolean in the processor's format, only the most-significant bit of the Boolean word should be considered, since the least-significant 31 bits may or may not be zeros. This is accomplished by the following instruction:

```
cpge gr96, gr96, 0
```

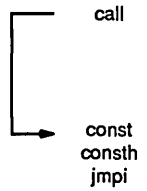
The Boolean is in GR96 in this example. This instruction is based on the observation that a Boolean TRUE is a negative integer, since the Boolean bit coincides with the integer sign bit. If the operand of this instruction is a negative integer (i.e., TRUE), the result is the Boolean FALSE. If the operand is non-negative (i.e., the Boolean FALSE), the result is TRUE.

2.7.5 Large Jump and Call Ranges

The 16-bit relative branch displacement provided by processor instructions is sufficient in the majority of cases. However, addresses with a greater range occasionally are needed. In these cases, the CONST and CONSTH instructions generate the large branch-target address in a register. An indirect jump or call then uses this address to branch to the appropriate location.

When program modules are compiled separately, the compiler cannot determine whether or not the 16-bit displacement of a CALL instruction is sufficient to reach an external procedure, even though it is sufficient in most cases. Instead of generating instructions for the worst case (i.e., the CONST, CONSTH, and CALLI described above), it is more efficient to generate a CALL as if it were appropriate, with the worst-case sequence (in this case, CONST, CONSTH, and JMPI) also appearing in the generated code somewhere (e.g., at the end of a compiled procedure).

When the above scheme is used, the linker is able to determine whether or not the CALL is sufficient. If it is not, the CALL can be retargeted to the worst-case sequence in the code. In other words, when the CALL is not sufficient, the linker causes the execution sequence to be:



In this manner, the longer execution time for the call occurs only when necessary.

2.7.6 NO-OPs

When a NO-OP is required for proper operation (e.g., as described in Section 5.6), it is important that the selected instruction not perform any operation, regardless of program operating conditions. For example, the NO-OP cannot access general-purpose registers, because a register may be protected from access in some situations. The suggested NO-OP is:

```
aseq 0x40, gr1, gr1
```

This instruction asserts that the Stack Pointer (GR1) is equal to itself. Since the assertion is always true, there is no trap. Note also that the Stack Pointer cannot be protected, and that the assert instruction cannot affect any processor state.

2.8 VIRTUAL ARITHMETIC PROCESSOR

In order to be object-code compatible with present and future implementations of the 29K family of microprocessors, the Am29030 and Am29035 microprocessors provide a virtual arithmetic interface. A virtual interface is the means by which a processor appears to perform functions that it does not actually perform. In the case of the

Am29030 and Am29035 virtual arithmetic interface, the processor defines arithmetic instructions, control, and status which are not directly supported by hardware, but which are implemented by system software.

2.8.1 Trapping Arithmetic Instructions

The processor does not incorporate hardware to directly support floating-point operations, nor does it directly support full multiply and divide instructions. However, instructions to perform these operations are included in the instruction set. These instructions are included for compatibility with processor implementations, such as the Am29050 microprocessor, that include hardware to perform these operations.

In application programs that must be fully object-code compatible across several processor versions—while taking advantage of the performance of the versions having arithmetic hardware—the defined instructions should be used to perform floating-point, multiplication and division operations.

In the Am29030 and Am29035 microprocessors, the Floating-Point, CLASS, CONVERT, MULTIPLY, MULTM, MULTIPLU, MULTMU, DIVIDE, DIVIDU, and SQRT instructions cause traps. The indirect pointers are set at the time the trap occurs, so that a trap handler can gain access to the operands of the instruction and can determine where the result is to be stored. A trap handler can directly emulate the execution of the instruction.

2.8.2 Virtual Registers

The processor does not incorporate hardware to directly support the Floating-Point Environment Register (FPE), Integer Environment Register (INTE), or Floating-Point Status Register (FPS). When one of these registers is referenced by a MTSR/MFSR instruction (or a variant), a Protection Violation trap occurs. The Protection Violation trap handler must establish that the faulting instruction is a MTSR/MFSR and that the register specified by the instruction is one of the registers supported by the virtual interface. This is accomplished by obtaining the faulting instruction from memory and examining the OPCODE and SRC/DEST fields. The trap handler then simulates the operation of the register.

2.9 MULTIPROCESSING

The Am29030 and Am29035 microprocessors provide several facilities for the implementation of multi-programming and multi-processing systems. These facilities help provide mutual exclusion, synchronization, and communication between multiple processes, whether these processes execute on a single processor or multiple processors.

Binary semaphores are supported by the Load and Set (LOADSET) instruction. This instruction loads the contents of an external location into a register and automatically sets the contents of the location to the integer –1. This instruction requires no special hardware support in the system, since all sequencing is performed by the processor. Also, the LOADSET is available to User-mode programs. This eliminates the overhead of an operating-system call in the use of binary semaphores.

The instructions Load and Lock (LOADL) and Store and Lock (STOREL) support the locking of external devices and memories, or the locking of particular locations within an external device or memory. This prevents access by any process or processor other than the one that performed the lock, and provides the flexibility of locking in a

manner appropriate to the system and application. The LOADL and STOREL instructions are available to User-mode programs.

To indicate that a LOADL or STOREL is being executed, the processor asserts the $\overline{\text{LOCK}}$ output during the external access (see Section 10.1 and 10.6 for a description of the $\overline{\text{LOCK}}$ output). Since the processor cannot directly control the behavior of external devices and memories, system hardware must support locking, if required.

Note that the protocol for locking and unlocking devices and memories must be defined by the system. For example, the protocol may be defined such that a LOADL locks the device or memory, and a STOREL unlocks the device or memory. Between the execution of the LOADL and the STOREL, the device can be accessed with any combination of normal loads and stores.

For the implementation of a general-purpose exclusion, synchronization, and/or communication scheme, the processor allows Supervisor-mode programs to set the Lock (LK) bit in the Current Processor Status Register (see Section 8.1.1). This bit activates the $\overline{\text{LOCK}}$ pin and prevents the processor from relinquishing the bus to another bus master. If another master already has control of the channel when the LK bit is set, the LK bit does not take affect until control of the bus is returned to the processor.

The LK bit allows a Supervisor-mode program to execute with mutual exclusion for any sequence of instructions. However, because interrupts must also be disabled for true exclusion, this may have a negative impact on system performance if used improperly.

Bit 20: Set Byte Pointer/Sign Bit (SB)—If the SB bit is 1 for a load, the loaded byte or half-word is sign-extended in the destination register; if the SB bit is 0, the byte or half-word is zero-extended. When the SB bit is 1 for either a load or store, each bit of the Byte Pointer Register is written with the complement of the Byte Order bit of the Configuration Register. The Byte Pointer Register is set in this case to provide software compatibility across different types of memory systems and 29K family processors. If the SB bit is 0, the Byte Pointer Register is not affected.

Bit 19: User Access (UA)—The UA bit allows programs executing in the Supervisor mode to emulate User-mode accesses. This allows checking of the authorization of an access requested by a User-mode program. It also causes address translation (if applicable) to be performed using the PID field of the MMU Configuration Register, rather than the fixed Supervisor-mode process identifier zero.

If the UA bit is 1 for a Supervisor-mode load or store, the access associated with the instruction is performed in the User mode. In this case, the User mode affects only MMU protection-checking, the SUP/$\overline{US}$ output, and the use of the PID field in translation; it has no effect on the registers that can be accessed by the instruction. If the UA bit is 0, the program mode for the access is controlled by the SM bit.

If the UA bit is 1 for a User-mode load or store, a Protection Violation trap occurs.

Bits 18–16: Option (OPT)—This field is placed on the OPT(2–0) outputs during the address cycle of the access. There is a one-to-one correspondence between the OPT field and the OPT(2–0) outputs; that is, the most-significant OPT bit is placed on OPT2, and so on.

The OPT field controls system functions as described in Section 3.3.6.

Bits 15–8: (RA)—The data for the access is written into the general-purpose register RA for a load, and is supplied by register RA for a store.

Bits 7–0: (RB or I)—In load and store instructions, the RB or I field specifies the address for the access. The address is either the content of a general-purpose register with register number RB, or a constant value I (zero-extended to 32 bits). The M bit of the operation code (bit 24) determines whether the register or the constant is used.

Load and store operations are overlapped with the execution of instructions that follow the load or store instruction. Only one load or store may be in progress on any given cycle. If a load or store instruction is encountered while another load or store operation is in progress, the processor enters the Pipeline Hold mode until the first operation completes (see Section 5.2).

3.3.3 Load Operations

The processors provide the following instructions for performing load operations: Load (LOAD), Load and Lock (LOADL), Load and Set (LOADSET), and Load Multiple (LOADM). All of these instructions transfer data from an external device or memory into one or more general-purpose registers.

The LOADL instruction supports the implementation of device and memory interlocks in a multi-processor configuration. It activates the $\overline{LOCK}$ output during the address cycle of the access.

The LOADSET instruction implements a binary semaphore. It loads a general-purpose register and atomically writes the accessed location with a word which has 1 in every bit position (that is, the write is indivisible from the read). The $\overline{LOCK}$ output is asserted during both the read and write access. Note that, if address translation is enabled for the LOADSET instruction, the MMU memory-protection bits must allow

both the read and write access. If either the read or write access is not allowed, neither access is performed.

The LOADM instruction loads a specified number of registers from sequential addresses, as explained below in Section 3.3.5.

Load operations are overlapped with the execution of instructions that follow the load instruction. The processor detects any dependencies on the loaded data that subsequent instructions may have and, if such a dependency is detected, enters the Pipeline Hold mode until the data is returned by the external device or memory. If a register that is the target of an incomplete load is written with the result of a subsequent instruction, the processor does not write the returning data into the register when the load completes; the Not Needed (NN) bit in the Channel Control Register is set in this case.

3.3.4 Store Operations

The processors provide the following instructions for performing store operations: Store (STORE), Store and Lock (STOREL), and Store Multiple (STOREM). All of these instructions transfer data from one or more general-purpose registers to an external device or memory.

The STOREL instruction supports the implementation of device and memory interlocks in a multi-processor configuration. It activates the $\overline{\text{LOCK}}$ output during the address cycle of the access.

The STOREM instruction stores a specified number of registers to sequential addresses, as explained below.

Store operations are overlapped with the execution of instructions that follow the store instruction. However, no data dependencies can exist, since the store prevents any subsequent load or store accesses until it completes.

3.3.5 Multiple Accesses

The Load Multiple (LOADM) and Store Multiple (STOREM) instructions move contiguous words of data between general-purpose registers and external devices and memories. The number of transfers is determined by the Load/Store Count Remaining Register.

The Load/Store Count Remaining (CR) field in the Load/Store Count Remaining Register specifies the number of transfers to be performed by the next LOADM or STOREM executed in the instruction sequence. The CR field is in the range of 0 to 255, and is zero-based: a count value of 0 represents one transfer, and a count value of 255 represents 256 transfers. The CR field also appears in the Channel Control Register.

Before a LOADM or STOREM is executed, the CR field is set by a Move To Special Register. A LOADM or STOREM uses the most-recently written value of the CR field. If an attempt is made to alter the CR field, and the Channel Control Register contains information for an external access that has not yet completed, the processor enters the Pipeline Hold mode until the access completes. Note that since the CR is set independently of the LOADM and STOREM, the CR field may represent valid state of an interrupted program even if the Contents Valid (CV) bit of the Channel Control Register is 0 (see also Section 8.6.2).

Because of the pipelined implementation of LOADM and STOREM, at least one instruction (e.g., the instruction that sets the CR field) must separate two successive LOADM and/or STOREM instructions.

After the CR field is set, the execution of a LOADM or STOREM begins the data transfer. As with any other load or store operation, the LOADM or STOREM waits until any pending load or store operation is complete before starting. The LOADM instruction specifies the starting address and starting destination general-purpose register. The STOREM instruction specifies the starting address and the starting source general-purpose register.

During the execution of the LOADM or STOREM instruction, the processor updates the address and register number after every access, incrementing the address by 4 and the register number by 1. This continues until either all accesses are completed or an interrupt or trap is taken.

For a load-multiple or store-multiple address sequence, addresses wrap from the largest possible value (hexadecimal FFFFFFFC) to the smallest possible value (hexadecimal 00000000).

The processors increment absolute register numbers during the load-multiple or store-multiple sequence. Absolute-register numbers wrap from 127 to 128 and from 255 to 128. Thus, a sequence that begins in the global registers may move to the local registers, but a sequence that begins in the local registers remains in the local registers. Also, note that the local registers are addressed circularly.

The normal restrictions on register accesses apply for the load-multiple and store-multiple sequences. For example, if a protected general-purpose register is encountered in the sequence for a User-mode program, a Protection Violation trap occurs.

Intermediate addresses are stored in the Channel Address Register, and register numbers are stored in the Target Register (TR) field of the Channel Control Register. For the STOREM instruction, the data for every access is stored in the Channel Data Register (this register also is set during the execution of the LOADM instruction, but has no interpretation in this case). The CR field is updated on the completion of every access, so that it indicates the number of accesses remaining in the sequence.

Load-multiple and store-multiple operations are indicated by the Multiple Operation (ML) bit in the Channel Control Register. The ML bit is used to restart a multiple operation on an interrupt return; if it is set independently by a Move To Special Register before a load or store instruction is executed, the results are unpredictable.

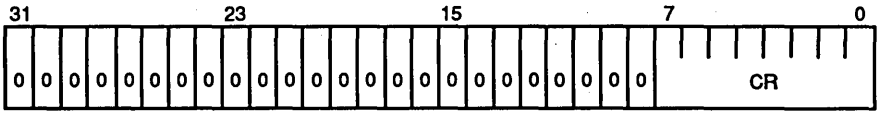
While a multiple load or store is executing, the processor is in the Pipeline Hold mode, suspending any subsequent instruction execution until the multiple access completes. If an interrupt or trap is taken, the Channel Address, Channel Data, and Channel Control registers contain the state of the multiple access at the point of interruption. The multiple access may be resumed at this point, at a later time, by an interrupt return.

The processors attempt to complete multiple accesses using the burst-mode capability of the bus (see Section 10.4.10). For this reason, multiple accesses of individual bytes and half-words is not supported. If the external device or memory cannot support burst-mode accesses, a sequence of simple single accesses are performed. If the address sequence causes a virtual page-boundary crossing, the processor preempts the burst-mode access, translates the address for the new page, and re-establishes the burst-mode access using the new physical address.

3.3.5.1 LOAD/STORE COUNT REMAINING (CR, Register 135)

This unprotected special-purpose register (Figure 3-8) provides alternate access to the CR field in the Channel Control Register.

Figure 3-8 **Load/Store Count Remaining Register**



Bits 31–8: Zeros.

Bits 7–0: Load/Store Count Remaining (CR)—The CR field indicates the remaining number of transfers for a load-multiple or store-multiple operation that encountered an exception or was interrupted before completion. This number is zero-based; for example, a value of 28 in this field indicates that 29 transfers remain to be completed.

This register allows a User-mode program to change the CR field in the Channel Control Register without affecting other fields in the Channel Control Register, and is used to initialize the value before a Load Multiple or Store Multiple instruction is executed.

3.3.5.2 MOVEMENT OF LARGE DATA BLOCKS

The movement of large blocks of data—for example, to perform a memory-to-memory move—can be performed by an alternating series of loads and stores. However, it is typically more efficient to move large blocks of data by using an alternating series of Load Multiple and Store Multiple instructions. These instructions take better advantage of the data-movement capabilities of the processor, though they require the use of a larger number of registers.

During data movement, it is possible to perform alignment operations by a series of EXTRACT instructions between the Load Multiple and Store Multiple. Also, since the Load Multiple and Store Multiple are interruptible, these instructions may be used to move large amounts of data without affecting interrupt latency.

3.3.6 Option Bits

The Option field in the load and store instructions supports system functions, such as byte and half-word accesses. The definition of this field for a load or store, depending on the AS bit of the instruction, is as follows:

AS	OPT2	OPT1	OPT0	Meaning
x	0	0	0	Word-length access
x	0	0	1	Byte access
x	0	1	0	Half-word access
0	1	1	0	Hardware-development system accesses
	—All Others —			Reserved

Note that some of these encodings do not affect processor operation and could have other interpretations in a particular system. Non-standard uses of the OPT field have an implication on the portability of software between different systems.

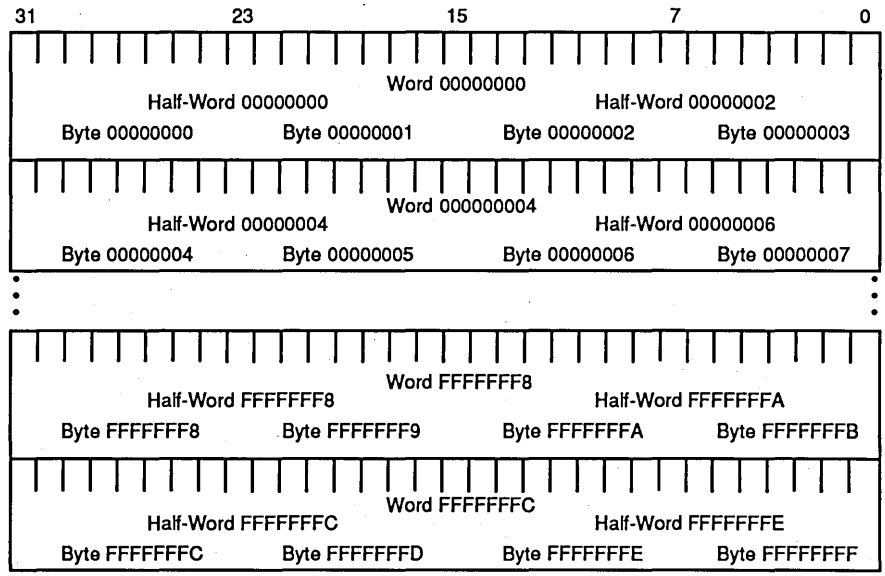
3.3.7 Addressing and Alignment

3.3.7.1 BYTE AND HALF-WORD ADDRESSING

The Am29030 and Am29035 microprocessors generate word-oriented byte addresses for accesses to external devices and memories. Addresses are word-oriented because loads, stores, and instruction fetches access words. However, addresses are byte addresses because they permit byte selection within accessed words. For load and store operations, the processor provides for using the least-significant address bits to access bytes and half-words within external words.

For all external byte and half-word accesses, the selection of a byte within an external word is determined by the two least-significant bits of an address and the Byte Order (BO) bit of the Configuration Register. The selection of a half-word within an external word is determined by the next-to-least significant bit of an address and the BO bit. Figure 3-9 illustrates the addressing of bytes and half-words when the BO bit is 0 (big endian), and Figure 3-10 illustrates the addressing of bytes and half-words when the BO bit is 1 (little endian). In Figure 3-9 and Figure 3-10, addresses are represented in hexadecimal notation.

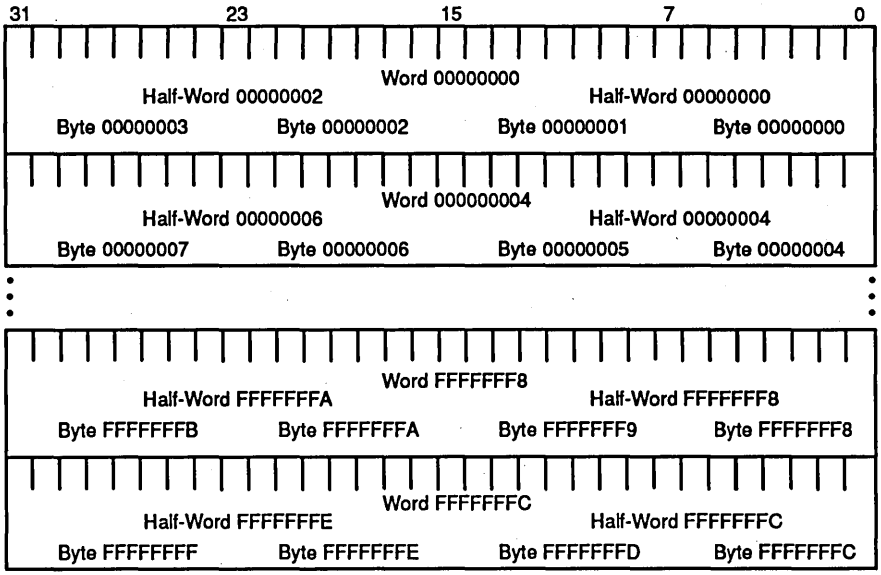
Figure 3-9 Byte and Half-Word Addressing with BO = 0 (Big Endian)



For all byte and half-word operations in the processor, the byte or half-word within a register is selected either by the two bits of the BP field or the two least-significant bits of an external address. The BO bit affects only the interpretation of the BP field and the two least-significant address bits.

If the BO bit is 0, bytes are ordered within words such that a 00 in the BP field or in the two least-significant address bits selects the high-order byte of a word, and a 11 selects the low-order byte. If the BO bit is 1, a 00 in the BP field or in the two least-significant address bits selects the low-order byte of a word, and a 11 selects the high-order byte.

Figure 3-10 Byte and Half-Word Addressing with BO = 1 (Little Endian)



If the BO bit is 0, half-words are ordered within words such that a 0 in the most-significant bit of the BP field or the next-to-least-significant address bit selects the high-order half-word, and a 1 selects the low-order half-word. If the BO bit is 1, a 0 in the most-significant bit of the BP field or the next-to-least-significant address bit selects the low-order half-word of a word, and a 1 selects the high-order half-word. Note that since the least-significant bit of the BP field or an address does not participate in the selection of half-words, the alignment of half-words is forced to half-word boundaries in this case.

3.3.7.2 BYTE AND HALF-WORD ACCESSES

During a load, the processor selects a byte or half-word from the loaded word depending on: the Option (OPT) bits of the load instruction, the Byte Order (BO) bit of the Configuration Register, and the two least-significant bits of the address (for bytes) or the next-to-least-significant bit of the address (for half-words). The selected byte or half-word is right-justified within the destination register. If the SB bit of the load instruction is 0, the remainder of the destination register is zero-extended. If the SB bit is 1, the remainder of the destination register is sign-extended with the sign bit of the selected byte or half-word.

During a store, the processor replicates the low-order byte or half-word in the source register into every byte and half-word position of the stored word. The processor generates the appropriate byte and/or half-word write enables, based on the OPT(2–0) signals and the two least-significant bits of the address, to write the byte or half-word in the selected device or memory. The SB bit does not affect the operation of a store, except for setting the BP field as described below.

If the SB bit is 1 for either a load or store, both bits of the BP field are set to the complement of the BO bit when the load or store is executed. This does not directly affect the load or store access, but supports compatibility for software developed for word-write-only systems and other 29K family processors.

3.3.7.3 ALIGNMENT OF WORDS AND HALF-WORDS

Since only byte addressing is supported, it is possible that the address for an access of a word or half-word is not aligned to the desired word or half-word. The Am29030 and Am29035 microprocessors either ignore or force alignment in most cases. However, some systems may require that unaligned accesses be supported, for compatibility reasons. Because of this, the Am29030 and Am29035 microprocessors provide an option to trap when a non-aligned access is attempted. This trap allows software emulation of the non-aligned accesses, in a manner which is appropriate for the particular system.

The detection of unaligned accesses is activated by a 1 in the Trap Unaligned Access (TU) bit of the Current Processor Status Register. Unaligned-access detection is based on the data length as indicated by the OPT field of a load or store instruction and on the two least-significant bits of the specified address. Only addresses for instruction/data memory accesses are checked; alignment is ignored for input/output accesses.

An Unaligned Access trap occurs only if the TU bit is 1 and any of the following combinations of OPT field and address bits is detected for a load or store to instruction/data memory:

OPT2	OPT1	OPT0	A1	A0	Meaning
0	0	0	1	0	Unaligned Word access
0	0	0	0	1	Unaligned Word access
0	0	0	1	1	Unaligned Word access
0	1	0	0	1	Unaligned Half-word access
0	1	0	1	1	Unaligned Half-word access

The trap handler for the Unaligned Access trap is responsible for generating the correct sequence of aligned accesses and performing any necessary shifting, masking and/or merging. Note that a virtual page-boundary crossing may also have to be considered.

3.3.7.4 ALIGNMENT OF INSTRUCTIONS

In the Am29030 and Am29035 microprocessors, all instructions are 32 bits in length and are aligned on word-address boundaries. The processor's Program Counter is 30 bits in length, and the least-significant two bits of processor-generated instruction addresses are always 00. An unaligned address can be generated by indirect jumps and calls. However, alignment is ignored by the processor in this case, and the processor expects the system to force alignment (i.e., by interpreting the two least-significant address bits as 00, regardless of their values).

CHAPTER 4

PROCEDURE LINKAGE

This chapter describes the run-time storage organization recommended for the Am29030 and Am29035 microprocessors and describes the use of the local registers to improve the performance of procedure calls. The presentation in this chapter is intended to be used as a guide in the implementation of software systems for the processor, not necessarily as a strict definition of how these systems should be implemented.

Programming languages that use recursive procedures, such as C and Pascal, generally use a stack to store data objects that are dynamically allocated at run-time. The organization of the run-time storage, including the run-time stack, determines how data objects are stored and how procedures are called at the machine level. The Am29030 and Am29035 microprocessors are designed to minimize the overhead of calling a procedure, passing parameters to a procedure, and returning results from a procedure. This chapter describes the run-time storage organization and procedure-calling conventions.

4.1 RUN-TIME STACK ORGANIZATION AND USE

A run-time stack consists of consecutive overlapping structures called activation records. An activation record contains dynamically allocated information specific to a particular activation (or call) of a procedure (such as local data objects). Because of recursion, multiple copies of a procedure may be active at any given time. Each active procedure has its own unique activation record, allocated somewhere on the run-time stack. The local variables required by a particular procedure activation are contained in the activation record associated with that activation. Thus, the local variables for different activations do not interfere with one another. A compiler generates the instructions to create and manage the run-time stack, and compiler-generated instructions are based on its existence.

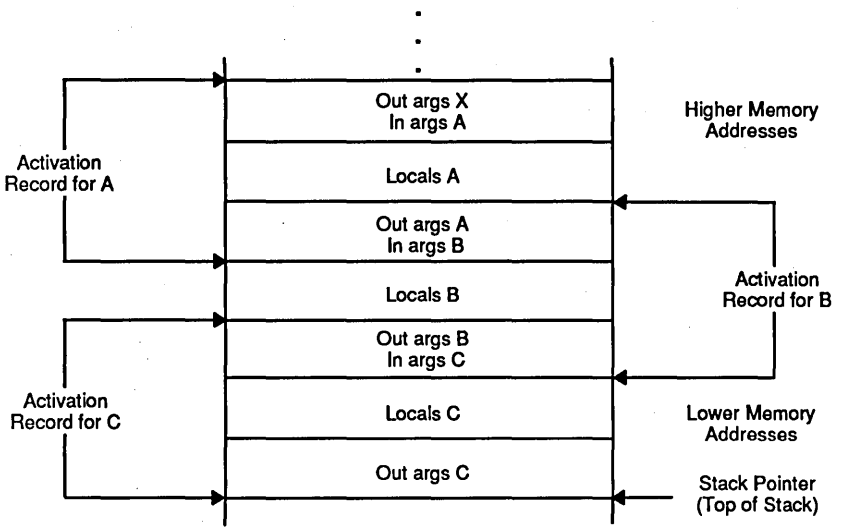
As an example, Figure 4-1 shows three activation records on a run-time stack. This stack configuration was generated by procedure A calling procedure B, which in turn called procedure C. The fact that procedure C is the currently active procedure is reflected by its activation record being on the top of the run-time stack. The Stack Pointer points to the top of procedure C's activation record.

In Figure 4-1, the storage areas labeled Out args and In args are the outgoing arguments area (for the caller) or the incoming arguments area (for the callee). These are shared between the caller procedure and the callee for the communication of parameters and results. The areas labeled locals contain storage for local variables, temporary variables (for example, for expression evaluation) and any other items required for the proper execution of the procedure.

4.1.1 Management Of The Run-time Stack

A run-time stack starts at a high address in memory and grows toward lower memory addresses as procedures are called. The bottom of the stack is the location, with a

Figure 4-1 Run-time Stack Example



high address, at which the stack starts; the top of the stack is the location, with a lower address, at which the most recent activation record has been allocated.

When a procedure is called, a new activation record might need to be allocated on the run-time stack. An activation record is allocated by subtracting from the stack pointer the number of locations needed by the new activation record. The stack pointer is decremented so that variables referenced during procedure execution are referenced in terms of positive offsets from the stack pointer.

When storage for an activation record is allocated, the number of storage locations allocated is the sum of the number of locations needed for:

1. Local variables;
2. Restarting the caller, such as locations for return addresses; and
3. Arguments of procedures that may be called in turn by the called procedure (the outgoing arguments area).

Note that, in some cases, no storage is required for one or more of the above items. Also, the incoming arguments area, though it is part of the activation record of the callee, is not allocated storage at this time, because this storage was allocated as the outgoing arguments area of the calling procedure.

An activation record is de-allocated, just prior to returning to the caller, by adding to the stack pointer the value that was subtracted during allocation.

In Am29030 and Am29035 microprocessors, run-time storage is actually implemented as two stacks: the Register Stack and the Memory Stack. Storage is allocated and de-allocated on these stacks at the same time. The Register Stack stores activation records associated with all active procedures (except leaf routines, as described later). The Memory Stack stores activation-record information that does not fit into the Register Stack or that must be kept in memory for other reasons (e.g., because of pointer dereferences). Both the Register Stack and the Memory Stack are stored in the external data memory. However, a portion of the Register Stack is kept in the

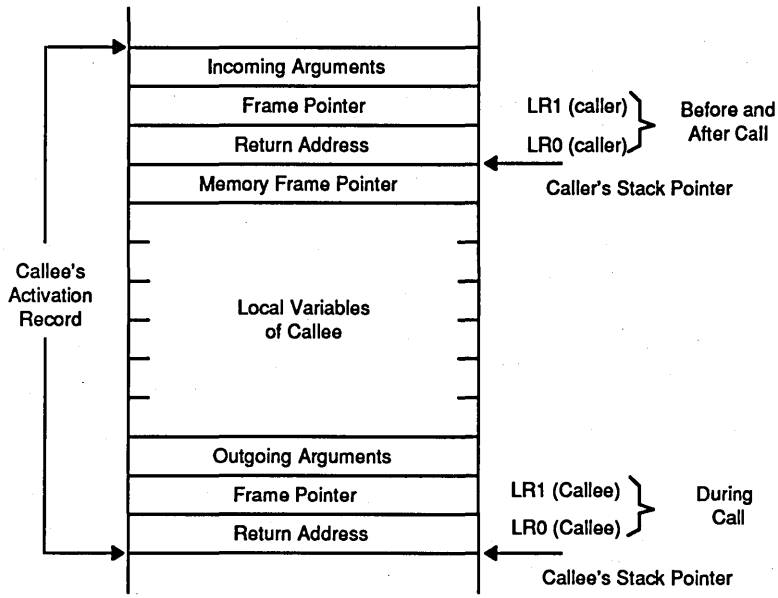
processor's local registers for performance. The term *stack cache* in this section refers to the use of the local registers to contain a portion of the Register Stack.

4.1.2 The Register Stack

The Register Stack contains activation records for active procedures (Figure 4-2). An activation record in the Register Stack stores the following information:

- Input arguments to the called procedure. This portion of the activation record is shared between a caller and the callee. It is allocated by the caller as part of the caller's activation record.
- The caller's frame pointer. This is the address of the lowest-addressed byte above the highest-address word of the caller's activation record, and is used to manage the Register Stack. This portion of the activation record is shared between a caller and the callee. It is allocated by the caller as part of the caller's activation record.
- The caller's return address. This is used to resume the execution of the caller after the called procedure terminates. This is also part of the caller's activation record.
- The memory frame pointer. This is the address of the top of the caller's Memory Stack (see below). This address is stored by the callee (if required), and used to restore the memory stack upon return.
- The local variables of the called procedure, if any.
- Outgoing parameters of the called procedure, if any.
- The frame pointer of the called procedure, if the procedure calls another procedure.
- The return address for the called procedure, if the procedure calls another procedure. This location is allocated in the Register Stack, and is used when the called procedure calls another procedure.

Figure 4-2 An Activation Record in the Register Stack



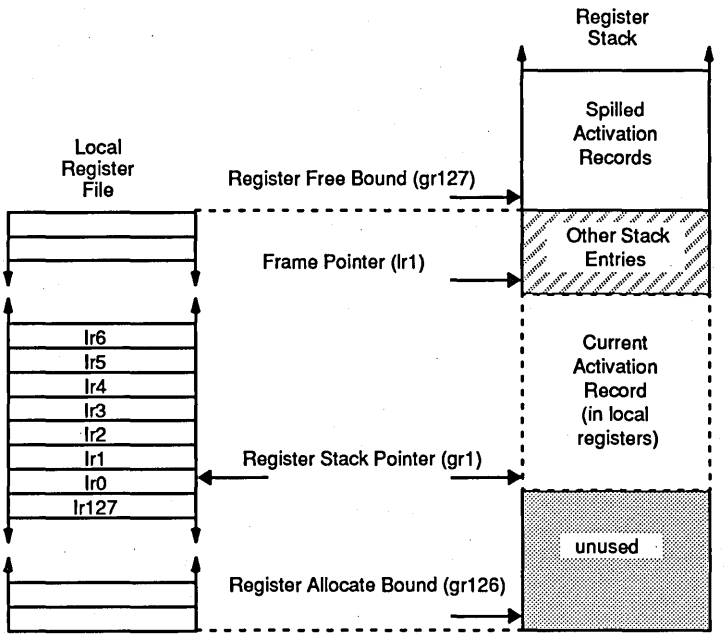
4.1.3 Local Registers As A Stack Cache

The Am29030 and Am29035 microprocessors are designed for efficient implementation of the Register Stack. Specifically, the Am29030 and Am29035 microprocessors can use the large number of relatively addressed local registers to cache portions of the Register Stack, yielding a significant gain in performance. Allocation and de-allocation of activation records occurs largely within the confines of the high-speed local registers, and most procedure calls occur without external references. Furthermore, during procedure execution, most data accesses occur without external references, because activation-record data are referenced most frequently. The principle of locality of reference—which allows any cache to be effective—also applies to the stack cache. The entries in the stack cache are likely to remain there for re-use, because the size of the Register Stack does not change very much over long intervals of program execution. Activation records are typically small, so the 128 locations in the local register file can hold many activation records.

Allocating Register-Stack activation records in the local registers is facilitated by the Stack Pointer in Global Register 1. During the execution of a procedure, the Stack Pointer points simultaneously to the top of the Register Stack in memory and to the local register at the top of the stack cache. In other words, Global Register 1, a word-length register, contains the 32-bit address of the top of the Register Stack, while bits 8–2 of Global Register 1 (with a 1 appended to the most-significant bit) indicate the absolute register number of Local Register 0. Allocation and de-allocation of the Register Stack is accomplished by subtracting from or adding to, respectively, the value of the Stack Pointer.

Using this register-addressing scheme, locations from the Register Stack are automatically mapped into the local register file. Figure 4-3 shows the relationship

Figure 4-3 Relationship of Stack Cache and Register Stack



between the Register Stack and the stack cache in the local registers. As shown, pointers are required to define the boundaries between the Register Stack and the stack cache.

- The register free bound pointer (*rfb*, gr127) defines the boundary between the portion of the Register Stack that is cached in the local registers and the portion that is stored in the external data memory. The *rfb* pointer contains the address of the first word in the Register Stack that is not contained in the local registers, but which is in memory.
- The frame pointer (*fp*, lr1) contains the memory address of the lowest-addressed word not in the current activation record. The current activation record is not necessarily in the data memory: the *fp* is used to determine whether or not an activation record is contained in the local registers when a procedure returns from a call, as described later.
- The register stack pointer (*rsp*, gr1) points to the top of the Register Stack either in the local registers or the data memory; the *rsp* is contained in the local-register Stack Pointer (Global Register 1). The top of the Register Stack may or may not be contained in the data memory—the *rsp* simply defines the location of the top of the Register Stack.
- The register allocate bound pointer (*rab*, gr126) defines the lowest-addressed stack location that can be cached within the local registers. This defines the limit to which local registers can be allocated in the Register Stack.

Several activation records may exist in the Register Stack at any given time, but only one stack location may be mapped to a local register at a given time. When the Register Stack grows beyond the 128-word capacity of the local registers, some movement of data between the stack cache and the Register Stack in data memory must occur.

Stack overflow occurs when a procedure is called, but the activation record of the callee requires more registers than can be allocated in the stack cache (this is detected by comparing *rsp* with *rab*); Figure 4-4 illustrates stack overflow. In this case, the contents of a number of registers must be moved to data memory. The number of registers involved must be sufficient to allow the entire activation record of the callee to reside in the local registers. A block of the registers is copied, or *spilled* into an area of external data memory, freeing space in the local register file for the most recent procedure call.

Stack underflow occurs when a procedure returns to the caller, but the entire activation record of the caller is not resident in the stack cache (this is detected by comparing *fp* with *rfb*); Figure 4-5 illustrates stack underflow. In this case, the non-resident portion of the caller's stack must be moved from data memory to the local registers. Underflow occurs because overflow occurred at some previous point during program execution, causing part of the Register Stack to be moved to data memory.

The processors perform no hardware management of the stack cache and cannot detect a reference to a quantity that is not in the stack cache. Consequently, software must keep the size of an activation record less than or equal to the size of the local register file (128 words). Any additional storage requirements are satisfied by the Memory Stack.

Figure 4-4 Stack Overflow

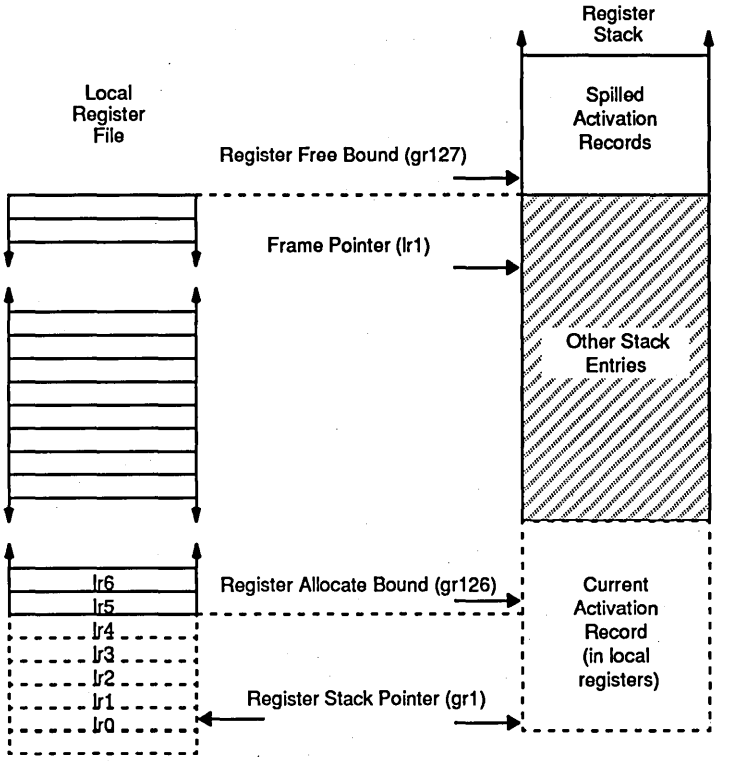
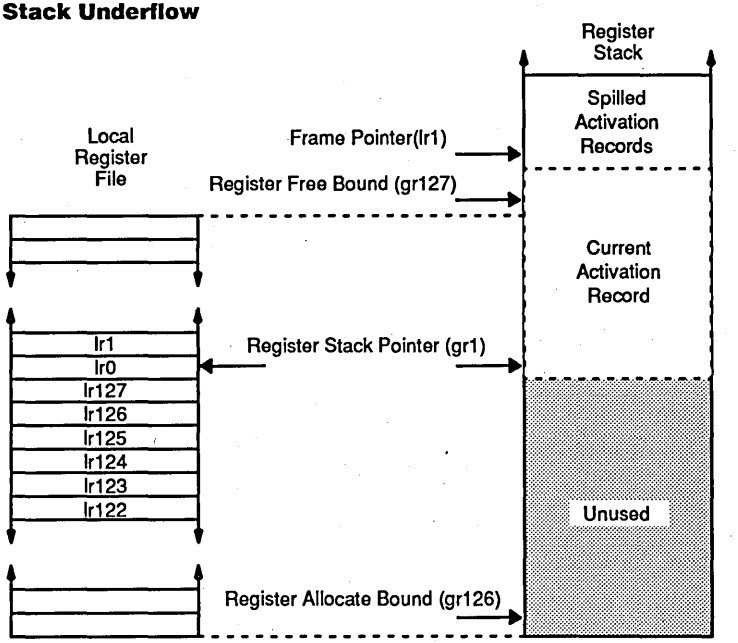


Figure 4-5 Stack Underflow



4.1.4 The Memory Stack

In general, the Memory Stack is used to augment the Register Stack, holding additional information associated with activation records. For example, the Memory Stack holds large data structures than cannot fit into the Register Stack. Similar to the Register Stack, the Memory Stack contains a series of (possibly overlapping) activation records, each corresponding to a procedure activation. However, a Memory Stack activation record need not exist for a procedure that does not need a Memory Stack Area. The Memory Stack contains the following information:

- Overflow incoming arguments. These are incoming arguments that do not fit in the allowed incoming arguments area of the Register Stack activation record.
- Spilled incoming arguments. These are incoming arguments that cannot be kept in the Register Stack. For example, if the address of an argument is used in a called procedure, the associated value must be in the Memory Stack.
- Any procedure-local variable not allocated to a register.
- Local block space. This storage is allocated dynamically on the Memory Stack. It is used to implement functions such as the *alloca()* function in the C programming language.
- Overflow outgoing arguments. These are outgoing arguments that do not fit in the allowed outgoing arguments area of the Register Stack activation record.

In contrast to the Register Stack, the Memory Stack is not cached and has no fixed size limit. The top of the Memory Stack is defined by the memory stack pointer (*msp*), which is stored in Global Register 125 by convention.

4.2 PROCEDURE LINKAGE CONVENTIONS

The procedure linkage conventions define the standard sequences of instructions used to call and return from procedures. These instruction sequences perform the following operations (other, more general operations may also be required, as described later):

- Put procedure arguments into the outgoing arguments area of the activation record. This may or may not involve copying the arguments; copying is not necessary if the arguments are placed into the appropriate registers as the result of computation.
- Branch to the procedure using a call instruction, which also places the return address in a register.
- Allocate a *frame* on the Register Stack. A frame is the storage that contains the procedure's activation record.
- If overflow occurs during frame allocation, spill the least-recently used locations of the Register Stack. The number of spilled locations must be sufficient to allow the new frame to reside entirely within the local registers.
- Determine the frame-pointer value of the called procedure, if this procedure may call another procedure.
- Execute the procedure.
- Place return values into the appropriate registers.
- De-allocate the activation-record frame.
- Fill locations of the local registers from the Register Stack in external memory, if underflow occurs.
- Branch to the procedure's return address.

This section describes the routines that implement the procedure linkage conventions. The operations described here are not required on every procedure call. In some cases, operations can be omitted or simpler routines used; these cases and the accompanying simplifications are also described here.

4.2.1 Argument Passing

The linkage convention allows up to 16 words of arguments to be passed from the caller to the callee in local registers. These arguments are passed in Local Register 2 through Local Register 17 of the caller (note that the local-register numbers are different for the caller and the callee, because of Stack-Pointer addressing).

When more than 16 words are required to pass arguments, the additional words are passed on the Memory Stack. In this case, the memory stack pointer (in Global Register 125) points to the 17th word of the arguments, and the remaining argument words have higher memory addresses. Multi-word arguments may be split across the Register Stack and the Memory Stack. For example, if a multi-word argument starts on the 16th word of the outgoing arguments, the first word of the argument is passed in the Register Stack, and the remainder of the argument is passed in the Memory Stack.

All arguments occupy at least one word; arguments which are a byte or half-word in length (for example, a character) are padded to 32 bits and passed as a full word. However, an array or structure composed of multiple byte or half-word components can be passed as a single, packed array or structure of bytes or half-words rather than an array or structure of padded bytes or half-words.

No argument is aligned to other than a word address boundary, including multi-word arguments. Some multi-word arguments are referenced as a single object (for example, double-precision Floating-Point values). Note that it may be necessary to copy such arguments to an aligned memory or register area before use.

4.2.2 Procedure Prologue

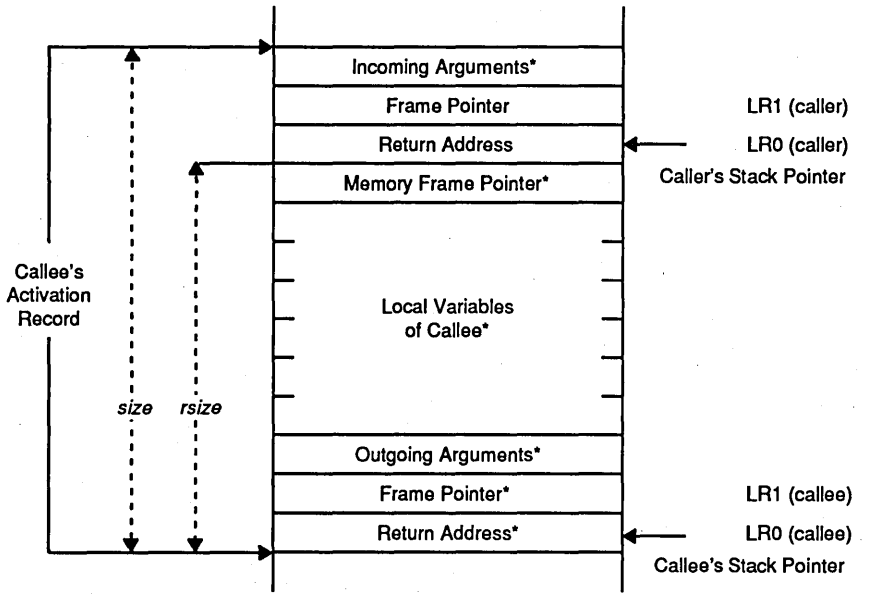
When a procedure is called, and the procedure may call another procedure, the callee must allocate a frame for itself on the Register Stack (this is not required for *leaf* procedures that do not call other procedures, as described later). A frame is allocated by decrementing the register stack pointer to accommodate the size of the required activation record. The procedure *prologue* is the instruction sequence that allocates the callee's Register Stack frame.

To allocate the stack frame, the prologue routine decrements the register stack pointer by the amount *rsize* (see Figure 4-6). The value of *rsize* must be an even number given by the following formula:

$$rsize \geq (\text{size of local variable area}) + (\text{size of outgoing arguments area}) + 2$$

The value 2 in this formula accounts for the space required by the return address (in Local Register 0) and the frame pointer (in Local Register 1). The size of the local variable area includes the space for the memory frame pointer, if required. If the formula total is an odd value, the total must be adjusted (by adding 1) so that the resulting *rsize* value is even. This aligns the top of the Register Stack on a double-word boundary. The reason for this alignment is that double-precision Floating-Point values must be aligned to registers with even absolute-register numbers. Alignment of double-precision values is accomplished by placing these values into even-numbered local registers and making *rsize* even (it is also assumed that the register stack pointer is initialized on an even-word boundary).

Figure 4-6 Definition of *size* and *rsize* Values



Note that *rsize* is not the size of the entire activation record of the callee, because the callee's activation record includes storage that was allocated as part of the caller's activation record frame (e.g., the caller's outgoing arguments area, which is the callee's incoming arguments area). The size of the callee's entire activation record is denoted *size*, and is given by the following formula:

$$size = rsize + (\text{size of the incoming arguments area}) + 2$$

In the prologue routine, the following instruction is used to allocate the stack frame (rsp=gr1):

```
prologue:
        sub     rsp,rsp,rsize*4          ; *4 converts words to bytes
```

However, this instruction does not account for the fact that there may not be enough room in the local registers to contain the activation record. There must be additional instructions to detect stack overflow and to cause spilling if overflow occurs. This is accomplished by comparing the new value of the register stack pointer with the value of the register allocate bound and invoking a trap handler (with vector number V_SPILL) if overflow is detected.

Furthermore, if the procedure calls another procedure, the prologue must compute a frame pointer. The frame pointer will be used by procedures called in turn by the callee to insure that the callee's activation record is in the local registers upon return (i.e., that it has not been spilled onto the Register Stack in data memory). The frame pointer is computed in the prologue because it need only be computed once, regardless of how many procedures are called by a given procedure.

The complete procedure prologue is then (fp = lr1):

```
prologue:
        sub     rsp, rsp, rsize*4       ; allocate frame
        asgeu   V_SPILL, rsp, rab       ; call spill handler if needed
        add     fp, rsp, size*4         ; compute frame pointer
```

4.2.3 Spill Handler

If overflow occurs, the assert instruction in the prologue fails, causing a trap. The trap handler invokes a User-mode routine in the trapping process to spill Register Stack locations from the local registers to external memory. Having most of the spill handling in a User-mode routine minimizes the amount of time that interrupts are disabled and insures that spilling is performed using the correct virtual-memory configuration.

The spill handler uses two registers. The first register, Global Register 121, normally contains a trap-handler argument (*tav*), but is used by the spill handler as a temporary register. The second register, Global Register 122, stores a trap handler return address (*tpc*). This register is used by the User-mode spill handler to return to the trapping procedure. It is assumed that the address of the User-mode spill handler is contained in a global register, denoted user_spill_reg in the following instruction sequence.

The complete spill handler is:

```
Spill:                                  ; operating-system routine
        mfsr    tpc, PC1                ; save return address
        mtsr    PC1, user_spill_reg     ; branch to User spill via interrupt return
        add     tav, user_spill_reg, 4
        mtsr    PC0, tav
        iret

user_spill:                             ; User-mode spill handler
        sub     tav, rab, rsp           ; compute spill: allocate bound – rsp
        srl     tav, tav, 2             ; shift to get number of words
        sub     tav, tav, 1             ; count is one less
        mtsr    CR, tav                 ; set Count Remaining Register
        sub     tav, rab, rsp
        sub     tav, rfb, tav           ; compute new free bound
        add     rab, rsp, 0             ; adjust allocate bound
        storem  0, 0, lr0, tav          ; spill
        jmpi    tpc                     ; return to trapping procedure
        add     rfb, tav, 0             ; adjust free bound
```

4.2.4 Return Values

If the called procedure returns one or more results, the first 16 words of the result(s) are returned in Global Register 96 through Global Register 111, starting with Global Register 96.

If more than 16 words are required for the results, the additional words are returned in memory locations allocated by the caller. In this case, a large return pointer (*lrp*) provided by the caller in Global Register 123 at the time of the call points to the 17th word of the results, and subsequent words are stored at higher memory addresses.

4.2.5 Procedure Epilogue

The procedure epilogue de-allocates the stack frame that was allocated by the procedure prologue and returns to the calling procedure. Stack de-allocation is accomplished by adding the *rsize* value back to the register stack pointer, after which the de-allocated registers are no longer used and are considered invalid. The epilogue also detects stack underflow and causes register filling if underflow occurs. This is accomplished by comparing the value of the caller's frame pointer with the register free bound and invoking a trap handler (with vector number V_FILL) if underflow is detected. Finally, the epilogue returns to the caller using the caller's return address.

The complete procedure epilogue is:

```
epilogue:
        add     rsp, rsp, rsize*4       ; add back rsize count
        nop                             ; cannot reference a local register here
        asleu   V_FILL, fp, rfb         ; call fill handler if needed
        jmpi    lr0                     ; jump to return address
        nop                             ; delay slot
```

4.2.6 Fill Handlers

If underflow occurs, the assert instruction in the epilogue fails, causing a trap. The trap handler invokes a User-mode routine in the trapping process to fill Register Stack locations from the external memory to local registers. The fill handler is similar in organization to the spill handler discussed above.

The complete fill handler is:

```
Fill:                                   ; operating-system routine
        mfsr    tpc, PC1                ; save return address
        mtsr    PC1, user_fill_reg      ; branch to User fill via interrupt return
        add     tav, user_fill_reg, 4
        mtsr    PC0, tav
        iret

user_fill:                              ; User-mode fill handler
        sub     tav, rfb,rab            ; local register has high bit set
        or      tav, tav, rfb           ; put starting register number into Indirect
                                        ; Pointer A
        mtsr    IPA, tav
        sub     tav, fp, rfb            ; compute number of bytes to fill
        add     rab, rab, tav           ; adjust the allocate bound
        srl     tav, tav, 2             ; change byte count to word count
        sub     tav, tav, 1             ; make count zero-based
        mtsr    CR, tav                 ; set Count Remaining register
        loadm   0, 0, gr0, tav          ; fill
        jmpi    tpc                     ; return to trapping procedure
        add     rfb, fp, 0              ; adjust the free bound
```

4.2.7 The Register Stack Leaf Frame

A leaf procedure is one that does not call any other procedure. The incoming arguments of a leaf procedure are already allocated in the calling procedure's activation-record frame, and the leaf routine is not required to allocate locations for any outgoing arguments, frame pointer or return address (since it performs no call). Hence, a leaf procedure need not allocate a stack frame in the local registers, and can avoid the overhead of the procedure prologue and epilogue routines. Instead, a leaf routine can use a set of global registers for local variables; Global Register 96 through Global

Register 124 are reserved for this purpose (among other purposes). If there is an insufficient number of global registers, the leaf procedure may allocate a frame on the Register Stack.

4.2.8 Local Variables And Memory-Stack Frames

A called procedure can store its local variables and temporaries in space allocated in the Register Stack frame by the procedure prologue. The values are referenced as an offset from the *rsp* base address, using the Stack-Pointer addressing of the local registers. No object in a register is aligned on anything smaller than a register boundary, and all objects take at least one register.

Because there are 128 local registers, the total Register Stack activation-record size can not be greater than 128 words. If the callee needs more space for local variables and temporaries, it must allocate a frame on the Memory Stack to hold these objects. To allocate a Memory-Stack frame, the procedure prologue decrements the memory stack pointer (*msp*, in gr125). The procedure epilogue de-allocates the Memory-Stack frame by incrementing the *msp*.

A procedure that extends the Memory Stack dynamically (e.g., using *alloca()*) must make a copy of the *msp* at procedure entry, before allocating the Memory-Stack frame. The *msp* is stored in the memory frame pointer (*mfp*) entry of the activation record in the Register Stack. The procedure then can change the *msp* during execution, according to the needs of dynamic allocation. On procedure return, the Memory-Stack frame is de-allocated using the *mfp* to restore the *msp*. A procedure that does not extend the Memory Stack dynamically need not have an *mfp* entry in its activation record.

The following prologue and epilogue routines are used if there is no dynamic allocation of the Memory Stack during procedure execution, but a Memory Stack frame is otherwise required (Figure 4-6 contains a diagram of register usage):

```
prologue:
        sub     rsp, rsp, <rsize>*4     ; allocate register frame
        asgeu   V_SPILL, rsp, rab       ; call spill handler if needed
        add     fp, rsp, <size>*4       ; compute register frame pointer
        sub     msp, msp, <msize>       ; allocate memory frame
                                        ; msize = size of memory  frame in words
epilogue:
        add     rsp, rsp, <rsize>*4     ; de-allocate register frame
        add     msp, msp, <msize>       ; de-allocate memory frame
        jmpi    lr0                     ; return
        asleu   V_FILL, fp, rfb         ; call fill handler if needed
```

The following prologue and epilogue routines are used if there is dynamic allocation of the Memory Stack during procedure execution:

```
prologue:
        sub     rsp, rsp, <rsize>*4     ; allocate register frame
        asgeu   V_SPILL, rsp, rab       ; call spill handler if needed
        add     fp, rsp, <size>*4       ; compute register frame pointer
        add     lr{<rsize>-1}, msp, 0   ; save memory frame pointer
                                        ; lr{rsize-1} is last reg in new frame
        sub     msp, msp, <msize>       ; allocate memory frame,
                                        ; msize = size of memory frame in words
```

```
epilogue:
        add     msp, lr{<rsize>-1},0    ; restore memory stack pointer
                                        ; de-allocate memory frame
        add     rsp, rsp, <rsize>*4     ; de-allocate register frame
        nop                             ; cannot reference a local register here
        jmpi    lr0                     ; return
        asleu   V_FILL, fp, rfb         ; call fill handler if needed
```

4.2.9 Static Link Pointer

Some programming languages (notably Pascal) permit nested procedure declarations, introducing the possibility that a procedure may reference variables and arguments which are defined and managed by another procedure. This other procedure is a *static parent* of the callee. A static parent is determined by the declarations of procedures in the program source, and is not necessarily the calling procedure; the calling procedure is the *dynamic parent*. Since procedures can be nested at a number of levels, a given procedure may have a number of hierarchically organized static parents.

A called procedure can locate its dynamic parent and the variables of the dynamic parent because of the return address and frame pointer in the Register Stack. However, these are not adequate to locate variables of the static parent which may be referenced in the procedure. If such references appear in a procedure, the procedure must be provided with a static link pointer (*slp*). In the run-time organization, the *slp* is stored in Global Register 124. Since there can be a hierarchy of static parents, the *slp* points to the *slp* of the immediate parent, which in turn points to the *slp* of its immediate parent, and so on. Note that the contents of Global Register 124 may be destroyed by a procedure call, so a procedure needing to reference the variables of a static parent may need to preserve the *slp* until these references are no longer necessary.

4.2.10 Transparent Procedures

A transparent procedure is one that requires very little overhead for managing run-time storage. Transparent procedures are used primarily to implement compiler-specific support functions, such as integer divide.

A transparent routine does not allocate any activation-record frames. Parameters are passed to a transparent procedure using *tav* and the Indirect Pointer A, B, and C registers. The return address is stored in *tpc*. This convention allows a leaf procedure to call a transparent procedure without changing its status as a leaf procedure. There is a tight relationship between a compiler and the transparent procedures it calls. Some transparent procedures may need more temporary registers and the compiler must account for this.

4.3 REGISTER USAGE CONVENTION

The run-time organization standardizes the uses of the local and global registers. This section summarizes register use and the nomenclature for register values:

- GR1: Register stack pointer (*rsp*).
- GR2–GR63: Unimplemented.
- GR64–GR95: Reserved for operating-system use.

- GR96–GR111: Procedure return values. Lower-numbered registers are used before higher-numbered registers. If more than 16 words are needed, the additional words are stored in the Memory Stack (see GR123, large return pointer). These registers are also used for temporary values that are destroyed upon a procedure call.
- GR112–GR115: Reserved for programmer. These registers are not used by the compiler, except as directed by the programmer.
- GR116–GR120: Compiler temporaries.
- GR121: Trap handler argument/temporary (*tav*)—This register is used to communicate arguments to a software-invoked trap routine. It can be destroyed by the trap, but not by other traps and interrupts not explicitly generated by the program (for example, a Timer trap).
- GR122: Trap handler return address/temporary (*tpc*). This register also is used by software-invoked traps. It can be destroyed by the trap, but not by other traps and interrupts not explicitly generated by the program (for example, a Timer trap).
- GR123: Large return pointer/temporary (*lrp*).
- GR124: Static link pointer/temporary (*slp*).
- GR125: Memory stack pointer (*msp*).
- GR126: Register allocate bound (*rab*).
- GR127: Register free bound (*rfb*).
- LR0: Return address.
- LR1: Frame pointer.

In this convention, registers must be handled by software according to system requirements. The following practices are recommended:

- GR64–GR95 should be protected from User-mode access by the Register Bank Protect Register.
- The contents of GR96-GR124 should be assumed destroyed by a procedure call, unless the procedure is a transparent procedure.
- The contents of GR121 and GR122 should be assumed destroyed by any procedure call or any program-generated trap.
- The contents of GR125 are always preserved by a procedure call.
- The contents of GR126 and GR127 are managed by the spill and fill handlers and should not be modified except by these handlers.

4.4 EXAMPLE OF A COMPLEX PROCEDURE CALL

The following code sequence demonstrates a complex procedure call, illustrating how registers are used in the run-time organization:

caller:

```
(other code)
        add     lrp, msp, 32            ; pass lrp
        add     slp, msp, 120           ; pass a static link
        call    lr0, callee
        const   lr2, 1                  ; 1 as first argument
(other code)
```

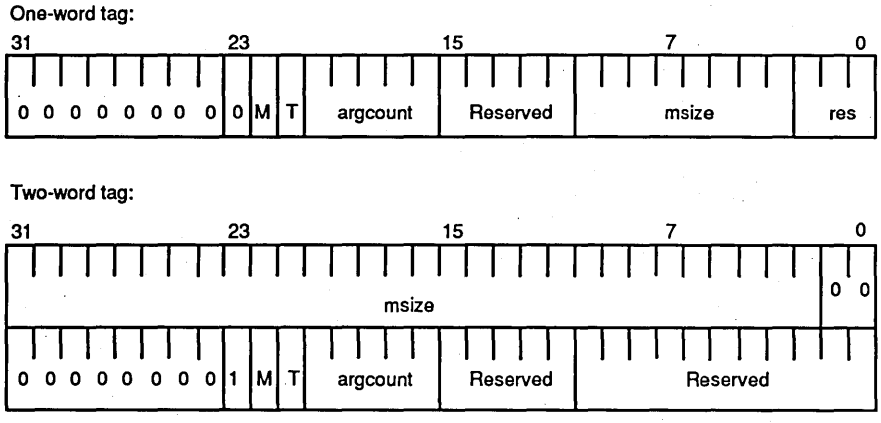
```
callee:
        const   tav, (126–2)*4          ; giant register allocation
        sub     rsp, rsp, tav           ; allocate register frame
        asgeu   V_SPILL, rsp, rab
        const   tav, (126–2)*4+(3*4)    ; incoming arguments and overhead
        add     fp, rsp, tav            ; create frame pointer
        add     lr123, msp, 0           ; for dynamic Memory-Stack allocation
        const   tav, memory_frame_size  ; big msize
        consth  tav, memory_frame_size  ; high half of msize
        sub     msp, msp, tav           ; allocate memory frame
        add     lr18, lrp, 0            ; save lrp for later
        add     lr19, slp, 0            ; save slp for later

(other code)
        add     msp, lr123, 0           ; de-allocate memory frame
        const   tav, (126–2)*4          ; giant allocation size
        add     rsp, rsp, tav           ; de-allocate register frame
        const   gr96, 1                 ; return value
        jmpi    lr0                     ; return to caller
        asleu   V_FILL, fp, rfb         ; insure caller's registers in frame
```

4.5 TRACE-BACK TAGS

A *trace-back tag* is either one or two words of information included at the beginning of every procedure. This information permits a debug routine to determine the sequence of procedure calls and the values of program variables at a given point in execution. The trace-back tag describes the memory frame size and the number of local registers used by the associated procedure. A one-word tag is used if the memory frame size is less than 2K words; otherwise, the two-word tag is used. Regardless of tag length, the tag directly precedes the first instruction of the procedure. Figure 4-7 shows the format of the trace-back tags.

The first word of a trace-back tag starts with the invalid operation code 00 (hexadecimal). This unique, invalid instruction operation code allows the debugger to locate the beginning of the procedure in the absence of other information related to the beginning of the procedure, such as from a symbol table. This is particularly useful after a

Figure 4-7 Trace-Back Tags



program crash, in which case the debug routine may have only an arbitrary instruction address within a procedure. The call sequence up to the current point in execution can be determined from the *rsize* and *msize* values in the trace-back tag. However, for procedures that perform dynamic stack allocation (e.g., using *alloca()*), the memory frame pointer must be used.

The tag word immediately preceding a procedure contains the following fields. Reserved fields must be zero.

Bits	Item	Description
31–24	opcode	Hexadecimal 00 (an invalid opcode)
23	tag type	0/one-word tag; 1/two-word tag
22	m	0/no *mfp*; 1/*mfp* used
21	t	0/normal; 1/transparent procedure
20–16	argcount	Number of arguments in registers (includes lr0 and lr1)
15–11	Reserved	Reserved, must be zero
10–3	msize	Memory frame size in doublewords (if bit 23 is 0) or reserved (if bit 23 is 1)
2–0	Reserved	Reserved, must be zero

If the procedure uses a Memory-Stack frame size 2K words or more, the msize field is contained in the second tag word immediately preceding the first tag word.

CHAPTER 5

PIPELINING AND INSTRUCTION SCHEDULING

This chapter describes the operation of the Am29030 and Am29035 microprocessor pipelines. A description of the Am29030 and Am29035 microprocessor pipelines is presented only to offer the reader a general overview of the internal operation of this pipeline, with the intent to aid understanding of the effects that the pipeline has on program execution and of the behavior of the microprocessors under certain conditions, especially the behavior of the system interfaces described in Chapter 10.

The operation of the functional units is coordinated by Pipeline Hold mode, which insures that operations are performed in the proper order. This chapter also describes the Pipeline Hold mode. In certain cases, the pipeline is exposed during instruction execution, in that the execution of certain instructions is dependent on the execution of previous instructions. This chapter discusses the cases where the pipeline is exposed to software and describes the resulting effect on instruction execution.

5.1 FOUR-STAGE PIPELINE

The Am29030 and Am29035 microprocessors implement a four-stage pipeline for instruction execution, as shown in detail in Figure 5-1. The four stages are fetch, decode, execute, and write-back. For operations, the pipeline is organized so that the effective instruction-execution rate may be as high as one instruction per cycle.

During the fetch stage, the Instruction Fetch Unit determines the location of the next processor instruction and issues the instruction to the decode stage. The instruction is fetched either from the Instruction Prefetch Buffer, the Instruction Cache, or an external instruction memory.

During the decode stage, the instruction issued from the fetch stage is decoded, and the required operands are fetched and/or assembled. Addresses for branches, loads, and stores are also evaluated.

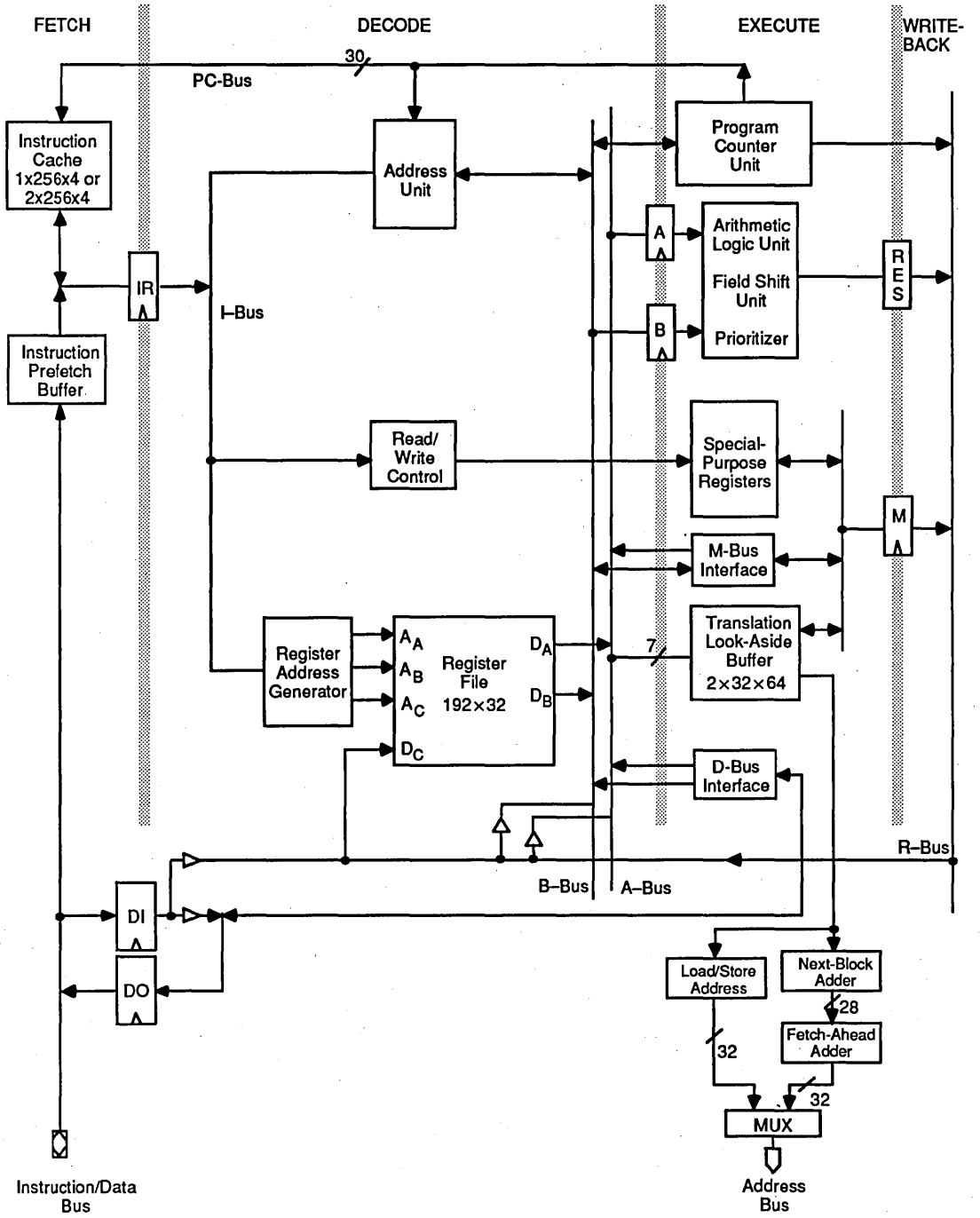
During the execute stage, the Execution Unit performs the operation specified by the instruction. In the case of branches, loads, and stores, the Memory Management Unit (see Chapter 7) performs address translation if required.

During the write-back stage, the results of the operation performed during the execute stage are stored. In the case of branches, loads, and stores, the physical address resulting from translation during the execute stage is transmitted to an external device or memory.

Most pipeline dependencies that are internal to the processor are handled by forwarding logic in the processor. For those dependencies that result from the external system, the Pipeline Hold mode insures proper operation.

In a few special cases, the processor pipeline is exposed to software executing on the Am29030 and Am29035 microprocessors (see Sections 5.4, 5.5, and 5.6).

Figure 5-1 Am29030 and Am29035 Microprocessors Data Flow



5.2 PIPELINE HOLD MODE

The Pipeline Hold mode is activated whenever sequential processor operation cannot be guaranteed. When this mode is active, the pipeline stages do not advance, and most internal processor state is not modified. The processor places itself in the Pipeline Hold mode in the following situations:

1. The processor requires an instruction that has either not been fetched or not been returned by the external instruction memory.
2. The processor requires data from an in-progress load and the operation has not completed.
3. The processor attempts to execute a load or store instruction while another load or store is in progress.
4. The processor is reading or writing the Cache Interface Register. During this operation, the bus that couples the Instruction Prefetch Buffer and the Instruction Cache is used to move data to or from the Instruction Cache. The processor exits the Pipeline Hold mode in the next processor cycle, unless one of the other conditions listed causes a further pipeline hold to occur.
5. The processor must perform a serialization operation as described in Section 5.3.
6. The processor is performing a sequence of load-multiple or store-multiple accesses. The Pipeline Hold mode in this case prevents further instruction execution until the completion of the load-multiple or store-multiple sequence.
7. The processor has taken an interrupt or trap, and the first instruction of the interrupt or trap handler has not entered the execute stage. The Pipeline Hold mode in this case prevents the processor pipeline from advancing until the interrupt or trap handler can begin execution.
8. The processor has executed an interrupt return, and the target instruction of the interrupt return has not entered the execute stage. The Pipeline Hold mode in this case prevents the processor pipeline from advancing until the interrupt return sequence is complete.

The Pipeline Hold mode is exited whenever the causing conditions no longer exist, or when the $\overline{\text{WARN}}$ or $\overline{\text{RESET}}$ input is asserted.

5.3 SERIALIZATION

The Am29030 and Am29035 microprocessors overlap external data references with other operations. When an external data reference might have to be restarted, however, the processor context must be the same as when the operation was first attempted. To insure this, certain operations are serialized.

The processor serializes by entering the Pipeline Hold mode in any of the following circumstances:

1. An external access is not yet completed, and one of the following instructions is encountered:
   Move to Special Register
   Move to Special Register Immediate
   Move to TLB
   Interrupt Return
   Interrupt Return and Invalidate
   Halt

2. An external access is not yet completed, and an interrupt or trap, other than a $\overline{\text{WARN}}$ trap, is taken.

If the processor is in the Pipeline Hold mode due to serialization, it enters the Executing mode once the external access is completed. Note that the processor may immediately take a Data Access Exception trap.

5.4 DELAYED BRANCH

The effect of jump and call instructions is delayed by one cycle to allow the processor pipeline to achieve maximum throughput. When one of these branches is successful, the instruction immediately following the jump or call is executed before the target instruction of the jump or call is executed. Jump and call instructions collectively are referred to as delayed branches, and the instruction immediately following is called the delay instruction (sometimes referred to as a delay slot).

For example, in the following code fragment:

```
        .
        .
        cpeq    gr96, lr6, lr7    (1)
        jmpf    gr96, label       (2)
        sub     lr6, lr6, 1       (3)
        const   lr6, 0            (4)
        .
        .
label:  call    lr0, sort         (5)
        add     lr2, lr5, 0       (6)
        cpneq   lr3, gr96, 0      (7)
        .
        .
```

The sub instruction (3) is executed regardless of the outcome of the jmpf instruction (2). Of course, if the jmpf is not successful, the const instruction (4) is also executed. If the jmpf is successful, then the instruction sequence is: (3), (5), (6), and then the first instruction of the sort procedure. Note that the call instruction (5) is also a delayed branch, so the instruction immediately following it, (6), is always executed. After the sort procedure executes the return sequence, the cpneq instruction (7) is the next instruction executed.

The benefit of delayed branches is improved performance and a simplified processor implementation. Performance is improved because the processor pipeline executes useful instructions in a larger number of cycles, compared to an implementation without delayed branches.

For example, ignoring all other effects on performance, and assuming that 15% of all instructions are taken branches, then a processor without delayed branches would take at least two cycles for 15% of its instructions, leading to $0.85(1) + 0.15(2) = 1.15$ cycles per instruction, on average. This represents a 15% performance degradation compared to a processor with delayed branches (assuming, for this simple example, that the delay instruction is always useful).

The cost of having delayed branches is either the extra effort required when the compiler takes advantage of delayed branches (by re-organizing code), or the extra NO-OP instruction which the compiler inserts after every branch to guarantee correct program operation. Since the compiler expends only a small amount of effort to avoid wasting time and space with NO-OPs, and since the performance improvement resulting from this effort is significant, delayed branches are beneficial overall.

When two immediately adjacent branches are taken, the target of the first branch pre-empts execution of the delay cycle of the second branch, and the target of the second branch then follows the target of the first branch. For example, in the following code fragment:

```
        .
        .
        jmp l1                          (1)
        jmp l2                          (2)
        add     lr4, lr4, lr5           (3)
        .
        .
L1:     sub     gr96, gr96, 1           (4)
        subc    gr97, gr97, 0           (5)
        .
        .
L2:     const   gr100, 0xff0f           (6)
        subr    gr101, gr101, 1         (7)
        or      gr100, gr100, gr101     (8)
        .
        .
```

An unconditional jmp instruction (1) is followed immediately by another unconditional jmp instruction (2). (In this example, unconditional jmps are used; however, any two immediately adjacent taken branches exhibit the same behavior.) The sequence of executed instructions in this case is: jmp instruction (1), jmp instruction (2), sub instruction (4), const instruction (6), subr instruction (7), or instruction (8), and so on. Note that the add instruction (3) is not executed. Also, the target of the first jmp instruction (1) was merely visited; control did not continue sequentially from L1 but rather continued from L2.

5.5 OVERLAPPED LOADS AND STORES

The Am29030 and Am29035 microprocessors overlap external data references with other operations. Certain programming practices are necessary to exploit this parallelism to improve program performance.

In order to make full use of overlapped storage accesses, some instruction reorganization may be necessary. For example, in the following sequence:

```
loop:   .
        .
        sll     gr121, gr119, 2         (1)
        add     gr121, gr120, gr121     (2)
        load    0, 0, gr121, gr121      (3)
        add     gr96, gr96, gr121       (4)
        sub     gr96, gr96, 3           (5)
        add     gr119, gr119, 1         (6)
        cplt    gr122, gr119, lr2       (7)
        jmpt    gr122, loop             (8)
        nop                             (9)
        .
        .
```

the add instruction (4) uses the result of the load instruction (3). However, the following four instructions do not depend on the result of the load. Therefore, the add instruction (4) can be moved past the jmpt (8)—since it always will be executed even if

the jmpt is taken—and can replace the NO-OP instruction (9). The resulting sequence is:

```
loop:   .
        .
        sll     gr121, gr119, 2         (1)
        add     gr121, gr120, gr121     (2)
        load    0, 0, gr121, gr121      (3)
        sub     gr96, gr96, 3           (4)
        add     gr119, gr119, 1         (5)
        cplt    gr122, gr119, lr2       (6)
        jmpt    gr122, loop             (7)
        add     gr96, gr96, gr121       (8)
        .
        .
```

The instructions (4) through (7) are likely to be executed while external memory satisfies the load request, resulting in improved throughput. The processor thus allows parallelism to be exploited by instruction reordering.

The overlapped load feature may be used to improve processor performance, but imposes no constraints on instruction sequences, as delayed branches do. The processor implements the proper pipeline interlocks to make this parallelism transparent to a running program.

5.6 DELAYED EFFECTS OF REGISTERS

The modification of some registers has a delayed effect on processor behavior, because of the processor pipeline. The affected registers are the Stack Pointer (Global Register 1), Indirect Pointers A, B, and C, the MMU Configuration Register, and the Current Processor Status Register.

An instruction that writes to the Stack Pointer can be followed immediately by an instruction that reads the Stack Pointer. However, any instruction that references a local register also uses the value of the Stack Pointer to calculate an absolute-register number. At least one cycle of delay must separate an instruction that updates the Stack Pointer and an instruction that references a local register. In most systems, this affects procedure call and return only (see Section 4.2). In general, though, an instruction that immediately follows a change to the Stack Pointer should not reference a local register (however, note that this restriction does not apply to a reference of a local register via an indirect pointer).

The indirect pointers have an implementation similar to the Stack Pointer, and exhibit similar behavior. At least one cycle of delay must separate an instruction that modifies an indirect pointer and an instruction that uses that indirect pointer to access a register.

Note that it normally is not possible to guarantee that the delayed effect of the Stack Pointer and indirect pointers is visible to a program. If an interrupt or trap is taken immediately after one of these registers is set, then the interrupted routine sees the effect of the setting in the following instruction, because many cycles elapse between the two instructions. For this reason, a program should not be written in a manner that relies on the delayed effect; the results of this practice may be unpredictable.

At least one cycle of delay must separate a Move To Special Register that modifies the Page Size (PS) field of the MMU Configuration Register and an instruction that performs address translation. The latter instruction includes successful branches, loads, and stores.

If the Freeze (FZ) bit of the Current Processor Status Register is reset from 1 to 0, two cycles are required before all program state is reflected properly in the registers affected by the FZ bit. This implies that interrupts and traps cannot be enabled until two cycles after the FZ bit is reset, for proper sequencing of program state.

An access to the Cache Data Register (CDR) cannot immediately follow a write to the Cache Interface Register (CIR). At least one instruction must separate the access of the CDR from the write to the CIR.

CHAPTER 6

SYSTEM PROTECTION

The Am29030 and Am29035 microprocessors provide protection for system resources, including general-purpose registers, special-purpose registers, Translation Look-Aside Buffer registers, and external locations. Certain processor operations are also protected. This chapter describes the processor's protection mechanisms.

6.1 USER AND SUPERVISOR MODES

At any given time, the Am29030 and Am29035 microprocessors operate in one of two mutually exclusive program modes: the Supervisor mode or the User mode. All system-protection features of the Am29030 and Am29035 microprocessors are based on the difference between these two modes.

6.1.1 Supervisor Mode

The processor operates in the Supervisor mode whenever the Supervisor Mode (SM) bit of the Current Processor Status Register is 1 (see Section 8.1.1). In the Supervisor mode, executing programs have access to all processor resources. Virtual pages mapped by the Memory Management Unit (MMU), however, are protected from Supervisor access (read, write, or execute) when the appropriate bit (SR, SW, or SE, respectively) in the corresponding Translation Look-Aside Buffer (TLB) Entry is 0 (see Chapter 7).

Any attempt to access a special-purpose register in the range of 160 to 255 causes a Protection Violation to occur, in either Supervisor or User mode. This permits virtualization of these registers. Supervisor-mode accesses are permitted for any general-purpose register, regardless of protection.

The attempted execution of a translated load or store for which the AS bit is 1 causes a Protection Violation trap, in either Supervisor or User mode.

During the address cycle of a bus request, the Supervisor mode is indicated by the SUP/$\overline{\text{US}}$ output being High.

6.1.2 User Mode

The processor operates in the User mode whenever the SM bit in the Current Processor Status Register is 0. In the User mode, any of the following actions by an executing program causes a Protection Violation trap to occur:

1. An attempted access of any TLB register.
2. An attempted access of any general-purpose register for which a bit in the Register Bank Protect Register is 1 (see Section 6.2).
3. An attempted execution of a load or store instruction for which the PA bit is 1, for which the AS bit is 1, or for which the UA bit is 1 (see Section 3.3).
4. An attempted execution of one of the following instructions: Interrupt Return, Interrupt Return and Invalidate, Invalidate, or Halt. However, a hardware-

development system can disable protection checking for the Halt instruction, so that this instruction may be used to implement instruction breakpoints in User-mode programs (see Sections 11.2 and 11.4).

5. An attempted access of special-purpose register in the range of 0 to 127 or 160 to 255.
6. An attempted execution of an assert or Emulate instruction which specifies a vector number between 0 and 63, inclusive (see Chapter 8).
7. An attempted access (read, write, or execute) in a virtual page mapped by the Memory Management Unit, when the appropriate permission bit (UR, UW, or UE, respectively) in the corresponding TLB Entry is 0.

Devices and memories on the bus can also implement protection and generate traps based on the value of the SM bit. During the address cycle of a bus request, the User mode is indicated by the SUP/$\overline{US}$ output being Low.

6.2 REGISTER PROTECTION

General-purpose registers are divided into register banks and are protected by the Register Bank Protection Register. The Register Bank Protection Register allows parameters for the operating system to be kept in general-purpose registers and protected from corruption by User-mode programs. Register banks consist of 16 registers (except for Bank 0, which contains Registers 2 through 15) and are partitioned according to absolute-register numbers, as shown in Figure 6-1.

Figure 6-1 Register Bank Organization

Register Bank Protect Register Bit	Absolute-Register Numbers	General-Purpose Registers
0	2 through 15	Bank 0 (not implemented)
1	16 through 31	Bank 1 (not implemented)
2	32 through 47	Bank 2 (not implemented)
3	48 through 63	Bank 3 (not implemented)
4	64 through 79	Bank 4
5	80 through 95	Bank 5
6	96 through 111	Bank 6
7	112 through 127	Bank 7
8	128 through 143	Bank 8
9	144 through 159	Bank 9
10	160 through 175	Bank 10
11	176 through 191	Bank 11
12	192 through 207	Bank 12
13	208 through 223	Bank 13
14	224 through 239	Bank 14
15	240 through 255	Bank 15

The Register Bank Protect Register contains 16 protection bits, where each bit controls User-mode accesses (read or write) to a bank of registers. Bits 0–15 of the Register Bank Protect Register, protect Register Banks 0 through 15, respectively.

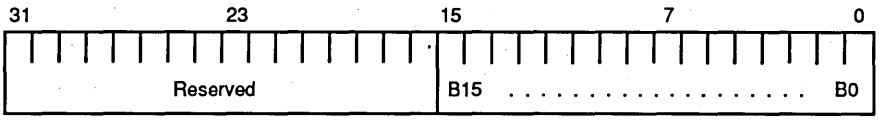
When a bit in the Register Bank Protect Register is 1, and a register in the corresponding bank is specified as an operand register or result register by a User-mode instruction, a Protection Violation trap occurs. Note that protection is based on absolute-register numbers; in the case of local registers, Stack-Pointer addition is performed before protection checking.

When the processor is in the Supervisor mode, the Register Bank Protect Register has no effect on general-purpose register accesses.

6.2.1 Register Bank Protect (RBP, Register 7)

This protected special-purpose register (Figure 6-2) protects banks of general-purpose registers from User-mode program accesses.

Figure 6-2 Register Bank Protect Register



The general-purpose registers are partitioned into 16 banks of 16 registers each (except that Bank 0 contains 14 registers). The banks are organized as shown in Figure 6-1.

Bits 31–16: Reserved.

Bits 15–0: Bank 15 through Bank 0 Protection Bits (B15–B0)—In the Register Bank Protect Register, each bit is associated with a particular bank of registers, and the bit number gives the associated bank number (e.g., B11 determines the protection for Bank 11).

6.3 MEMORY PROTECTION

Memory and input/output access protection is provided by the MMU. Each TLB entry in the MMU contains protection bits which determine whether or not an access is permitted to the page associated with the entry.

There is a set of protection bits for Supervisor-mode programs and a separate set for User-mode programs. Thus, for the same virtual page, the access authority of programs executing in the Supervisor mode can be different than the authority of programs executing in the User mode.

If address translation is performed successfully as described in Section 7.4.2, the relevant TLB entry is used to perform protection checking for the access. Six bits are provided for this purpose: Supervisor Read (SR), Supervisor Write (SW), Supervisor Execute (SE), User Read (UR), User Write (UW), and User Execute (UE). These bits restrict accesses, depending on the program mode of the access, as shown in Table 6-1 (the value *x* is a *don't care*).

Note that for the Load and Set (LOADSET) instruction, the protection bits must be set to allow both the load and store access. If this condition does not hold, neither access is performed.

Table 6-1 Access Protection

SR	SW	SE	UR	UW	UE	Type of Access Allowed
x	x	x	0	0	0	No User access
x	x	x	0	0	1	User instruction
x	x	x	0	1	0	User store
x	x	x	0	1	1	User store or instruction
x	x	x	1	0	0	User load
x	x	x	1	0	1	User load or instruction
x	x	x	1	1	0	User load or store
x	x	x	1	1	1	Any User access
0	0	0	x	x	x	No Supervisor access
0	0	1	x	x	x	Supervisor instruction
0	1	0	x	x	x	Supervisor store
0	1	1	x	x	x	Supervisor store or instruction
1	0	0	x	x	x	Supervisor load
1	0	1	x	x	x	Supervisor load or instruction
1	1	0	x	x	x	Supervisor load or store
1	1	1	x	x	x	Any Supervisor access

If protection checking indicates that a given access is not allowed, a Data MMU Protection Violation or Instruction MMU Protection Violation trap occurs. The cause of the trap can be determined by inspecting the Program Counter 1 Register for an Instruction MMU Protection Violation, or by inspecting the contents of the Channel Address and Channel Control registers for a Data MMU Protection Violation.

6.4 EXTERNAL ACCESS PROTECTION

Other than the protection offered by the Memory Management Unit, the processor provides no specific protection for external devices and memories. However, the SUP/$\overline{US}$ output reflects the value of the SM bit during the address cycle of an external access. This can signal external devices and memories to provide protection. Any protection violations can be reported via the $\overline{ERR}$ input.

CHAPTER 7

MEMORY MANAGEMENT

The Am29030 and Am29035 microprocessors incorporate a Memory Management Unit (MMU) for performing virtual-to-physical address translation and memory access protection. This chapter describes the logical operation of the MMU. Address translation is performed by the Translation Look-Aside Buffer (TLB), which is the fundamental component of the MMU. This chapter describes the structure of the TLB and the issues related to software management of the TLB.

7.1 TRANSLATION LOOK-ASIDE BUFFER

The MMU stores the most-recently performed address translations in a special cache, the Translation Look-Aside Buffer (TLB). The TLB reflects information in the system page tables, except that it specifies the translation for many fewer pages; this restriction allows the TLB to be incorporated on the processor chip where the performance of address translation is maximized.

A diagram of the TLB is shown in Figure 7-1. The TLB is a table of 64 entries, divided into two equal columns, called Column 0 and Column 1. Within each column, entries are numbered 0 to 31. Entries in different columns which have equivalent entry-numbers are grouped into a unit called a set; there are thus 32 sets in the TLB, numbered 0 to 31.

Each TLB entry is 64 bits long, and contains mapping and protection information for a single virtual page. TLB entries may be inspected and modified by processor instructions executed in the Supervisor mode. The layout of TLB entries is described in Section 7.2.

The TLB stores information about the ownership of the TLB entries in an 8-bit Task Identifier (TID) field in each entry. This makes it possible for the TLB to be shared by several independent processes without the need for invalidation of the entire TLB as processes are activated. It also increases system performance by permitting processes to warm-start (i.e., to start execution on the processor with a certain number of TLB entries remaining in the TLB from a previous execution).

Each TLB entry contains a Usage bit to assist management of the TLB entries. The Usage bit indicates which block of the entry within a given set was least recently used to perform an address translation. Usage bits for two entries in the same set are equivalent.

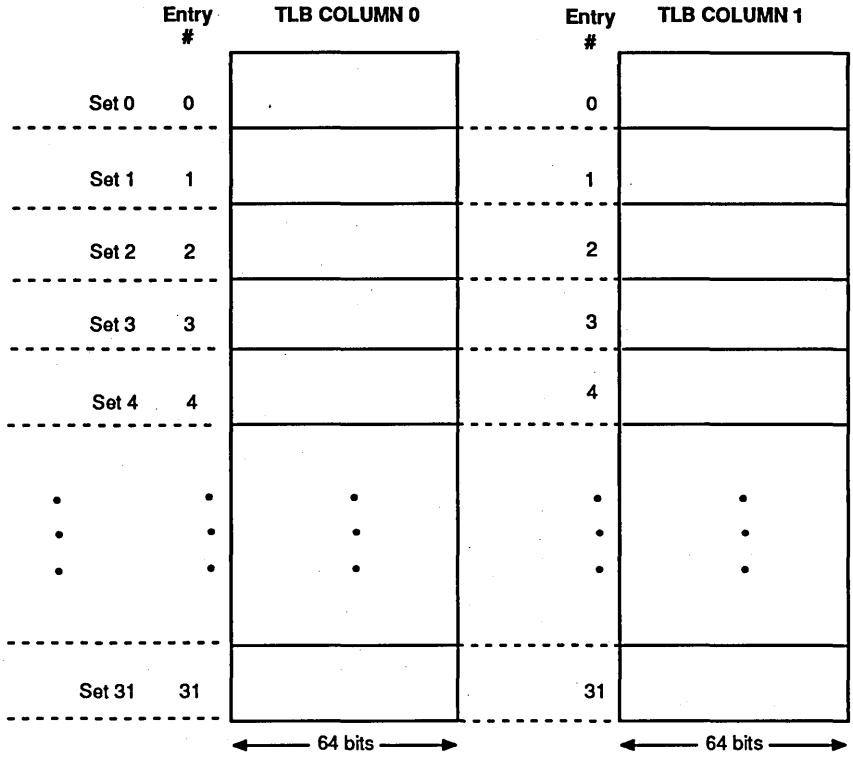
The TLB contains other fields which are described in the following sections.

7.2 TLB REGISTERS

The Am29030 and Am29035 microprocessors contain 128 Translation Look-Aside Buffer (TLB) registers. The organization of the TLB registers is shown in Figure 7-2.

The TLB registers comprise the TLB entries and are provided so that programs may inspect and alter TLB entries. This allows the loading, invalidation, saving, and restoring of TLB entries.

Figure 7-1 Translation Look-Aside Buffer Organization



TLB registers contain fields that are reserved for future processor implementations. When a TLB register is read, a bit in a reserved field is read as a 0. An attempt to write a reserved bit with a 1 has no effect; however, this should be avoided because of upward-compatibility considerations.

The Translation Look-aside Buffer (TLB) registers are accessed only by explicit data movement by Supervisor-mode programs. Instructions that move data to or from a TLB register specify a general-purpose register containing a TLB register number. The TLB register number is given by the contents of bits 6–0 of the general-purpose register. TLB register numbers may be specified only indirectly by general-purpose registers.

TLB entries are accessed as registers numbered 0–127. Since two words are required to completely specify a TLB entry, two registers are required for each TLB entry. The words corresponding to an entry are paired as two sequentially numbered registers starting on an even-numbered register. The word with the even register number is called Word 0, and the word with the odd register number is called Word 1. The entries for TLB Column 0 are in registers numbered 0–63, and the entries for TLB Column 1 are in registers numbered 64–127.

7.2.1 TLB Entry Word 0

The TLB Entry Word 0 register is shown in Figure 7-3.

Figure 7-2 Translation Look-Aside Buffer Registers

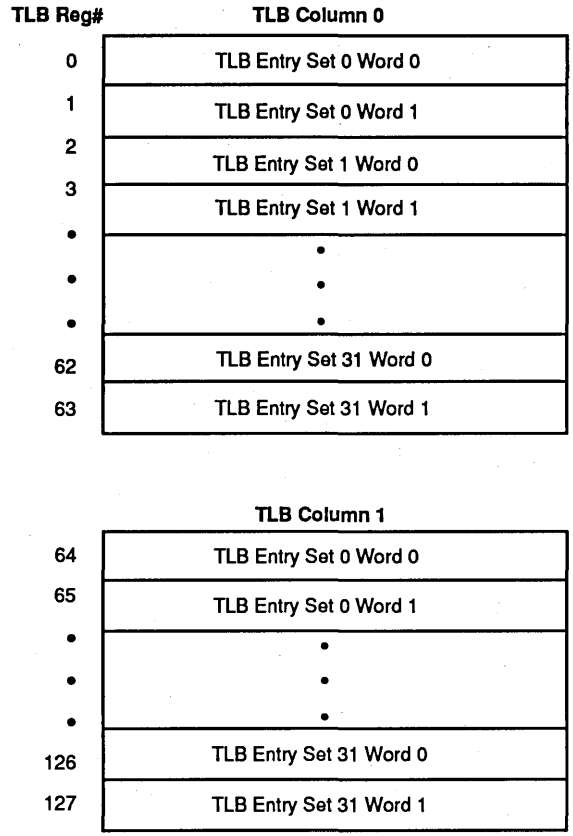
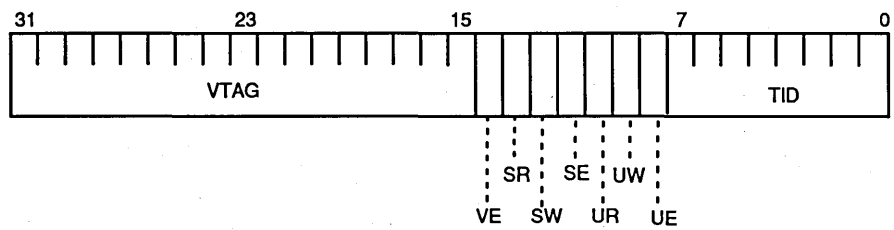


Figure 7-3 TLB Entry Word 0 Register



Bits 31–15: Virtual Tag (VTAG)—When the TLB is searched for an address translation, the VTAG field of the TLB entry must match the most-significant 17, 16, 15, or 14 bits of the address being translated—for page sizes of 1, 2, 4, and 8K bytes, respectively—for the search to be successful.

When software loads a TLB entry with an address translation, the most-significant 14 bits of the Virtual Tag are set with the most-significant 14 bits of the virtual address whose translation is being loaded into the TLB. The remaining three bits of the Virtual

Tag must be set either to the corresponding bits of the address or to zeros depending on the page size, as follows (*A* refers to corresponding address bits):

Page Size	VTAG 2–0 (TLB Word 0 bits 17–15)
1K bytes	AAA
2K bytes	AA0
4K bytes	A00
8K bytes	000

Bit 14: Valid Entry (VE)—If this bit is 1, the associated TLB entry is valid; if it is 0, the entry is invalid.

Bit 13: Supervisor Read (SR)—If the SR bit is 1, Supervisor-mode load operations from the virtual page are allowed; if it is 0, Supervisor-mode loads are not allowed.

Bit 12: Supervisor Write (SW)—If the SW bit is 1, Supervisor-mode store operations to the virtual page are allowed; if it is 0, Supervisor-mode stores are not allowed.

Bit 11: Supervisor Execute (SE)—If the SE bit is 1, Supervisor-mode instruction accesses to the virtual page are allowed; if it is 0, Supervisor-mode instruction accesses are not allowed.

Bit 10: User Read (UR)—If the UR bit is 1, User-mode load operations from the virtual page are allowed; if it is 0, User-mode loads are not allowed.

Bit 9: User Write (UW)—If the UW bit is 1, User-mode store operations to the virtual page are allowed; if it is 0, User-mode stores are not allowed.

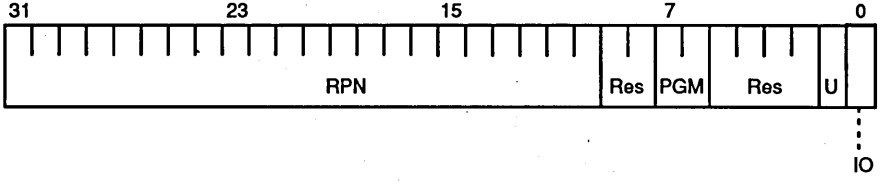
Bit 8: User Execute (UE)—If the UE bit is 1, User-mode instruction accesses to the virtual page are allowed; if it is 0, User-mode instruction accesses are not allowed.

Bits 7–0: Task Identifier (TID)—When the TLB is searched for an address translation, the TID must match the Process Identifier (PID) in the MMU Configuration Register for the translation to be successful. This field allows the TLB entry to be associated with a particular process.

7.2.2 TLB Entry Word 1

The TLB Entry Word 1 Register is shown in Figure 7-4.

Figure 7-4 TLB Entry Word 1 Register



Bits 31–10: Real Page Number (RPN)—The RPN field gives the most-significant 22, 21, 20, or 19 bits of the physical address of the page for page sizes of 1, 2, 4, and 8K bytes, respectively. It is concatenated to bits 9–0, 10–0, 11–0, or 12–0 of the address being translated—for 1, 2, 4, and 8K byte page sizes, respectively—to form the physical address for the access.

When software loads a TLB entry with an address translation, the most-significant 19 bits of the Real Page Number are set with the most-significant 19 bits of the physical address associated with the translation. The remaining three bits of the Real Page Number must be set either to the corresponding bits of the physical address, or to

zeros, depending on the page size, as follows (*A* refers to corresponding address bits):

Page Size	RPN 2–0 (TLB Word 1 bits 12–10)
1K bytes	AAA
2K bytes	AA0
4K bytes	A00
8K bytes	000

Bits 7–6: User Programmable (PGM)—These bits are placed on the MPGM(1–0) outputs when the address is transmitted for an access. They have no predefined effect on the access; any effect is defined by logic external to the processor.

Bit 1: Usage (U)—This bit indicates which entry in a given TLB set was least recently used to perform an address translation. If this bit is a 0, the entry in Column 0 in the set is least recently used; if it is 1, the entry in Column 1 is least recently used. This bit has an equal value for both entries in a set. Whenever a TLB entry is used to translate an address, the Usage bit of each entry in the set used for translation is set according to the TLB set containing the translation. This bit is set whenever the translation is valid, regardless of the outcome of memory-protection checking.

Bit 0: Input/Output (IO)—The IO bit determines whether the access is directed to the instruction/data memory (IO = 0) or the input/output (IO = 1) address space.

7.3 ADDRESS TRANSLATION CONTROLS

Address translation is controlled by the MMU Configuration Register and the Current Processor Status (CPS) register. This section discusses the control of the MMU through the use of these registers.

7.3.1 Enabling and Disabling Address Translation

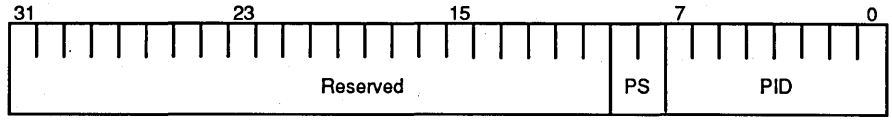
The processor attempts to perform address translation for the following external accesses.

1. Instruction accesses, if the Physical Addressing/Instructions (PI) bit of the Current Processor Status (CPS) register is 0.
2. User-mode accesses to instruction/data memory if the Physical Addressing/Data (PD) bit of the CPS is 0.
3. Supervisor-mode accesses to instruction/data memory if the Physical Address (PA) bit of the load or store instruction performing the access is 0, and the PD bit of the CPS is 0.

7.3.2 MMU Configuration Register (MMU, Register 13)

This protected special-purpose register (Figure 7-5) specifies parameters associated with the MMU.

Figure 7-5 MMU Configuration Register



Bits 31–10: Reserved.

Bits 9–8: Page Size (PS)—The PS field specifies the page size for address translation. The page size affects translation as discussed in Sections 7.2.1, 7.2.2, and 7.4. The PS field has a delayed effect on address translation (see Section 5.6). At least one cycle of delay must separate an instruction which sets the PS field and an instruction that performs address translation. The PS field is encoded as follows:

PS	Page Size
0 0	1K bytes
0 1	2K bytes
1 0	4K bytes
1 1	8K bytes

Bits 7–0: Process Identifier (PID)—For translated User-mode loads and stores, this 8-bit field is compared to Task Identifier (TID) fields in Translation Look-Aside Buffer entries when address translation is performed. For the address translation to be valid, the PID field must match the TID field in an entry. This allows a separate 32-bit virtual-address space to be allocated to each active User-mode process (within the limit of 255 such processes). Translated Supervisor-mode loads and stores use a fixed process identifier of zero, and require that the TID field be zero for successful translation.

7.4 ADDRESS TRANSLATION DESCRIPTION

For the purpose of address translation, the virtual instruction/data address-space of a process is typically partitioned into regions of fixed size, called pages. Pages are mapped into equivalent-sized regions of physical memory, called page frames. All accesses to instructions or data contained within a given page use the same virtual-to-physical address translation.

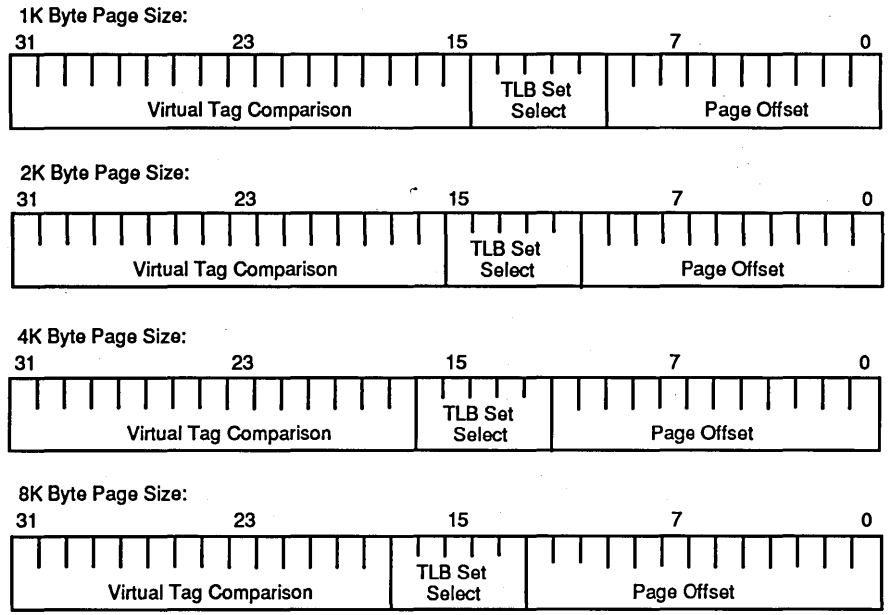
7.4.1 Virtual Address Structure

Virtual addresses are partitioned into three fields for TLB address translation, as shown in Figure 7-6. The partitioning of the virtual address is based on the page size. Pages may be of size 1, 2, 4, or 8K bytes, as specified by the MMU Configuration Register.

7.4.2 Address-Translation Process

The TLB address-translation process is diagrammed in Figure 7-7. Address translation is performed by the following fields in the TLB entry: the Virtual Tag (VTAG), the Task Identifier (TID), the Valid Entry (VE) bit, the Real Page Number (RPN) field, and the Input/Output (IO) bit. To perform an address translation, the processor accesses the TLB set whose number is given by certain bits in the virtual address. The bits used depend on the page size as follows.

Figure 7-6 Virtual Address for 1, 2, 4, and 8K Byte Pages



Page Size	Virtual Address Bits (for Set Access)
1K bytes	14–10
2K bytes	15–11
4K bytes	16–12
8K bytes	17–13

The accessed set contains two TLB entries, which in turn contain two VTAG fields. The VTAG fields are both compared to bits in the virtual address. This comparison depends on the page size as follows (note that VTAG bit numbers are relative to the VTAG field, not the TLB entry).

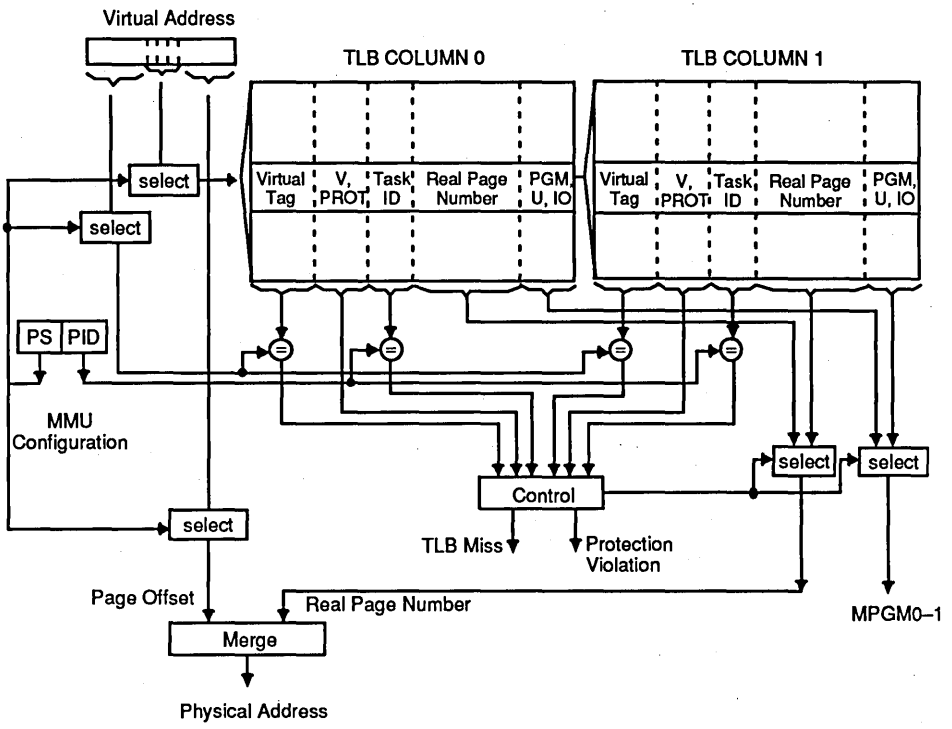
Page Size	Virtual Address Bits	VTAG Bits
1K bytes	31–15	16–0
2K bytes	31–16	16–1
4K bytes	31–17	16–2
8K bytes	31–18	16–3

Certain bits of the VTAG field do not participate in the comparison for page sizes larger than 1K byte. These bits of the VTAG field are required to be zero.

For an address translation to be valid, the following conditions must be met:

1. The virtual address bits match corresponding bits of the VTAG field as specified above.
2. For a User-mode access, the TID field in the TLB entry matches the PID field in the MMU Configuration Register. For a Supervisor-mode access, the TID field is zero.

Figure 7-7 TLB Address-Translation Process



3. The VE bit in the TLB entry is 1.
4. Only one entry in the set meets conditions 1, 2, and 3 above. If this condition is not met, the results of the translation may be treated as valid by the processor, but the results are unpredictable.

If the address translation is valid for one TLB entry in the selected set, the RPN field in this entry is used to form the physical address of the access. The RPN field gives the portion of the physical address that depends on the translation; the remaining portion of the virtual address—called the Page Offset—is invariant with address translation.

The Page Offset comprises the low-order bits of the virtual address and gives the location of a byte within the virtual page (because of byte addressing). This byte is located at the same position in the physical page frame, so the Page Offset also comprises the low-order bits of the physical address.

The 32-bit physical address is the concatenation of certain bits of the RPN field and Page Offset, where the bits from each depend on the page size as follows (note that RPN bit numbers are relative to the RPN field, not the TLB entry).

Page Size	RPN Bits	Virtual Address Bits for Page Offset
1K bytes	21–0	9–0
2K bytes	21–1	10–0
4K bytes	21–2	11–0
8K bytes	21–3	12–0

Note: Certain bits of the RPN field are not used in forming the physical address for page sizes greater than 1K byte. These bits of the RPN are required to be zero.

The address space of the physical address is determined by the Input/Output (IO) bit of the TLB entry. If the IO bit is 0, the address is in the instruction/data memory address space. If the IO bit is 1, the address is in the input/output address space.

7.4.3 Successful and Unsuccessful Translations

If an address translation is successful, the TLB entry is further used to perform protection checking for the access. Bits in the TLB make it possible to restrict accesses—independently for Supervisor-mode and User-mode accesses—to any combination of load, store, and instruction accesses, or to no access. Section 6.3 describes MMU protection in more detail.

If the address translation is valid, and no protection violation is detected, the physical address from the translation is placed on the processor's Address Bus, and the access is initiated. If the translation is not valid, or a protection violation is detected, a trap occurs.

Also, if the address translation is successful, and there is no protection violation, the PGM bits from the TLB entry used for translation are placed on the MPGM(1–0) outputs during the address cycle for the access. If address translation is not performed, these pins are both Low for the address cycle.

If the TLB cannot translate an address, a TLB miss occurs. The MMU causes a trap if either a TLB miss occurs, or the translation is successful and a protection violation is detected. The processor distinguishes between traps caused by instruction and data accesses, and between traps caused by User- and Supervisor-mode accesses, as follows:

Trap Vector Number	Type of Trap
8	User-Mode Instruction TLB Miss
9	User-Mode Data TLB Miss
10	Supervisor-Mode Instruction TLB Miss
11	Supervisor-Mode Data TLB Miss
12	Instruction MMU Protection Violation
13	Data MMU Protection Violation

The distinction between the above traps is made to assist trap handling, particularly the routines that load TLB entries.

7.4.4 Instruction Cache Considerations

The Instruction Cache is accessed with virtual as well as physical addresses, depending on whether address translation is enabled for instruction accesses. Because of

this, the Instruction Cache may contain entries that might be considered valid, even though they are not.

For example, address translation may be changed by modifying the Process Identifier of the MMU Configuration Register. This change is not reflected in the Instruction Cache tags, so the tags do not necessarily perform valid comparisons.

If a TLB miss occurs during the address translation for either a branch target instruction or an instruction on a new virtual page, the processor considers the contents of the Instruction Cache to be invalid. This is required to properly sequence the LRU Recommendation Register, and does not solve the problem just described. If the TLB is changed at some point, so that the TLB miss does not occur, the Instruction Cache still may perform an invalid comparison.

To avoid the above problem, the contents of the Instruction Cache must be invalidated explicitly whenever address translation is changed. This can be accomplished by executing an Invalidate (INV) instruction whenever an address translation is changed. The INV instruction causes all entries of the Instruction Cache to become invalid (after the next successful branch or cache block boundary). However, since the change in address translation rarely affects the program performing the change, the INV may unnecessarily affect the performance of this program.

The IRETINV instruction has the same effect on the Instruction Cache as the INV instruction, but can reduce the performance impact. The IRETINV delays invalidation until an interrupt return is executed, eliminating the need to disrupt an operating-system routine when the routine changes address translation. At the point of interrupt return, the contents of the Instruction Cache are most likely not of much use anyway.

Note that the Instruction Cache is not invalidated when the Instruction Cache Disable (ID) bit of the Configuration Register is set. When the ID bit is 1, the Instruction Cache retains its previous contents, but the processor considers its contents to be invalid. Thus, the ID bit cannot be used to invalidate the cache, and, furthermore, the Instruction Cache may have to be invalidated whenever the ID bit is to be reset (i.e., when the cache is to be enabled).

The Instruction Cache distinguishes between virtual and physical addresses and between User-mode and Supervisor-mode addresses. Thus, the Instruction Cache does not have to be invalidated on transitions between these address spaces. This improves the performance of applications that make heavy use of operating-system routines in either physical or virtual address space.

7.4.5 Selecting the Virtual Page Size

The selection of page size is based on several considerations:

1. For a given page size, any allocation of pages to a process will, on average, waste half of one page. With smaller page sizes, the waste is smaller. In systems with a large number of processes, each with a small amount of memory, small page sizes can reduce waste significantly.
2. Smaller page sizes allow finer memory-protection granularity.
3. The maximum amount of memory that can be referenced by Translation Look-Aside Buffer (TLB) entries is set by the number of TLB entries and the page size. Larger page sizes allow the fixed number of TLB entries to address more memory, and generally reduce the number of TLB misses. For example, with 1-Kbyte pages, a process requiring 8K bytes of contiguous memory would create eight TLB misses; with 8-Kbyte pages, the process would create only one TLB miss.

4. The page is usually the unit of memory moved between memory and backing storage. The design of the backing storage sub-system also may influence the choice of page size, because of transfer-efficiency considerations. For example, if the backing storage is a disk, the disk seek time is large compared to transfer time. Thus, it is more efficient to transfer large amounts of data with a single seek. Efficiency may also depend on disk organization (i.e., the number of seeks possibly required to transfer a page).

7.5 HANDLING TLB MISSES

The address translation performed by the MMU is ultimately determined by routines that place entries into the Translation Look-Aside Buffer (TLB). TLB entries normally are based on system page tables, which give the translation for a large number of pages. The TLB simply caches the currently needed translations, so that system page tables do not have to be accessed for every translation.

If a required address translation cannot be performed by any entry in the TLB, a TLB miss trap occurs. The trap handling routine—called the TLB reload routine—accesses the system page tables to determine the required translation and sets the appropriate TLB entry. Note that the access requiring this translation can be restarted by the interrupt return at the end of the TLB reload routine (see Section 8.6.2).

A large number of different page-table organizations are possible. Since the TLB reload routine is a sequence of processor instructions, the page tables may have a structure and access method that satisfies trade-offs of page table size, translation lookup time, and memory-allocation strategies.

Another possibility supported by the TLB reload mechanism is that of a second-level TLB. The TLB reload routine is not required to access the system page tables immediately upon a TLB miss, but may access an external TLB, which can be much larger than the processor's TLB. The amount of time required to access the external TLB normally is much smaller than the amount of time required to access the page tables, leading to an overall improvement in performance. Of course, if a translation is not in the external TLB, a page table lookup still must be performed.

Because the TLB reload routine may depend on the type of access causing the TLB miss, the processor differentiates between misses on instruction and data accesses and between misses by Supervisor-mode and User-mode programs. This eliminates any time which might be spent by the TLB reload routine in making the same determination. Performance is also enhanced by the LRU Recommendation Register, which gives the TLB register number for Word 0 of the TLB entry to be replaced by the TLB reload routine (the least recently used entry).

7.5.1 TLB Reload

So that the MMU may support a large variety of memory-management architectures, it does not directly load TLB entries that are required for address translation. It simply causes a TLB miss trap when address translation is unsuccessful. The trap causes a program—called the TLB reload routine—to execute. The TLB reload routine is defined according to the structure and access method of the page table contained in an external device or memory.

When a TLB miss trap occurs, the LRU Recommendation Register contains the TLB register number for Word 0 of the TLB entry to be used by the TLB reload routine. For instruction accesses, the Program Counter 1 Register contains the instruction address that was not successfully translated. For data accesses, the Channel Address Register contains the data address that was not successfully translated.

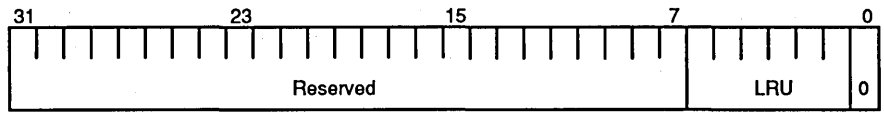
The TLB reload routine determines the translation for the address given by the Program Counter 1 Register or Channel Address Register, as appropriate. The TLB reload routine uses an external page table to determine the required translation, and loads the TLB entry indicated by the LRU Recommendation Register so that the entry may perform this translation. In a demand-paged environment, the TLB reload routine may additionally invoke a page-fault handler when the translation cannot be performed.

TLB entries are written by the Move To TLB (MTTLB) instruction, which copies the contents of a general-purpose register into a TLB register. The TLB register number is specified by bits 6–0 of a general-purpose register. TLB entries are read by the Move From TLB (MFTLB) instruction, which copies the contents of a TLB register into a general-purpose register. Again, the TLB register number is specified by a general-purpose register.

7.5.2 LRU Recommendation (LRU, REGISTER 14)

This protected special-purpose register (Figure 7-8) assists Translation Look-Aside Buffer (TLB) reloading by indicating the least recently used TLB entry in the required replacement set.

Figure 7-8 LRU Recommendation Register



Bits 31–7: Reserved.

Bits 6–1: Least-Recently Used Entry (LRU)—The LRU field is updated whenever a TLB miss occurs during an address translation. It gives the TLB register number of the TLB entry selected for replacement. The LRU field also is updated whenever a memory-protection violation occurs; however, it has no interpretation in this case.

Bit 0: Zero—The appended 0 serves to identify Word 0 of the TLB entry.

7.5.3 Page Reference And Change Information

In a demand-paged environment, it is important to be able to collect information on the use and modification of pages. The processor does not collect this information directly, but the information may be collected by the operating system, without requiring hardware support.

Each TLB entry contains six bits which specify the type of accesses that are permitted for the corresponding page. When a TLB entry is loaded, the TLB reload routine can set the protection bits so that an access to the corresponding page is not allowed. If an access is attempted, an MMU Protection Violation traps occurs. This trap may be used to signal that the page is being referenced. After noting this fact, the trap handler may set the protection bits to allow the access and return to the trapping routine.

A technique similar to the one just described can be used to collect information on the modification of a page. However, in this case, the TLB protection bits initially are set so that a store is not allowed.

It is also possible to create reference information by noting references during TLB reload. For example, reference bits normally are reset periodically, so that they reflect current references. When reference bits are reset, the entire TLB may be invalidated. Reference bits are set as TLB entries are loaded. Note that this scheme relies on the fact that a TLB miss implies a reference to the corresponding page. Also, this scheme does not account for page change information.

The disadvantage of both of the above schemes is one of possible performance loss. This is the result of the additional traps required to monitor page references and changes. If the performance impact is unacceptable, references and changes can be monitored easily by hardware that detects reads and writes to page frames in instruction or data memory.

7.5.4 Warm Start

When a process switch occurs, there is a high probability that most of the TLB entries of the old process will not be used by the new process. Thus, the new process most likely creates many TLB miss traps early in its execution. This is unavoidable on the first initiation of a process, but may be prevented on subsequent initiations.

When a given process is suspended, the operating system can save a copy of the process' TLB contents. When the process is restarted, the copy can be loaded back into the TLB. This warm start prevents many of the process' initial TLB misses, at the expense of the time required to save and restore the copy of the TLB entries. However, this time may be much shorter than the time required to individually perform all TLB reloads.

Note that if this warm-start strategy is adopted, any change in address translation must be reflected in all copies of TLB entries for all affected processes. If address translation is often changed so that it affects more than one process, warm start may not be advantageous.

7.5.5 Minimum Number Of Resident Pages

In any processor that supports demand paging, there are a minimum number of pages that must be resident for any active process. This minimum is determined by the maximum number of pages that might be referenced by an atomic operation in the processor's architecture (e.g., an instruction, normally). If this maximum number is not guaranteed to be resident in memory, some operations might never complete, since they may never have all of the required pages resident in memory at one time.

For the Am29030 and Am29035 microprocessors, two pages are required for a process to make progress through the system. The reason for this requirement is that the Am29030 and Am29035 microprocessors, on interrupt return, restart an interrupted Load Multiple or Store Multiple only after fetching two instructions (see Section 8.3.4). The first of these instructions must be resident in memory—and mapped by the TLB—and the page required to complete the Load Multiple or Store Multiple must also be resident—and mapped by the TLB—for the interrupt return to complete successfully.

7.6 INVALIDATING TLB ENTRIES

There are two methods for invalidating TLB entries that are no longer required at a given point in program execution. The first involves resetting the Valid Entry bit of a single entry (this is done by a Move To TLB instruction). The second involves changing the value of the Process Identifier (PID) field of the MMU Configuration Register;

this invalidates all entries whose Task Identifier (TID) fields do not match the new value.

If an entry is invalidated by changing the PID field, the TLB entry still remains valid in some sense. If the PID field is changed again to match the TID field, the entry may once again participate in address translation. This ability can be used to reduce the number of TLB misses in a system during process switching. However, it is important to manage TLB entries so that an invalid match cannot occur between the PID field and the TID field of an old TLB entry.

INTERRUPTS AND TRAPS

8.1 OVERVIEW

Interrupts and traps cause the Am29030 and Am29035 microprocessors to suspend the execution of an instruction sequence and to begin the execution of a new sequence. The processor may or may not later resume the execution of the original instruction sequence.

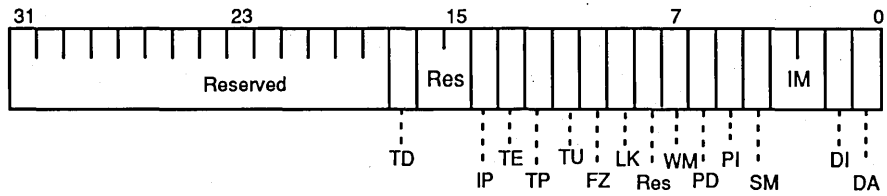
The distinction between interrupts and traps is largely one of causation and enabling. Interrupts allow external devices and the Timer Facility to control processor execution and are always asynchronous to program execution. Traps are intended to be used for certain exceptional events that occur during instruction execution and are generally synchronous to program execution.

A distinction is made between the point at which an interrupt or trap occurs and the point at which it is taken. An interrupt or trap is said to occur when all conditions that define the interrupt or trap are met. However, an interrupt or trap that occurs is not necessarily recognized by the processor, either because of various enables or because of the processor's operational mode (e.g., Halt mode). An interrupt or trap is taken when the processor recognizes the interrupt or trap and alters its behavior accordingly.

8.1.1 Current Processor Status (CPS, Register 2)

This protected special-purpose register (see Figure 8-1) controls the behavior of the processor and its ability to recognize exceptional events.

Figure 8-1 Current Processor Status Register



Bits 31–18: Reserved.

Bits 17: Timer Disable (TD)—When the TD bit is 1, the Timer interrupt is disabled. When this bit is 0, the Timer interrupt is dependant on the value of the IE bit of the Timer Reload Register. Note that Timer interrupts may be disabled by the DA bit regardless of the value of either TD or IE. The intent of this bit is to provide a means of disabling Timer interrupts without having to perform a non-atomic read-modify-write operation on the Timer Reload Register.

Bit 16–15: Reserved.

Bit 14: Interrupt Pending (IP)—This bit allows software to detect the presence of external interrupts while the interrupts are disabled. The IP bit is set if one or more of the external signals $\overline{\text{INTR}}$(3–0) is active, but the processor is disabled from taking the resulting interrupt due to the value of the DA, DI, or IM bits. If all external interrupt signals are subsequently de-asserted while still disabled, the IP bit is reset.

Bits 13–12: Trace Enable, Trace Pending (TE, TP)—The TE and TP bits implement a software-controlled, instruction single-step facility. Single stepping is not implemented directly, but rather emulated by trap sequences controlled by these bits. The value of the TE bit is copied to the TP bit whenever an instruction completes execution. When the TP bit is 1, a Trace trap occurs. Section 11.1 describes the use of these bits in more detail.

Bit 11: Trap Unaligned Access (TU)—The TU bit enables checking of address alignment for external data-memory accesses. When this bit is 1, an Unaligned Access trap occurs if the processor either generates an address for an external word that is not aligned on a word address-boundary (i.e., either of the least-significant two bits is 1) or generates an address for an external half-word that is not aligned on a half-word address boundary (i.e., the least-significant address bit is 1). When the TU bit is 0, data-memory address alignment is ignored.

Alignment is ignored for input/output accesses. The alignment of instruction addresses is also ignored (unaligned instruction addresses can be generated only by indirect jumps). Interrupt/trap vector addresses always are aligned properly by the processor.

Bit 10: Freeze (FZ)—The FZ bit prevents certain registers from being updated during interrupt and trap processing, except by explicit data movement. The affected registers are: Channel Address, Channel Data, Channel Control, Program Counter 0, Program Counter 1, Program Counter 2, and the ALU Status Register.

When the FZ bit is 1, these registers hold their values. An affected register can be changed only by a Move-To-Special-Register instruction. When the FZ bit is 0, there is no effect on these registers, and they are updated by processor instruction execution as described in this manual.

The FZ bit is set whenever an interrupt or trap is taken, holding critical state in the processor so that it is not modified unintentionally by the interrupt or trap handler.

Bit 9: Lock (LK)—The LK bit controls the value of the $\overline{\text{LOCK}}$ external signal. If the LK bit is 1, the $\overline{\text{LOCK}}$ signal is active. If the LK bit is 0, the $\overline{\text{LOCK}}$ signal is controlled by the execution of the instructions Load and Set, Load and Lock, and Store and Lock. This bit is provided for the implementation of multi-processor synchronization protocols.

Bit 8: Reserved.

Bit 7: WAIT Mode (WM)—The WM bit places the processor in the Wait mode. When this bit is 1, the processor performs no operations. The Wait mode is reset by an interrupt or trap for which the processor is enabled, or by the assertion of the $\overline{\text{RESET}}$ pin.

Bit 6: Physical Addressing/Data (PD)—The PD bit determines whether address translation is performed for load or store operations. Address translation is performed for an access only when this bit is 0 and the Physical Address (PA) bit in the load or store instruction causing the access is also 0.

Bit 5: Physical Addressing/Instructions (PI)—The PI bit determines whether address translation is performed for external instruction accesses. Address translation is performed only when this bit is 0.

Bit 4: Supervisor Mode (SM)—The SM bit protects certain processor context, such as protected special-purpose registers. When this bit is 1, the processor is in the Supervisor mode, and access to all processor context is allowed. When this bit is 0, the processor is in the User mode, and access to protected processor context is not allowed; an attempt to access (either read or write) protected processor context causes a Protection Violation trap.

Section 6.1 describes the processor state protected from User-mode access.

For an external access, the User Access (UA) bit in the load or store instruction also controls access to protected processor context. When the UA bit is 1, the Memory Management Unit and bus perform the access as if the program causing the access were in User mode.

Bits 3–2: Interrupt Mask (IM)—The IM field is an encoding of the processor priority with respect to external interrupts. The interpretation of the interrupt mask is specified in Section 8.1.2.

Bit 1: Disable Interrupts (DI)—The DI bit prevents the processor from being interrupted by external interrupt requests $\overline{\text{INTR}}$(3–0). When this bit is 1, the processor ignores all external interrupts. However, note that traps (both internal and external), Timer interrupts, and Trace traps may be taken. When this bit is 0, the processor takes any interrupt enabled by the IM field, unless the DA bit is 1.

Bit 0: Disable All Interrupts and Traps (DA)—The DA bit prevents the processor from taking any interrupts and most traps. When this bit is 1, the processor ignores interrupts and traps, except for the $\overline{\text{WARN}}$, Instruction Access Exception, and Data Access Exception traps. When the DA bit is 0, all traps are taken, and interrupts are taken if otherwise enabled.

8.1.2 Interrupts

Interrupts are caused by signals applied to any of the external inputs $\overline{\text{INTR}}$(3–0) or by the Timer Facility (see Section 8.7). The processor may be disabled from taking certain interrupts by the masking capability provided by the Disable All Interrupts and Traps (DA) bit, Disable Interrupts (DI) bit, and Interrupt Mask (IM) field in the Current Processor Status Register.

The DA bit disables all interrupts. The DI bit disables external interrupts without affecting the recognition of traps and Timer interrupts. The 2-bit IM field selectively enables external interrupts as follows:

IM Value	Result
00	$\overline{\text{INTR}}$0 enabled
01	$\overline{\text{INTR}}$(1–0) enabled
10	$\overline{\text{INTR}}$(2–0) enabled
11	$\overline{\text{INTR}}$(3–0) enabled

Note that the $\overline{\text{INTR}}$0 interrupt cannot be disabled by the IM field. Also, note that no external interrupt is taken if either the DA or DI bit is 1. The Interrupt Pending bit in the Current Processor Status indicates that one or more of the signals $\overline{\text{INTR}}$(3–0) is active, but that the corresponding interrupt is disabled due to the value of either DA, DI, or IM.

8.1.3 Traps

Traps are caused by signals applied to one of the inputs $\overline{\text{TRAP}}$(1–0), or by exceptional conditions such as protection violations. Except for the Instruction Access Exception and Data Access Exception traps, traps are disabled by the DA bit in the Current Processor Status; a 1 in the DA bit disables traps, and a 0 enables traps. It is not possible to selectively disable individual traps.

8.1.4 External Interrupts And Traps

An external device causes an interrupt by asserting one of the $\overline{\text{INTR}}$(3–0) inputs, and causes a trap by asserting one of the $\overline{\text{TRAP}}$(1–0) inputs. Transitions on each of these inputs may be asynchronous to the processor clock; they are protected against metastable states. For this reason, an assertion of one of these inputs that meets the proper set-up-time criteria does not cause the corresponding interrupt or trap until the second following cycle.

The $\overline{\text{INTR}}$(3–0) inputs are prioritized with respect to each other and with respect to the processor. To resolve conflicts between these inputs, the inputs are prioritized in order, so that the interrupt caused by $\overline{\text{INTR}}$0 has the highest priority, and the interrupt caused by $\overline{\text{INTR}}$3 has the lowest priority.

The $\overline{\text{TRAP}}$(1–0) inputs are prioritized with respect to each other, so that the trap caused by $\overline{\text{TRAP}}$0 has priority over the trap caused by $\overline{\text{TRAP}}$1 when a conflict occurs. Both $\overline{\text{TRAP}}$0 and $\overline{\text{TRAP}}$1 have priority over the $\overline{\text{INTR}}$(3–0) inputs. The $\overline{\text{TRAP}}$(1–0) inputs cannot be disabled selectively. Both traps, however, can be disabled by the DA bit in the Current Processor Status Register.

The $\overline{\text{INTR}}$(3–0) and $\overline{\text{TRAP}}$(1–0) inputs are level-sensitive. Once asserted, they must be held active until the corresponding interrupt or trap is acknowledged by the interrupt or trap handler (this acknowledgment is system-dependent, since there is no interrupt-acknowledge mechanism defined for the processor).

If any of these inputs is asserted, then de-asserted before it is acknowledged, it is not possible to predict (unless the interrupt or trap is masked) whether or not the processor has taken the corresponding interrupt or trap. During interrupt and trap processing, the vector number is determined in part by which of the $\overline{\text{INTR}}$(3–0) and $\overline{\text{TRAP}}$(1–0) inputs is active. If the input causing an interrupt or trap is de-asserted before the vector number is determined, the vector number is unpredictable, with the result that processor operation is also unpredictable. Typically, this situation results in the processor taking an Illegal Opcode trap.

There is a three-cycle latency from the de-assertion of an $\overline{\text{INTR}}$(3–0) or $\overline{\text{TRAP}}$(1–0) input to the time that the corresponding interrupt or trap is actually not recognized by the processor. The de-assertion must be timed so that, when the corresponding mask is reset, the processor does not recognize the interrupt or trap. Otherwise, a spurious interrupt or trap may occur.

8.1.5 Wait Mode

A wait-for-interrupt capability is provided by the Wait mode. The processor is in the Wait mode whenever the Wait Mode (WM) bit of the Current Processor Status is 1. While in Wait mode, the processor neither fetches nor executes instructions and performs no external accesses. The Wait mode is exited when an interrupt or trap is taken.

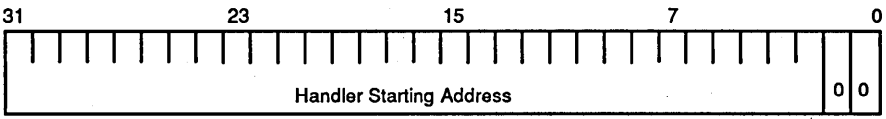
Note that the processor can take only those interrupts or traps for which it is enabled, even in the Wait mode. For example, if the processor is in the Wait mode with a DA bit of 1, it can leave the Wait mode only via a processor reset (see Section 10.2) or a $\overline{\text{WARN}}$ trap (see Section 8.4).

8.2 VECTOR AREA

Interrupt and trap processing relies on the existence of a user-managed Vector Area in external instruction/data memory. The Vector Area begins at an address specified by the Vector Area Base Address Register and provides for as many as 256 different interrupt and trap handling routines. The processor reserves 64 routines for system operation and instruction emulation. The number and definition of the remaining 192 possible routines are system dependent.

The structure of the Vector Area is a table of vectors in instruction/data memory. The layout of a single vector is shown in Figure 8-2. Each vector gives the beginning word-address of the associated interrupt or trap handling routine.

Figure 8-2 Vector Table Entry

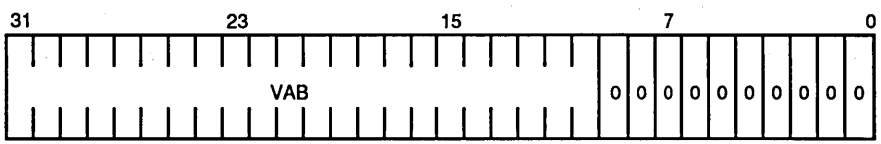


8.2.1 Vector Area Base Address (VAB, Register 0)

This protected special-purpose register (see Figure 8-3) specifies the beginning address of the interrupt/trap Vector Area. The Vector Area is a table of 256 vectors which point to interrupt and trap handling routines.

When an interrupt or trap is taken, the vector number for the interrupt or trap (see Section 8.2.2) replaces bits 9–2 of the value in the Vector Area Base Address Register to generate the physical address for a vector contained in instruction/data memory.

Figure 8-3 Vector Area Base Address Register



Bits 31–10: Vector Area Base (VAB)—The VAB field gives the beginning physical address of the Vector Area. This address is constrained to begin on a 1K-Byte address-boundary in instruction/data memory.

Bits 9–0: Zeros—These bits force the alignment of the Vector Area to a 1K-Byte boundary.

8.2.2 Vector Numbers

When an interrupt or trap is taken, the processor determines an 8-bit vector number associated with the interrupt or trap. The vector number gives the number of a vector table entry. The physical address of the vector table entry is generated by replacing bits 9–2 of the value in the Vector Area Base Address Register with the vector number.

Vector numbers are either predefined or specified by an instruction causing the trap. The assignment of vector numbers is shown in Table 8-1 (vector numbers are in decimal notation). Vector numbers 64 to 255 are for use by trapping instructions; the definition of the routines associated with these numbers is system dependent.

8.3 INTERRUPT AND TRAP HANDLING

Interrupt and trap handling consists of two distinct operations: taking the interrupt or trap and returning from the interrupt or trap handler. If the interrupt or trap handler returns directly to the interrupted routine, the interrupt or trap handler need not save and restore processor state.

8.3.1 Old Processor Status (OPS, Register 1)

This protected special-purpose register has the same format as the Current Processor Status Register. The Old Processor Status Register stores a copy of the Current Processor Status Register when an interrupt or trap is taken. This is required since the Current Processor Status Register is modified to reflect the status of the interrupt/trap handler.

During an interrupt return, the Old Processor Status Register is copied into the Current Processor Status Register. This allows the Current Processor Status Register to be set as required for the routine that is the target of the interrupt return.

8.3.2 The Program Counter Stack

The Program Counter Unit, shown in Figure 8-4, forms and sequences instruction addresses for the Instruction Fetch Unit. It contains the Program Counter (PC), the Program-Counter Multiplexer (PC MUX), the Return Address Latch, and the Program-Counter Buffer (PC Buffer).

The PC forms addresses for sequential instructions executed by the processor. The master of the PC Register, PC L1, contains the address of the instruction being fetched in the Instruction Fetch Unit. The slave of the PC Register, PC L2, contains the next sequential address, which may be fetched by the Instruction Fetch Unit in the next cycle.

The Return Address Latch passes the address of the instruction following the delayed instruction of a call to the register file. This address is the return address of the call.

The PC Buffer stores the addresses of instructions in various stages of execution when an interrupt or trap is taken. The registers in this buffer—Program Counters 0, 1, and 2 (PC0, PC1, and PC2)—are normally updated from the PC as instructions flow through the processor pipeline.

When an interrupt or trap is taken, the Freeze (FZ) bit in the Current Processor Status is set, holding the quantities in the PC Buffer. When the FZ bit is set, PC0, PC1, and PC2 contain the addresses of the instructions in the decode, execute, and write-back stages of the pipeline, respectively.

Table 8-1 Vector Number Assignments

Number	Type of Trap or Interrupt	Cause
0	Illegal Opcode	Executing undefined instruction[1]
1	Unaligned Access	Access on unnatural boundary, TU = 1
2	Out of Range	Overflow or underflow
3–4	Reserved	
5	Protection Violation	Invalid User-mode operation[2]
6	Instruction Access Exception	$\overline{ERR}$ response while instruction fetching
7	Data Access Exception	$\overline{ERR}$ response, doing load or store
8	User-Mode Instruction TLB Miss	No TLB entry for translation
9	User-Mode Data TLB Miss	No TLB entry for translation
10	Supervisor-Mode Instruction TLB Miss	No TLB entry for translation
11	Supervisor-Mode Data TLB Miss	No TLB entry for translation
12	Instruction MMU Protection Violation	TLB UE/SE = 0
13	Data MMU Protection Violation	TLB UR/SR = 0, UW/SW = 0 on write
14	Timer	Timer Facility
15	Trace	Trace Facility
16	$\overline{INTR0}$	$\overline{INTR0}$ input
17	$\overline{INTR1}$	$\overline{INTR1}$ input
18	$\overline{INTR2}$	$\overline{INTR2}$ input
19	$\overline{INTR3}$	$\overline{INTR3}$ input
20	$\overline{TRAP0}$	$\overline{TRAP0}$ input
21	$\overline{TRAP1}$	$\overline{TRAP1}$ input
22	Floating-Point Exception	Unmasked floating-point exception[3]
23	Reserved	
24–29	Reserved for instruction emulation (opcodes D8–DD)	
30	MULTM	MULTM instruction
31	MULTMU	MULTMU instruction
32	MULTIPLY	MULTIPLY instruction
33	DIVIDE	DIVIDE instruction
34	MULTIPLU	MULTIPLU instruction
35	DIVIDU	DIVIDU instruction
36	CONVERT	CONVERT instruction
37	SQRT	SQRT instruction
38	CLASS	CLASS instruction
39–41	Reserved for instruction emulation (opcode E7–E9)	
42	FEQ	FEQ instruction
43	DEQ	DEQ instruction
44	FGT	FGT instruction
45	DGT	DGT instruction
46	FGE	FGE instruction
47	DGE	DGE instruction
48	FADD	FADD instruction
49	DADD	DADD instruction
50	FSUB	FSUB instruction
51	DSUB	DSUB instruction
52	FMUL	FMUL instruction
53	DMUL	DMUL instruction

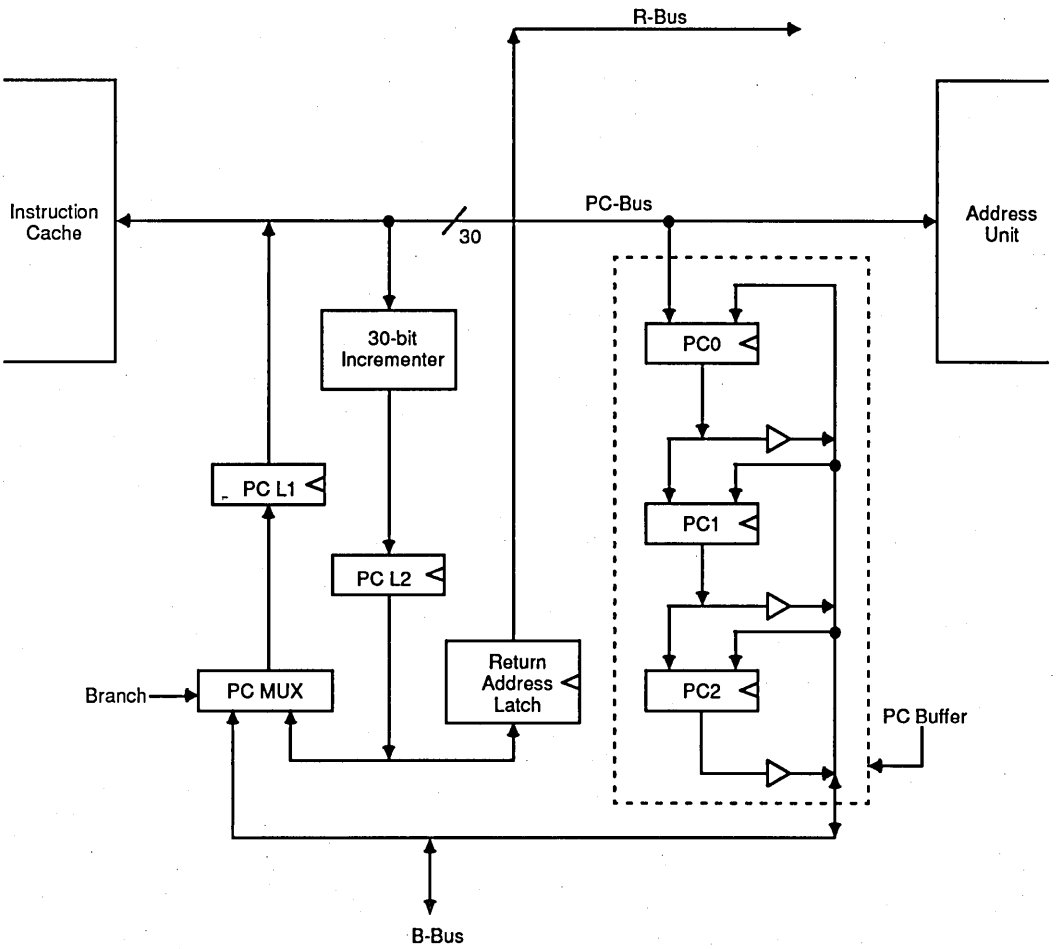
1. This vector number also results if an external device removes $\overline{INTR3}$–$\overline{INTR0}$ or $\overline{TRAP1}$–$\overline{TRAP0}$ before the corresponding interrupt or trap is taken by the processor.
2. Some Supervisor-mode operations cause Protection Violations, to facilitate virtualization of certain operations.
3. The Floating-Point Exception trap is not generated by the processor hardware. It must be generated by software support.

Vector Number Assignments (continued)

Number	Type of Trap or Interrupt	Cause
54	FDIV	FDIV instruction
55	DDIV	DDIV instruction
56	Reserved for instruction emulation (opcode F8)	
57	FDMUL	FDMUL instruction
58–63	Reserved for instruction emulation (opcode FA–FF)	
64–255	ASSERT and EMULATE instruction traps (vector number specified by instruction)	

Note: Some of Vector Numbers 64–255 are reserved for software compatability (see Sections 4.2.3 and 4.2.6). These are documented in Chapter 4 and in the Host Interface (HIF) Specification, available from AMD.

Figure 8-4 Program Counter Unit



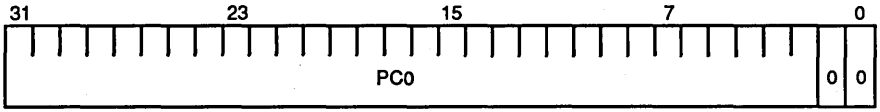
Upon the execution of an interrupt return, the target instruction stream is restarted using the instruction addresses in PC0 and PC1. Two registers are required here because the processor implements delayed branches. An interrupt or trap may be taken when the processor is executing the delay instruction of a branch and decoding the target of the branch. This discontinuous instruction sequence must be restarted properly upon an interrupt return. Restarting the instruction pipeline using two separate registers correctly handles this special case; in this case PC1 points to the delay instruction of the branch, and PC0 points to its target. PC2 does not participate in the interrupt return, but is included to report the addresses of instructions causing certain exceptions.

The PC is not defined as a special-purpose register. It cannot be modified or inspected by instructions. Instead, the interrupting and restarting of the pipeline is done by the PC Buffer registers PC0 and PC1.

8.3.2.1 PROGRAM COUNTER 0 (PC0, Register 10)

This protected special-purpose register (Figure 8-5) is used, on an interrupt return, to restart the instruction which was in the decode stage when the original interrupt or trap was taken.

Figure 8-5 Program Counter 0 Register



Bits 31–2: Program Counter 0 (PC0)—This field captures the word-address of an instruction as it enters the decode stage of the processor pipeline, unless the Freeze (FZ) bit of the Current Processor Status Register is 1. If the FZ bit is 1, PC0 holds its value.

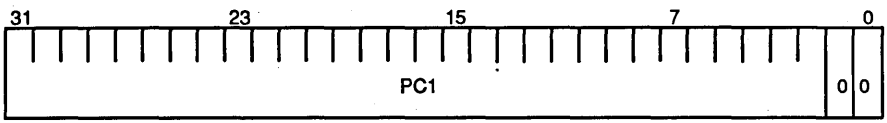
When an interrupt or trap is taken, the PC0 field contains the word-address of the instruction in the decode stage; the interrupt or trap has prevented this instruction from executing. The processor uses the PC0 field to restart this instruction on an interrupt return.

Bits 1–0: Zeros—These bits are zero, since instruction addresses are always word aligned.

8.3.2.2 PROGRAM COUNTER 1 (PC1, Register 11)

This protected special-purpose register (Figure 8-6) is used, on an interrupt return, to restart the instruction that was in the execute stage when the original interrupt or trap was taken.

Figure 8-6 Program Counter 1 Register



Bits 31–2: Program Counter 1 (PC1)—This field captures the word-address of an instruction as it enters the execute stage of the processor pipeline, unless the Freeze

(FZ) bit of the Current Processor Status Register is 1. If the FZ bit is 1, PC1 holds its value.

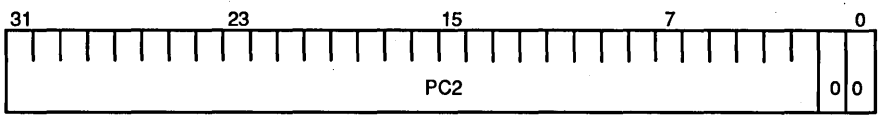
When an interrupt or trap is taken, the PC1 field contains the word-address of the instruction in the execute stage; the interrupt or trap has prevented this instruction from completing execution. The processor uses the PC1 field to restart this instruction on an interrupt return.

Bits 1–0: Zeros—These bits are zero, since instruction addresses are always word aligned.

8.3.2.3 PROGRAM COUNTER 2 (PC2, Register 12)

This protected special-purpose register (Figure 8-7) reports the address of certain instructions causing traps.

Figure 8-7 Program Counter 2 Register



Bits 31–2: Program Counter 2 (PC2)—This field captures the word address of an instruction as it enters the write-back stage of the processor pipeline, unless the Freeze (FZ) bit of the Current Processor Status Register is 1. If the FZ bit is 1, PC2 holds its value.

When an interrupt or trap is taken, the PC2 field contains the word address of the instruction in the write-back stage. In certain cases PC2 contains the address of the instruction causing a trap. The PC2 field is used to report the address of this instruction and has no other use in the processor.

Bits 1–0: Zeros—These bits are zero, since instruction addresses are always word aligned.

8.3.3 Taking An Interrupt Or Trap

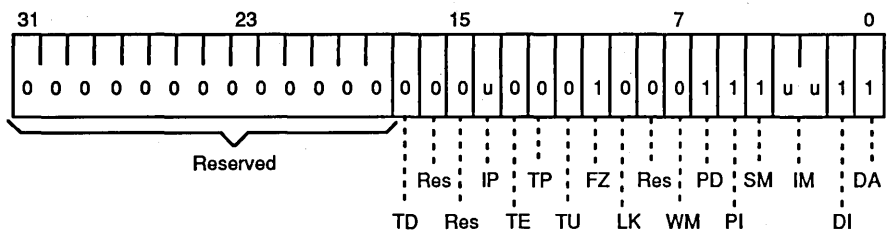
The following operations are performed in sequence by the processor when an interrupt or trap is taken:

1. Instruction execution is suspended.
2. Instruction fetching is suspended.
3. Any in-progress load or store operation is completed. Any additional operations are canceled in the case of load multiple and store multiple.
4. The contents of the Current Processor Status Register are copied into the Old Processor Status Register.
5. The Current Processor Status register is modified as shown in Figure 8-8 (the value *u* means unaffected). Note that setting the Freeze (FZ) bit freezes the Channel Address, Channel Data, Channel Control, Program Counter 0, Program Counter 1, Program Counter 2, and ALU Status Registers.
6. The address of the first instruction of the interrupt or trap handler is determined. The address is obtained by accessing a vector from instruction/data memory, using the physical address obtained from the Vector Area Base Address Register and the vector number. This access appears on the bus as a data access, and the OPT(2–0) signals indicate a word-length access.

7. An instruction fetch is initiated using the instruction address determined in step 6. At this point, normal instruction execution resumes.

Note that the processor does not explicitly save the contents of any registers when an interrupt is taken. If register saving is required, it is the responsibility of the interrupt- or trap-handling routine. For proper operation, registers must be saved before any further interrupts or traps may be taken. The FZ bit must be reset at least two instructions before interrupts or traps are re-enabled, to allow program state to be reflected properly in processor registers if an interrupt or trap is taken.

Figure 8-8 Current Processor Status After an Interrupt or Trap



8.3.4 Returning From An Interrupt Or Trap

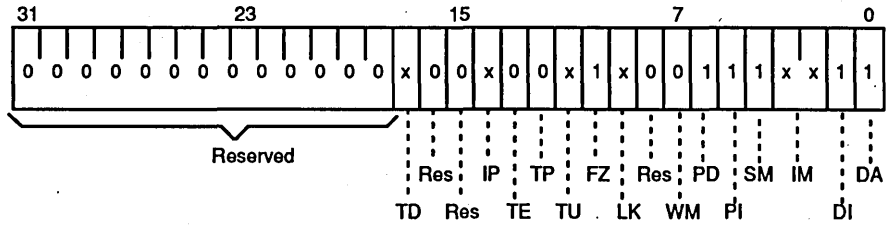
Two instructions are used to resume the execution of an interrupted program: Interrupt Return (IRET), and Interrupt Return and Invalidate (IRETINV). These instructions are identical except in one respect: the IRETINV instruction resets all Valid bits in the Instruction Cache, whereas the IRET instruction does not affect the Valid bits.

In some situations, the processor state must be set properly by software before the interrupt return is executed. The following is a list of operations normally performed in such cases:

1. The Current Processor Status is configured as shown in Figure 8-9 (the value *x* is a *don't care*). Note that setting the FZ bit freezes the registers listed below so that they may be set for the interrupt return.
2. The Old Processor Status is set to the value of the Current Processor Status for the target routine.
3. The Channel Address, Channel Data, and Channel Control registers are set to restart or resume uncompleted external accesses of the target routine.
4. The Program Counter 1 and Program Counter 0 registers are set to the addresses of the first and second instructions, respectively, to be executed in the target routine.
5. Other registers are set as required. These may include registers such as the ALU Status, Q, and so forth, depending on the particular situation. Some of these registers are unaffected by the FZ bit, so they must be set in such a manner that they are not modified unintentionally before the interrupt return.

Once the processor registers are configured properly, as described above, an interrupt return instruction (IRET or IRETINV) performs the remaining steps necessary to return to the target routine. The following operations are performed by the interrupt return instruction:

Figure 8-9 Current Processor Status Before Interrupt Return



1. Any in-progress load or store operation is completed. If a load-multiple or store-multiple sequence is in progress, the interrupt return is not executed until the sequence completes.
2. Interrupts and traps are disabled, regardless of the settings of the DA, DI, and IM fields of the Current Processor Status, for steps 3 through 10.
3. If the interrupt return instruction is an IRETINV, all Valid bits in the Instruction Cache memory are reset, except for those portions of the Instruction Cache which are locked (see Section 9.1).
4. The contents of the Old Processor Status Register are copied into the Current Processor Status Register. This normally resets the FZ bit, allowing the Program Counter 0, 1, 2, Channel Address, Data, Control, and ALU Status registers to update normally. Since certain bits of the Current Processor Status Register always are updated by the processor, this copy operation may be irrelevant for certain bits (e.g., the Interrupt Pending bit).
5. If the Contents Valid (CV) bit of the Channel Control Register is 1, and the Not Needed (NN) and Multiple Operation (ML) bits are both 0, an external access is started. This operation is based on the contents of the Channel Address, Channel Data, and Channel Control registers. The Current Processor Status Register conditions the access—as is normally the case. Note that load-multiple and store-multiple operations are not restarted at this point.
6. The address in Program Counter 1 is used to fetch an instruction. The Current Processor Status Register conditions the fetch. This step is treated as a branch in the sense that the processor searches the Instruction Cache for the target of the fetch.
7. The instruction fetched in step 6 enters the decode stage of the pipeline.
8. The address in Program Counter 0 is used to fetch an instruction. The Current Processor Status Register conditions the fetch. This step is treated as a branch in the sense that the processor searches the Instruction Cache for the target of the fetch.
9. The instruction fetched in step 6 enters the execute stage of the pipeline, and the instruction fetched in step 8 enters the decode stage.
10. If the CV bit in the Channel Control Register is a 1, the NN bit is 0, and the ML bit is 1, a load-multiple or store-multiple sequence is started, based on the contents of the Channel Address, Channel Data, and Channel Control registers.
11. Interrupts and traps are enabled per the appropriate bits in the Current Processor Status Register.
12. The processor resumes normal operation.

8.3.5 Lightweight Interrupt Processing

The registers affected by the FZ bit of the Current Processor Status Register are those which are modified by almost any usual sequence of instructions. Since the FZ bit is set by an interrupt or trap, the interrupt or trap handler is able to execute while not disturbing the state of the interrupted routine, though its execution is somewhat restricted. Thus, it is not necessary in many cases for the interrupt or trap handler to save the registers that are affected by the FZ bit. This permits the implementation of lightweight interrupt handlers that do not have all of the overhead normally associated with interrupt handlers.

The processor provides an additional benefit to lightweight interrupts if the Program Counter 0 and Program Counter 1 Registers are not modified by the interrupt or trap handler. If Program Counters 0 and 1 contain the addresses of sequential instructions when an interrupt or trap is taken, and if they are not modified before an interrupt return is executed, step 8 of the interrupt return sequence above occurs as a sequential fetch—instead of a branch—for the interrupt return. The performance impact of a sequential fetch is normally less than that of a non-sequential fetch.

Because the registers affected by the FZ bit are sometimes required for instruction execution, it is not possible for the lightweight interrupt or trap handler to execute all instructions, unless the required registers are first saved elsewhere (e.g., in one or more global registers). Most of the restrictions due to register dependencies are obvious (e.g., the Byte Pointer for byte extracts) and will not be discussed here. Other less obvious restrictions are listed below:

1. Load Multiple and Store Multiple. The Channel Address, Channel Data, and Channel Control registers are used to sequence load-multiple and store-multiple operations, so these instructions cannot be executed while the registers are frozen. However, note that other external accesses may occur; the Channel Address, Channel Data, and Channel Control registers are required only to restart an access after an exception, and the interrupt or trap handler is not expected to encounter any exceptions.
2. Loads and stores which set the Byte Pointer. If the Set Byte Pointer (SB) of a load or store instruction is 1, and the FZ bit is also 1, there is no effect on the Byte Pointer. Thus, the execution of external byte and half-word accesses using this mechanism is not possible.
3. Extended arithmetic. The Carry bit of the ALU Status Register is not updated while the FZ bit is 1.
4. Divide step instructions. The Divide Flag of the ALU Status Register is not updated when the FZ bit is 1.

If the interrupt or trap handler does not save the state of the interrupted routine, it cannot allow additional interrupts and traps. Also, the operation of the interrupt or trap handler cannot depend on any trapping instructions (e.g., Floating-Point instructions, illegal operation codes, arithmetic overflow, etc.), since these are disabled. There are certain cases, however, where traps are unavoidable; these are discussed in Section 8.6.3 and 8.6.4. Special considerations for these cases are discussed in Section 8.6.6.

8.3.6 Simulation Of Interrupts And Traps

Assert instructions may be used by a Supervisor-mode program to simulate the occurrence of various interrupts and traps defined for the processor. Only an assert instruction executed in Supervisor mode can specify a vector number between 0 and

63. If this instruction causes a trap, the effect is to create an interrupt or trap which is similar to that associated with the specified vector number.

Thus, the interrupt and trap routines defined for basic processor operation can be invoked without creating any particular hardware condition. For example, an $\overline{\text{INTR1}}$ interrupt may be simulated by an assert instruction that specifies a vector number of 17, without the activation of the $\overline{\text{INTR1}}$ signal.

8.4 $\overline{\text{WARN}}$ TRAP

The processor recognizes a special trap, caused by the activation of the $\overline{\text{WARN}}$ input, which cannot be masked. The $\overline{\text{WARN}}$ trap is intended to be used for severe system-error or deadlock conditions. It allows the processor to be placed in a known, operable state, while preserving much of its original state for error reporting and possible recovery. Therefore, it shares some features in common with the Reset mode as well as features common to other traps described in this section.

The major differences between the $\overline{\text{WARN}}$ trap and other traps are:

1. The processor does not wait for an in-progress external access to complete before taking the trap, since this access might not complete. However, the information related to any outstanding access is retained by the Channel Address, Channel Data, and Channel Control registers when the trap is taken.
2. The vector-fetch operation is not performed when the $\overline{\text{WARN}}$ trap is taken. Instead instruction fetching begins immediately at address 16 in the instruction memory. The trap handler executes directly from the instruction memory.

Note that the $\overline{\text{WARN}}$ trap may disrupt the state of the routine that is executing when it is taken, prohibiting this routine from being restarted.

8.4.1 $\overline{\text{WARN}}$ Input

An inactive-to-active transition on the $\overline{\text{WARN}}$ input causes a $\overline{\text{WARN}}$ trap to be taken by the processor. The $\overline{\text{WARN}}$ trap cannot be disabled; the processor responds to the $\overline{\text{WARN}}$ input regardless of its internal condition, unless the $\overline{\text{RESET}}$ input is also asserted. The $\overline{\text{WARN}}$ input is provided so that the system can gain control of the processor in extreme situations, such as when system power is about to be removed or when a severe non-recoverable error occurs.

The $\overline{\text{WARN}}$ input is edge-sensitive, so that an active level on the $\overline{\text{WARN}}$ input for long intervals does not cause the processor to take multiple $\overline{\text{WARN}}$ traps. However, $\overline{\text{WARN}}$ must be held active for at least 4 cycles in order to be properly recognized by the processor. The processor still takes the $\overline{\text{WARN}}$ trap if $\overline{\text{WARN}}$ is de-asserted after four cycles. Another $\overline{\text{WARN}}$ trap occurs if $\overline{\text{WARN}}$ makes another inactive-to-active transition.

The processor enters the Executing mode when the $\overline{\text{WARN}}$ input is asserted, regardless of its previous operational mode. Either seven or eight cycles after $\overline{\text{WARN}}$ is asserted (depending on internal synchronization time), the processor performs a trap-handler instruction access on the bus. This access is directed to address 16 in the instruction/data memory.

If the CNTL(1–0) inputs are 10 or 01 when the trap-handler instruction fetch completes, the processor enters the Halt or Step mode, respectively. Before the completion of this instruction fetch, the CNTL(1–0) inputs are irrelevant, except that the Load Test Instruction mode cannot be entered directly after a $\overline{\text{WARN}}$ trap is taken. If the

CNTL(1–0) inputs are 00 immediately after $\overline{\text{WARN}}$ is de-asserted (indicating entry into the Load Test Instruction mode), the effect on processor operation is unpredictable. If the CNTL(1–0) inputs are 11, the processor remains in the Executing mode.

8.5 SEQUENCING OF INTERRUPTS AND TRAPS

On every cycle, the processor decides either to execute instructions or to take an interrupt or trap. Since there are multiple sources of interrupts and traps, more than one interrupt or trap may be pending on a given cycle.

To resolve conflicts, interrupts and traps are taken according to the priority shown in Table 8-2. In this table, interrupts and traps are listed in order of decreasing priority. This section discusses the first three columns of Table 8-2. The last two columns are discussed in Section 8.6.

In Table 8-2, interrupts and traps fall into one of two categories depending on the timing of their occurrence relative to instruction execution. These categories are indicated in the third column of Table 8-2 by the labels *Inst* and *Async.* These labels have the following meaning:

1. Inst—Generated by the execution or attempted execution of an instruction.
2. Async—Generated asynchronous to and independent of the instruction being executed, although it may be a result of an instruction executed previously.

The principle for interrupt and trap sequencing is that the highest priority interrupt or trap is taken first. Other interrupts and traps remain active until they can be taken or are regenerated when they can be taken. This is accomplished, depending on the type of interrupt or trap, as follows:

1. All traps in Table 8-2 with priority 13 through 15 are regenerated by the re-execution of the causing instruction.
2. Most of the interrupts and traps of priority 4 through 12 must be held by external hardware until they are taken. The exceptions to this are listed in item 3) below.
3. The exceptions to 2 above are the Data Access Exception trap, the Timer interrupt, and the Trace trap. These are caused by bits in various registers in the processor and are held by these registers until taken or cleared. The relevant bits are: the Transaction Faulted (TF) bit of the Channel Control Register for Data Access Exception trap, the Interrupt (IN) bit of the Timer Reload Register for Timer interrupts, and the Trace Pending (TP) bit of the Current Processor Status Register for Trace traps.
4. All traps of priority 2 and 3 in Table 8-2, except for the Unaligned Access trap, are not regenerated. These traps are mutually exclusive, and are given high priority because they cannot be regenerated; they must be taken if they occur. If one of these traps occurs at the same time as a reset or $\overline{\text{WARN}}$ trap, it is not taken, and its occurrence is lost.
5. The Unaligned Access trap is regenerated internally when an external access is restarted by the Channel Address, Channel Data, and Channel Control registers. Note that this trap is not necessarily exclusive to the traps discussed in item 4) above.

The Channel Address, Channel Data, and Channel Control registers are set for a $\overline{\text{WARN}}$ trap only if an external access is in progress when the trap is taken.

Table 8-2 Interrupt and Trap Priority Table

Priority	Type of Interrupt or Trap	Inst/Async	PC1	Channel Regs
1 (Highest)	$\overline{\text{WARN}}$	Async	Next	See Note 1
2	User-Mode Data TLB Miss	Inst	Next	All
	Supervisor-Mode Data TLB Miss	Inst	Next	All
	Data MMU Protection Violation	Inst	Next	All
3	Unaligned Access	Inst	Next	All
	Out of Range	Inst	Next	N/A
	Assert Instructions	Inst	Next	N/A
	Floating-Point Instructions	Inst	Next	N/A
	Integer Multiply/Divide Instructions	Inst	Next	N/A
	EMULATE	Inst	Next	N/A
4	Data Access Exception	Async	Next	All
5	$\overline{\text{TRAP0}}$	Async	Next	Multiple
6	$\overline{\text{TRAP1}}$	Async	Next	Multiple
7	$\overline{\text{INTR0}}$	Async	Next	Multiple
8	$\overline{\text{INTR1}}$	Async	Next	Multiple
9	$\overline{\text{INTR2}}$	Async	Next	Multiple
10	$\overline{\text{INTR3}}$	Async	Next	Multiple
11	Timer	Async	Next	Multiple
12	Trace	Async	Next	Multiple
13	User-Mode Instruction TLB Miss	Inst	Curr	N/A
	Supervisor-Mode Instr. TLB Miss	Inst	Curr	N/A
	Instruction MMU Protection Violation	Inst	Curr	N/A
	Instruction Access Exception	Inst	Curr	N/A
14 (Lowest)	Illegal Opcode	Inst	Curr	N/A
	Protection Violation	Inst	Curr	N/A

Note 1: The Channel Address, Channel Data, and Channel Control registers are set for a $\overline{\text{WARN}}$ trap only if an external access is in progress when the trap is taken.

8.6 EXCEPTION REPORTING AND RESTARTING

When an instruction encounters an exceptional condition, the Program Counter 0, Program Counter 1, and Program Counter 2 registers report the relevant instruction address(es) and allow the instruction sequence to be restarted once the exceptional condition has been remedied (if possible). Similarly, when an external access encounters an exceptional condition, the Channel Address, Channel Data, and Channel Control registers report information on the access or transfer and allow it to be restarted. This section describes the interpretation and use of these registers.

The *PC1* column in Table 8-2 describes the value held in the Program Counter 1 Register (PC1) when the interrupt or trap is taken. For traps in the *Inst* category, PC1 contains either the address of the instruction causing the trap, indicated by *Curr,* or the address of the instruction following the instruction causing the trap, indicated by *Next.*

For interrupts and traps in the *Async* category, PC1 contains the address of the first instruction which was not executed due to the taking of the interrupt or trap. This is the next instruction to be executed upon interrupt return, as indicated by *Next* in the PC1 column.

8.6.1 Instruction Exceptions

For traps caused by the execution of an instruction (e.g., the Out of Range trap), the Program Counter 2 Register contains the address of the instruction causing the trap. In all of these cases, PC1 is in the *Next* category.

The traps associated with instruction fetches (i.e., those of priority 13) occur only if the processor attempts the execution of the associated instruction. An exception may be detected during an instruction prefetch, but the associated trap does not occur if the processor branches before it attempts to execute the invalid instruction. This prevents spurious instruction exceptions.

8.6.2 Restarting Faulting External Accesses

In a demand-paged system environment, virtual pages and their associated virtual-to-physical mappings are made available to programs on demand. In other words, the memory-management routines generally execute only when a given page or mapping is needed by a program. This need is signaled by a page fault trap caused by a program access (normally, the page fault occurs during a TLB reload).

Since the page fault trap is part of normal system operation, and does not represent an error, the access that causes the trap must be restarted—once the trapping condition is remedied—in a manner that cannot be detected by the program causing the trap.

Additionally, in the Am29030 and Am29035 microprocessors, the TLB reload mechanism relies on the ability to restart an access that causes a TLB miss trap. This restart also must be accomplished in a manner that cannot be detected by the trapping program.

The Am29030 and Am29035 microprocessors overlap external accesses with the execution of instructions. Thus, traps caused by accesses are imprecise: the address of the instruction that initiated the access cannot be determined by the trap handler. Since the address of the initiating instruction is unknown, the access cannot be restarted by re-executing this instruction. Even if the address could be determined, the instruction might not be restartable, since an instruction executed before the trap

occurred, but after the access began, may have altered the conditions of the access, such as by altering the address source register.

In order to provide for the restarting of loads and stores that cause exceptions, the processor saves all information required to restart these accesses in the Channel Address, Channel Data, and Channel Control registers. The Contents Valid (CV) and Not Needed (NN) bits in the Channel Control Register indicate that the information contained in these registers represents an access that must be restarted. The CV bit indicates that the access did not complete, and the NN bit indicates whether or not the data from the access is required by the processor.

Note that since instruction execution is overlapped with external accesses, an instruction that executes after a load may alter the destination register for the load. If a trap occurs in this situation, the access information in the Channel Address, Data, and Control registers is correct, but the load cannot be restarted, because it will destroy the new value in the destination register. The NN bit provides correct operation in this case.

When an interrupt or trap is taken, the handling routine has access to the Channel Address, Data, and Control registers; the contents of these registers may contain information relevant to an incomplete access and can be preserved for restarting this access. Since these registers are frozen (due to the FZ bit of the Current Processor Status), they are not available to monitor any external accesses in the interrupt or trap handler until their contents are saved and the FZ bit is reset.

Note that the exception handler for the Data Access Exception trap must clear the Transaction Faulted (TF) bit in the Channel Control Register. Failure to clear the TF bit results in the processor taking the trap again, once the exception handler returns, causing an infinite series of traps.

The processor restarts an access, using the Channel Address, Channel Data, and Channel Control registers, upon an interrupt return (IRET or IRETINV). The access is initiated if the CV bit of the Channel Control Register is 1 and the NN bit is 0. The restart cannot be detected in the logical operation of the restarted routine, although the timing of execution is altered.

The mechanism used to restart faulting accesses has the additional benefit of allowing a fast interrupt-response time when the processor is performing a load-multiple or store-multiple operation. An interrupted load-multiple or store-multiple is restarted as if it had faulted. In this case, the operation resumes from the point of interruption, not from the beginning of the sequence.

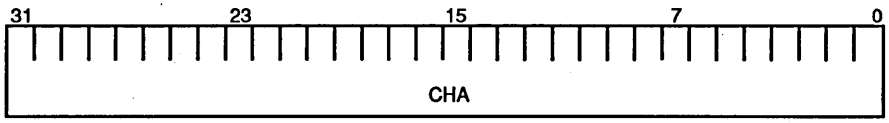
8.6.2.1 CHANNEL ADDRESS (CHA, Register 4)

This protected special-purpose register (Figure 8-10) is used to report exceptions during external accesses. It also is used to restart interrupted load-multiple and store-multiple operations and to restart other external accesses when possible (e.g., after TLB misses are serviced).

The Channel Address Register is updated on the execution of every load or store instruction and on every load or store in a load-multiple or store-multiple sequence, except when the Freeze (FZ) bit in the Current Processor Status Register is 1.

Bits 31–0: Channel Address (CHA)—This field contains the address of the current bus access (if the FZ bit of the Current Processor Status Register is 0). For external data accesses, the address is virtual if address translation was enabled for the access, or physical if translation was disabled.

Figure 8-10 Channel Address Register

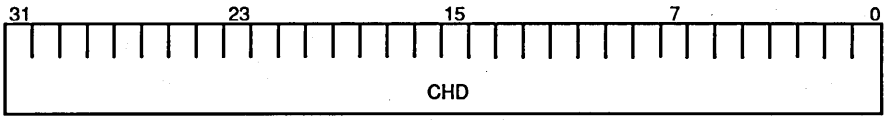


8.6.2.2 CHANNEL DATA (CHD, Register 5)

This protected special-purpose register (Figure 8-11) is used to report exceptions during external accesses. It also is used to restart the first store of an interrupted store-multiple operation and to restart other external accesses when possible (e.g., after TLB misses are serviced).

The Channel Data Register is updated on the execution of every load or store instruction and on every load or store in a load-multiple or store-multiple sequence, except when the Freeze (FZ) bit in the Current Processor Status Register is 1. When the Channel Data Register is updated for a load operation, the resulting value is unpredictable.

Figure 8-11 Channel Data Register



Bits 31–0: Channel Data (CHD)—This field contains the data (if any) associated with the current bus access (if the FZ bit of the Current Processor Status Register is 0). If the current bus access is not a store, the value of this field is irrelevant.

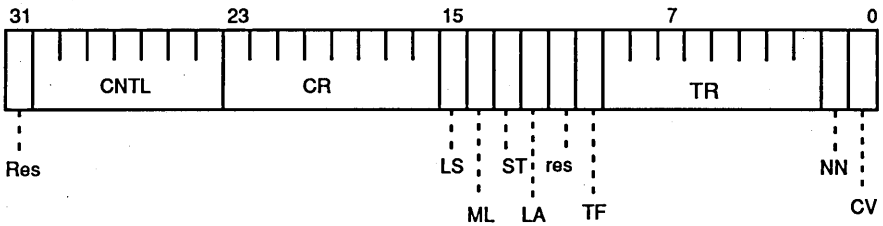
8.6.2.3 CHANNEL CONTROL (CHC, Register 6)

This protected special-purpose register (Figure 8-12) is used to report exceptions during external accesses. It also is used to restart interrupted load-multiple and store-multiple operations and to restart other external accesses when possible (e.g., after TLB misses are serviced).

The Channel Control Register is updated on the execution of every load or store instruction and on every load or store in a load-multiple or store-multiple sequence, except when the Freeze (FZ) bit in the Current Processor Status Register is 1.

Bits 31–24:—These bits are a direct copy of bits 23–16 from the load or store instruction that started the current bus access (see Section 3.3).

Figure 8-12 Channel Control Register



Bits 23–16: Load/Store Count Remaining (CR)—The CR field indicates the remaining number of transfers for a load-multiple or store-multiple operation that encountered an exception or was interrupted before completion. This number is zero-based; for example, a value of 28 in this field indicates that 29 transfers remain to be completed.

Bit 15: Load/Store (LS)—The LS bit is 0 if the bus access is a store operation and is 1 if the bus access is a load operation.

Bit 14: Multiple Operation (ML)—The ML bit is 1 if the current bus access is a partially-complete load-multiple or store-multiple operation; otherwise it is 0.

Bit 13: Set (ST)—The ST bit is 1 if the current bus access is for a Load and Set instruction; otherwise it is 0.

Bit 12: Lock Active (LA)—The LA bit is 1 if the current bus access is for a Load and Lock or Store and Lock instruction; otherwise it is 0. Note that this bit is not set as the result of the Lock (LK) bit in the Current Processor Status Register.

Bit 11: Reserved.

Bit 10: Transaction Faulted (TF)—The TF bit indicates that the current bus access did not complete due to some exceptional circumstance. This bit is set only for exceptions reported via the $\overline{ERR}$ input, and it causes a Data Access Exception trap to occur when it is 1.

The TF bit allows the proper sequencing of externally reported errors that get preempted by higher-priority traps (see Section 8.6). It is reset by software that handles the resulting trap.

Bits 9–2: Target Register (TR)—The TR field indicates the absolute register number of the data operand for the current access (either a load target or store data source). Since the register number in this field is absolute, it reflects the Stack-Pointer addition when the indicated register is a local register.

Bit 1: Not Needed (NN)—The NN bit indicates that, even though the Channel Address, Channel Data, and Channel Control registers contain a valid representation of an incomplete load operation, the data requested is not needed. This situation arises when a load instruction is overlapped with an instruction that writes the load target register.

Bit 0: Contents Valid (CV)—The CV bit indicates that the contents of the Channel Address, Channel Data, and Channel Control registers are valid.

8.6.3 Integer Exceptions

Some integer add and subtract instructions—ADDS, ADDU, ADDCS, ADDCU, SUBS, SUBU, SUBCS, SUBCU, SUBRS, SUBRU, SUBRCS, and SUBRCU—cause an Out of Range trap upon overflow or underflow of a 32-bit signed or unsigned result, depending on the instruction.

Two integer multiply instructions—MULTIPLY and MULTIPLU—cause an Out of Range trap upon overflow of a 32-bit signed or unsigned result, respectively, if the MO bit of the Integer Environment Register is 0. If the MO bit is 1, these multiply instructions cannot cause an Out of Range trap. Since the processor does not contain hardware to directly support these instructions, the Out of Range trap must be generated by software support.

Two integer divide instructions—DIVIDE and DIVIDU—take the Out of Range trap upon overflow of a 32-bit signed or unsigned result, respectively, if the DO bit of the Integer Environment Register is 0. If the DO bit is 1, the divide instructions cannot

cause an Out of Range trap unless the divisor is zero. If the divisor is zero, an Out of Range trap always occurs, regardless of the DO bit.

For the MULTIPLY, MULTIPLU, DIVIDE, and DIVIDU instructions, the destination register or registers are unchanged if an Out of Range trap is taken.

8.6.4 Floating-Point Exceptions

A Floating-Point Exception trap occurs when an exception is detected during a floating-point operation and the exception is not masked by the corresponding bit of the Floating-Point Mask Register. In this context, a floating-point operation is defined as any operation that accepts a floating-point number as a source operand, that produces a floating-point result, or both. Thus, for example, the CONVERT instruction may create an exception while attempting to convert a floating-point value to an integer value or vice versa.

In addition to the operations described in Section 8.3.3, the following operations are performed when a Floating-Point Exception trap is taken:

1. The status of the trapping operation is written into the trap status bits of the Floating-Point Status Register. The written status bits do not depend on the values of the corresponding mask bits in the Floating-Point Environment Register.
2. The destination register or registers are left unchanged.

8.6.5 Correcting Out-of-Range Results

Some Arithmetic instructions cause an Out of Range trap if the arithmetic operation causes an overflow or underflow. When an Out of Range trap occurs, the result of the operation—though incorrect—is written into the destination register. Furthermore, the Program Counter 2 Register contains the address of the trapping instruction, and the ALU Status Register contains an indication of the cause of the trap. It is possible, if required, for the trap handler to use this information to form the correct result.

The ALU Status indicates the cause of the Out of Range trap, based on the operation performed, as follows:

1. Signed overflow. If the Out of Range trap is caused by signed, two's-complement overflow (this can occur for both signed adds and subtracts), the V bit is 1.
2. Unsigned overflow. If the Out of Range trap is caused by unsigned overflow (this can occur only for unsigned adds), the C bit is 1.
3. Unsigned underflow. If the Out of Range trap is caused by unsigned underflow (this can occur only for unsigned subtracts), the C bit is 0.

The multiply instructions MULTIPLY and MULTIPLU can cause an Out of Range trap if the MO bit of the Integer Environment Register is 0 and the operation overflows. However, these instructions do not set the ALU Status Register. This exception is detected by reading the trapping instruction, whose address is in the PC2 Register.

8.6.6 Exceptions During Interrupt and Trap Handling

In most cases, interrupt and trap handling routines are executed with the DA bit in the Current Processor Status having a value of 1. It is normally assumed that these routines do not create many of the exceptions possible in most other processor routines.

If these assumptions are not valid for a particular interrupt or trap handler, it is important that the handler save the state of the processor and reset the FZ bit of the

Current Processor Status, so that the handler itself may be restarted properly. This must be accomplished before any interrupts or traps can be taken. In this case, the state (or the state of some other process) must be restored before an interrupt return is executed.

8.7 TIMER FACILITY

The processor has a built-in Timer Facility that can be configured to cause periodic interrupts. The Timer Facility consists of two special-purpose registers—the Timer Counter and the Timer Reload registers—that are accessible only to Supervisor-mode programs. Also, the Current Processor Status Register contains a control bit as part of the timer facility. These registers implement timing functions independent of program execution.

8.7.1 Timer Facility Operation

The Timer Counter Register has a 24-bit Timer Count Value (TCV) field that decrements by one on every processor cycle. If the TCV field decrements to zero, it is written with the Timer Reload Value (TRV) field of the Timer Reload Register on the next cycle; the Interrupt (IN) bit of the Timer Reload register is set at the same time. Reloading the TCV field by the TRV field maintains the accuracy of the Timer Facility.

The Timer Reload Register contains the 24-bit TRV field and the control bits Overflow (OV), Interrupt (IN), and Interrupt Enable (IE). The TCV field and IN bit were just described. If the IN bit is 1 and the IE bit also 1, a Timer interrupt occurs. If the IN bit is 1 when the TCV field decrements to zero, the OV bit is also set. The OV bit indicates that a Timer interrupt may have occurred before a previous interrupt was serviced.

The Current Processor Status Register contains the Timer Disable (TD) control bit. If the TD bit is 1, Timer interrupts are disabled. The TD bit and the IE bit have equivalent functions; the TD bit is provided so that the timer may be diabled without having to perform a non-atomic read-modify-write operation on the Timer Reload Register. There is a possiblilty that the TCV might decrement to zero and set the IN bit as the modified value is written back to the Timer Reload Register, causing a Timer interrupt to be missed.

8.7.2 Timer Facility Initialization

To initialize the Timer Facility, the following steps should be taken in the specified order (it is assumed that Timer interrupts are disabled by the DA bit of the Current Processor Status Register or the TD bit of the Current Processor Status Register during the following steps):

1. Set the TCV field with the desired interval count for the first timing interval. Note that this interval must be sufficiently large to allow the execution of the next step before the TCV field decrements to zero (this normally is the case).
2. Set the TRV field with the desired interval count for the second timing interval. The OV and IN bits are reset and the IE bit is set as desired. Note that the second timing interval may be equivalent to the first timing interval.

8.7.3 Handling Timer Interrupts

The following is a suggested list of actions to be taken to handle a Timer interrupt:

1. Read the Timer Reload register into a general-purpose register.
2. Reset the IN bit in the general-purpose register.
3. Set the TRV field in the general-purpose register to the desired value for the next timing interval. Note that, at this time, the Timer Counter is timing the current interval. Also, this step may be omitted, if all intervals are equivalent.
4. Write the contents of the general-purpose register back into the Timer Reload register.
5. Test the general-purpose-register copy of the OV bit and, if it is set, report the error as appropriate.
6. Perform any system operations required for the Timer interrupt.
7. Execute an interrupt return.

8.7.4 Timer Facility Uses

Since the Timer Facility has a resolution of a single processor cycle, it may be used to perform precise timing of system events. For example, it may be used to determine an exact measurement of the number of cycles between two events in the system or to perform precise time-critical control functions. Note that the Timer interrupt is enabled and disabled separately from other processor interrupts, so that its priority can be specified.

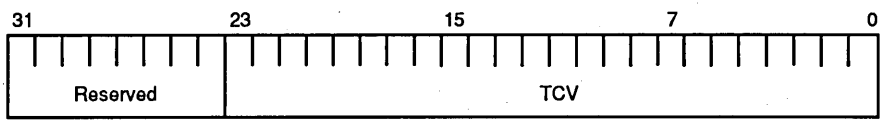
The Timer Facility can be used to generate time intervals for collecting virtual page usage information (see Section 7.5.3). For example, if memory management relies on a working-set page-replacement algorithm, the Timer Facility can establish the working-set window.

The Timer Facility can be shared among multiple processes. This sharing is accomplished by the implementation of a queue for timer events, which are sorted in order of increasing event time. On each occurrence of a Timer interrupt, the TRV field is set for the interval between the next two events in the queue, while the Timer Counter Register is counting the current interval (because of a previous setting of the TRV field). The event at the beginning of the queue identifies other system actions to be taken for the Timer interrupt. This event is removed from the queue after the appropriate actions are taken.

8.7.5 Timer Counter (TMC, Register 8)

This protected special-purpose register (Figure 8-13) contains the counter for the Timer Facility.

Figure 8-13 Timer Counter Register



Bits 31–24: Reserved.

Bits 23–0: Timer Count Value (TCV)—The 24-bit TCV field decrements by one on each processor clock. When the TCV field decrements to zero, it is reloaded with the

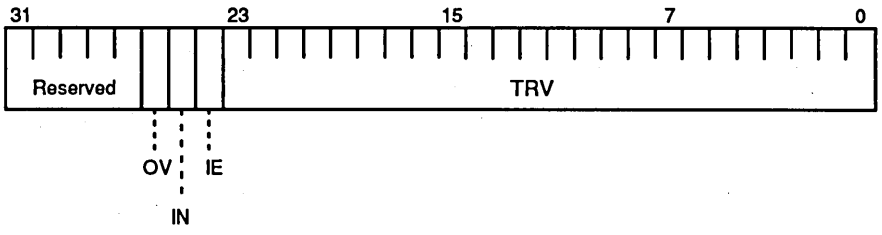
content of the Timer Reload Value field in the Timer Reload Register. At this time, the Interrupt bit in the Timer Reload Register is set.

The TCV field is zero-based with respect to the Timer interrupt interval; for example, a value of 28 in the TCV field causes the IN bit to be set in the 29th subsequent processor cycle. The reason for this is that the TCV field is zero for a complete cycle before the IN bit is set.

8.7.6 Timer Reload (TMR, Register 9)

This protected special-purpose register (Figure 8-14) maintains synchronization of the Timer Counter Register, enables Timer interrupts, and maintains Timer Facility status information.

Figure 8-14 Timer Reload Register



Bits 31–27: Reserved.

Bit 26: Overflow (OV)—The OV bit indicates that a Timer interrupt occurred before a previous Timer interrupt was serviced. It is set if the Interrupt (IN) bit is1 when the Timer Count Value (TCV) field of the Timer Counter Register decrements to zero. In this case, a Timer interrupt caused by the IN bit has not been serviced when another interrupt is created.

Bit 25: Interrupt (IN)—The IN bit is set whenever the TCV field decrements to zero. If this bit is 1 and the IE bit is also 1, a Timer interrupt occurs. Note that the IN bit is set when the TCV field decrements to zero, regardless of the value of the IE bit. The IN bit is reset by software that handles the Timer interrupt.

Bit 24: Interrupt Enable (IE)—When the IE bit is 1, the Timer interrupt is enabled, and the Timer interrupt occurs whenever the IN bit is 1. When this bit is 0, the Timer interrupt is disabled. Note that the Timer interrupt may be disabled by the DA bit of the Current Processor Status Register regardless of the value of the IE bit.

Bits 23–0: Timer Reload Value (TRV)—The value of this field is written into the Timer Count Value (TCV) field of the Timer Counter Register when the TCV field decrements to zero.

INSTRUCTION CACHE OPERATION

This chapter details the operation of the instruction cache. It presents the cache organization and the formats of the entries. It also describes the special-purpose registers used to read and write the contents of the cache. Finally, this chapter describes the operation of the cache and the operation of the instruction prefetch buffer that supplies instructions to the cache and the processor from external memory.

9.1 INSTRUCTION CACHE OVERVIEW

The instruction cache stores the instructions most recently referenced by the processor. The Am29030 and Am29035 instruction caches are oriented around a unit of cache storage called a block. For the Am29030 and Am29035 microprocessors, each block contains four instructions and an address tag/status word, as shown in Figure 9-1. The Am29030 microprocessor has an 8K byte, two-way-set-associative instruction cache containing 512 blocks. The Am29035 microprocessor has a 4K byte, direct-mapped instruction cache containing 256 blocks. The organization of the instruction cache contained in the Am29030 processor is shown in Figure 9-2.

The instruction cache always stores complete cache blocks, so a cache block is never partially valid. The first instruction in a cache block is always aligned on a quad-word boundary in external memory.

The instruction cache can be addressed either by physical addresses if address translation is disabled or by virtual addresses if address translation is enabled. The instruction cache differentiates between virtual and physical addresses; however, it does not store information related to the Process Identifier. For this reason, the instruction cache may not correctly differentiate two identical virtual addresses in different process spaces. The instruction cache differentiates only between identical physical and virtual addresses.

The instruction cache is controlled by two fields of the Configuration Register. The instruction cache is enabled and disabled by the Instruction Cache Disable (ID) bit. If the ID bit is 0, enabling the instruction cache, instructions may be issued from the instruction cache to satisfy an instruction request from the processor (if the instruction is contained in the cache). If the ID bit is 1, disabling the cache, instruction requests are not satisfied by the instruction cache.

Figure 9-1 Instruction Cache Block Organization

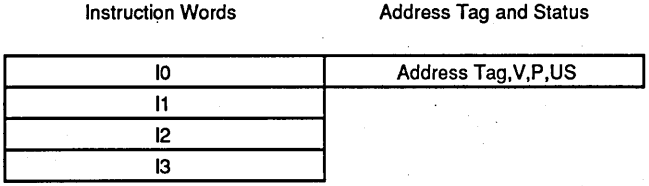
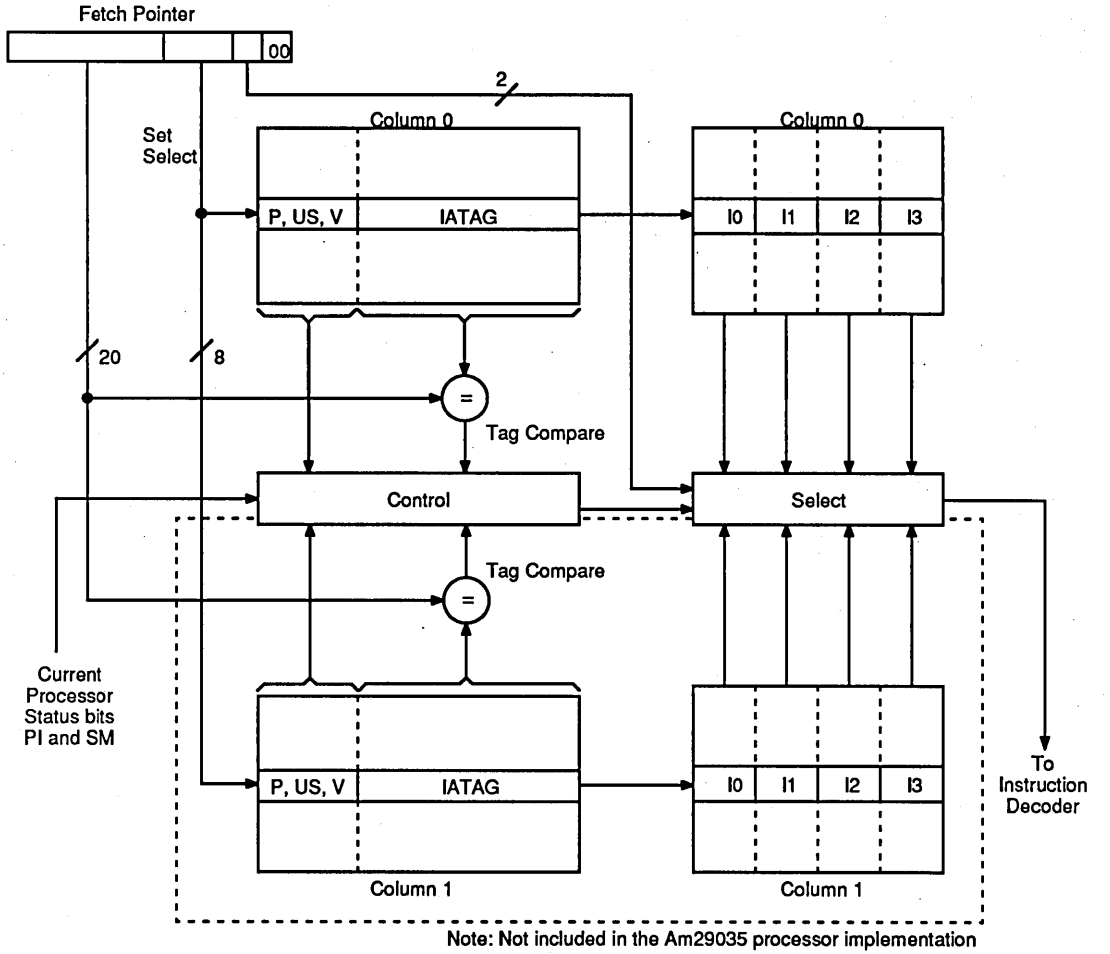


Figure 9-2 Instruction Cache Organization



The instruction cache is locked by the Instruction Cache Lock (IL) field of the Configuration Register. The value contained in the IL field determines which portion of the cache is locked. Locking a block prevents replacement of the block if it is valid; however, an invalid block may be replaced. In the Am29030 microprocessor, the IL field has the effect of locking the entire cache, locking only the blocks contained in column 0, or not locking the cache and allowing normal replacement. In the Am29035 microprocessor, the IL field value has the effect of either locking the entire cache or allowing normal replacement. Cache invalidation is affected by the ID and IL fields as discussed in Section 10.2.

9.2 ACCESSING CACHE FIELDS

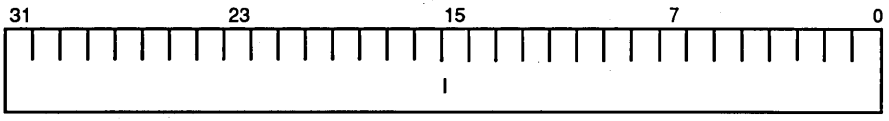
The processor allows software to read and write the instruction cache for testing and for programming purposes such as preloading. Reading and writing are performed via two special-purpose registers: the Cache Interface Register and the Cache Data Register. The Cache Interface Register provides cache addressing and control, and

the Cache Data Register provides the data for transfers. This section describes the cache fields accessed via the Cache Interface Register and the Cache Data Register and then describes these registers.

9.2.1 Instruction Words

Each block in the instruction cache contains four instruction words. Figure 9-3 shows the format of an instruction word as it appears in the Cache Data Register before a write or after a read.

Figure 9-3 Instruction Words in the Cache Data Register

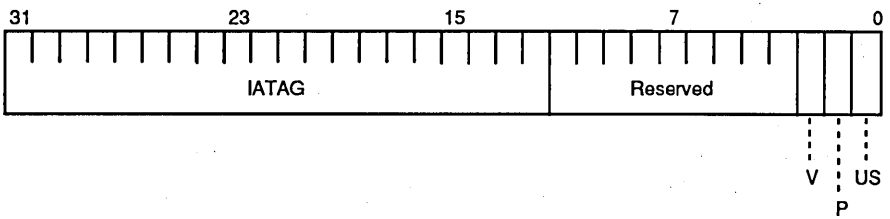


Bits 31–0: Instruction (I)—This is the 32-bit instruction that is read from or written into the instruction cache.

9.2.2 Address Tag and Status Information

Each cache block contains one address tag and status word. Figure 9-4 shows the format of a tag/status word as it appears in the Cache Data Register before a write or after a read.

Figure 9-4 Address Tag and Status Information in the Cache Data Register



Bits 31–12: Instruction Address Tag (IATAG)—The IATAG field specifies which address in instruction/data memory is mapped by the cache block.

Bit 11–3: Reserved.

Bit 2: Valid (V)—If the V bit is 1, the cache block contains valid information and a valid mapping to external memory. If the V bit is 0, the cache block does not contain a valid mapping. The V bits in all cache blocks can be cleared in a single processor cycle by a processor reset or by the execution of the instructions INV or IRETINV.

Bit 1: Physical Address (P)—The P bit reflects the state of the PI bit in the Current Processor Status Register at the time the cache block was fetched and validated. If the P bit is 0, the cache block contains instructions from a virtual address space. If the P bit is 1, the cache block contains instructions from a physical address space.

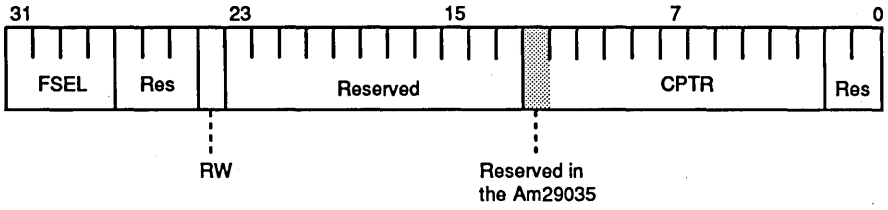
Bit 0: User or Supervisor Block (US)—The US bit reflects the state of the SM bit in the Current Processor Status Register at the time the cache block was fetched and validated. If the US bit is 1, the cache block contains instructions from a Supervisor-

mode program. If the US bit is 0, the cache block contains instructions from a User-mode program.

9.2.3 Cache Interface Register (CIR, Register 29)

This protected special-purpose register (Figure 9-5) allows software to address the cache and provides control information for cache access. The cache is accessed as the result of a write to the Cache Interface Register.

Figure 9-5 Cache Interface Register



Bits 31–28: Cache Field Select (FSEL)—The FSEL field selects the cache field that is read or written when the Cache Interface Register is written, as follows:

FSEL Value	Cache Field Selected/Bits of Cache Data Register
0000	instruction word/31–0
0001	instruction address tag/31–12, status/2–0
0010–1111	reserved

Bits 27–25: Reserved.

Bit 24: Read/Write (RW)—If the RW bit is 0, the cache field selected by the FSEL field is read into the Cache Data Register when the Cache Interface Register is written. If the RW bit is 1, the contents of the Cache Data Register are written into the cache field selected by the FSEL field when the Cache Interface Register is written.

Bits 23–13: Reserved.

Bits 12–2: Cache Pointer (CPTR)—The CPTR field selects the instruction word or address tag/status word within the cache to be read or written. The CPTR field selects the particular block containing the instruction or address/tag status word. If the FSEL field selects an address tag/status word, CPTR bits 12–4 address the appropriate address tag/status word in the cache. If the FSEL field selects an instruction word, CPTR bits 12–2 address the appropriate instruction word in the cache. In the Am29035 microprocessor, CPTR bit 12 is reserved and should be 0.

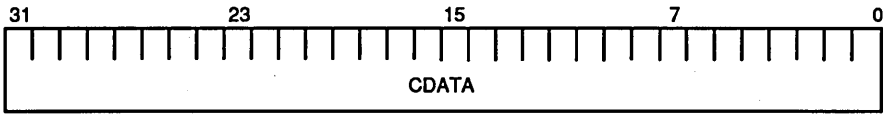
Bits 1–0: Reserved.

9.2.4 Cache Data Register (CDR, Register 30)

This protected special-purpose register (Figure 9-6) transfers data to or from the instruction cache. When the Cache Interface Register is written and the RW bit of the Cache Interface Register is 1, the contents of the Cache Data Register are written into the appropriate cache word. If the RW bit is a 0, the contents of the selected cache word are transferred to the Cache Data Register. The contents of the Cache Data

Register do not survive across a cache write. The Cache Data Register must be written with data each time a write is to be performed.

Figure 9-6 Cache Data Register



Bits 31–0: Cache Data (CDATA)—When the RW bit is 1, the CDATA field is written into the specified cache word. When the RW bit is 0, the specified cache word is written into the CDATA field.

9.3 CACHE HITS AND MISSES

A *cache hit* occurs when the instruction cache contains a valid instruction that corresponds to an instruction fetch, and the cache is able to satisfy the fetch. A *cache miss* occurs when the instruction cache does not contain a valid mapping and the external memory must satisfy the fetch.

The address for the current instruction fetch is contained in the processor's program counter (PC). The PC is used to access the cache and tag arrays each cycle. In the Am29030 processor, bits 11–4 of the PC are used to address both cache columns. In the Am29035 processor, bits 11–4 of the PC are used to address the single cache column. Bits 31–12 of the PC are used to compare against the IATAG field of the appropriate set.

If the conditions for a cache hit are met for one of the columns (or the single column in the Am29035 processor), the instruction fetch is satisfied by the instruction cache. If the conditions are not met by either column (or the single column in the Am29035 processor), a cache miss occurs and the instruction fetch is satisfied by an external memory access. The conditions for a cache hit are:

1. Bits 31–12 of the PC match the address tag for one of the tags in the set.
2. The valid status bit is set for the block containing the matching tag.
3. The status bits P and US match the Current Processor Status Register bits PI and SM, respectively, for the block containing the matching tag.
4. The ID bit of the Configuration Register is 0.

In the Am29030 microprocessor, if the above conditions are met by both entries of the set (as can happen as a result of software setting the cache entries), the effect on the processor is unpredictable.

9.4 EXTERNAL FETCHING AND CACHE RELOAD

When a cache miss occurs, the processor attempts to place the missing block of instructions into the cache by initiating an instruction fetch from the external instruction/data memory. This is called *cache reloading*. If the cache is disabled, the missing instructions are not placed into the cache, since the processor does not update a disabled cache. Similarly, the processor does not replace a valid block in a locked column.

The processor takes advantage of fetch-ahead logic to perform cache reloads at the earliest possible opportunity after a cache miss occurs, while minimizing the number of unnecessary instruction fetch requests.

9.4.1 Cache Replacement

When a miss is detected, a candidate block normally is selected for replacement, and the reloaded instructions are placed into this block. Replacement operates as follows:

1. If one of the blocks accessed during the cache search is invalid, this invalid block is selected for replacement. If both of the columns contain invalid blocks, the block in column 0 is selected.
2. If both blocks are valid and neither is locked, the replaced block is chosen at random.
3. On the Am29030 processor, if the block in column 0 is locked and valid, the block in column 1 is selected.
4. If the entire cache is locked, and the blocks in all columns are valid, the cache does not replace either block, and the instruction fetch is satisfied from the external instruction/data memory without updating the cache.

9.4.2 Overview of External Instruction Fetching

All external instruction fetches performed by the processor are oriented around a cache block. The processor always fetches a complete block of instructions—the external fetch always begins on a cache-block boundary and ends on a cache-block boundary. The first instruction in a cache block always corresponds to an address that is quad-word aligned in the external memory. This fetching behavior applies even for instruction fetches that occur because the cache is disabled or locked. However, the processor does not wait until the entire cache block is reloaded before it issues an instruction to the decoder. As soon as an instruction required by the processor is received from external instruction/data memory, it and other subsequent instructions are issued to the decoder while they are written into the cache.

If the processor pipeline stalls during instruction fetching, the rest of the instructions that are received are placed into the instruction prefetch buffer. These instructions remain in the prefetch buffer until the processor pipeline resumes execution, and then the instructions are issued to the decoder. Since the processor always fetches instructions in complete cache blocks, the prefetch buffer typically contains the required number of instructions to complete the cache reload.

The processor must successfully fetch an entire cache block, with no errors, before it sets the block valid bit. If an error occurs for an instruction that the processor requires, an error indication is sent to the decoder with the instruction, causing an Instruction Access Exception trap. If an error occurs for an instruction that the processor does not require because it is filling the remainder of the cache block after a taken branch, the valid bit for the block is not set. This causes the processor to refetch the block later if the instruction is required. If an error occurs for an instruction that the processor does not require because it is filling the cache ahead of a branch target, the processor treats the error as if the instruction were required, and an error indication is sent to the decoder. The instruction fetch is cancelled in this case, so the processor will not receive the instruction of interest, and thus the error must be reported to the processor.

If a taken branch, a load, or a store is executed during reload, the reload is completed and the cache block validated before the branch is taken or the load or store is executed.

9.4.3 The Instruction Fetch Pointer

The processor maintains an instruction fetch pointer in the bus interface logic. The instruction fetch pointer always contains the physical address of the next sequential instruction block in the processor's current instruction stream. This pointer is used in conjunction with a fetch-ahead adder to reduce the latency for cache misses that occur while executing sequentially through programs. The goal of this hardware is to allow the processor to drive the address bus as soon as one cycle after a cache miss is detected.

During the execution of a branch instruction, the fetch pointer is set to the first instruction of the instruction block that sequentially follows the target instruction block. If this operation does not cross a 1K byte address boundary, the instruction fetch pointer is valid and contains the physical address of the first instruction of the next block beyond the target block.

Between branches, the instruction fetch pointer is updated every time the PC enters a new instruction block, so that the fetch pointer always points to the next sequential block. The fetch pointer remains valid unless a 1K byte page boundary is crossed.

If a cache miss occurs during sequential instruction fetching, and the instruction fetch pointer is valid, the reload penalty may be reduced to two cycles as explained in the next section.

9.4.4 Cache Misses During Sequential Instruction Fetching

If the processor detects a cache miss while fetching sequential instructions, the processor performs a demand fetch to obtain the missing instructions from external memory. If the instruction fetch pointer is valid, the demand fetch is a fast demand fetch. If the instruction fetch pointer is not valid, the demand fetch is a full demand fetch.

In the case of a fast demand fetch, the processor drives the instruction fetch pointer on the address bus to initiate the instruction fetch, reducing the amount of time taken to perform the fetch.

In the case of a full demand fetch, the PC must be transferred to the bus before the instruction fetch can begin. Since this transfer uses resources in the processor that are normally used for instruction execution, the transfer does not take place until all of the instructions in the current block have been completed. For this reason, a full demand fetch takes more time than a fast demand fetch.

9.5 INSTRUCTION PREFETCHING

This section discusses issues relating to instruction prefetching.

9.5.1 Operation During Prefetching

While servicing a cache miss, the processor checks for the presence of the next sequential block in the instruction cache. If the next block is not in the cache, the processor continues the instruction fetch for the next block as soon as all of the instruction requests for the current block have been performed, unless a branch is taken which causes the processor to no longer need the next block. Checking for the next block while sequentially fetching reduces any unnecessary penalties by

permitting burst-mode memory accesses to continue while traversing sequential cache blocks.

If the next sequential block is not in the cache, the processor may initiate the fetch for this block, but the processor does not allocate the block until it is certain that the next block is required. This avoids needless invalidation of cache blocks that might otherwise be required. If an instruction fetch is started for an instruction block that is not needed, the instructions returned from this instruction fetch are discarded. The processor caches only instruction blocks in which at least one of the instructions is executed.

9.5.2 The Role of the Prefetch Buffer

Since instructions are requested externally in advance of execution, based on a predicted need, it is possible that a prefetched instruction is not required immediately for execution when the prefetch completes. To accommodate this possibility, the processor contains a five-word Instruction Prefetch Buffer (IPB). The IPB is a circularly addressed buffer which acts as a First-In/First-Out (FIFO) queue for instructions.

Instructions are stored in the IPB as they are returned from the external instruction memory. An instruction is held in the IPB until it is required for execution. When required, the instruction is issued to the decoder and written into the instruction cache (writing into the cache only occurs if a cache block has been allocated; that is if the cache is not locked or disabled). The IPB location is then freed to receive a subsequent instruction.

The primary purpose of the prefetch buffer is to decouple instruction fetching from instruction decoding. Decoupling allows the processor to safely hold the pipeline while permitting instruction fetching to continue to the end of a cache block. For example, a load or store must wait until the reload of an instruction block is complete. If the instruction following the load has a dependency on the result of the load, the processor enters Pipeline Hold Mode (see Section 5.2). The remaining instructions returned to the processor are queued in the prefetch buffer. When the reload of the instruction block completes, the bus performs the load. When the load is complete, the processor exits the Pipeline Hold Mode. The processor issues the remainder of the reloaded block to the instruction decoder from the prefetch buffer.

9.5.3 Terminating Instruction Prefetching Because of a Cache Hit

If the processor detects a cache hit in the next sequentially-addressed block during prefetching, the hit is detected early enough to stop all external fetches for the next block. The processor completes any reload in progress before resuming instruction fetching from the cache. After the external fetching is terminated, the instruction fetch pointer remains valid and can be used if the processor needs to perform a fast demand fetch.

9.5.4 Terminating Instruction Prefetching Because of a Branch

When a branch is taken during prefetching, there are cases where there is not enough time to stop external fetching before the processor begins prefetching the next sequential block. Thus, some bus capacity is taken for unnecessary fetches beyond the branch, and these extra instructions are discarded.

If the external fetch of the current block is not complete by the end of the execute stage of the branch, the processor enters the Pipeline Hold Mode until the reload is

complete. During this time, the instructions returned to the processor are written into the cache until the block is complete.

If an external fetch is required for the target of a branch, this fetch is initiated immediately after the last instruction is received for the current block.

9.5.5 Collisions Between Instruction Fetching and Loads or Stores

Since the processor is capable of decoding instructions while they are reloaded, it is possible for a load or a store to be executed while instruction fetching is in progress. In this case, the processor completes the fetch of the current block before performing the load or store. The rationale for this is that the instruction fetch is probably being performed with a burst-mode bus access, and it is probably more efficient to continue the burst-mode access until the current block is reloaded.

A load or store instruction is allowed to complete execution while it is waiting on a reload, and therefore the external load/store access begins immediately after the instruction block has been fetched.

Once the load or store is complete, the processor can resume external instruction fetching. This is triggered by the normal mechanisms used to detect cache misses and to start external fetches.

If a load or store is the delay instruction of a branch whose target misses in the cache, the fetch for the target block is completed before the load or store is performed.

9.6 CACHE INVALIDATION

Since the instruction cache can be accessed with virtual addresses, and since the cache does not use PIDs to differentiate between different virtual address spaces, the cache must be flushed of all contents in certain circumstances. Flushing is accomplished by resetting the Valid bit for each cache block, and is accomplished in a single cycle. Valid bits are reset by a processor reset or by the execution of the instructions Invalidate (INV) or Interrupt Return and Invalidate (IRETINV). The INV and IRETINV instructions must be executed in the Supervisor mode.

When an INV instruction is executed, the processor does not reset the valid bits until the next branch or the next cache-block boundary, whichever occurs first. If the INV instruction occurs as the last instruction of a block, the block boundary at which invalidation occurs is the end of the next block. This allows the processor pipeline to complete the execution of the instruction in decode when the INV instruction is executed, without forcing the instruction to be invalidated in the pipeline and refetched externally.

The processor does not invalidate locked cache blocks, unless the cache is also disabled.

CHAPTER 10

SYSTEM INTERFACE

This chapter describes the attachment of the Am29030 and Am29035 microprocessors to their hardware environments. It describes the external bus that allows the processor to communicate with external devices and memories. Processor reset, clock generation, and master/slave checking are also described in this chapter.

10.1 SIGNAL DESCRIPTION

In this section, certain outputs are described as being three-state or bi-directional. However, all outputs (except MSERR) may be placed in a high-impedance state by the Test mode. The three-state and bidirectional terminology in this section is for those outputs (except MEMCLK) that are disabled when an external device is granted the bus.

A(31–0) **Address Bus (Three-State Outputs, Synchronous)**
The Address Bus transfers the byte address for all accesses, including burst-mode accesses.

$\overline{\text{BREQ}}$ **Bus Request (Output, Synchronous)**
This output indicates that the processor needs to perform an external access.

$\overline{\text{BGRT}}$ **Bus Grant (Input, Synchronous)**
This input signals the processor that it has control of the external bus. This signal may be asserted even when $\overline{\text{BREQ}}$ is not active, in which case the processor still has control of the bus. The processor drives $\overline{\text{REQ}}$ High when granted an unrequested bus.

R/$\overline{\text{W}}$ **Read/Write (Three-state Output, Synchronous)**
This signal indicates whether data is being transferred from the processor to the external system (Low), or from the external system to the processor (High).

SUP/$\overline{\text{US}}$ **Supervisor/User Mode (Three-State Output, Synchronous)**
This output indicates the program mode for an access. If the access is performed under Supervisor mode, SUP/$\overline{\text{US}}$ is High. If the access is performed under User mode, SUP/$\overline{\text{US}}$ is Low.

$\overline{\text{LOCK}}$ **Lock (Three-State Output, Synchronous)**
This output allows the implementation of various bus and device interlocks. It may be active only for the duration of an access or for an extended period of time under control of the Lock bit in the Current Processor Status Register.

The processor does not relinquish the bus (in response to $\overline{\text{BGRT}}$) when $\overline{\text{LOCK}}$ is active.

MPGM(1–0) **MMU Programmable (Three-State Outputs, Synchronous)**
These outputs reflect the value of the two PGM bits in the Translation Look-Aside Buffer entry associated with the access. If no address translation is performed, these signals are both Low.

ID(31–0) **Instruction/Data Bus (Bidirectional, Synchronous)**
The Instruction/Data Bus transfers instructions to, and data to and from, the processor.

$\overline{\text{REQ}}$ **Request (Three-State Output, Synchronous)**
This signal requests an access. When $\overline{\text{REQ}}$ is Low, the address for the access appears on the Address Bus. This signal is precharged and then maintained by a weak internal pullup when the bus is granted to another master, to prevent spurious requests.

$\overline{\text{RDY}}$ **Ready (Input, Synchronous)**
For a read, this input indicates that a valid instruction or data word is on the Instruction/Data Bus. For a write, it indicates that the write is complete and that the data need no longer be driven on the Instruction/Data Bus. The processor ignores this signal for the first cycle of a simple access and for the first cycle of a burst-mode access (all other burst-mode cycles are not subject to this restriction).

I/$\overline{\text{D}}$ **Instruction or Data Access (Three-State Output, Synchronous)**
This signal is High during an access to indicate that the access is for an instruction and Low to indicate that the access is for data.

IO/$\overline{\text{MEM}}$ **Input/Output or Memory Access (Three-State Output, Synchronous)**
For a data access, this signal indicates whether the access is to the input/output (I/O) address space (High) or the instruction/data memory address space (Low).

$\overline{\text{BWE}}$(3–0) **Byte Write Enables (Three-State Outputs, Synchronous)**
These signals are asserted during an external write to indicate which bytes should be written. An assertion of $\overline{\text{BWE}}$3 indicates that the most significant byte (corresponding to ID(31–24)) should be written, and so on. The correspondence between the $\overline{\text{BWE}}$(3–0) and the ID(31–0) signals does not depend on the Byte Order (BO) bit of the Configuration Register. However, the correspondence between the $\overline{\text{BWE}}$(3–0) signals and A(1–0) does depend on the BO bit. These signals are asserted only for writes, and the set of signals asserted depends on the data width of the access.

$\overline{\text{ERR}}$ **Error (Input, Synchronous)**
This input indicates that an error occurred during the current access. For a read, the processor ignores the Instruction/Data Bus. For a store, the access is terminated. In either case, a Data Access Exception or Instruction Access Exception trap can occur. The processor ignores this signal if there is no pending access. This signal cannot end an access; it is sampled only when the $\overline{\text{RDY}}$ input is active.

$\overline{\text{BURST}}$ **Burst Request (Three-State Output, Synchronous)**
This signal indicates a burst-mode access. The addresses for burst-mode accesses appear on the Address Bus. The $\overline{\text{BURST}}$ signal is provided to aid the implementation of high-bandwidth transfers by informing the external system that the processor can complete an access as often as every cycle after the first cycle.

$\overline{\text{PGMODE}}$ **Page-Mode Access (Three-State Output, Synchronous)**
This indicates that the address for an access is in the same page-mode block as the address for the previous access.

$\overline{\text{ERLYA}}$ **Early Address (Input, Synchronous)**
This input is used to request the early transmission of burst-mode addresses for interleaved memories (see Section 10.4.10.4).

$\overline{\text{RDN}}$ **Read Narrow (Input, Synchronous)**
This input indicates that the accessed memory is an 8- or 16-bit device attached to ID(31–24) or to ID(31–16), respectively. This signal can be asserted for any read access—though it is probably most useful for ROM accesses—and causes the processor to perform additional read accesses to obtain the remainder of a word or half-word, if required. This signal is ignored on a write access.

The $\overline{\text{RDN}}$ signal is sampled when $\overline{\text{RESET}}$ is asserted. The level of $\overline{\text{RDN}}$ during the four cycles before the de-assertion of $\overline{\text{RESET}}$ determines whether a narrow access is 8 or 16 bits wide. If $\overline{\text{RDN}}$ is Low in each of these cycles, a narrow access is 16 bits wide. If $\overline{\text{RDN}}$ is High, a narrow access is 8 bits wide. The width of the narrow access is undefined if $\overline{\text{RDN}}$ changes in the four cycles before $\overline{\text{RESET}}$ is de-asserted.

OPT(2–0) **Option Control (Three-State Outputs, Synchronous)**
These outputs reflect the value of bits 18–16 of the load or store instruction which begins an access. Bit 18 of the instruction is reflected on OPT2, bit 17 on OPT1, and bit 16 on OPT0.

The standard definitions of these signals are as follows:

OPT2	OPT1	OPT0	Meaning
0	0	0	Word-length access
0	0	1	Byte access
0	1	0	Half-word access
1	1	0	Hardware-development system accesses
—All Others—			Reserved

During an interrupt/trap vector fetch, the OPT(2–0) signals indicate a word-length access (000). Also, the external system should return an entire, aligned word for a read, regardless of the indicated data length: the processor performs the necessary alignment.

$\overline{\text{WARN}}$ **Warn (Input, Asynchronous, Edge-Sensitive)**
A High-to-Low transition on this input causes a non-maskable $\overline{\text{WARN}}$ trap to occur. This trap bypasses the normal trap vector fetch sequence and is useful in situations where the vector fetch may not work (e.g., when data memory is faulty—see Section 8.4).

$\overline{\text{INTR}}$(3–0) **Interrupt Requests (Inputs, Asynchronous)**
These inputs generate prioritized interrupt requests. The interrupt caused by $\overline{\text{INTR}}$0 has the highest priority, and the interrupt caused by $\overline{\text{INTR}}$3 has the lowest priority. The interrupt requests are masked in prioritized order by the Interrupt Mask field in the Current Processor Status Register.

$\overline{\text{TRAP}}$(1–0) **Trap Requests (Inputs, Asynchronous)**
These inputs generate prioritized trap requests. The trap caused by $\overline{\text{TRAP}}$0 has the highest priority. These trap requests are disabled by the DA bit of the Current Processor Status Register.

STAT(2–0) **CPU Status (Outputs, Synchronous)**
These outputs indicate the state of the processor's execution stage on the previous cycle (see Section 11.3). They are encoded as follows:

STAT2	STAT1	STAT0	Condition
0	0	0	Halt or Step Modes
0	0	1	Pipeline Hold Mode
0	1	0	Load Test Instruction Mode, Halt/Freeze
0	1	1	Wait Mode
1	0	0	Interrupt Return
1	0	1	Taking Interrupt or Trap
1	1	0	Non-sequential Instruction Fetch
1	1	1	Executing Mode

CNTL(1–0) **CPU Control (Inputs, Asynchronous)**
These inputs control the processor mode (see Section 11.4) and are encoded as follows:

CNTL1	CNTL0	Mode
0	0	Load Test Instruction
0	1	Step
1	0	Halt
1	1	Normal

$\overline{\text{RESET}}$ **Reset (Input, Asynchronous)**
This input places the processor in the Reset mode (see Section 10.2.2).

$\overline{\text{TEST}}$ **Test Mode (Input, Asynchronous)**
When this input is active, the processor is in Test mode. All outputs and bi-directional lines, except MSERR, are forced to the high-impedance state.

MSERR **Master/Slave Error (Output, Synchronous)**
This output shows the result of the comparison between processor outputs and the signals provided internally to the off-chip drivers. If there is a difference for any enabled driver, this line is asserted.

INCLK **Input Clock (Input)**
This is an oscillator input to the processor, at the processor's operating frequency. It is driven with TTL levels.

MEMCLK **Memory Clock (Bidirectional)**
This is either a clock output or an input from an external clock generator, as determined by PWRCLK. It can be either at the processor's operating frequency or at one-half of this frequency, as controlled by the $\overline{\text{DIV2}}$ signal. It is driven with CMOS levels.

$\overline{\text{DIV2}}$ **Divide Clock By 2 (Input)**
If this signal is Low, the MEMCLK signal operates at one-half of the processor's operating frequency. If this signal is High, the MEMCLK signal operates at the processor's operating frequency.

The following pins are included as part of the IEEE 1149.1–1990 compliant Standard Test Access Port (see Section 11.7).

TCK — **Test Clock Input (Input, Asynchronous)**
This input clocks the Test Access Port.

TMS — **Test Mode Select (Input, Synchronous to TCK)**
This input controls the operation of the Test Access Port.

TDI — **Test Data Input (Input, Synchronous to TCK)**
This signal supplies data to the test logic from an external source. It is sampled on the rising edge of TCK.

TDO — **Test Data Output (Three-state Output, Synchronous to TCK)**
This output supplies data from the test logic to an external destination. It changes on the falling edge of TCK.

$\overline{\text{TRST}}$ — **Test Reset Input (Input, Asynchronous)**
This input asynchronously resets the Test Access Port. The reset places the test logic in a state such that it does not cause an output driver to be enabled. The $\overline{\text{TRST}}$ input must be asserted in conjunction with the $\overline{\text{RESET}}$ input for correct processor initialization.

The following pin is not a signal pin, but is named in the Am29030 and Am29035 processors' documentation because of its special role in the processor and external system.

PWRCLK — **Power Supply for MEMCLK Driver**
This pin is a power supply for the MEMCLK output driver. It isolates the MEMCLK driver and is used to determine whether or not the processor generates the clock for the external system. If power (+5 volts) is applied to this pin, the processor generates a clock on the MEMCLK output. If this pin is grounded, the processor accepts a clock generated by the external system on the MEMCLK input. Since PWRCLK supplies power to the MEMCLK output, PWRCLK should be connected directly to power (+5 volts) or ground.

The following pin is defined for use with hardware development systems (emulators). This pin does not exist on any package, and this definition is provided solely for the purposes of standardizing its location.

$\overline{\text{EMACC}}$ — **Emulator Access (output, synchronous)**
This output indicates the current access is generated by an emulator. If $\overline{\text{EMACC}}$ is Low, the access is emulator specific and the external system must not respond to the access. If $\overline{\text{EMACC}}$ is High, the access is directed to the external system. To ensure proper operation, $\overline{\text{EMACC}}$ should be connected to V_{CC} through a pullup resistor. Details regarding the operation of $\overline{\text{EMACC}}$ are available through hardware–development system vendors. To ensure emulation compatiblity with your design, contact your emulator vendor prior to layout. An overview of emulation support can be found in the Fusion29K Catalog.

The following pins are defined for future processors and should be tied High, through individual pullup resistors: $\overline{\text{HIT}}$, $\overline{\text{DI}}$, and $\overline{\text{WBC}}$.

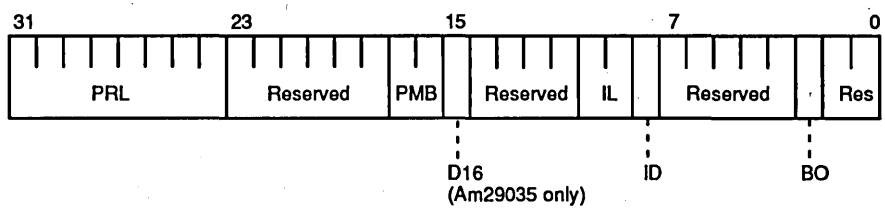
10.2 PROCESSOR RESET AND INITIALIZATION

When power is first applied to the processor, it is in an indeterminate state and must be placed in a known state. Also, under certain circumstances, it may be necessary to place the processor in a defined state. This is accomplished by the Reset mode, which places the processor into a predefined state.

10.2.1 Configuration (CFG, Register 3)

This protected special-purpose register (Figure 10-1) controls certain processor and system options. Most fields normally are modified only during system initialization. The Configuration Register is defined as follows:

Figure 10-1 Configuration Register



Bits 31–24: Processor Release Level (PRL)—The PRL field is an 8-bit, read-only identification number which specifies the processor version.

Bits 23–18: Reserved.

Bit 17–16: Page-Mode Block (PMB)—The PMB field determines the size of a page-mode block. A page-mode block is a region of the external instruction/data memory that has a common row address. The correspondence between the field value and the page-mode block size is as follows:

PMB Value	Page-Mode Block Size
00	2K byte
01	4K byte
10	8K byte
11	16K byte

Bit 15: Data Width 16 Bits (D16—Am29035 only)—The D16 bit determines the data width of all external instruction/data bus transfers. If the D16 bit is 1, all external accesses are 16 bits wide. If a 32-bit word is accessed externally, the processor performs two accesses to read or write the entire word in units of 16 bits. If the D16 bit is 0, external data accesses are 32 bits wide. This bit is implemented only in the Am29035 microprocessor.

Bit 14–11: Reserved.

Bit 10–9: Instruction Cache Lock (IL)—The IL field controls the locking of all or a portion of the instruction cache. When a cache block is locked, it is not invalidated (unless the cache is disabled), and it is not replaced if it is valid. It can be allocated for replacement if it is invalid. When a block is not locked, replacement and invalidation occur normally as described in Chapter 9. The IL field values are defined as follows:

IL Value	Effect on Cache Lock
00	Unlock cache for Am29030 processor
01	Entire cache locked
10	Blocks in column 0 locked (Am29030 processor)
11	Unlock cache for Am29035 processor

Bit 8: Instruction Cache Disable (ID)—If the ID bit is 1, the instruction cache is disabled, and the instruction cache does not satisfy any processor instruction fetch. Also, fetched instructions are not stored into a disabled cache. However, a disabled cache may be invalidated by an INV or IRETINV instruction. If the ID bit is 0, the instruction cache is enabled, and the instruction cache satisfies all instruction fetches for which it contains the appropriate instruction. If the ID bit is changed, the effect of this change is delayed until the next subsequent cache block boundary.

Bit 7–3: Reserved.

Bit 2: Byte Order (BO)—The BO bit determines the ordering of bytes and half-words within words. Section 3.3.7.1 describes the interpretation of the BO bit and its effect on byte and half-word addressing.

Bit 1–0: Reserved.

10.2.2 Reset Mode

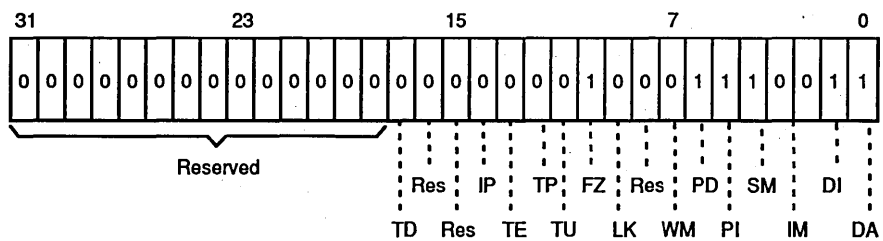
The Reset mode is invoked by asserting the $\overline{\text{RESET}}$ input. The Reset mode is entered within four processor cycles after $\overline{\text{RESET}}$ is asserted. The $\overline{\text{RESET}}$ input must be asserted for at least four processor cycles to accomplish a processor reset.

The Reset mode can be entered from any other processor mode (see Section 11.5). If the $\overline{\text{RESET}}$ input is asserted at the time power is first applied to the processor, the processor enters the Reset mode only after four cycles have occurred on the MEMCLK pin.

The Reset mode configures the processor state as follows:

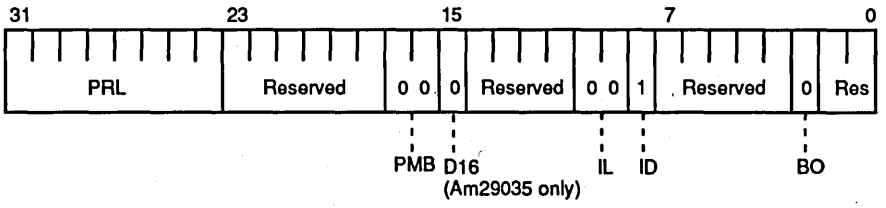
1. Instruction execution is suspended.
2. Instruction fetching is suspended.
3. Any interrupt or trap conditions are ignored.
4. The Current Processor Status Register is set as shown in Figure 10-2.

Figure 10-2 Current Processor Status Register In Reset Mode



5. The Configuration Register is set as shown in Figure 10-3.
6. The Contents Valid (CV) bit of the Channel Control Register is reset.

Figure 10-3 Configuration Register in Reset Mode



Except as previously noted, the contents of all general-purpose registers, special-purpose registers, and TLB registers are undefined. The contents of the Instruction Cache are also undefined.

The Reset mode is exited when the $\overline{\text{RESET}}$ input is de-asserted. Either three or four cycles after $\overline{\text{RESET}}$ is de-asserted (depending on internal synchronization time), the processor performs an initial instruction access on the external interface. The initial instruction access is directed to address 0 in instruction/data memory.

If the CNTL(1–0) inputs are 11 after reset, the processor enters the Executing mode. If the CNTL(1–0) inputs are 10 or 01 when $\overline{\text{RESET}}$ is de-asserted, the processor enters the Halt or Step mode, respectively upon completion of the initial instruction access. If the processor enters the Halt mode after reset, the protection checking that normally applies to the Halt instruction is disabled, so that the Halt instruction can be used as an instruction breakpoint in a User-mode program (see Section 11.5).

The Load Test Instruction mode cannot be directly entered from the Reset mode. If the CNTL(1–0) inputs are 00 immediately after $\overline{\text{RESET}}$ is de-asserted (Load Test Instruction mode), the effect on processor operation is unpredictable.

10.2.3 Am29035 Processor Initialization Considerations

Only a subset of the Instruction Cache Lock (IL) fields of the Configuration Register apply to the Am29035 microprocessor. To enable the cache, the IL field must contain the value 11. To lock the contents of the Am29035 processor's instruction cache, the IL field must contain the value 01. For either the enabled or the locked state, the Instruction Cache Disable (ID) bit of the Configuration Register must be 0 . To disable the Am29035 processor's instruction cache, the ID bit must be set (see Section 10.2.1).

10.3 CLOCKS

The processor's clocks are derived from the INCLK signal, which runs at the processor's operating frequency. An INCLK signal must always be provided.

The MEMCLK signal is the timing reference for all external accesses. The external interface operates at either the processor's frequency or at one-half of this frequency. The frequency of the external interface, with respect to the processor's operating frequency, is controlled by the DIV2 signal. If DIV2 is High, the MEMCLK signal and the external interface operate at the processor's operating frequency. If DIV2 is Low, the MEMCLK signal and the external interface logic operate at one-half of the processor's operating frequency. This external bus timing is designed to allow operating at high frequencies. The half-frequency option is provided to simplify the interface design while allowing the processor to operate at high frequencies.

The MEMCLK rising edge can be synchronized to INCLK via the synchronous deassertion of RESET.

MEMCLK can be either an output or an input, as controlled by the PWRCLK power-supply pin. If power (i.e., +5 volts) is applied to the PWRCLK pin, the processor is configured to generate MEMCLK for the system. If PWRCLK is grounded, the processor is configured to receive an externally generated MEMCLK.

If MEMCLK is an input, the processor has better input setup times when MEMCLK and INCLK are tied together than when they are driven separately, because tying MEMCLK and INCLK together reduces their skew to a minimum. However, tying these signals together is possible only if the interface operates at the processor's frequency. In any case, the MEMCLK signal must not precede the INCLK signal and must follow the INCLK signal by a specified minimum time.

10.3.1 Electrical Specifications

INCLK is driven with TTL levels and MEMCLK is driven with CMOS levels. If MEMCLK is driven as an input, it must be driven with CMOS levels.

Note that the MEMCLK pin is placed in the high-impedance state by the Test mode.

10.4 BUS DESCRIPTION

The external interface provides the bandwidth required for performance while permitting the connection of many different types of devices. This section describes the external interface and the methods of connecting devices and memories to the processor. Timing diagrams for the operations described in this chapter appear in Appendix A.

10.4.1 Bus Overview

The external interface consists of two 32-bit synchronous buses with associated control and status signals: the Address Bus and Instruction/Data Bus. The Address Bus transfers addresses and control to devices and memories. The Instruction/Data Bus transfers data to and from devices and memories, and transfers instructions to the processor from instruction memories. In addition, a set of signals allows control of the bus to be requested by and granted to the processor.

There are four logical groups of signals performing four distinct functions, as follows:

1. Address transfer and access requests: A(31–0), $\overline{\text{REQ}}$, R/$\overline{\text{W}}$, I/$\overline{\text{D}}$, IO/$\overline{\text{MEM}}$, $\overline{\text{BWE}}$(3–0), $\overline{\text{BURST}}$, $\overline{\text{PGMODE}}$, SUP/$\overline{\text{US}}$, $\overline{\text{LOCK}}$, OPT(2–0), MPGM(1–0).
2. Instruction and Data transfer: ID(31–0), $\overline{\text{RDY}}$, $\overline{\text{ERR}}$.
3. Address and access sequencing: $\overline{\text{ERLYA}}$, $\overline{\text{RDN}}$.
4. Arbitration: $\overline{\text{REQ}}$ and $\overline{\text{BGRT}}$.

The signals in the first group are used throughout an access to indicate the address and status of the access—or sequence of accesses in the case of a burst-mode access. The signals in the second group are used to transfer data to and from the processor and to indicate the validity of an access. The signals in the third group are responses from the slave device or memory that cause the processor to take certain actions during an access. Finally, the signals in the fourth group are used by the processor to request use of the bus and by the external system to grant the bus to the processor.

All signals change at the rising edge of MEMCLK. There are no signals that change at the half-cycle points. Also, the external system has at least one cycle to generate all responses to the processor. There are no cases where the external system must generate a response to a signal by the end of the cycle in which the signal is changed.

10.4.2 User-Defined Signals

Two types of user-defined outputs on the processor control devices and memories in a system-dependent manner. Each of these outputs is valid simultaneously with—and for the same duration as—the address for an access.

The first set of user-defined signals, MPGM(1–0), is determined by the PGM bits in the Translation Look-Aside Buffer entry used in address translation. If address translation is not performed, these outputs are both Low.

The second set of signals, OPT(2–0), are determined by bits 18-16 of the load or store instruction that initiates an access. These signals are valid only for data accesses. They are driven Low for instruction accesses.

Standard interpretations of OPT(2–0) are given in Section 10.1. Since the OPT(2–0) signals are determined by instructions, they have an impact on application-software compatibility, and system hardware should use the given definitions of OPT(2–0). The OPT(2–0) signals are used to encode byte and half-word accesses. However, for a load, the system should return an entire, aligned word, regardless of the indicated data width. In this case, the processor performs the required alignment.

Note that the standard interpretations of OPT(2–0) apply only to data accesses to instruction/data memory and input/output.

For interrupt and trap vector fetches, the MPGM(1–0) and OPT(2–0) outputs are all Low.

10.4.3 Instruction Accesses

An instruction access is indicated by a High level on the I/$\overline{D}$ output during an access. Instruction accesses are always performed in groups of four instructions and always begin and end on a cache-block boundary, regardless of whether or not the cache is enabled.

If a taken branch is executed while the processor is fetching the last instruction of a block boundary, the processor will attempt to terminate the burst so as not to fetch an entire block of unrequired instructions. This will result in the burst terminating after the first instruction of the next block is returned from memory. This instruction will be discarded.

Some of the protocol signals are not required, but are driven to default levels during an instruction access. These signals and the default levels are:

- R/$\overline{W}$: High
- IO/$\overline{MEM}$: Low, unless the instruction address is translated and the IO bit in the corresponding TLB entry is 1
- $\overline{BWE}$(3–0): all High
- OPT(2–0): all Low

Even though instruction accesses are indicated by the I/$\overline{D}$ signal, this signal is not intended as a means of implementing separate instruction and data address spaces. The I/$\overline{D}$ signal is primarily informative: it is defined in the interface in anticipation of

cache-consistency protocols to indicate whether or not a consistency operation might be required for the access.

10.4.4 Data Accesses

A data access is indicated by a Low level on the I/$\overline{D}$ output during an access. Data accesses are performed as single accesses except when initiated by a Load Multiple or Store Multiple instruction.

Data reads and writes are differentiated by High and Low levels on the R/$\overline{W}$ line, respectively. For a read, the processor expects data to be driven as soon as the second cycle after the access begins. For a simple write access, the processor provides data in the second cycle after the access begins. Write data access is delayed by one cycle so that the A(31–0) and ID(31–0) outputs do not all switch in any given cycle and to avoid read/write bus collision. This reduces power-distribution problems at high frequencies. Burst-mode write accesses are described in Section 10.4.10. In the cycle after $\overline{RDY}$ is asserted for the write, the processor places the Instruction/Data bus in the high-impedance state, unless the write is one in a sequence of burst-mode writes and is not the final write in the sequence.

A data access can be either for a byte, a half-word, or a word. In the case of a read, the external system always returns a word and is not concerned with the data width, because the processor performs alignment and sign extension, if required. In the case of a write, the processor asserts the appropriate write enables on the $\overline{BWE}$(3–0) outputs, and the external system need not examine OPT(2–0), R/$\overline{W}$, or A(1–0) to form the correct enables. For a byte or half-word write, the selected byte or half-word is replicated in all byte or half-word positions on the ID(31–0) lines. The $\overline{BWE}$(3–0) signals are asserted only when R/$\overline{W}$ is Low and the access is valid (for example, there are no MMU protection violations). The $\overline{BWE}$(3–0) signals depend on OPT(2–0), A(1–0), and the Byte Order (BO) bit of the Configuration Register as follows (the value "0" is Low, "1" is High, and "x" is a don't care):

BO	OPT(2–0)	A(1–0)	$\overline{BWE}$(3–0) (on write)	
0	001	00	0111	(LSB, Big Endian)
0	001	01	1011	
0	001	10	1101	
0	001	11	1110	(MSB, Big Endian)
0	010	0x	0011	(LSHW, Big Endian)
0	010	1x	1100	(MSHW, Big Endian)
1	001	00	1110	(LSB, Little Endian)
1	001	01	1101	
1	001	10	1011	
1	001	11	0111	(MSB, Little Endian)
1	010	0x	1100	(LSHW, Little Endian)
1	010	1x	0011	(MSHW, Little Endian)
x	000	xx	0000	(word access)
x	110	xx	1111	(hardware development)
	—all other writes—		0000	

10.4.5 Read-Only Memories

The processor includes two provisions for simplifying the interface to read-only memories (ROMs). First, it is possible to connect to the bus a ROM which is only 8- or 16-bits wide. The processor performs all sequencing to access full words. Also,

the processor provides for an external ROM that contains both instructions and data and which is in the instruction/data memory address space. The full range of accesses provided for the instruction/data memory, such as byte and half-word accesses, are also available for the ROM.

10.4.5.1 NARROW READ INTERFACE

When an 8-bit-wide ROM is attached to the bus, it must be connected to ID(31–24). A 16-bit-wide ROM must be connected to ID(31–16). The ROM can respond to any read access—either an instruction or data access—and indicates that it is less than 32-bits wide by asserting the $\overline{\text{RDN}}$ signal along with the $\overline{\text{RDY}}$ signal at the end of an access. The $\overline{\text{RDN}}$ response is valid only for a read and is ignored on a write. Because the $\overline{\text{RDN}}$ signal is sampled for every read, accesses to narrow ROMs may be freely mixed with other 32-bit accesses.

The narrow ROM option is supported only for systems in which the BO bit is 0 (big-endian). In addition, the $\overline{\text{RDN}}$ signal is ignored during reads from the hardware-development system (OPT(2–0) = 110).

The level driven on $\overline{\text{RDN}}$ during a processor reset determines whether a narrow ROM is 8- or 16-bits wide. If $\overline{\text{RDN}}$ is High in the four cycles before $\overline{\text{RESET}}$ is de-asserted, a narrow access is 8 bits wide. If $\overline{\text{RDN}}$ is Low in the four cycles before $\overline{\text{RESET}}$ is de-asserted, a narrow access is 16-bits wide. If $\overline{\text{RDN}}$ changes in the four cycles before $\overline{\text{RESET}}$ is de-asserted, the width of a narrow access is unpredictable. Narrow accesses in a particular system are all either 8- or 16-bits wide, and the width cannot be changed after a processor reset.

10.4.5.2 8-BIT NARROW ACCESSES

If the processor expects a half-word or a word on a read (that is, if the access is not a byte read), and a narrow ROM is 8-bits wide, the $\overline{\text{RDN}}$ response causes the processor to generate one (for a half-word) or three (for a word) more requests immediately following the first access. The address for each subsequent access is the same as the address for the first access, except that A(1–0) are incremented by one for each access. The processor can accept a $\overline{\text{RDY}}$ response in the first cycle that the second access appears on the bus (the access is similar to a burst-mode access in this respect). The slave must drive $\overline{\text{RDY}}$ High if it cannot respond to the second access immediately. The slave also must continue to assert $\overline{\text{RDN}}$ throughout all remaining accesses in conjunction with $\overline{\text{RDY}}$. The $\overline{\text{ERR}}$ signal may be asserted for any access in the sequence, but an error is reported only if $\overline{\text{ERR}}$ is asserted for the final access. The $\overline{\text{ERLYA}}$ signal is ignored throughout the access.

The processor assembles the final word or half-word by placing the first received byte in the high-order byte position of the word or half-word, the second received byte in the next-lower-order byte position, and so on until the entire word or half-word is assembled.

If the read access is a byte access, the processor performs only one access. Note that, for a word or half-word read, the processor is sensitive to $\overline{\text{RDY}}$ in the cycle following the first access whereas, for a byte read, the processor is not sensitive to $\overline{\text{RDY}}$ in the cycle after the first (and final) access. Because of this, the OPT(2–0) signals indicate whether or not an 8-bit ROM should drive $\overline{\text{RDY}}$ following the first access, because OPT(2–0) indicate the access data width.

If the processor generates an unaligned half-word or word read, the $\overline{\text{RDN}}$ response does not permit the implementation of the unaligned read. The address sequence

generated by the processor to assemble the half-word or word wraps within the half-word or word.

10.4.5.3 16-BIT NARROW ACCESSES

If the processor expects a word on a read, and a narrow ROM is 16-bits wide, the $\overline{RDN}$ response causes the processor to generate one more request immediately following the first access. The address for the second access is the same as the address for the first access, except that A(1–0) are incremented by two for the second access. The processor can accept a $\overline{RDY}$ response in the first cycle that the second access appears on the bus. The slave must drive $\overline{RDY}$ High if it cannot respond in this cycle. The slave also must continue to assert $\overline{RDN}$ during the second access. The $\overline{ERR}$ signal may be asserted for either access, but an error is reported only if $\overline{ERR}$ is asserted for the second access. The $\overline{ERLYA}$ signal is ignored throughout the access.

The processor assembles the final word by placing the first received half-word in the high-order, half-word position of the word and the second received half-word in the low-order, half-word position.

If the read access is a byte or half-word access, the processor performs only one access. Note that, for a word read, the processor is sensitive to $\overline{RDY}$ in the cycle following the first access whereas, for a byte or half-word read, the processor is not sensitive to $\overline{RDY}$ in the cycle after the first (and final) access. Because of this, the OPT(2–0) signals indicate whether or not a 16-bit ROM should drive $\overline{RDY}$ following the first access, because OPT(2–0) indicate the access data width.

If the processor generates an unaligned word read, the $\overline{RDN}$ response does not permit the implementation of the unaligned read. The address sequence generated by the processor to assemble the word wraps within the word.

10.4.5.4 ROM ADDRESS MAPPING

The processor performs the instruction fetches for $\overline{RESET}$ and the $\overline{WARN}$ trap from locations 00000000 and 00000010 (hexadecimal) of the instruction/data memory, respectively. If a ROM is present, it must map to the instruction/data address space starting at location 00000000. Any defined read access can be performed on the ROM. Also, address translation is possible for ROM accesses. If a ROM is mapped at location 00000000, the interrupt/trap Vector Area must be located in an area beyond this location in memory.

10.4.6 Programmable Bus Sizing (Am29035 Processor Only)

For a data access, the Instruction/Data Bus can be programmed to be either 16- or 32-bits wide by the D16 bit of the Configuration Register. If the D16 bit is 0, the ID bus is 32-bits wide. If the D16 bit is 1, the ID bus is 16-bits wide, and only ID(31–16) are used to transfer data to and from the processor. If the ID Bus is 16-bits wide for data accesses, the processor performs two accesses to read or write a full word. The 16-bit bus option is supported only for systems in which the BO bit is 0 (big-endian). A hardware-development system access (identified by OPT(2–0) = 110) always uses a 32-bit bus.

To read a 32-bit word, the processor first reads the high-order 16 bits of the word, then generates a second access to read the low-order 16 bits of the word. The address is incremented by two for the second access. The processor is sensitive to $\overline{RDY}$ in the cycle immediately following the first access, so the slave must control $\overline{RDY}$ in this cycle—$\overline{RDY}$ can be driven Low if the slave can respond in this cycle, but must

be High if the slave cannot respond. The slave can assert $\overline{ERR}$ for any access in the sequence, but an error is reported only if $\overline{ERR}$ is asserted for the second access. The $\overline{ERLYA}$ signal is ignored during the entire access.

To read an 8-bit byte or 16-bit half-word on a 16-bit bus, the processor performs only a single access. Alignment and sign extension are performed as usual, except that the required byte or half-word is received on ID(31–16).

To write a 32-bit word, the processor first writes the high-order 16 bits of the word, then generates a second access to write the low-order 16 bits of the word. The address is incremented by two for the second access, and the low-order bits of the word appear on ID(31–15). The processor is sensitive to $\overline{RDY}$ in the cycle immediately following the first access, so the slave must control $\overline{RDY}$ in this cycle—$\overline{RDY}$ can be driven Low if the slave can respond in this cycle, but must be High if the slave cannot respond. The slave can assert $\overline{ERR}$ for any access in the sequence, but an error is reported only if $\overline{ERR}$ is asserted for the second access. The $\overline{ERLYA}$ signal is ignored during the entire access.

To write an 8-bit byte or 16-bit half-word on a 16-bit bus, the processor performs only a single access. For a byte write, the appropriate byte is replicated on both ID(31–24) and ID(23–16). For a half-word write, the appropriate half-word appears on ID(31–16). The $\overline{BWE}$(3–0) signals are asserted as follows (the value "0" is Low, "1" is High, and "x" is a don't care):

OPT(2–0)	A(1–0)	$\overline{BWE}$(3–0) (on write)	
001	00	0111	
001	01	1011	
001	10	0111	
001	11	1011	
010	0x	0011	
010	1x	0011	
110	xx	1111	(hardware development)
—all other writes (two cycles)—		0011	

Note that, for a word access, the processor is sensitive to $\overline{RDY}$ in the cycle following the first access whereas, for a byte or half-word access, the processor is not sensitive to $\overline{RDY}$ in the cycle after the first (and final) access. Because of this, the OPT(2–0) signals indicate whether or not a 16-bit slave should drive $\overline{RDY}$ following the first access, because OPT(2–0) indicate the access data width.

If $\overline{RDN}$ is asserted by a 8-bit ROM in response to a data read on a 16-bit bus (D16 = 1), the $\overline{RDN}$ response takes precedence. The processor treats the access as an 8-bit narrow access.

In order to set the D16 bit and activate the reduced bus size, the processor must be able to fetch instructions. If the memory devices in the system are all either 8- or 16-bits wide, they must use narrow reads to supply the instructions that set the Configuration Register.

10.4.7 Reporting Errors

The $\overline{ERR}$ signal is used to report external errors. However, the $\overline{ERR}$ signal cannot be used to end an access. The $\overline{RDY}$ signal alone ends an access, and $\overline{ERR}$ is sampled only when $\overline{RDY}$ is active. The $\overline{ERR}$ signal has a shorter setup time than $\overline{RDY}$ to simplify the implementation of error checking. For example, parity checking can be

performed with combinatorial logic that directly drives $\overline{ERR}$. This logic has more time to setup the $\overline{ERR}$ signal than it would have to setup the $\overline{RDY}$ signal. Also, the system design need not be concerned with spurious assertions of $\overline{ERR}$ that might be caused by the combinatorial logic, because $\overline{ERR}$ cannot end an access.

If the processor receives an $\overline{ERR}$ response in conjunction with a $\overline{RDY}$ during an instruction or data access, it ignores the content of the Instruction/Data Bus. An $\overline{ERR}$ response to an instruction fetch causes an Instruction Access Exception trap, unless it is one of the residual instructions in a block that are not executed due to a prior non-sequential instruction fetch (see Section 9.4.2). An $\overline{ERR}$ response to a data access causes a Data Access Exception trap.

The processor supports the restarting of unsuccessful accesses upon an interrupt return. In the case of an unsuccessful instruction access, the restart is performed by the Program Counter 0 and Program Counter 1 registers. In the case of an unsuccessful data access, the restart is performed by the Channel Address, Channel Data, and Channel Control registers. In any event, the control program must determine whether or not an access can and/or should be restarted.

The Instruction Access Exception and Data Access Exception traps cannot be masked. If one of these traps occurs within an interrupt or trap handler, the processor state may not be recoverable.

10.4.8 Access Protocols

The processor implements two access protocols: simple and burst-mode. Both access protocols share some common features. First, the $\overline{RDY}$ signal is ignored during the first cycle of a simple access and the first cycle of the initial burst-mode access. Second, both accesses may be further specified as being a page-mode access.

To simplify the implementation of interleaved memories, the processor has provision for supplying addresses early during burst-mode accesses.

10.4.8.1 PAGE-MODE ACCESSES

A page-mode access, indicated by the $\overline{PGMODE}$ signal being Low, can be performed either for a simple access or for an access in a sequence of burst-mode accesses. A page-mode access is one that is within the same page-mode block as the previous processor access. The Page-Mode Block field of the Configuration Register defines the size of a page-mode block. A page-mode access is possible only in the instruction/data address space and cannot be the first access after the processor is granted the bus.

10.4.9 Simple Accesses

The processor performs simple accesses only for single data reads and writes. The address, $\overline{REQ}$, and associated controls are driven throughout the access until $\overline{RDY}$ is asserted; these signals do not have to be latched by the external system. The $\overline{RDY}$ signal is ignored during the first cycle of any access, to relax timing constraints on the external system response.

The processor can begin a new access in the cycle after $\overline{RDY}$ is asserted. The $\overline{RDY}$ signal is ignored in the first cycle of the second access even if the access is to the same device or memory as the first access, unless the second access is in a sequence of burst-mode accesses (Section 10.4.10.1), in a sequence of accesses to assemble a word or half-word from a narrow ROM (Section 10.4.5), or the second of a

pair of accesses to perform a word access with a 16-bit bus width (see Section 10.4.6)

10.4.10 Burst-Mode Accesses

Burst-mode accesses are used for all instruction fetches and for load multiple and store multiple accesses. The intent of the burst-mode access is to simplify the implementation of high-bandwidth, sequential accesses.

10.4.10.1 BURST-MODE OVERVIEW

The processor indicates a burst-mode access by asserting the $\overline{\text{BURST}}$ signal during the first cycle of the access. The $\overline{\text{BURST}}$ signal indicates that, once the current access is complete, the processor will require another access at the next sequential address.

The address bus of the processor transmits every address in the burst-mode sequence (unless the external system requests early addresses for interleaved memories, as explained below). For example, if $\overline{\text{BURST}}$ is asserted for an access, the processor generates a new request and transmits a new sequential address in the cycle after $\overline{\text{RDY}}$ is received for the first access. The second access differs from a simple access only because the processor does not ignore $\overline{\text{RDY}}$ in the first cycle of the second access, so the external system can generate responses at a rate of one per cycle.

Even if a slave device does not support burst-mode accesses, it still must sample the $\overline{\text{BURST}}$ pin. Unlike the initial access in the burst-mode sequence, the processor does not ignore $\overline{\text{RDY}}$ for the first cycle of each subsequent access in the burst-mode sequence, so the slave must drive $\overline{\text{RDY}}$ High in the first cycle if it is not ready to respond to the subsequent access. Other than this, the slave can ignore the burst-mode access.

The processor does not suspend a burst-mode access. The only internal conditions that might cause the suspension of a burst-mode access is a pipeline hold during instruction prefetching. However, the pipeline hold can occur only because of a load or store. The load or store pre-empts instruction prefetching, so it is pointless to suspend the burst-mode instruction access.

The sequential addresses transmitted during the burst-mode accesses depend on the $\overline{\text{RDN}}$ input and the width of the ID bus. If $\overline{\text{RDN}}$ is not asserted and the D16 bit is 0, the address is incremented by 4 for each access. If $\overline{\text{RDN}}$ is asserted or the ID bus is 16-bits wide, the address is incremented by 1 or 2 for each access. If $\overline{\text{RDN}}$ is asserted or the bus is 16-bits wide, the processor also takes the other actions that apply to a narrow access. The $\overline{\text{RDN}}$ signal must remain asserted throughout the burst-mode access.

10.4.10.2 PROCESSOR PRE-EMPTION, TERMINATION, OR CANCELLATION OF A BURST-MODE ACCESS

The processor pre-empts a burst-mode access when an external bus master regains control of the bus, or when a burst-mode access crosses a potential virtual-page boundary. Since the minimum page size is 1K byte, burst-mode instruction and data accesses are preempted whenever the address sequence crosses a 1K-byte address boundary. The burst-mode access is re-established as soon as a new address translation is performed (if required). A new physical address is transmitted when the burst-mode access is re-established.

The processor terminates a burst-mode access whenever all instructions or data have been accessed. Burst-mode instruction accesses are terminated after a taken branch is executed or when the bus is required for a data access. In either case, the processor will terminate the instruction fetch at the next quad-word address boundary.

The processor cancels a burst-mode access when an interrupt or trap is taken. Note that a trap may be caused by the burst-mode access, for example if the destination register of a LOADM instruction is protected by the Register Bank Protection Register. If the processor cancels a burst-mode access when an access in the sequence remains to be complete, this access must be completed in spite of the cancellation.

Canceled burst-mode data accesses may be restarted at some (possibly much later) point in execution via the Channel Address, Channel Data, and Channel Control registers. In this case, the burst-mode is restarted at the point at which it was canceled rather than at the beginning of the original address sequence (see Section 8.6.2).

In the Am29030 and Am29035 microprocessors, pre-emption, termination, and cancellation of a burst-mode access appears the same to the external memory system. The $\overline{\text{REQ}}$ output remains asserted throughout a burst-mode access, because the processor is always requesting an access. The processor terminates a burst-mode access by de-asserting $\overline{\text{BURST}}$ during the final access, providing a positive indication that the burst-mode access is ending. The $\overline{\text{REQ}}$ signal is de-asserted in the cycle after the $\overline{\text{RDY}}$ response to the final access, unless the processor requests a new access.

When $\overline{\text{BURST}}$ is de-asserted, the processor is expecting one more access.

10.4.10.3 SLAVE CANCELLATION OF A BURST-MODE ACCESS

The slave can cancel a burst-mode access by asserting $\overline{\text{ERR}}$ and $\overline{\text{RDY}}$ when the processor is expecting an access in the sequence. The processor de-asserts $\overline{\text{REQ}}$ and $\overline{\text{BURST}}$ in the cycle following the $\overline{\text{ERR}}$ response and does not expect any more accesses. The processor can generate a new request in the second cycle following the $\overline{\text{ERR}}$ response. Note that the $\overline{\text{ERR}}$ response may cause an Instruction Access Exception Trap for an instruction access and does cause a Data Access Exception trap for a data access.

10.4.10.4 USING $\overline{\text{ERLYA}}$ FOR INTERLEAVED MEMORY SYSTEMS

There are a number of ways to use burst-mode accesses to improve the performance of the memory system. Some devices, such as video DRAMs and burst-mode ROMs, use the indication of a burst-mode access to implement fast, sequential accesses. For other devices, such as page-mode DRAMs, it may be necessary to interleave banks of memories to achieve high bandwidth and take advantage of burst-mode accesses. Because the Address Bus is available throughout a burst-mode access, the processor can use the Address Bus to simplify the implementation of interleaved memories. This feature can also be used to simplify the implementation of 64-bit and 128-bit-wide memories to provide bandwidth.

So that bank accesses can be started sufficiently ahead of time, an interleaved memory requires addresses to be available earlier than does a burst-mode memory. These addresses are requested early through use of the $\overline{\text{ERLYA}}$ signal. If $\overline{\text{ERLYA}}$ is asserted in the second cycle of the access, the processor transmits, on the next cycle, an incremented address according to the following table (the value "x" is a don't care):

A3	A2	A1	Next Address
0	0	x	current address + 8
1	x	x	current address + 4
x	1	x	current address + 4

The intent of this addressing pattern is to skip the addresses for odd-addressed banks and provide early the addresses for even-addressed banks. This allows the external system to begin accesses ahead of time, using the early addresses, while completing accesses from other memory banks. The external system can easily generate the addresses for odd-addressed banks using the addresses for even-addressed banks, without using an incrementer.

Also, because of timing constraints, the processor must be able to generate the first early address without incrementing, but rather by toggling address bit A3. This requires that word accesses be quad-word aligned. If the alignment restriction is not met, the address is incremented by 4 until the address becomes aligned, and then is incremented by 8 as described below.

Following the first early address, the processor ignores $\overline{\text{ERLYA}}$ for one cycle and then increments the address by 8 on the following assertion of $\overline{\text{ERLYA}}$. Since incrementing is controlled by $\overline{\text{ERLYA}}$, the external system can control the frequency with which addresses are incremented.

Once the external system has started requesting addresses early, it must continue to request addresses with the proper timing. The processor does not increment addresses during the burst-mode access without an active level on $\overline{\text{ERLYA}}$, if $\overline{\text{ERLYA}}$ has been asserted at any point in the access. Also, the processor does not indicate which addresses are beyond those required for the access, such as those that are beyond the termination point of the burst-mode access or those that are beyond a 1-Kbyte address boundary. The memory may begin these unneeded accesses early, but responses to the processor are still controlled by $\overline{\text{BURST}}$ and $\overline{\text{REQ}}$. At a 1-Kbyte address boundary, the address wraps to the beginning of the 1-Kbyte block to prevent spurious accesses beyond the boundary. Whether or not addresses are requested early, the processor always tracks the address of the current access for the purpose of exception reporting and recovery.

The $\overline{\text{ERLYA}}$ signal is ignored for simple accesses and is ignored if $\overline{\text{RDN}}$ is asserted or if the ID bus is 16-bits wide.

10.4.11 Arbitration

The processor requests the bus by asserting $\overline{\text{BREQ}}$ whenever it might want to use the bus, and it does not drive the bus until the cycle after $\overline{\text{BGRT}}$ is asserted. The $\overline{\text{BREQ}}$ signal is asserted at the beginning of the execute stage of loads and stores and upon detection of a cache miss that might cause a demand fetch. This approach causes the processor to request the bus at the earliest possible time, but also may cause the processor to request the bus when the bus is not really required. For example, the bus can be requested for a demand fetch that does not occur because of a branch. In such cases, the processor de-asserts $\overline{\text{BREQ}}$ as soon as it discovers it does not need the bus. If the bus is granted, the processor keeps $\overline{\text{REQ}}$ High so that it does not generate a spurious request.

To improve processor performance, it is possible for the external system to grant the bus to the processor before the bus is requested. This is accomplished simply by asserting $\overline{\text{BGRT}}$. If $\overline{\text{BGRT}}$ is active at the end of any cycle, the processor may use the bus in the next cycle, even though $\overline{\text{BGRT}}$ is not asserted in direct response to $\overline{\text{BREQ}}$

being asserted. The processor may or may not assert $\overline{BREQ}$ before driving the request, but at least will assert $\overline{BREQ}$ in the same cycle in which the request is driven.

After performing all required accesses, the processor de-asserts $\overline{BREQ}$ in the cycle following the $\overline{RDY}$ response to the final access. The $\overline{REQ}$ signal is driven High in the first half of the the same cycle; in the second half of this cycle, the High level on the $\overline{REQ}$ signal is maintained by a weak (easily over-driven) internal pullup device. This allows another device to drive $\overline{REQ}$ in the cycle following release of the bus without the possibility of a bus collision. The external system can de-assert $\overline{BGRT}$ following the de-assertion of $\overline{BREQ}$, but this is not required. If $\overline{BGRT}$ remains active, the processor keeps $\overline{REQ}$ inactive as long as it does not require the bus.

Bus arbitration in the Am29030 and Am29035 microprocessors is pre-emptive, in that the external system can force the processor off of the bus by de-asserting $\overline{BGRT}$. When $\overline{BGRT}$ is de-asserted, the processor progresses up to an appropriate stopping point (for example, the end of the cache block boundary in the case of a cache reload) and then de-asserts $\overline{REQ}$ in the cycle following the $\overline{RDY}$ response to the final access. The $\overline{RDY}$ signal is driven High in the first half of this cycle, then maintained by a weak pullup in the second half of the cycle. Note that the processor does not relinquish the bus in response to $\overline{BGRT}$ when the $\overline{LOCK}$ signal is active. Also, the processor may not complete all accesses for a load multiple or store multiple before it relinquishes the bus, but will resume the load multiple or store multiple at the appropriate point when it regains the bus.

Whenever the external system takes control of the bus, the processor disables the page-mode comparator. When the bus is granted to the processor, the first access cannot be a page-mode access. This prevents the processor from generating a page-mode access for which the external system may be unprepared.

10.4.11.1 USING THE PROCESSOR AS AN ARBITER

The processor can be used as a bus arbiter, even though this is not immediately apparent from the preceding description of bus arbitration. This is possible because arbitration is pre-emptive and the processor can act as a default master, by simply not driving the bus if the bus is granted but not requested.

To use the processor as an arbiter, the $\overline{BGRT}$ signal acts as a request, and $\overline{REQ}$ acts as a grant to the external system. The external system requests the bus by de-asserting $\overline{BGRT}$, then waits until $\overline{REQ}$ is High before using the bus. (If the system uses the $\overline{LOCK}$ output, $\overline{LOCK}$ must also be High—see below.) The $\overline{REQ}$ signal may already be High when $\overline{BGRT}$ is de-asserted, but the system should wait until the end of the cycle in which $\overline{BGRT}$ is de-asserted in case $\overline{REQ}$ is asserted during this cycle. (When $\overline{BGRT}$ is de-asserted, any access in progress at the same time—and which may have just started, as in Figure A-41—is completed before the processor grants the bus by de-asserting $\overline{REQ}$.) When the external system is finished with the bus, it asserts $\overline{BGRT}$. The processor resumes control of the bus in the cycle following the assertion of $\overline{BGRT}$ and either drives an active request or drives $\overline{REQ}$ High. The external system is responsible for maintaining a proper level on $\overline{REQ}$ while the bus is transferred to the processor. The processor is always maintaining a weak pullup on the $\overline{REQ}$ signal, so the external system can drive $\overline{REQ}$ High actively before transferring the bus, then place the $\overline{REQ}$ driver into a high-impedance state while the bus is transferred to the processor. The weak pullup on the processor maintains the inactive level on $\overline{RDY}$, as long as the inactive level has been established before the external system driver is placed into the high-impedance state.

For systems using the $\overline{LOCK}$ output (LOADL, STOREL, LOADSET, or setting the CPS Lock Bit), the bus is not granted to an external master until both $\overline{REQ}$ and $\overline{LOCK}$ are High. The bus will not be granted when $\overline{LOCK}$ is asserted (see section 10.6).

The $\overline{BREQ}$ signal indicates that the processor may be requesting the bus. When an external system has control of the bus, $\overline{BREQ}$ indicates that the processor is stalled (data access pending or instruction cache miss), and is no longer able to execute without a bus access.

10.5 BUS SHARING—ELECTRICAL CONSIDERATIONS

When buses are shared among multiple masters and slaves, it is important to avoid situations where these devices are driving a bus at the same time. This may occur when more than one master or slave is allowed to drive a bus in the same cycle, because bus arbitration is incompletely or incorrectly performed. However, it also occurs when a master or slave releases a bus in the same cycle that another master or slave gains control, and the first master or slave is slow in disabling its bus drivers compared to the point at which the second master or slave begins to drive the bus. The latter situation is called a bus collision in the following discussion.

In addition to the logical errors that can occur when multiple devices drive a bus simultaneously, such situations may cause bus drivers to carry large amounts of electrical current. This can have a significant impact on driver reliability and power dissipation. Since bus collisions usually occur for a small amount of time, they are of less concern but may contribute to high-frequency electromagnetic emissions.

The Am29030 and Am29035 external interface is defined to prevent all situations where multiple drivers are driving a bus simultaneously. Bus collisions are also easily avoided.

In the case of the Am29030 and Am29035 external interface, arbitration prevents the processor from driving the Address and Data buses at the same time as another bus master. If there is more than one active device, the external system design must include some means for ensuring that only one device gains control of the bus and that no other device gains control of the external interface at the same time as the processor.

When the processor relinquishes control of the interface to another device, bus collisions may be prevented by not allowing the device to drive any bus during the cycle in which $\overline{REQ}$ de-asserts. This ensures that all processor outputs are disabled by the time the external master takes control of the bus. However, there is nothing in the bus protocol to prevent the external master from taking control as soon as $\overline{REQ}$ is de-asserted.

Bus collisions may be further prevented by restricting all devices to avoid driving the Instruction/Data Bus in the first cycle of a new simple or burst-mode access. Since the processor cannot sample data during this cycle, the cycle can be used to disable drivers of one slave before the drivers of another slave are enabled.

When the processor performs a store immediately following a load, it drives the Instruction/Data Bus for the store in the second cycle following the cycle in which the data for the load appears on the Instruction/Data Bus. This provides a complete cycle for the slave involved in the load to disable its data drivers. The processor continues to drive the Instruction/Data Bus until it receives a $\overline{RDY}$ in response to the store; it disables its output drivers in the cycle following the response.

10.6 MULTIPROCESSING AND THE $\overline{LOCK}$ OUTPUT

The $\overline{LOCK}$ output provides synchronization and exclusion of accesses in a multiprocessor environment. $\overline{LOCK}$ has no predefined effect on a system, other than the

fact that the processor does not relinquish the external interface to another device while $\overline{\text{LOCK}}$ is active.

The $\overline{\text{LOCK}}$ output is asserted for the address cycle of the Load and Lock and Store and Lock instructions and is asserted for both the read and write accesses of a Load and Set instruction. $\overline{\text{LOCK}}$ also may be active for an extended period of time under control of the Lock bit in the Current Processor Status Register (this capability is available only to Supervisor-mode programs).

$\overline{\text{LOCK}}$ may be defined to provide any level of resource locking for a particular system. For example, it may lock the interface, an individual device or memory, or a location within a device or memory.

When a resource is locked, it is available for access only by the processor with the appropriate access privilege. The mechanisms for restricting accesses and the methods for reporting attempted violations of the restrictions are system-dependent.

10.7 MASTER/SLAVE CHECKING

Each Am29030 and Am29035 microprocessor output has associated logic which compares the signal on the output with the signal that the processor is providing internally to the output driver. The comparison between the two signals is made any time a given driver is enabled and any time the driver is disabled only because of the Test mode. If, when the comparison is made, the output of a driver does not agree with its input, the processor asserts the MSERR output on the second following cycle.

When the processor asserts MSERR, it takes no other actions with respect to the detected miscomparison. In particular, no traps occur. However, MSERR may be used externally to perform any system function, including the generation of a trap.

10.7.1 Master/Slave Operation

If there is a single processor in the system, the MSERR output indicates that a processor driver is faulty or that there is a short-circuit in a processor output. However, a much higher level of fault detection is possible if a second processor (called a slave) is connected in parallel with the first (called a master), where the slave processor has its outputs disabled by the Test mode.

The slave processor, by comparing its outputs to the outputs of the master processor, performs a comprehensive check of the operation of the master processor. In addition, if the slave processor is connected at the proper position on the external interface, it may detect open circuits and other faults in the electrical path between the master processor and its local devices and memories. Note that the master processor still performs the comparison on its outputs in this configuration.

10.7.2 Preventing Spurious Errors

When two processors are connected in a master/slave configuration, it is necessary to prevent spurious assertions of MSERR. These result from situations where the outputs of the slave processor do not agree with the outputs of the master processor, but both processors are operating correctly.

There are several potential sources of spurious errors in a master/slave configuration that are avoided by the Am29030 and Am29035 microprocessor designs:

1. Unimplemented bits in processor registers that are reflected on processor outputs. This is avoided in the Am29030 and Am29035 microprocessors by having all unimplemented bits be read as 0.

2. Unpredictable values for bus signals. If $\overline{ERR}$ is asserted in response to an access, the Instruction/Data Bus may be at an indeterminate level (e.g., high-impedance), causing the master and slave processors to detect different values. If these values are later reflected on processor outputs, a spurious MSERR assertion may occur. The Am29030 and Am29035 microprocessors avoid this problem by ignoring the instruction or data word returned with $\overline{ERR}$.
3. Unpredictable power-up state that is reflected on processor outputs. The Am29030 and Am29035 microprocessors avoid this problem upon reset by forcing to a known value any state that might be reflected on outputs before the completion of initialization.

Another source of spurious errors is a lack of synchronization between the master and slave processors. To maintain synchronization between the master and slave processors, it is first necessary that they operate with identical clocks. This is accomplished by having the master processor drive MEMCLK, with the slave processor receiving MEMCLK as an input, or by driving both processors' MEMCLK inputs with the same externally generated clock.

However, the fact that both processors operate with the same MEMCLK clock is not sufficient to guarantee synchronization. Asynchronous processor inputs, if they are truly asynchronous to the operation of the master and slave processors, may affect the master processor a cycle sooner or later than they affect the slave processor. For this reason, the relevant asynchronous inputs (i.e., $\overline{WARN}$, $\overline{INTR}$(3–0), $\overline{TRAP}$(1–0), CNTL(1–0), and $\overline{RESET}$) must be externally synchronized to both the master and slave processors. Note that, in the case of $\overline{RESET}$, only the active-to-inactive transition must be synchronized.

10.7.3 Switching Master and Slave Processors

In some master/slave configurations, it might be desirable to give the slave processor control over the system when an error is isolated to the master processor. It is possible to grant control of the system to the slave processor by taking it out of the Test mode and placing the master processor into the Test Mode. Note that synchronization must be maintained when this is accomplished (e.g., using the Halt mode).

If the original master processor is configured to generate MEMCLK in this case, the slave processor must also generate MEMCLK when it becomes a master. Because of this, both processors must be configured to generate MEMCLK.

In this master/slave configuration, the slave processor still receives MEMCLK from the master processor as described previously. The slave processor does not drive MEMCLK because of the Test mode. However, when the slave processor is taken out of the Test mode, it is able to drive MEMCLK as required.

Note that this processor-switching scheme may be generalized to more than two processors.

CHAPTER 11

DEBUGGING AND TESTING

This chapter details the features of the Am29030 and Am29035 microprocessors that support debugging and testing. The chapter first describes the Trace Facility and instruction breakpoints which aid in software debugging. Next, the Test/Development Interface is described. This interface includes the processor inputs CNTL(1–0) and $\overline{\text{TEST}}$ and the outputs STAT(2–0). Finally, the Test Access Port and the Boundary Scan Architecture is discussed.

11.1 TRACE FACILITY

Software debug is supported by the Trace Facility. The Trace Facility guarantees exactly one trap after the execution of any instruction in a program being tested. This allows a debug routine to follow the execution of instructions and to determine the state of the processor and system at the end of each instruction.

Tracing is controlled by the Trace Enable (TE) and Trace Pending (TP) bits of the Current Processor Status Register. The value of the TE bit is always copied into the TP bit when an instruction enters the write-back stage of the processor pipeline. A Trace trap occurs whenever the TP bit is 1. As with most traps, the Trace trap can be disabled only by the DA bit of the Current Processor Status Register.

In order to trace the execution of a program, the debug routine performs an interrupt return to cause the program to begin or resume execution. However, before the interrupt return is executed, the TE and TP bits of the Old Processor Status are set with the values 1 and 0, respectively. The interrupt return causes these bits to be copied into the TE and TP bits of the Current Processor Status.

When the target instruction of the interrupt return (whose address is contained in the Program Counter 1 Register when the interrupt return is executed) enters the write-back stage, the processor copies the value of the TE bit into the TP bit. Since the TP bit is a 1, a Trace trap occurs. This trap prevents any further instruction execution in the target routine until the interrupt is taken and the routine is resumed with an interrupt return. When the Trace trap is taken, the TE and TP bits are both reset automatically, preventing any further Trace traps.

Since the Trace Facility is managed by the Old and Current Processor Status registers, it operates properly in the event that the processor takes an interrupt or trap—unrelated to the Trace Facility—before the above trace sequence completes. When the unrelated interrupt or trap is taken, the state of the Trace Facility (i.e., the values of the TE and TP bits) is copied into the Old Processor Status from the Current Processor Status. The Trace Facility then resumes operation when the interrupted routine is restarted by an interrupt return.

Note that it is possible to cause a Trace trap by directly setting the TP and/or TE bits in the Current Processor Status Register. This may be accomplished only by a Supervisor-mode program.

11.2 INSTRUCTION BREAKPOINTS

The HALT instruction can be used as an instruction breakpoint by the hardware-development system. However, the HALT instruction normally is a privileged instruction, causing a Protection Violation trap upon attempted execution by a User-mode program. The hardware-development system can disable this Protection Violation by holding the CNTI(1–0) inputs at 10 during reset; this disables protection checking for HALT instructions until the next processor reset.

The Assert class of instructions and the Illegal Opcode trap can be used by software to implement instruction breakpoints. An instruction breakpoint is set by replacing an instruction with the Assert instruction or an illegal opcode in the program under test. When the breakpoint instruction is encountered, the instruction breakpoint causes a trap. The illegal opcode is preferred since the Program Counter 1 (PC1) points to the illegal opcode when the trap is taken, whereas PC1 points to the instruction following the breakpoint if an Assert is used.

11.3 PROCESSOR STATUS OUTPUTS

The STAT(2–0) outputs indicate certain information about processor modes along with other information about processor operation. STAT(2–0) may be used to provide feedback of processor behavior during normal processor operation and when the processor is under the control of a hardware-development system. The behavior of the STAT(2–0) outputs depends on the relative frequencies of the processor and the system.

The encoding of STAT(2–0) is as follows:

STAT2	STAT1	STAT0	Mode or Condition
0	0	0	Halt or Step Modes
0	0	1	Pipeline Hold Mode
0	1	0	Load Test Instruction Mode, Halt/Freeze
0	1	1	Wait Mode
1	0	0	Interrupt Return
1	0	1	Taking Interrupt or Trap
1	1	0	Non-Sequential Instruction Fetch
1	1	1	Executing Mode

When the external interface is running at the processor's internal frequency, the STAT(2–0) signals in any given cycle reflect the condition of the processor's execute stage on the previous cycle. Where the conditions listed above are not mutually exclusive, the condition listed first is the one reflected on STAT(2–0).

When the external interface is running at half the processor's internal frequency, two of the conditions reported by STAT(2–0) may apply during a single external cycle. Since the STAT(2–0) outputs can only reflect one of these conditions, the processor determines which conditions to report as shown in Figure 11-1. Where the conditions are not mutually exclusive, the condition listed first in the above table is the one reported.

If the processor's condition during the first cycle of the external cycle (when MEMCLK is High) is one of Interrupt Return, Taking an Interrupt or Trap, Non-Sequential Instruction Fetch, or Executing, and the condition on the second cycle is anything other than these, the condition in the first cycle is reported on STAT(2–0) in the next external cycle. If the condition during both the first and second cycle is one of Halt, Pipeline Hold, Load Test Instruction, or Wait, the condition in the second cycle is reported in

Figure 11-1 STAT Output Reporting with High-Frequency Interface

	Mode/Condition	Mode/Condition	STAT
1	IRET,Interrupt, Branch,Executing	Halt,Pipehold,Load Test Inst., Wait	IRET,Interrupt, Branch,Executing
2	Halt,Pipehold,Load Test Inst.,Wait	Halt,Pipehold,Load Test Inst., Wait	Halt,Pipehold,Load Test Inst.,Wait
3	Halt,Pipehold,Load Test Inst.,Wait, Executing	IRET,Interrupt,Branch	IRET,Interrupt,Branch

the next external cycle. If the condition during the second cycle is one of Interrupt Return, Taking an Interrupt or Trap, or Non-Sequential Instruction Fetch, this condition is reported in the next external cycle.

The rationale for reporting the conditions in this manner is that the conditions Interrupt Return, Taking Interrupt or Trap, and Non-Sequential Instruction Fetch are probably most important to the hardware-development system and should be reported on either cycle. Moreover, the conditions Halt, Pipeline Hold, Load Test Instruction and Wait probably last for several cycles and, for this reason, will be reported in cases where the hardware-development system needs to be aware of them.

The first cycle of a multi-cycle instruction (Load Multiple, Store Multiple, Interrupt Return, or Interrupt Return and Invalidate) is indicated as an "Executing Mode" cycle. When an interrupt or trap is taken, the first cycle is indicated as a "Taking Interrupt or Trap" cycle. Additional cycles of these multi-cycle operations are indicated as "Pipeline Hold" cycles.

A Low level on STAT2 indicates that the processor is idle and may be used as an indication of processor performance when the external interface operates at the processor's frequency. Since most processor instructions execute in a single cycle, and since extra cycles spent executing multiple-cycle operations are counted as Pipeline Hold cycles, a count of the number of cycles within a given time interval that the processor is not idle (i.e., a count of the number of cycles for which STAT2 is High) is a close approximation to the number of instructions executed within that interval and thus approximates the instruction-execution rate. The only source of error in this approximation are the cycles in which the processor takes an interrupt or trap. If desired, this source of error can be eliminated by fully decoding the STAT(2–0) outputs.

The STAT2 output also may be used to implement processor timeouts for reliability. For example, a Low level on STAT2 may be used to start a hardware timeout counter, with a High level resetting and stopping the counter. If the counter exceeds a maximum expected count of idle cycles for a system, it is likely that an error has occurred. This error can be reported by the $\overline{\text{WARN}}$ trap (see Section 8.4.1).

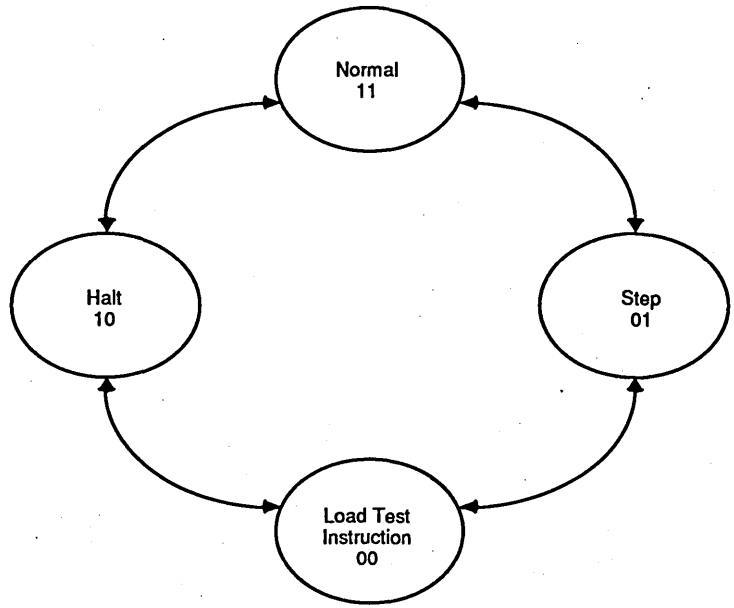
11.4 CPU CONTROL INPUTS

Certain processor operational modes are under control of the CNTL(1–0) inputs. These inputs affect the processor mode as follows:

CNTL1	CNTL0	Mode
0	0	Load Test Instruction
0	1	Step
1	0	Halt
1	1	Normal

These inputs are asynchronous to the processor clock. In addition, changes on the CNTL(1–0) inputs are restricted so that only CNTL1 or CNTL0, but not both, may change in any given processor cycle. The allowed transitions are shown in Figure 11-2. The restriction on transitions of CNTL(1–0) allows these inputs to be driven directly by an external hardware-development system or tester without any intervening logic. Proper operation is insured by making only single-input changes on CNTL(1–0) and by restricting the interval between all changes to be greater than a processor cycle. If these restrictions are violated, processor operation is unpredictable, and a processor reset is required to resume predictable operation.

Figure 11-2 Valid Transitions on CNTL(1–0) Inputs



Note that, because of the restriction described above, it is not possible to transition directly between all possible modes that are controlled by these inputs. For example, the processor cannot go from the Load Test Instruction mode to Normal operation without first entering the Halt or Step modes.

11.5 IMPLEMENTING A HARDWARE-DEVELOPMENT SYSTEM

The Halt, Step, and Load Test Instruction modes of operation are defined to support the debug of the processor system by a hardware-development system (both hardware and software debug). This section describes the use of these modes during debug and describes the corresponding activity on the CNTL(1–0) and STAT(2–0) lines.

11.5.1 Halt Mode

The Halt mode allows the hardware-development system to stop processor operation while preserving its internal state. The Halt mode is defined so that normal operation may resume from the point at which the processor enters the Halt mode. All external accesses are completed before the Halt mode is entered, so a minimum amount of system logic is required to support the Halt mode.

The Halt mode can be invoked by applying a value of 10 to the CNTL(1–0) inputs. The processor enters the Halt mode within two or three cycles after the CNTL(1–0) inputs are changed (depending on synchronization time), except that it first completes any external data access in progress.

The Halt mode can also be entered as the result of executing a HALT instruction . When a HALT instruction is executed, the processor enters the Halt mode on the next cycle, except that it completes any external data accesses in progress. In this case, the processor remains in the Halt mode even though the CNTL(1–0) inputs are 11. However, the processor cannot exit the Halt mode except as the result of the CNTL(1–0) or $\overline{\text{RESET}}$ inputs. If the instruction following a Halt instruction has an exception (e.g., instruction TLB Miss), the trap associated with the exception is taken before the processor enters the Halt mode.

The Halt instruction is designed to be used as an instruction breakpoint by the hardware-development system. However, the Halt instruction normally is a privileged instruction, causing a Protection Violation trap upon attempted execution by a User-mode program. The hardware-development system can disable this Protection Violation by holding the CNTL(1–0) inputs at 10 during a reset; this signals the presence of an external debugger and disables protection checking for Halt instructions until the next processor reset.

In most cases, the STAT(2–0) outputs have a value of 000 whenever the processor is in the Halt mode; these outputs can be used as a verification that the processor is in Halt mode. However, the STAT(2–0) outputs have a value of 010 if the Freeze (FZ) bit of the Current Processor Status Register is 1 when the Halt mode is entered. This indicates that visible registers do not reflect the current program state.

While in the Halt mode, the processor does not execute instructions and performs no external accesses. The Timer Facility does not operate (i.e., the Timer Counter Register does not change).

The Halt mode is exited whenever the Reset mode is entered or the CNTL(1–0) lines place the processor into another mode. The only valid transitions on the CNTL(1–0) lines from the value of 10 are to the value 00, which places the processor into the Load Test Instruction mode, or to the value 11, which causes the processor to resume normal execution.

11.5.2 Step Mode

The Step mode causes the Am29030 and Am29035 microprocessors to execute at a rate determined by the hardware-development system, allowing the hardware-

development system to easily control and monitor processor operation. The Step mode is defined so that normal operation may resume after stepping is complete. Since all external accesses are completed during any step, a minimum amount of system logic is required to support the slower rate of execution.

The Step mode is invoked by the application of a value of 01 to the CNTL(1–0) inputs. The processor enters the Step mode within two or three cycles after the CNTL(1–0) inputs are changed (depending on synchronization time), except that it first completes any external data access in progress.

In most cases, the STAT(2–0) outputs have a value of 000 whenever the processor is in the Step mode; these outputs can be used as a verification that the processor is in Step mode. However, the STAT(2–0) outputs have a value of 010 if the Freeze (FZ) bit of the Current Processor Status Register is 1 when the Step mode is entered. This indicates that visible registers do not reflect the current program state.

While in the Step mode, the processor does not execute instructions and performs no external accesses. The Timer Facility does not operate (i.e., the Timer Counter Register does not change) while the processor is in the Step mode.

The Step mode is identical to the Halt mode in every respect except one. This difference is apparent on the transition of the CNTL(1–0) lines from the value 01 (Step mode) to the value 11 (Normal). On this transition, the processor steps. That is, the processor state advances by one pipeline stage, and it completes any external access which is initiated by this state change.

If the processor immediately enters the Pipeline Hold mode on a step, the step may require multiple cycles to execute, since the processor pipeline cannot advance while the processor is in the Pipeline Hold mode. The STAT(2–0) lines reflect the state of the processor for every cycle of the step; STAT2 is High for one cycle, and only one cycle, before the step completes.

The Timer Counter decrements by one for every cycle of the step; if the Timer Counter decrements to zero, the usual Timer-Facility actions are performed, and a Timer interrupt may occur.

After the step is performed, the processor re-enters the Step mode and remains in the Step mode even though the CNTL(1–0) inputs have the value 11 (this prevents the need for a time-critical transition on the CNTL(1–0) inputs). The processor remains in this condition until the CNTL(1–0) inputs transition to 10 or 01 (or $\overline{\text{RESET}}$ is asserted). The transition to 10 causes the processor to enter the Halt mode and is used to clear the Step mode. The transition to 01 causes the processor to remain in the Step mode, so that it may perform additional steps.

If the processor is placed in the Halt or Step mode while either a LOADM or STOREM instruction is being executed, the STAT(2–0) outputs indicate the Halt or Step mode for one cycle (STAT(2–0) = 000). They then indicate the Pipeline Hold mode (STAT(2–0) = 001) until the final access of the LOADM or STOREM is complete, at which time they return to indicating the Halt or Step mode. A hardware-development system must therefore ignore any single-cycle Halt/Step mode indication on the STAT(2–0) outputs as an indication that the processor is halted.

11.5.3 Load Test Instruction Mode

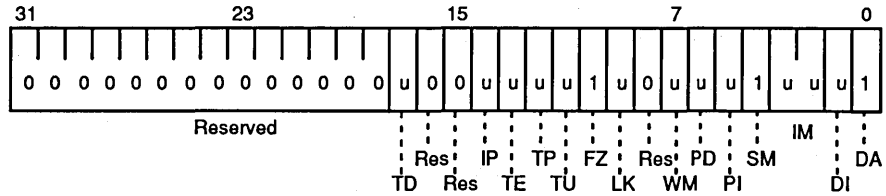
The processor incorporates an Instruction Register (IR) that holds instructions while they are decoded. In the Load Test Instruction mode, the IR is enabled to receive the content of the Instruction Bus regardless of the state of the processor's instruction fetcher. This allows the hardware-development system to provide instructions for

execution directly, thereby providing means for the hardware-development system to examine and modify the internal state of the processor without altering the processor's instruction stream.

The hardware-development system can place an instruction in the IR by first placing 00 on CNTL(1–0). The processor enters the Load Test Instruction mode within two or three cycles after the CNTL(1–0) inputs are changed (depending on synchronization time), but it first completes and terminates any established burst-mode instruction access. The Load Test Instruction mode can be entered only from the Halt or Step modes.

When the processor enters the Load Test Instruction Mode, the processor behaves as though the Current Processor Status Register were forced to the value shown in Figure 11-3, even though the register is not changed (the value "u" means unaffected).

Figure 11-3 Processor Status While in Load Test Instruction Mode



The visible processor state remains unchanged while the processor is in the Load Test Instruction Mode. The processor status shown in Figure 11-3 remains in effect until the next transition to the Normal Mode via the Halt Mode.

While the processor is in the Load Test Instruction mode, it ignores all interrupts and traps, except for the Data Access Exception and the $\overline{\text{RESET}}$ and $\overline{\text{WARN}}$ inputs.

The STAT(2–0) lines have a value of 010 while the processor is in the Load Test Instruction mode; this may be used as a verification that the processor is loading the IR.

While the processor is in the Load Test Instruction mode, the IR is continually storing the value on the Instruction/Data Bus; any change in the value on this bus is reflected in the IR on the next cycle. The hardware-development system can place a desired instruction into the IR by driving this instruction on the Instruction/Data Bus. The value of $\overline{\text{RDY}}$ and $\overline{\text{ERR}}$ are irrelevant.

The processor exits the Load Test Instruction mode in the second cycle following a change on the CNTL(1–0) inputs. The only valid change here is either to the Halt mode (CNTL(1–0) = 10) or the Step mode (CNTL(1–0) = 01).

When the Load Test Instruction mode is exited, the most recent value stored into the IR is held. If the processor is placed in the Step mode, the IR is marked as having valid content, enabling the processor to decode and execute the instruction. If the processor is placed in the Halt mode, it ignores any instruction placed in the IR by the Load Test Instruction mode and reverts to its normal instruction-fetch mechanism.

Once the IR has been set by the Load Test Instruction mode, the instruction in the IR may be executed via the Step mode as discussed in the previous section. A single step is sufficient to cause the execution of this instruction. However, because of pipelining, multiple steps may be required before the instruction completes execution. If more than one step is performed, the processor executes the instruction in the IR on

every step. If it is desired to step an instruction to completion without repeated execution, a NO-OP may be set into the IR (using the Load Test Instruction mode) after the first step.

The Load Test Instruction mode may be used to cause the execution of most processor instructions (restrictions are discussed below). This allows inspection and modification of the processor state.

The hardware-development system uses load and store instructions, executed via the Load Test Instruction mode, to alter and inspect the contents of general-purpose registers. The OPT field for these loads and stores have the value 110: this causes the system to ignore the resulting access. Furthermore, it causes the Am29030 and Am29035 microprocessors to ignore the $\overline{\text{RDY}}$ and $\overline{\text{ERR}}$ responses for the access; the Am29030 and Am29035 microprocessors complete the access at the end of the next stepped instruction, rather than upon the assertion of $\overline{\text{RDY}}$. This eliminates the need for the hardware-development system to generate a synchronous $\overline{\text{RDY}}$ in response to the load or store.

Because of sequencing constraints, the Load Test Instruction mode cannot be used to cause the execution of the following instructions: conditional jumps, Load Multiple, Store Multiple, Interrupt Return, and Interrupt Return and Invalidate. Unconditional jumps and calls are permitted, but affect only the Program Counter (instruction sequencing is not affected).

It is not possible to execute a load directly following a store—nor a store directly following a load—using the Load Test Instruction mode. At least one NO-OP (or other operation) must be executed between adjacent loads and stores, because of control conflicts that arise when these instructions are stepped in a system that performs the resulting accesses at normal speed. However, a sequence of only loads or only stores is permitted without restriction.

The contents of the Program Counter 0, Program Counter 1, Program Counter 2, Channel Address, Channel Data, Channel Control, and ALU Status registers are not updated while instructions are executed via the Load Test Instruction mode, except explicitly by Move To Special Register instructions. Instructions executed using the Load Test Instruction mode may access the protected processor state even though the processor is in the User mode.

Instructions executed via the Load Test Instruction mode may be used to access an external device or memory. Recall that the processor completes any normal data access before completing a step. This allows the processor to access devices and memories on behalf of the hardware-development system and simplifies the timing constraints on the hardware-development system.

During processor execution via the Load Test Instruction mode, the processor retains the information required to resume normal operation. If any processor state is modified by the hardware-development system, this state must be restored properly for normal operation to resume properly.

Once all instructions have been executed via the Load Test Instruction mode, the Halt mode (CNTL(1–0)=10) prepares the processor to resume normal operation. When the CNTL(1–0) inputs transition to 11, the processor resumes normal operation. The sequence for the CNTL(1–0) inputs to clear the Load Test Instruction mode and resume normal operation is thus 00/10/11.

11.5.4 Summary Of Development System Operation

When the capabilities provided by the Halt, Step, and Load Test Instruction Register modes are combined, an extremely flexible test and development interface results. The following is an example sequence performed by the hardware-development system during debug:

1. Halt the processor either by a HALT instruction or by a 10 on the CNTL(1–0) inputs. The HALT instruction may be used as a primitive operation in the implementation of a general instruction-breakpoint capability.
2. Load the IR with an instruction to inspect or alter the processor state. The hardware-development system should wait for the value 010 on STAT(2–0) (Load Test Instruction mode) before driving the Instruction Bus. After the IR is loaded, the hardware-development system sets CNTL(1–0) to 01 (Step mode).
3. Step the processor by a transition of CNTL(1–0) from 01 to 11 and back to 01. Data may be supplied on the Data Bus during one of the steps to satisfy a load operation; the data must be held valid until the stepped instruction completes.
4. Repeat steps 2 and 3 as desired.
5. After the final step, enter the Halt mode by placing 10, instead of 01, on CNTL(1–0).
6. Resume normal execution by placing 11 on CNTL(1–0).

11.6 IN-CIRCUIT TESTING

The Test mode in the Am29030 and Am29035 microprocessors allows processor outputs to be driven directly for testing or diagnostic purposes. The Test mode places all processor outputs (except MSERR) into the high-impedance state, so that they do not interfere electrically with externally supplied signals. In all other respects, processor operation is unchanged.

The Test mode is invoked by an active level on the $\overline{\text{TEST}}$ input, regardless of the processor's operational mode (for example, the Test mode is not affected by the Halt mode). The disabling of processor outputs is performed combinatorially and is asynchronous to MEMCLK.

For some outputs, the transition to the high-impedance state that results from the Test mode may occur at a much slower rate than that which applies during normal system operation (for example, when the processor relinquishes the bus to another master). For this reason, the Test mode may not be appropriate for special user-defined purposes.

Note that MEMCLK is also placed in the high-impedance state by the Test mode. This allows the testing of external clock-distribution circuits, but care must be taken to insure that a high-impedance MEMCLK output does not have an adverse effect on the system.

11.7 TEST ACCESS PORT

The Am29030 and Am29035 microprocessors implement the Standard Test Access Port (TAP) and Boundary-Scan Architecture as specified by the IEEE Specification 1149.1–1990 (JTAG), with the exception that the INCLK pin is not part of the boundary-scan register. The 1149.1–1990 Specification includes many details that are omitted from the discussion in this section and are included by reference. The following description discusses Am29030 and Am29035 microprocessor-specific considerations.

11.7.1 Boundary Scan Cells

The Test Access Port can access, affect, and sample the processor inputs and outputs because a Boundary Scan Register (BSR) and Parallel Data Register (PDR) are incorporated into the design of the input and output cells. The Boundary Scan Register allows data to be serially loaded into or read out of the processor input/output boundary. The Parallel Data Register holds data stable at inputs and outputs during scanning, so that system signals are not adversely affected during scanning.

An input or output cell incorporating a BSR and PDR register bit is referred to as a boundary scan cell. There are many possible configurations of boundary scan cells. The configuration of the Am29030 and Am29035 processors is designed to allow certain operations to be performed. This section describes the implementation of the Am29030 and Am29035 boundary scan cells.

Figure 11-4 shows the design of an input boundary scan cell, and Figure 11-5 shows the design of an output boundary scan cell. Bi-directional signals use both of these designs in the same cell. Multiplexor selects, when active, select the lower multiplexor input.

The Shift and Update clocks, when used to sample or drive processor and system signals, are synchronized to the processor internal clocks so that all signals (except the TAP signals) are sampled or driven synchronously to system clocks. However, the Shift and Update clocks still satisfy the JTAG constraints that inputs are sampled after the rising edge of TCK and that outputs change after the falling edge of TCK and that TCK is the only control needed to affect sampling and driving.

The 1149.1–1990 Specification requires that it be possible to force the processor 3-state outputs to be enabled. This is accomplished by cells that have no associated input pin. The outputs of these cells force groups of output drivers to be enabled. The requirement to disable all outputs is satisfied by the boundary scan cell for the $\overline{\text{TEST}}$ input. The MSERR output has an additional bit in the BSR and PDR to control the 3-state enable on the MSERR output.

Figure 11-4 Input Boundary-Scan Cell

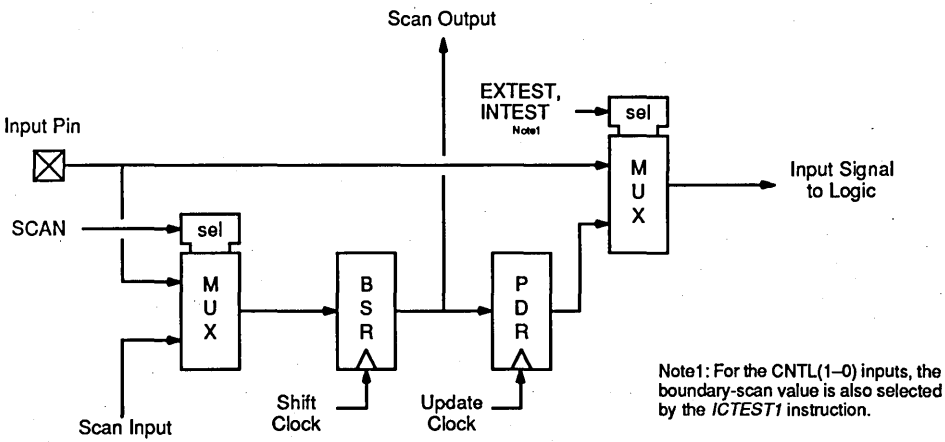
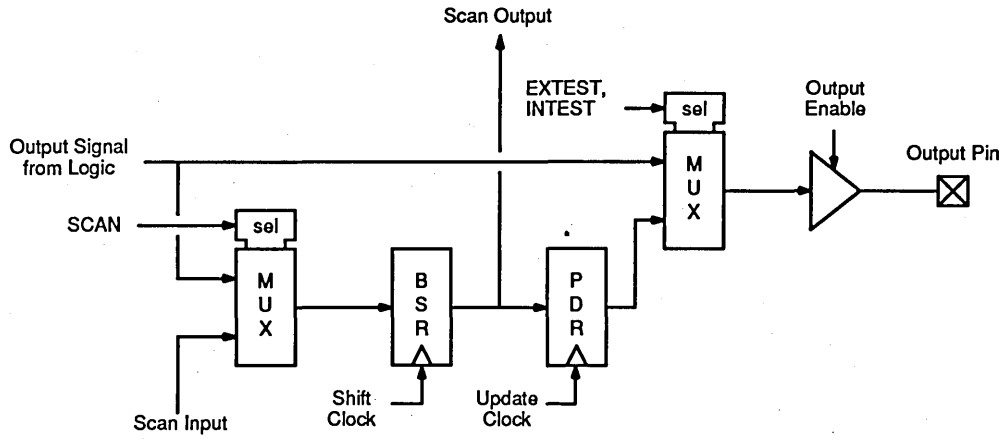


Figure 11-5 Output Boundary-Scan Cell



The boundary scan cells for the CNTL(1–0) inputs and STAT(2–0) outputs are part of the BSR and are accessible by scanning the BSR. However, they also can be scanned individually using the *ICTEST1* instruction (see Section 11.7.2). If the *ICTEST1* instruction is active, no other boundary scan cell is scanned. However, the contents of the other scan cells are undefined after this operation.

The INCLK input is not a boundary scan cell. The clocks to the processor must continue to operate even if the Test Access Port is active. However, a fault on this input is readily visible in the operation of the Test Access Port.

The MEMCLK pin has both an input and an output boundary scan cell—the input or output cell is selected based on whether MEMCLK is used as an input or an output. Because of electrical constraints, when MEMCLK is used as an input, the boundary scan cell can sample the level driven on the MEMCLK pin but cannot drive the internal MEMCLK signal. The internal MEMCLK is driven by the MEMCLK pin alone. If MEMCLK is used as an output, the *EXTEST* and *INTEST* instructions hold the MEMCLK signal at a single logic level for the duration of the instruction, resulting in unpredictable processor behavior.

11.7.2 Instruction Register and Implemented Instructions

The Instruction Register (IREG) of the Test Access Port is a 3-bit register. The least-significant bit (IREG0) is the bit nearest the TDO output. Instructions are encoded as follows:

IREG2	IREG1	IREG0	Instruction
0	0	0	EXTEST
0	0	1	preloaded value (acts like BYASS)
0	1	0	ICTEST2
0	1	1	reserved (acts like BYPASS)
1	0	0	INTEST
1	0	1	SAMPLE
1	1	0	ICTEST1
1	1	1	BYPASS

The *EXTEST, BYPASS, INTEST,* and *SAMPLE* instructions are specified by the 1149.1–1990 Specification. Reserved instructions behave as *BYPASS* instructions to conform to the specification. *ICTEST1 and ICTEST2* are AMD public instructions.

Most of these instructions are described in detail in 1149.1–1990. Below is a brief description of the special considerations in the Am29030 and Am29035 microprocessor implementations.

11.7.2.1 EXTEST

The *EXTEST* instruction is provided for external continuity and logic tests. It allows the Test Access Port to drive outputs and sample inputs.

EXTEST selects the BSR for scanning. During execution:

1. Processor outputs are driven from the PDR.
2. Processor internal output signals are sampled into the BSR. This is default behavior.
3. Processor inputs are sampled into the BSR.
4. Processor internal input signals are driven from the PDR. This prevents internal logic from seeing invalid combinations of input signals that may be received from other chips during the test.

11.7.2.2 INTEST

The *INTEST* instruction is provided to test the processor's internal logic. Its primary value is to allow a hardware-development system to drive the processor's Test Interface without a direct electrical connection to all pins of the package. Due to the restrictions on the MEMCLK signal, INTEST may be performed only on a system in which the MEMCLK signal is an input.

INTEST selects the BSR for scanning. During execution:

1. Processor outputs are driven from the PDR. This prevents external logic from seeing invalid combinations of output signals.
2. Processor internal output signals are sampled into the BSR.
3. Processor inputs are sampled into the BSR. This is default behavior.
4. Processor internal input signals are driven from the PDR.

Note that the *INTEST* instruction allows the hardware-development system to alter and inspect internal registers, using processor load and store instructions, without having the external system see any bus activity. This eliminates the need for the special OPT code point (OPT(2–0)=110), though this code point is still reserved for other types of hardware-development systems.

11.7.2.3 SAMPLE

The *SAMPLE* instruction is provided to inspect the processor's external signals without interfering with system operations.

SAMPLE selects the BSR for scanning. During execution:

1. Processor outputs are driven by the processor.
2. Processor internal output signals are sampled into the BSR.
3. Processor inputs are sampled into the BSR.
4. Processor internal input signals are driven from the processor inputs.

11.7.2.4 ICTEST1

The *ICTEST1* instruction is defined for AMD processors using the extension mechanisms permitted by 1149.1–1990. It is provided to drive the CNTL(1–0) inputs and sample the STAT(2–0) outputs while leaving other inputs and outputs in their normal system connection. This allows a hardware-development system to control the processor and system using the Test Access Port.

ICTEST1 selects a subset of the BSR for scanning. During execution:

1. Processor outputs are driven by the processor.
2. Processor internal output signals are sampled into the BSR. This is default behavior.
3. Processor input signals are sampled into the BSR. This is default behavior for most signals, but allows the sampling of STAT(2–0).
4. Processor internal inputs for CNTL(1–0) are driven by the PDR. Processor internal inputs for inputs other than CNTL(1–0) are driven from the processor inputs.

11.7.2.5 ICTEST2

The *ICTEST2* instruction is defined for AMD processors using the extension mechanisms permitted by 1149.1–1990. *ICTEST2* is similar to *EXTEST* with the exception that the scan path for *ICTEST2* does not include the MEMCLK scan cell. It is provided for external continuity and logic tests without requiring that the tester be concerned with MEMCLK. It also allows a hardware-development system to access and modify processor internal state without disrupting the system.

1. Processor outputs are driven from the PDR. This allows the hardware-development system to keep the external system in a valid quiescient state.
2. Processor internal output signals are sampled into the BSR. This is default behavior.
3. Processor inputs are sampled into the BSR. This is the default behavior.
4. Processor internal input signals are driven from the PDR. This allows a hardware-development system to control the processor independent of the system controls.

11.7.2.6 BYPASS

The *BYPASS* instruction is provided to bypass the BSR and shorten access times to other devices at the board level.

BYPASS selects the Bypass Register for scanning. The processor is not otherwise affected.

11.7.3 Order of Scan Cells in Boundary Scan Path

This section documents the scan paths and the order of scan cells in the paths. The cells are listed in order from TDI to TDO. In the Am29030 and Am29035 microprocessor, there are five scan paths from TDI to TDO: 1) the instruction path, 2) the bypass path, 3) the main data path, 4) the *ICTEST1* path, and 5) the *ICTEST2* path.

11.7.3.1 INSTRUCTION PATH

This is a 3-bit path which is used to scan into the Instruction Register. When the instruction path is selected, the captured data always is IREG(2–0) = 001 and the instruction is set by scanning. The preloaded pattern 001 is used to test for faults in the boundary scan connections at the board level. The instructions are specified in Section 11.7.2.

Bit	Cell Name
1	IREG2
2	IREG1
3	IREG0

11.7.3.2 BYPASS PATH

This is a one-bit path which is used to bypass the processor and shorten access to other devices at the board level. When the bypass path is selected, the captured data is always 0 and the scan in data has no effect on the processor.

11.7.3.3 MAIN DATA PATH

This is a 141-bit path used to access the processor pins. This path is divided into five sets of cells. Each set has a cell which enables the outputs of the set to be driven on the processor's pins. These cells are not connected to a processor pin. For convenience, the drive enable cells are shown in italics. The sets of cells are divided logically as follows: 1) instruction/data bus, 2) address bus, 3) control signals, 4) MEMCLK, and 5) MSERR and $\overline{\text{BREQ}}$ outputs. Note that sets 3 and 5 differ in that when the processor is not granted the bus, set 3 signals are disabled and set 5 signals are enabled.

Bit	Cell Name	Comments
1	*ENDDRV*	Enables the driving of the ID(31–0) outputs
2	ID0	input
3	ID0	output
4	ID1	input
5	ID1	output
.	.	
.	.	
63	ID31	input
64	ID31	output
65	A31	
66	A30	
.	.	
.	.	
95	A2	

Bit	Cell Name	Comments
96	*ENADRV*	Enables the driving of the A(31–0) outputs
97	*ENCDRV*	Enables the driving of the control outputs
98	A1	
99	$\overline{A0}$	
100	$\overline{RDN}$	
101	$\overline{ERLYA}$	
102	$\overline{ERR}$	
103	$\overline{RDY}$	
104	$\overline{BGRT}$	
105	$\overline{PGMODE}$	
106	$\overline{BURST}$	
107	$\overline{REQ}$	
108	$\overline{BREQ}$	
109	*PWRCLK*	If the PWRCLK pin is connected to Vcc, value of the PWRCLK scan cell is used to enable or disable MEMCLK. If the PWRCLK scan cell is 1, MEMCLK is an output. If the PWRCLK scan cell is 0, MEMCLK is disabled. If the PWRCLK pin is connected to ground, the MEMCLK driver is disabled.
110	MEMCLK	Input or Output: When the PWRCLK scan cell is 1 and MEMCLK is enabled, the MEMCLK scan cell functions as an output scan cell: it captures processor internal MEMCLK and substitutes the scanned value for the output. When the PWRCLK scan cell is 0, the MEMCLK scan cell functions as a partial input: it samples the MEMCLK input pin, but is unable to substitute the scanned value for the internal clock.
111	$\overline{DIV2}$	
112	CNTL0	
113	$\overline{CNTL1}$	
114	$\overline{RESET}$	
115	$\overline{TRAP0}$	
116	$\overline{TRAP1}$	
117	$\overline{INTR3}$	
118	$\overline{INTR2}$	
119	$\overline{INTR1}$	
120	$\overline{INTR0}$	
121	*ENAMS*	Enables the driving of MSERR and $\overline{BREQ}$ ($\overline{BREQ}$ is not enabled if the processor is in TEST mode).
122	MSERR	
123	STAT2	
124	STAT1	
125	STAT0	
126	MPGM1	
127	MPGM0	
128	OPT2	
129	OPT1	
130	$\overline{OPT0}$	
131	$\overline{LOCK}$	
132	$\overline{SUP}$/US	
133	$\overline{BWE3}$	
134	$\overline{BWE2}$	
135	$\overline{BWE1}$	
136	$\overline{BWE0}$	
137	$\overline{WARN}$	
138	IO/$\overline{MEM}$	
139	$\overline{I/D}$	
140	R/$\overline{W}$	
141	$\overline{TEST}$	

11.7.3.4 ICTEST1 PATH

This is a 5-bit path which is used to provide quick access to the CNTL(1–0) input signals and STAT(2–0) output signals and to allow the other inputs and outputs to remain in their normal system connection.

Bit	Cell Name	Comments
1	CNTL0	input
2	CNTL1	
3	STAT2	output: These signals are scanned out and are shown on the TDO pin. The scan in values do not replace the processor output values. In ICTEST1, the processor outputs STAT(2–0) continue to reflect the internal processor signals.
4	STAT1	
5	STAT0	

If the *ICTEST1* path is scanned, the contents of the shift register bits in the other scan cells become undefined. This occurs because all scan paths share the same shift clocks

11.7.3.5 ICTEST2 PATH

The *ICTEST2* path is the same as the main data path with the exception that MEMCLK is not included. This allows a hardware-development system to control the processor while the system is unaffected.

INSTRUCTION SET

This chapter provides a specification of the Am29030 and Am29035 instruction set. Sections 12.1 through 12.2 describe the terminology and the instruction formats. Section 12.3 describes each instruction in detail; instructions are presented alphabetically by assembler mnemonic. Finally, Section 12.4 gives an index of instructions by operation code.

12.1 INSTRUCTION-DESCRIPTION NOMENCLATURE

To simplify the specification of the instruction set, special terminology is used throughout this chapter. This section defines the terminology and symbols used to describe instruction operands, operations, and the assembly-language syntax.

This section does not describe all terminology used. It excludes certain descriptive terms that have an obvious meaning.

12.1.1 Operand Notation and Symbols

Throughout this chapter, instruction operands are signed, two's-complement, word integers, unless otherwise noted. The term register is used consistently to denote a general-purpose register; other types of registers are described explicitly.

The following notation is used in the description of instruction operands:

0I16	16-bit immediate data, zero-extended to 32 bits.
1I16	16-bit immediate data, one-extended to 32 bits.
BP	The Byte Pointer (BP) field of the ALU Status Register. The BP field selects a byte or half-word within a word, and is interpreted according to the Byte Order bit of the configuration Register.
C	The Carry (C) bit of the ALU Status Register. The C bit is logically zero-extended to 32 bits when it is involved in a word operation.
COUNT	The value of the Count Remaining field of the Channel Control Register. Note that COUNT does not refer to this field directly, but rather to the value of the field at the beginning of a LOADM or STOREM instruction.
DEST	The general-purpose register that is the destination of an instruction (i.e., the register used to store the result).
EXTERNAL WORD[*n*]	The word in an external device or memory with address *n*.
FALSE	The Boolean constant FALSE.
FC	The Funnel Shift Count (FC) field of the ALU Status Register.
h'*n*'	The hexadecimal constant *n*.
I16	16-bit immediate data.

IPA	Indirect Pointer A Register.
IPB	Indirect Pointer B Register.
IPC	Indirect Pointer C Register.
PC	The Program Counter Register. This register is not explicitly accessible by instruction, but does appear as an operand for certain instructions. The Program Counter always contains the word address of the instruction being executed, and is 30 bits in length.
Q	The Q Register.
Register RA Register RB Register RC	These designate the general-purpose registers specified by the instruction fields RA, RB, and RC (see Section 12.2).
SPDEST	The special-purpose register that is the destination of an instruction.
SPECIAL	The contents of a special-purpose register, used as an instruction operand.
Special-purpose Register SA	Designates the special-purpose register specified by the instruction field SA (see Section 12.2).
SRCA SRCB	The contents of general-purpose registers, used as instruction operands.
SRCA.BYTE*n* SRCB.BYTE*n*	Designate the byte numbered n within the SRCA or SRCB operand.
TARGET	The target-instruction address specified by a jump or call instruction. This address is either absolute or Program-Counter relative.
TLB[*n*]	The Translation Look-Aside Buffer Register with register number *n*.
TRUE	The Boolean constant TRUE.
TWIN	General-purpose registers are paired by absolute-register number, such that even-numbered registers are paired with odd-numbered registers having the next-highest register number. The twin of a given register is the other register in the pair to which the given register belongs. For example, Local Register 5 is the twin of Local Register 4, and vice versa.

12.1.2 Operator Symbols

The following symbols are used to describe instruction operations:

A << B	Left shift of the A operand by the shift amount given by the B operand.
A >> B	Right shift of the A operand by the shift amount given by the B operand.
A // B	Concatenation. The B operand is appended to the A operand. In the resulting quantity, the A operand makes up the high-order part, and the B operand makes up the low-order part.
A & B	Bitwise AND.
A | B	Bitwise OR.

A ^ B	Bitwise exclusive-OR.
~ A	One's-complement.
A ← exp	Assignment of the A location by the result of the expression on the right side.
A = B	Equal to.
A <> B	Not equal to.
A > B	Greater than.
A ≥ B	Greater than or equal to.
A < B	Less than.
A ≤ B	Less than or equal to.
A + B	Addition.
A − B	Subtraction.
A * B	Multiplication.
A / B	Division.
A .. B	A subrange which includes the A operand and the B operand. This symbol is used for subranges of bits as well as subranges of words.
A OR B	Logical OR of two Boolean conditions.

12.1.3 Control-Flow Terminology

The following terminology is used to describe the control functions performed during the execution of various instructions:

Continue	Continue execution of the current instruction sequence.
IF condition THEN operations ELSE operations	The condition following the IF is tested. If the condition holds, the operations following the THEN are performed. If the condition does not hold, the operations following the ELSE are performed. If the ELSE is not present and the condition does not hold, no operation is performed.
Signed overflow	This condition is present when the result of an add or subtract of two's-complement operands cannot be represented by a signed word integer.
Trap(*n*)	Specifies a trap with vector number *n*. The vector number *n* may be specified indirectly (e.g., Trap (VN)) or explicitly by symbolic name (e.g., Trap (Out of Range)).
Unsigned overflow	This condition is present when the result of an add of unsigned operands cannot be represented by an unsigned word integer.
Unsigned underflow	This condition is present when the result of a subtract of unsigned operands cannot be represented by an unsigned integer (i.e., when the result is less than zero).
VN	Designates the trap vector number specified by the instruction field VN (see Section 8.2.2).

12.1.4 Assembler Syntax

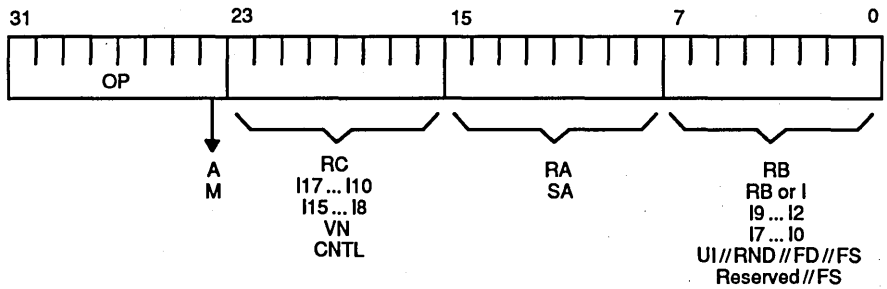
This chapter does not contain a full description of the instruction assembler, but provides a rudimentary description of the assembler syntax. The following notation is used to describe assembler tokens:

cntl	Determines the 7-bit control field in a load or store instruction.
const8	Specifies a constant that can be expressed by 8 bits.
const16	Specifies a constant that can be expressed by 16 bits.
ra rb rc	These tokens name general-purpose registers. In a formal sense, these represent the same token, since the name of a register does not depend on its instruction use. However, three distinct tokens are used to clarify the relationship between the assembler syntax, instruction operands, and instruction fields.
spid	A symbolic identifier for a special-purpose register.
target	A symbolic label for the target of a jump or call instruction.
vn	Specifies a trap vector number.

12.2 INSTRUCTION FORMATS

All instructions for the Am29030 and Am29035 microprocessors are 32 bits in length and are divided into four fields, as shown in Figure 12-1. These fields have several alternative definitions, as discussed below. In certain instructions, one or more fields are not used, and are reserved for future use. Even though they have no effect on processor operation, bits in reserved fields should be 0 to insure compatibility with future processor versions.

Figure 12-1 Instruction Format



The instruction fields are defined as follows:

Bits 31–24

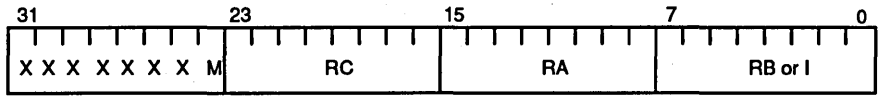
OP	This field contains an operation code, defining the operation to be performed. In some instructions, the least-significant bit of the operation code selects between two possible operands. For this reason, the least-significant bit is sometimes labeled A or M with the following interpretations:
A	(Absolute): The A bit is used to differentiate between Program-Counter relative (A = 0) and absolute (A = 1) instruction addresses, when these addresses appear within instructions.

M	(Immediate): The M bit selects between a register operand (M = 0) and an immediate operand (M = 1), when the alternative is allowed by an instruction.
Bits 23–16	
RC	The RC field contains a global or local register number.
I17 ... I10	This field contains the most-significant eight bits of a 16-bit instruction address. This is a word address, and may be program-counter relative or absolute, depending on the A bit of the operation code.
I15 ... I8	This field contains the most-significant eight bits of a 16-bit instruction constant.
VN	This field contains an 8-bit trap vector number.
CNTL	This field controls a load or store access, as described in Section 3.3.2.
Bits 15–8	
RA	The RA field contains a global or local register number.
SA	The SA field contains a special-purpose register number.
Bits 7–0	
RB	The RB field contains a global or local register number.
RB or I	This field contains either a global or local register number, or an 8-bit instruction constant, depending on the value of the M bit of the operation code.
I9 ... I2	This field contains the least-significant eight bits of a 16-bit instruction address. This is a word address, and may be program-counter relative or absolute, depending on the A bit of the operation code.
I7 ... I0	This field contains the least-significant eight bits of a 16-bit instruction constant.
UI // RND // FD // FS	This field controls the operation of the CONVERT instruction.
reserved // FS	This field is the FS portion of the above field and specifies the operand format for the CLASS and SQRT instructions.

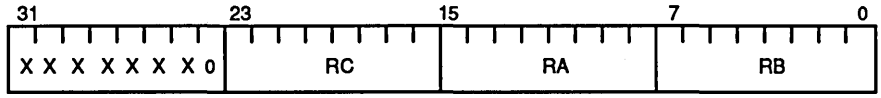
The fields described above may appear in many combinations. However, certain combinations that appear frequently are shown in Figure 12-2.

Figure 12-2 Frequently Occurring Instruction Field Uses

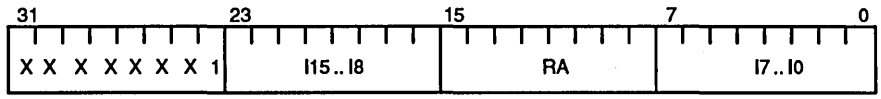
Three operands, with possible 8-bit constant:



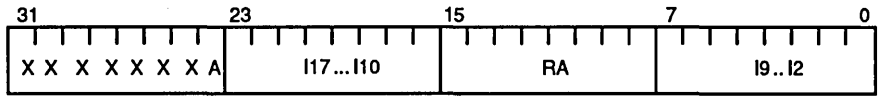
Three operands, without constant:



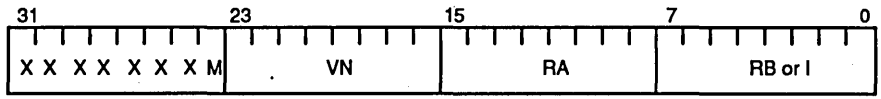
One register operand, with 16-bit constant:



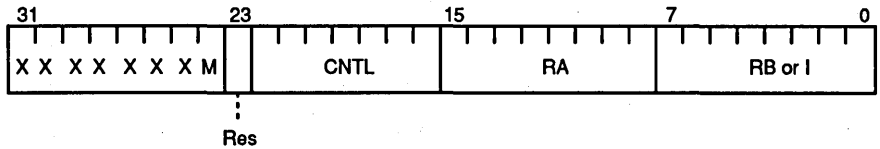
Jumps and calls with 16-bit instruction address:



Two operands with trap vector number:



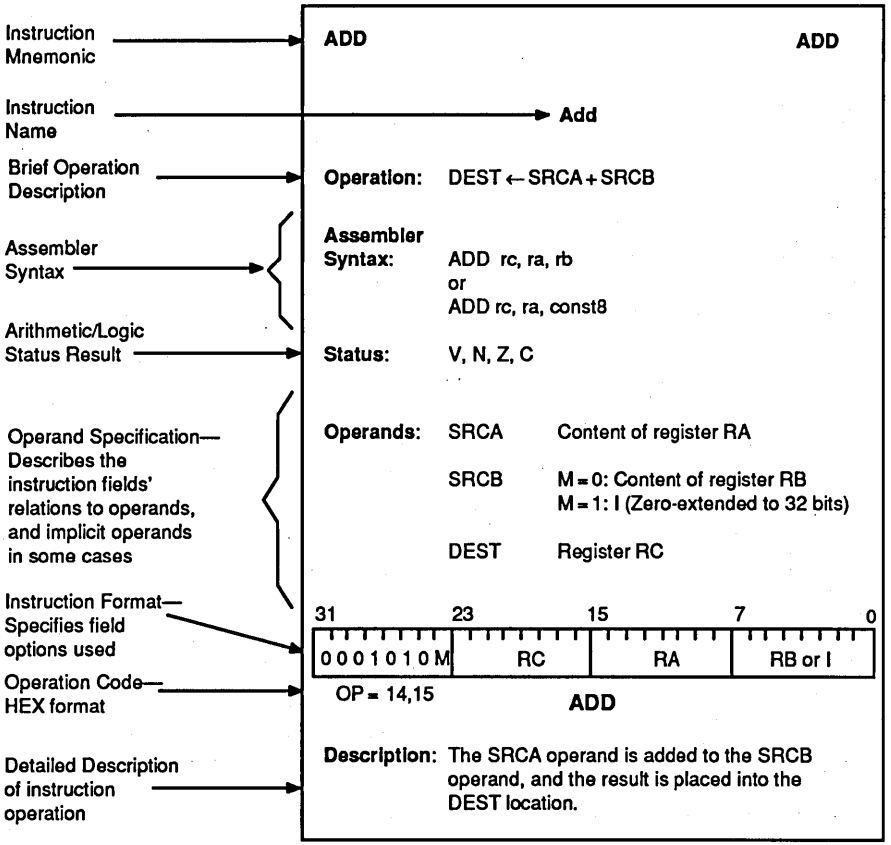
Loads and stores:



12.3 INSTRUCTION DESCRIPTION

This section describes each Am29030 and Am29035 microprocessor instruction in detail. Figure 12-3 illustrates the layout of the information given for each description.

Figure 12-3 Instruction-Description Format



Add

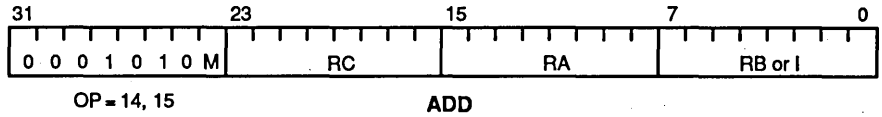
Operation: DEST ← SRCA + SRCB

Assembler Syntax:
ADD rc, ra, rb
or
ADD rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



Description: The SRCA operand is added to the SRCB operand, and the result is placed into the DEST location.

Add with Carry

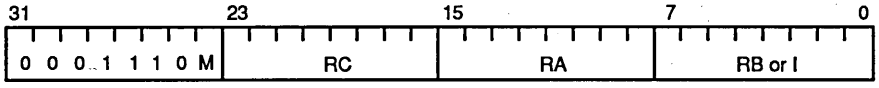
Operation: DEST ← SRCA + SRCB + C

Assembler Syntax:
ADDC rc, ra, rb
or
ADDC rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 1C, 1D

ADDC

Description: The SRCA operand is added to the SRCB operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location.

Add with Carry, Signed

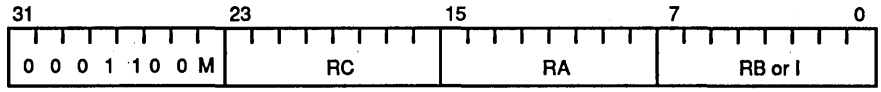
Operation: DEST ← SRCA + SRCB + C
IF signed overflow THEN Trap (Out of Range)

Assembler Syntax:
ADDCS rc, ra, rb
or
ADDCS rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 18, 19

ADDCS

Description: The SRCA operand is added to the SRCB operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location. If the add operation causes a two's-complement signed overflow, an Out of Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

Add with Carry, Unsigned

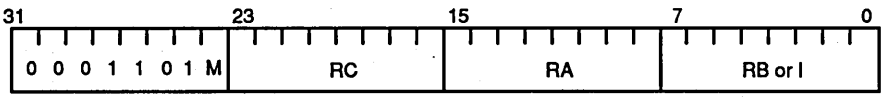
Operation: DEST ← SRCA + SRCB + C
IF unsigned overflow THEN Trap (Out of Range)

Assembler Syntax: ADDCU rc, ra, rb
or
ADDCU rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 1A, 1B

ADDCU

Description: The SRCA operand is added to the SRCB operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location. If the add operation causes an unsigned overflow, an Out of Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

Add, Signed

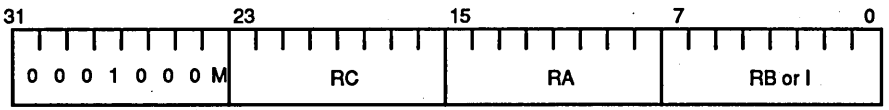
Operation: DEST ← SRCA + SRCB
IF signed overflow THEN Trap (Out of Range)

Assembler Syntax:
ADDS rc, ra, rb
or
ADDS rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 10, 11 **ADDS**

Description: The SRCA operand is added to the SRCB operand, and the result is placed into the DEST location. If the add operation causes a two's-complement signed overflow, an Out of Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

Add, Unsigned

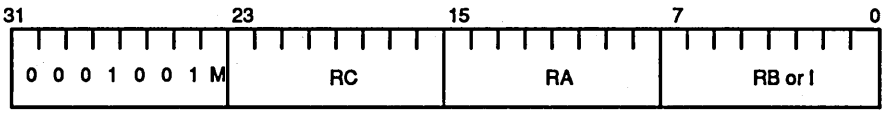
Operation: DEST ← SRCA + SRCB
IF unsigned overflow THEN Trap (Out of Range)

Assembler Syntax:
ADDU rc, ra, rb
or
ADDU rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 12, 13

ADDU

Description: The SRCA operand is added to the SRCB operand, and the result is placed into the DEST location. If the add operation causes an unsigned overflow, an Out of Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

AND Logical

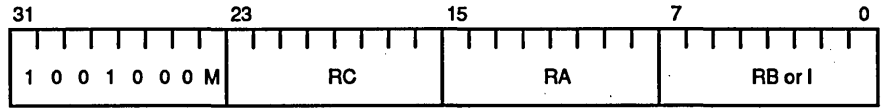
Operation: DEST ← SRCA & SRCB

Assembler Syntax:
AND rc, ra, rb
or
AND rc, ra, const8

Status: N, Z

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 90, 91 **AND**

Description: The SRCA operand is logically ANDed, bit-by-bit, with the SRCB operand, and the result is placed into the DEST location.

AND-NOT Logical

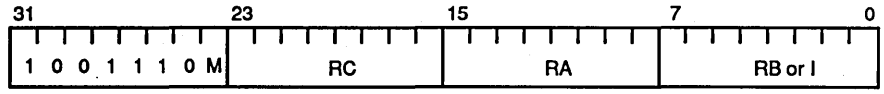
Operation: DEST ← SRCA & ~SRCB

Assembler Syntax:
ANDN rc, ra, rb
or
ANDN rc, ra, const8

Status: N, Z

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	register RC



OP = 9C, 9D

ANDN

Description: The SRCA operand is logically ANDed, bit-by-bit, with the one's-complement of the SRCB operand, and the result is placed into the DEST location.

Assert Equal To

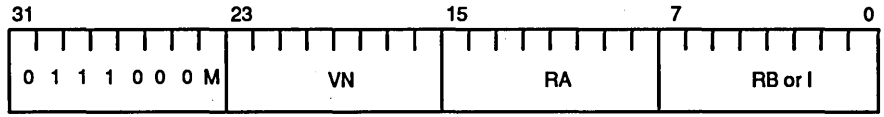
Operation: IF SRCA = SRCB THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASEQ vn, ra, rb
or
ASEQ vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 70, 71 **ASEQ**

Description: If the SRCA operand is equal to the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Greater Than or Equal To

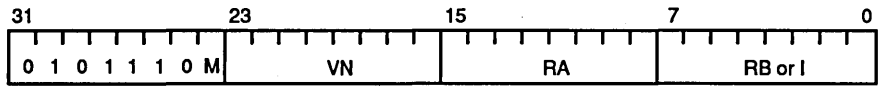
Operation: IF SRCA ≥ SRCB THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASGE vn, ra, rb
or
ASGE vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M=0: Content of register RB M=1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 5C, 5D

ASGE

Description: If the value of the SRCA operand is greater than or equal to the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Greater Than or Equal To, Unsigned

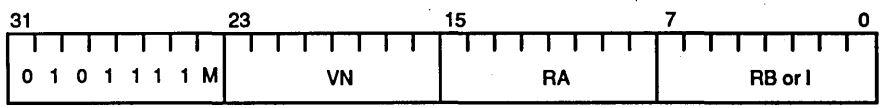
Operation: IF SRCA ≥ SRCB (unsigned) THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASGEU vn, ra, rb
or
ASGEU vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 5E, 5F

ASGEU

Description: If the value of the SRCA operand is greater than or equal to the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs. For the comparison, both operands are treated as unsigned integers.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Greater Than

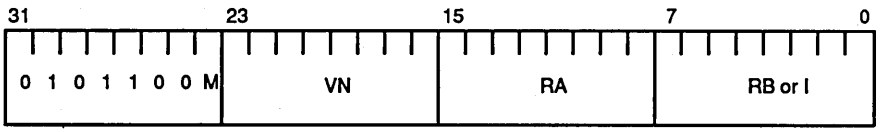
Operation: IF SRCA > SRCB THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASGT vn, ra, rb
or
ASGT vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 58, 59

ASGT

Description: If the value of the SRCA operand is greater than the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Greater Than, Unsigned

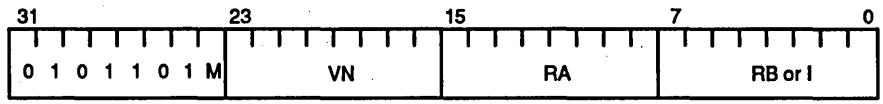
Operation: IF SRCA > SRCB (unsigned) THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASGTU vn, ra, rb
or
ASGTU vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 5A, 5B

ASGTU

Description: If the value of the SRCA operand is greater than the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs. For the comparison, both operands are treated as unsigned integers.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Less Than or Equal To

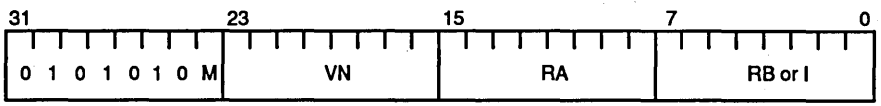
Operation: IF SRCA ≤ SRCB THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASLE vn, ra, rb
or
ASLE vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 54, 55

ASLE

Description: If the value of the SRCA operand is less than or equal to the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Less Than or Equal To, Unsigned

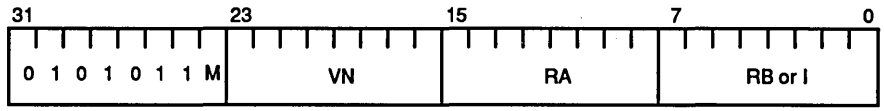
Operation: IF SRCA ≤ SRCB (unsigned) THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASLEU vn, ra, rb
or
ASLEU vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 56, 57

ASLEU

Description: If the value of the SRCA operand is less than or equal to the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs. For the comparison, both operands are treated as unsigned integers.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Less Than

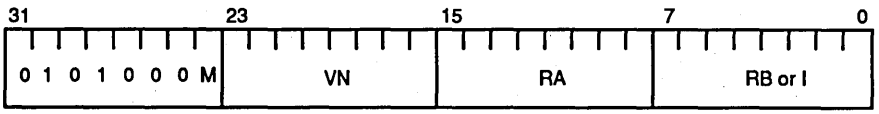
Operation: IF SRCA < SRCB THEN Continue
ELSE Trap(VN)

Assembler Syntax: ASLT vn, ra, rb
or
ASLT vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 50, 51

ASLT

Description: If the value of the SRCA operand is less than the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Less Than, Unsigned

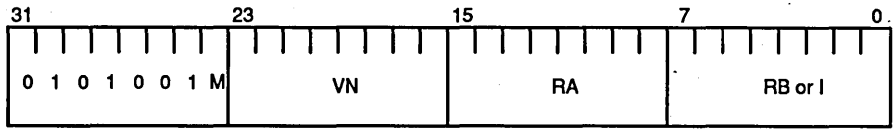
Operation: IF SRCA < SRCB (unsigned) THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASLTU vn, ra, rb
or
ASLTU vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 52, 53

ASLTU

Description: If the value of the SRCA operand is less than the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs. For the comparison, both operands are treated as unsigned integers.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Not Equal To

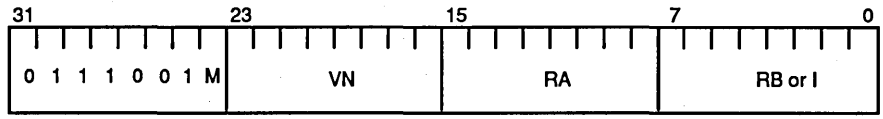
Operation: IF SRCA <> SRCB THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASNEQ vn, ra, rb
or
ASNEQ vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 72, 73

ASNEQ

Description: If the SRCA operand is not equal to the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Call Subroutine

Operation: DEST ← PC//00 + 8
PC ← TARGET
Execute delay instruction

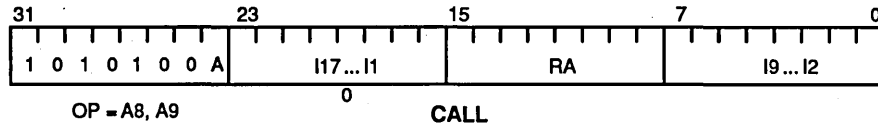
Assembler Syntax: CALL ra, target

Status: Not affected

Operands:

TARGET	A = 0: I17 ... I10 // I9 ... I2 (sign-extended to 30 bits) + PC
	A = 1: I17 ... I10 // I9 ... I2 (zero-extended to 30 bits)

DEST	Register RA



Description: The address of the second following instruction is placed into the DEST location, and a non-sequential instruction fetch occurs to the instruction address given by the TARGET operand. The instruction following the CALL is executed before the non-sequential fetch occurs.

Call Subroutine, Indirect

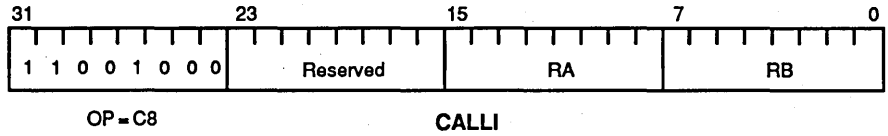
Operation: DEST ← PC // 00 + 8
PC ← SRCB
Execute delay instruction

Assembler Syntax: CALLI ra, rb

Status: Not affected

Operands:

SRCB	Content of register RB
DEST	Register RA



Description: The address of the second following instruction is placed into the DEST location, and a non-sequential instruction fetch occurs to the instruction address given by the SRCB operand. The instruction following the CALLI is executed before the non-sequential fetch occurs.

Classify Floating-Point Operand

Operation: DEST ← CLASS(SRCA)

Assembler Syntax: CLASS rc, ra, FS

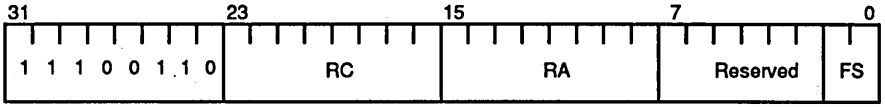
Status: None

Operands:

SRCA	Content of register RA (single-precision f.p.) or Content of register RA and the twin of register RA (Double-precision f.p.)
DEST	Register RC

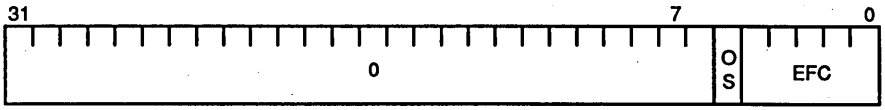
Control:

FS	Format of source operand SRCA
00	Reserved for future use
01	Single-precision floating-point
10	Double-precision floating-point
11	Reserved for future use



OP = E6 CLASS

Description: A 32-bit classification code for operand SRCA is placed into the DEST location. Operand SRCA is a single- or double-precision operand, as specified by FS. The classification code has the following format:



Bits 31–6: reserved (forced to 0).

Bit 5: Operand Sign (OS). The OS bit is 1 for a negative operand (including negative zero) and 0 for a non-negative operand.

Bits 4–0: Exponent-Fraction Class (EFC). This field classifies the biased exponent and fraction fields of the source operand as follows:

EFC	Biased Exp (bexp)	Fraction (frac)	Comments
00000	0	0	zero
00001			unused
00010	0	0 < frac < .111 ... 1	denormalized
00011	0	.111...1	denormalized
00100	1		0
00101			unused
00110	1		0 < frac < .111 ... 1
00111	1	.111 ... 1	
01000	1 < bexp < Max	0	
01001			unused
01010	1 < bexp < Max	0 < frac < .111 ... 1	
01011	1 < bexp < Max	.111... 1	
01100	Max	0	
01101			unused
01110	Max	0 < frac < .111 ... 1	
01111	Max	.111 ... 1	
10000	Max + 1	0	infinity
10001			unused
10010	Max + 1, frac MSB = 0	<> 0	SNaN
10011	Max + 1, frac MSB = 1	<> 0	QNaN

Note: Max is the largest biased exponent that can be used to represent a finite number in a given format. Max is 254 for single-precision and 2,046 for double-precision.

This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes a CLASS trap. When the trap occurs, the IPA and IPC registers are set to reference SRCA and DEST, and the IPB Register is set with the value of the FS field.

Count Leading Zeros

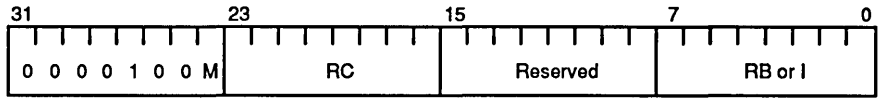
Operation: Determine number of leading zeros in a word

Assembler Syntax:
CLZ rc, rb
or
CLZ rc, const8

Status: Not affected

Operands:

SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 08,09 CLZ

Description: A count of the number of zero-bits to the first one-bit in the SRCB operand is placed into the DEST location. If the most-significant bit of the SRCB operand is 1, the resulting count is zero. If the SRCB operand is zero, the resulting count is 32.

Constant

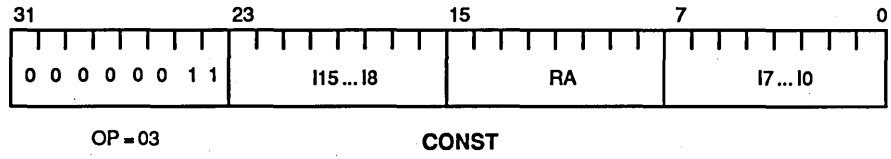
Operation: DEST ← 0I16

Assembler Syntax: CONST ra, const16

Status: Not affected

Operands:

0I16	I15 ... 8 // I7 ... I0 (Zero-extended to 32 bits)
DEST	Register RA



Description: The 0I16 operand is placed into the DEST location.

Constant, High

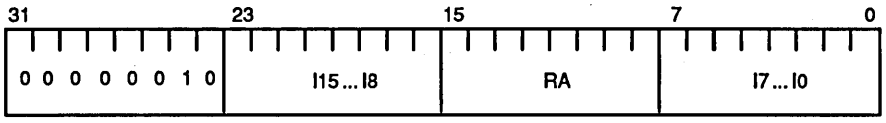
Operation: Replace high-order half-word of SRCA by I16

Assembler Syntax: CONSTH ra, const16

Status: Not affected

Operands:

	SRCA	Content of register RA
	I16	I15 ... I8 // I7 ... I0
	DEST	Register RA



OP = 02

CONSTH

Description: The low-order half-word of the SRCA operand is appended to the I16 operand, and the result is placed into the DEST operand. Note that the destination register for this instruction is the same as the source register.

Constant, Negative

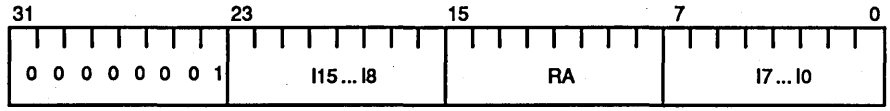
Operation: DEST ← 1I16

Assembler Syntax: CONSTN ra, const16

Status: Not affected

Operands:

1I16	I15 ... I8 // I7 ... I0 (ones-extended to 32 bits)
DEST	Register RA



Description: The 1I16 operand is placed into the DEST location.

Convert Data Format

Operation: DEST ← SRCA, with format modified per UI, RND, FD, FS

Assembler Syntax: CONVERT rc, ra, UI, RND, FD, FS

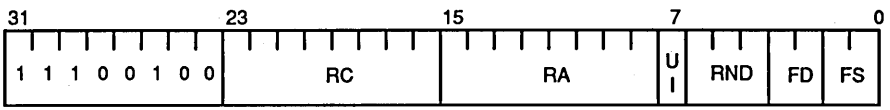
Status: fpX, fpU, fpV, fpR, fpN

Operands:

SRCA	Content of register RA (single-precision f.p.) or Content of register RA and the twin of register RA (Double-precision f.p.)
DEST	Content of register RC (single-precision f.p.) or Content of register RC and the twin of register RA (Double-precision f.p.)

Control:

UI	0 = signed integer 1 = unsigned integer
RND	Round mode
000	Round to nearest
001	Round to minus infinity
010	Round to plus infinity
011	Round to zero
100	Round using f.p. round mode (FRM)
101–111	Reserved
FS,FD	Format of source operand, format of destination operand
00	Integer
01	Single-precision floating-point
10	Double-precision floating-point
11	Reserved



OP = E4 **CONVERT**

Description: The SRCA operand with format FS is converted to format FD and rounded according to RND, then placed into the DEST location. If the source or destination operand is an integer, it is a signed or unsigned value according to the value of UI.

Note: Converting from format to like format is not supported, and will produce unpredictable results.

This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes a CONVERT trap. When the trap occurs, the IPA and IPC registers are set to reference SRCA and DEST, and the IPB Register is set with the value of the UI//RND//FD//FS field. If the UI bit is 1, the contents of the IPB Register reflect the value of this field after Stack-Pointer addition. The Stack Pointer must be subtracted from the contents of the IPB Register to recover the original value of this field.

Compare Bytes

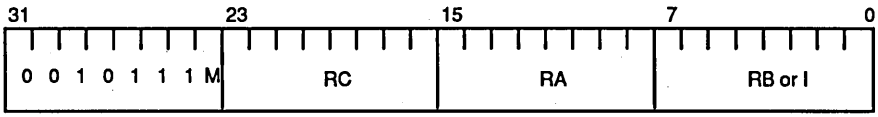
Operation: IF (SRCA.BYTE0 = SRCB.BYTE0) OR
(SRCA.BYTE1 = SRCB.BYTE1) OR
(SRCA.BYTE2 = SRCB.BYTE2) OR
(SRCA.BYTE3 = SRCB.BYTE3) THEN
DEST ← TRUE ELSE DEST ← FALSE

Assembler Syntax: CPBYTE rc, ra, rb
or
CPBYTE rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 2E, 2F

CPBYTE

Description: Each byte of the SRCA operand is compared to the corresponding byte of the SRCB operand. If any corresponding bytes are equal, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location.

Compare Equal To

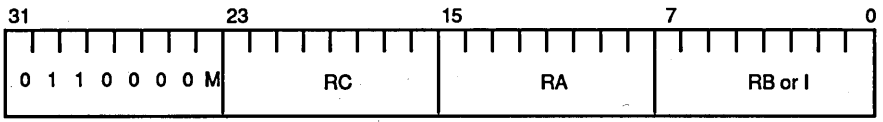
Operation: IF SRCA = SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax:
CPEQ rc, ra, rb
or
CPEQ rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 60, 61

CPEQ

Description: If the SRCA operand is equal to the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location.

Compare Greater Than or Equal To

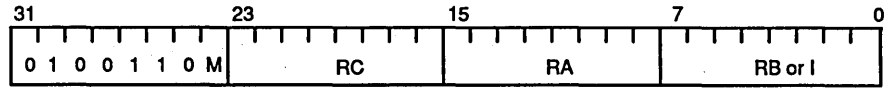
Operation: IF SRCA ≥ SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax:
CPGE rc, ra, rb
or
CPGE rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 4C, 4D

CPGE

Description: If the value of the SRCA operand is greater than or equal to the value of the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location.

Compare Greater Than or Equal To, Unsigned

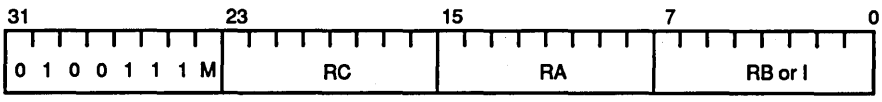
Operation: IF SRCA ≥ SRCB (unsigned) THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax:
CPGEU rc, ra, rb
or
CPGEU rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 4E, 4F

CPGEU

Description: If the value of the SRCA operand is greater than or equal to the value of the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location. For the comparison, both operands are treated as unsigned integers.

Compare Greater Than

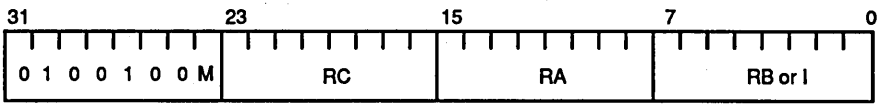
Operation: IF SRCA > SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax:
CPGT rc, ra, rb
or
CPGT rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 48, 49

CPGT

Description: If the value of the SRCA operand is greater than the value of the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location.

Compare Greater Than, Unsigned

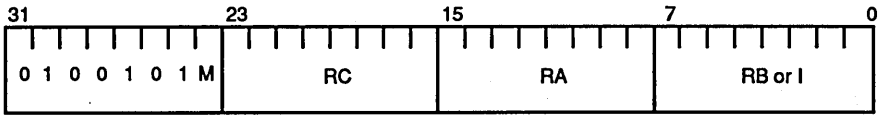
Operation: IF SRCA > SRCB (unsigned) THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax:
CPGTU rc, ra, rb
or
CPGTU rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 4A, 4B **CPGTU**

Description: If the value of the SRCA operand is greater than the value of the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location. For the comparison, both operands are treated as unsigned integers.

Compare Less Than or Equal To

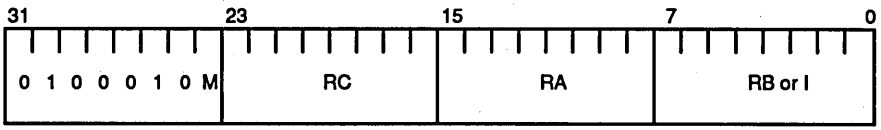
Operation: IF SRCA ≤ SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax:
CPLE rc, ra, rb
or
CPLE rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 44, 45 CPLE

Description: If the value of the SRCA operand is less than or equal to the value of the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location.

Compare Less Than or Equal To, Unsigned

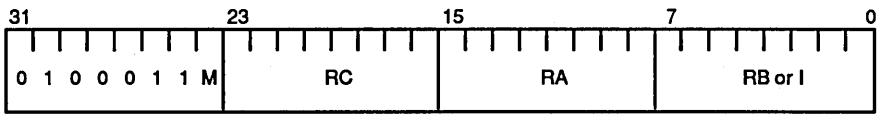
Operation: IF SRCA ≤ SRCB (unsigned) THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax:
CPLEU rc, ra, rb
or
CPLEU rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 46, 47 **CPLEU**

Description: If the value of the SRCA operand is less than or equal to the value of the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location. For the comparison, both operands are treated as unsigned integers.

Compare Less Than

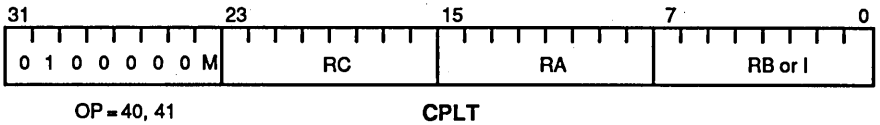
Operation: IF SRCA < SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax:
CPLT rc, ra, rb
or
CPLT rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



Description: If the value of the SRCA operand is less than the value of the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location.

Compare Less Than, Unsigned

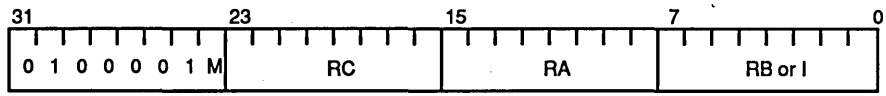
Operation: IF SRCA < SRCB (unsigned) THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax:
CPLTU rc, ra, rb
or
CPLTU rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 42, 43 **CPLTU**

Description: If the value of the SRCA operand is less than the value of the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location. For the comparison, both operands are treated as unsigned integers.

Compare Not Equal To

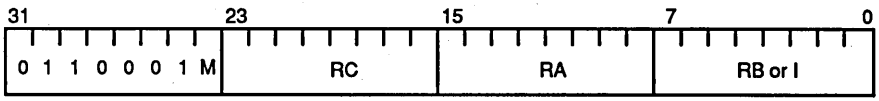
Operation: IF SRCA <> SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax:
CPNEQ rc, ra, rb
or
CPNEQ rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 62, 63

CPNEQ

Description: If the SRCA operand is not equal to the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location.

Floating-Point Add, Double-Precision

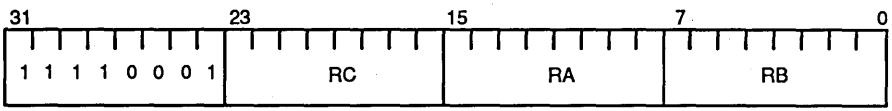
Operation: DEST (double-precision) ← SRCA (double-precision) + SRCB (double-precision)

Assembler Syntax: DADD rc, ra, rb

Status: fpX, fpU, fpV, fpR, fpN

Operands:

SRCA	Content of register RA and the twin of register RA
SRCB	Content of register RB and the twin of register RB
DEST	Register RC and the twin of register RC



OP = F1 **DADD**

Description: The SRCA operand is added to the SRCB operand; the result is rounded according to FRM field of the Floating-Point Environment Register and placed into the DEST location. The operands and result of the addition are double-precision floating-point numbers.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes a DADD trap. When the trap occurs, the IPA, IPB and IPC registers are set to reference SRCA, SRCB and DEST.

Floating-Point Divide, Double-Precision

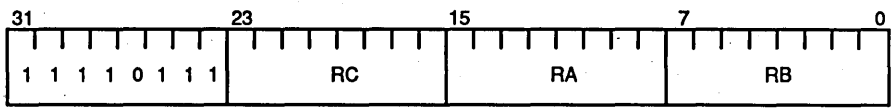
Operation: DEST (double-precision) ← SRCA (double-precision) / SRCB (double-precision)

Assembler Syntax: DDIV rc, ra, rb

Status: fpD, fpX, fpU, fpV, fpR, fpN

Operands:

SRCA	Content of register RA and the twin of register RA
SRCB	Content of register RB and the twin of register RB
DEST	Register RC and the twin of register RC



OP = F7 DDIV

Description: The SRCA operand is divided by the SRCB operand; the result is rounded according to FRM field of the Floating-Point Environment Register and placed into the DEST location. The operands and result of the division are double-precision floating-point numbers.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes a DDIV trap. When the trap occurs, the IPA, IPB and IPC registers are set to reference SRCA, SRCB and DEST.

Floating-Point Equal To, Double-Precision

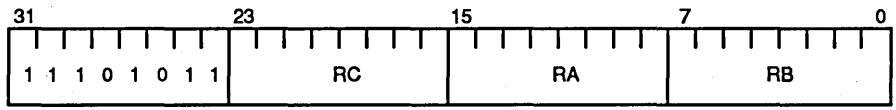
Operation: IF SRCA (double-precision) = SRCB (double-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax: DEQ rc, ra, rb

Status: fpl

Operands:

SRCA	Content of register RA and the twin of register RA
SRCB	Content of register RB and the twin of register RB
DEST	Register RC



OP = EB DEQ

Description: If the SRCA operand is equal to the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location. SRCA and SRCB are double-precision floating-point numbers.

The rounding mode specified by the FRM field of the Floating-Point Environment Register has no effect on this operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes a DEQ trap. When the trap occurs, the IPA, IPB and IPC registers are set to reference SRCA, SRCB and DEST.

Floating-Point Greater Than Or Equal To, Double-Precision

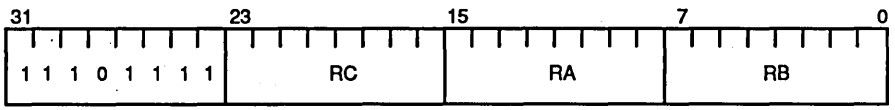
Operation: IF SRCA (double-precision) ≥ SRCB (double-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax: DGE rc, ra, rb

Status: fpl

Operands:

SRCA	Content of register RA and the twin of register RA
SRCB	Content of register RB and the twin of register RB
DEST	Register RC



OP = EF

DGE

Description: If the SRCA operand is greater than or equal to the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location. SRCA and SRCB are double-precision floating-point numbers.

The rounding mode specified by the FRM field of the Floating-Point Environment Register has no effect on this operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes a DGE trap. When the trap occurs, the IPA, IPB and IPC registers are set to reference SRCA, SRCB and DEST.

Floating-Point Greater Than, Double-Precision

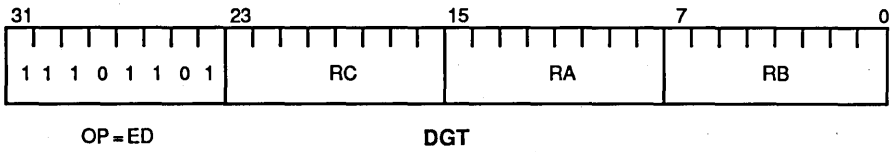
Operation: IF SRCA (double-precision) > SRCB (double-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax: DGT rc, ra, rb

Status: fpl

Operands:

SRCA	Content of register RA and the twin of register RA
SRCB	Content of register RB and the twin of register RB
DEST	Register RC



Description: If the SRCA operand is greater than the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location. SRCA and SRCB are double-precision floating-point numbers.

The rounding mode specified by the FRM field of the Floating-Point Environment Register has no effect on this operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes a DGT trap. When the trap occurs, the IPA, IPB and IPC registers are set to reference SRCA, SRCB and DEST.

Divide Step

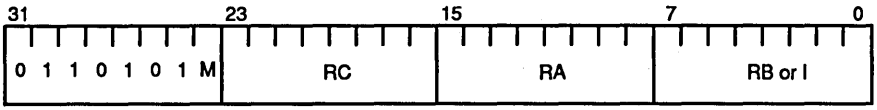
Operation: Perform one-bit step of a divide operation (unsigned)

Assembler Syntax:
DIV rc, ra, rb
or
DIV rc, ra, const 8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M=0: Content of register RB M=1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 6A, 6B DIV

Description: If the Divide Flag (DF) bit of the ALU Status Register is 1, the SRCB operand is subtracted from the SRCA operand. If the DF bit is 0, the SRCB operand is added to the SRCA operand.

The carry-out of the add or subtract operation is exclusive-ORed with the value of the DF bit and the value of the Negative (N) bit of the ALU Status Register; the resulting value is complemented and placed into the DF bit. The sign of the result of the add or subtract is placed into the N bit.

The content of the Q Register is appended to the result of the add or subtract, and the resulting 64-bit value is shifted left by one bit position; the value computed for the DF bit above fills the vacated bit position. The high-order 32 bits of the 64-bit shifted value are placed into the DEST location. The low-order 32 bits of the shifted value are placed into the Q Register.

Examples of integer divide operations appear in Section 2.6.3.

Divide Initialize

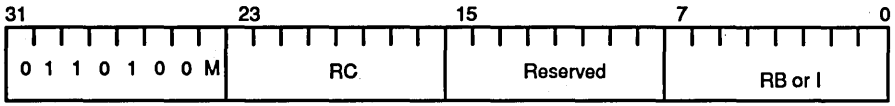
Operation: Initialize for a sequence of divide steps (unsigned)

Assembler Syntax:
DIV0 rc, rb
or
DIV0 rc, const8

Status: V, N, Z, C

Operands:

SRCB	M=0: Content of register RB M=1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 68, 69 DIV0

Description: The Divide Flag (DF) bit of the ALU Status Register is set. The sign of the SRCB operand is placed into the Negative bit of the ALU Status Register.

The content of the Q register is appended to the SRCB operand, and the resulting 64-bit value is shifted left by one bit position; a 0 fills the vacated bit position. The high-order 32 bits of the 64-bit shifted value are placed into the DEST location. The low-order 32 bits of the shifted value are placed into the Q Register.

Examples of integer divide operations appear in Section 2.6.3.

Integer Divide, Signed

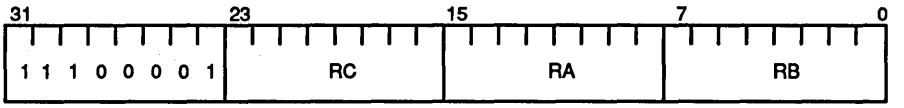
Operation: DEST ← (Q // SRCA) / SRCB (signed)
Q ← Remainder

Assembler Syntax: DIVIDE rc, ra, rb

Status: Not affected

Operands:

Q	Content of the Q Register
SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = E1 DIVIDE

Description: The SRCA operand is appended to the content of the Q register. The resulting 64-bit value is divided by the SRCB operand, and the result is placed into the DEST location. This operation treats the operands as signed two's-complement integers and produces a signed two's-complement result.

The remainder is placed into the Q register. A non-zero remainder always has the same sign as the dividend.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes a DIVIDE trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Integer Divide, Unsigned

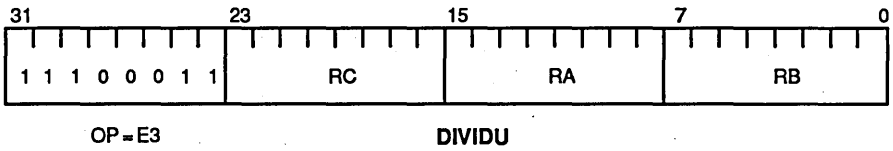
Operation: DEST ← (Q // SRCA) / SRCB (unsigned)
Q ← Remainder

Assembler Syntax: DIVIDU rc, ra, rb

Status: Not affected

Operands:

Q	Content of the Q Register
SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



Description: The SRCA operand is appended to the content of the Q Register. The resulting 64-bit value is divided by the SRCB operand, and the result is placed into the DEST location. This operation treats the operands as unsigned integers, and produces an unsigned result.

The remainder is placed into the Q Register. The remainder is also unsigned.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes a DIVIDU trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Divide Last Step

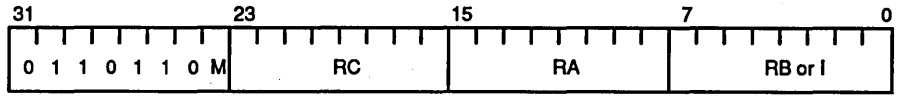
Operation: Complete a sequence of divide steps (unsigned)

Assembler Syntax: DIVL rc, ra, rb

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M=0: Content of register RB M=1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 6C, 6D

DIVL

Description: If the Divide Flag (DF) bit of the ALU Status Register is 1, the SRCB operand is subtracted from the SRCA operand. If the DF bit is 0, the SRCB operand is added to the SRCA operand. The result is placed into the DEST location.

The carry-out of the add or subtract operation is exclusive-ORed with the value of the DF bit and the value of the Negative (N) bit of the ALU Status Register; the resulting value is complemented and placed into the DF bit. The sign of the result of the add or subtract is placed into the N bit.

The content of the Q register is shifted left by one bit position; the value computed for the DF bit above fills the vacated bit position. The shifted value is placed into the Q Register.

Examples of integer divide operations appear in Section 2.6.3.

Divide Remainder

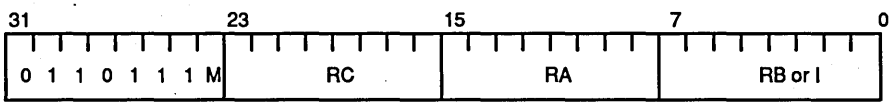
Operation: Generate remainder for divide operation (unsigned)

Assembler Syntax:
DIVREM rc, ra, rb
or
DIVREM rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M=0: Content of register RB M=1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 6E, 6F **DIVREM**

Description: If the Divide Flag (DF) bit of the ALU Status Register is 1, the SRCA operand is placed into the DEST location.

If the DF bit is 0, the SRCB operand is added to the SRCA operand, and the result is placed into the DEST location.

Examples of integer divide operations appear in Section 2.6.3.

Floating-Point Multiply, Double-Precision

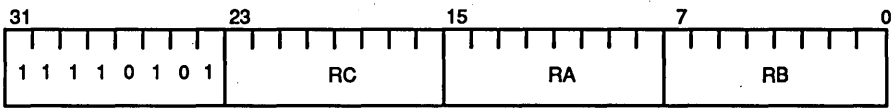
Operation: DEST (double-precision) ← SRCA (double-precision) * SRCB (double-precision)

Assembler Syntax: DMUL rc, ra, rb

Status: fpX, fpU, fpV, fpR, fpN

Operands:

SRCA	Content of register RA and the twin of register RA
SRCB	Content of register RB and the twin of register RB
DEST	Register RC



OP = F5 DMUL

Description: The SRCB operand is multiplied by the SRCA operand; the result is rounded according to FRM field of the Floating-Point Environment Register and placed into the DEST location. The operands and result of the multiplication are double-precision floating-point numbers.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes a DMUL trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Floating-Point Subtract, Double-Precision

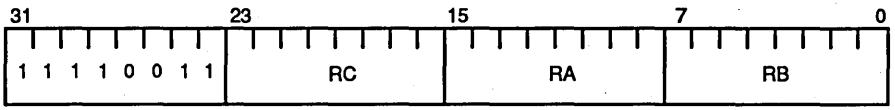
Operation: DEST (double-precision) ← SRCA (double-precision) − SRCB (double-precision)

Assembler Syntax: DSUB rc, ra, rb

Status: fpX, fpU, fpV, fpR, fpN

Operands:

SRCA	Content of register RA and the twin of register RA
SRCB	Content of register RB and the twin of register RB
DEST	Register RC



OP = F3 DSUB

Description: The SRCB operand is subtracted from the SRCA operand; the result is rounded according to FRM field of the Floating-Point Environment Register and placed into the DEST location. The operands and result of the subtraction are double-precision floating-point numbers.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes a DSUB trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Trap to Software Emulation Routine

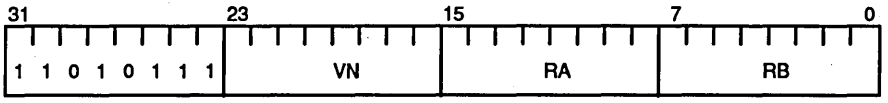
Operation: Load IPA and IPB registers with operand register-numbers and Trap (VN)

Assembler Syntax: EMULATE vn, ra, rb

Status: Not affected

Operands: Absolute-register numbers for registers RA and RB

VN Trap vector number



OP = D7 **EMULATE**

Description: The IPA and IPB registers are set to the register numbers of registers RA and RB, respectively. A trap with the specified vector number occurs.

Note that the IPC register also is affected by this instruction, but that its value has no interpretation.

For programs in the User mode, a Protection Violation trap occurs—instead of the EMULATE trap—if a vector number between 0 and 63 is specified.

Extract Byte

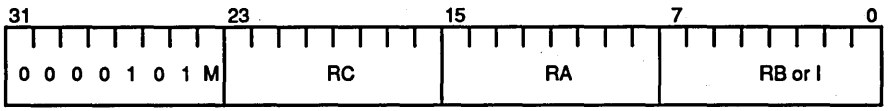
Operation: DEST ← SRCB, with low-order byte replaced by byte in SRCA selected by BP

Assembler Syntax:
EXBYTE rc, ra, rb
or
EXBYTE rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 0A, 0B EXBYTE

Description: A byte in the SRCA operand is selected by the Byte Pointer (BP) field of the ALU Status Register and the Byte Order (BO) bit of the Configuration Register. The selected byte replaces the low-order byte of the SRCB operand and the resulting word is placed into the DEST location.

Note: The selection of bytes within words is specified in Section 3.3.7.1.

Extract Half-Word

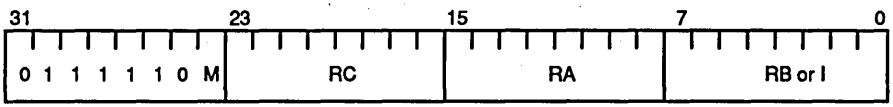
Operation: DEST ← SRCB, with low-order half-word replaced by half-word in SRCA selected by BP

Assembler Syntax:
EXHW rc, ra, rb
or
EXHW rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 7C, 7D EXHW

Description: A half-word in the SRCA operand is selected by the Byte Pointer (BP) field of the ALU Status Register and the Byte Order (BO) bit of the Configuration Register. The selected half-word replaces the low-order half-word of the SRCB operand, and the resulting word is placed into the DEST location.

Note: The selection of half-words within words is specified in Section 3.3.7.1.

Extract Half-Word, Sign-Extended

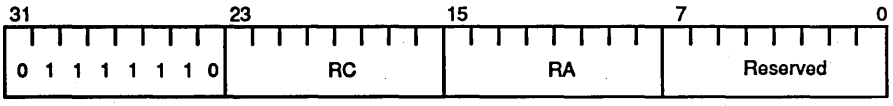
Operation: DEST ← half-word in SRCA selected by BP, sign-extended to 32 bits

Assembler Syntax: EXHWS rc, ra

Status: Not affected

Operands:

SRCA	Content of register RA
DEST	Register RC



OP = 7E

EXHWS

Description: A half-word in the SRCA operand is selected by the Byte Pointer (BP) field of the ALU Status Register and the Byte Order (BO) bit of the Configuration Register. The selected half-word is sign-extended to 32 bits, and the resulting word is placed into the DEST location.

Note: The selection of half-words within words is specified in Section 3.3.7.1.

Extract Word, Bit-Aligned

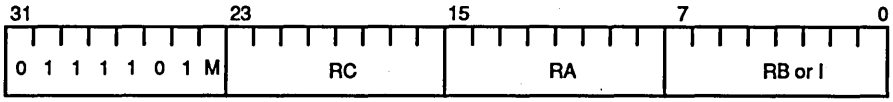
Operation: DEST ← high-order word of (SRCA // SRCB << FC)

Assembler Syntax:
EXTRACT rc, ra ,rb
or
EXTRACT rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 7A, 7B EXTRACT

Description: The SRCB operand is appended to the SRCA operand, and the resulting 64-bit value is shifted left by the number of bit-positions specified by the Funnel Shift Count (FC) field of the ALU Status register. The high-order 32 bits of the 64-bit shifted value are placed in the DEST location.

If the SRCB operand is the same as the SRCA operand, the EXTRACT instruction performs a rotate operation.

Floating-Point Add, Single-Precision

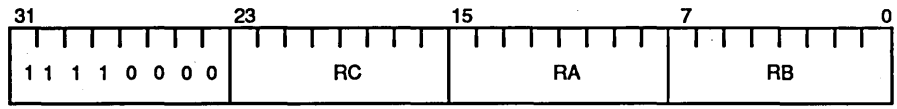
Operation: DEST (single-precision) ← SRCA (single-precision) + SRCB (single-precision)

Assembler Syntax: FADD rc, ra, rb

Status: fpX, fpU, fpV, fpR, fpN

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = F0

FADD

Description: The SRCA operand is added to the SRCB operand; the result is rounded according to the FRM field of the Floating-Point Environment Register and placed into the DEST location. The operands and result of the addition are single-precision floating-point numbers.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes an FADD trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB and DEST.

Floating-Point Divide, Single-Precision

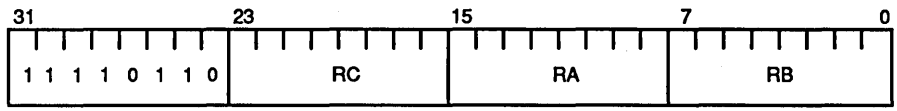
Operation: DEST (single-precision) ← SRCA (single-precision) / SRCB (single-precision)

Assembler Syntax: FDIV rc, ra, rb

Status: fpD, fpX, fpU, fpV, fpR, fpN

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = F6

FDIV

Description: The SRCA operand is divided by the SRCB operand; the result is rounded according to the FRM field of the Floating-Point Environment Register and placed into the DEST location. The operands and result of the division are single-precision floating-point numbers.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes an FDIV trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB and DEST.

Floating-Point Multiply, Single-to-Double Precision

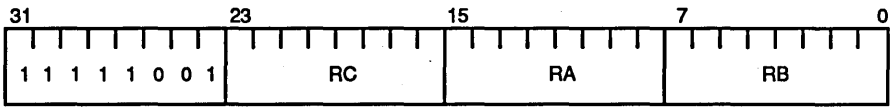
Operation: DEST (double-precision) ← SRCA (single-precision) * SRCB (single-precision)

Assembler Syntax: FDMUL rc, ra, rb

Status: fpR, fpN

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = F9 FDMUL

Description: The SRCB operand is multiplied by the SRCA operand; the result is placed into the DEST location. SRCA and SRCB are single-precision floating-point numbers; the result is produced in double-precision format. Because the product of two single-precision operands can always be represented exactly as a double-precision number, the FDMUL result does not depend on the FRM field of the Floating-Point Environment Register.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes an FDMUL trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB and DEST.

Floating-Point Equal To, Single-Precision

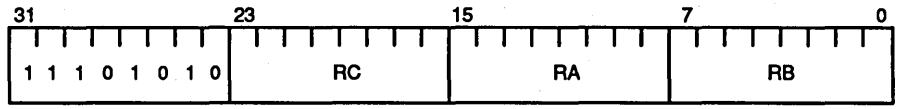
Operation: IF SRCA (single-precision) = SRCB (single-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax: FEQ rc, ra, rb

Status: fpN

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = EA FEQ

Description: If the SRCA operand is equal to the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location. SRCA and SRCB are single-precision floating-point numbers.

The rounding mode specified by the FRM field of the Floating-Point Environment Register has no effect on this operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes an FEQ trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB and DEST.

Floating-Point Greater Than Or Equal To, Single-Precision

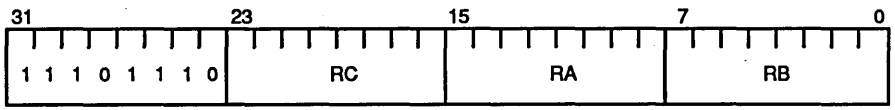
Operation: IF SRCA (single-precision) ≥ SRCB (single-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax: FGE rc, ra, rb

Status: fpN

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = EE FGE

Description: If the SRCA operand is greater than or equal to the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location. SRCA and SRCB are single-precision floating-point numbers.

The rounding mode specified by the FRM field of the Floating- Point Environment Register has no effect on this operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes an FGE trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB and DEST.

Floating-Point Greater Than, Single-Precision

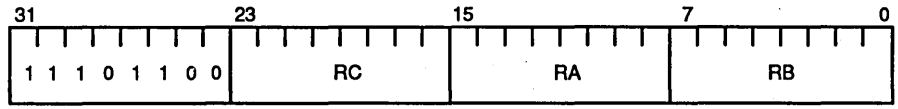
Operation: IF SRCA (single-precision) > SRCB (single-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax: FGT rc, ra, rb

Status: fpN

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = EC FGT

Description: If the SRCA operand is greater than the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location. SRCA and SRCB are single-precision floating-point numbers.

The rounding mode specified by the FRM field of the Floating-Point Environment Register has no effect on this operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes an FGT trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB and DEST.

Floating-Point Multiply, Single-Precision

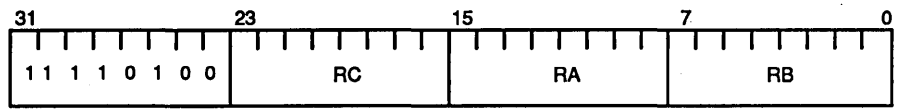
Operation: DEST (single-precision) ← SRCA (single-precision) * SRCB (single-precision)

Assembler Syntax: FMUL rc, ra, rb

Status: fpX, fpU, fpV, fpR, fpN

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = F4 FMUL

Description: The SRCA operand is multiplied by the SRCB operand; the result is rounded according to the FRM field of the Floating-Point Environment Register and placed into the DEST location. The operands and result of the multiplication are single-precision floating-point numbers.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes an FMUL trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB and DEST.

Floating-Point Subtract, Single-Precision

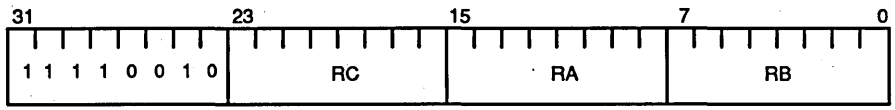
Operation: DEST (single-precision) ← SRCA (single-precision) – SRCB (single-precision)

Assembler Syntax: FSUB rc, ra, rb

Status: fpX, fpU, fpV, fpR, fpN

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = F2 **FSUB**

Description: The SRCB operand is subtracted from the SRCA operand; the result is rounded according to the FRM field of the Floating-Point Environment Register and placed into the DEST location. The operands and result of the subtraction are single-precision floating-point numbers.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes an FSUB trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB and DEST.

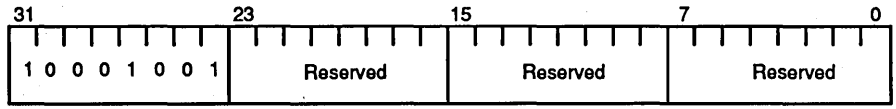
Enter Halt Mode

Operation: Enter Halt mode on next cycle

Assembler Syntax: HALT

Status: Not affected

Operands: Not applicable



OP = 89 HALT

Description: The processor is placed into the Halt mode on the next cycle, except that any external data accesses are completed.

This instruction may be executed only by Supervisor-mode programs. An attempted execution by a User-mode program causes a Protection Violation trap to occur unless the Protection Violation trap was disabled during reset (see Sections 11.2 and 10.2.2).

If the instruction following a Halt instruction has an exception (e.g., TLB Miss), the trap associated with this exception is taken before the processor enters the Halt mode.

Insert Byte

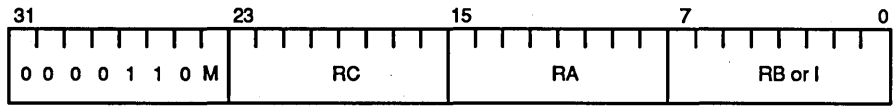
Operation: DEST ← SRCA, with byte selected by BP replaced by low-order byte of SRCB

Assembler Syntax:
INBYTE rc, ra, rb
or
INBYTE rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP =0C, 0D **INBYTE**

Description: A byte in the SRCA operand is selected by the Byte Pointer (BP) field of the ALU Status Register and the Byte Order (BO) bit of the Configuration Register. The selected byte is replaced by the low-order byte of the SRCB operand, and the resulting word is placed into the DEST location.

Note: The selection of bytes within words is specified in Section 3.3.7.1.

Insert Half-Word

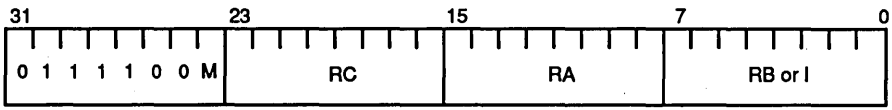
Operation: DEST ← SRCA, with half-word selected by BP replaced by low-order half-word of SRCB

Assembler Syntax:
INHW rc, ra, rb
or
INHW rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 78, 79 INHW

Description: A half-word in the SRCA operand is selected by the Byte Pointer (BP) field of the ALU Status Register and the Byte Order (BO) bit of the Configuration Register. The selected half-word is replaced by the low-order half-word of the SRCB operand, and the resulting word is placed into the DEST location.

Note: The selection of half-words within words is specified in Section 3.3.7.1.

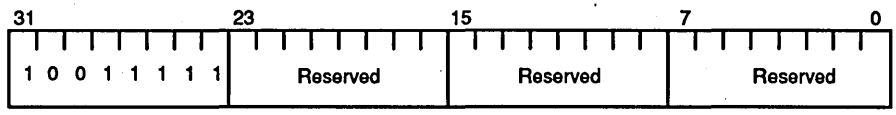
Invalidate

Operation: Reset all Valid bits in the Instruction Cache

Assembler Syntax: INV

Status: Not affected

Operands: Not applicable



OP = 9F INV

Description: This instruction causes all Instruction Cache Valid bits to be reset on the execution of the next successful branch, unless the blocks are locked and the cache is enabled (see Sections 9.1 and 10.2). This causes all Instruction Cache blocks to become invalid.

This instruction may be executed only by Supervisor-mode programs. An attempted execution by a User-mode program causes a Protection Violation trap to occur.

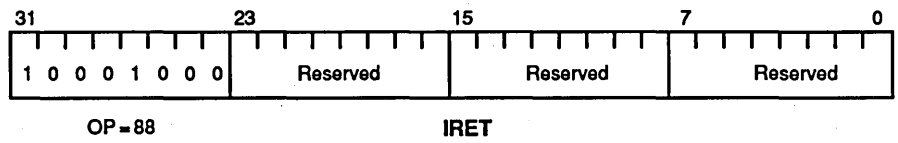
Interrupt Return

Operation: Perform an interrupt return sequence

Assembler Syntax: IRET

Status: Not affected

Operands: Not applicable



Description: This instruction performs the interrupt return sequence described in Section 8.3.4.

This instruction may be executed only by Supervisor-mode programs. An attempted execution by a User-mode program causes a Protection Violation trap to occur.

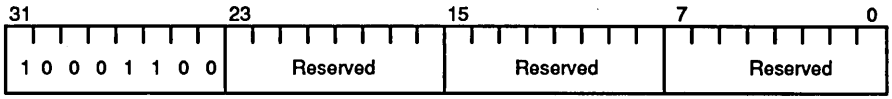
Interrupt Return and Invalidate

Operation: Perform an interrupt return sequence, and reset all valid bits in the Instruction Cache

Assembler Syntax: IRETINV

Status: Not affected

Operands: Not applicable



OP = 8C

IRETINV

Description: This instruction performs the interrupt return sequence described in Section 8.3.4. When the sequence begins, all Instruction Cache Valid bits are reset to zeros. This causes all Instruction Cache blocks to become invalid, unless the blocks are locked and the cache is enabled (see Sections 9.1 and 10.2).

This instruction may be executed only by Supervisor-mode programs. An attempted execution by a User-mode program causes a Protection Violation trap to occur.

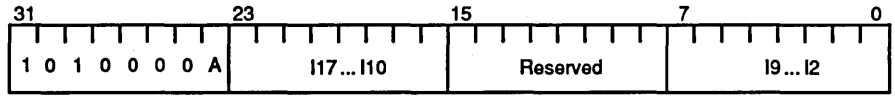
Jump

Operation: PC←TARGET
Execute delay instruction

Assembler Syntax: JMP target

Status: Not affected

Operands: TARGET A=0: I17 ... I10 // I9 ... I2 (sign-extended to 30 bits) + PC
A=1: I17 ... I10//I9 ... I2 (zero-extended to 30 bits)



OP = A0, A1

JMP

Description: A non-sequential instruction fetch occurs to the instruction address given by the TARGET operand. The instruction following the JMP is executed before the non-sequential fetch occurs.

Jump False

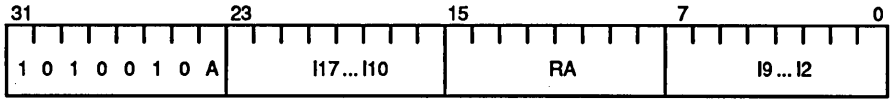
Operation: IF SRCA = FALSE THEN PC ← TARGET
Execute delay instruction

Assembler Syntax: JMPF ra, target

Status: Not affected

Operands: SRCA Content of register RA

TARGET A = 0: I17 ... I10 // I9 ... I2 (sign-extended to 30 bits) + PC
A = 1: I17 ... I10 // I9 ... I2 (zero-extended to 30 bits)



OP = A4, A5 JMPF

Description: If SRCA is a Boolean FALSE, a non-sequential instruction fetch occurs to the instruction address given by the TARGET operand.

If SRCA is a Boolean TRUE, this instruction has no effect.

The instruction following the JMPF is executed regardless of the value of SRCA.

Jump False and Decrement

Operation:

```
IF SRCA = FALSE THEN
    SRCA ← SRCA – 1
    PC ← TARGET
ELSE
    SRCA ← SRCA – 1
Execute delay instruction
```

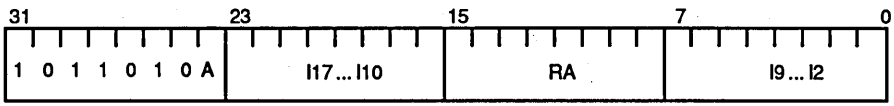
Assembler Syntax: JMPFDEC ra, target

Status: Not affected

Operands:

SRCA	Content of register RA

TARGET	A = 0: I17 ... I10 // I9 ... I2 (sign-extended to 30 bits) + PC
	A = 1: I17 ... I10 // I9 ... I2 (zero-extended to 30 bits)



OP = B4, B5

JMPFDEC

Description: If SRCA is a Boolean FALSE, a non-sequential instruction fetch occurs to the instruction address given by the TARGET operand.

If SRCA is a Boolean TRUE, this instruction has no effect on the instruction-execution sequence.

The SRCA operand is decremented by one, regardless of whether or not the non-sequential instruction fetch occurs. Note that a negative number for the SRCA operand is a Boolean TRUE.

The instruction following the JMPFDEC is executed regardless of the value of SRCA.

Jump False Indirect

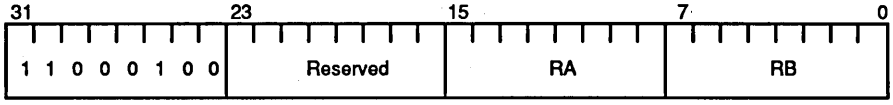
Operation: IF SRCA = FALSE THEN PC ← SRCB
Execute delay instruction

Assembler Syntax: JMPFI ra, rb

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	Content of register RB



OP = C4 JMPFI

Description: If the SRCA is a Boolean FALSE, a non-sequential instruction fetch occurs to the instruction address given by the SRCB operand.

If SRCA is a Boolean TRUE, this instruction has no effect.

The instruction following the JMPFI is executed regardless of the value of SRCA.

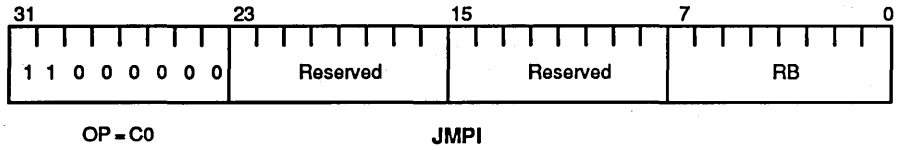
Jump Indirect

Operation: PC ← SRCB
Execute delay instruction

Assembler Syntax: JMPI rb

Status: Not affected

Operands: SRCB Content of register RB



Description: A non-sequential instruction fetch occurs to the instruction address given by the SRCB operand. The instruction following the JMPI is executed before the non-sequential fetch occurs.

Jump True

Operation: IF SRCA = TRUE THEN PC ← TARGET
Execute delay instruction

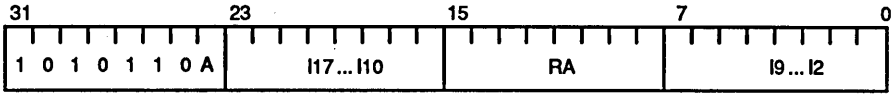
Assembler Syntax: JMPT ra, target

Status: Not affected

Operands:

SRCA	Content of register RA

TARGET	A = 0: I17 ... I10 // I9 ... I2 (sign-extended to 30 bits) + PC
	A = 1: I17 ... I10 // I9 ... I2 (zero-extended to 30 bits)



OP = AC, AD JMPT

Description: If SRCA is a Boolean TRUE, a non-sequential instruction fetch occurs to the instruction address given by the TARGET operand.

If SRCA is a Boolean FALSE, this instruction has no effect.

The instruction following the JMPT is executed regardless of the value of SRCA.

Jump True Indirect

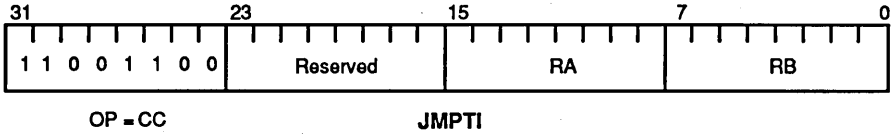
Operation: IF SRCA = TRUE THEN PC ← SRCB
Execute delay instruction

Assembler Syntax: JMPTI ra, rb

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	Content of register RB



Description: If the SRCA is a Boolean TRUE, a non-sequential instruction fetch occurs to the instruction address given by the SRCB operand.

If SRCA is a Boolean FALSE, this instruction has no effect.

The instruction following the JMPTI is executed regardless of the value of SRCA.

Load

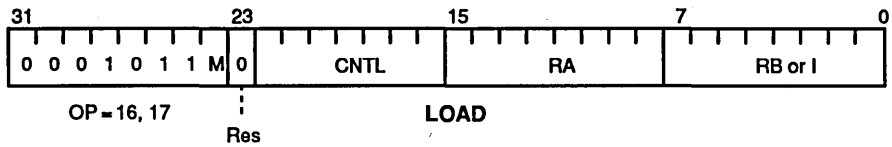
Operation: DEST ← EXTERNAL WORD [SRCB]

Assembler Syntax:
LOAD 0, cntl, ra, rb
or
LOAD 0, cntl, ra, const8

Status: Not affected

Operands:

SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RA



Description: The external word addressed by the SRCB operand is placed into the DEST location.

The CNTL field of the LOAD instruction affects the bus access as described in Section 3.3.2.

Load and Lock

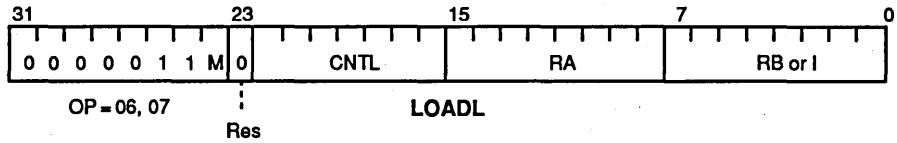
Operation: DEST ← EXTERNAL WORD [SRCB], assert $\overline{\text{LOCK}}$ output during access

Assembler Syntax:
LOADL 0, cntl, ra, rb
or
LOADL 0, cntl, ra, const8

Status: Not affected

Operands:

SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RA



Description: The external word addressed by the SRCB operand is placed into the DEST location.

The CNTL field of the LOADL instruction affects the bus access as described in Section 3.3.2.

The $\overline{\text{LOCK}}$ output is asserted during the bus access.

Load Multiple

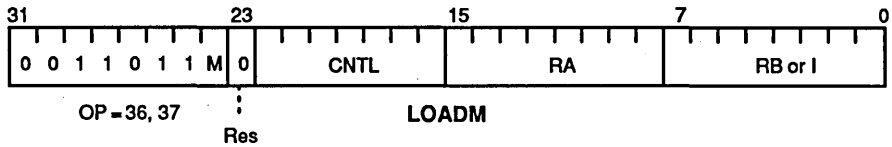
Operation: DEST... DEST+COUNT ← EXTERNAL WORD [SRCB] ...
EXTERNAL WORD [SRCB + (COUNT * 4)]

Assembler Syntax:
LOADM 0, cntl, ra, rb
or
LOADM 0, cntl, ra, const8

Status: Not affected

Operands:

SRCB	M = 0: Content of register RB M = 1: I (zero-extended to 32 bits)
DEST	register RA



Description: External words at consecutive word addresses, beginning with the word addressed by the SRCB operand, are placed into consecutive registers, beginning with the DEST location.

The total number of words accessed in the sequence is specified by the Count Remaining (CR) field of the Channel Control Register (which also appears in the Load/Store Count Remaining Register) at the beginning of the bus access. The total number of words is the value of the CR field plus one. The CNTL field of the LOADM instruction affects the bus access as described in Section 3.3.2.

Note: The address and register-number sequences for the LOADM instruction are specified in Section 3.3.5.

Load and Set

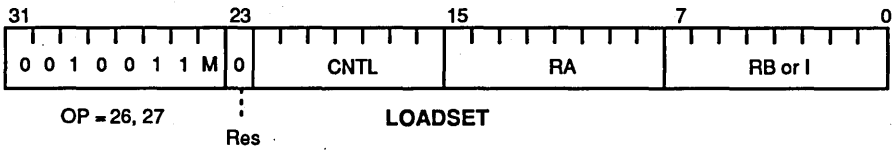
Operation: DEST ← EXTERNAL WORD [SRCB]
EXTERNAL WORD [SRCB] ← h'FFFFFFFF',
assert $\overline{\text{LOCK}}$ output during access

Assembler Syntax:
LOADSET 0, cntl, ra, rb
or
LOADSET 0, cntl, ra, const8

Status: Not affected

Operands:

SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RA



Description: The external word addressed by the SRCB operand is placed into the DEST location. After the DEST location is altered, the external word addressed by the SRCB operand is written, atomically, with a word consisting of a 1 in every bit position.

The CNTL field of the LOADSET instruction affects the bus access as described in Section 3.3.2.

The $\overline{\text{LOCK}}$ output is asserted throughout the LOADSET operation.

Move from Special Register

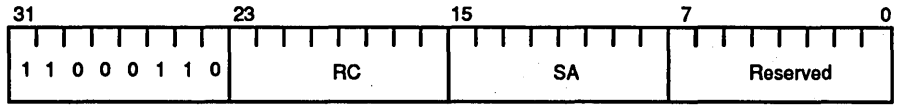
Operation: DEST ← SPECIAL

Assembler Syntax: MFSR rc, spid

Status: Not affected

Operands:

SPECIAL	Content of special-purpose register SA
DEST	Register RC



OP = C6 MFSR

Description: The SPECIAL operand is placed into the DEST location.

For programs in the User mode, a Protection Violation trap occurs if SA specifies a protected special-purpose register. If a trap occurs, the DEST location is not altered.

Move from Translation Look-Aside Buffer Register

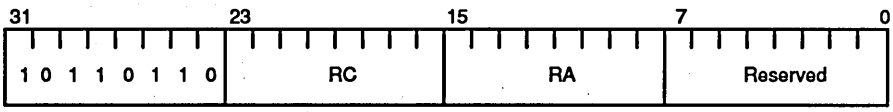
Operation: DEST ← TLB [SRCA]

Assembler Syntax: MFTLB rc, ra

Status: Not affected

Operands:

SRCA	Content of register RA, bits 6 ... 0
DEST	Register RC



OP = B6

MFTLB

Description: The Translation Look-Aside Buffer (TLB) register whose register number is specified by the SRCA operand is placed into the DEST location.

This instruction may be executed only by Supervisor-mode programs. An attempted execution by a User-mode program causes a Protection Violation trap to occur. If a trap occurs, the DEST location is not altered.

Move to Special Register

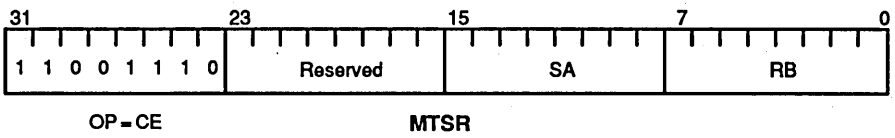
Operation: SPDEST ← SRCB

Assembler Syntax: MTSR spid, rb

Status: Not affected, unless the destination is the ALU Status Register

Operands:

SRCB	Content of register RB
SPDEST	Special-purpose register SA



Description: The SRCB operand is placed into the SPECIAL location.

For programs in the User mode, a Protection Violation trap occurs if SA specifies a protected special-purpose register. If a trap occurs, the SPDEST location is not altered.

Move to Special Register Immediate

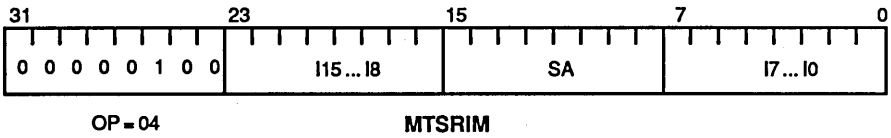
Operation: SPDEST ← 0I16

Assembler Syntax: MTSRIM spid, const16

Status: Not affected, unless the destination is the ALU Status Register

Operands:

0I16	I15 ... I8 // I7 ... I0 (zero-extended to 32 bits)
SPDEST	Special-purpose register SA



Description: The 0I16 operand is placed into the SPECIAL location.

For programs in the User mode, a Protection Violation trap occurs if SA specifies a protected special-purpose register. If a trap occurs, the SPDEST location is not altered.

Move to Translation Look-Aside Buffer Register

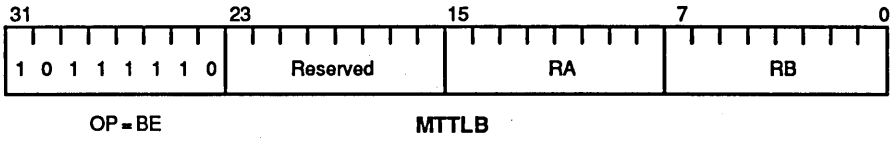
Operation: TLB [SRCA] ← SRCB

Assembler Syntax: MTTLB ra, rb

Status: Not affected

Operands:

SRCA	Content of register RA, bits 6...0
SRCB	Content of register RB



Description: The SRCB operand is placed into the Translation Look-Aside Buffer (TLB) register whose register-number is specified by the SRCA operand.

This instruction may be executed only by Supervisor-mode programs. An attempted execution by a User-mode program causes a Protection Violation trap to occur. If a trap occurs, the TLB register is not altered.

Multiply Step

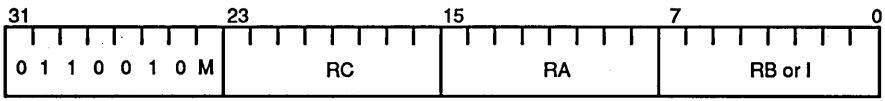
Operation: Perform one-bit step of a multiply operation

Assembler Syntax:
MUL rc, ra, rb
or
MUL rc, ra, const 8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M=0: Content of register RB M=1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 64, 65 MUL

Description: If the least-significant bit of the Q Register is 1, the SRCA operand is added to the SRCB operand. If the least-significant bit of the Q register is 0, a zero word is added to the SRCB operand.

The content of the Q Register is appended to the result of the add, and the resulting 64-bit value is shifted right by one bit position; the true sign of the result of the add fills the vacated bit position (i.e., the sign of the result is complemented if an overflow occurred during the add operation). The high-order 32 bits of the 64-bit shifted value are placed into the DEST location. The low-order 32 bits of the shifted value are placed into the Q Register.

Examples of integer multiply operations appear in Section 2.6.2.

Multiply Last Step

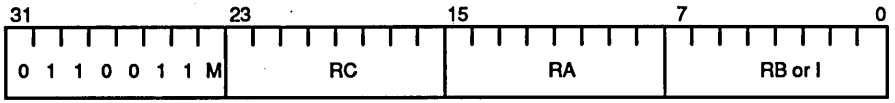
Operation: Complete a sequence of multiply steps (for signed multiply)

Assembler Syntax:
MULL rc, ra, rb
or
MULL rc, ra, const 8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 66, 67 MULL

Description: If the least-significant bit of the Q Register is 1, the SRCA operand is subtracted from the SRCB operand. If the least-significant bit of the Q register is 0, a zero word is subtracted from the SRCB operand.

The content of the Q Register is appended to the result of the subtract, and the resulting 64-bit value is shifted right by one bit position; the true sign of the result of the subtract fills the vacated bit position (i.e., the sign of the result is complemented if an overflow occurred during the subtract operation). The high-order 32 bits of the 64-bit shifted value are placed into the DEST location. The low-order 32 bits of the shifted value are placed into the Q Register.

Examples of integer multiply operations appear in Section 2.6.2.

Integer Multiply, Unsigned

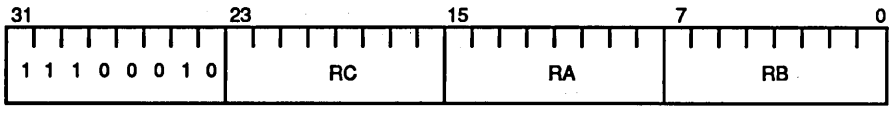
Operation: DEST ← SRCA * SRCB

Assembler Syntax: MULTIPLU rc, ra, rb

Status: None

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = E2 MULTIPLU

Description: The SRCA operand is multiplied by the SRCB operand. The low-order 32 bits of the 64-bit result are placed into the DEST location. This operation treats the SRCA and SRCB operands as unsigned integers and produces an unsigned result.

The contents of the Q register are undefined after a MULTIPLU operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes an MULTIPLU trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB and DEST.

Integer Multiply, Signed

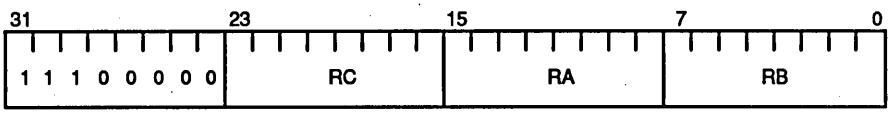
Operation: DEST ← SRCA * SRCB

Assembler Syntax: MULTIPLY rc, ra, rb

Status: None

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = E0

MULTIPLY

Description: The SRCA operand is multiplied by the SRCB operand. The low-order 32 bits of the 64-bit result are placed into the DEST location. This operation treats the SRCA and SRCB operands as two's-complement integers and produces a two's-complement result.

The contents of the Q register are undefined after a MULTIPLY operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes an MULTIPLY trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB and DEST.

Integer Multiply Most-Significant Bits, Signed

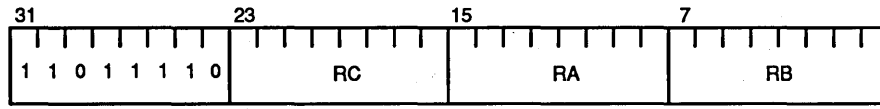
Operation: DEST ← SRCA * SRCB

Assembler Syntax: MULTM rc, ra, rb

Status: None

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = DE

MULTM

Description: The SRCA operand is multiplied by the SRCB operand. The high-order 32 bits of the 64-bit result are placed into the DEST location. This operation treats the SRCA and SRCB operands as two's-complement integers and produces a two's-complement result.

The contents of the Q register are undefined after a MULTM operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes an MULTM trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB and DEST.

Integer Multiply Most-Significant Bits, Unsigned

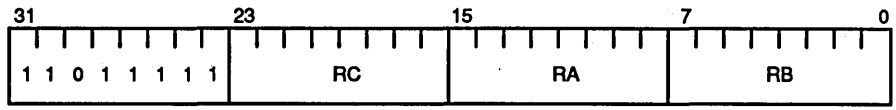
Operation: DEST ← SRCA * SRCB

Assembler Syntax: MULTMU rc, ra, rb

Status: None

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = DF

MULTMU

Description: The SRCA operand is multiplied by the SRCB operand. The high-order 32 bits of the 64-bit result are placed into the DEST location. This operation treats the SRCA and SRCB operands as unsigned integers and produces an unsigned result.

The contents of the Q register are undefined after a MULTMU operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes an MULTMU trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB and DEST.

Multiply Step, Unsigned

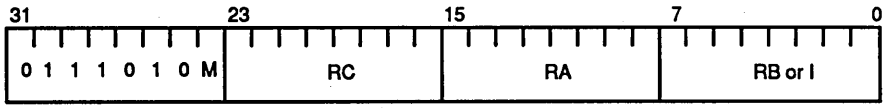
Operation: Perform one-bit step of a multiply operation (unsigned)

Assembler Syntax:
MULU rc, ra, rb
or
MULU rc, ra, const 8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M=0: Content of register RB M=1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 74, 75 **MULU**

Description: If the least-significant bit of the Q Register is 1, the SRCA operand is added to the SRCB operand. If the least-significant bit of the Q register is 0, a zero word is added to the SRCB operand.

The content of the Q register is appended to the result of the add, and the resulting 64-bit value is shifted right by one bit position; the carry-out of the add fills the vacated bit position. The high-order 32 bits of the 64-bit shifted value are placed into the DEST location. The low-order 32 bits of the shifted value are placed into the Q Register.

NAND Logical

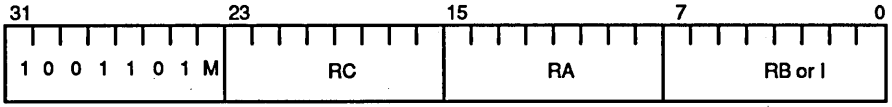
Operation: DEST ← ~(SRCA & SRCB)

Assembler Syntax:
NAND rc, ra, rb
or
NAND rc, ra, const8

Status: N, Z

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 9A, 9B

NAND

Description: The SRCA operand is logically ANDed, bit-by-bit, with the SRCB operand. The one's-complement of the result is placed into the DEST location.

NOR Logical

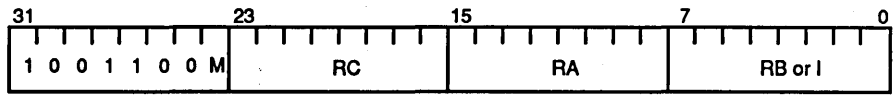
Operation: DEST ← ~(SRCA | SRCB)

Assembler Syntax:
NOR rc, ra, rb
or
NOR rc, ra, const8

Status: N, Z

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 98, 99 **NOR**

Description: The SRCA operand is logically ORed, bit-by-bit, with the SRCB operand. The one's-complement of the result is placed into the DEST location.

OR Logical

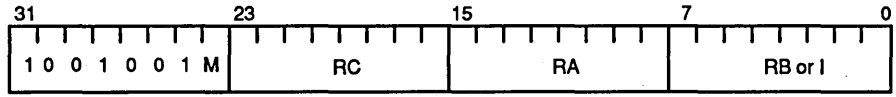
Operation: DEST ← SRCA | SRCB

Assembler Syntax:
OR rc, ra, rb
or
OR rc, ra, const8

Status: N, Z

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 92, 93 **OR**

Description: The SRCA operand is logically ORed, bit-by-bit, with the SRCB operand, and the result is placed into the DEST location.

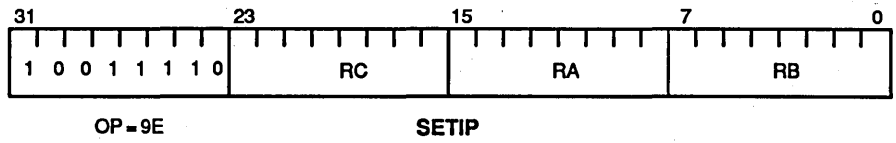
Set Indirect Pointers

Operation: Load IPA, IPB, and IPC registers with operand-register numbers

Assembler Syntax: SETIP rc, ra, rb

Status: Not affected

Operands: Absolute-register numbers for registers RA, RB, and RC



Description: The IPA, IPB, and IPC registers are set to the register numbers of registers RA, RB, and RC, respectively.

For programs in the User mode, a Protection Violation trap occurs if RA, RB, or RC specifies a register that is protected by the Register Bank Protect Register.

Note: This instruction has a delayed effect on the indirect pointer registers as discussed in Section 5.6.

Shift Left Logical

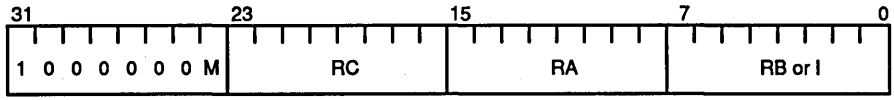
Operation: DEST ← SRCA << SRCB (zero fill)

Assembler Syntax:
SLL rc, ra, rb
or
SLL rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB, bits 4 ... 0 M = 1: I, bits 4...0
DEST	Register RC



OP = 80, 81 SLL

Description: The SRCA operand is shifted left by the number of bit positions specified by the SRCB operand; zeros fill vacated bit positions. The result is placed into the DEST location.

Floating-Point Square Root

Operation: DEST ← SQRT(SRCA)

Assembler Syntax: SQRT rc, ra, FS

Status: fpX, fpR, fpN

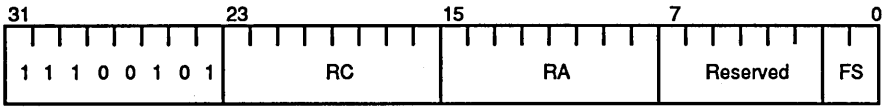
Operands:

SRCA Content of register RA (single-precision f.p.)
or
Content of register RA and the twin of register RA (double-precision f.p.)

DEST Register RC (single-precision f.p.)
or
Register RC and twin of Register RC (double-precision f.p.)

Control:

FS Format of source operand SRCA
00 Reserved for future use
01 Single-precision floating-point
10 Double-precision floating-point
11 Reserved for future use



OP = E5

SQRT

Description: This operation computes the square root of floating-point operand SRCA; the result is rounded according to FRM field of the Floating-Point Environment Register and placed into the DEST location. The operand and result are single- or double-precision floating-point numbers, as specified by FS.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes an SQRT trap. When the trap occurs, the IPA and IPC registers are set to reference SRCA and DEST, and the IPB Register is set with the value of the FS field.

Shift Right Arithmetic

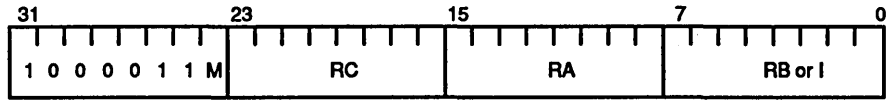
Operation: DEST ← SRCA >> SRCB (sign fill)

Assembler Syntax:
SRA rc, ra, rb
or
SRA rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB, bits 4 ... 0 M = 1: I, bits 4 ... 0
DEST	Register RC



OP = 86, 87 SRA

Description: The SRCA operand is shifted right by the number of bit positions specified by the SRCB operand; the sign of the SRCA operand fills vacated bit positions. The result is placed into the DEST location.

Shift Right Logical

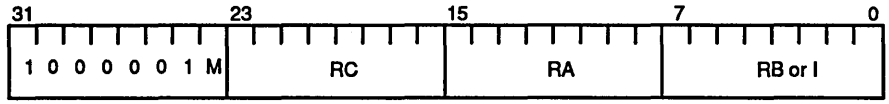
Operation: DEST ← SRCA >> SRCB (zero fill)

Assembler Syntax:
SRL rc, ra, rb
or
SRL rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB, bits 4 ... 0 M = 1: I, bits 4 ... 0
DEST	Register RC



OP = 82, 83 **SRL**

Description: The SRCA operand is shifted right by the number of bit positions specified by the SRCB operand; zeros fill vacated bit positions. The result is placed into the DEST location.

Store

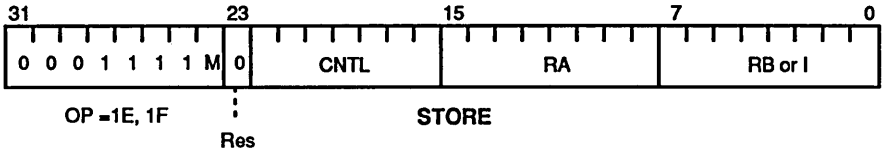
Operation: EXTERNAL WORD [SRCB] ← SRCA

Assembler Syntax:
STORE 0, cntl, ra, rb
or
STORE 0, cntl, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)



Description: The SRCA operand is placed into the external word addressed by the SRCB operand.

The CNTL field of the STORE instruction affects the bus access as described in Section 3.3.2.

Store and Lock

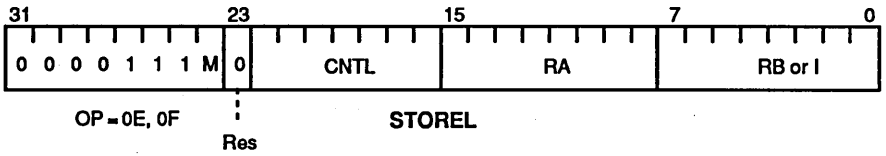
Operation: EXTERNAL WORD [SRCB] ← SRCA,
assert $\overline{\text{LOCK}}$ output during access

Assembler Syntax:
STOREL 0, cntl, ra, rb
or
STOREL 0, cntl, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)



Description: The SRCA operand is placed into the external word addressed by the SRCB operand.

The CNTL field of the STOREL instruction affects the bus access as described in Section 3.3.2.

The $\overline{\text{LOCK}}$ output is asserted during the bus access.

Store Multiple

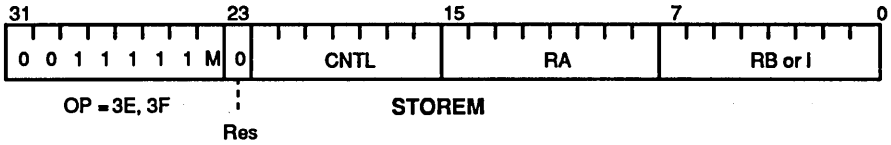
Operation: EXTERNAL WORD [SRCB] ... EXTERNAL WORD [SRCB + (COUNT * 4)]
← SRCA ... SRCA+COUNT

Assembler Syntax:
STOREM 0, cntl, ra, rb
or
STOREM 0, cntl, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)



Description: The contents of consecutive registers, beginning with the SRCA operand, are placed into external words at consecutive word addresses, beginning with the word addressed by the SRCB operand.

The total number of words accessed in the sequence is specified by the Count Remaining (CR) field of the Channel Control Register (which also appears in the Load/Store Count Remaining Register) at the beginning of the bus access. The total number of words is the value of the CR field plus one. The CNTL field of the STOREM instruction affects the access as described in Section 3.3.2.

Note: The address and register-number sequences for the STOREM instruction are specified in Section 3.3.5.

Subtract

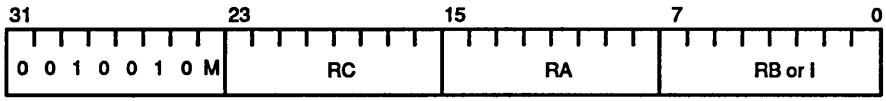
Operation: DEST ← SRCA – SRCB

Assembler Syntax:
SUB rc, ra, rb
or
SUB rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 24, 25 SUB

Description: The SRCA operand is added to the two's-complement of the SRCB operand, and the result is placed into the DEST location.

Subtract with Carry

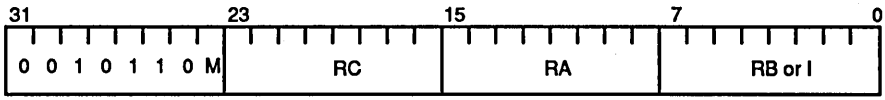
Operation: DEST ← SRCA − SRCB − 1 + C

Assembler Syntax:
SUBC rc, ra, rb
or
SUBC rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 2C, 2D

SUBC

Description: The SRCA operand is added to the one's-complement of the SRCB operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location.

Subtract with Carry, Signed

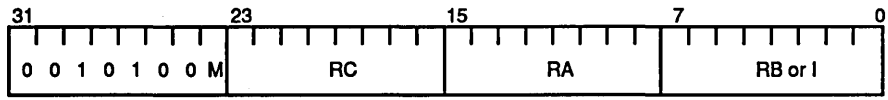
Operation: DEST ← SRCA − SRCB − 1 + C
IF signed overflow THEN Trap (Out of Range)

Assembler Syntax:
SUBCS rc, ra, rb
or
SUBCS rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 28, 29 **SUBCS**

Description: The SRCA operand is added to the one's-complement of the SRCB operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location. If the add operation causes a two's-complement signed overflow, an Out of Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

Subtract with Carry, Unsigned

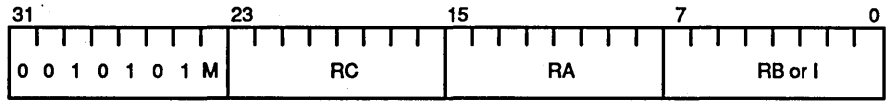
Operation: DEST ← SRCA − SRCB − 1 + C
IF unsigned underflow THEN Trap (Out of Range)

Assembler Syntax:
SUBCU rc, ra, rb
or
SUBCU rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 2A, 2B

SUBCU

Description: The SRCA operand is added to the one's-complement of the SRCB operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location. If the add operation causes an unsigned underflow, an Out of Range trap occurs.

Note that the DEST location is altered whether or not an underflow occurs.

Subtract Reverse

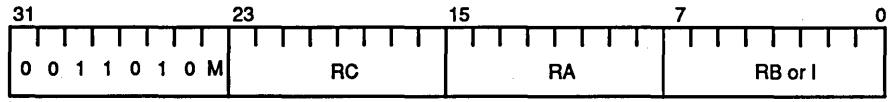
Operation: DEST ← SRCB − SRCA

Assembler Syntax:
SUBR rc, ra, rb
or
SUBR rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 34, 35 SUBR

Description: The SRCB operand is added to the two's-complement of the SRCA operand and the result is placed into the DEST location.

Subtract Reverse with Carry

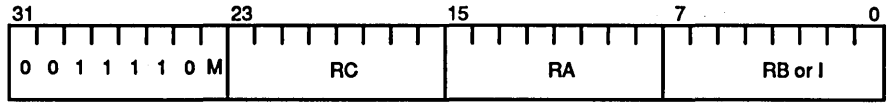
Operation: DEST ← SRCB − SRCA − 1 + C

Assembler Syntax:
SUBRC rc, ra, rb
or
SUBRC rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 3C, 3D

SUBRC

Description: The SRCB operand is added to the one's-complement of the SRCA operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location.

Subtract Reverse with Carry, Signed

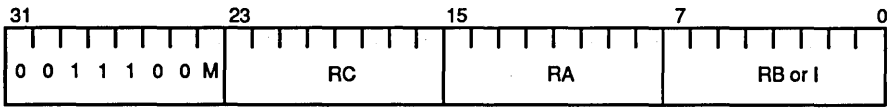
Operation: DEST ← SRCB − SRCA − 1 + C
IF signed overflow THEN Trap (Out of Range)

Assembler Syntax:
SUBRCS rc, ra, rb
or
SUBRCS rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 38, 39 SUBRCS

Description: The SRCB operand is added to the one's-complement of the SRCA operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location. If the add operation causes a two's-complement signed overflow, an Out of Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

Subtract Reverse with Carry, Unsigned

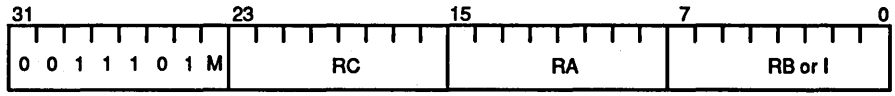
Operation: DEST ← SRCB − SRCA − 1 + C
IF unsigned underflow THEN Trap (Out of Range)

Assembler Syntax:
SUBRCU rc, ra, rb
or
SUBRCU rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 3A, 3B

SUBRCU

Description: The SRCB operand is added to the one's-complement of the SRCA operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location. If the add operation causes an unsigned underflow, an Out of Range trap occurs.

Note that the DEST location is altered whether or not an underflow occurs.

Subtract Reverse, Signed

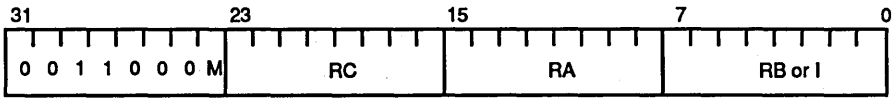
Operation: DEST ← SRCB − SRCA
IF signed overflow THEN Trap (Out of Range)

Assembler Syntax:
SUBRS rc, ra, rb
or
SUBRS rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 30, 31 **SUBRS**

Description: The SRCB operand is added to the two's-complement of the SRCA operand, and the result is placed into the DEST location. If the add operation causes a two's-complement signed overflow, an Out of Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

Subtract Reverse, Unsigned

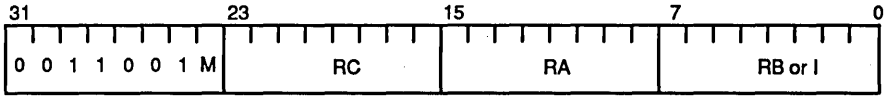
Operation: DEST ← SRCB − SRCA
IF unsigned underflow THEN Trap (Out of Range)

Assembler Syntax: SUBRU rc, ra, rb
or
SUBRU rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 32, 33 **SUBRU**

Description: The SRCB operand is added to the two's-complement of the SRCA operand, and the result is placed into the DEST location. If the add operation causes an unsigned underflow, an Out of Range trap occurs.

Note that the DEST location is altered whether or not an underflow occurs.

Subtract, Signed

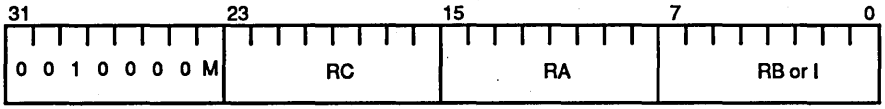
Operation: DEST ← SRCA − SRCB
IF signed overflow THEN Trap (Out of Range)

Assembler Syntax:
SUBS rc, ra, rb
or
SUBS rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 20, 21 **SUBS**

Description: The SRCA operand is added to the two's-complement of the SRCB operand, and the result is placed into the DEST location. If the add operation causes a two's-complement signed overflow, an Out of Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

Subtract, Unsigned

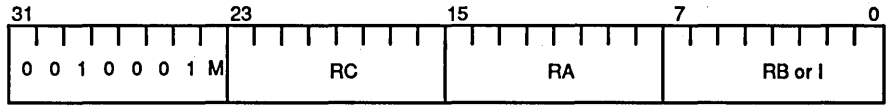
Operation: DEST ← SRCA − SRCB
IF unsigned underflow THEN Trap (Out of Range)

Assembler Syntax:
SUBU rc, ra, rb
or
SUBU rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 22, 23 **SUBU**

Description: The SRCA operand is added to the two's-complement of the SRCB operand, and the result is placed into the DEST location. If the add operation causes an unsigned underflow, an Out of Range trap occurs.

Note that the DEST location is altered whether or not an underflow occurs.

Exclusive-NOR Logical

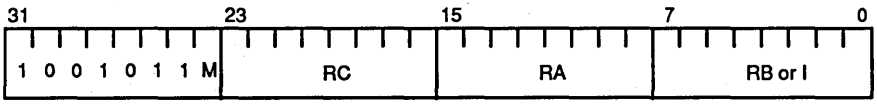
Operation: DEST ← ~ (SRCA ^ SRCB)

Assembler Syntax:
XNOR rc, ra, rb
or
XNOR rc, ra, const8

Status: N, Z

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 96, 97 **XNOR**

Description: The SRCA operand is logically exclusive-ORed, bit-by-bit, with the SRCB operand. The one's-complement of the result is placed into the DEST location.

Exclusive-OR Logical

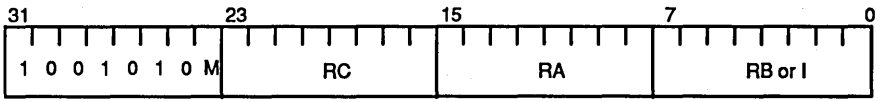
Operation: DEST ← SRCA ^ SRCB

Assembler Syntax:
XOR rc, ra, rb
or
XOR rc, ra, const8

Status: N, Z

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 94, 95 **XOR**

Description: The SRCA operand is logically exclusive-ORed, bit-by-bit, with the SRCB operand, and the result is placed into the DEST location.

12.4 INSTRUCTION INDEX BY OPERATION CODE

01	CONSTN	Constant, Negative
02	CONSTH	Constant, High
03	CONST	Constant
04	MTSRIM	Move to Special Register Immediate
06,07	LOADL	Load and Lock
08,09	CLZ	Count Leading Zeros
0A,0B	EXBYTE	Extract Byte
0C,0D	INBYTE	Insert Byte
0E,0F	STOREL	Store and Lock
10,11	ADDS	Add, Signed
12,13	ADDU	Add, Unsigned
14,15	ADD	Add
16,17	LOAD	Load
18,19	ADDCS	Add with Carry, Signed
1A,1B	ADDCU	Add with Carry, Unsigned
1C,1D	ADDC	Add with Carry
1E,1F	STORE	Store
20,21	SUBS	Subtract, Signed
22,23	SUBU	Subtract, Unsigned
24,25	SUB	Subtract
26,27	LOADSET	Load and Set
28,29	SUBCS	Subtract with Carry, Signed
2A,2B	SUBCU	Subtract with Carry, Unsigned
2C,2D	SUBC	Subtract with Carry
2E,2F	CPBYTE	Compare Bytes
30,31	SUBRS	Subtract Reverse, Signed
32,33	SUBRU	Subtract Reverse, Unsigned
34,35	SUBR	Subtract Reverse
36,37	LOADM	Load Multiple
38,39	SUBRCS	Subtract Reverse with Carry, Signed
3A,3B	SUBRCU	Subtract Reverse with Carry, Unsigned
3C,3D	SUBRC	Subtract Reverse with Carry
3E,3F	STOREM	Store Multiple
40,41	CPLT	Compare Less Than
42,43	CPLTU	Compare Less Than, Unsigned
44,45	CPLE	Compare Less Than or Equal To
46,47	CPLEU	Compare Less Than or Equal To, Unsigned
48,49	CPGT	Compare Greater Than
4A,4B	CPGTU	Compare Greater Than, Unsigned
4C,4D	CPGE	Compare Greater Than or Equal To
4E,4F	CPGEU	Compare Greater Than or Equal To, Unsigned
50,51	ASLT	Assert Less Than
52,53	ASLTU	Assert Less Than, Unsigned
54,55	ASLE	Assert Less Than or Equal To
56,57	ASLEU	Assert Less Than or Equal To, Unsigned
58,59	ASGT	Assert Greater Than

5A,5B	ASGTU	Assert Greater Than, Unsigned
5C,5D	ASGE	Assert Greater Than or Equal To
5E,5F	ASGEU	Assert Greater Than or Equal To, Unsigned
60,61	CPEQ	Compare Equal To
62,63	CPNEQ	Compare Not Equal To
64,65	MUL	Multiply Step
66,67	MULL	Multiply Last Step
68,69	DIV0	Divide Initialize
6A,6B	DIV	Divide Step
6C,6D	DIVL	Divide Last Step
6E,6F	DIVREM	Divide Remainder
70,71	ASEQ	Assert Equal To
72,73	ASNEQ	Assert Not Equal To
74,75	MULU	Multiply Step, Unsigned
78,79	INHW	Insert Half-Word
7A,7B	EXTRACT	Extract Word, Bit-Aligned
7C,7D	EXHW	Extract Half-Word
7E	EXHWS	Extract Half-Word, Sign-Extended
80,81	SLL	Shift Left Logical
82,83	SRL	Shift Right Logical
86,87	SRA	Shift Right Arithmetic
88	IRET	Interrupt Return
89	HALT	Enter HALT Mode
8C	IRETINV	Interrupt Return and Invalidate
90,91	AND	AND Logical
92,93	OR	OR Logical
94,95	XOR	Exclusive-OR Logical
96,97	XNOR	Exclusive-NOR Logical
98,99	NOR	NOR Logical
9A,9B	NAND	NAND Logical
9C,9D	ANDN	AND-NOT Logical
9E	SETIP	Set Indirect Pointers
9F	INV	Invalidate
A0,A1	JMP	Jump
A4,A5	JMPF	Jump False
A8,A9	CALL	Call Subroutine
AC,AD	JMPT	Jump True
B4,B5	JMPFDEC	Jump False and Decrement
B6	MFTLB	Move from Translation Look-Aside Buffer Register
BE	MTTLB	Move to Translation Look-Aside Buffer Register
C0	JMPI	Jump Indirect
C4	JMPFI	Jump False Indirect
C6	MFSR	Move from Special Register
C8	CALLI	Call Subroutine, Indirect
CC	JMPTI	Jump True Indirect
CE	MTSR	Move to Special Register
D7	EMULATE	Trap to Software Emulation Routine

D8–DD	Reserved for emulation (trap vector numbers 24–29)	
DE	MULTM	Integer Multiply Most-Significant Bits, Signed
DF	MULTMU	Integer Multiply Most-Significant Bits, Unsigned
E0	MULTIPLY	Integer Multiply, Signed
E1	DIVIDE	Integer Divide, Signed
E2	MULTIPLU	Integer Multiply, Unsigned
E3	DIVIDU	Integer Divide, Unsigned
E4	CONVERT	Convert Data Format
E5	SQRT	Square Root
E6	CLASS	Classify Floating-Point Operand
E7–E9	Reserved for emulation (trap vector number 39–41)	
EA	FEQ	Floating-Point Equal To, Single-Precision
EB	DEQ	Floating-Point Equal To, Double-Precision
EC	FGT	Floating-Point Greater Than, Single-Precision
ED	DGT	Floating-Point Greater Than, Double-Precision
EE	FGE	Floating-Point Greater Than or Equal To, Single-Precision
EF	DGE	Floating-Point Greater Than or Equal To, Double-Precision
F0	FADD	Floating-Point Add, Single-Precision
F1	DADD	Floating-Point Add, Double-Precision
F2	FSUB	Floating-Point Subtract, Single-Precision
F3	DSUB	Floating-Point Subtract, Double-Precision
F4	FMUL	Floating-Point Multiply, Single-Precision
F5	DMUL	Floating-Point Multiply, Double-Precision
F6	FDIV	Floating-Point Divide, Single-Precision
F7	DDIV	Floating-Point Divide, Double-Precision
F8	Reserved for emulation (trap vector number 56)	
F9	FDMUL	Floating-Point Multiply, Single-to-Double-Precision
FA–FF	Reserved for emulation (trap vector numbers 58–63)	

APPENDIX A

BUS SUMMARY AND TIMING DIAGRAMS

Table A-1 Signal Summary

Signal Name	Signal Function	Type (1)	Synch Async
A(31–0)	Address Bus	Three-State Output	Synch
$\overline{\text{BGRT}}$	Bus Grant	Input	Synch
$\overline{\text{BREQ}}$	Bus Request	Output	Synch
$\overline{\text{BURST}}$	Burst Request	Three-State Output	Synch
$\overline{\text{BWE}}$(3–0)	Byte Write Enable	Three-State Output	Synch
CNTL(1–0)	CPU Control	Input	Async
$\overline{\text{DI}}$	Reserved	Tied High	
$\overline{\text{DIV2}}$	Divide Clock By 2	Input	N/A
$\overline{\text{ERLYA}}$	Early Address	Input	Synch
$\overline{\text{ERR}}$	Data Error	Input	Synch
$\overline{\text{HIT}}$	Reserved	Tied High	
I/$\overline{\text{D}}$	Instruction or Data Access	Three-State Output	Synch
ID(31–0)	Instruction/Data Bus	Bi-directional	Synch
INCLK	Input Clock	Input	N /A
$\overline{\text{INTR}}$(3–0)	Interrupt Request	Input	Async
IO/$\overline{\text{MEM}}$	Input/Output or Memory Access	Three-State Output	Synch
$\overline{\text{LOCK}}$	Lock	Three-State Output	Synch
MEMCLK	Memory Clock	Bidirectional	N/A
MPGM(1–0)	MMU Programmable	Three-State Output	Synch
MSERR	Master/Slave Error	Output	Synch
OPT(2–0)	Option Control	Three-State Output	Synch
$\overline{\text{PGMODE}}$	Page-Mode Access	Three-State Output	Synch

(1) The signals labeled "Three-state output" and "bi-directional" (except MEMCLK) are disabled when the bus is granted to an external master. All outputs (except MSERR) may be disabled by asserting the $\overline{\text{TEST}}$ input, or through the IEEE1149.1-1990 (JTAG) Test Access Port.

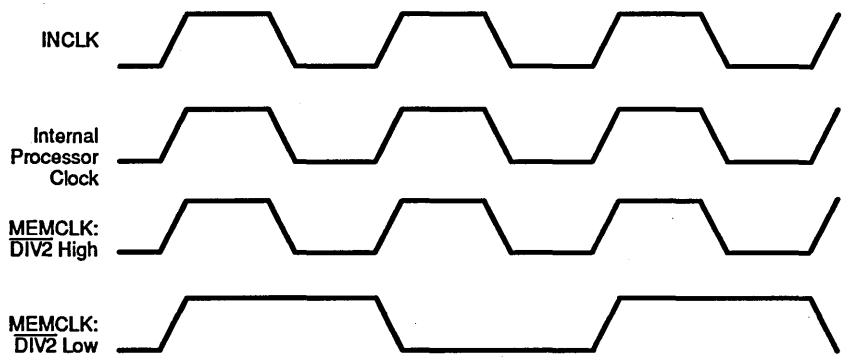
Table A-1 Signal Summary (continued)

Signal Name	Signal Function	Type (1)	Synch Async
PWRCLK	Power for MEMCLK Driver	MEMCLK Power	N/A
R/$\overline{W}$	Read/Write	Three-State Output	Synch
$\overline{RDN}$	Read Narrow	Input	Synch
$\overline{RDY}$	Data Ready	Input	Synch
$\overline{RESET}$	Reset	Input	Async
$\overline{REQ}$	Data Request	Three-State Output	Synch
STAT(2–0)	CPU Status	Output	Synch
SUP/$\overline{US}$	Supervisor/User Mode	Three-State Output	Synch
TCK	Test Clock Input	Input	Async
TDI	Test Data Input	Input	Synch*
TDO	Test Data Output	Three-State Output	Synch*
$\overline{TEST}$	Test Mode	Input	Async
TMS	Test Mode Select	Input	Synch*
$\overline{TRST}$	Test Reset	Input	Async
$\overline{TRAP}$(1–0)	Trap Request	Input	Async
$\overline{WARN}$	Warn	Edge-Sensitive Input	Async
$\overline{WBC}$	Reserved	Tied High	

(1) The signals labeled "Three-state output" and "bidirectional" (except MEMCLK) are disabled when the channel is granted to an external master. All outputs (except MSERR) may be disabled by asserting the $\overline{TEST}$ input, or through the IEEE1149.1-1990 (JTAG) Test Access Port.

(*) The signals TDI, TDO and TMS are all Synchronous to the Test Clock Input (TCK).

Figure A-1 Relationship of INCLK, Internal Processor Clock, and MEMCLK



Note: The level applied to PWRCLK does not affect the relationship of the signals depicted above.

Figure A-2 Processor Reset

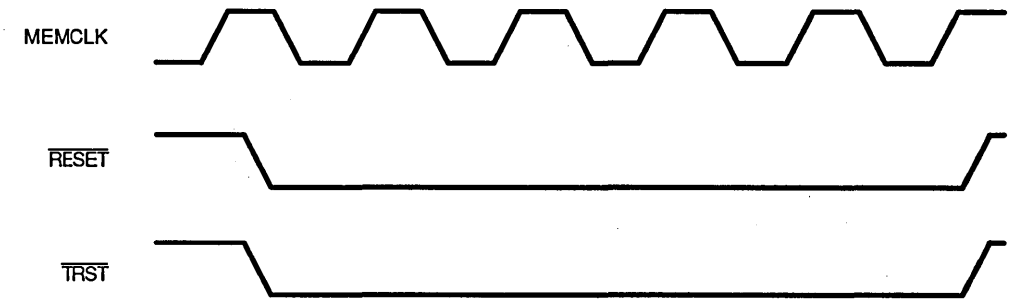


Figure A-3 Processor Reset—8-Bit Narrow Read Interface

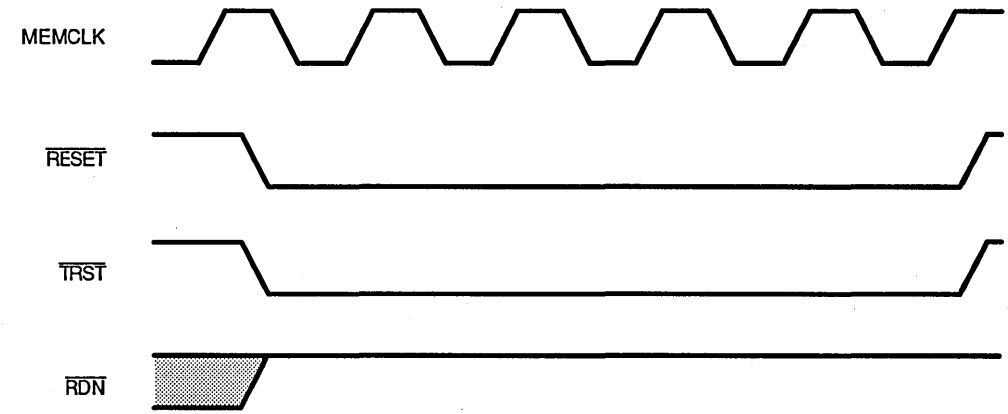


Figure A-4 Processor Reset—16-Bit Narrow Read Interface

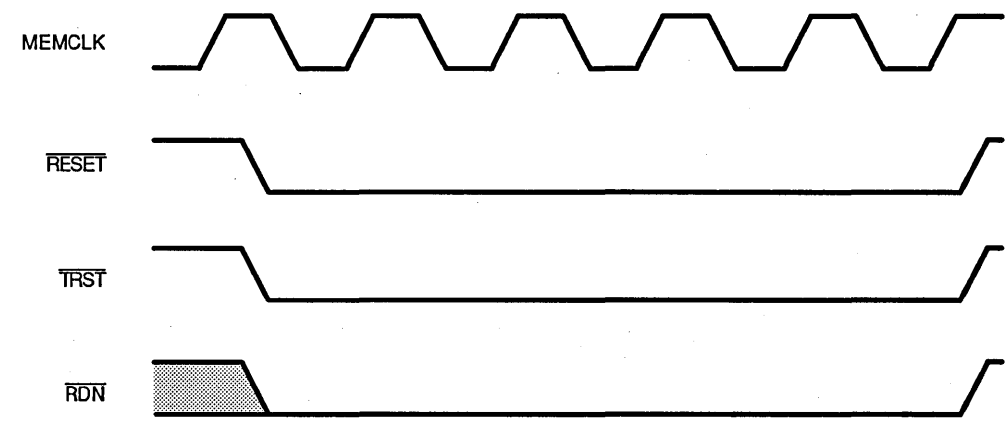
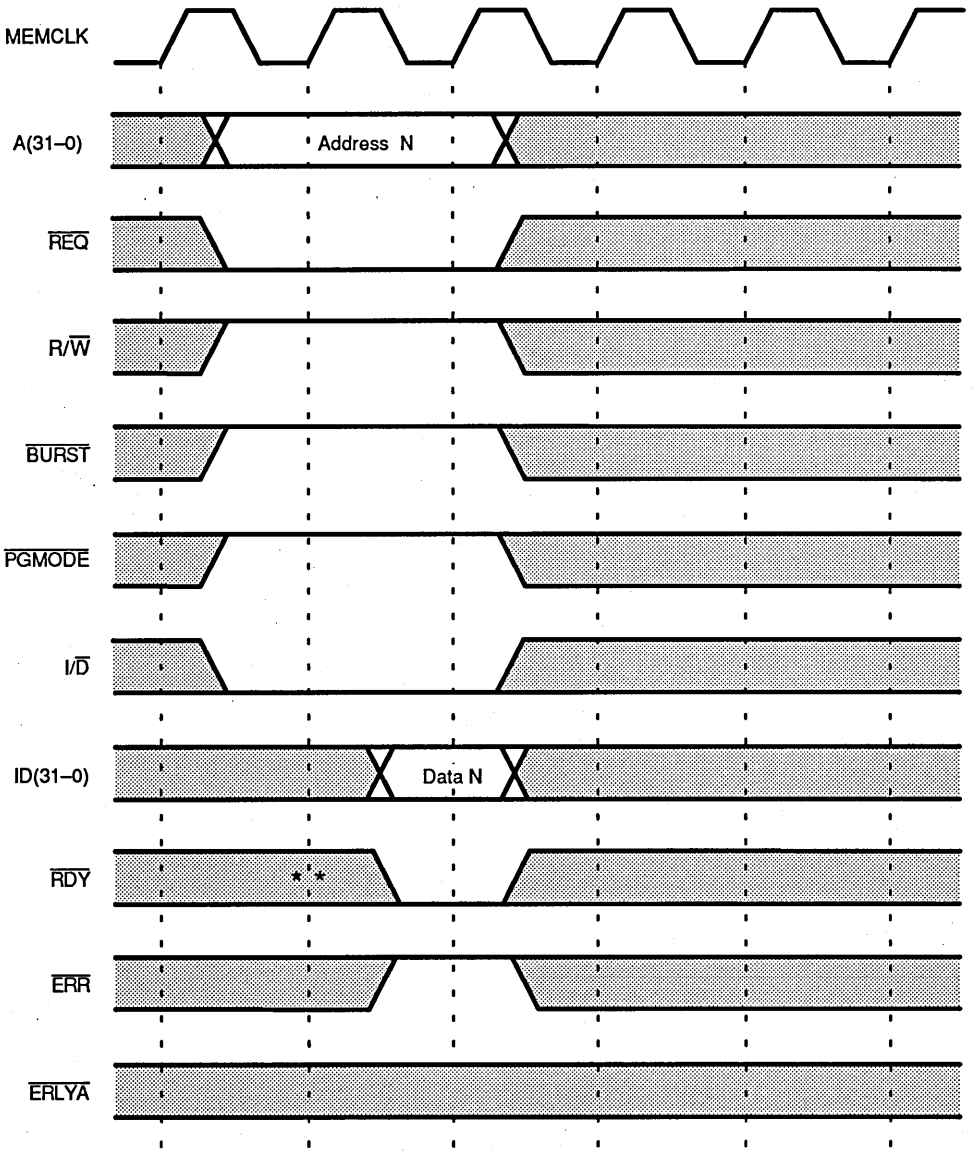
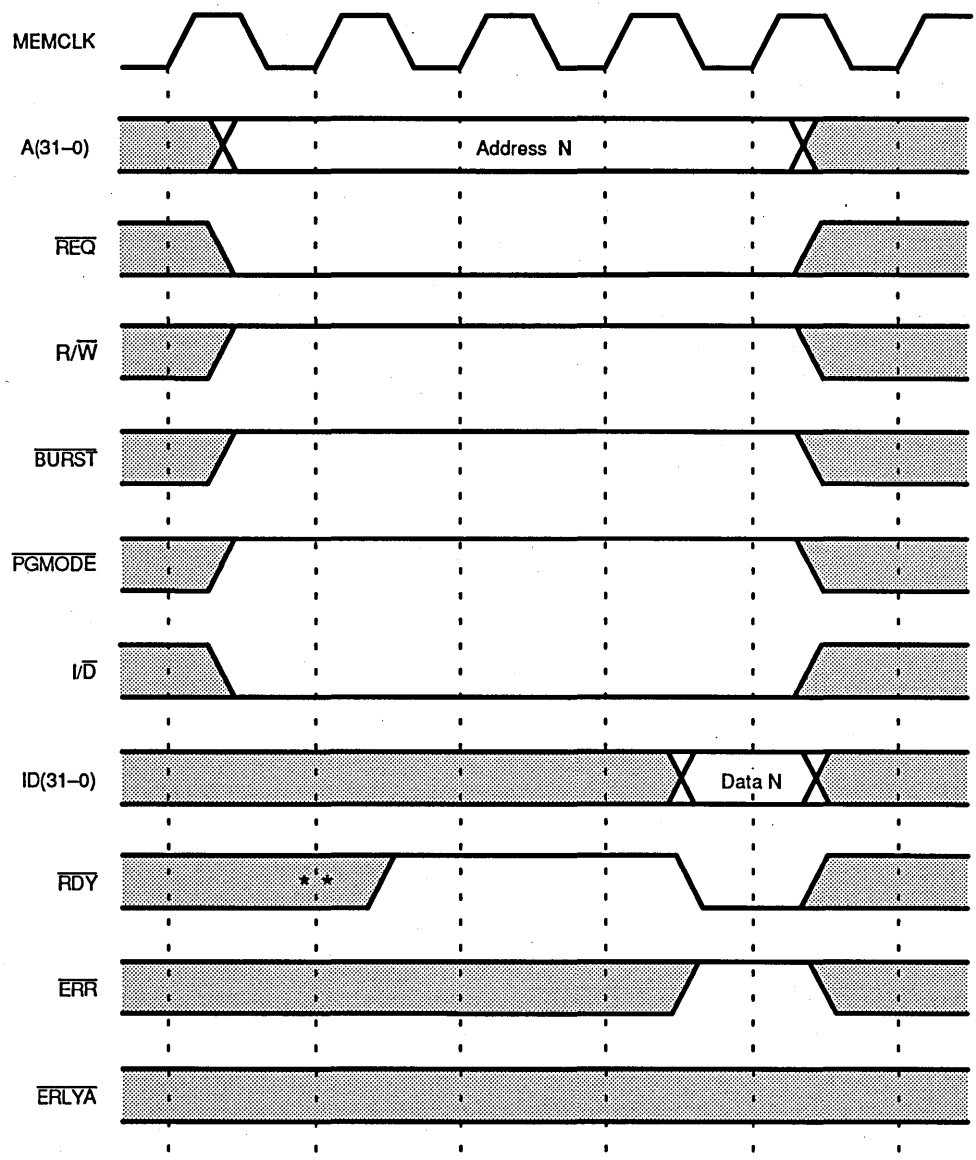


Figure A-5 Simple Data Read Access



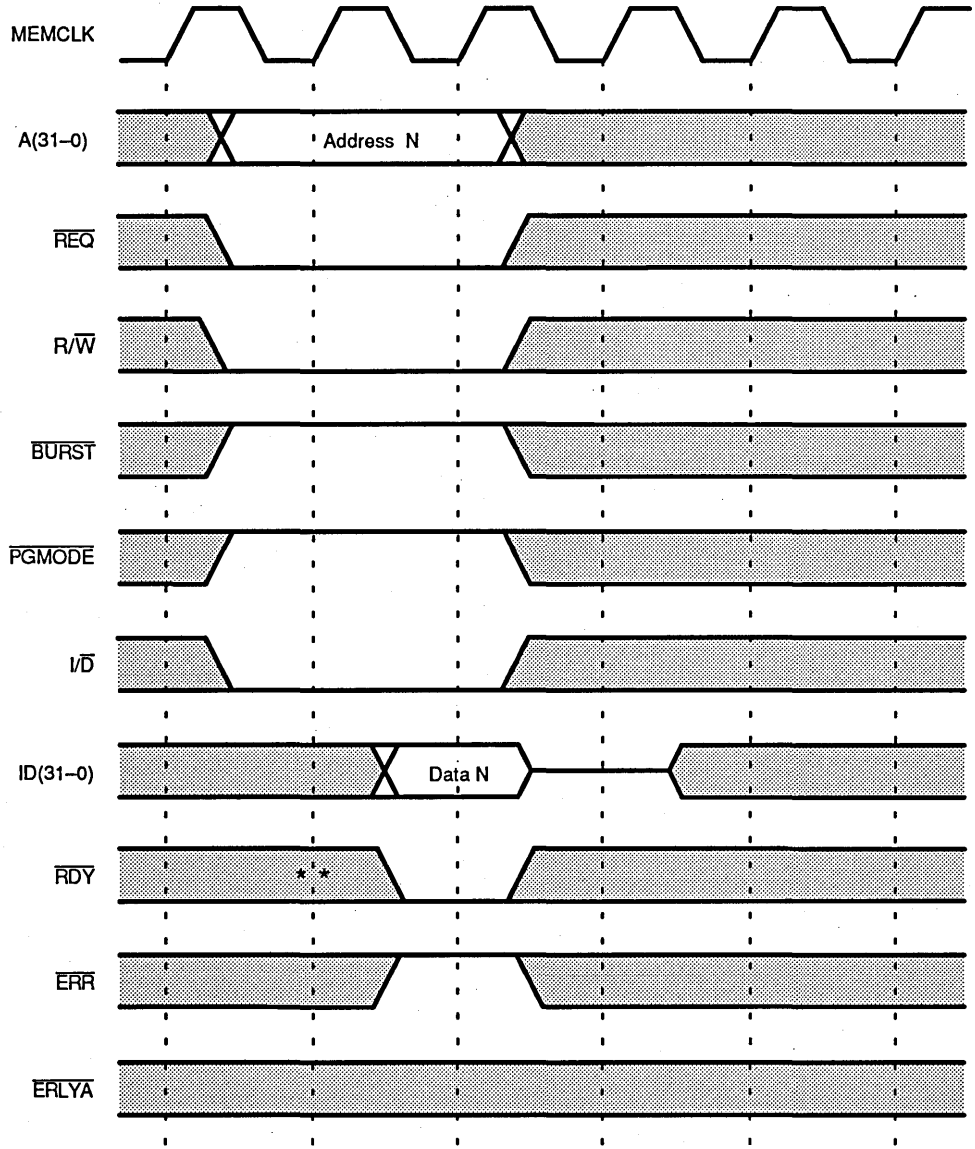
** The RDY signal is always ignored in the first cycle of all simple accesses and the first cycle of all initial burst-mode accesses.

Figure A-6 Simple Data Read Access (Multi-Cycle)



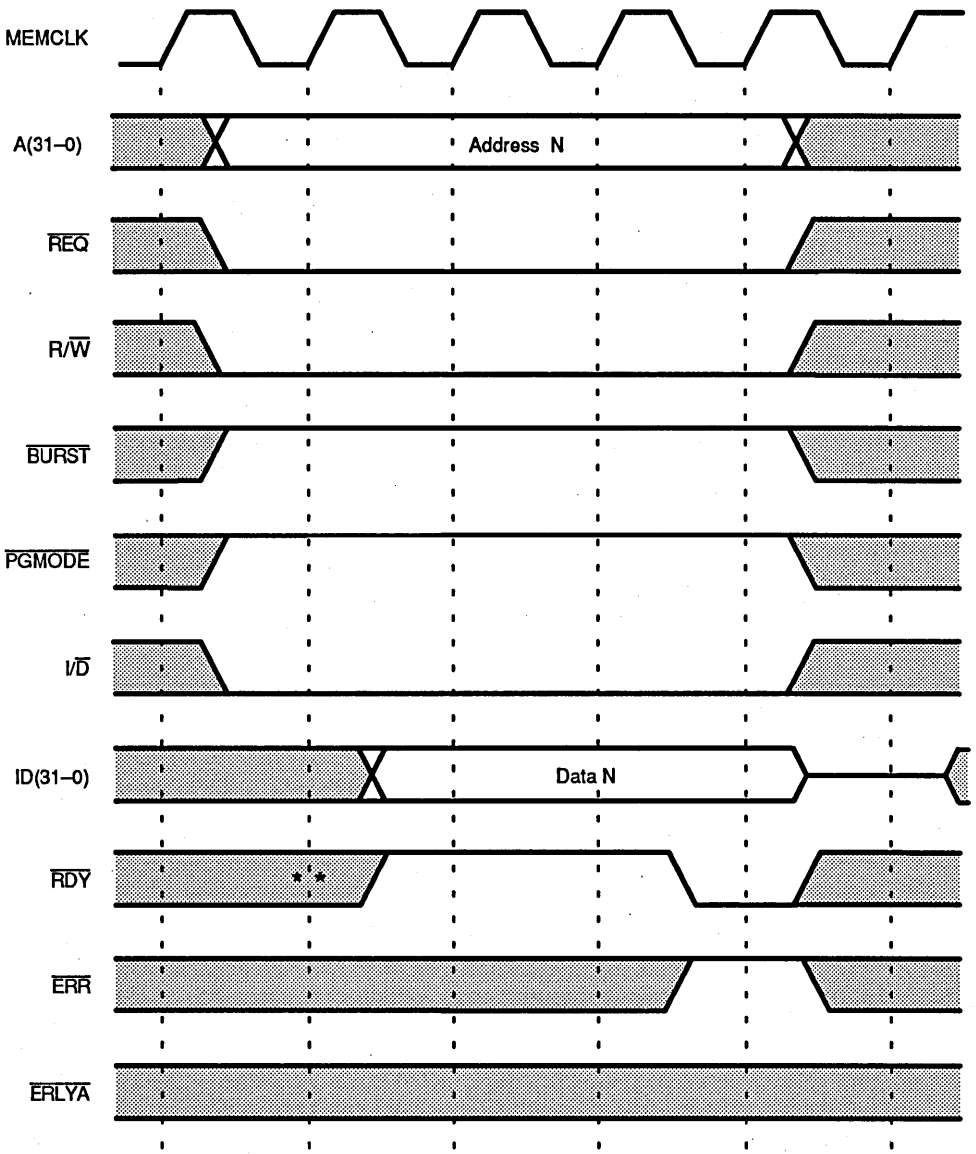
* * The RDY signal is always ignored in the first cycle of all simple accesses and the first cycle of all initial burst-mode accesses.

Figure A-7 Simple Data Write Access



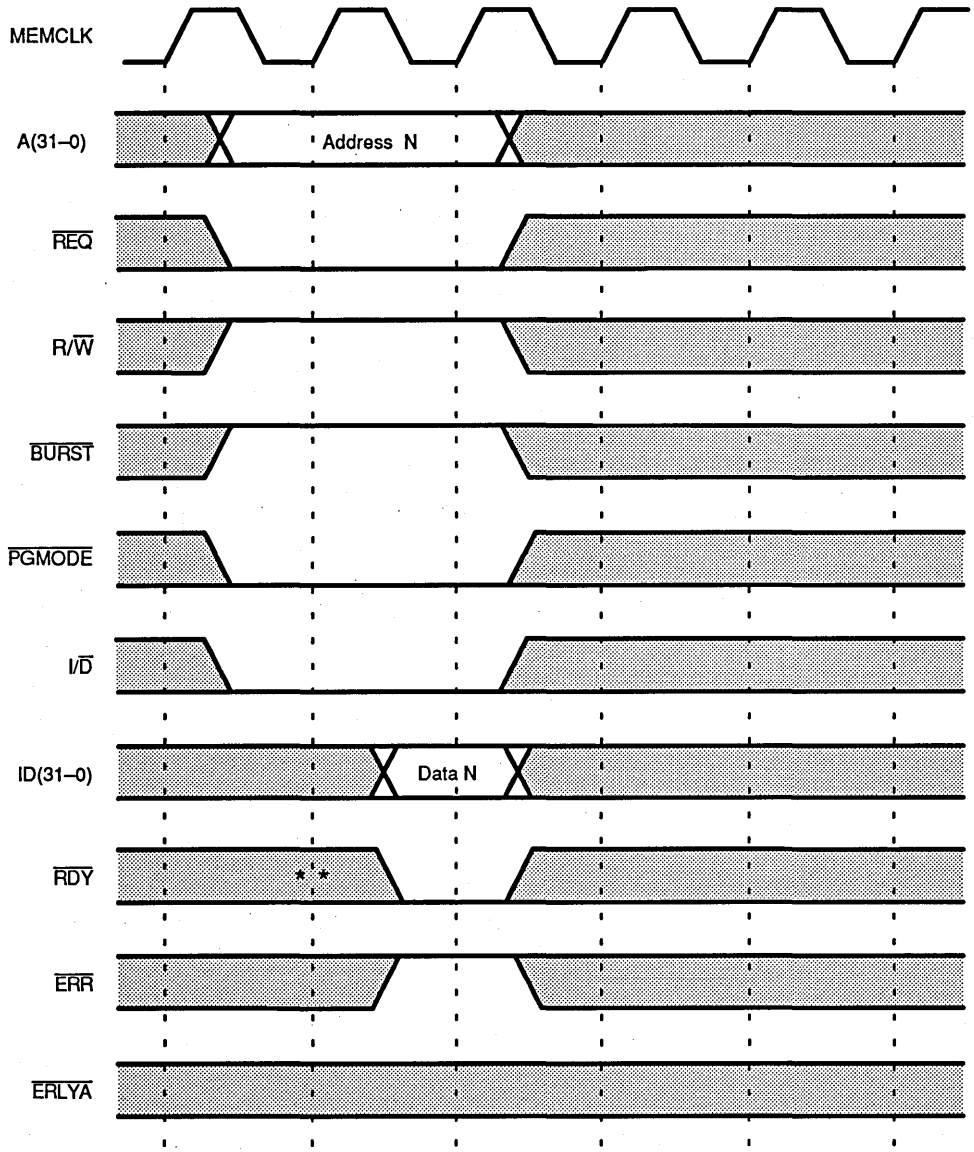
** The RDY signal is always ignored in the first cycle of all simple accesses and the first cycle of all initial burst-mode accesses.

Figure A-8 Simple Data Write Access (Multi-Cycle)



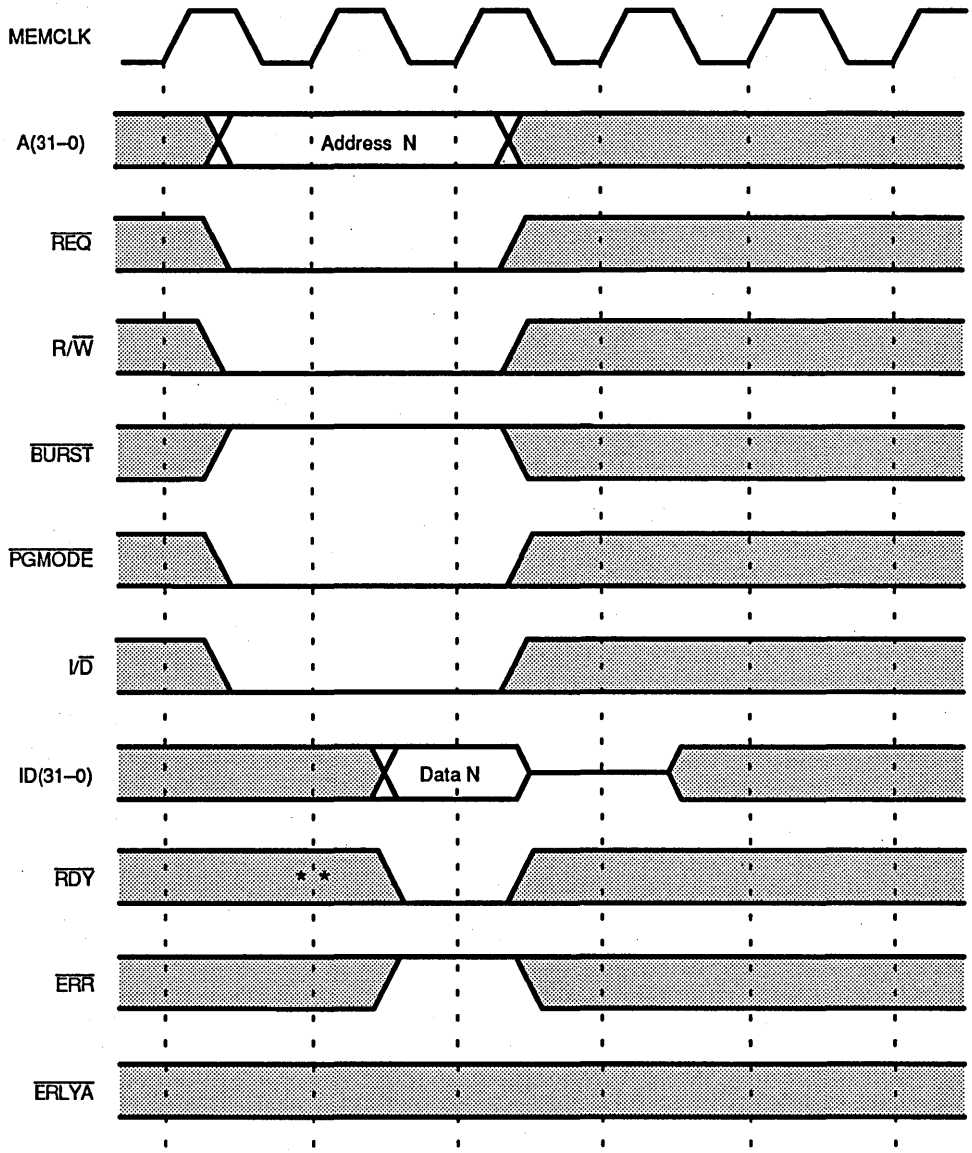
** The RDY signal is always ignored in the first cycle of all simple accesses and the first cycle of all initial burst-mode accesses.

Figure A-9 Page-Mode Read Access



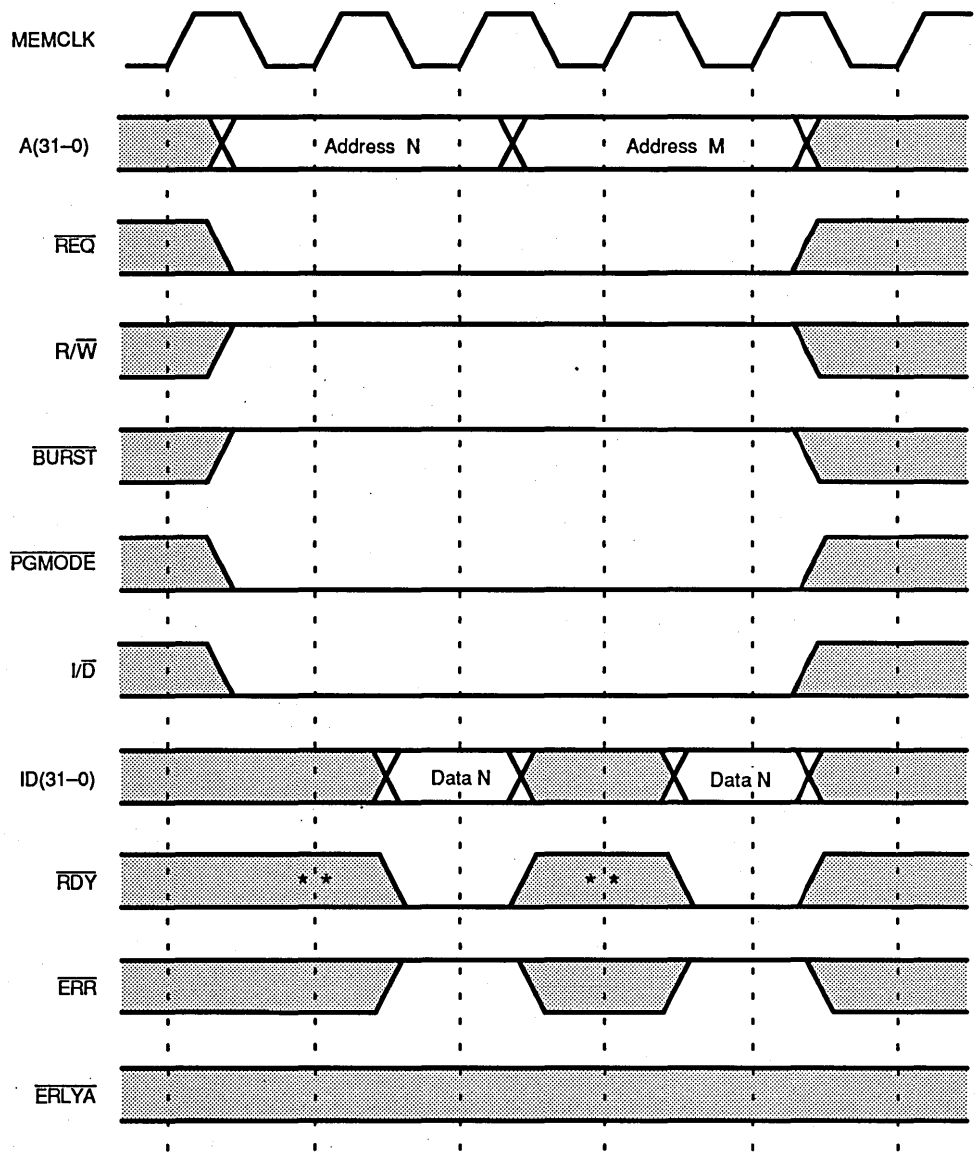
** The RDY signal is always ignored in the first cycle of all simple accesses and the first cycle of all initial burst-mode accesses.

Figure A-10 Page-Mode Write Access



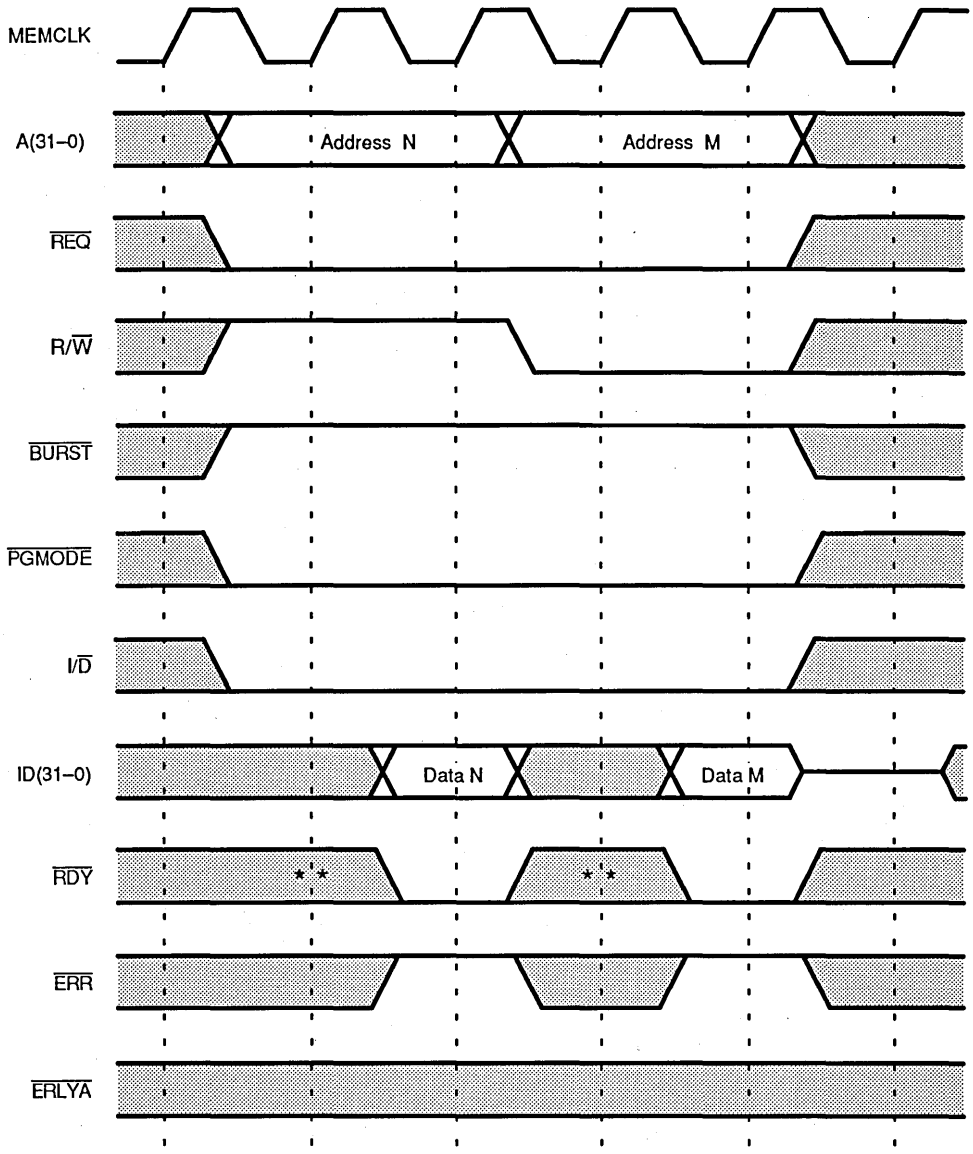
** The RDY signal is always ignored in the first cycle of all simple accesses and the first cycle of all initial burst-mode accesses.

Figure A-11 Read Access Followed by a Read Access (Page-Mode)



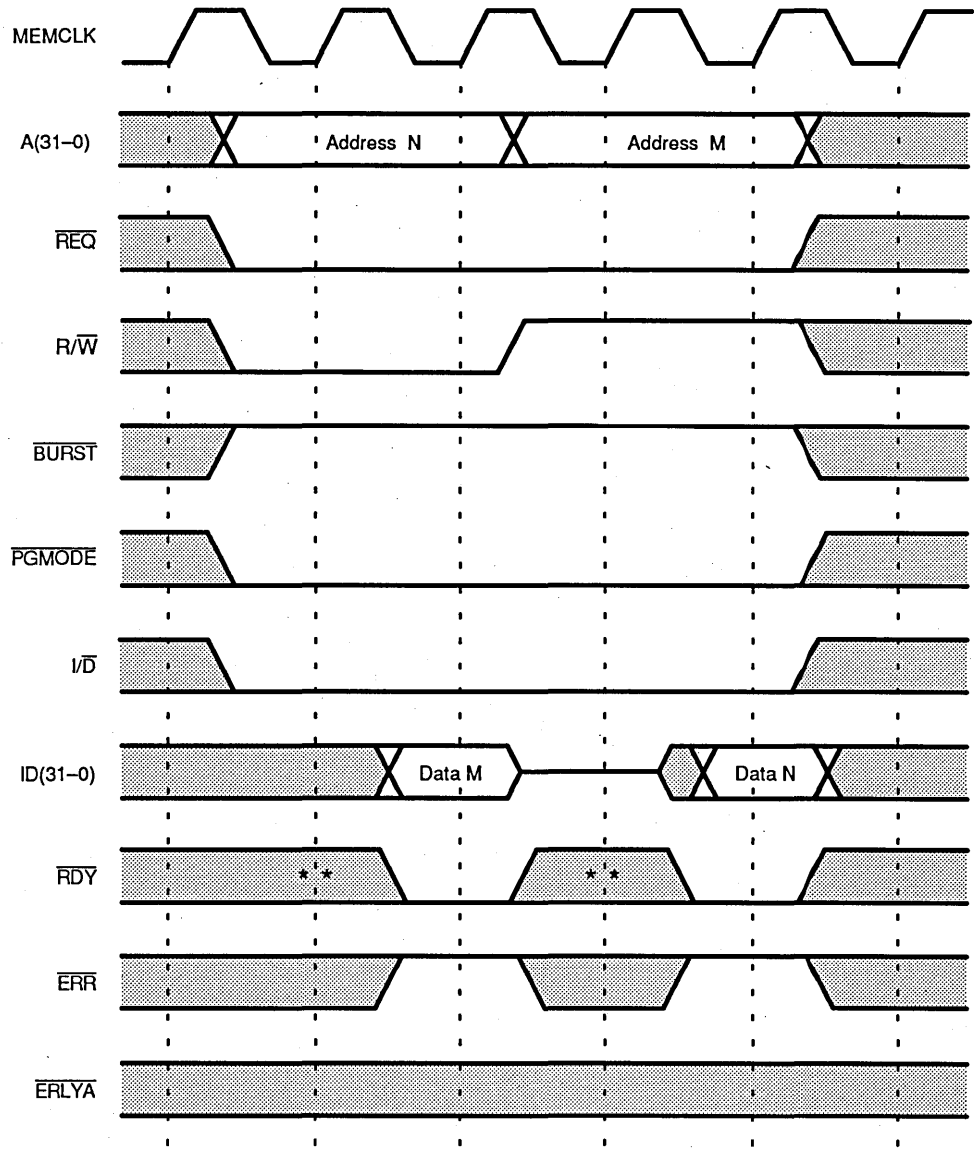
** The RDY signal is always ignored in the first cycle of all simple accesses and the first cycle of all initial burst-mode accesses.

Figure A-12 Read Access Followed by a Write Access (Page-Mode)



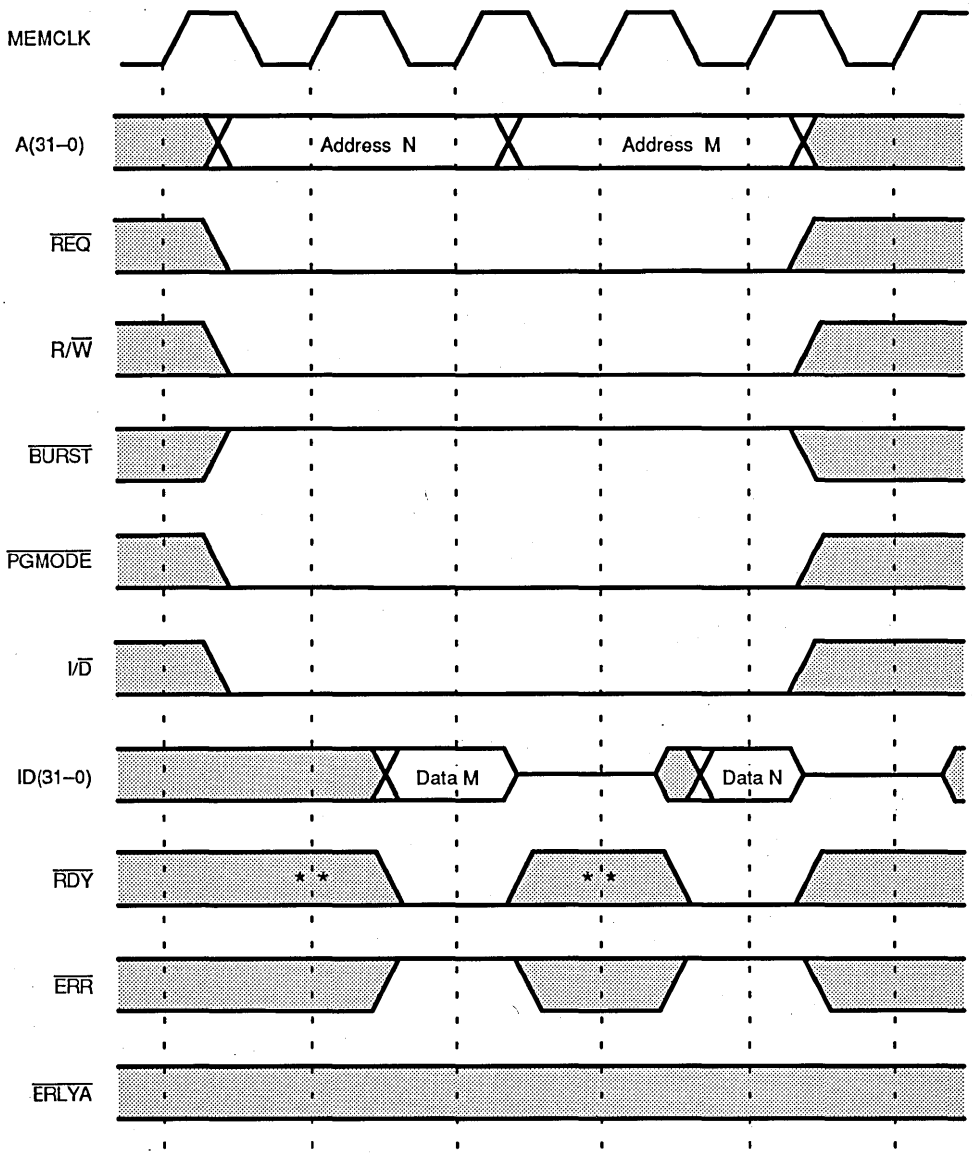
** The RDY signal is always ignored in the first cycle of all simple accesses and the first cycle of all initial burst-mode accesses.

Figure A-13 Write Access Followed by a Read Access (Page-Mode)



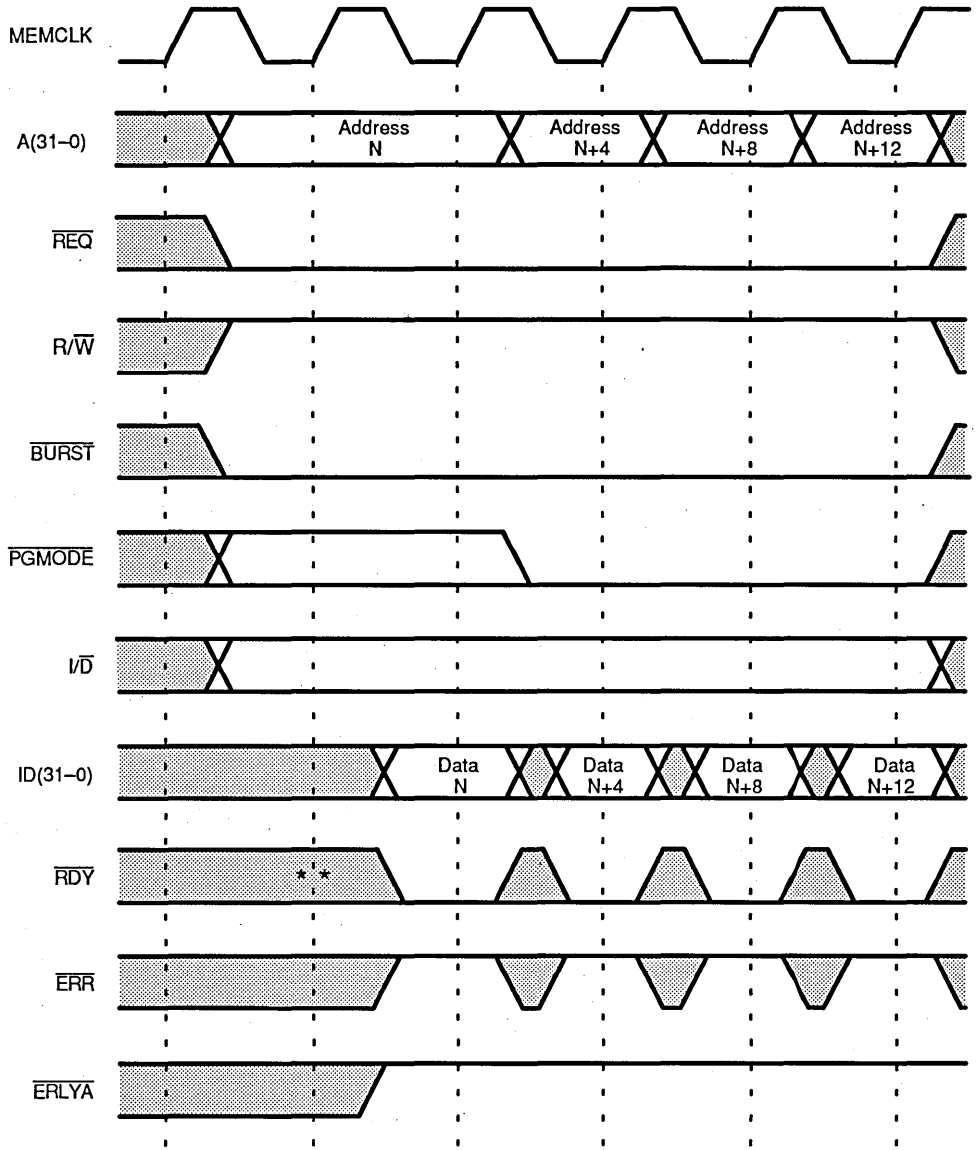
** The RDY signal is always ignored in the first cycle of all simple accesses and the first cycle of all initial burst-mode accesses.

Figure A-14 Write Access Followed by a Write Access (Page-Mode)



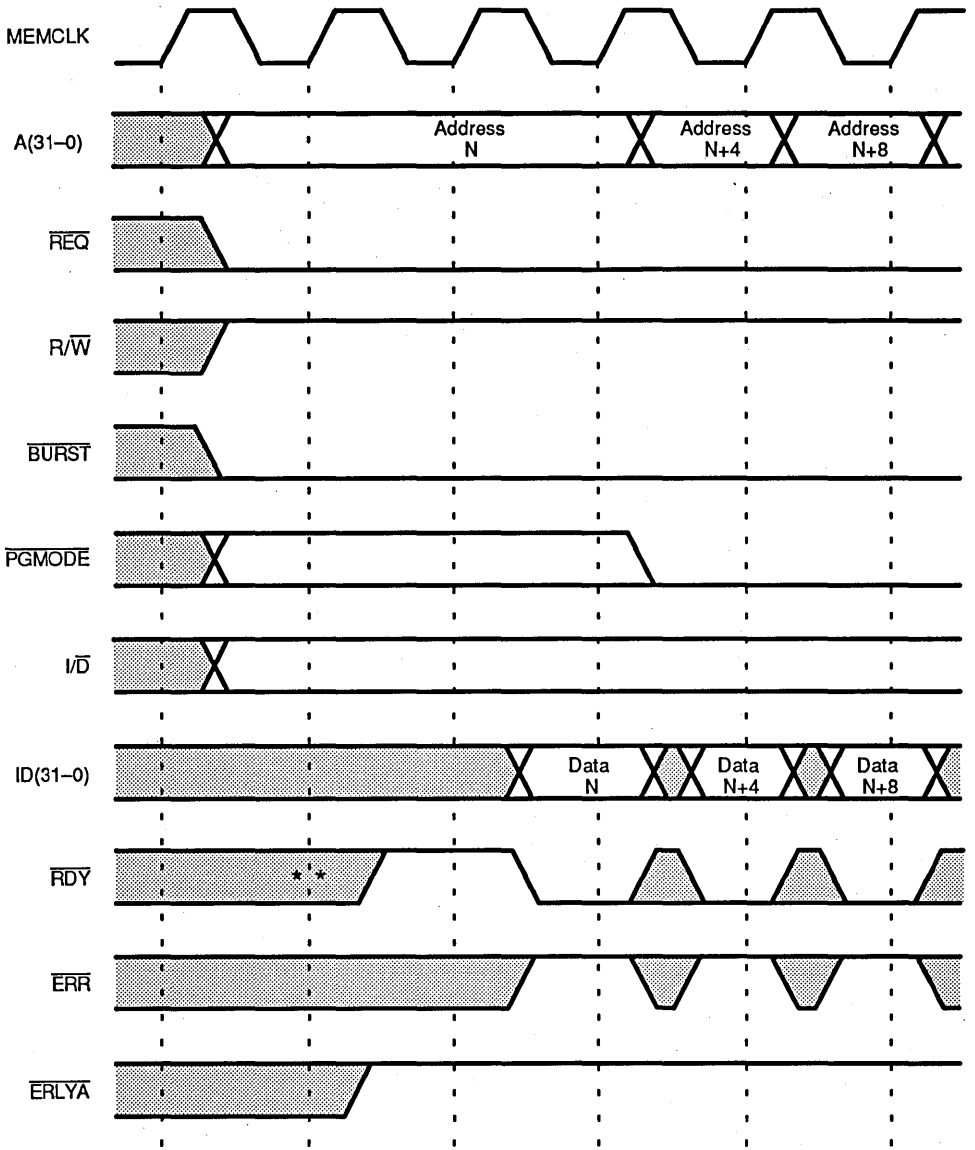
** The RDY signal is always ignored in the first cycle of all simple accesses and the first cycle of all initial burst-mode accesses.

Figure A-15 Burst-Mode Read Access



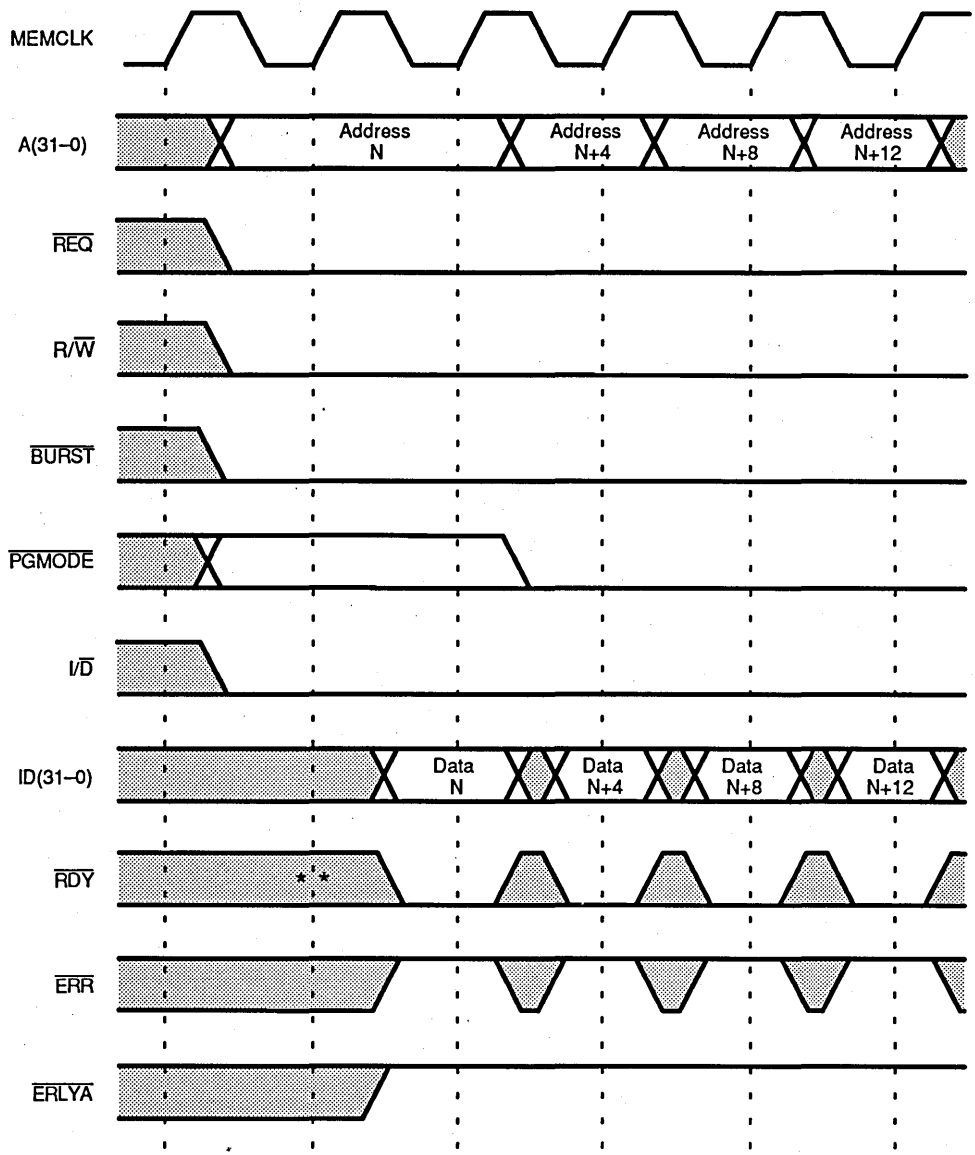
** The RDY signal is always ignored in the first cycle of all simple accesses and the first cycle of all initial burst-mode accesses.

Figure A-16 Burst-Mode Read Access (Multi-Cycle Initial Access)



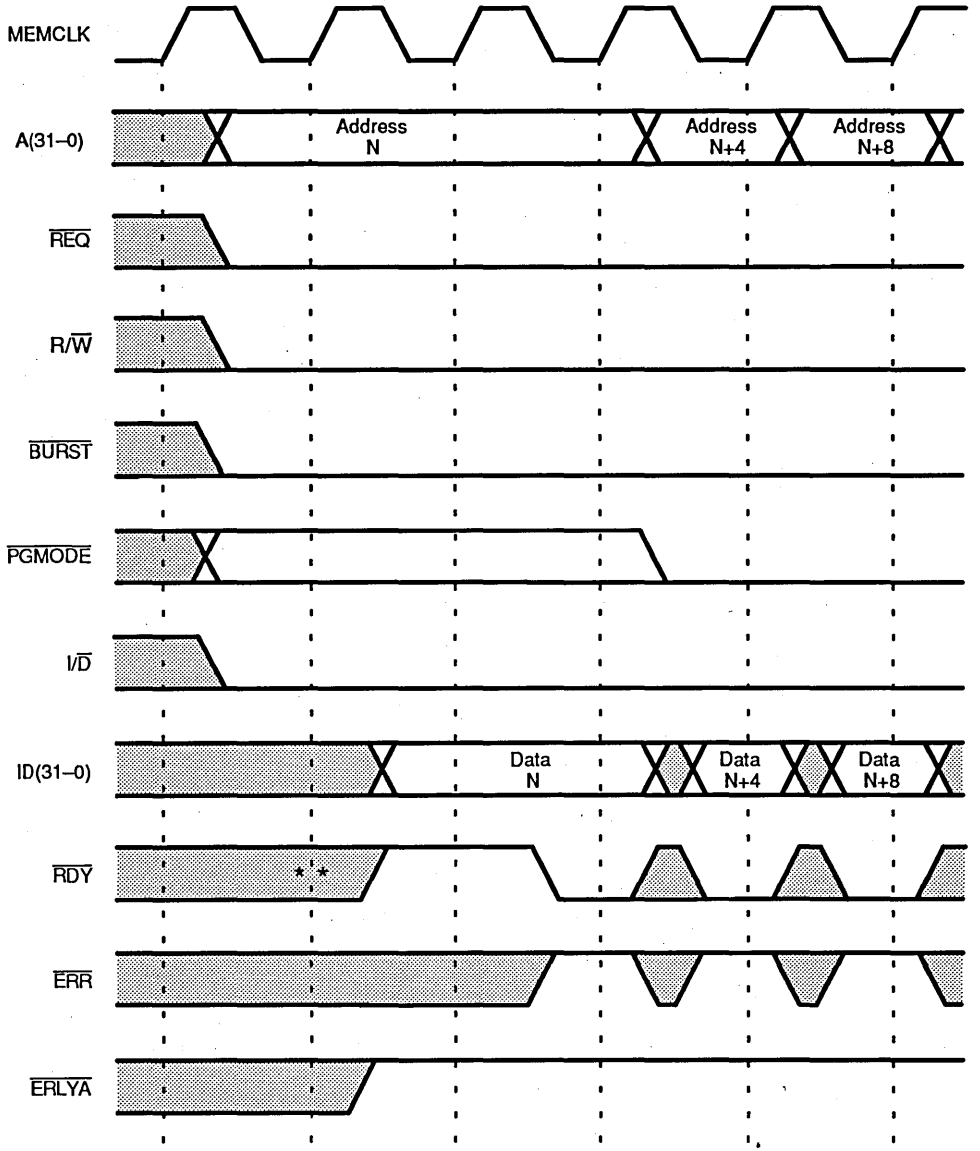
** The RDY signal is always ignored in the first cycle of all simple accesses and the first cycle of all initial burst-mode accesses.

Figure A-17 Burst-Mode Write Access



** The RDY signal is always ignored in the first cycle of all simple accesses and the first cycle of all initial burst-mode accesses.

Figure A-18 Burst-Mode Write Access (Multi-Cycle Initial Access)



**. The RDY signal is always ignored in the first cycle of all simple accesses and the first cycle of all initial burst-mode accesses.

Figure A-19 Processor Preemption, Termination or Cancellation of a Burst-Mode Read Access

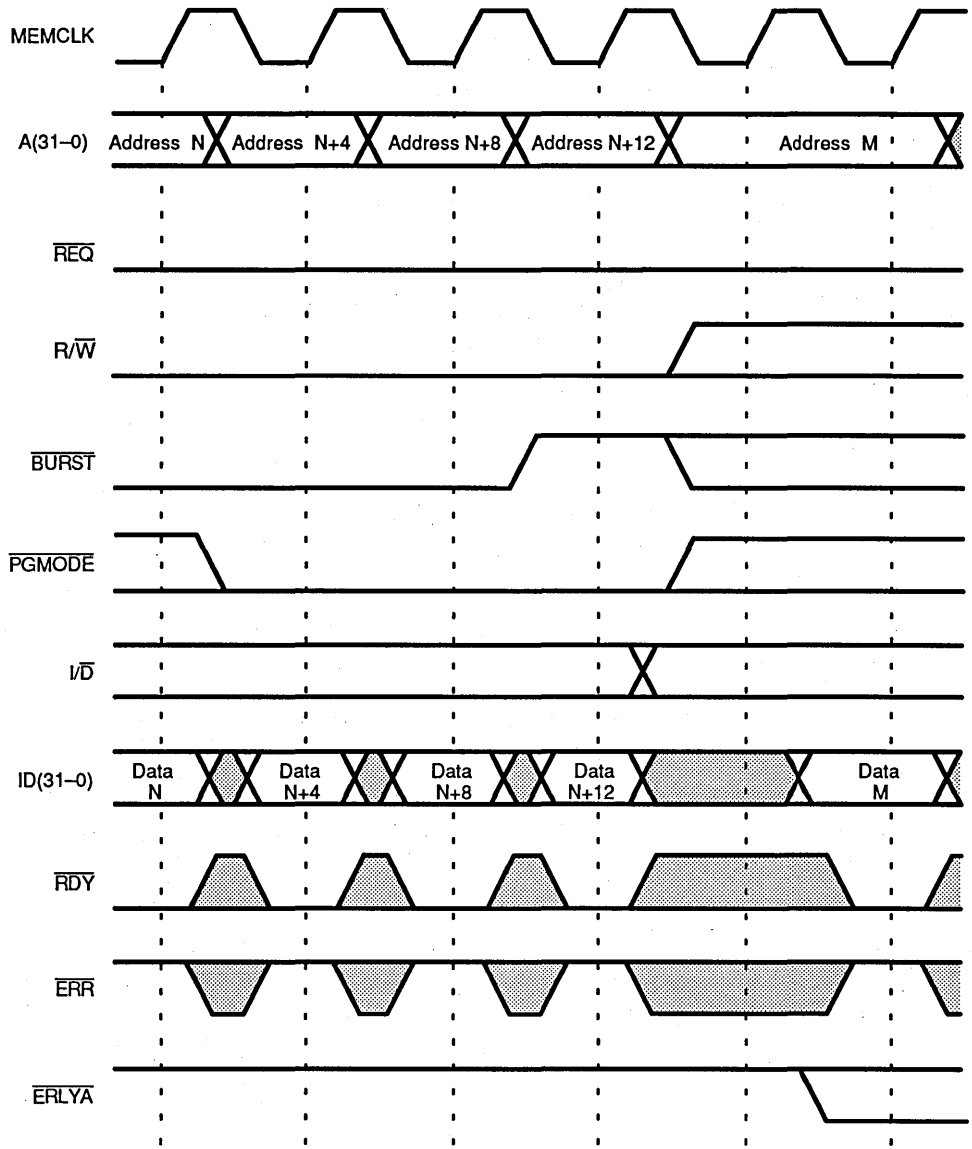


Figure A-20 **Processor Preemption, Termination or Cancellation of a Burst-Mode Write Access**

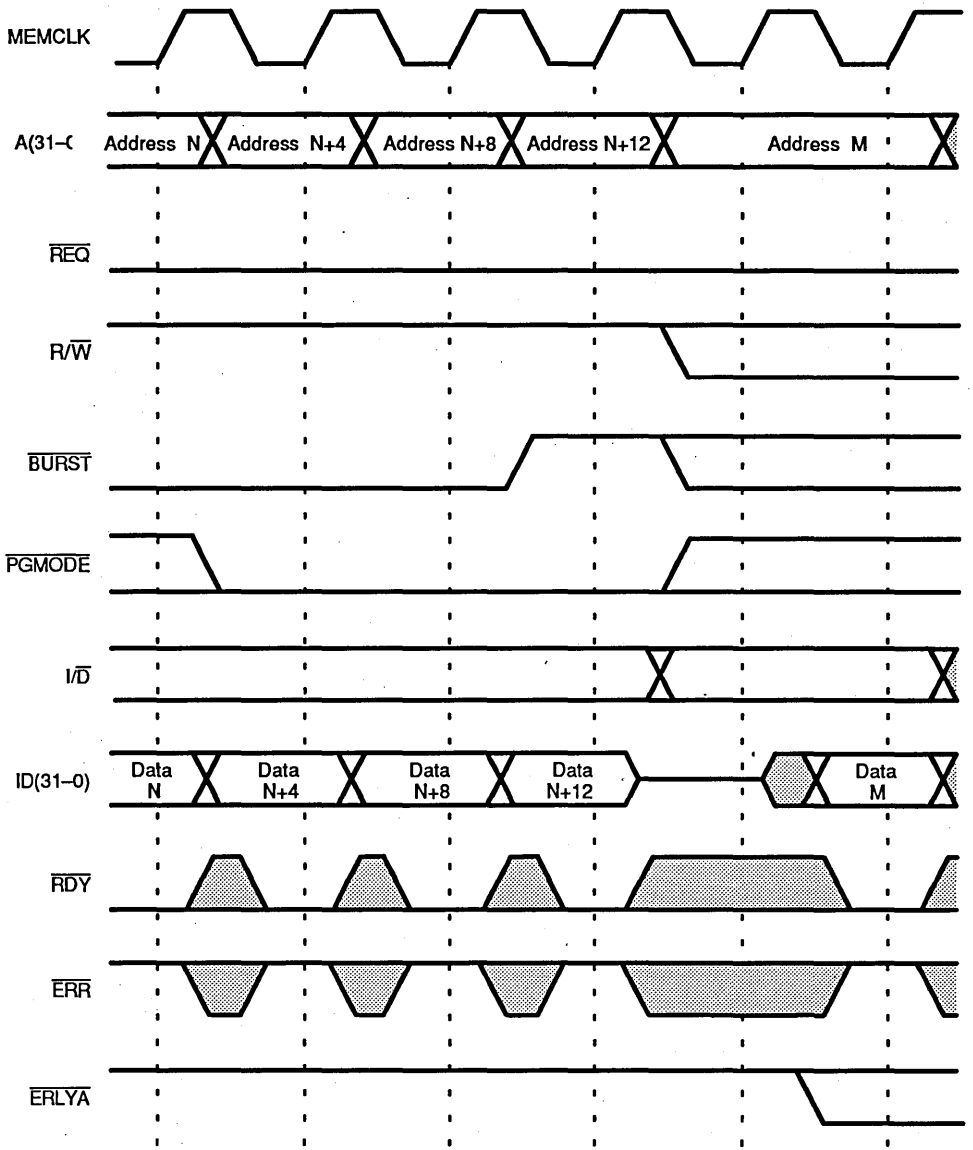
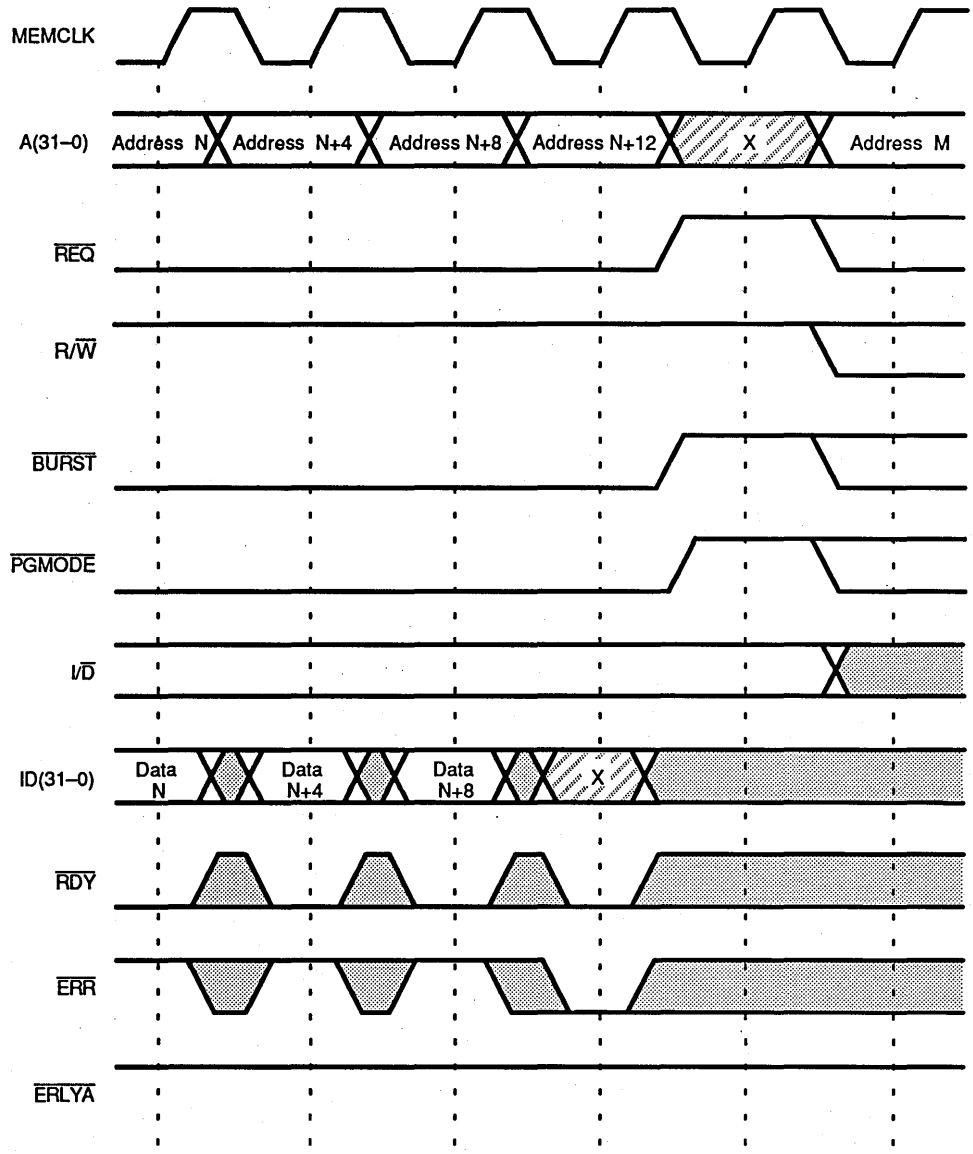
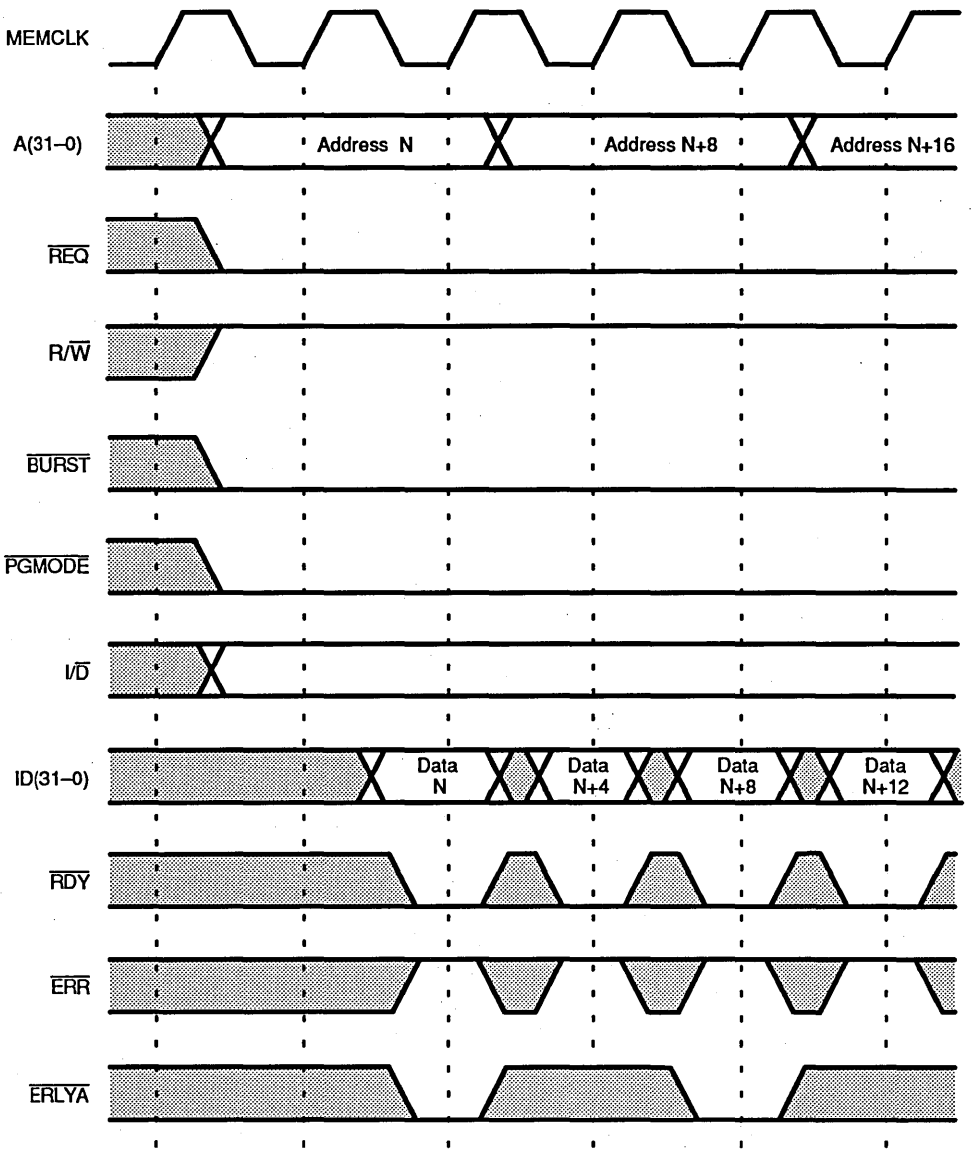


Figure A-21 Slave Cancellation of a Burst-Mode Read Access



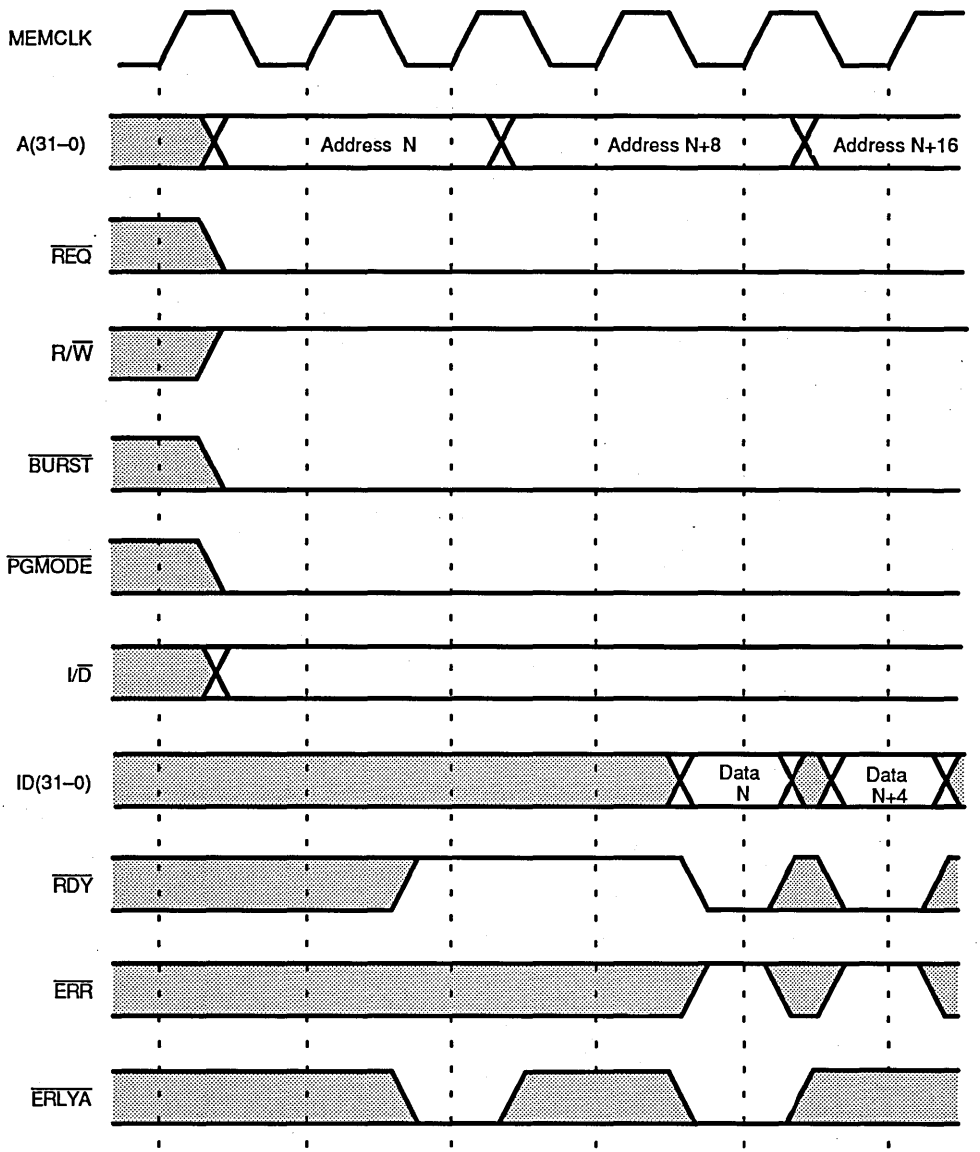
Note: This may cause an Instruction Access Exception trap or a Data Access Exception trap.

Figure A-22 $\overline{\text{ERLYA}}$ Burst-Mode Read Access



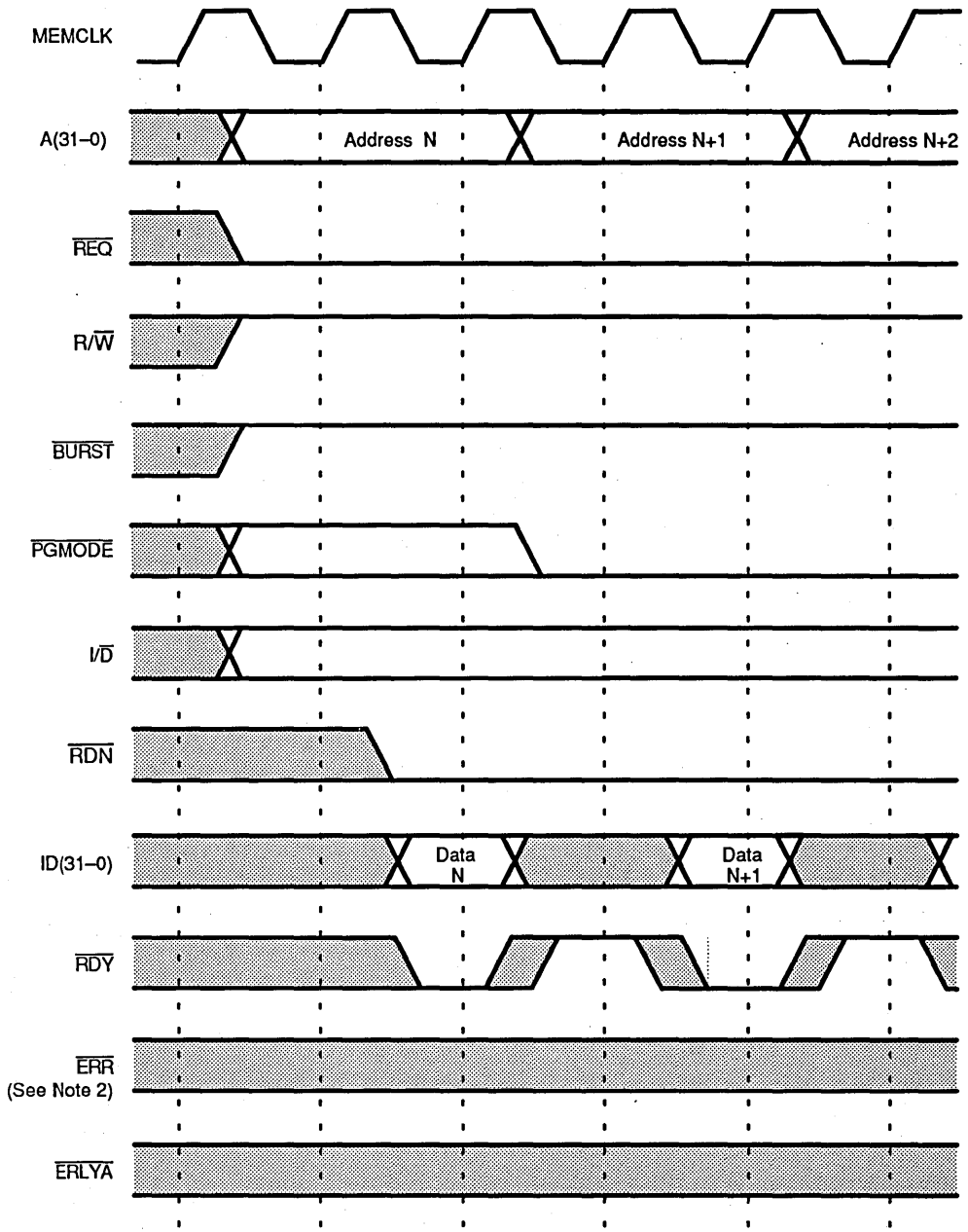
Note: Two-way interleaved example (page-mode assumed) In this example, the memory is capable of responding in each cycle after the initial access.

Figure A-23 $\overline{\text{ERLYA}}$ Burst-Mode Read Access (Multi-Cycle)



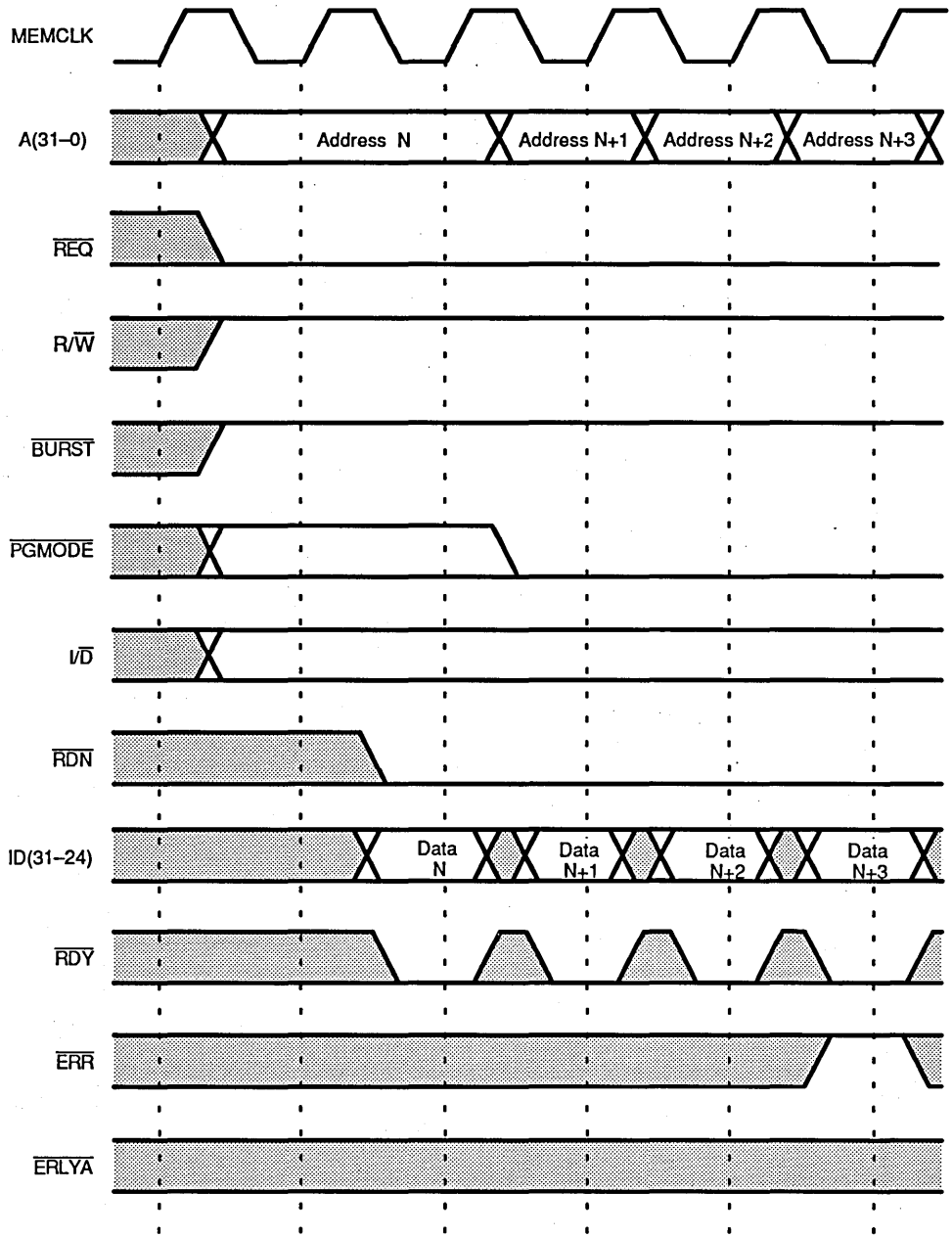
Note: Four-way interleaved example. This example assumes that the initial burst-mode access is a page-mode access with a latency of four cycles.

Figure A-24 Simple 8-Bit Narrow Read Word Access



Note 1: A total of four accesses occur—the final two accesses are not shown
Note 2: The ERR response is relevant only for the final access

Figure A-25 Simple 8-Bit Narrow Read Access with Fast Subsequent Accesses



Note: The narrow memory can perform a burst-mode access, even if the processor does not request one, by responding in every cycle after the first. However, if the processor is not requesting a burst-mode access, the memory must be aware of the termination point (the final access), because termination is not indicated by the processor.

Figure A-26 Burst-Mode 8-Bit Narrow Read Access

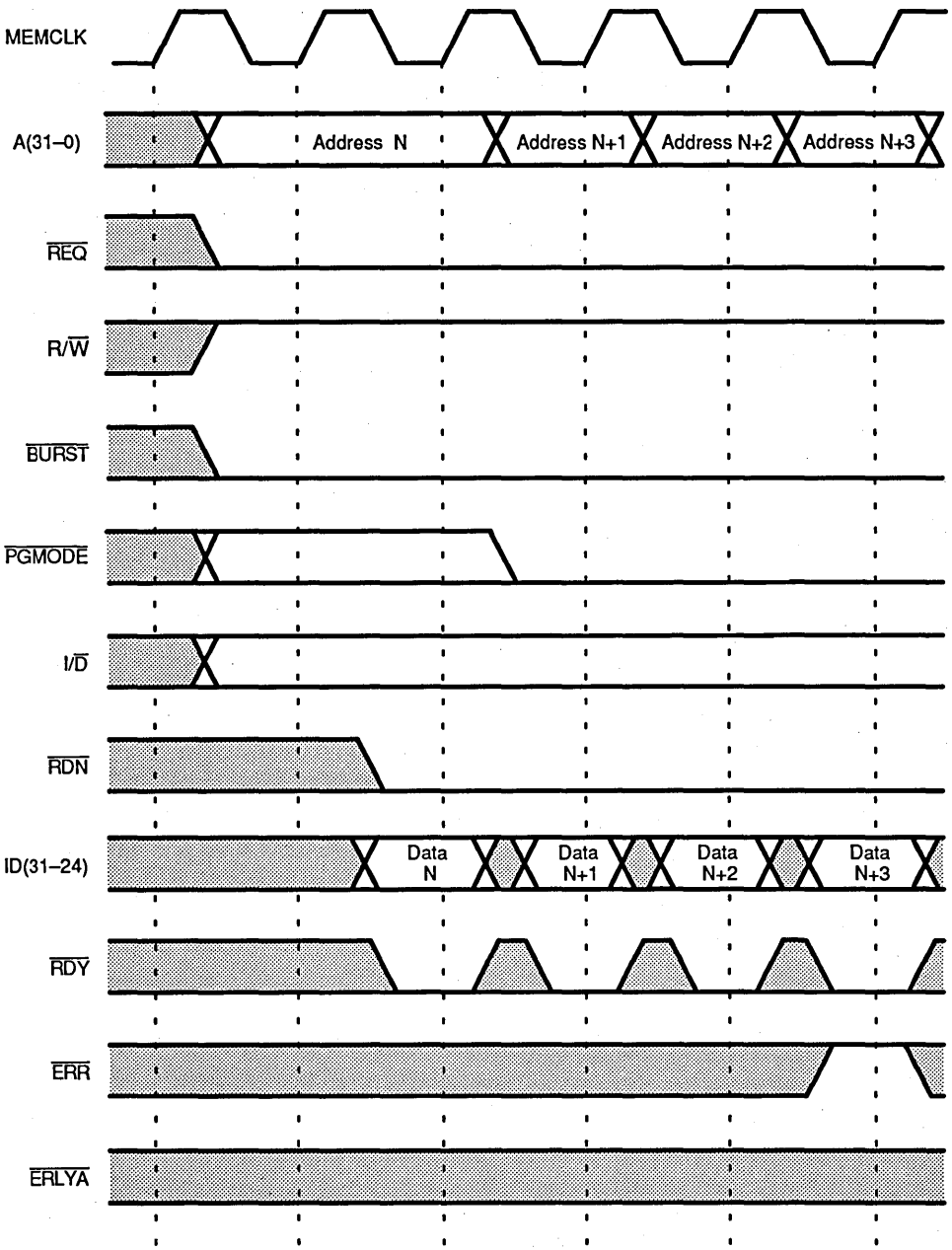


Figure A-27 Simple 16-Bit Narrow Read Word Access

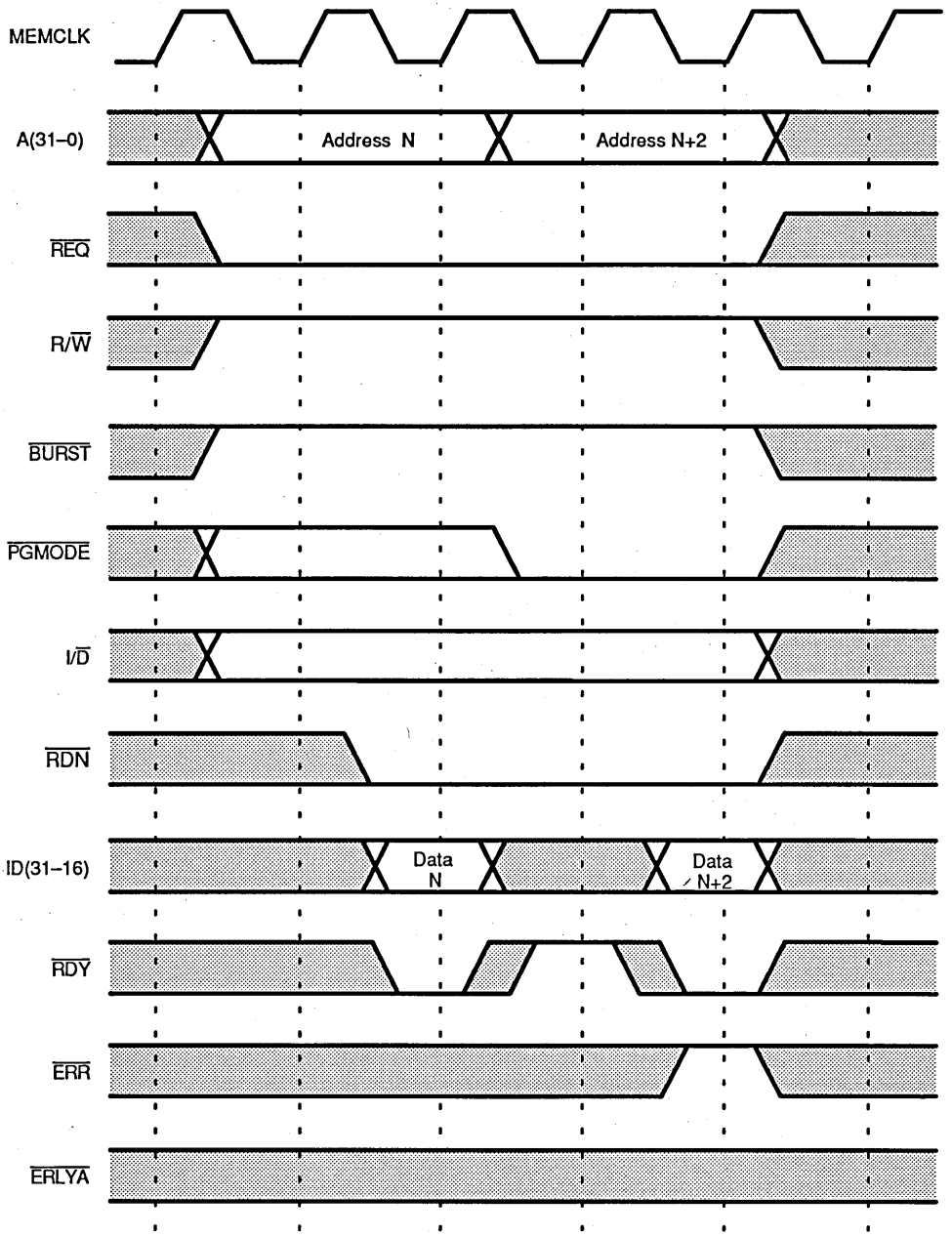
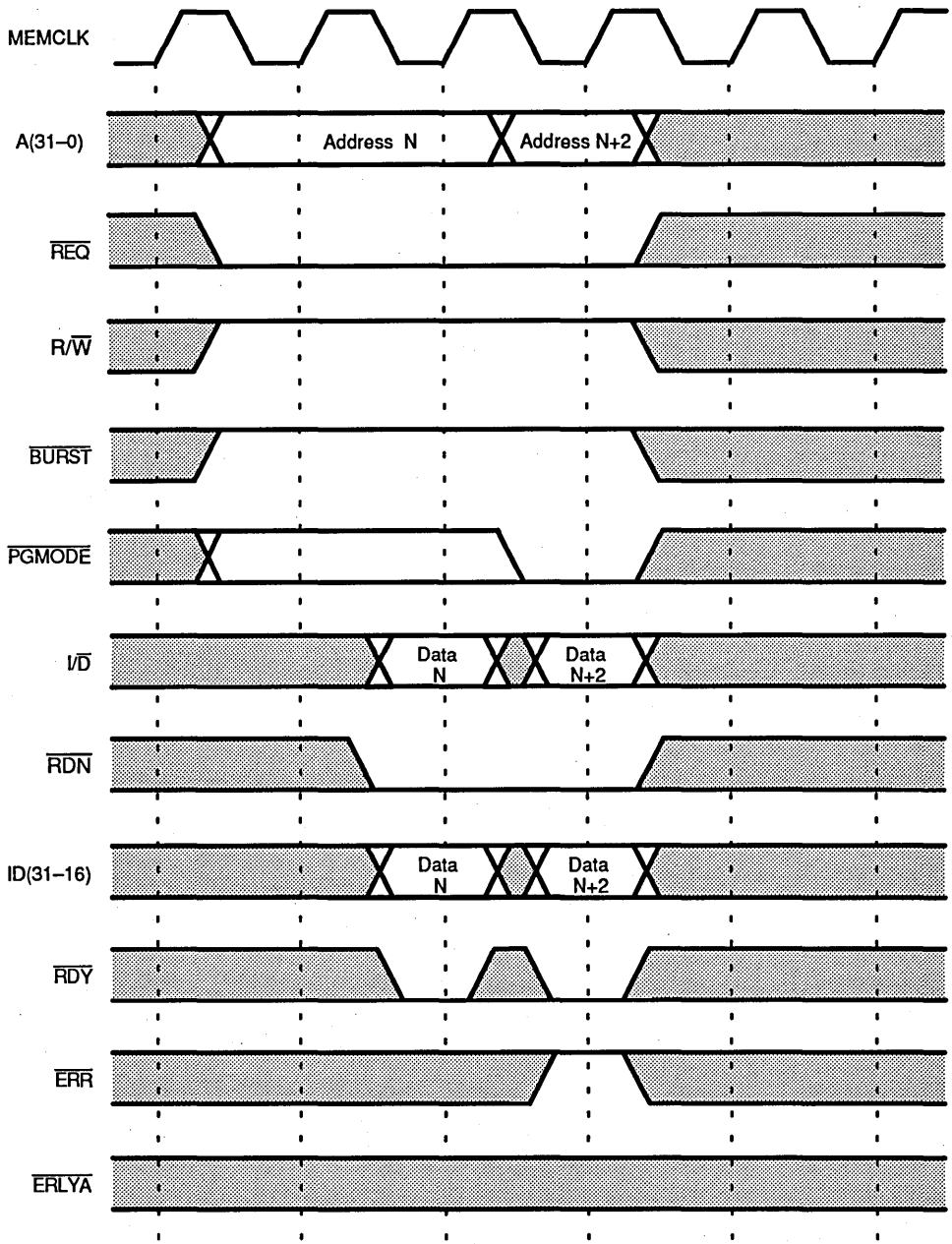


Figure A-28 Simple 16-Bit Narrow Read Word Access with Fast Second Access



Note: The narrow memory can perform a burst-mode access, even if the processor does not request one, by responding in every cycle after the first. However, if the processor is not requesting a burst-mode access, the memory must be aware of the termination point (the final access), because termination is not indicated by the processor.

Figure A-29 Burst-Mode 16-Bit Narrow Read Access

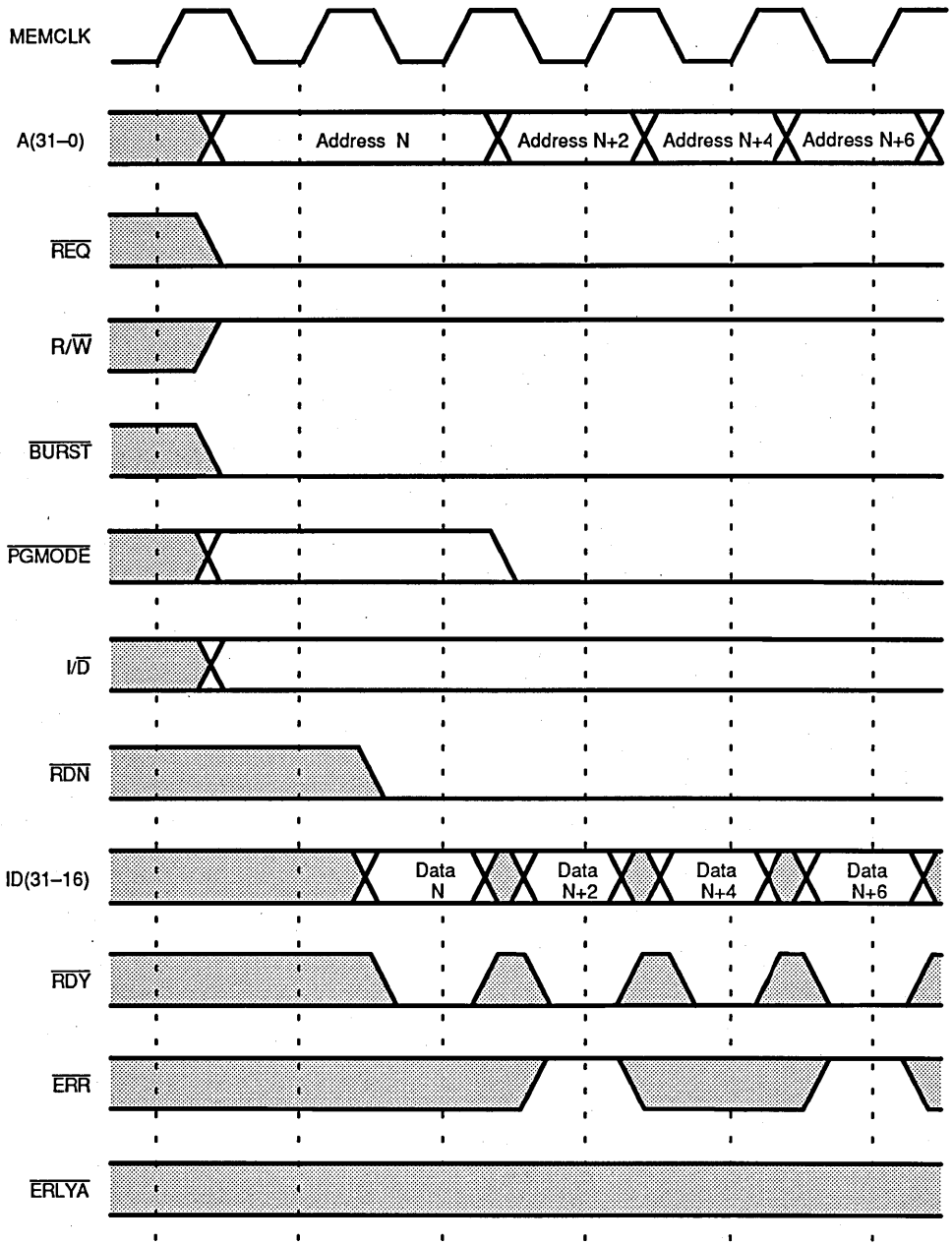


Figure A-30 Simple 16-Bit Narrow Write Word Access (Am29035 Microprocessor only)

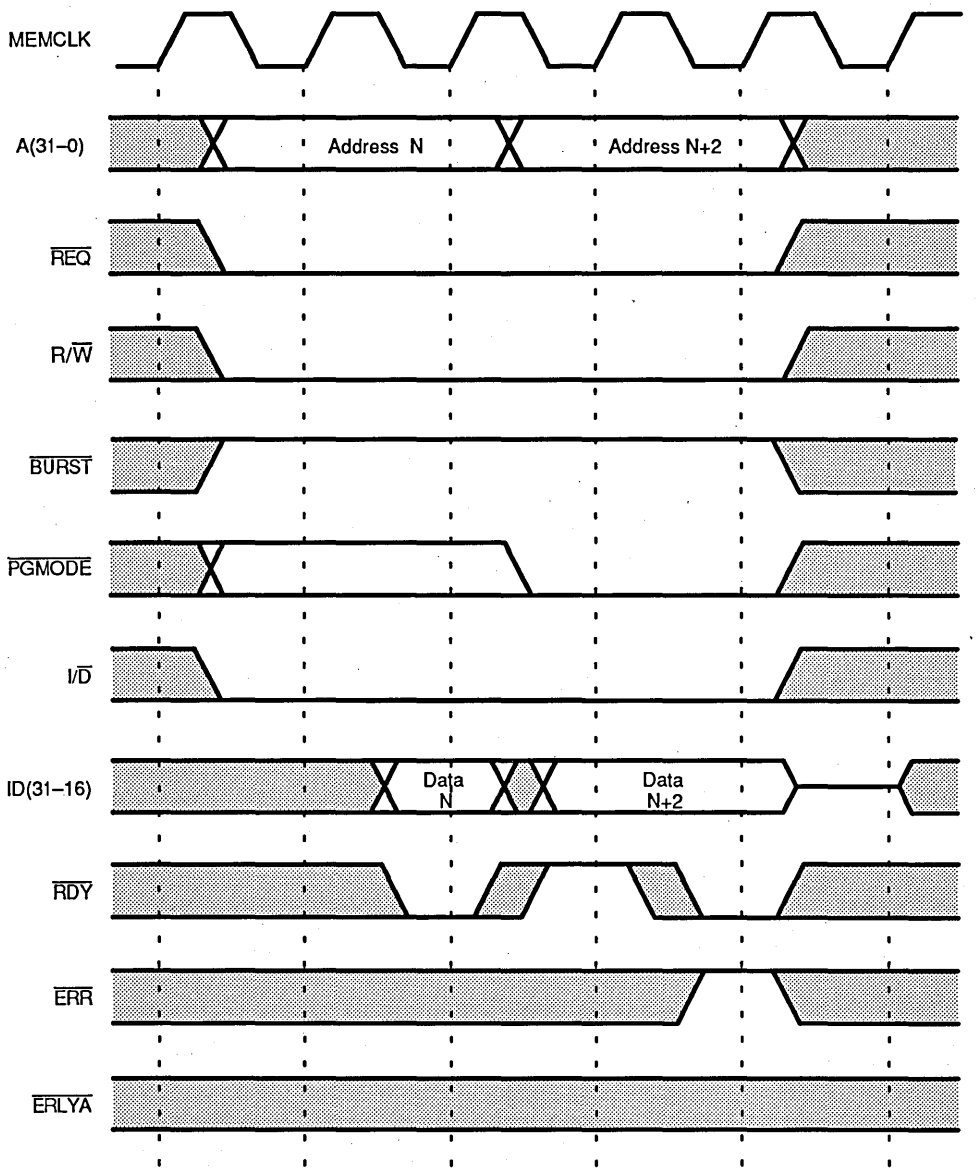


Figure A-31 Simple 16-Bit Narrow Write Word Access with Fast Subsequent Access (Am29035 Microprocessor only)

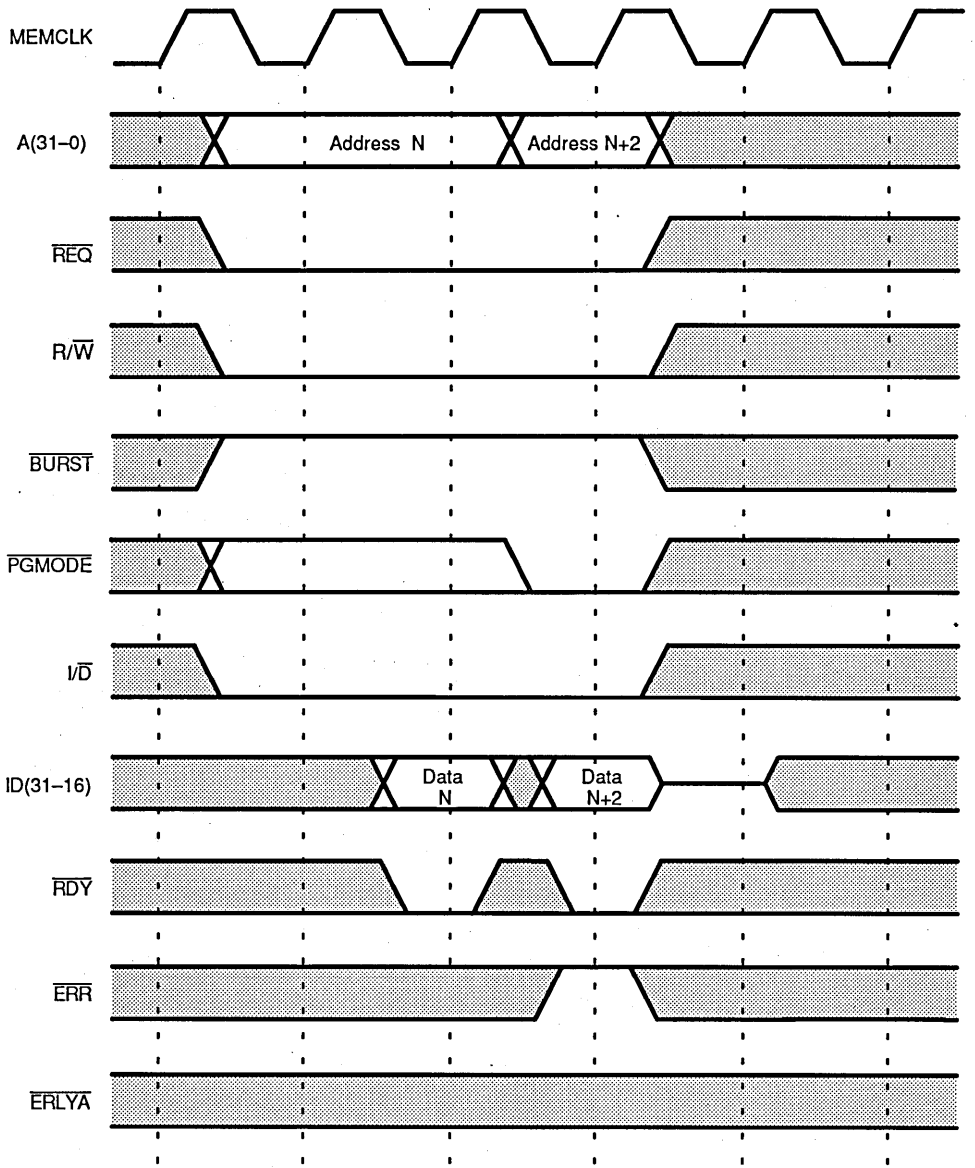


Figure A-32 Burst-Mode 16-Bit Narrow Write Access (Am29035 Microprocessor only)

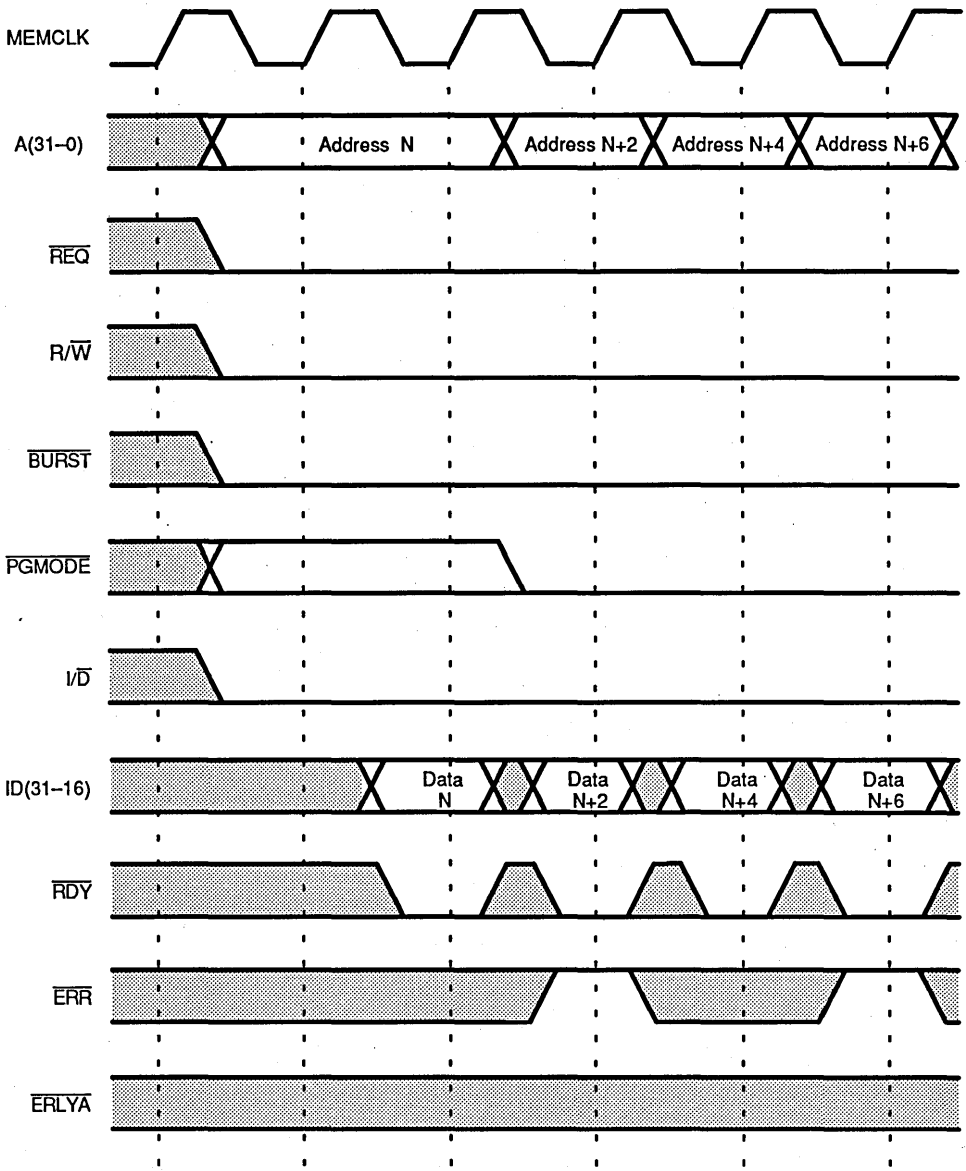


Figure A-33 Load and Set Instruction (Page Mode)

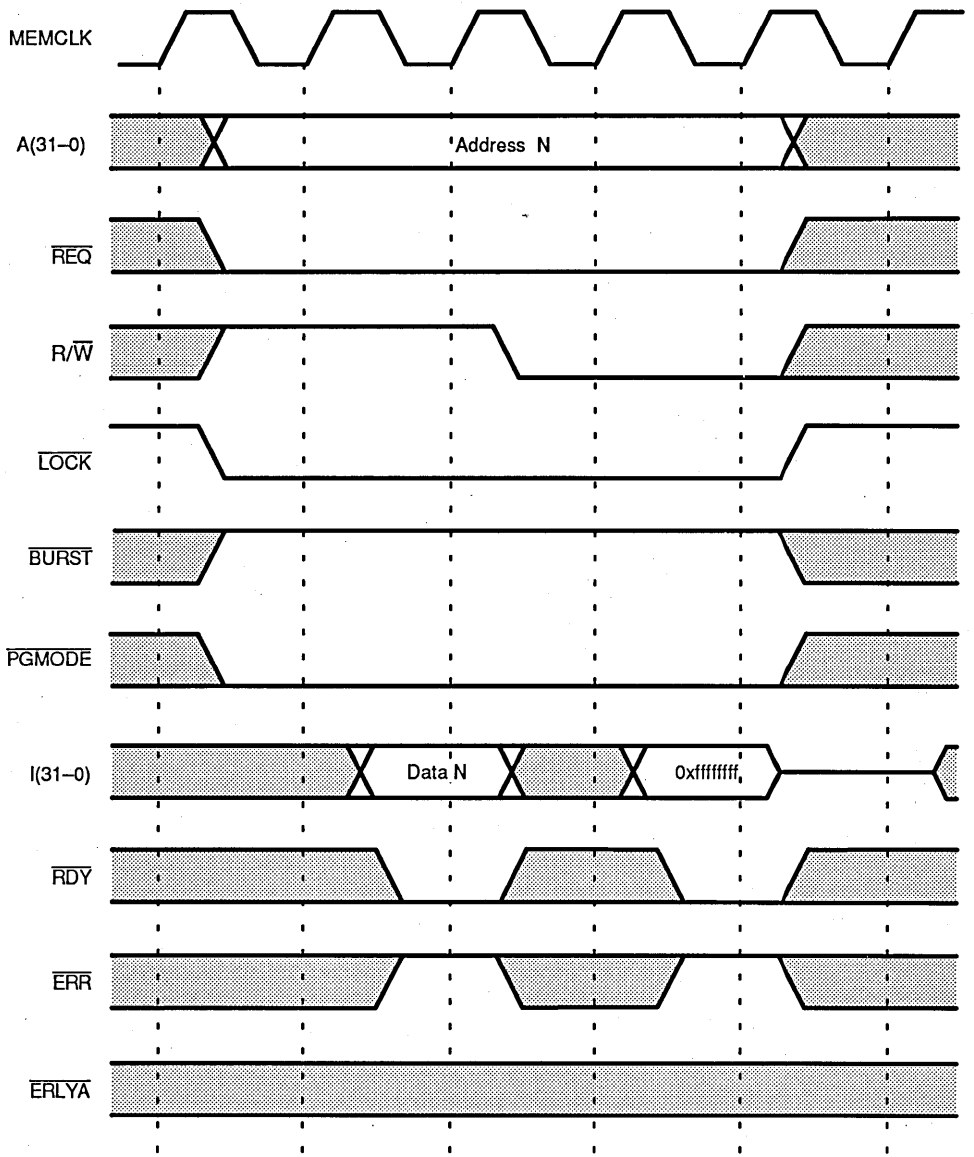


Figure A-34 TLB Miss or Protection Violation on Read or Write

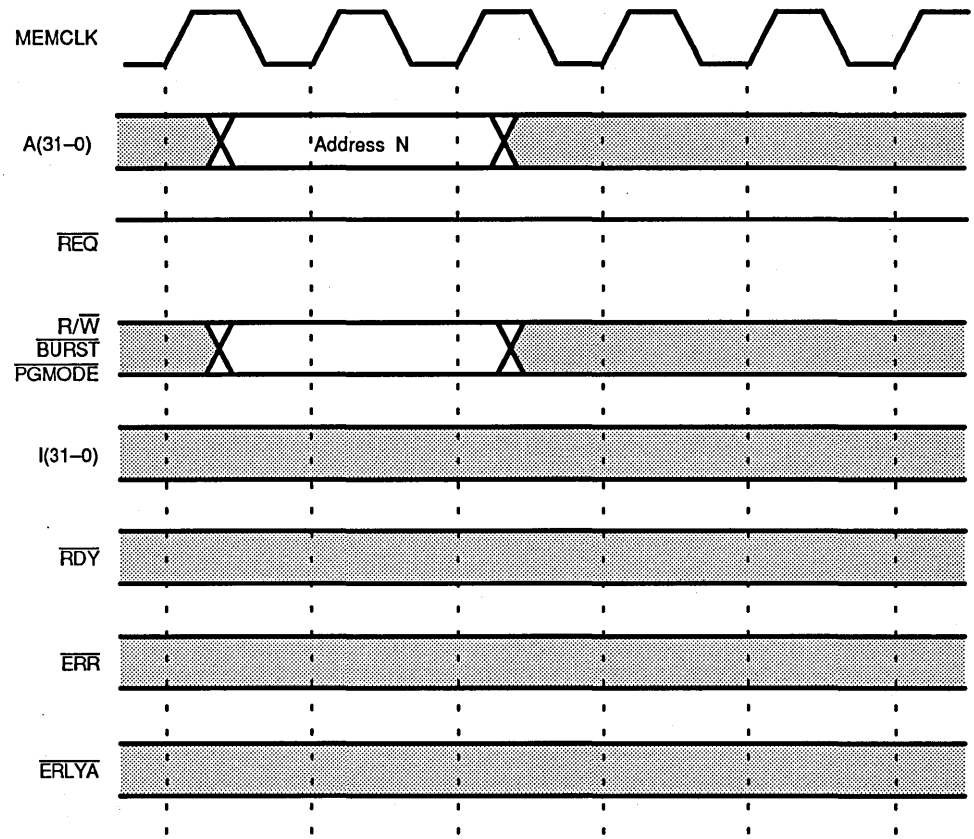


Figure A-35 Bus Arbitration—Normal Transfer to Processor

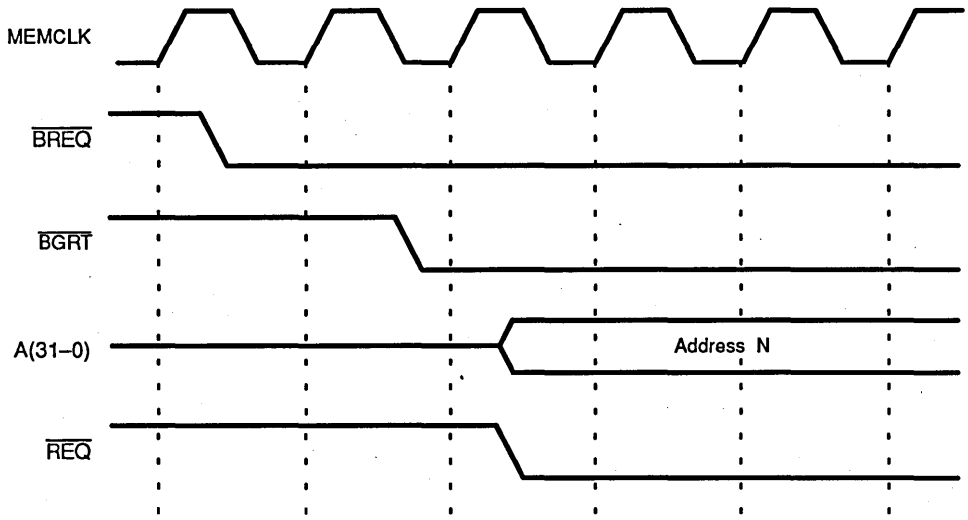


Figure A-36 Bus Arbitration—Fast Transfer to Processor

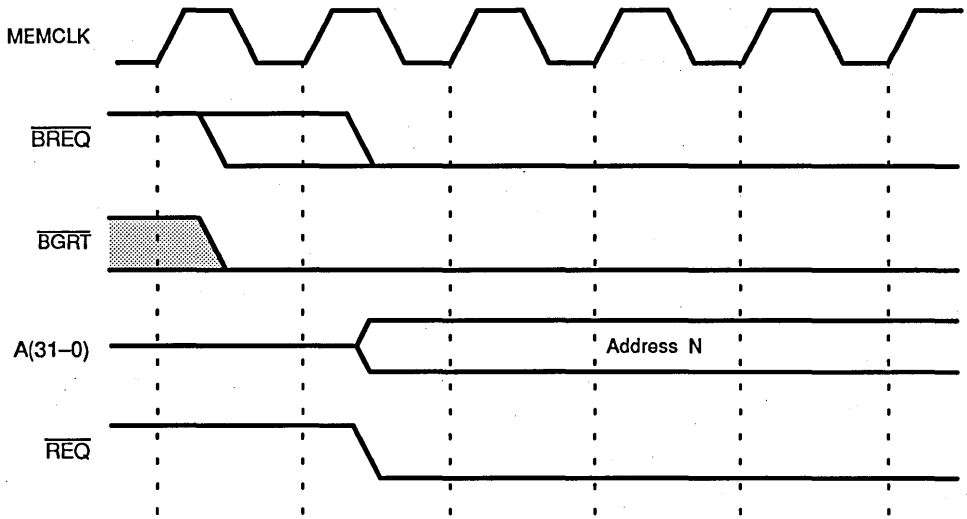


Figure A-37 Bus Arbitration—False Processor Request

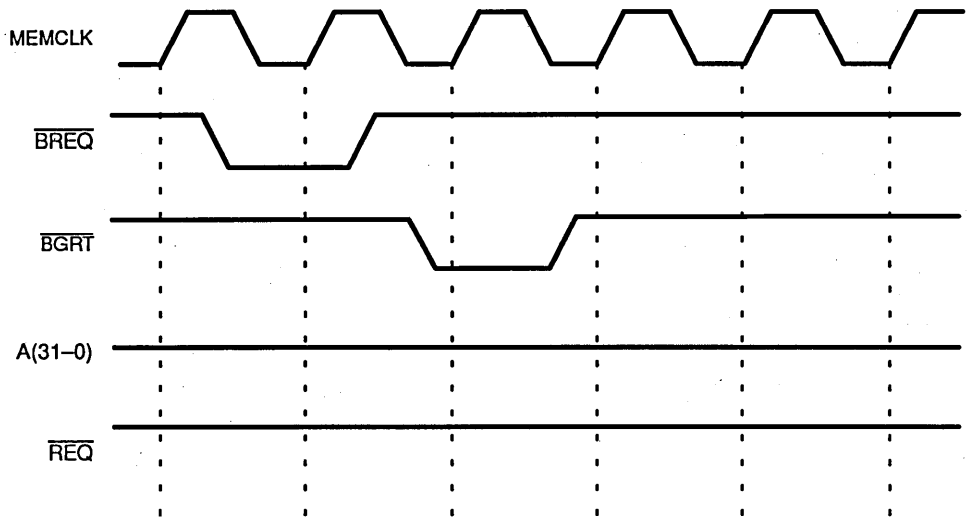


Figure A-38 Bus Arbitration—Granting of Unrequested Bus

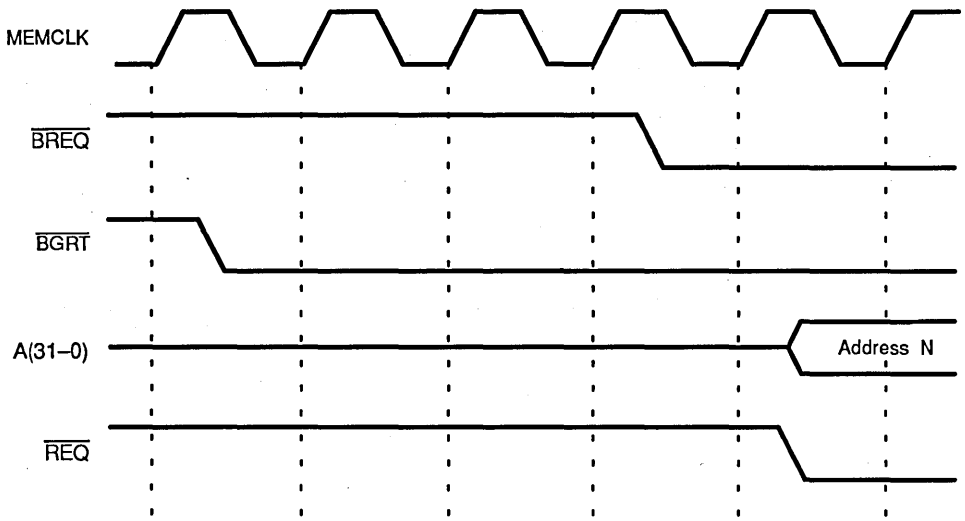


Figure A-39 Bus Arbitration—Normal Transfer from Processor

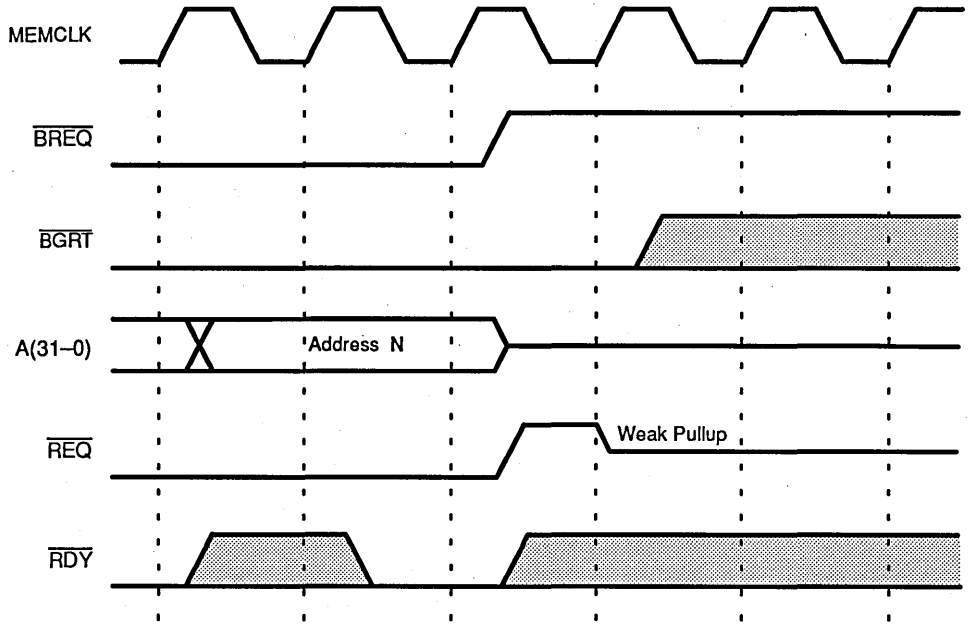


Figure A-40 Bus Arbitration—Preempting Bus from Processor

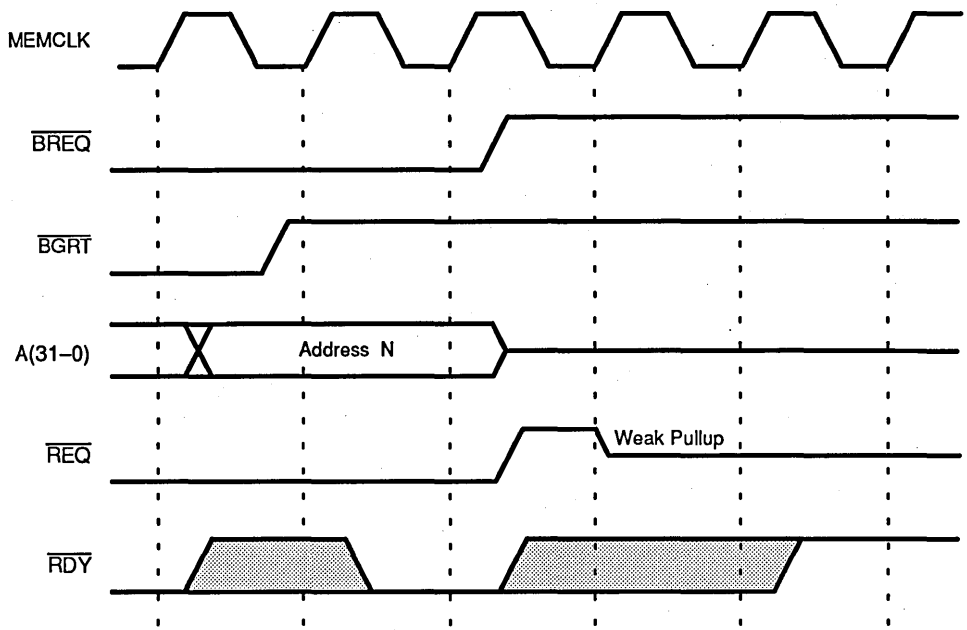
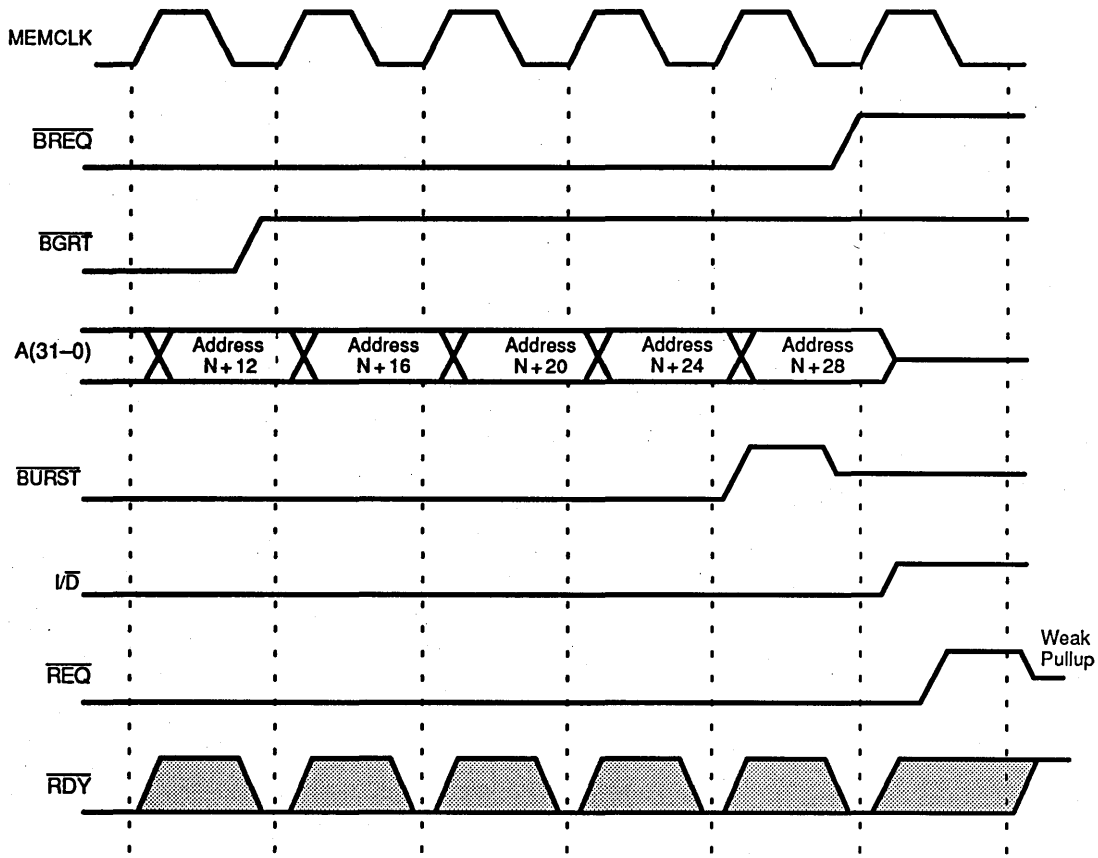


Figure A-41 Bus Arbitration—Preempting Bus from Processor (worst case)



Note: This example shows the arbiter removing BGRT at the end of a block boundary. The processor will fetch one more block before releasing the bus.

REGISTER SUMMARY

Figure B-1 General-Purpose Register Organization

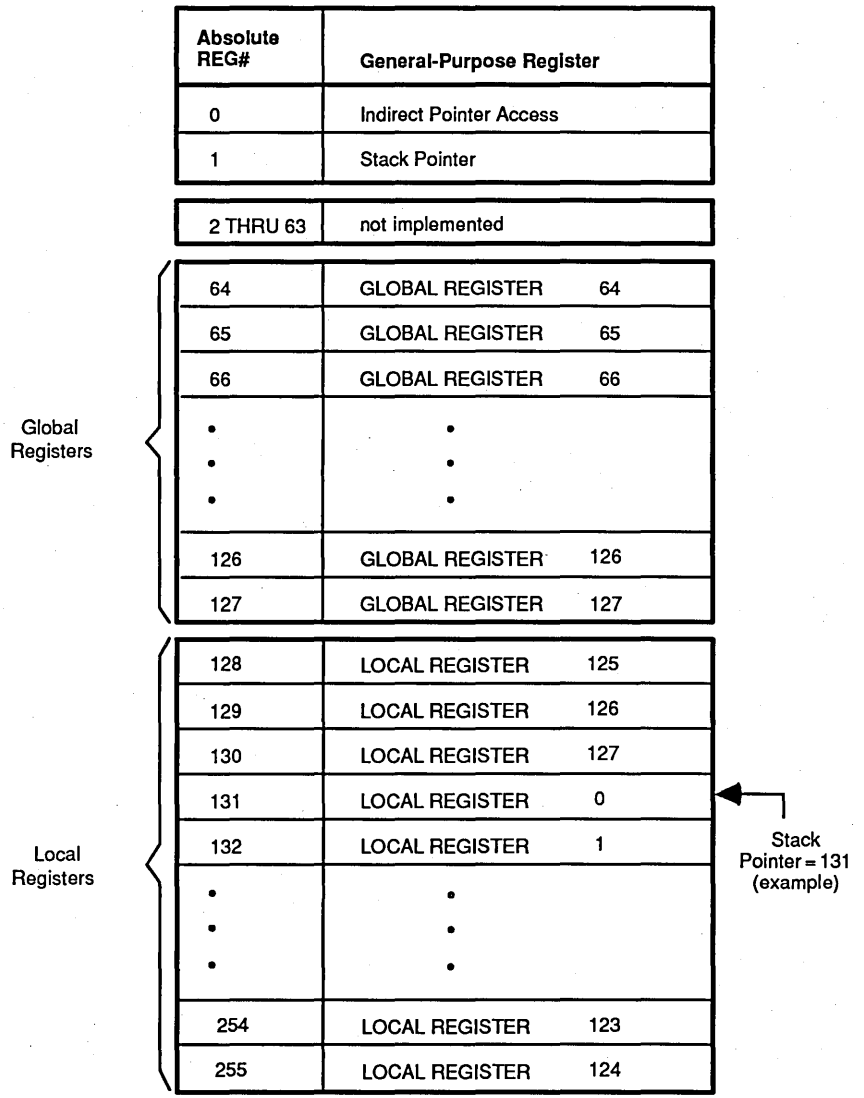


Figure B-2 Register Bank Organization

Register Bank Protect Register Bit	Absolute-Register Numbers	General-Purpose Registers
0	2 through 15	Bank 0 (unimplemented)
1	16 through 31	Bank 1 (unimplemented)
2	32 through 47	Bank 2 (unimplemented)
3	48 through 63	Bank 3 (unimplemented)
4	64 through 79	Bank 4
5	80 through 95	Bank 5
6	96 through 111	Bank 6
7	112 through 127	Bank 7
8	128 through 143	Bank 8
9	144 through 159	Bank 9
10	160 through 175	Bank 10
11	176 through 191	Bank 11
12	192 through 207	Bank 12
13	208 through 223	Bank 13
14	224 through 239	Bank 14
15	240 through 255	Bank 15

Figure B-3 Special Purpose Registers

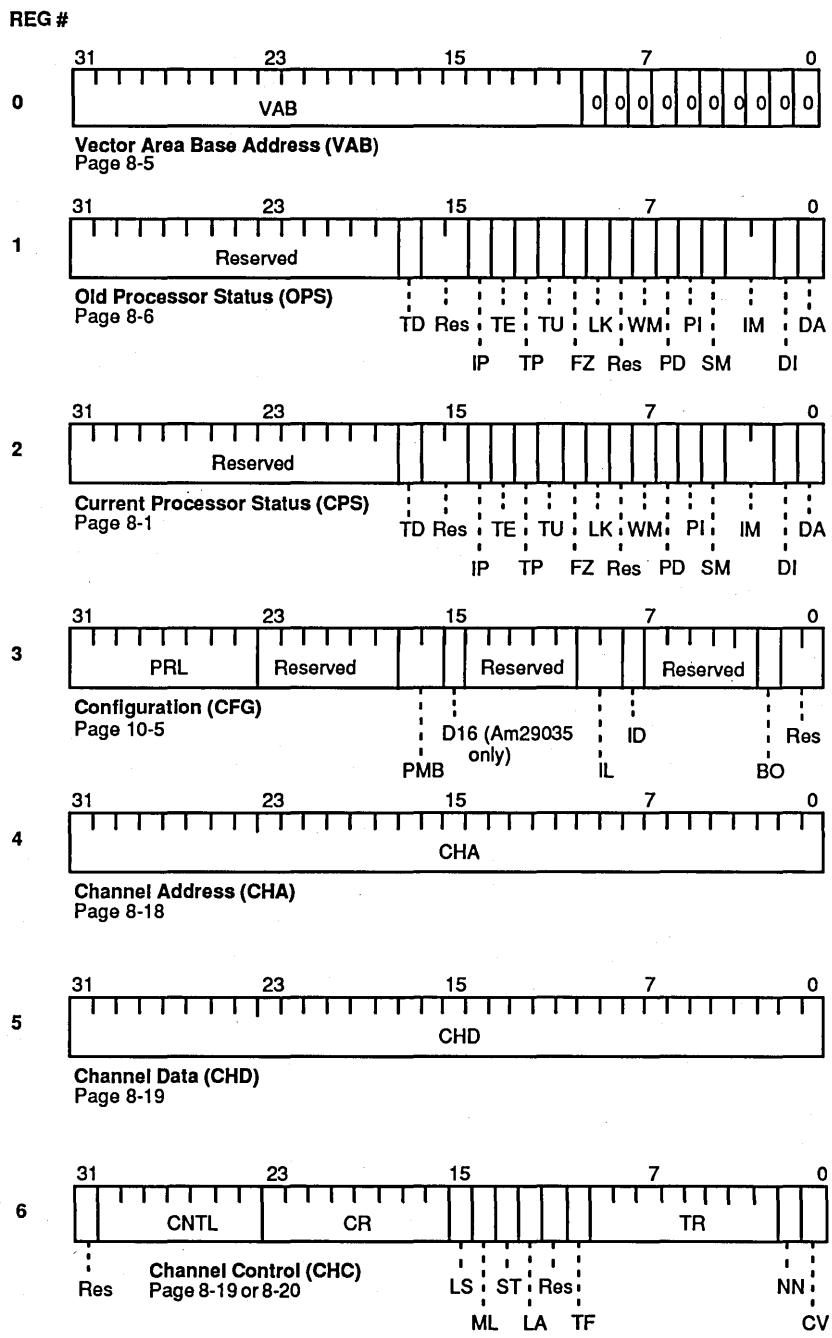
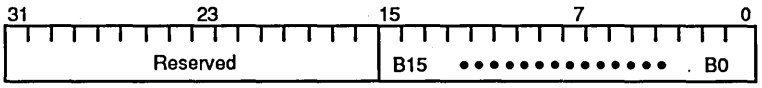


Figure B-3 Special Purpose Registers (continued)

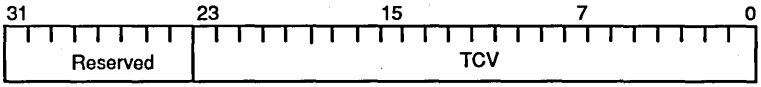
REG #

7



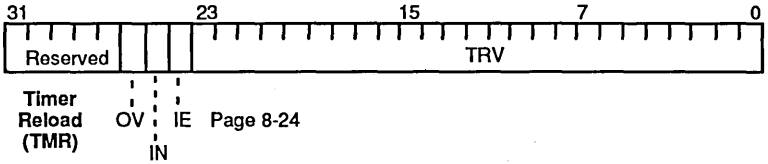
Register Bank Protect (RBP)
Page 6-3

8

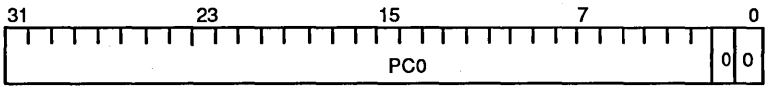


Timer Counter (TMC) Page 8-23

9

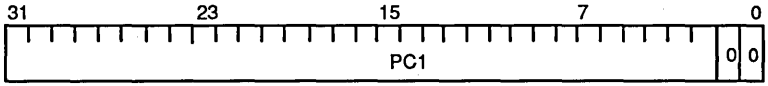


10



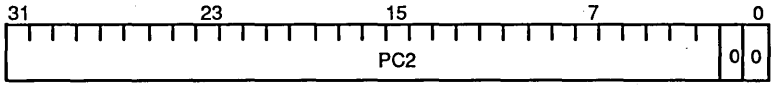
Program Counter 0 (PC0)
Page 8-9

11



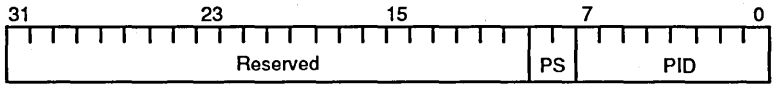
Program Counter 1 (PC1)
Page 8-9

12



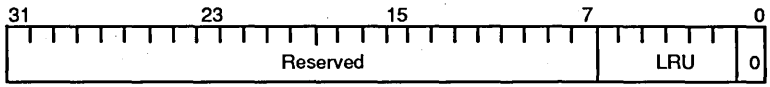
Program Counter 2 (PC2)
Page 8-10

13



MMU Configuration (MMU)
Page 7-5

14



LRU Recommendation (LRU)
Page 7-12

Figure B-3 Special Purpose Registers (continued)

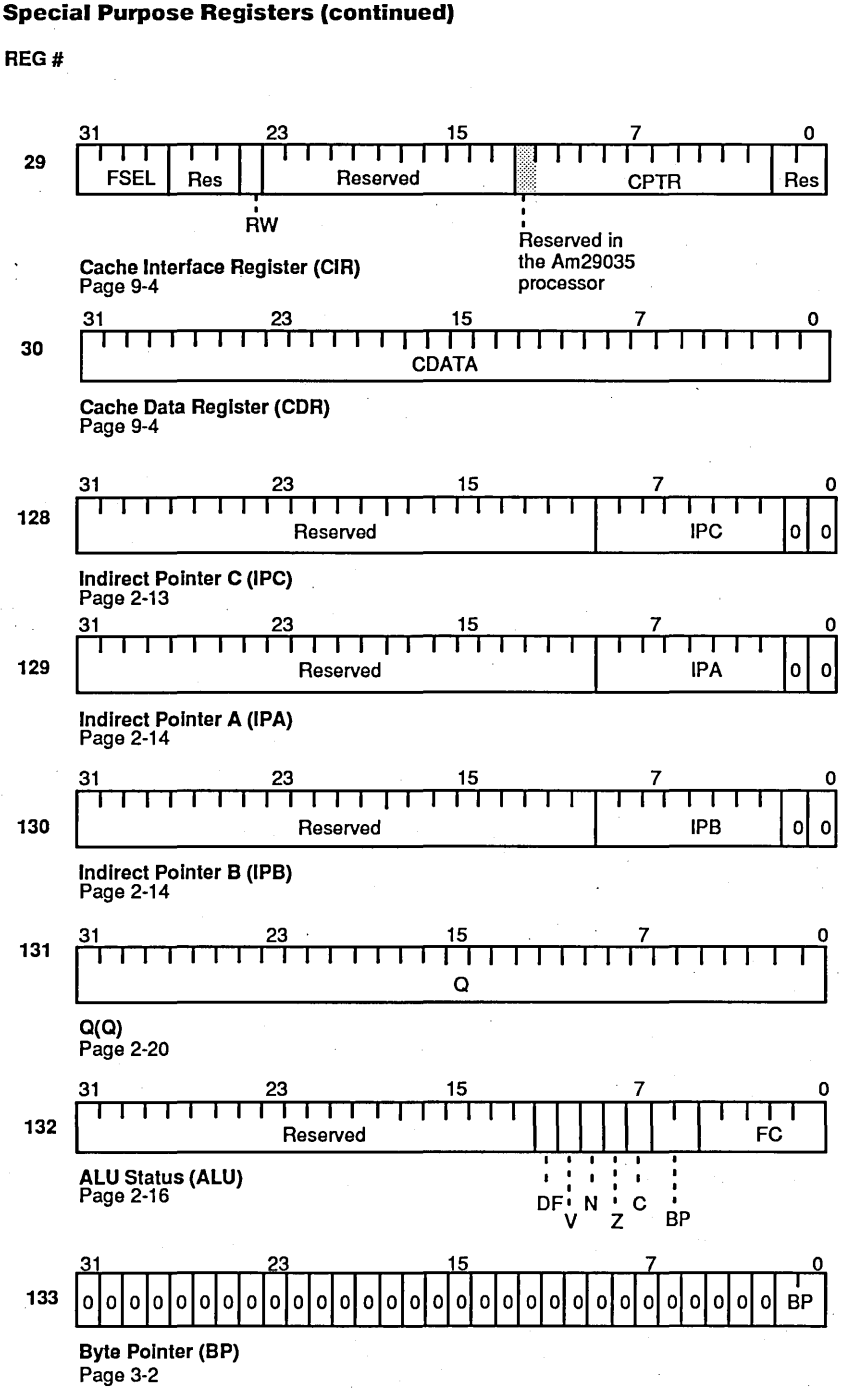


Figure B-3 Special Purpose Registers (continued)

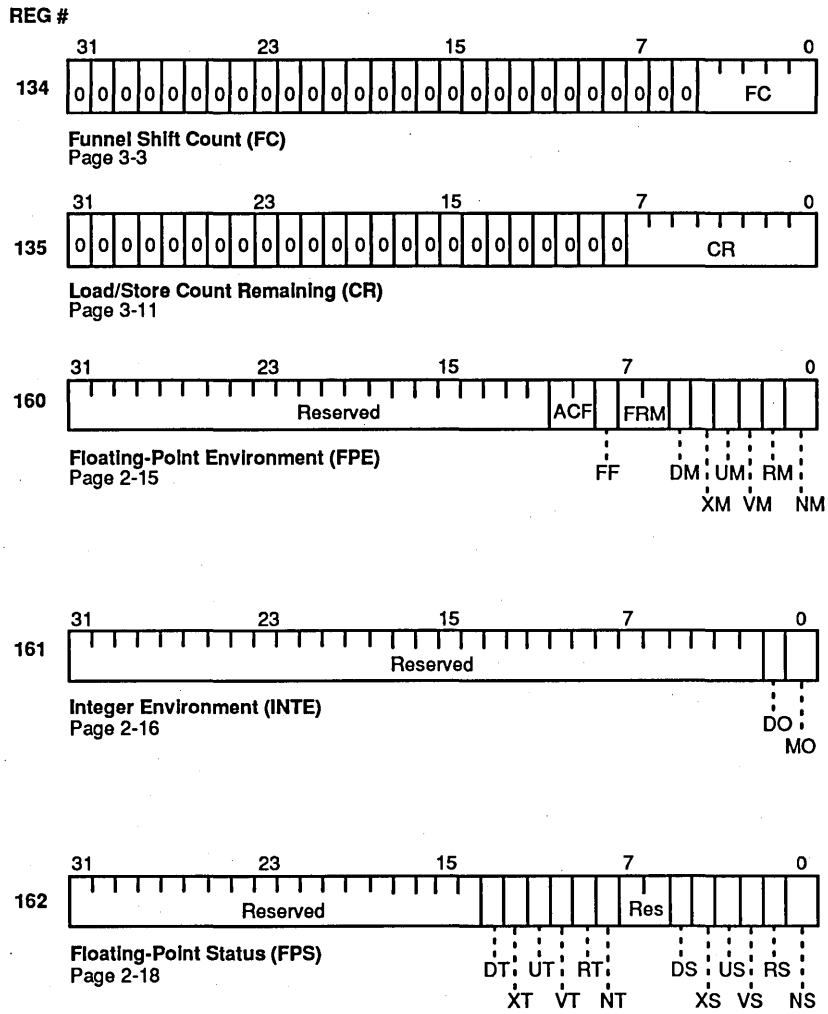


Figure B-4 Special Purpose Registers

REG #	TLB Column 0
0	TLB Entry Set 0 Word 0
1	TLB Entry Set 0 Word 1
2	TLB Entry Set 1 Word 0
3	TLB Entry Set 1 Word 1
≈	≈
60	TLB Entry Set 30 Word 0
61	TLB Entry Set 30 Word 1
62	TLB Entry Set 31 Word 0
63	TLB Entry Set 31 Word 1
	TLB Column 1
64	TLB Entry Set 0 Word 0
65	TLB Entry Set 0 Word 1
66	TLB Entry Set 1 Word 0
67	TLB Entry Set 1 Word 1
≈	≈
124	TLB Entry Set 30 Word 0
125	TLB Entry Set 30 Word 1
126	TLB Entry Set 31 Word 0
127	TLB Entry Set 31 Word 1

Figure B-5 Translation Look-Aside Buffer Entries

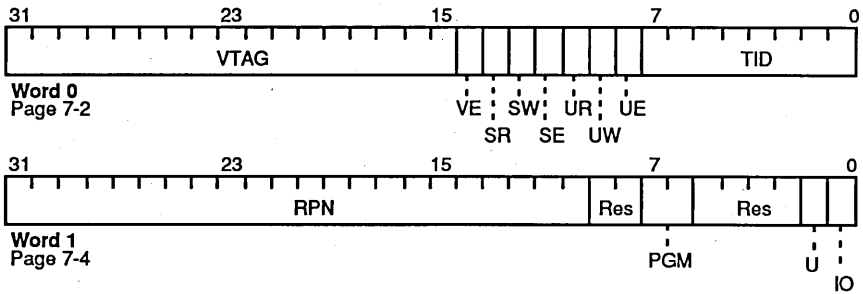


Table B-1 Register Field Summary

Label	Field Name	Register	Bit
ACF	Accumulator Format	Floating-Point Environment	10–9
B0	Bank 0 Protection Bit	Register Bank Protect	0
B1	Bank 1 Protection Bit	Register Bank Protect	1
B2	Bank 2 Protection Bit	Register Bank Protect	2
B3	Bank 3 Protection Bit	Register Bank Protect	3
B4	Bank 4 Protection Bit	Register Bank Protect	4
B5	Bank 5 Protection Bit	Register Bank Protect	5
B6	Bank 6 Protection Bit	Register Bank Protect	6
B7	Bank 7 Protection Bit	Register Bank Protect	7
B8	Bank 8 Protection Bit	Register Bank Protect	8
B9	Bank 9 Protection Bit	Register Bank Protect	9
B10	Bank 10 Protection Bit	Register Bank Protect	10
B11	Bank 11 Protection Bit	Register Bank Protect	11
B12	Bank 12 Protection Bit	Register Bank Protect	12
B13	Bank 13 Protection Bit	Register Bank Protect	13
B14	Bank 14 Protection Bit	Register Bank Protect	14
B15	Bank 15 Protection Bit	Register Bank Protect	15
BO	Byte Order	Configuration	2
BP	Byte Pointer	ALU Status Byte Pointer	6–5 1–0
C	Carry	ALU Status	7
CDATA	Cache Data	Cache Data Register	31–0
CHA	Channel Address	Channel Address	31–0
CHD	Channel Data	Channel Data	31–0
CNTL	Control	Channel Control	30–24
CPTR	Cache Pointer	Cache Interface Register	12–2
CR	Load/Store Count Remaining	Channel Control Load/Store Count Remaining	23–16 7–0
CV	Contents Valid	Channel Control	0
D16	Data Width 16 Bits	Configuration	15
DA	Disable All Interrupts and Traps	Current Processor Status Old Processor Status	0 0
DF	Divide Flag	ALU Status	11

Table B-1 Register Field Summary (continued)

Label	Field Name	Register	Bit
DI	Disable Interrupts	Current Processor Status Old Processor Status	1 1
DM	Floating-Point Divide By Zero Mask	Floating-Point Environment	5
DO	Integer Division Overflow Mask	Integer Environment	1
DS	Floating-Point Divide By Zero Sticky	Floating-Point Status	5
DT	Floating-Point Divide By Zero Trap	ALU Status	13
FF	Fast Floating-Point Select	Floating-Point Environment	8
FC	Funnel Shift Count	ALU Status Funnel Shift Count	4–0 4–0
FRM	Floating-Point Round Mode	Floating-Point Environment	7–6
FSEL	Cache Field Select	Cache Interface Register	31–28
FZ	Freeze	Current Processor Status Old Processor Status	10 10
ID	Instruction Cache Disable	Configuration	8
IE	Interrupt Enable	Timer Reload	24
IL	Instruction Cache Lock	Configuration	10–9
IM	Interrupt Mask	Old Processor Status Current Processor Status	3–2 3–2
IN	Interrupt	Timer Reload	25
IO	Input/Output	TLB Entry Word 1	0
IP	Interrupt Pending	Current Processor Status Old Processor Status	14 14
IPA	Indirect Pointer A	Indirect Pointer A	9–2
IPB	Indirect Pointer B	Indirect Pointer B	9–2
IPC	Indirect Pointer C	Indirect Pointer C	9–2
LA	Lock Active	Channel Control	12
LK	Lock	Current Processor Status Old Processor Status	9 9
LRU	Least-Recently Used Entry	LRU Recommendation	6–1
LS	Load/Store	Channel Control	15
ML	Multiple Operation	Channel Control	14
MO	Integer Multiplication Overflow Mask	Integer Environment	0
N	Negative	ALU Status	9
NM	Floating-Point Invalid Operation Mask	Floating-Point Environment	0
NN	Not Needed	Channel Control	1
NS	Floating-Point Invalid Operation Sticky	Floating-Point Status	0
NT	Floating-Point Invalid Operation Trap	Floating-Point Status	8
OV	Overflow	Timer Reload	26

Table B-1 Register Field Summary (continued)

Label	Field Name	Register	Bit
PC0	Program Counter 0	Program Counter 0	31–2
PC1	Program Counter 1	Program Counter 1	31–2
PC2	Program Counter 2	Program Counter 2	31–2
PD	Physical Addressing Data	Current Processor Status Old Processor Status	6 6
PGM	User Programmable	TLB Entry Word 1	7–6
PI	Physical Addressing Instructions	Current Processor Status Old Processor Status	5 5
PID	Process Identifier	MMU Configuration	7–0
PMB	Page-Mode Block	Configuration	17–16
PRL	Processor Release Level	Configuration	31–24
PS	Page Size	MMU Configuration	9–8
Q	Quotient/Multiplier	Q Register	31–0
RM	Floating-Point Reserved Operand Mask	Floating-Point Environment	1
RPN	Real Page Number	TLB Entry Word 1	31–10
RS	Floating-Point Reserved Operand Sticky	Floating-Point Status	1
RT	Floating-Point Reserved Operand Trap	Floating-Point Status	9
RW	Read/Write	Cache Interface Register	24
SE	Supervisor Execute	TLB Entry Word 0	11
SM	Supervisor Mode	Current Processor Status Old Processor Status	4 4
SR	Supervisor Read	TLB Entry Word 0	13
ST	Set	Channel Control	13
SW	Supervisor Write	TLB Entry Word 0	12
TCV	Timer Count Value	Timer Counter	23–0
TD	Timer Disable	Current Processor Status Old Processor Status	17 17
TE	Trace Enable	Current Processor Status Old Processor Status	13 13
TF	Transaction Faulted	Channel Control	10
TID	Task Identifier	TLB Entry Word 0	7–0
TP	Trace Pending	Current Processor Status Old Processor Status	12 12
TR	Target Register	Channel Control	9–2
TRV	Timer Reload Value	Timer Reload	23–0

Table B-1 Register Field Summary (continued)

Label	Field Name	Register	Bit
TU	Trap Unaligned Access	Current Processor Status Old Processor Status	11 11
U	Usage	TLB Entry Word 1	1
UE	User Execute	TLB Entry Word 0	8
UM	Floating-Point Underflow Mask	Floating-Point Environment	3
UR	User Read	TLB Entry Word 0	10
US	Floating-Point Underflow Sticky	Floating-Point Status	3
UT	Floating-Point Underflow Trap	Floating-Point Status	11
UW	User Write	TLB Entry Word 0	9
V	Overflow	ALU Status	10
VAB	Vector Area Base	Vector Area Base Address	31–10
VE	Valid Entry	TLB Entry Word 0	14
VM	Floating-Point Overflow Mask	Floating-Point Environment	2
VS	Floating-Point Overflow Sticky	Floating-Point Status	2
VT	Floating-Point Overflow Trap	Floating-Point Status	10
VTAG	Virtual Tag	TLB Entry Word 0	31–15
WM	Wait Mode	Current Processor Status Old Processor Status	7 7
XM	Floating-Point Inexact Result Mask	Floating-Point Environment	4
XS	Floating-Point Inexact Result Sticky	Floating-Point Status	4
XT	Floating-Point Inexact Result Trap	Floating-Point Status	12
Z	Zero	ALU Status	8

PRELIMINARY

Am29030™ and Am29035™

RISC Microprocessors with 8-Kbyte/4-Kbyte Instruction Cache

Advanced Micro Devices

Am29030 MICROPROCESSOR DISTINCTIVE CHARACTERISTICS

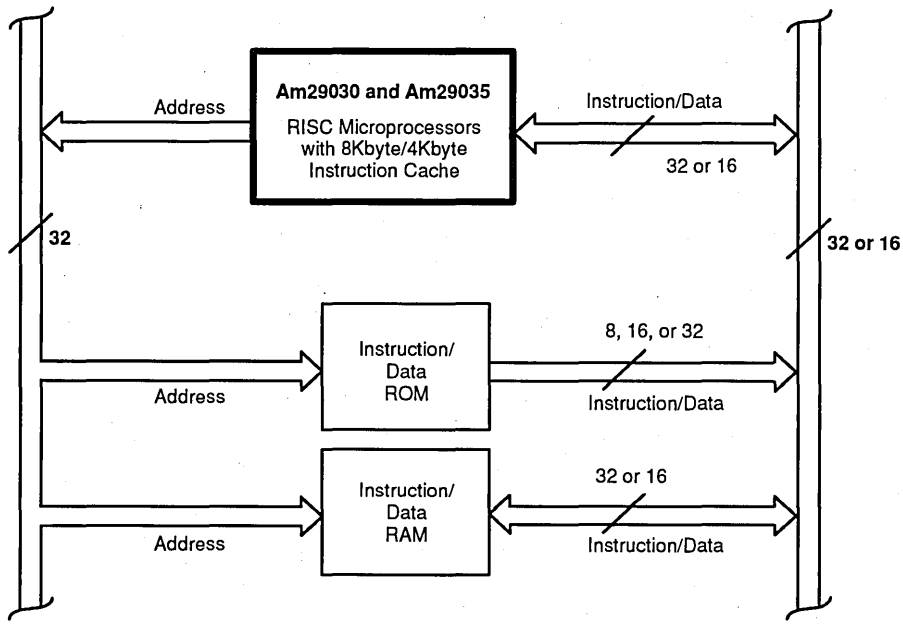
- Full 32-bit architecture
- 26 million instructions per second (MIPS) sustained at 33 MHz
- 8-Kbyte, two-way set-associative instruction cache
- 33- and 25-MHz operating frequencies
- Scalable Clocking™ technology
- Programmable 16- or 32-bit data bus width
- CMOS technology/TTL-compatible
- 4-Gbyte virtual address space with demand paging
- Streamlined system interface for simplified, high-frequency operation
- Burst-mode and page-mode access support
- 8-, 16-, or 32-bit ROM interface
- 64-entry Memory Management Unit on-chip
- Fully pipelined
- On-chip timer facility
- 192 general-purpose registers
- Three-address instruction architecture
- Master/slave chip/output checking
- Software compatible with Am29005™ and Am29000™ microprocessors
- Advanced debugging support
- IEEE Std. 1149.1–1990 (JTAG) compliant Standard Test Access Port and Boundary Scan Architecture implementation

Am29035 MICROPROCESSOR DISTINCTIVE CHARACTERISTICS

The Am29035 microprocessor is similar to the Am29030 microprocessor except for the following differences:

- 4-Kbyte, direct-mapped instruction cache
- 16-MHz operating frequency
- 12 million instructions per second (MIPS) sustained at 16 MHz

SIMPLIFIED BLOCK DIAGRAM



Publication# 18150 Rev. A Amendment /0
Issue Date: August 1993

GENERAL DESCRIPTION

The Am29030 and Am29035 RISC microprocessors are high-performance, general-purpose, 32-bit microprocessors implemented in CMOS technology. Through high circuit densities and a high degree of on-chip integration, the Am29030 and Am29035 microprocessors are capable of operating at high internal frequencies while providing the designer with a simple streamlined external interface.

The Am29030 and Am29035 microprocessors were designed to meet the common requirements of embedded applications such as laser beam printers, graphics processors, X terminals and servers, Application Program Interface (API) accelerators, and scanners. The Am29030 and Am29035 microprocessors are well suited for these applications since they provide high performance at low cost, and offer the designer complete design flexibility. Coupled with hardware and software development tools from AMD® and AMD's Fusion29K℠ partners, no design is ever far from the marketplace.

The Am29030 microprocessor is available in a 145-lead pin-grid-array (PGA) package and a 144-pin cerquad package. The PGA has 111 signal pins, 26 power and ground pins, 7 reserved pins, and 1 alignment/ground pin. The cerquad has 111 signal pins, 30 power and ground pins, and 3 reserved pins.

The Am29035 microprocessor is available in a 144-pin cerquad package.

29K™ Family Development Support Products

Contact your local AMD representative for information on the complete set of development support tools.

Software development products on several hosts:

- Optimizing compilers for common high-level languages
- Assembler and utility packages
- Source- and assembly-level software debuggers
- Target-resident development monitors
- Simulators
- EZ-030, an Am29030 microprocessor-based evaluation board kit

RELATED AMD PRODUCTS

29K Family Devices

Device	Description
Am29000™	Streamlined Instruction Microprocessor
Am29005™	Low-cost Streamlined Instruction Microprocessor
Am29050™	Streamlined Instruction Microprocessor with On-chip Floating Point
Am29200™	RISC Microcontroller
Am29205™	Low-cost RISC Microcontroller

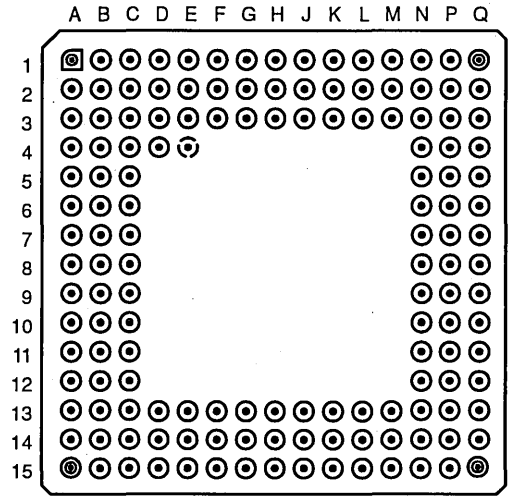
Third-Party Development Support Products

The Fusion29K Program of Partnerships for Application Solutions provides the user with a vast array of products designed to meet critical time-to-market needs.

Products and solutions available through AMD's Fusion29K partners include:

- Silicon products
- Software generation and debug tools
- Hardware development tools
- Board level products
- Laser printer solutions
- Multiuser, kernel, and real-time operating systems
- Graphics solutions
- Networking and communications solutions
- Manufacturing support
- Custom support

CONNECTION DIAGRAM
145-Lead PGA
Bottom View



Pin Number E-4 is defined for emulator access and is not a physical pin on the package (see the Am29030 and Am29035 Microprocessors User's Manual, order #15723, Signal Description section).

Note: *Pinout observed from pin side of package (pins facing viewer).*

PGA PIN DESIGNATION
(Sorted by Pin No.)

Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
A–1	ID4	C–8	GND	H–14	MEMCLK	N–12	Vcc
A–2	ID0	C–9	Vcc	H–15	$\overline{DIV2}$	N–13	A1
A–3	$\overline{TRST}$	C–10	GND	J–1	ID21	N–14	$\overline{ERLYA}$
A–4	TDI	C–11	Vcc	J–2	ID22	N–15	$\overline{ERR}$
A–5	TCK	C–12	STAT1	J–3	GND	P–1	ID31
A–6	$\overline{TEST}$	C–13	$\overline{WBC}$	J–13	INCLK	P–2	A30
A–7	I/$\overline{D}$	C–14	$\overline{HIT}$	J–14	PWRCLK	P–3	A27
A–8	IO/$\overline{MEM}$	C–15	$\overline{INTR0}$	J–15	$\overline{BREQ}$	P–4	A24
A–9	$\overline{BWE0}$	D–1	ID12	K–1	ID23	P–5	A22
A–10	$\overline{BWE2}$	D–2	ID11	K–2	ID24	P–6	A20
A–11	SUP/$\overline{US}$	D–3	ID9	K–3	Vcc	P–7	A18
A–12	OPT0	D–4	GND	K–13	GND	P–8	A15
A–13	OPT2	D–13	$\overline{DI}$	K–14	$\overline{REQ}$	P–9	A13
A–14	MPGM1	D–14	$\overline{INTR1}$	K–15	$\overline{BURST}$	P–10	A10
A–15	STAT2	D–15	$\overline{INTR2}$	L–1	ID25	P–11	A8
B–1	ID7	E–1	ID14	L–2	ID26	P–12	A6
B–2	ID6	E–2	ID13	L–3	Vcc	P–13	A4
B–3	ID3	E–3	GND	L–13	Vcc	P–14	A2
B–4	ID1	E–13	GND	L–14	$\overline{BGRT}$	P–15	A0
B–5	TDO	E–14	$\overline{INTR3}$	L–15	$\overline{PGMODE}$	Q–1	A31
B–6	TMS	E–15	$\overline{TRAP1}$	M–1	ID27	Q–2	A29
B–7	R/$\overline{W}$	F–1	ID16	M–2	ID28	Q–3	A25
B–8	$\overline{WARN}$	F–2	ID15	M–3	NC	Q–4	A23
B–9	$\overline{BWE1}$	F–3	NC	M–13	GND	Q–5	A21
B–10	$\overline{BWE3}$	F–13	Vcc	M–14	$\overline{RDN}$	Q–6	A19
B–11	$\overline{LOCK}$	F–14	$\overline{RESET}$	M–15	$\overline{RDY}$	Q–7	A17
B–12	OPT1	F–15	$\overline{TRAP0}$	N–1	ID29	Q–8	A16
B–13	MPGM0	G–1	ID18	N–2	ID30	Q–9	A14
B–14	STAT0	G–2	ID17	N–3	NC	Q–10	A12
B–15	MSERR	G–3	Vcc	N–4	A28	Q–11	A11
C–1	ID10	G–13	Vcc	N–5	A26	Q–12	A9
C–2	ID8	G–14	CNTL0	N–6	GND	Q–13	A7
C–3	NC*	G–15	CNTL1	N–7	Vcc	Q–14	A5
C–4	ID5	H–1	ID20	N–8	GND	Q–15	A3
C–5	ID2	H–2	ID19	N–9	GND		
C–6	Vcc	H–3	GND	N–10	GND		
C–7	GND	H–13	GND	N–11	Vcc		

Notes:

**1. NC = No connection internally.*

2. Pin Number D–4 is the alignment/ground pin and must be electrically connected to ground.

3. Pin Number E–4 is defined for emulator access and is not a physical pin on the package (see Am29030 and Am29035 Microprocessors User's Manual, Signal Description section).

PGA PIN DESIGNATION (Sorted by Pin Name)

Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
P–15	A0	B–10	$\overline{BWE3}$	F–1	ID16	C–12	STAT1
N–13	A1	G–14	CNTL0	G–2	ID17	A–15	STAT2
P–14	A2	G–15	CNTL1	G–1	ID18	A–11	SUP/$\overline{US}$
Q–15	A3	H–15	$\overline{DIV2}$	H–2	ID19	A–5	TCK
P–13	A4	N–14	$\overline{ERLYA}$	H–1	ID20	A–4	TDI
Q–14	A5	N–15	$\overline{ERR}$	J–1	ID21	B–5	TDO
P–12	A6	C–7	GND	J–2	ID22	A–6	$\overline{TEST}$
Q–13	A7	C–8	GND	K–1	ID23	B–6	TMS
P–11	A8	C–10	GND	K–2	ID24	F–15	$\overline{TRAP0}$
Q–12	A9	D–4	GND	L–1	ID25	E–15	$\overline{TRAP1}$
P–10	A10	E–3	GND	L–2	ID26	A–3	$\overline{TRST}$
Q–11	A11	E–13	GND	M–1	ID27	C–6	Vcc
Q–10	A12	H–3	GND	M–2	ID28	C–9	Vcc
P–9	A13	H–13	GND	N–1	ID29	C–11	Vcc
Q–9	A14	J–3	GND	N–2	ID30	F–13	Vcc
P–8	A15	K–13	GND	P–1	ID31	G–3	Vcc
Q–8	A16	M–13	GND	J–13	INCLK	G–13	Vcc
Q–7	A17	N–6	GND	C–15	$\overline{INTR0}$	K–3	Vcc
P–7	A18	N–8	GND	D–14	$\overline{INTR1}$	L–3	Vcc
Q–6	A19	N–9	GND	D–15	$\overline{INTR2}$	L–13	Vcc
P–6	A20	N–10	GND	E–14	$\overline{INTR3}$	N–7	Vcc
Q–5	A21	A–7	I/$\overline{D}$	A–8	IO/$\overline{MEM}$	N–11	Vcc
P–5	A22	A–2	ID0	B–11	$\overline{LOCK}$	N–12	Vcc
Q–4	A23	B–4	ID1	H–14	MEMCLK	B–8	$\overline{WARN}$
P–4	A24	C–5	ID2	B–13	MPGM0		
Q–3	A25	B–3	ID3	A–14	MPGM1		
N–5	A26	A–1	ID4	B–15	MSERR		
P–3	A27	C–4	ID5	A–12	OPT0		
N–4	A28	B–2	ID6	B–12	OPT1		
Q–2	A29	B–1	ID7	A–13	OPT2		
P–2	A30	C–2	ID8	L–15	$\overline{PGMODE}$		
Q–1	A31	D–3	ID9	J–14	PWRCLK		
L–14	$\overline{BGRT}$	C–1	ID10	M–14	$\overline{RDN}$		
J–15	$\overline{BREQ}$	D–2	ID11	M–15	$\overline{RDY}$		
K–15	$\overline{BURST}$	D–1	ID12	F–14	$\overline{RESET}$	RESERVED*	
A–9	$\overline{BWE0}$	E–2	ID13	K–14	$\overline{REQ}$	D–13	$\overline{DI}$
B–9	$\overline{BWE1}$	E–1	ID14	B–7	R/$\overline{W}$	C–14	$\overline{HIT}$
A10	$\overline{BWE2}$	F–2	ID15	B–14	STAT0	C–13	$\overline{WBC}$

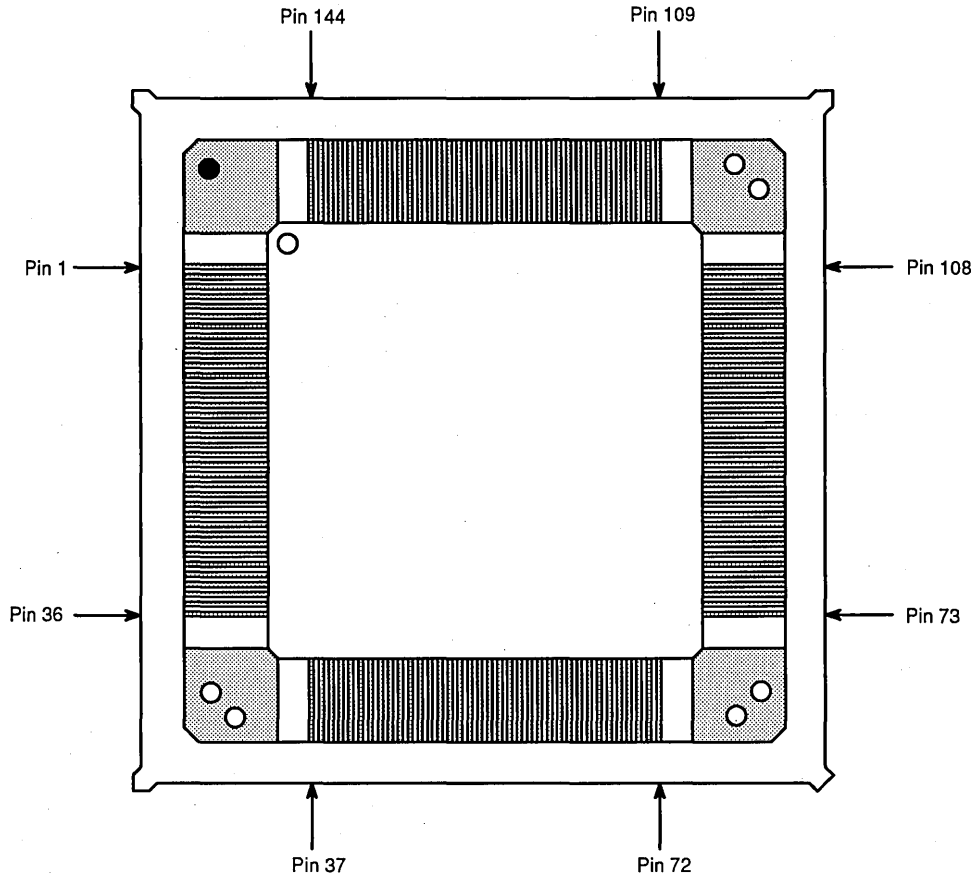
Notes:

**1. These following signals are reserved for future processor implementations. To maintain compatibility with future processor implementations, these pins should be connected to V_{CC} by individual pull-up resistors.*

2. Pin Number D–4 is the alignment/ground pin and must be electrically connected to ground.

3. Pin Number E–4 is defined for emulator access and is not a physical pin on the package (see Am29030 and Am29035 Microprocessors User's Manual, Signal Description section).

CONNECTION DIAGRAM
144-Lead Cerquad
Top View



CERQUAD PIN DESIGNATION
(Sorted by Pin No.)

Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
1	V_{CC}	37	V_{CC}	73	V_{CC}	109	A1
2	GND	38	GND	74	GND	110	A0
3	MSERR	39	ID5	75	A29	111	$\overline{ERLYA}$
4	STAT2	40	ID6	76	A28	112	$\overline{ERR}$
5	STAT1	41	ID7	77	A27	113	$\overline{RDN}$
6	STAT0	42	ID8	78	A26	114	$\overline{RDY}$
7	MPGM1	43	ID9	79	A25	115	$\overline{BGRT}$
8	MPGM0	44	ID10	80	A24	116	GND
9	OPT2	45	ID11	81	A23	117	V_{CC}
10	OPT1	46	ID12	82	A22	118	$\overline{PGMODE}$
11	OPT0	47	ID13	83	A21	119	$\overline{BURST}$
12	$\overline{LOCK}$	48	ID14	84	A20	120	$\overline{REQ}$
13	SUP/$\overline{US}$	49	ID15	85	A19	121	$\overline{BREQ}$
14	$\overline{BWE3}$	50	V_{CC}	86	A18	122	GND
15	$\overline{BWE2}$	51	GND	87	A17	123	INCLK
16	$\overline{BWE1}$	52	ID16	88	A16	124	V_{CC}
17	$\overline{BWE0}$	53	ID17	89	V_{CC}	125	PWRCLK
18	V_{CC}	54	ID18	90	GND	126	MEMCLK
19	GND	55	ID19	91	V_{CC}	127	GND
20	$\overline{WARN}$	56	ID20	92	GND	128	V_{CC}
21	IO/$\overline{MEM}$	57	ID21	93	A15	129	GND
22	I/$\overline{D}$	58	ID22	94	A14	130	$\overline{DIV2}$
23	GND	59	ID23	95	A13	131	CNTL0
24	V_{CC}	60	GND	96	A12	132	CNTL1
25	R/$\overline{W}$	61	V_{CC}	97	A11	133	V_{CC}
26	$\overline{TEST}$	62	GND	98	A10	134	GND
27	TCK	63	ID24	99	A9	135	$\overline{RESET}$
28	TMS	64	ID25	100	A8	136	$\overline{TRAP0}$
29	TDI	65	ID26	101	A7	137	$\overline{TRAP1}$
30	TDO	66	ID27	102	A6	138	$\overline{INTR3}$
31	$\overline{TRST}$	67	ID28	103	A5	139	$\overline{INTR2}$
32	ID0	68	ID29	104	A4	140	$\overline{INTR1}$
33	ID1	69	ID30	105	A3	141	$\overline{INTR0}$
34	ID2	70	ID31	106	A2	142	$\overline{DI}$
35	ID3	71	A31	107	GND	143	$\overline{HIT}$
36	ID4	72	A30	108	V_{CC}	144	$\overline{WBC}$

CERQUAD PIN DESIGNATION (Sorted by Pin Name)

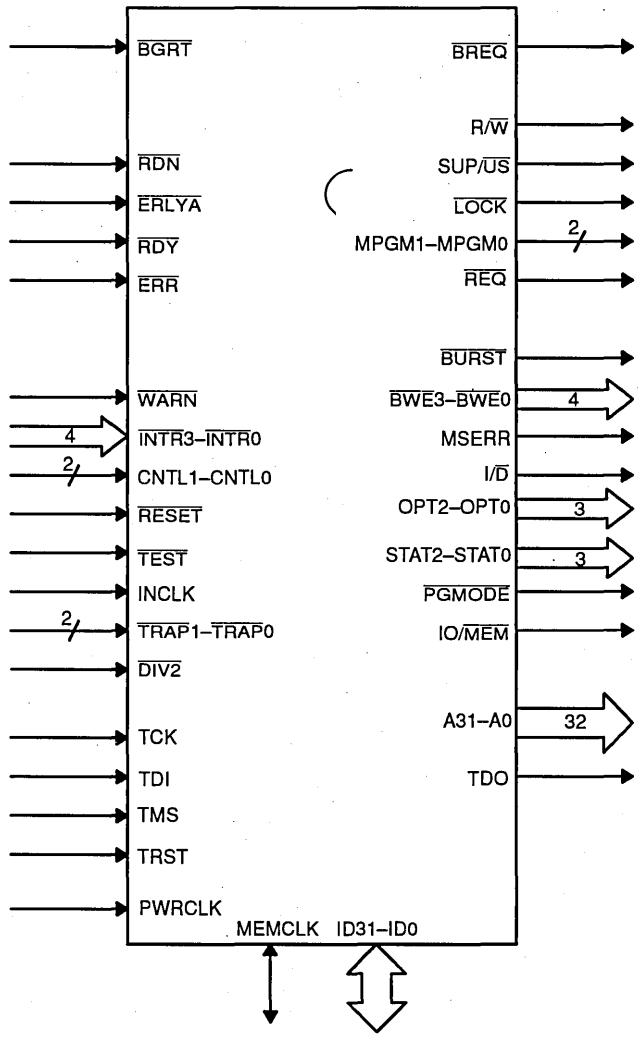
Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
110	A0	16	$\overline{BWE1}$	45	ID11	125	PWRCLK
109	A1	15	$\overline{BWE2}$	46	ID12	113	$\overline{RDN}$
106	A2	14	$\overline{BWE3}$	47	ID13	114	$\overline{RDY}$
105	A3	131	CNTL0	48	ID14	135	$\overline{RESET}$
104	A4	132	CNTL1	49	ID15	120	$\overline{REQ}$
103	A5	130	$\overline{DIV2}$	52	ID16	25	R/$\overline{W}$
102	A6	111	$\overline{ERLYA}$	53	ID17	6	STAT0
101	A7	112	$\overline{ERR}$	54	ID18	5	STAT1
100	A8	2	GND	55	ID19	4	STAT2
99	A9	19	GND	56	ID20	13	SUP/$\overline{US}$
98	A10	23	GND	57	ID21	27	TCK
97	A11	38	GND	58	ID22	29	TDI
96	A12	51	GND	59	ID23	30	TDO
95	A13	60	GND	63	ID24	26	$\overline{TEST}$
94	A14	62	GND	64	ID25	28	TMS
93	A15	74	GND	65	ID26	136	$\overline{TRAP0}$
88	A16	90	GND	66	ID27	137	$\overline{TRAP1}$
87	A17	92	GND	67	ID28	31	$\overline{TRST}$
86	A18	107	GND	68	ID29	1	Vcc
85	A19	116	GND	69	ID30	18	Vcc
84	A20	122	GND	70	ID31	24	Vcc
83	A21	127	GND	123	INCLK	37	Vcc
82	A22	129	GND	141	$\overline{INTR0}$	50	Vcc
81	A23	134	GND	140	$\overline{INTR1}$	61	Vcc
80	A24	22	I/$\overline{D}$	139	$\overline{INTR2}$	73	Vcc
79	A25	32	ID0	138	$\overline{INTR3}$	89	Vcc
78	A26	33	ID1	21	IO/$\overline{MEM}$	91	Vcc
77	A27	34	ID2	12	$\overline{LOCK}$	108	Vcc
76	A28	35	ID3	126	MEMCLK	117	Vcc
75	A29	36	ID4	8	MPGM0	124	Vcc
72	A30	39	ID5	7	MPGM1	128	Vcc
71	A31	40	ID6	3	MSERR	133	Vcc
115	$\overline{BGRT}$	41	ID7	11	OPT0	20	$\overline{WARN}$
121	$\overline{BREQ}$	42	ID8	10	OPT1		
119	$\overline{BURST}$	43	ID9	9	OPT2		
17	$\overline{BWE0}$	44	ID10	118	$\overline{PGMODE}$		

Note: *The following signals are reserved for future processor implementations:*

Pin No.	Pin Name
142	$\overline{DI}$
143	$\overline{HIT}$
144	$\overline{WBC}$

To maintain compatibility with future processor implementations, these pins should be connected to V_{CC} by individual pull-up resistors.

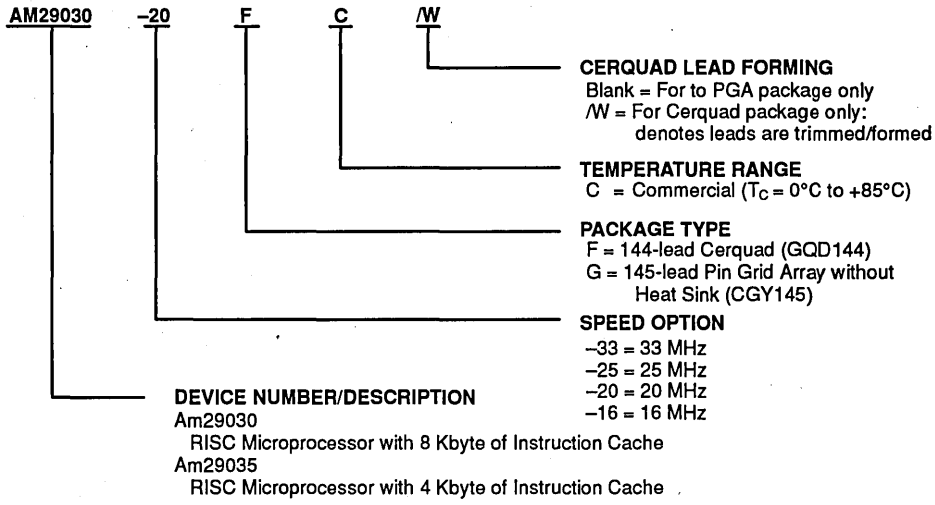
LOGIC SYMBOL



ORDERING INFORMATION

Standard Products

AMD standard products are available in several packages and operating ranges. The order number (Valid Combination) is formed by a combination of the elements below.



Valid Combinations	
AM29030–33 AM29030–25	GC
AM29030–20 AM29035–16	FC/W

Valid Combinations

Valid Combinations list configurations planned to be supported in volume for this device. Consult the local AMD sales office to confirm availability of specific valid combinations and to check on newly released combinations.

ABSOLUTE MAXIMUM RATINGS

Storage Temperature ..... −65°C to +150°C
Voltage on any Pin
with Respect to GND ..... −0.5 to V_{CC} +0.5 V

Stresses above those listed under ABSOLUTE MAXIMUM RATINGS may cause permanent device failure. Functionality at or above these limits is not implied. Exposure to absolute maximum ratings for extended periods may affect device reliability.

OPERATING RANGES

Commercial (C) Devices

Case Temperature (T_C) ..... 0°C to +85°C
Supply Voltage (V_{CC}) ..... +4.75 to +5.25 V

Operating ranges define those limits between which the functionality of the device is guaranteed.

DC CHARACTERISTICS over COMMERCIAL operating ranges

Parameter Symbol	Parameter Description	Test Conditions	Preliminary		
			Min	Max	Unit
V_{IL}	Input Low Voltage		−0.5	0.8	V
V_{IH}	Input High Voltage		2.0	V_{CC} +0.5	V
$V_{ILINCLK}$	INCLK Input Low Voltage		−0.5	0.8	V
$V_{IHINCLK}$	INCLK Input High Voltage		2.0	V_{CC} +0.5	V
$V_{ILMEMCLK}$	MEMCLK Input Low Voltage		−0.5	0.8	V
$V_{IHMEMCLK}$	MEMCLK Input High Voltage		V_{CC} −0.8	V_{CC} +0.5	V
V_{OL}	Output Low Voltage for All Outputs except MEMCLK	I_{OL} = 3.2 mA		0.45	V
V_{OH}	Output High Voltage for All Outputs except MEMCLK	I_{OH} = −400 µA	2.4		V
I_{LI}	Input Leakage Current	0.45 V ≤ V_{IN} ≤ V_{CC} −0.45 V		±10	µA
I_{LO}	Output Leakage Current	0.45 V ≤ V_{OUT} ≤ V_{CC} −0.4 V		±10	µA
I_{CCOP}	Operating Power Supply Current	V_{CC} = 5.25 V, Outputs Floating; Holding $\overline{RESET}$ active with externally supplied MEMCLK 16.7 MHz Cerquad 20 MHz Cerquad 25 MHz PGA 33.3 MHz PGA		 30 30 28 27	mA/MHz
V_{OLC}	MEMCLK Output Low Voltage	I_{OLC} = 20 mA		0.6	V
V_{OHC}	MEMCLK Output High Voltage	I_{OHC} = 20 mA	V_{CC} −0.6		V
I_{OSGND}	MEMCLK GND Short Circuit Current	V_{CC} = 5.0 V	100		mA
I_{OSVCC}	MEMCLK V_{CC} Short Circuit Current	V_{CC} = 5.0 V	100		mA

CAPACITANCE

Parameter Symbol	Parameter Description	Test Conditions	Preliminary		
			Min	Max	Unit
C_{IN}	Input Capacitance	fC = 10 MHz		15	pF
C_{INCLK}	INCLK Input Capacitance			20	pF
C_{MEMCLK}	MEMCLK Capacitance			20	pF
C_{OUT}	Output Capacitance			20	pF
$C_{I/O}$	I/O Pin Capacitance			20	pF

SWITCHING CHARACTERISTICS over COMMERCIAL operating range (PGA)

No.	Parameter Description	Test Conditions	Preliminary				Unit
			25 MHz		33 MHz		
			Min	Max	Min	Max	
1	INCLK Period (T)		40	100	30	100	ns
2	INCLK High Time		16	84	14.4	85.6	ns
3	INCLK Low Time		16	84	14.4	85.6	ns
4	INCLK Rise Time		0	6	0	6	ns
5	INCLK Fall Time		0	6	0	6	ns
6	MEMCLK Delay From INCLK	Notes 1, 2	0	6	0	6	ns
7	Synchronous Output Valid Delay for signals not broken out below	MEMCLK Output MEMCLK Input	1 2	13 17	1 2	13 17	ns
7a	Synchronous Output Valid Delay for ID31–ID0	MEMCLK Output MEMCLK Input	1 2	13 17	1 2	13 17	ns
7b	Synchronous Output Valid Delay for $\overline{\text{PGMODE}}$	MEMCLK Output MEMCLK Input	1 2	13 19	1 2	13 19	ns
8	Synchronous Output Invalid Delay for signals not broken out below	MEMCLK Output MEMCLK Input	1 2	13 17	1 2	13 17	ns
8a	Synchronous Output Invalid Delay for ID31–ID0	MEMCLK Output MEMCLK Input	1 2	13 17	1 2	13 17	ns
8b	Synchronous Output Invalid Delay for $\overline{\text{PGMODE}}$	MEMCLK Output MEMCLK Input	1 2	13 19	1 2	13 19	ns
9	Synchronous Input Setup Time for signals not broken out below (Note 3)	MEMCLK Output MEMCLK Input MEMCLK = INCLK	17 17 12		17 17 12		ns
9a	Synchronous Input Setup Time for ID31–ID0 (Note 3)	MEMCLK Output MEMCLK Input MEMCLK = INCLK	9 9 6		9 9 6		ns
9b	Synchronous Input Setup Time for $\overline{\text{ERR}}$ (Note 3)	MEMCLK Output MEMCLK Input MEMCLK = INCLK	11 11 7		11 11 7		ns
9c	Synchronous Input Setup Time for $\overline{\text{RDN}}$ (Note 3)	MEMCLK Output MEMCLK Input MEMCLK = INCLK	18 18 12		18 18 12		ns
10	Synchronous Input Hold Time	MEMCLK Output MEMCLK= INCLK	0 2		0 2		ns
11	Setup Time for Synchronous $\overline{\text{RESET}}$ Deassertion			2		2	ns
12	Hold Time for Synchronous $\overline{\text{RESET}}$ Deassertion			5		5	ns
13	$\overline{\text{WARN}}$ Pulse Width		4T		4T		ns
14	Asynchronous Input Pulse Width		T+10		T+10		ns
15	MEMCLK High Time	MEMCLK Period=T MEMCLK Period=2T	16 T–3	84 T+3	14.4 T–3	85.6 T+3	ns
16	MEMCLK Low Time	MEMCLK Period=T MEMCLK Period=2T	16 T–3	84 T+3	14.4 T–3	85.6 T+3	ns
17	MEMCLK Rise Time		0	5	0	5	ns
18	MEMCLK Fall Time		0	5	0	5	ns

Notes:

1. *MEMCLK as an input is always CMOS level.*
2. *MEMCLK can drive an external load of 150 pF.*
3. *The input setup times with MEMCLK used as an input are improved if MEMCLK and INCLK are tied to the same clock input. This is possible only if the processor and bus operate at the same frequency.*
4. *Except where noted, measurement conditions are the same as the Am29000 microprocessor.*
5. *All output valid delays are measured with V_{OL} = 1.5 V and V_{OH} = 1.5 V.*

SWITCHING CHARACTERISTICS over COMMERCIAL operating range (Cerquad)

No.	Parameter Description	Test Conditions	Preliminary Information				Unit
			16 MHz		20 MHz		
			Min	Max	Min	Max	
1	INCLK Period (T)		60	100	50	100	ns
2	INCLK High Time		24	76	20	80	ns
3	INCLK Low Time		24	76	20	80	ns
4	INCLK Rise Time		0	6	0	6	ns
5	INCLK Fall Time		0	6	0	6	ns
6	MEMCLK Delay From INCLK	Notes 1, 2	0	6	0	6	ns
7	Synchronous Output Valid Delay for signal not broken out below	MEMCLK Output MEMCLK Input	1 2	20 24	1 2	18 22	ns
7a	Synchronous Output Valid Delay for ID31–ID0	MEMCLK Output MEMCLK Input	1 2	22 26	1 2	20 24	ns
7b	Synchronous Output Valid Delay for $\overline{\text{PGMODE}}$	MEMCLK Output MEMCLK Input	1 2	22 26	1 2	20 24	ns
8	Synchronous Output Invalid Delay for signals not broken out below	MEMCLK Output MEMCLK Input	1 2	20 24	1 2	18 22	ns
8a	Synchronous Output Invalid Delay for ID31–ID0	MEMCLK Output MEMCLK Input	1 2	22 26	1 2	20 24	ns
8b	Synchronous Output Invalid Delay for $\overline{\text{PGMODE}}$	MEMCLK Output MEMCLK Input	1 2	22 26	1 2	22 26	ns
9	Synchronous Input Setup Time for signals not broken out below (Note 3)	MEMCLK Output MEMCLK Input MEMCLK = INCLK	21 21 17		19 19 15		ns
9a	Synchronous Input Setup Time for ID31–ID0 (Note 3)	MEMCLK Output MEMCLK Input MEMCLK = INCLK	17 17 13		15 15 11		ns
9b	Synchronous Input Setup Time for $\overline{\text{ERR}}$ (Note 3)	MEMCLK Output MEMCLK Input MEMCLK = INCLK	17 17 13		15 15 11		ns
9c	Synchronous Input Setup Time for $\overline{\text{RDN}}$ (Note 3)	MEMCLK Output MEMCLK Input MEMCLK = INCLK	21 21 17		19 19 15		ns
10	Synchronous Input Hold Time	MEMCLK Output MEMCLK Input	2 3		2 3		ns
11	Setup Time for Synchronous $\overline{\text{RESET}}$ Deassertion			4		4	ns
12	Hold Time for Synchronous $\overline{\text{RESET}}$ Deassertion			7		7	ns
13	$\overline{\text{WARN}}$ Pulse Width		4T		4T		ns
14	Asynchronous Input Pulse Width		T+10		T+10		ns
15	MEMCLK High Time	MEMCLK Period=T MEMCLK Period=2T	24 T–3	76 T+3	20 T–3	80 T+3	ns
16	MEMCLK Low Time	MEMCLK Period=T MEMCLK Period=2T	24 T–3	76 T+3	20 T–3	80 T+3	ns
17	MEMCLK Rise Time		0	6	0	6	ns
18	MEMCLK Fall Time		0	6	0	6	ns

Notes:

1. MEMCLK as an input is always CMOS level.

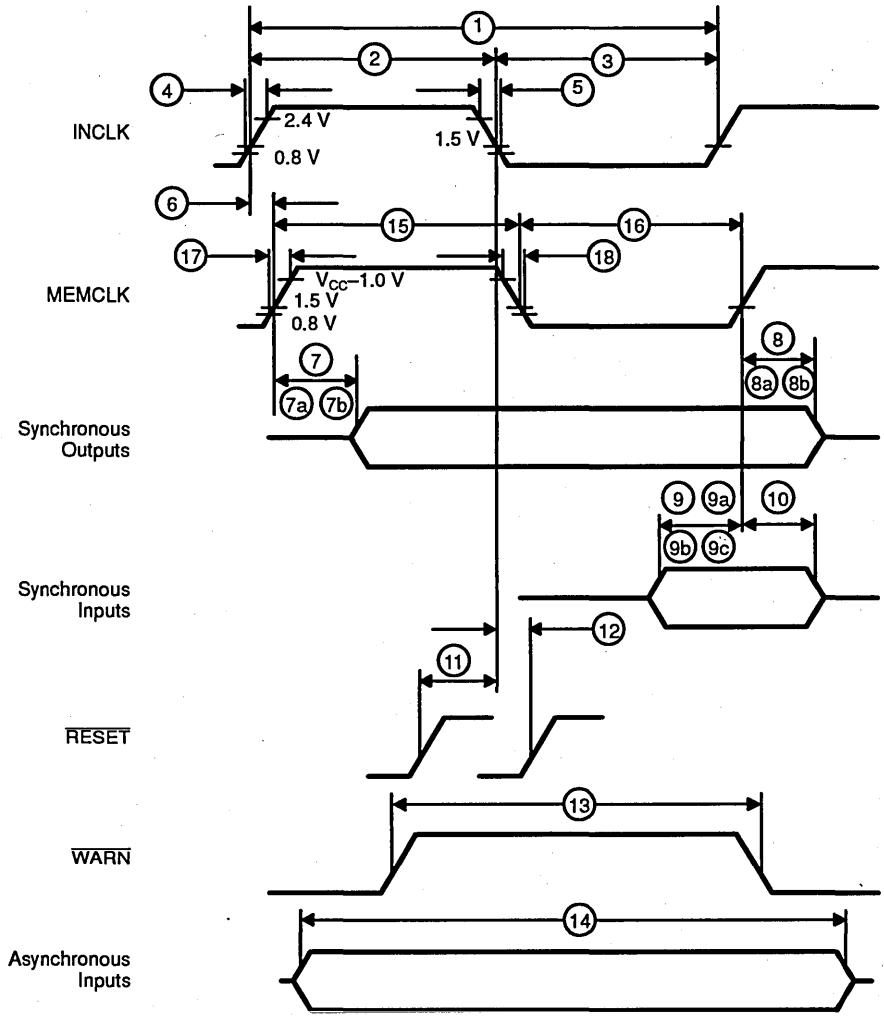
2. MEMCLK can drive an external load of 150 pF.

3. The input setup times with MEMCLK used as an input are improved if MEMCLK and INCLK are tied to the same clock input. This is possible only if the processor and bus operate at the same frequency.

4. Except where noted, measurement conditions are the same as the Am29000 microprocessor.

5. All output valid delays are measured with V_{OL} = 1.5 V and V_{OH} = 1.5 V.

SWITCHING WAVEFORMS



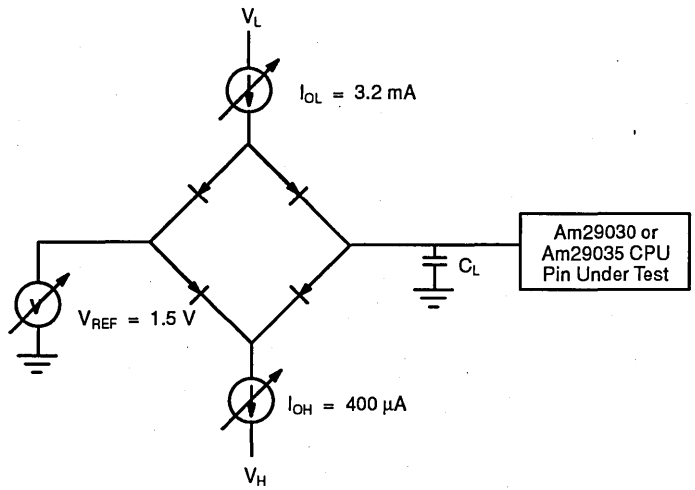
Capacitive Output Delays

For loads greater than 80 pF

The following table describes the additional output delays for capacitive loads greater than 80 pF. Values in the Maximum Additional Delay column should be added to the value listed in the Switching Characteristics table. For loads less than or equal to 80 pF, refer to the delays listed in the Switching Characteristics table. This table applies to the PGA package only.

No.	Parameter Description	Preliminary	
		Total External Capacitance (pF)	Maximum Additional Delay (ns)
7	Synchronous MEMCLK Output Valid Delay	100 pF 150 pF 200 pF 250 pF 300 pF	+1 ns +2 ns +4 ns +6 ns +8 ns
7a	Synchronous MEMCLK Output Valid Delay for ID31–ID0	100 pF 150 pF 200 pF 250 pF 300 pF	+1 ns +6 ns +10 ns +15 ns +19 ns

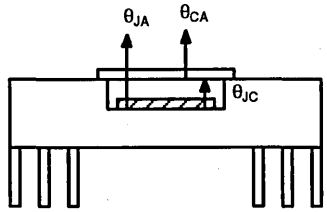
SWITCHING TEST CIRCUIT



Note: *C_L is guaranteed to 80 pF. For capacitive loading greater than 80 pF, refer to the Capacitive Output Delay table.*

THERMAL CHARACTERISTICS

Pin-Grid-Array Package



$\theta_{JA} = \theta_{JC} + \theta_{CA}$

Thermal Resistance – °C/Watt

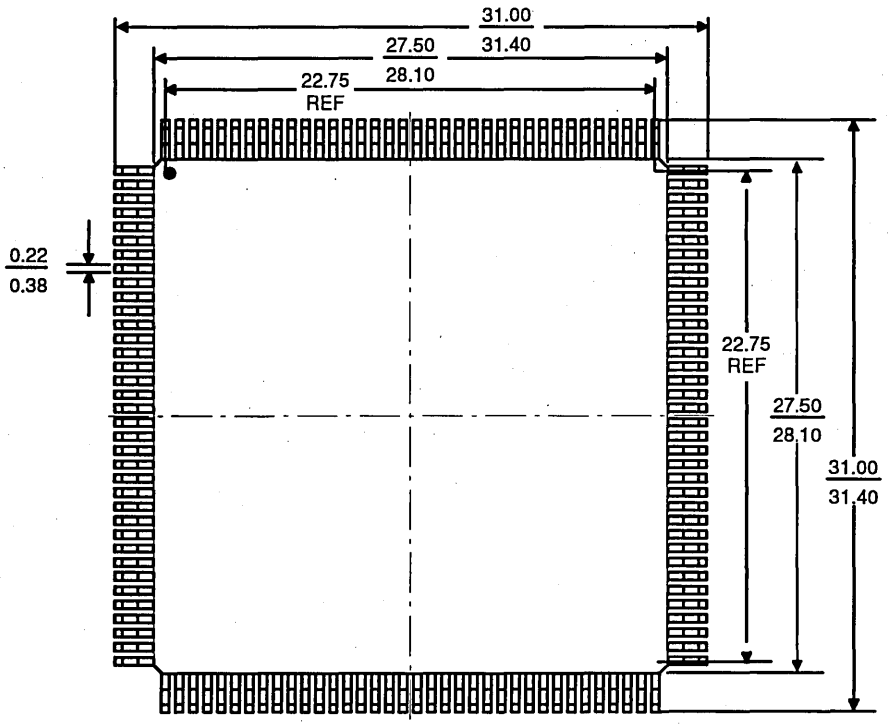
Pin-Grid-Array Package

		Airflow (LFPM)			
		0	150	300	500
θ_{JC}	Junction-to-Case	3	–	–	–
θ_{CA}	Case-to-Ambient	20	18	16	13

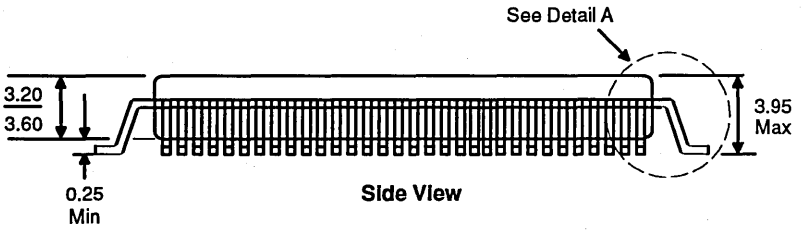
Cerquad Package

		Airflow (LFPM)			
		0	150	300	500
θ_{JC}	Junction-to-Case	7.5	–	–	–
θ_{CA}	Case-to-Ambient	20.8	18.7	16.4	13.3

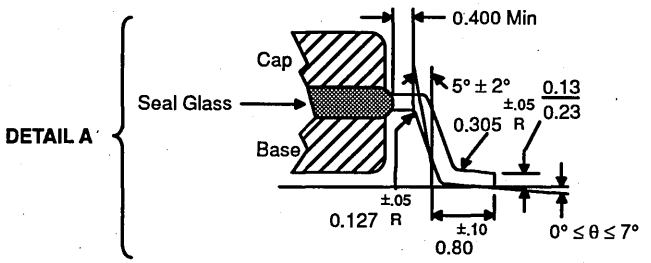
GQD 144 — Cerquad Trimmed and Formed (metric unit)



Top View

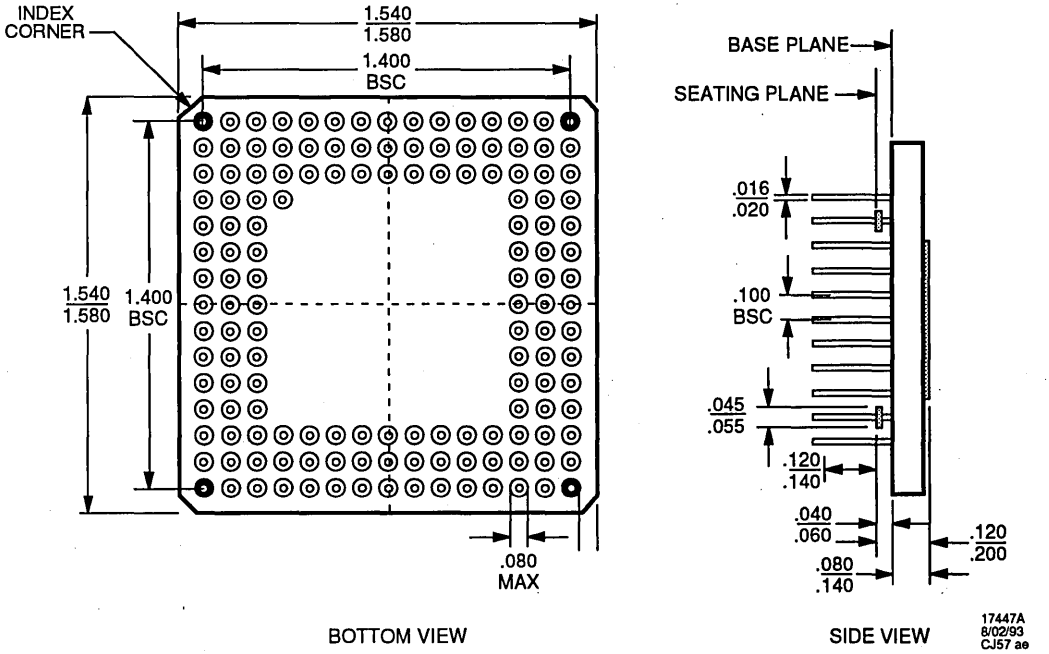


Side View



20000A
CK 64
08/03/93 MH

CGY 145 – Pin Grid Array (PGA)



Note: *Bottom view observed from pin side of package (pins facing viewer).*

INDEX

Sales Offices

North American

ALABAMA ..... (205) 882-9122
ARIZONA ..... (602) 242-4400
CALIFORNIA,
- Culver City ..... (310) 645-1524
- Newport Beach ..... (714) 752-6262
- Sacramento(Roseville) ..... (916) 786-6700
- San Diego ..... (619) 560-7030
- San Jose ..... (408) 452-0500
- Woodland Hills ..... (818) 878-9988

CANADA, Ontario,
- Kanata ..... (613) 592-0060
- Willowdale ..... (416) 222-7800

COLORADO ..... (303) 741-2900
CONNECTICUT ..... (203) 264-7800
FLORIDA,
- Clearwater ..... (813) 530-9971
- Boca Raton ..... (407) 361-0050
- Orlando (Longwood) ..... (407) 862-9292

GEORGIA ..... (404) 449-7920
IDAHO ..... (208) 377-0393
ILLINOIS,
- Chicago (Itasca) ..... (708) 773-4422
- Naperville ..... (708) 505-9517

MARYLAND ..... (301) 381-3790
MASSACHUSETTS ..... (617) 273-3970
MINNESOTA ..... (612) 938-0001
NEW JERSEY,
- Cherry Hill ..... (609) 662-2900
- Parsippany ..... (201) 299-0002

NEW YORK,
- Brewster ..... (914) 279-8323
- Rochester ..... (716) 425-8050

NORTH CAROLINA
- Charlotte ..... (704) 875- 3091
- Raleigh ..... (919) 878- 8111

OHIO,
- Columbus (Westerville) ..... (614) 891-6455
- Dayton ..... (513) 439-0268

OREGON ..... (503) 245-0080
PENNSYLVANIA ..... (215) 398-8006
TEXAS,
- Austin ..... (512) 346-7830
- Dallas ..... (214) 934-9099
- Houston ..... (713) 376-8084

International

BELGIUM, Antwerpen .... TEL ..... (03) 248 43 00
FAX ..... (03) 248 46 42
FRANCE, Paris ..... TEL ..... (1) 49-75-10-10
FAX ..... (1) 49-75-10-13
GERMANY,
- Bad Homburg ..... TEL ..... (06172)-24061
  FAX ..... (06172)-23195
- München ..... TEL ..... (089) 45053-0
  FAX ..... (089) 406490

HONG KONG, ... TEL ..... (852) 865-4525
- Wanchai FAX ..... (852) 865-4335

ITALY, Milano ..... TEL ..... (02) 3390541
FAX ..... (02) 38103458
JAPAN,
- Atsugi ..... TEL ..... (0462) 29-8460
  FAX ..... (0462) 29-8458
- Kanagawa ..... TEL ..... (0462) 47-2911
  FAX ..... (0462) 47-1729

International (Continued)

Tokyo ..... TEL ..... (03) 3346-7550
FAX ..... (03) 3342-5196
Osaka ..... TEL ..... (06) 243-3250
FAX ..... (06) 243-3253
KOREA, Seoul ..... TEL ..... (82) 2-784-0030
FAX ..... (82) 2-784-8014
LATIN AMERICA,
- Ft. Lauderdale ..... TEL ..... (305) 484-8600
  FAX ..... (305) 485-9736

SINGAPORE ..... TEL ..... (65) 3481188
FAX ..... (65) 3480161
SWEDEN,
- Stockholm area ..... TEL ..... (08) 98 61 80
- (Bromma) FAX ..... (08) 98 09 06

TAIWAN, Taipei ..... TEL ..... (886) 2-7153536
FAX ..... (886) 2-7122183
UNITED KINGDOM,
- Manchester area ..... TEL ..... (0925) 830380
- (Warrington) FAX ..... (0925) 830204
- London area ..... TEL ..... (0483) 740440
- (Woking) FAX ..... (0483) 756196

North American Representatives

CANADA
Burnaby, B.C. – DAVETEK MARKETING ..... (604) 430-3680
Kanata, Ontario – VITEL ELECTRONICS ..... (613) 592-0060
Mississauga, Ontario – VITEL ELECTRONICS . (416) 564-9720
Lachine, Quebec – VITEL ELECTRONICS ..... (514) 636-5951
ILLINOIS
- Skokie – INDUSTRIAL REPRESENTATIVES,INC ..... (708) 967-8430

IOWA
- LORENZ SALES ..... (319) 377-4666

KANSAS
- Merriam – LORENZ SALES ..... (913) 469-1312
- Wichita – LORENZ SALES ..... (316) 721-0500

MICHIGAN
- Holland – COM-TEK SALES, INC ..... (616) 335-8418
- Brighton – COM-TEK SALES, INC ..... (313) 227-0007

MINNESOTA
- Mel Foster Tech. Sales, Inc. ..... (612) 941-9790

MISSOURI
- LORENZ SALES ..... (314) 997-4558

NEBRASKA
- LORENZ SALES ..... (402) 475-4660

NEW MEXICO
- THORSON DESERT STATES ..... (505) 883-4343

NEW YORK
- East Syracuse – NYCOM, INC ..... (315) 437-8343
- Hauppauge – COMPONENT CONSULTANTS, INC ..... (516) 273-5050

OHIO
- Centerville – DOLFUSS ROOT & CO ..... (513) 433-6776
- Columbus – DOLFUSS ROOT & CO ..... (614) 885-4844
- Westlake – DOLFUSS ROOT & CO ..... (216) 899-9370

PENNSYLVANIA
- RUSSELL F. CLARK CO.,INC. ..... (412) 242-9500

PUERTO RICO
- COMP REP ASSOC, INC ..... (809) 746-6550

UTAH
- Front Range Marketing ..... (801) 288-2500

WASHINGTON
- ELECTRA TECHNICAL SALES ..... (206) 821-7442

WISCONSIN
- Brookfield – INDUSTRIAL REPRESENTATIVES,INC ..... (414) 574-9393

Advanced Micro Devices, Inc. 901 Thompson Place, P.O. Box 3453, Sunnyvale, CA 94088, USA
Tel: (408) 732-2400 • TWX: 910-339-9280 • TELEX: 34-6306 • TOLL FREE: (800) 538-8450
APPLICATIONS HOTLINE & LITERATURE ORDERING • TOLL FREE: (800) 222-9323 • (408) 749-5703

15723C 5/28/93
Ban-2.55M-11/93-0 Printed in USA

ADVANCED MICRO DEVICES, INC.
901 Thompson Place
P.O. Box 3453
Sunnyvale
California 94088-3453
(408) 732-2400
(800) 538-8450
TWX: 910-339-9280
TELEX: 34-6306

APPLICATIONS HOTLINE & LITERATURE ORDERING
USA (408) 749-5703
JAPAN 011-81-3-3346-7561
UK & EUROPE 44-(0)256-811101
TOLL-FREE
USA (800) 222-9323
FRANCE 0590-8621
GERMANY 0130-813875
ITALY 1678-77224

29K LITERATURE ORDERING
(800) 292-9263
(512) 602-5651

29K SUPPORT PRODUCTS ENGINEERING HOTLINE
USA (800) 2929-AMD
JAPAN 0-031-11-1129

RECYCLED &
RECYCLABLE

Printed in USA
Ban-2.55M-11/93-0
15723C