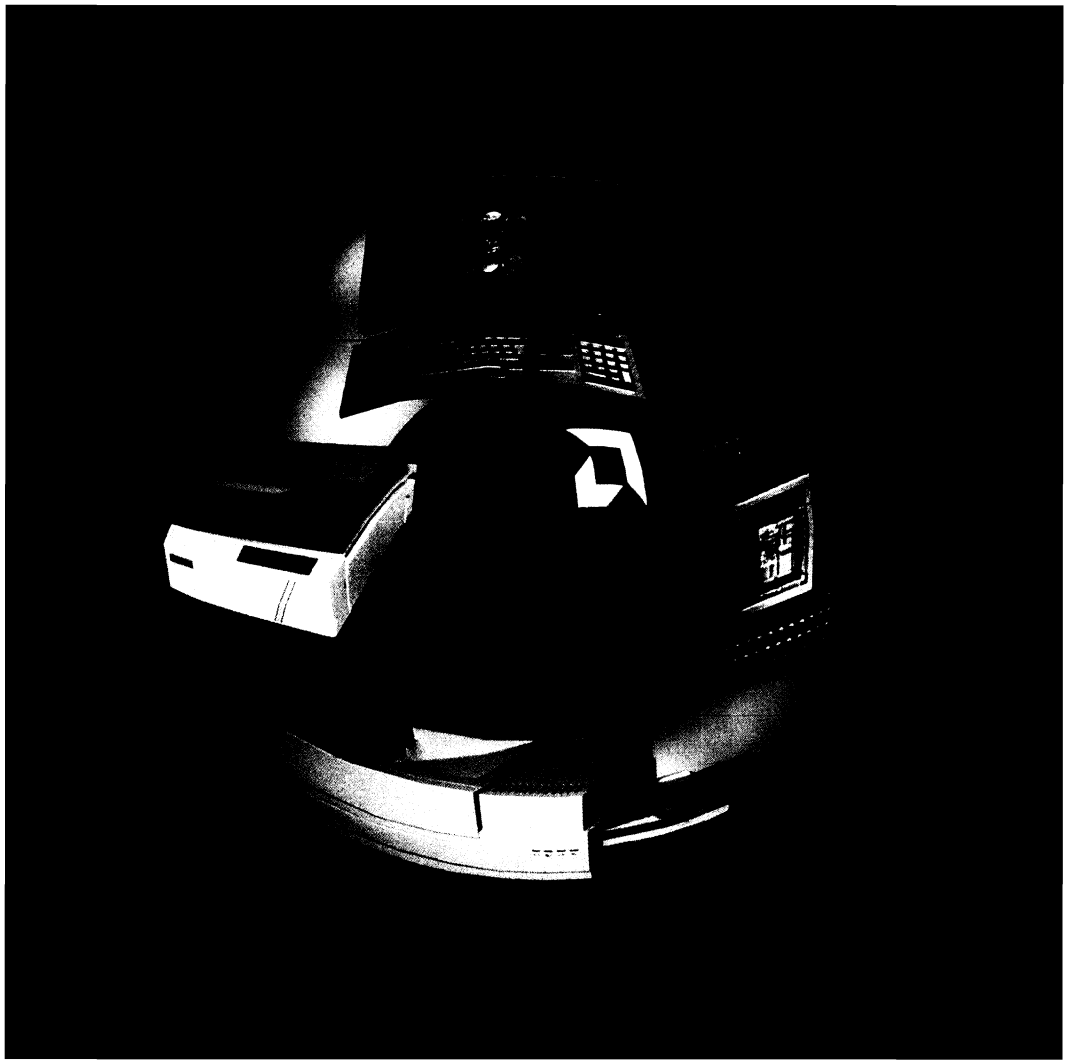
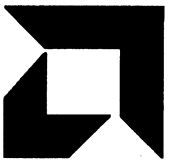
Am29000 User's Manual

32-Bit Streamlined Instruction Processor

Advanced
Micro
Devices



Advanced Micro Devices



Am29000 User's Manual

written and edited by
Mike Johnson
with major contributions from
Bob Perlman, Gigy Baror, Brian Case,
Philip Freidin, Smeeta Gupta, Tim Olson,
and Dave Sorensen

10620C

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

Design Philosophy

The Am29000™ Streamlined Instruction Processor is 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; this, in fact, 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 storage 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 does 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

Am29000 USER MANUAL OVERVIEW

This manual contains information on the Am29000 processor that is essential for computer hardware and software architects and system 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 eight chapters, each viewing the processor from a different perspective, and each with a specific objective:

Chapter 1 introduces the features and performance aspects of the Am29000.

Chapter 2 contains brief technical descriptions of the processor architecture and implementation.

Chapter 3 describes the details of the Am29000 architecture.

Chapter 4 details the operation of the processor's internal functional units.

Chapter 5 describes the operation of the external interfaces of the Am29000.

Chapter 6 describes the attachment and use of coprocessors for the Am29000.

Chapter 7 discusses the implementation of software systems for the processor, focusing on programming features that deserve more coverage than is provided by other chapters.

Chapter 8 specifies the instruction set of the Am29000. It describes the instruction formats in detail, and provides a detailed description of every instruction.

This manual is organized around readers' concerns and objectives. Each chapter focuses on a particular aspect of the processor, and is organized so that it may be read independently, insofar as possible.

For those readers desiring only a brief overview of the Am29000, Chapters 1 and 2 identify the outstanding features of the processor, and give a brief overview of the processor. These chapters address both software and hardware concerns.

For software architects and system programmers interested mainly in software-related issues, Chapters 3, 7, and 8 provide the necessary information.

For hardware architects and systems hardware designers interested mainly in hardware-related issues, Chapters 4 and 5 provide most of the required information; Chapter 8 also provides some related information.

For those readers interested in the coprocessor interface, Chapter 6 describes the interface both from a software and hardware point-of-view.

TECHNICAL CHANGES IN THE THIRD EDITION

Several enhancements and changes have been made to the technical definition of the Am29000 since the second edition of the Am29000 User's Manual. These changes are highlighted here to aid users already familiar with the Am29000.

External Byte and Half-word Accesses

The Am29000 has been enhanced to support byte and half-word loads and stores. This feature is provided as an option, requiring that an external device or memory be able to write individual bytes and/or half-words of a word. The Am29000 can perform all necessary padding, sign-extension, and alignment within the word. Furthermore, this feature is defined to be compatible with existing Am29000 software.

Floating-Point Architecture

This edition of the user's manual specifies Am29000 floating-point arithmetic. Floating-point operations and control registers are implemented by a virtual interface provided by the Am29000, to permit standardization in anticipation of future processor implementations. A new instruction permits classifying floating-point operands (CLASS), and two new instructions have been added for the operations square root (SQRT) and single-to-double-precision multiply (FDMUL). Special-purpose registers added to control floating-point and integer operations include: the Floating-Point Environment Register, the Integer Environment Register, the Floating-Point Status Register, and the Exception Opcode Register.

Multiply Instructions

To improve the potential performance of 32-bit integer multiply operations, these operations have been changed so as not to involve the Q Register. The MULTIPLY and MULTIPLU instructions now place the low-order 32 bits of a result directly into a destination register, avoiding an additional cycle to move these bits from the Q Register at the end of the operation. To support 64-bit integer multiply operations, two new instructions—MULTM and MULTMU—place the high-order 32 bits of a result into a destination register. Integer divide operations continue to use the Q Register.

Test/Development Interface Enhancements

The previous implementation of the Test/Development Interface required that the ADAPT29K™ or other hardware-development system have knowledge of the behavior of the Am29000 pipeline in order to examine internal registers. Furthermore, to transfer data to and from the processor, the hardware-development system was required to synchronously assert and de-assert the *DRDY signal, though all other signals in the Test/Development Interface were asynchronous to the processor clock. Both of these problems have been remedied by re-defining the operation of loads and stores that are executed via the Load Test Instruction mode. These now operate so that there is full asynchronous handshaking on the Test/Development Interface for data transfers.

Hardware Breakpoints

Halt instructions are used by a hardware-development system as instruction breakpoints. However, Halt instructions are also privileged, making it difficult to set hardware breakpoints in a User-mode

program without causing a Protection Violation trap. Previously, this problem was circumvented by a special undocumented mode of operation used by the ADAPT29K. The Am29000 now provides, for general hardware-development system use, a simple means for a hardware-development system to disable protection checking for the Halt instruction, so that this instruction can easily be used to implement hardware breakpoints.

CHAPTER 1

FEATURES AND PERFORMANCE

This chapter provides an evaluation of the Am29000 as an aid in considering a particular application. A detailed technical description of the Am29000 is contained in subsequent chapters. This chapter informally describes the features of the processor, concentrating on features which distinguish the Am29000 from other available processors.

1.1 DISTINCTIVE CHARACTERISTICS

- Full 32-bit architecture.
- CMOS technology / TTL-compatible.
- 20 million instructions per second sustained, at a 30-MHz operating frequency.
- 1.5 clock cycles per instruction average.
- 4 giga-byte virtual address space.
- Double-precision, Floating-Point Arithmetic Unit (Am29027™).
- 192 general-purpose registers.
- Three-address instruction architecture.
- Non-multiplexed, pipelined address, instruction and data buses.
- Concurrent instruction and data accesses.
- Burst-mode access support.
- 512-byte Branch Target Cache™ on-chip.
- 64-entry Memory Management Unit on-chip.
- Demand paging.
- Fully pipelined.
- On-chip Timer Facility.
- On-chip clock generation.
- On-chip debugging support.
- Master/slave chip output checking.

1.2 INTRODUCTION

The Am29000 Streamlined-Instruction Processor is a high-performance, general-purpose, 32-bit microprocessor implemented in complementary metal-oxide semiconductor (CMOS) technology. It supports a variety of applications, using a flexible architecture and rapid execution of simple instructions which are common to a wide range of tasks.

The Am29000 efficiently performs operations common to all systems, while deferring most decisions on system policies to the system architect. It is well-suited for application in high-performance real-time controllers, laser printer controllers, network protocol converters, and many other applications where high performance, flexibility, and the ability to program using standard software tools is important.

The Am29000 instruction set has been influenced by the results of high-level-language, optimizing-compiler research. It is appropriate for a variety of languages, because it efficiently

executes operations which are common to all languages. Consequently, the Am29000 is an ideal target for high-level languages such as C, FORTRAN, Pascal, and Ada.

The Am29000 is packaged in a 169-pin, pin-grid-array (PGA) package, with 141 signal pins, 27 power and ground pins, and 1 alignment pin. A representative system diagram is shown in Figure 1-1.

1.3 PERFORMANCE OVERVIEW

The Am29000 provides a significant margin of performance over other processors in its class, since the majority of processor features were defined with the maximum achievable performance in mind. This section describes the features of the Am29000 from the point-of-view of system performance.

1.3.1 CYCLE TIME

The Am29000 is implemented in CMOS technology, with a 1.2 micron effective transistor-channel length. This technology allows the processor to operate at a frequency of 30 MHz. The processor cycle time is a single, 33-ns clock period. The processor interface drivers can drive 80 pF loads at this frequency.

The Am29000 architecture and system interfaces are designed so that the processor cycle time can decrease with technology improvements. Special attention has been given to the memory interface at high frequencies.

1.3.2 FOUR-STAGE PIPELINE

The Am29000 utilizes a four-stage pipeline, allowing it to execute one instruction every clock cycle. The processor can complete an instruction on every cycle, even though four cycles are required from the beginning of an instruction to its completion.

At a 30-MHz operating frequency, the maximum instruction execution rate is 30 million instructions per second (MIPS). For most other processors, the maximum MIPS rate has little meaning, because it can be achieved only under special circumstances. However, the Am29000 pipeline is designed so that the Am29000 can operate at the maximum instruction-execution rate a significant portion of the time.

Pipeline interlocks are implemented by processor hardware. Except for a few special cases, it is not necessary to re-arrange programs to avoid pipeline dependencies.

1.3.3 SYSTEM INTERFACE

One of the most difficult tasks in the definition of a high-speed micro-processor is the definition of an off-chip interface which supports the operating frequency of the processor, and does not restrict the ability of the processor to fetch instructions and data. If the external interface of a microprocessor cannot support an instruction fetch rate of one instruction every cycle, there is little prospect that the processor will execute at this rate, even though it supports such a rate internally.

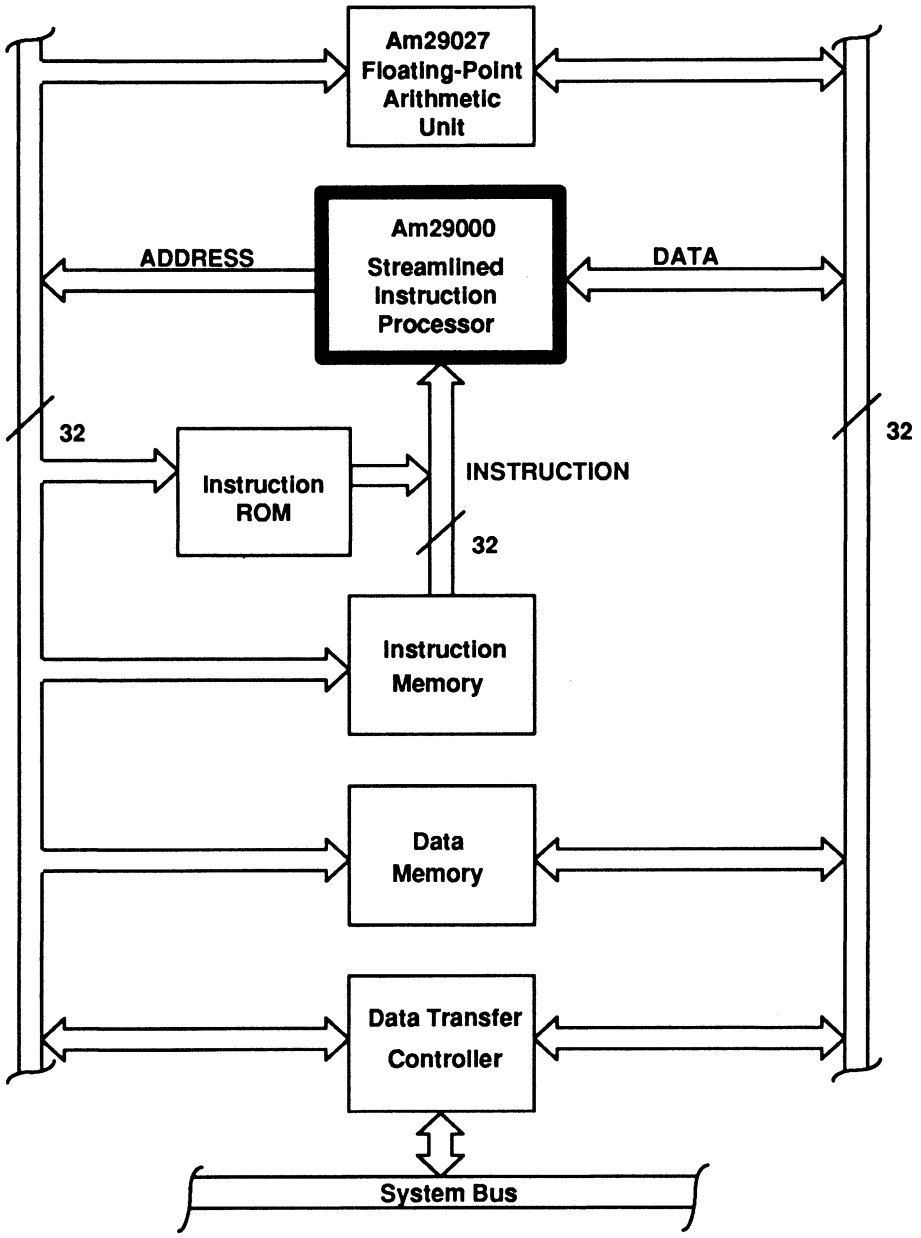


Figure 1-1. Simplified System Diagram

The Am29000 accesses external instructions and data using three non-multiplexed buses. These buses are referred to collectively as the channel. The channel protocol minimizes the logic chains involved in a transfer, and provides a maximum transfer rate of 240 Mbyte/sec.

Separate Address, Instruction, and Data Buses

The Am29000 incorporates two 32-bit buses for instruction and data transfers, and a third address bus which is shared between instruction and data accesses. This bus structure allows simultaneous instruction and data transfers, even though the address bus is shared. The channel achieves the performance of four separate 32-bit buses at a much reduced pin count.

Pipelined Addresses

The Am29000 address bus is pipelined, so that it can be released before an instruction or data transfer is completed. This allows a subsequent access to begin before the first has completed, and allows the processor to have two accesses in progress simultaneously.

Support of Burst Devices and Memories

Burst-mode accesses provide high transfer rates for instructions and data at sequential addresses. For such accesses, the address of the first instruction or datum is sent, and subsequent requests for instructions or data at sequential addresses do not require additional address transfers. These instructions or data are transferred until either party involved in the transfer terminates the access.

Burst-mode accesses can occur at the rate of one access per cycle after the first address has been processed. At 30 MHz, the maximum achievable transfer bandwidth for either instructions or data is 120 Mbyte/sec.

Burst-mode accesses may occur to input/output devices, if the system design permits.

Interface to Fast Devices and Memories

The processor can be interfaced to devices and memories which complete accesses within one cycle. The channel protocol takes maximum advantage of such devices and memories by allowing data to be returned to the processor during the cycle in which the address is transmitted. This allows a full range of memory-speed trade-offs to be made within a particular system.

1.3.4 REGISTER FILE

An on-chip Register File containing 192 general-purpose registers allows most instruction operands to be fetched without the delay of an external access. The Register File incorporates several features which aid the retention of data required by an executing program. Because of the number of general-purpose registers, the frequency of external references for the Am29000 is significantly lower than the frequency of references in processors having only 16 or 32 registers.

Triple-port access to the Register File allows two source-operands to be fetched, in one cycle, while a previously computed result is written. Three 32-bit internal buses prevent contention in the routing of operands. All operand fetches and result write-backs for instruction execution can be performed in a single cycle.

The registers allow efficient procedure linkage, by caching a portion of a compiler's run-time stack. On the average, procedure calls and returns can be executed 5 to 10 times faster (on a cycle-by-cycle basis) than in processors which require the implementation of a run-time stack in external memory (with the attendant loading and storing of registers on procedure call and return).

The registers can contain variables, constants, addresses, and operating-system values. In multi-tasking applications, they can be used to hold the processor status and variables for as many as eight different tasks. A register-banking option allows the register file to be divided into segments which can be individually protected. In this configuration, a task switch can occur in as few as 17 cycles.

1.3.5 INSTRUCTION EXECUTION

The Am29000 uses 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.

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

1.3.6 BRANCH TARGET CACHE

In general, the Am29000 meets its instruction bandwidth requirements via instruction prefetching. However, instruction prefetching is ineffective when a branch occurs. The Am29000 therefore incorporates an on-chip Branch Target Cache to supply instructions for a branch—if this branch has been taken previously—while a new prefetch stream is established.

If branch-target instructions are in the Branch Target Cache, branches execute in a single cycle. This has a very positive effect on processor performance, due to the amount of time the processor could otherwise be idle waiting for the new instruction stream.

As an example, consider that successful branches are 20% of a dynamic instruction mix, and that five cycles are required to restart the processor pipeline after a branch. For 20% of the instructions, the processor would take one cycle to execute the branch instruction and wait five cycles to refill the instruction pipeline. The overhead of branch instructions would be six cycles. If the remaining 80% of the instructions require a single cycle to execute, the latency involved in branching would reduce the average execution rate from one cycle per instruction to two, thus halving the performance.

The Branch Target Cache in the Am29000 has a average hit rate of 60%. In other words, it eliminates the branch latency for 60% of all successful branches on the average.

1.3.7 BRANCHING

Branch conditions in the Am29000 are based on Boolean data contained in general-purpose registers, rather than on arithmetic condition codes. Using a condition-code register for the purpose

of branching—which is common in other processors—inhibits certain compiler optimizations, because the condition-code register is modified by many different instructions. It is difficult for an optimizing compiler to schedule this shared use. By treating branch conditions as any other instruction operand, the Am29000 avoids this problem.

The Am29000 executes branches in a single cycle, for those cases where the target of the branch is in the Branch Target Cache. The single-cycle branch is unusual for a pipelined processor, and is due to processor hardware which allows much of the branch instruction operation to be performed early in the execution of the branch. Single-cycle branching has a dramatic effect on performance, since successful branches typically represent 15% to 25% of a processor's instruction mix.

The techniques used to achieve single-cycle branching also minimize the execution time of branches in those cases where the target is not in the Branch Target Cache. To keep the pipeline operating at the maximum rate, the instruction following the branch, referred to as the delay instruction, is executed regardless of the outcome of the branch. An optimizing compiler can define a useful instruction for the delay instruction in approximately 90% of branch instructions, thereby increasing the performance of branches.

1.3.8 LOADS AND STORES

The performance degradation of load and store operations is minimized in the Am29000 by overlapping them with instruction execution, by taking advantage of pipelining, and by organizing the flow of external data onto the processor so that the impact of external accesses is minimized.

Overlapped Loads and Stores

In the Am29000, a load or store is performed concurrently with execution of instructions which do not have dependencies on the load or store operation. An optimizing compiler can schedule loads and stores in the instruction sequence so that, in most cases, data accesses are overlapped with instruction execution.

Overlapped load and store operations can achieve up to a 30% improvement in performance when data memory has a two-cycle access time. Processor hardware detects dependencies while overlapped loads and stores are being performed, so dependencies have no software implications.

A classical problem in the implementation of overlapped loads and stores is that of dealing with address-translation exceptions in a demand-paged environment. Overlap is not possible if any load or store which encounters an address-translation exception must be restarted by the re-execution of the initiating instruction. In this case, the processor would have to hold instruction execution until the success of every load or store were insured. The Am29000 exception restart mechanism automatically saves information required to restart any load or store, until the operation successfully completes. Thus, it allows the overlapped execution of loads and stores while properly handling address-translation exceptions.

A second problem in the implementation of overlapped loads concerns the handling of data which is returned to the processor upon completion of the load. This data must be written to the register file,

but it contends for register-file write-cycles with other instructions which are being overlapped with the load. This contention may be eliminated by adding a special write port to the register file. However, due to the size of the register file in the Am29000, a fourth port for writing incoming load data is not economical.

The Am29000 data-flow organization avoids the one-cycle penalty which would result from the contention between load data and the results of overlapped instruction execution. Load data is buffered in a latch while awaiting an opportunity to be written into the register file. This opportunity is guaranteed to arise before the next load is executed. While the data is buffered in this latch, it may be used as an instruction operand in place of the destination register for the load.

Load Multiple and Store Multiple

These instructions allow the transfer of the contents of multiple registers to or from external memories or devices. This transfer can occur at a rate of one register-content per cycle.

The advantage of Load Multiple and Store Multiple is best seen in task switching, register-file saving and restoring, and in block data moves. In many systems, such operations require a significant percentage of execution time.

The load-multiple and store-multiple sequences are interruptible, so that they do not affect interrupt latency.

Forwarding of Load Data

Data which is sent to the processor at the completion of a load is forwarded directly to the appropriate execution unit if the data is required immediately by an instruction. This avoids the common one-cycle delay from bus transfer to use of data, and reduces the access latency of external data by one cycle.

1.3.9 MEMORY MANAGEMENT

A 64-entry Translation Look-Aside Buffer (TLB) on the Am29000 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) Early Address Translation—Address translations for load, store, and branch instructions occur during the cycle in which these instructions are executed. This allows the physical address to be transferred externally in the next cycle.

3) Task Identifiers—Task Identifiers allow TLB entries to be matched to different processes, so that TLB invalidation is not required during task switches.

4) Least-Recently Used Hardware—This hardware allows immediate selection of a TLB set to be replaced.

5) Software Reload—Software reload allows the operating system to use a page-mapping scheme which is best matched to its environment. Paged-segmented, one-level-page mapping, two-level-page mapping, or any other user-defined page-mapping scheme can be supported. Because Am29000 instructions execute at an average rate of nearly one instruction per cycle, software reload has a performance approaching that of hardware TLB reload.

1.3.10 INTERRUPTS AND TRAPS

When the Am29000 takes an interrupt or trap, it does not automatically save its current state information. This greatly improves the performance of temporary interruptions such as TLB reload, floating-point emulation, 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 Area, which directs the processor to a routine to handle a given interrupt or trap. The Vector Area may be relocated in memory by the modification of a processor register. There may be multiple Vector Areas in the system, though only one is active at any given time.

The Vector Area is either a table of pointers to the interrupt and trap handlers, or a segment of instruction memory (possibly read-only memory) containing the handlers themselves. The choice between the two possible Vector Area definitions is determined by the cost/performance trade-offs made for a particular system.

If the Vector Area is a table of vectors in data memory, it requires only 1 Kbyte of memory. However, this structure requires that the processor perform a vector fetch every time an interrupt or trap is taken. The vector fetch requires at least 3 cycles, in addition to the number of cycles required for the basic memory access.

If the Vector Area is a segment of instruction memory, it requires a maximum of 64 Kbytes of memory. The advantage of this structure is that the processor begins the execution of the interrupt or trap handler in the minimum amount of time.

1.3.11 FLOATING-POINT ARITHMETIC UNIT

The Am29027 is a Double-Precision, Floating-Point Arithmetic Unit for the Am29000. It can provide an order-of-magnitude performance increase over floating-point operations performed in software. It performs both single-precision and double-precision operations, using IEEE and other floating-point formats. The Am29027 also supports 32-and 64-bit integer functions.

The Am29027 performs floating-point operations using combinatorial—rather than sequential—logic, so that operations with the Am29027 require only five Am29000 cycles.

Floating-point operations may be overlapped with other processor operations. Furthermore, the Am29027 incorporates pipeline registers and eight operand registers, permitting very high throughput for certain types of operations (such as array computations).

The Am29027 attaches directly to the Am29000, using the coprocessor interface. The Am29000 can transfer two 32-bit quantities to the Am29027 in one cycle.

The Am29027 is described in the Am29027 Arithmetic Accelerator Data Sheet (PID# 09114B) and the Am29027 Handbook (PID# 11852A).

1.4 OPTIMIZING COMPILERS

The number of instructions used to perform a given task is minimized by optimizing compilers which are supplied for the Am29000. A full discussion of optimizing-compiler technology is beyond the scope of this manual, but there are a few concepts which should be mentioned here, because the Am29000 was designed to be an excellent target for optimizing compilers.

1.4.1 OPTIMIZING-COMPILER OVERVIEW

In addition to performing the same tasks as any other compiler, an optimizing compiler re-arranges the generated code to minimize its size and execution time. This optimization occurs after the initial phases of code generation have been completed. The optimizer inspects large portions of the compiled program for frequently occurring cases where the compiled results can be improved.

Many optimization opportunities arise precisely because the code is compiler generated. Code translation is an automated process, so the initial phases of the compiler often generate code that is much less than optimum. However, the optimizer can produce results which are often better than those produced by human assembly-language programmers, because it can deal with large portions of the program and an immense amount of data concerning program behavior.

1.4.2 OPTIMIZING-COMPILER OPERATION

Conceptually, the optimizer arranges program flow and the creation, modification, and use of program data to minimize the amount of time required to perform a given task. The reduction in program space is a normal side-benefit of the reduction in execution time. The optimizer is concerned not only with data explicit in the high-level program, but also with data created by other phases of the compiler in order to properly translate the program (for example, temporary values created during the evaluation of expressions). Optimization involves the following sorts of operations:

1) Reusing results rather than repeating computations. The optimizer attempts to eliminate redundant computations by performing a computation once, and saving the result for later use. Often these redundant computations are not apparent in the original program, but are created by the underlying definitions of high-level operations.

2) Reducing the amount of code executed within loops. In many cases, only a few computations change on different loop iterations. The optimizer attempts to reduce the amount of work

performed within loops to a minimum, by moving loop-invariant computations outside of loops.

3) Replacing slow operations by faster ones. The optimizer can recognize special cases of multiply and divide, for example, and replace them with faster shift and add instructions. The slow operations, again, often are generated by earlier phases of the compiler because these operations are most general, and the early code-generation phases cannot recognize the special cases which allow the operations to be replaced with faster ones.

4) Allocating processor registers so that they contain frequently used data. This reduces the number of relatively slow memory references, and replaces them by faster register references.

5) Scheduling the execution of instructions. The optimizer attempts to move instructions to a point in the program flow where they create fewer problems for the processor pipeline. For example, a register load may be moved to a point in the instruction sequence where its memory reference can be overlapped with other instructions.

Most optimizations performed rely heavily on two types of information collected by the optimizer: the first type deals with program flow, and the second with data dependencies which arise because of the program flow. The optimizer can tailor the code to the high-level task being compiled, not because it understands the task being performed by the high-level program, but because it understands the dependencies which arise in the generated code. As a result, it can adjust the instruction sequence to minimize the performance impact of these dependencies.

It is important to note that the optimizer does not directly optimize a given program, but rather optimizes a special representation of the program which is suitable for analysis and modification by the optimizer, which is, after all, just another program. The key to optimization is that this representation be easy to analyze for program and data-flow information, and that it be easy to rearrange when optimizations are performed.

1.4.3 THE Am29000 AND OPTIMIZING COMPILERS

General Principles

The primary principle behind the Am29000 instruction set is that it matches the internal representation used by optimizing compilers to perform optimization. As discussed above, this representation is not arbitrary, but is rather strictly defined by the optimization algorithms.

It is important to realize that optimizations performed for the Am29000 would have limited effectiveness if applied to so-called complex-instruction processors. There are several fundamental problems that limit the effectiveness of optimizations for these other processors.

The first problem with complex-instruction sets is that they normally provide a variety of instruction sequences which perform the same function as a sequence of instructions in the compiler's internal representation, but do not match it exactly. The trade-offs made by a compiler to decide among the available choices can be very complex.

In the first place, it is difficult for the compiler to determine the difference in execution time between multiple instruction sequences, because of the amount of information involved. For example, just changing the addressing mode of an instruction can change the execution time. This is further complicated in the cases where the compiled program is to be run on different implementations of the same processor, where execution times can depend on the implementation. If there is only one instruction sequence to choose from, and if all instructions execute in a single cycle, this problem is reduced greatly.

During the generation of code for a complex-instruction processor, it is nearly impossible to guarantee that the choice of a given code sequence will not force a less-than-optimum choice of code at some later point in the translation. Restrictions arise late in translation because of decisions made earlier. Often, these restrictions arise because of interactions between instructions; they are especially severe when instructions operate only on a specific register or group of registers.

An additional problem with complex instruction sets is that optimizations applied to them do not necessarily save execution time. An optimization may not be reflected in the final compiled code, because the instruction set may inhibit the realization of the optimization. However, in the case of the Am29000, an optimization is guaranteed to eliminate one or more execution cycles, because all processor operations are exposed to the compiler.

The greatest benefit of exposing all processor operations to the compiler appears within loops, which is where processors spend a great deal of their execution time. The problem with complex instruction sets here is that, when an instruction set forces multiple operations with one instruction, the processor spends much time performing redundant computations within loops. Many times, the redundant computations are performed by microcode, which cannot detect that a computation is loop-invariant, because it knows nothing of loops. The compiler is in no position to do much about this, because it cannot remove the loop-invariant computations from the micro-sequence; it is forced to accept the definitions of the instructions as they are.

If an instruction set is defined so that all hardware-level operations are available to the compiler, the compiler is free to construct any sequence of these operations. This allows the movement of loop-invariant computations out of loops, which can result in tremendous performance improvements.

Special Am29000 Features

In addition to the above considerations, there are several other central principles behind the definition of the Am29000.

The Am29000 instruction set reduces the number of instructions required for most general-purpose tasks by providing a complete set of operations. The instruction set is streamlined, but there is no attempt to minimize the number of instructions. Rather, the goal is to minimize the number of instructions required to execute most high-level language programs.

With a few minor exceptions, Am29000 instructions execute in a single cycle. As a result, the performance of an Am29000 instruction sequence is very easy to predict, simplifying the task of

compiler instruction-selection. In addition, single-cycle instruction execution allows the Am29000 to take the maximum advantage of a high-performance system design. Instructions are executed at approximately the rate at which they are supplied to the processor. The Am29000 does not artificially constrain the instruction-execution rate by forcing instructions to require multiple cycles for execution.

The Am29000 contains a large number of registers which facilitate compiler optimizations. These registers allow frequently used variables to be accessed quickly, provide a large number of temporary locations for the reuse of computational results, and simplify inter-procedural communication. The compiler is free to allocate these registers as required to improve performance. Register allocation is relatively simple, because there is such a large number of registers.

For other processors which have fixed register-addressing, a compiler has difficulties allocating the usage of registers, because registers must be allocated statically at compile time. Procedure calls present the greatest difficulty. It is impossible for the compiler to determine exactly which procedures will be called during execution, and in what order they will be called. Thus, it is impossible to precisely allocate the usage of registers across procedure-call boundaries.

Since the Am29000 local registers are addressed relative to a Stack Pointer, compiler register-allocation is simplified. The local registers are allocated dynamically during execution. Thus, the compiler need not be concerned about the allocation of registers across procedure boundaries; this is handled automatically by the local-register addressing.

Am29000 pipelining is exposed to the compiler in the form of delayed branches and overlapped loads and stores. The compiler is free to arrange instructions to reduce the performance impact of the processor pipeline. However, the compiler arranges instructions only because of the performance benefits. Pipeline interlocks in the Am29000 guarantee correct operation in any case.

CHAPTER 2

ARCHITECTURE HIGHLIGHTS

This chapter gives a brief overview of the Am29000 architecture, grouped into programming-related features, hardware features, and system interfaces. The technical information given in this chapter is also contained in subsequent chapters. Much of the detail is omitted here, since the objective is to provide a framework for understanding the information in later chapters.

Where appropriate, section titles in this chapter are followed by references to sections appearing in subsequent chapters. The referenced sections contain related detailed information.

2.1 PROGRAMMER REFERENCE OVERVIEW

This section gives a brief description of the Am29000 from a programmer's point of view. It introduces the processor's program modes, registers, and instructions. An overview of the processor's data formats and handling is given. This section also briefly describes interrupts and traps, memory management, and the coprocessor interface. Finally, the Timer Facility and Trace Facility are introduced.

2.1.1 PROGRAM MODES (see Section 3.1)

There are two mutually exclusive modes of program execution; the Supervisor mode, and the User mode. In the Supervisor mode, executing programs have access to all processor resources. In the User mode, certain processor resources may not be accessed; any attempted access causes a trap.

2.1.2 VISIBLE REGISTERS (see Section 3.2)

The Am29000 incorporates three classes of registers which are accessed and manipulated by instructions: general-purpose registers, special-purpose registers, and Translation Look-Aside Buffer (TLB) registers.

General-Purpose Registers (see Section 3.2.1)

The Am29000 has 192 general-purpose registers. With a few exceptions, general-purpose registers are not dedicated to any special use, and are available for any appropriate program use.

Most processor instructions are three-address instructions. An instruction specifies any three of the 192 registers for use in instruction execution. Normally, two of these registers contain source-operands for the instruction, and a third stores the result of the instruction.

The 192 registers are divided into 64 global and 128 local registers. Global registers are addressed with absolute register numbers, while local registers are addressed relative to an internal Stack Pointer.

For fast procedure calling, a portion of a compiler's run-time stack can be mapped into the local registers. Statically allocated variables, temporary values, and operating-system parameters are kept in the global registers.

The Stack Pointer for local registers is mapped to Global Register 1. The Stack Pointer is a full 32-bit virtual address for the top of the run-time stack.

The general-purpose registers may be accessed indirectly, with the register number specified by the content of a special-purpose register (see below) rather than by an instruction field. Three independent indirect register numbers are contained in three separate special-purpose registers. Indirect addressing is accomplished by specifying Global Register 0 as an instruction operand or result register. An instruction can specify an indirect register access for any or all of the source operands or result.

General-purpose registers may be partitioned into segments of 16 registers for the purpose of access protection. A register in a protected segment may be accessed only by a program executing in the Supervisor mode. An attempted access (either read or write) by a User-mode program causes a trap to occur.

Special-Purpose Registers (see Section 3.2.2)

The Am29000 contains 27 special-purpose registers. These registers provide controls and data for certain processor functions.

Special-purpose registers are accessed by data movement only. Any special-purpose register can be written with the contents of any general-purpose register, and any general-purpose register can be written with the contents of any special-purpose register. Operations cannot be performed directly on the contents of special-purpose registers.

Some special-purpose registers are protected, and can be accessed only in the Supervisor mode. This restriction applies to both read and write accesses. An attempt by a User-mode program to access a protected register causes a trap to occur.

The protected special-purpose registers are defined as follows:

1) Vector Area Base Address—Defines the beginning of the interrupt/trap Vector Area.

2) Old Processor Status—Receives a copy of the Current Processor Status (see below) when an interrupt or trap is taken. It is later used to restore the Current Processor Status on an interrupt return.

3) Current Processor Status—Contains control information associated with the currently executing process, such as interrupt disables and the Supervisor Mode bit.

4) Configuration—Contains control information which normally varies only from system to system, and usually is set only during system initialization.

5) Channel Address—Contains the address associated with an external access, and retains the address if the access does not complete successfully. The Channel Address Register, in conjunction with the Channel Data and Channel Control registers described below, allow the

restarting of unsuccessful external accesses. This might be necessary for an access encountering a page fault in a demand-paged environment, for example.

6) Channel Data—Contains data associated with a store operation, and retains the data if the operation does not complete successfully.

7) Channel Control—Contains control information associated with a channel operation, and retains this information if the operation does not complete successfully.

8) Register Bank Protect—Restricts access of User-mode programs to specified groups of 16 registers. This facilitates register banking for multi-tasking applications, and protects operating-system parameters kept in the global registers from corruption by User-mode programs.

9) Timer Counter—Supports real-time control and other timing-related functions.

10) Timer Reload—Maintains synchronization of the Timer Counter. It includes control bits for the Timer Facility.

11) Program Counter 0—Contains the address of the instruction being decoded when an interrupt or trap is taken. The processor restarts this instruction upon interrupt return.

12) Program Counter 1—Contains the address of the instruction being executed when an interrupt or trap is taken. The processor restarts this instruction upon interrupt return.

13) Program Counter 2—Contains the address of the instruction just completed when an interrupt or trap is taken. This address is provided for information only, and does not participate in an interrupt return.

14) MMU Configuration—Allows selection of various memory-management options, such as page size.

15) LRU Recommendation—Simplifies the reload of entries in the Translation Look-Aside Buffer (TLB) by providing information on the least-recently used entry of the TLB when a TLB miss occurs (see Section 2.1.6).

The unprotected special-purpose registers are defined as follows:

1) Indirect Pointer C—Allows the indirect access of a general-purpose register.

2) Indirect Pointer A—Allows the indirect access of a general-purpose register.

3) Indirect Pointer B—Allows the indirect access of a general-purpose register.

4) Q—Provides additional operand bits for multiply step, divide step, and divide operations.

5) ALU Status—Contains information about the outcome of integer arithmetic and logical operations, and holds residual control for certain instruction operations.

6) Byte Pointer—Contains an index of a byte or half-word within a word. This register is also accessible via the ALU Status Register.

7) Funnel Shift Count—Provides a bit offset for the extraction of word-length fields from double-word operands. This register is also accessible via the ALU Status Register.

8) Load/Store Count Remaining—Maintains a count of the number of loads and stores remaining for load-multiple and store-multiple operations. The count is initialized to the total number of loads or stores to be performed before the operation is initiated. This register is also accessible via the Channel Control Register.

9) Floating-Point Environment—Controls the operation of floating-point arithmetic, such as rounding modes and exception reporting.

10) Integer Environment—Enables and disables the reporting of exceptions which occur during integer multiply and divide operations.

11) Floating-Point Status—Contains information about the outcome of floating-point operations.

12) Exception Opcode—Reports the operation code of an instruction causing a trap. This register is provided primarily for recovery from floating-point exceptions, but is also set for other instructions that cause traps.

TLB Registers (see Section 3.2.3)

Translation Look-Aside Buffer (TLB) entries in the Am29000 Memory Management Unit are accessed via 128 TLB registers. A single TLB entry appears as two TLB registers; TLB registers are thus paired according to the corresponding TLB entry.

TLB registers are accessed by data movement only. Any TLB register can be written with the contents of any general-purpose register, and any general-purpose register can be written with the contents of any TLB register. Operations cannot be performed directly on the contents of TLB registers.

TLB registers can be accessed only in the Supervisor mode. This restriction applies to both read and write accesses. An attempt by a User-mode program to access a TLB register causes a trap to occur.

2.1.3 INSTRUCTION SET OVERVIEW (see Section 3.3 and Chapter 8)

The three-address architecture of the Am29000 instruction set allows a compiler or assembly-language programmer to prevent the destruction of operands, and aids register allocation and operand reuse. Instruction operands may be contained in any two of the 192 general-purpose registers, and instruction results may be stored in any of the 192 general-purpose registers.

The compiler or assembly-language programmer has complete freedom to allocate register usage. There is no dedication of a particular register or register group to a particular class of operations. The instruction set is designed to minimize the number of side effects and implicit operations of instructions.

Most Am29000 instructions can specify an 8-bit constant as one of the source operands. Larger constants are constructed using one or two additional instructions and a general-purpose register. Relative branch instructions specify a 16-bit, signed, word offset. Absolute branches specify a 16-bit word address.

The Am29000 instruction set contains 117 instructions. These instructions are divided into nine classes:

1) Integer Arithmetic—Perform integer add, subtract, multiply, and divide operations.

2) Compare—Perform arithmetic and logical comparisons. Some instructions in this class allow the generation of a trap if the comparison condition is not met.

3) Logical—Perform a set of bit-wise Boolean operations.

4) Shift—Perform arithmetic and logical shifts, and allow the extraction of 32-bit words from 64-bit double-words.

5) Data Movement—Perform movement of data fields between registers, and the movement of data to and from external devices and memories.

6) Constant—Allow the generation of large constant values in registers.

7) Floating-Point—Included for floating-point arithmetic, comparisons, and format conversions. These instructions are not currently implemented directly in processor hardware.

8) Branch—Perform program jumps and subroutine calls.

9) Miscellaneous—Perform miscellaneous control functions and operations not provided by other classes.

The Am29000 executes all instructions in a single cycle, except for interrupt returns, Load Multiple, and Store Multiple.

Table 2-1 shows a complete list of Am29000 instructions, listed alphabetically by instruction mnemonic. Table 2-1 is provided only to give a general overview of the instruction set. Section 3.3 defines the instructions grouped into classes, and Chapter 8 provides a detailed specification of the instruction set.

2.1.4 DATA FORMATS AND HANDLING (see Section 3.4)

This section introduces the data formats and data-manipulation mechanisms which are supported by the Am29000.

Table 2-1. Am29000 Instruction Set

Mnemonic	Instruction Name
ADD	Add
ADDC	Add with Carry
ADDCS	Add with Carry, Signed
ADDCU	Add with Carry, Unsigned
ADDS	Add, Signed
ADDU	Add, Unsigned
AND	AND Logical
ANDN	AND-NOT Logical
ASEQ	Assert Equal To
ASGE	Assert Greater Than or Equal To
ASGEU	Assert Greater Than or Equal To, Unsigned
ASGT	Assert Greater Than
ASGTU	Assert Greater Than, Unsigned
ASLE	Assert Less Than or Equal To
ASLEU	Assert Less Than or Equal To, Unsigned
ASLT	Assert Less Than
ASLTU	Assert Less Than, Unsigned
ASNEQ	Assert Not Equal To
CALL	Call Subroutine
CALLI	Call Subroutine, Indirect
CLASS	Classify Floating-Point Operand
CLZ	Count Leading Zeros
CONST	Constant
CONSTH	Constant, High
CONSTN	Constant, Negative
CONVERT	Convert Data Format
CPBYTE	Compare Bytes
CPEQ	Compare Equal To
CPGE	Compare Greater Than or Equal To
CPGEU	Compare Greater Than or Equal To, Unsigned
CPGT	Compare Greater Than
CPGTU	Compare Greater Than, Unsigned
CPLE	Compare Less Than or Equal To
CPLEU	Compare Less Than or Equal To, Unsigned
CPLT	Compare Less Than
CPLTU	Compare Less Than, Unsigned
CPNEQ	Compare Not Equal To

Table 2-1. Am29000 Instruction Set (Continued)

Mnemonic	Instruction Name
DADD	Floating-Point Add, Double-Precision
DDIV	Floating-Point Divide, Double-Precision
DEQ	Floating-Point Equal To, Double-Precision
DGE	Floating-Point Greater Than or Equal To, Double-Precision
DGT	Floating-Point Greater Than, Double-Precision
DIV	Divide Step
DIV0	Divide Initialize
DIVIDE	Integer Divide, Signed
DIVIDU	Integer Divide, Unsigned
DIVL	Divide Last Step
DIVREM	Divide Remainder
DMUL	Floating-Point Multiply, Double-Precision
DSUB	Floating-Point Subtract, Double-Precision
EMULATE	Trap to Software Emulation Routine
EXBYTE	Extract Byte
EXHW	Extract Half-Word
EXHWS	Extract Half-Word, Sign-Extended
EXTRACT	Extract Word, Bit-Aligned
FADD	Floating-Point Add, Single-Precision
FDIV	Floating-Point Divide, Single-Precision
FDMUL	Floating-Point Multiply, Single-to-Double Precision
FEQ	Floating-Point Equal To, Single-Precision
FGE	Floating-Point Greater Than or Equal To, Single-Precision
FGT	Floating-Point Greater Than, Single-Precision
FMUL	Floating-Point Multiply, Single-Precision
FSUB	Floating-Point Subtract, Single-Precision
HALT	Enter Halt Mode
INBYTE	Insert Byte
INHW	Insert Half-Word
INV	Invalidate
IRET	Interrupt Return
IRETINV	Interrupt Return and Invalidate
JMP	Jump
JMPF	Jump False
JMPFDEC	Jump False and Decrement
JMPFI	Jump False Indirect
JMPI	Jump Indirect
JMPT	Jump True
JMPTI	Jump True Indirect

Table 2-1. Am29000 Instruction Set (Continued)

Mnemonic	Instruction Name
LOAD	Load
LOADL	Load and Lock
LOADM	Load Multiple
LOADSET	Load and Set
MFSR	Move from Special Register
MFTLB	Move from Translation Look-Aside Buffer Register
MTSR	Move to Special Register
MTSRIM	Move to Special Register Immediate
MTTLB	Move to Translation Look-Aside Buffer Register
MUL	Multiply Step
MULL	Multiply Last Step
MULTIPLU	Integer Multiply, Unsigned
MULTIPLY	Integer Multiply, Signed
MULTM	Integer Multiply Most-Significant Bits, Signed
MULTMU	Integer Multiply Most-Significant Bits, Unsigned
MULU	Multiply Step, Unsigned
NAND	NAND Logical
NOR	NOR Logical
OR	OR Logical
SETIP	Set Indirect Pointers
SLL	Shift Left Logical
SQRT	Square Root
SRA	Shift Right Arithmetic
SRL	Shift Right Logical
STORE	Store
STOREL	Store and Lock
STOREM	Store Multiple
SUB	Subtract
SUBC	Subtract with Carry
SUBCS	Subtract with Carry, Signed
SUBCU	Subtract with Carry, Unsigned
SUBR	Subtract Reverse
SUBRC	Subtract Reverse with Carry
SUBRCS	Subtract Reverse with Carry, Signed
SUBRCU	Subtract Reverse with Carry, Unsigned
SUBRS	Subtract Reverse, Signed
SUBRU	Subtract Reverse, Unsigned
SUBS	Subtract Signed
SUBU	Subtract Unsigned
XNOR	Exclusive-NOR Logical
XOR	Exlusive-OR Logical

Data Types (see Sections 3.4.1, 3.4.2, and 3.4.3)

A word is defined as 32 bits of data. A half-word consists of 16 bits, and a double-word consists of 64 bits. Bytes are 8 bits in length. The Am29000 has direct support for word-integer (signed and

unsigned), word-logical, word-Boolean, half-word integer (signed and unsigned), and character (signed and unsigned) data.

Other data types, such as character strings, are supported with sequences of basic instructions and/or external hardware. Single- and double-precision floating-point types are defined for the Am29000, but are not supported directly by hardware.

The format for Boolean data used by the processor is such that the Boolean values TRUE and FALSE are represented by 1 and 0, respectively, in the most-significant bit of a word.

Figure 2-1 illustrates the numbering conventions for data units contained in a word. Within a word, bits are numbered in increasing order from right-to-left, starting with the number 0 for the least-significant bit. Bytes and half-words within a word are numbered in increasing order starting with the number 0. However, bytes and half-words may be numbered right-to-left (sometimes referred to as "little endian"), or left-to-right (sometimes referred to as "big endian", as controlled by the Configuration Register.

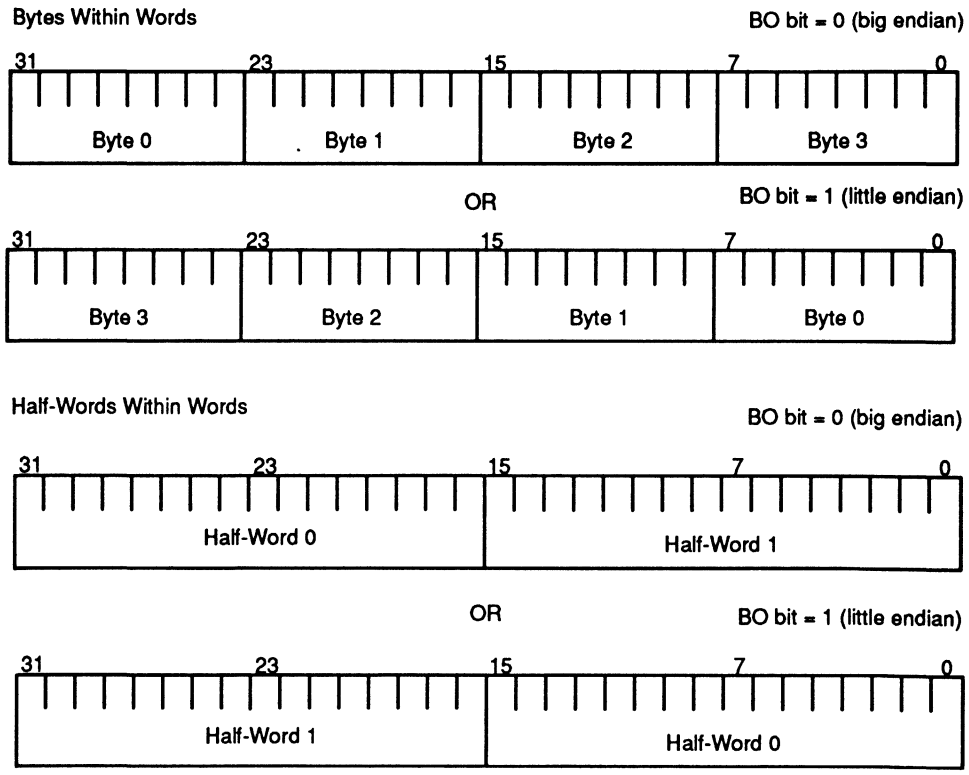


Figure 2-1. Data-Unit Numbering Conventions

Note that the numbering of bits within words is strictly for notational convenience. In contrast, the numbering conventions for bytes and half-words within words affect processor operations.

External Data Accesses (see Section 3.4.4)

External accesses move data between the processor and external devices and memories. These accesses occur only as a result of load and store instructions.

Load and store instructions move words of data to and from general-purpose registers. Each load and store instruction moves a single word. There are load and store instructions which support interlocking operations necessary for multi-processor exclusion, synchronization, and communication.

For the movement of multiple words, Load Multiple and Store Multiple instructions move the contents of sequentially addressed external locations to or from sequentially numbered general-purpose registers. The Load Multiple and Store Multiple allow the movement of up to 192 words at a maximum rate of one word per processor cycle. The multiple load and store sequences may be interrupted, and restarted at the point of interruption.

Load and store instructions provide no mechanism for computing the address associated with the external data access. All addresses are contained in a general-purpose register at the beginning of the access, or are given by an 8-bit instruction constant. Any address computation must be performed explicitly before the load or store instruction is executed. Since address computations are expressed directly, they are exposed for compiler optimizations as any other computations are.

External data accesses are overlapped with instruction execution. Processor performance is improved if instructions that follow loads do not immediately use externally referenced data. In this manner, the time required to perform the external access is overlapped with subsequent instruction execution. Because of hardware interlocks, this concurrency has no effect on the logical behavior of an executing program.

Addressing and Alignment (see Section 3.4.5)

External instructions and data are contained in one of four 32-bit address spaces:

1) Instruction/Data Memory.

2) Input/Output.

3) Coprocessor.

4) Instruction Read-Only Memory (Instruction ROM).

An address in the instruction/data memory address space may be treated as virtual or physical, as determined by the Current Processor Status Register. Address translation for data accesses is enabled separately from address translation for instruction accesses. A program in the Supervisor mode may temporarily disable address translation for individual loads and stores; this permits load-real and store-real operations.

Bits contained within load and store instructions distinguish between the instruction/data memory, input/output, and coprocessor address-spaces. Address translation also may determine whether an access is performed in the instruction/data memory or the input/output address space. The Current Processor Status register determines whether instruction accesses are directed to the instruction/data memory address space or to the instruction ROM address space.

The Am29000 does not support data accesses directly to the instruction ROM address space. However, this capability is possible as a system option.

All addresses are interpreted as byte addresses, although accesses are word-oriented. The number of a byte within a word is given by the two least-significant address bits. The number of a half-word within a word is given by the next-to-least-significant address bit.

Since only byte addressing is supported, it is possible that an address for the access of a word or half-word is not aligned to the desired word or half-word. For a word access, an unaligned address has a 1 in either or both of the two least-significant address bits. For a half-word access, an unaligned address has a 1 in the least-significant address bit. In many systems, address alignment can be ignored, with addresses truncated to access the word or half-word of interest. However, as a user option, the Am29000 creates a trap when a non-aligned access is attempted. The trap allows software emulation of non-aligned accesses.

In the Am29000, all instructions are 32 bits in length, and are aligned on word-address boundaries.

Byte and Half-Word Accesses (see Section 3.4.6)

The Am29000 supports the direct external access of bytes and half-words as an option. If this option is enabled, the Am29000 selects a byte or half-word within a word on a load, and aligns it to the low-order byte or half-word of a register. On a store, the low-order byte or half-word of a register is replicated in all byte or half-word positions, so that the external memory can easily write the required byte or half-word in memory. This option requires that the external memory system be able to write individual bytes and half-words within words.

To avoid the memory-system complexity caused by writing individual bytes and half-words, the Am29000 can perform byte and half-word accesses using software alone. The Am29000 can set a byte-position indicator in the ALU Status Register, as an option for load instructions, with the two least-significant bits of the address for the load. To load a byte or half-word, a word load is first performed. This load sets the byte-position indicator, and a subsequent instruction extracts the byte or half-word of interest from the accessed word. To store a byte or half-word, a load is also first performed; the byte or half-word of interest is inserted into the accessed word, and the resulting word then is stored. Even if the Am29000 is configured to perform byte and half-word accesses in hardware, this software-only technique operates correctly: this allows software to be upward-compatible from simpler systems to more complex systems.

2.1.5 INTERRUPTS AND TRAPS (see Section 3.5)

Normal program flow may be preempted by an interrupt or trap for which the processor is enabled. The effect on the processor is identical for interrupts and traps; the distinction is in the different

mechanisms by which interrupts and traps are enabled. It is intended that interrupts be used for suspending current program execution and causing another program to execute, while traps are used to report errors and exceptional conditions.

The interrupt and trap mechanism supports high-speed, temporary context switching and user-defined interrupt-processing mechanisms.

Temporary Context Switching

The basic interrupt/trap mechanism of the Am29000 supports temporary context switching. During the temporary context switch, the interrupted context is held in processor registers. The interrupt or trap handler can return immediately to this context.

Temporary context switching is useful for instruction emulation, floating-point operations, TLB reload routines, and so forth. Many of its features are similar to microprogram execution: processor context does not have to be saved; interrupts are disabled for the duration of the program; and all processor resources are accessible, even if the context that was interrupted is in the User mode. The associated routine may execute from instruction/data memory or instruction ROM.

User-Defined Interrupt Processing

Since the basic interrupt/trap mechanism for the Am29000 keeps the interrupted context in the processor, dynamically nested interrupts are not supported directly. The context in the processor must be saved before another interrupt or trap can be taken.

The interrupt or trap handler executing during a temporary context switch is not required to return to the interrupted context. This routine optionally may save the interrupted context, load a new one, and return to the new context.

The implementation of the saving and restoring of contexts is completely user-defined. Thus, the context save/restore mechanism used (e.g. interrupt stack, program status word area, etc.) and the amount of context saved may be tailored to the needs of the system.

Vector Area (see Section 3.5.4)

Interrupt and trap dispatching occurs through a relocatable Vector Area which accommodates as many as 256 interrupt and trap handling routines. Entries into the Vector Area are associated with various sources of interrupts and traps; some are pre-defined, while others are user-defined.

The Vector Area is either a table of vectors in data memory, where each vector points to the beginning of an interrupt or trap handler, or it is a segment of instruction/data memory (or instruction ROM) containing the actual routines. The latter configuration for the Vector Area yields better interrupt performance with the cost of additional memory.

2.1.6 MEMORY MANAGEMENT (see Section 3.6)

The Am29000 incorporates a Memory Management Unit (MMU) that accepts a 32-bit virtual byte-address and translates it to a 32-bit physical byte-address in a single cycle. The MMU is not dedicated to any particular address-translation architecture.

Address translation in the MMU is performed by a 64-entry Translation Look-Aside Buffer (TLB), an associative table which contains the most-recently used address translations for the processor. If the translation for a given address cannot be performed by the TLB, a TLB miss occurs, and causes a trap which allows the required translation to be placed into the TLB.

Processor hardware maintains information for each TLB line indicating which entry was least recently used; when a TLB miss occurs, this information is used to indicate the TLB entry to be replaced. Software is responsible for searching system page tables and modifying the indicated TLB entry as appropriate. This allows the page tables to be defined according to the system environment.

TLB entries are modified directly by processor instructions. A TLB entry consists of 64 bits and appears as two word-length TLB registers which may be inspected and modified by instructions.

TLB entries are tagged with a Task Identifier field, which allows the operating system to create a unique 32-bit virtual address space for each of 256 processes. In addition, TLB entries provide support for memory protection and user-defined control information.

2.1.7 COPROCESSOR PROGRAMMING (see Section 6.1)

The coprocessor interface for the Am29000 allows a program to communicate with an off-chip coprocessor for performing operations not supported by processor hardware directly.

The coprocessor interface allows the program to transfer operands and operation codes to the coprocessor, and then perform other operations while the coprocessor operation is in progress. The results of the operation are read from the coprocessor by a separate transfer. The processor may transfer multiple operands to the coprocessor without re-transferring operation codes or reading intermediate results. As many as 64 bits of information can be transferred to the coprocessor in a single cycle.

The Am29000 includes features that support the definition of the coprocessor as a system option. In this case, coprocessor operations are emulated by software when the coprocessor is not present in a system.

2.1.8 TIMER FACILITY (see Section 7.3.7)

The Timer Facility provides a counter for implementing a real-time clock or other software timing functions. This facility is comprised of two special-purpose registers: the Timer Counter Register, which decrements at a rate equal to the processor operating frequency, and the Timer Reload Register, which re-initializes the Timer Counter Register when it decrements to zero. The Timer Facility optionally may create an interrupt when the Timer Counter decrements to zero.

2.1.9 TRACE FACILITY (see Section 7.3.8)

The Trace Facility allows a debug program to emulate single-instruction stepping in a program under test. This facility allows a trap to be generated after the execution of any instruction in the program being tested.

Using the Trace Facility, the debug program can inspect and modify the state of the program at every instruction boundary. The Trace Facility is designed to work properly in the presence of normal system interrupts and traps.

2.2 HARDWARE OVERVIEW

This section briefly describes the operation of Am29000 hardware. It introduces the processor pipeline and the three major internal functional units: the Instruction Fetch Unit, the Execution Unit, and the Memory Management Unit. Finally, the processor's operational modes are described.

Figure 2-2 shows the Am29000 internal data-flow organization. The following sections refer to the various components on this data-flow diagram.

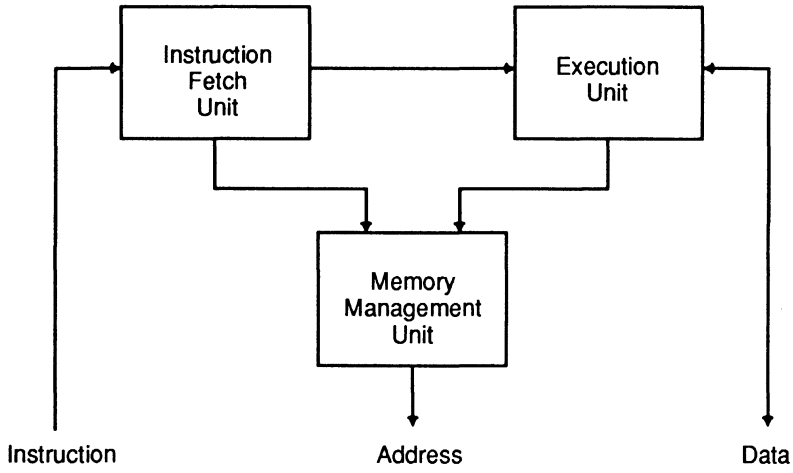


Figure 2-2. Am29000 Data Flow

2.2.1 FOUR-STAGE PIPELINE (see Section 4.1)

The Am29000 implements a four-stage pipeline for instruction execution. The four stages are: fetch, decode, execute, and write-back. The pipeline is organized so that the effective instruction-execution rate is as high as one instruction per cycle. Data forwarding and pipeline interlocks are handled by processor hardware.

2.2.2 INSTRUCTION FETCH UNIT (see Section 4.2)

The Instruction Fetch Unit fetches instructions, and supplies instructions to other functional units. It incorporates the Instruction Prefetch Buffer, the Branch Target Cache, and the Program Counter Unit. All components of the Instruction Fetch Unit operate during the fetch stage of the processor pipeline.

Instruction Prefetch Buffer (see Section 4.2.1)

Most instructions executed by the Am29000 are fetched from external instruction/data memory. The processor prefetches instructions so that they are requested at least four cycles before they are required for execution.

Prefetched instructions are stored in a four-word Instruction Prefetch Buffer while awaiting execution. An instruction-prefetch request occurs whenever there is a free location in this buffer (if the processor is otherwise enabled to fetch instructions). When a non-sequential instruction fetch occurs, prefetching is terminated, and then restarted for the new instruction stream.

Instruction prefetching de-couples the instruction-fetch rate from the instruction-access latency. For example, an instruction may be transferred to the processor two cycles after it is requested. However, as long as instructions are supplied to the processor at an average rate of one instruction per cycle, this latency has no effect on the instruction-execution rate.

Branch Target Cache (see Section 4.2.2)

The Am29000 incorporates a Branch Target Cache which contains as many as 128 instructions. The Branch Target Cache is a two-way, set-associative cache containing the first four target instructions of a number of recently taken branches. Each of the two sets in the Branch Target Cache contains 64 instructions, and the 64 instructions are further divided into 16 blocks of four instructions each.

The purpose of the Branch Target Cache is to provide instructions for the beginning of a non-sequential instruction-fetch sequence. This keeps the instruction pipeline full until the processor can establish a new instruction-prefetch stream from the external instruction/data memory.

The processor is organized so that branch instructions can execute in a single cycle if the target instruction sequence is present in the Branch Target Cache.

Program Counter Unit (see Section 4.2.4)

The Program Counter Unit creates and sequences addresses of instructions as they are executed by the processor.

2.2.3 EXECUTION UNIT (see Section 4.3)

The Execution Unit executes instructions. It incorporates the Register File, the Address Unit, the Arithmetic/Logic Unit, the Field Shift Unit, and the Prioritizer. The Register File and Address Unit operate during the decode stage of the pipeline. The Arithmetic/Logic Unit, Field Shift Unit, and Prioritizer operate during the execute stage of the pipeline. The Register File operates during the write-back stage.

Register File (see Section 4.3.1)

The general-purpose registers are implemented by a 192-location Register File. The Register File can perform two read accesses and one write access in a single cycle. Normally, two read accesses

are performed during the decode-pipeline stage to fetch operands required by the instruction being decoded. The write access during the same cycle completes the write-back stage of a previously executed instruction.

Addressing logic associated with the Register File distinguishes between the global and local general-purpose registers, and it performs the Stack-Pointer addressing for the local registers. Register File addressing functions are performed during the decode stage.

Address Unit (see Section 4.3.2)

The Address Unit evaluates addresses for branches, loads, and stores. It also assembles instruction-immediate data and computes addresses for load-multiple and store-multiple sequences.

Arithmetic/Logic Unit (see Section 4.3.3)

The ALU performs all logical, compare, and arithmetic operations (including multiply step and divide step).

Field Shift Unit (see Section 4.3.4)

The Field Shift Unit performs N-bit shifts. The Field Shift Unit also performs byte and half-word extract and insert operations, and it extracts words from double-words.

Prioritizer (see Section 4.3.5)

The Prioritizer provides a count of the number of leading zero bits in a 32-bit word; this is useful for performing floating-point normalization, for example. It can also be used to implement prioritization in a multi-level interrupt handler.

2.2.4 MEMORY MANAGEMENT UNIT (see Section 4.4)

The Memory Management Unit (MMU) performs address translation and memory-protection functions for all branches, loads, and stores. The MMU operates during the execute stage of the pipeline, so the physical address that it generates is available at the beginning of the write-back stage.

All addresses for external accesses are physical addresses. MMU operation is pipelined with external accesses, so that an address translation can occur while a previous access completes.

Address translation is not performed for the addresses associated with instruction prefetching. Instead, these addresses are generated by an instruction prefetch pointer which is incremented by the processor. Address translation is performed only at the beginning of the prefetch sequence (as the result of a branch instruction), and when the prefetch pointer crosses a potential virtual-page boundary.

2.2.5 PROCESSOR MODES

The Am29000 operates in several different modes to accomplish various processor and system functions. All modes except for Pipeline Hold (see below) are under direct control of instructions

and/or processor control inputs. The Pipeline Hold mode normally is determined by the relative timing between the processor and its external system for certain types of operations. The processor provides an external indication of its operational mode.

Executing

When the processor is in the Executing mode, it fetches and executes instructions as described in this manual. External accesses occur as required.

Wait (see Section 3.5.3)

When the processor is in the Wait mode, it does not execute instructions, and performs no external accesses. The Wait mode is controlled by the Current Processor Status Register. The processor leaves this mode when an interrupt or trap for which it is enabled occurs, or when a reset occurs.

Pipeline Hold (see Section 4.5)

Under certain conditions, processor pipelining might cause non-sequential instruction execution or timing-dependent results of execution. For example, the processor might attempt to execute an instruction that has not been fetched from instruction/data memory.

For such cases, pipeline-interlock hardware detects the anomalous condition and suspends processor execution until execution can proceed properly. While execution is suspended by the interlock hardware, the processor is in the Pipeline Hold mode. The processor resumes execution when the pipeline-interlock hardware determines that it is correct to do so.

Halt (see Section 5.3.3)

The Halt mode is provided so that the processor may be placed under the control of the ADAPT29K or other hardware-development system (see Section 2.3.2) for the purposes of hardware and software debug. The processor enters the Halt mode as the result of instruction execution, or as the result of external controls. In the Halt mode, the processor neither fetches nor executes instructions.

Step (see Section 5.3.3)

The Step mode allows the ADAPT29K or other hardware-development system to step through processor pipeline operation on a stage-by-stage basis. The Step mode nearly is identical to the Halt mode, except that it enables the processor to enter the Executing mode while the pipeline advances by one stage.

Load Test Instruction (see Section 5.3.3)

The Load Test Instruction mode permits the ADAPT29K or other hardware-development system to access data contained in the processor or system. This is accomplished by allowing the ADAPT29K to supply the processor with instructions, instead of having the processor fetch instructions from instruction/data memory. The Load Test Instruction mode is defined so that, once the processor has

completed the execution of instructions provided by the ADAPT29K, it may resume the execution of its normal instruction sequence.

Test (see Section 5.3.5)

The Test mode facilitates testing of hardware associated with the processor by disabling processor outputs so that they may be driven directly by test hardware. The Test mode also allows the addition of a second processor to a system, to monitor the outputs of the first and signal detected errors.

Reset (see Section 3.8 and Section 5.5)

The Reset mode provides initialization of certain processor registers and control state. This is used for power-on reset, for eliminating unrecoverable error conditions, and for supporting certain hardware-debug functions.

2.3 SYSTEM INTERFACE OVERVIEW

This section briefly describes the features of the Am29000 that allow it to be connected to other system components.

The two major interfaces of the Am29000, introduced in this section, are the channel and the Test/Development interface. The other topics briefly described here are clock generation, master/slave checking, and coprocessor attachment.

Section 5.1 contains a complete pin description of the Am29000. Appendix A contains timing diagrams and related information.

2.3.1 CHANNEL (see Section 5.2)

The Am29000 channel consists of the following 32-bit buses and related controls:

1) An Instruction Bus, which transfers instructions into the processor.

2) A Data Bus, which transfers data to and from the processor.

3) An Address Bus, which provides addresses for both instruction and data accesses. The Address Bus also is used to transfer data to a coprocessor.

The channel performs accesses and data transfers to all external devices and memories, including instruction/data memories, instruction caches, instruction read-only memories, data caches, input/output devices, bus converters, and coprocessors.

The channel defines three different access protocols: simple, pipelined, and burst-mode. For simple accesses, the Am29000 holds the address valid throughout the entire access. This is appropriate for high-speed devices that can complete an access in one cycle, and for low-cost devices that are accessed infrequently (such as read-only memories containing initialization routines). Pipelined and burst-mode accesses provide high performance with other types of devices and memories.

For pipelined accesses, the address transfer is decoupled from the corresponding data or instruction transfer. After transmitting an address for a request, the processor may transmit one more address before receiving the reply to the first request. This allows address transfer and decoding to be overlapped with another access.

On the other hand, burst-mode accesses eliminate the address-transfer cycle completely. Burst-mode accesses are defined so that once an address is transferred for a given access, subsequent accesses to sequentially increasing addresses may occur without re-transfer of the address. The burst may be terminated at any time by either the processor or responding device.

The Am29000 determines whether an access is simple, pipelined or burst-mode on a transfer-by-transfer (i.e., generally device-by-device) basis. However, an access that begins as a simple access may be converted to a pipelined or burst-mode access at any time during the transfer. This relaxes the timing constraints on the channel-protocol implementation, since addressed devices do not have to respond immediately to a pipelined or burst-mode request.

Except for the shared Address Bus, the channel maintains a strict division between instruction and data accesses. In the most common situation, the system supplies the processor with instructions using burst-mode accesses, with instruction addresses transmitted to the system only when a branch occurs. Data accesses can occur simultaneously without interfering with instruction transfer.

The Am29000 contains arbitration logic to support other masters on the channel. A single external master can arbitrate directly for the channel, while multiple masters may arbitrate using a daisy chain or other method that requires no additional arbitration logic. However, to increase arbitration performance in a multiple-master configuration, an external channel arbiter should be used. This arbiter works in conjunction with the processor's arbitration logic.

2.3.2 TEST/DEVELOPMENT INTERFACE (see Section 5.3)

The Am29000 supports the attachment of the ADAPT29K or other hardware-development system. This attachment is made directly to the processor in the system under development, without the removal of the processor from the system. The Test/Development Interface makes it possible for the hardware-development system to gain control over the Am29000, and inspect and modify its internal state (e.g., general-purpose register contents, TLB entries, etc.). In addition, the Am29000 may be used to access other system devices and memories on behalf of the hardware-development system.

The Test/Development Interface is comprised of controls and status signals provided on the Am29000, as well as the Instruction and Data buses. The Halt, Step, Reset, and Load Test Instruction modes allow the hardware-development system to control the operation of the Am29000. The hardware-development system may supply the processor with instructions on the Instruction Bus using the Load Test Instruction mode. Internal processor state can be inspected and modified via the Data Bus.

2.3.3 CLOCKS (see Section 5.7)

The Am29000 generates and distributes a system clock at its operating frequency. This clock is specially designed to reduce skews between the system clock and the processor's internal clocks.

The internal clock-generation circuitry requires a single-phase oscillator signal at twice the processor operating frequency.

For systems in which processor-generated clocks are not appropriate, the Am29000 also can accept a clock from an external clock generator.

The processor decides between these two clocking arrangements based on whether the power supply to the clock-output driver (PWRCLK) is tied to +5 volts or to GROUND.

2.3.4 MASTER/SLAVE OPERATION (see Section 5.8)

Each Am29000 output has associated logic that compares the signal on the output with the signal that the processor is providing internally to the output driver. The processor signals situations where the output of any enabled driver does not agree with its input.

For a single processor, the output comparison detects short circuits in output signals, but does not detect open circuits. It is possible to connect a second processor in parallel with the first, where the second processor has its outputs disabled due to the Test mode. The second processor detects open-circuit signals, as well as provides a check of the outputs of the first processor.

2.3.5 COPROCESSOR ATTACHMENT (see Section 6.2)

A coprocessor for the Am29000 attaches directly to the processor channel. However, this attachment has features that are different than those of other channel devices. The coprocessor interface is designed to support a high operand-transfer rate and to support the overlap of coprocessor operations with other processor operations, including other external accesses.

The coprocessor is assigned a special address space on the channel. This permits the transfer of operands and other information on the Address Bus without interfering with normal addressing functions. Since both the Address Bus and Data Bus are used for data transfer, the Am29000 can transfer 64 bits of information to the coprocessor in one cycle.

CHAPTER 3

PROGRAMMER REFERENCE

This chapter contains a formal description of the Am29000 architecture. It concentrates on the features of the Am29000 and their logical behavior. Chapter 7 discusses the use of some of these features.

3.1 PROGRAM MODES

All system-protection features of the Am29000 are based on two mutually exclusive program modes: the Supervisor mode, and the User mode.

3.1.1 SUPERVISOR MODE

The processor is in the Supervisor mode whenever the Supervisor Mode (SM) bit of the Current Processor Status Register (see Section 3.2.2) is 1. In this mode, executing programs have access to all processor resources.

During the address cycle of a channel request, the Supervisor mode is indicated by the SUP/*US output being High.

3.1.2 USER MODE

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

1) An attempted access of any TLB entry (see Section 3.2.3).

2) An attempted access of any general-purpose register for which a bit in the Register Bank Protect Register is 1 (see Section 3.2.1).

3) An attempted execution of a load or store instruction for which the PA bit is 1, or for which the UA bit is 1 (see Section 3.4.4). (The attempted execution of a translated load or store for which the AS bit is 1 also causes a Protection Violation trap. However, this trap occurs regardless of whether or not the processor is in the User mode.)

4) An attempted execution of one of the following instructions: Interrupt Return, Interrupt Return and Invalidate, Invalidate, or Halt. However, a hardware-development system such as the ADAPT29K can disable protection checking for the Halt instruction, so that this instruction may be used to implement instruction breakpoints in User-mode programs (see Section 5.3.3).

5) An attempted access of one of the following registers: Vector Area Base Address, Old Processor Status, Current Processor Status, Configuration, Channel Address, Channel Data,

Channel Control, Register Bank Protect, Timer Counter, Timer Reload, Program Counter 0, Program Counter 1, Program Counter 2, MMU Configuration, or LRU Recommendation (see Section 3.2.2).

6) An attempted execution of an assert or Emulate instruction which specifies a vector number between 0 and 63, inclusive (see Section 3.5.4).

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

3.2 VISIBLE REGISTERS

The Am29000 has three classes of registers that are accessible by instructions. These are general-purpose registers, special-purpose registers, and Translation Look-Aside Buffer (TLB) registers. Any operation available in the Am29000 can be performed on the general-purpose registers, while special-purpose registers and TLB registers are accessed only by explicit data movement to or from general-purpose registers. Various protection mechanisms prevent the access of some of these registers by User-mode programs.

A summary of the information in this section appears in Appendix B.

3.2.1 GENERAL-PURPOSE REGISTERS

The Am29000 incorporates 192 general-purpose registers. The organization of the general-purpose registers is diagrammed in Figure 3-1.

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

1) 32-bit data 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.

Am29000 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.

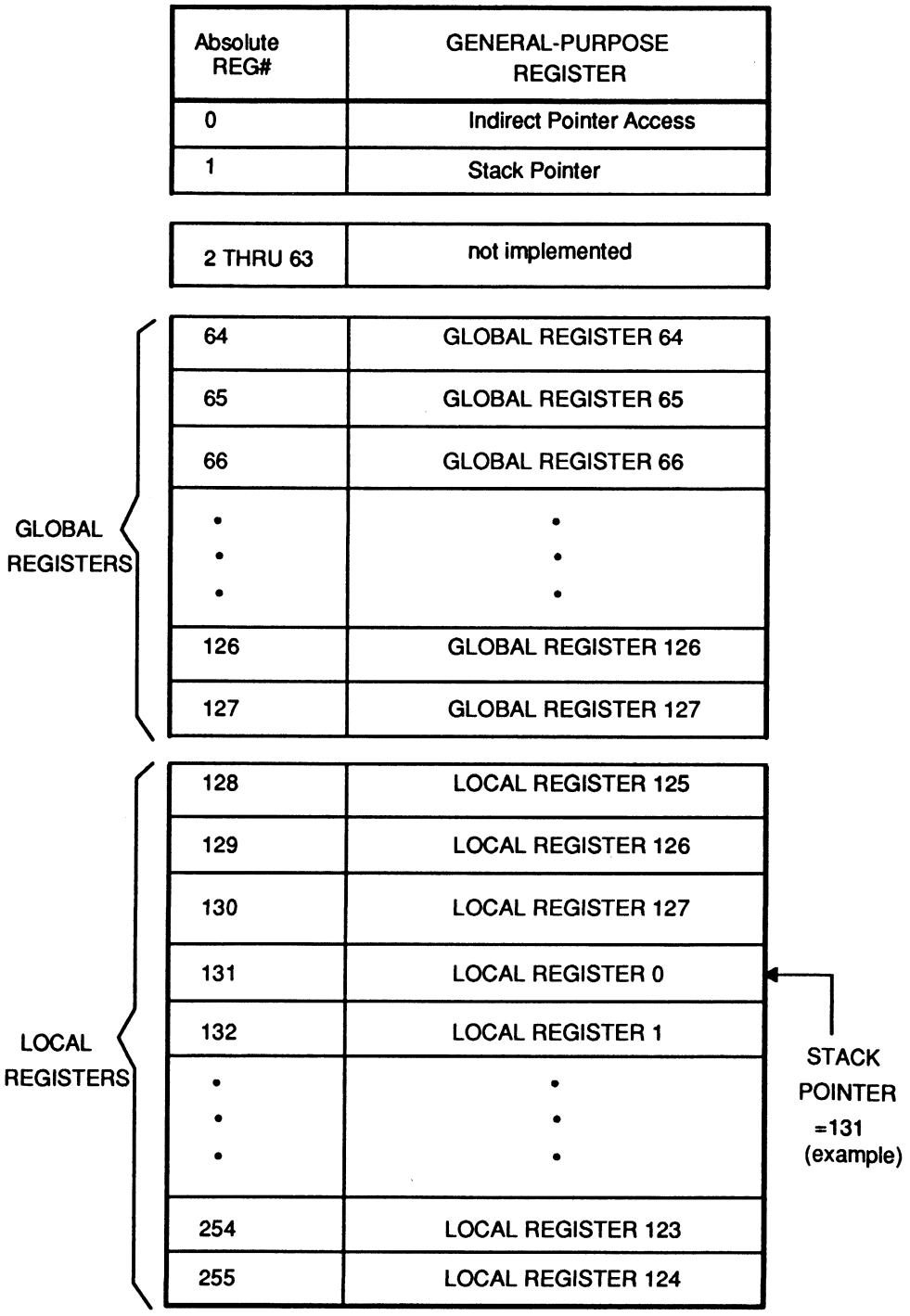


Figure 3-1. General-Purpose Register Organization

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.

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.

Global registers 2 through 63 are unimplemented. 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 below).

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; 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 as explained below.

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 to be used in local-register addressing. This shadow copy is set only with the results of Arithmetic and Logical instuctions. 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 7.4.3.

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 Section 7.1).

Register Banking

For the purpose of access restriction, the general-purpose registers are divided into register banks. Register banks consist of 16 registers (except for Bank 0, which contains unimplemented registers 2 through 15), and are partitioned according to absolute-register numbers, as shown in Figure 3-2.

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

Indirect Accesses

Specification of 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, Floating-Point, MULTIPLY, MULTM, MULTIPLU, MULTMU, DIVIDE, DIVIDU, SETIP, and EMULATE instructions.

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 3-2. Register Bank Organization

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 7.4.3.

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.

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

3.2.2 SPECIAL-PURPOSE REGISTERS

The Am29000 contains 27 special-purpose registers. The organization of the special-purpose registers is shown in Figure 3-3.

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.

An attempted read of an unimplemented special-purpose register yields an unpredictable value. An attempted write of an unimplemented special-purpose register has no effect; however, this should be avoided, because of upward-compatibility considerations.

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

Protected special-purpose registers numbered 0–127 are accessible only by programs executing in the Supervisor mode. An attempted read or write of a protected 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 either the User Mode or the Supervisor Mode. These register numbers identify virtual registers in the floating-point architecture, and any attempted access causes a Protection Violation trap.

The Floating-Point Environment Register, Integer Environment Register, Floating-Point Status Register, and Exception Opcode Register are not implemented in processor hardware. These registers are implemented via a virtual floating-point interface provided on the Am29000.

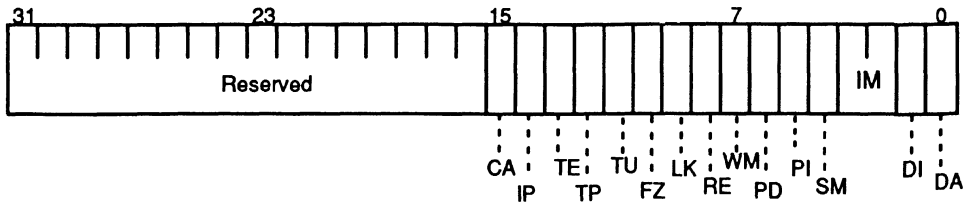


Figure 3-5. Current Processor Status Register

Bits 31-16 : reserved.

Bit 15 : Coprocessor Active (CA)—The CA bit is set and reset under the control of load and store instructions that transfer information to and from a coprocessor. This bit indicates that the coprocessor is performing an operation at the time that an interrupt or trap is taken. This notifies the interrupt or trap handler that the coprocessor contains state information to be preserved. Note that this notification occurs because the CA bit of the Old Processor Status is 1 in this case, not because of the value of the CA bit of the Current Processor Status.

Bit 14 : Interrupt Pending (IP)—This bit allows software to detect the presence of external interrupts while they are disabled. The IP bit is set if one or more of the external signals *INTR0—*INTR3 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 subsequently are 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 7.3.8 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 and coprocessor transfers. 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.

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 *LOCK external signal. If the LK bit is 1, the *LOCK signal is active. If the LK bit is 0, the *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 : ROM Enable (RE)—The RE bit enables instruction fetching from external instruction read-only memory (ROM). When this bit is 1, the IREQT signal directs all instruction requests to ROM. Instructions that are fetched from ROM are subject to capture and re-use by the Branch Target Cache when it is enabled; the Branch Target Cache distinguishes between instructions from ROM and those from non-ROM storage. When this bit is 0, off-chip requests for instructions are directed to instruction/data memory.

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 Reset mode.

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 3.1 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 channel 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 by the following table:

IM Value	Result
00	*INTR0 enabled
01	*INTR(1:0) enabled
10	*INTR(2:0) enabled
11	*INTR(3:0) enabled

Bit 1 : Disable Interrupts (DI)—The DI bit prevents the processor from being interrupted by external interrupt requests *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 will be taken. When this bit is 0, the processor will take any interrupt enabled by the IM field, unless the DA bit is 1 (see below).

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 *WARN, Instruction Access Exception, Data Access Exception, and Coprocessor Exception traps. When this bit is 0, all traps will be taken, and interrupts will be taken if otherwise enabled.

Configuration (CFG, Register 3)

This protected special-purpose register (see Figure 3-6) controls certain processor and system options. Most fields normally are modified only during system initialization. The Configuration Register is defined as follows:

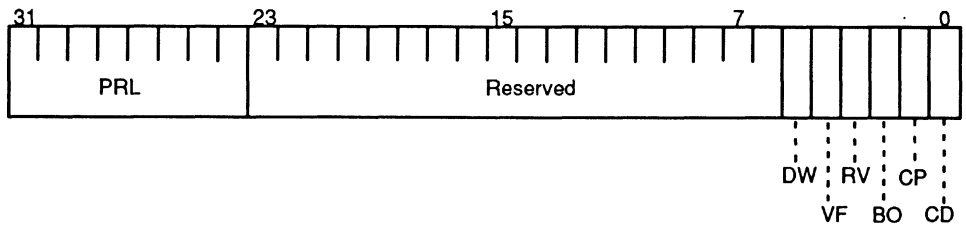


Figure 3-6. Configuration Register

Bits 31-24: Processor Release Level (PRL)—The PRL field is an 8-bit, read-only identification number which specifies the processor version.

Bits 23-6 : reserved.

Bit 5 : Data Width Enable (DW)—The DW bit enables and disables byte and half-word external accesses. If the DW bit is 0, byte and half-word accesses are not performed in hardware, and these accesses must be emulated by software. If the DW bit is 1, byte and half-word accesses are performed by hardware: this requires that external devices and memories be able to write individual bytes and half-words within a word.

Bit 4 : Vector Fetch (VF)—The VF bit determines the structure of the interrupt/trap Vector Area. If this bit is 1, the Vector Area is defined as a block of 256 vectors which specify the beginning

addresses of the interrupt and trap handling routines. If the VF bit is 0, the Vector Area is a segment of 256 64-instruction blocks that contain the actual routines.

Bit 3 : ROM Vector Area (RV)—If the VF bit is 0, the RV bit specifies whether the Vector Area is contained in instruction memory (RV = 0) or instruction read-only memory (RV = 1). The value of the RV bit is irrelevant if the VF bit is 1.

Bit 2 : Byte Order (BO)—The BO bit determines the ordering of bytes and half-words within words. If the BO bit is 0, bytes and half-words are numbered left-to-right within a word. If the BO bit is 1, bytes and half-words are numbered right-to-left. Section 3.4.5 describes the interpretation of the BO bit in more detail.

Bit 1 : Coprocessor Present (CP)—The CP bit indicates the presence of a coprocessor that may be used by the processor. If this bit is 1, it enables the execution of load and store instructions that have a Coprocessor Enable (CE) bit of 1. If the CP bit is 0 and the processor attempts to execute a load or store instruction with a CE bit of 1, a Coprocessor Not Present trap occurs. This feature may be used to emulate coprocessor operations as well as to protect the state of a coprocessor shared between multiple processes.

Bit 0 : Branch Target Cache Disable (CD)—The CD bit determines whether or not the Branch Target Cache is used for non-sequential instruction references. When this bit is 1, all instruction references are directed to external instruction memory or instruction ROM, and the Branch Target Cache is not used. When this bit is 0, the targets of non-sequential instruction fetches are stored in the Branch Target Cache and re-used as described in Section 4.2.2. The value of the CD bit does not take effect until the execution of the next branch instruction.

Channel Address (CHA, Register 4)

This protected special-purpose register (Figure 3-7) is used to report exceptions during external accesses or coprocessor transfers. 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 restarting of external accesses is described in Section 7.3.5.

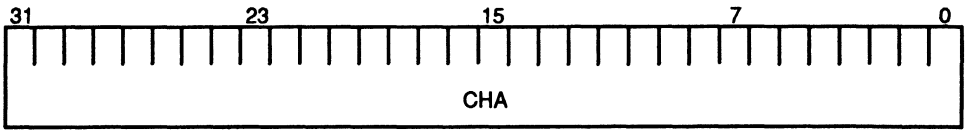


Figure 3-7. Channel Address Register

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 channel transaction (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. For transfers to the coprocessor, the CHA field contains data transferred to the coprocessor.

Channel Data (CHD, Register 5)

This protected special-purpose register (Figure 3-8) is used to report exceptions during external accesses or coprocessor transfers. It is also 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 restarting of external accesses is described in Section 7.3.5.

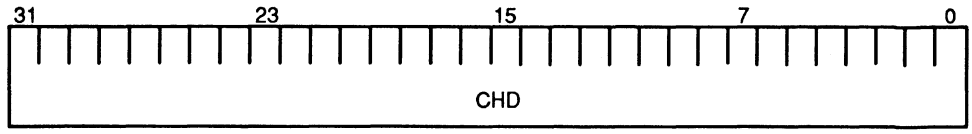


Figure 3-8. Channel Data Register

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.

Bits 31-0 : Channel Data (CHD)—This field contains the data (if any) associated with the current channel transaction (if the FZ bit of the Current Processor Status Register is 0). If the current channel transaction is not a store or a transfer to the coprocessor, the value of this field is irrelevant.

Channel Control (CHC, Register 6)

This protected special-purpose register (Figure 3-9) is used to report exceptions during external accesses or coprocessor transfers. 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 restarting of external accesses is described in Section 7.3.5.

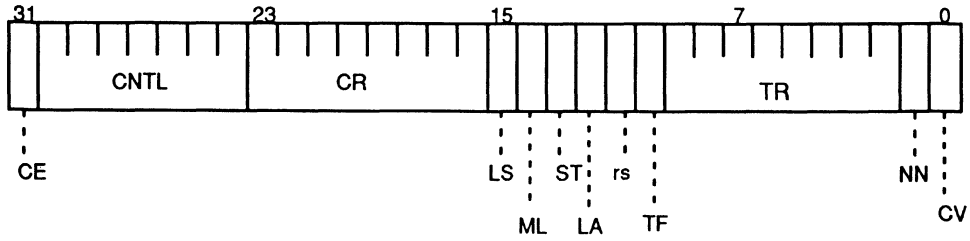


Figure 3-9. Channel Control Register

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 which started the current channel transaction (see Section 3.4.4 and Section 6.1.2).

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. If the fault or interrupt occurs on the last transaction, the CR field contains a value of 0, and the ML bit is 1 (see below).

Bit 15 : Load/Store (LS)—The LS bit is 0 if the channel transaction is a store operation, and 1 if it is a load operation.

Bit 14 : Multiple Operation (ML)—The ML bit is 1 if the current channel transaction 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 channel transaction is for a Load and Set instruction; otherwise it is 0.

Bit 12 : Lock Active (LA)—The LA bit is 1 if the current channel transaction 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 channel transaction did not complete due to some exceptional circumstance. This bit is set only for exceptions reported via the *DERR input, and it causes a Data Access Exception or Coprocessor Exception trap to occur (depending on the value of the CE bit) when it is 1.

The TF bit allows the proper sequencing of externally reported errors that get preempted by higher-priority traps (see Section 3.5.7). 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 data operand for the current transaction (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 uncompleted load operation, the data requested is not needed. This situation arises when a load instruction is overlapped with an instruction which 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.

Register Bank Protect (RBP, Register 7)

This protected special-purpose register (Figure 3-10) protects banks of general-purpose registers from User-mode program accesses.

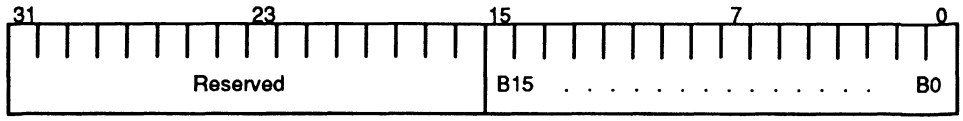


Figure 3-10. Register Bank Protect Register

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 3-2 of Section 3.2.1.

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).

When a protection bit is 1, the corresponding bank is protected from access by programs executing in the User mode. A Protection Violation trap occurs when a User-mode program attempts to access (either read or write) a register in a protected bank. When a bit in this register is 0, the corresponding bank is available to programs executing in the User mode.

Supervisor-mode programs are not affected by the Register Bank Protect Register.

Register protection is based on absolute-register numbers. For local registers, the protection checking is performed after the Stack-Pointer addition is performed.

Timer Counter (TMC, Register 8)

This protected special-purpose register (Figure 3-11) contains the counter for the Timer Facility.

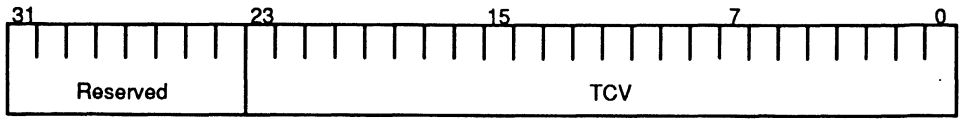


Figure 3-11. Timer Counter Register

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.

Timer Reload (TMR, Register 9)

This protected special-purpose register (Figure 3-12) maintains synchronization of the Timer Counter Register, enables Timer interrupts, and maintains Timer Facility status information.

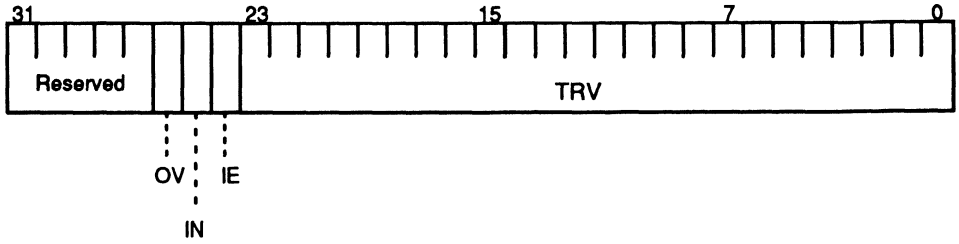


Figure 3-12. Timer Reload Register

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 is 1 (see below) 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.

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.

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 Timer interrupts 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.

Program Counter 0 (PC0, Register 10)

This protected special-purpose register (Figure 3-13) is used, on an interrupt return, to restart the instruction which was in the decode stage when the original interrupt or trap was taken.

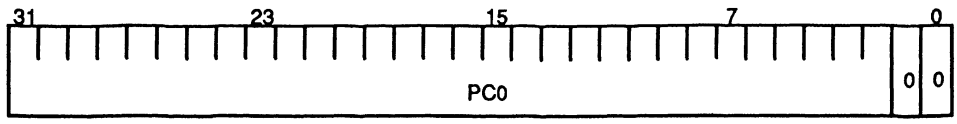


Figure 3-13. Program Counter 0 Register

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.

Program Counter 1 (PC1, Register 11)

This protected special-purpose register (Figure 3-14) is used, on an interrupt return, to restart the instruction that was in the execute stage when the original interrupt or trap was taken.

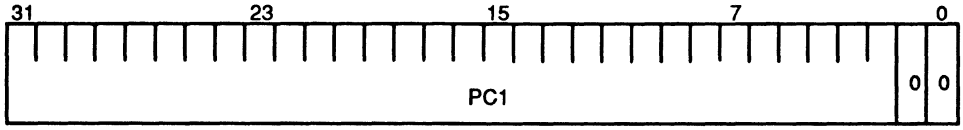


Figure 3-14. Program Counter 1 Register

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.

Program Counter 2 (PC2, Register 12)

This protected special-purpose register (Figure 3-15) reports the address of certain instructions causing traps.

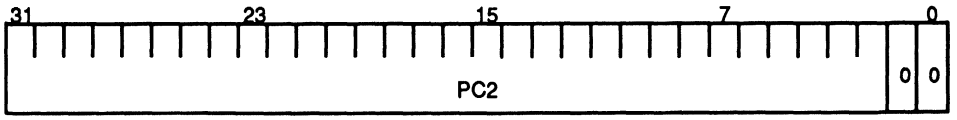


Figure 3-15. Program Counter 2 Register

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, as described in Section 3.5.8, 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.

MMU Configuration (MMU, Register 13)

This protected special-purpose register (Figure 3-16) specifies parameters associated with the Memory Management Unit (MMU).

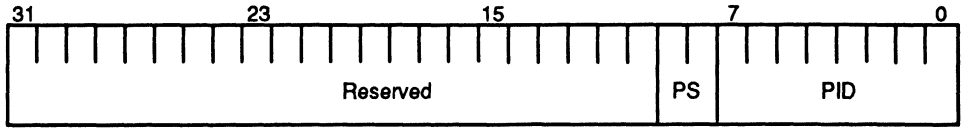


Figure 3-16. MMU Configuration Register

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 Section 3.6.2. The PS field has a delayed effect on address translation (see Section 7.3.3). 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
00	1 Kbyte
01	2 Kbyte
10	4 Kbyte
11	8 Kbyte

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.

LRU Recommendation (LRU, Register 14)

This protected special-purpose register (Figure 3-17) assists Translation Look-Aside Buffer (TLB) reloading by indicating the least-recently used TLB entry in the required replacement line.

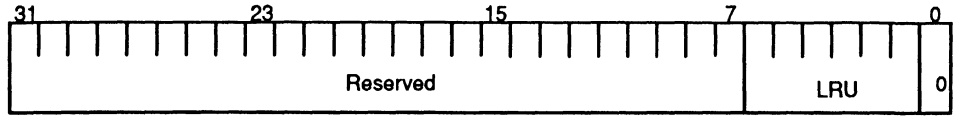


Figure 3-17. LRU Recommendation Register

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.

Indirect Pointer C (IPC, Register 128)

This unprotected special-purpose register (Figure 3-18) provides the RC-operand register number (see Section 8.3) when an instruction RC field has the value zero (i.e., when Global Register 0 is specified).

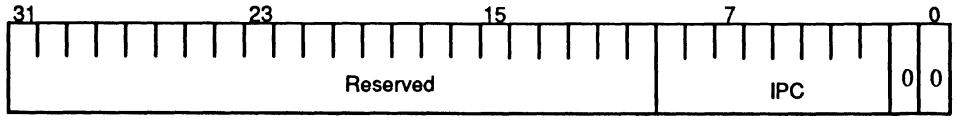


Figure 3-18. Indirect Pointer C Register

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.

Indirect Pointer A (IPA, Register 129)

This unprotected special-purpose register (Figure 3-19) provides the RA-operand register number (see Section 8.3) when an instruction RA field has the value zero (i.e., when Global Register 0 is specified).

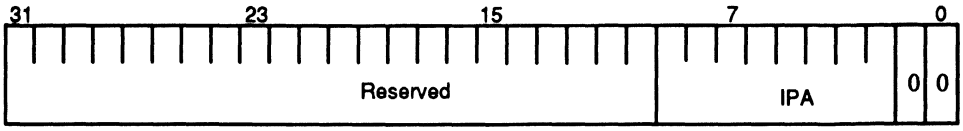


Figure 3-19. Indirect Pointer A Register

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.

Indirect Pointer B (IPB, Register 130)

This unprotected special-purpose register (Figure 3-20) provides the RB-operand register number (see Section 8.3) when an instruction RB field has the value zero (i.e., when Global Register 0 is specified).

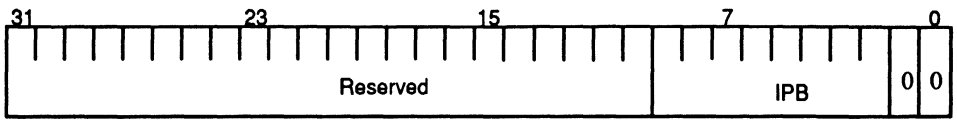


Figure 3-20. Indirect Pointer B Register

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.

Q (Q, Register 131)

The Q Register is an unprotected special-purpose register (Figure 3-21).

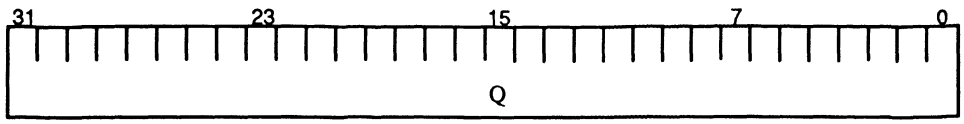


Figure 3-21. Q Register

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.

ALU Status (ALU, Register 132)

This unprotected special-purpose register (Figure 3-22) holds information about the outcome of Arithmetic/Logic Unit (ALU) operations as well as control for certain operations performed by the Execution Unit.

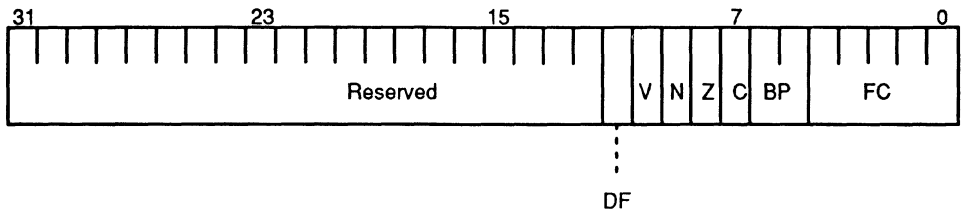


Figure 3-22. ALU Status Register

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, then 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 exact 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 exact 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 either with the two least-significant bits of the address (if the DW bit of the Configuration Register is 0) or with the complement of the Byte Order bit of the Configuration Register (if DW is 1).

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.

Byte Pointer (BP, Register 133)

This unprotected special-purpose register (Figure 3-23) provides an alternate access to the BP field in the ALU Status Register.

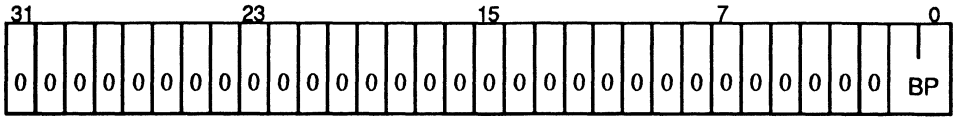


Figure 3-23. Byte Pointer

Bits 31-2 : Zeros.

Bits 1-0 : Byte Pointer (BP)—This field allows a program to change the BP field without affecting other fields in the ALU Status Register.

Funnel Shift Count (FC, Register 134)

This unprotected special-purpose register (Figure 3-24) provides an alternate access to the FC field in the ALU Status Register.

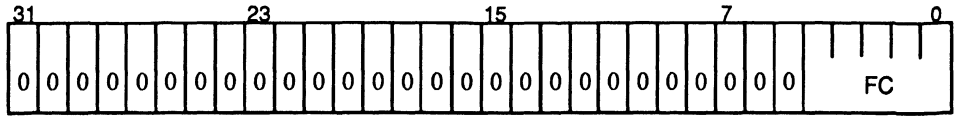


Figure 3-24. Funnel Shift Count

Bits 31-5 : Zeros.

Bits 4-0 : Funnel Shift Count (FC)—This field allows a program to change the FC field without affecting other fields in the ALU Status Register.

Load/Store Count Remaining (CR, Register 135)

This unprotected special-purpose register (Figure 3-25) provides alternate access to the CR field in the Channel Control Register.

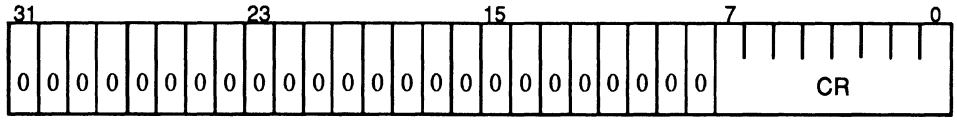


Figure 3-25. Load/Store Count Remaining

Bits 31-8 : Zeros.

Bits 7-0 : Load/Store Count Remaining (CR)—This field allows a program to change the CR field 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.

Floating-Point Environment (FPE, Register 160)

This unprotected special-purpose register (Figure 3-26) contains control bits that affect the execution of floating-point operations.

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 -∞
10	Round to +∞
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.

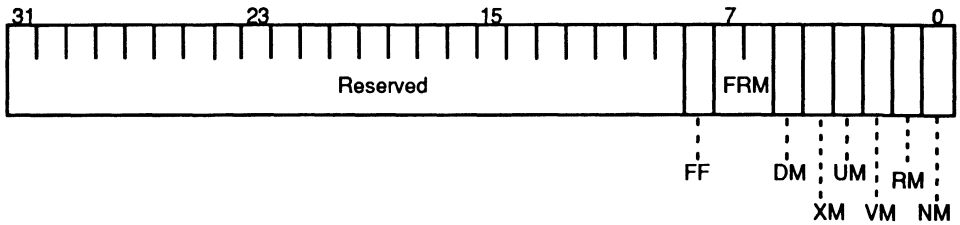


Figure 3-26. Floating-Point Environment

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.

Integer Environment (INTE, Register 161)

This unprotected special-purpose register (Figure 3-27) contains control bits which affect the execution of integer operations.

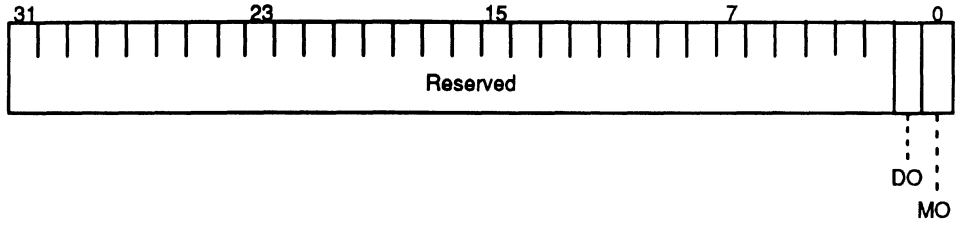


Figure 3-27. Integer Environment

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 DO bit is 1, an Out of Range trap does not occur for overflow during integer multiply operations.

Floating-Point Status (FPS, Register 162)

This unprotected special-purpose register (Figure 3-28) contains status bits indicating the outcome of floating-point operations. The bits of the Floating-Point Status Register are divided into two groups of status bits. The bits in each group correspond to the causes of Floating-Point Exception traps that are enabled and disabled by bits 5–0 of the Floating-Point Environment Register.

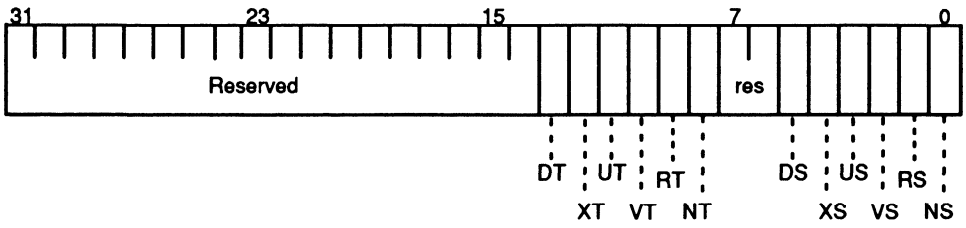


Figure 3-28. Floating-Point Status

The first group of status bits (bits 13–8) are trap status bits that report the cause of a Floating-Point Exception trap. The trap status bits are set only when a Floating-Point Exception trap occurs, and indicate all conditions that apply to the trapping operation. All other operations leave the status bits unchanged. A trap status bit is set regardless of the state of the corresponding mask bit of the Floating-Point Environment Register, except that at least one of the mask bits must be 0 for the trap to occur. When a Floating-Point Exception trap occurs, all trap status bits not relevant to the trapping operation are reset.

The second group of status bits (bits 5–0) are sticky status bits that, once set, remain set until explicitly cleared by a Move to Special Register (MTSR) or Move to Special Register Immediate (MTSRIM) instruction. A sticky status bit is set only when the a floating-point exception is detected and the corresponding mask bit of the Floating-Point Environment Register is 1. That is, the sticky status bit is set only if the corresponding cause of a Floating-Point Exception trap is disabled. Normally, this means that sticky status bits are not set when a Floating-Point Exception trap is taken. However, if multiple exceptions are detected, a sticky status bit corresponding to a masked exception may still be set if a Floating-Point Exception trap occurs for an unmasked exception.

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 either one or more input operands to the associated floating-point

operation is a reserved value or the result of this 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.

Exception Opcode (EXOP, Register 164)

This unprotected special-purpose register (Figure 3-29) reports the operation code (opcode) of an instruction causing a trap. It is provided primarily for recovery from floating-point exceptions, but reports the opcode of any trapping instruction.

Bits 31-8 : reserved.

Bits 7-0 : Instruction Opcode (IOP)—This field captures the opcode of an instruction causing a trap as a result of instruction execution; the opcode is captured as the instruction enters the write-back stage of the processor pipeline. Instructions that do not trap as a consequence of execution do not modify the IOP field.

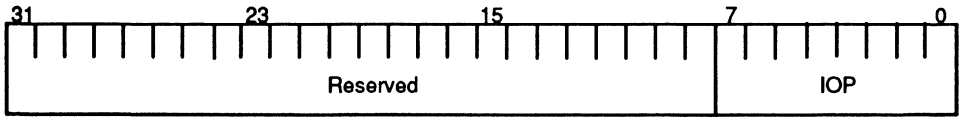


Figure 3-29. Exception Opcode

3.2.3 TLB REGISTERS

The Am29000 contains 128 Translation Look-Aside Buffer (TLB) registers. The organization of the TLB registers is shown in Figure 3-30.

TLB Reg#	TLB Set 0
0	TLB Entry Line 0 Word 0
1	TLB Entry Line 0 Word 1
2	TLB Entry Line 1 Word 0
3	TLB Entry Line 1 Word 1
•	•
•	•
•	•
62	TLB Entry Line 31 Word 0
63	TLB Entry Line 31 Word 1

	TLB Set 1
64	TLB Entry Line 0 Word 0
65	TLB Entry Line 0 Word 1
•	•
•	•
•	•
126	TLB Entry Line 31 Word 0
127	TLB Entry Line 31 Word 1

Figure 3-30. Translation Look-Aside Buffer Registers

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.

TLB registers have 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 only may be specified 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 Set 0 are in registers numbered 0-63, and the entries for TLB Set 1 are in registers numbered 64-127.

TLB Entry Word 0

The TLB Entry Word 0 register is shown in Figure 3-31.

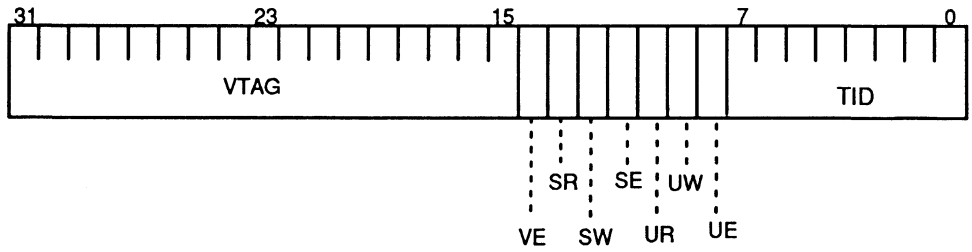


Figure 3-31. TLB Entry Word 0

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 8 Kbytes, 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)
1 Kbyte	AAA
2 Kbyte	AA0
4 Kbyte	A00
8 Kbyte	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 is allows the TLB entry to be associated with a particular process.

TLB Entry Word 1

The TLB Entry Word 1 register is shown in Figure 3-32.

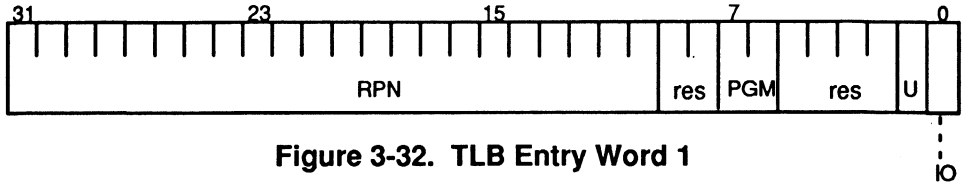


Figure 3-32. TLB Entry Word 1

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 8 Kbytes, 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 8 Kbyte 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)
1 Kbyte	AAA
2 Kbyte	AA0
4 Kbyte	A00
8 Kbyte	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 line was least recently used to perform an address translation. If this bit is a 0, then the entry in Set 0 in the line is least-recently-used; if it is 1, then the entry in Set 1 is least-recently-used. This bit has an equal value for both entries in a line. Whenever a TLB entry is used to translate an address, the Usage bit of both entries in the line used for translation are 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.

3.3 INSTRUCTION SET

The Am29000 implements 117 instructions. All instructions execute in a single cycle, except for IRET, IRETINV, LOADM, STOREM, and the trapping arithmetic instructions such as floating-point instructions.

Most instruction 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, external devices and memories, and coprocessors.

This section describes the nine instruction classes in the Am29000, and provides a brief summary of instruction operations. A detailed instruction specification is contained in Chapter 8. Section 8.1 describes the nomenclature used here.

If the processor attempts to execute an instruction which is not implemented, an Illegal Opcode trap occurs.

3.3.1 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 3-1.

Table 3-1. 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)

(Continued)

Table 3-1. Integer Arithmetic Instructions (Continued)

Mnemonic	Operation Description
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	Intitialize 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)

The instructions MULTIPLU, MULTMU, MULTIPLY, MULTM, DIVIDE, and DIVIDU are not implemented directly by processor hardware, but cause traps to instruction-emulation routines.

3.3.2 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 3.5.4).

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 3-2.

3.3.3 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 3-3.

Table 3-2. 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)

(Continued)

Table 3-2. Compare Instructions (Continued)

Mnemonic	Operation Description
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)

Table 3-3. 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)

3.3.4 SHIFT

The Shift instructions (Table 3-4) 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.

Table 3-4. 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)

3.3.5 DATA MOVEMENT

The Data Movement instructions (Table 3-5) move bytes, half-words, and words between processor registers. In addition, they move data between general-purpose registers and external devices, memories, and the coprocessor.

Table 3-5. 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 *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 *LOCK output during access

(Continued)

Table 3-5. Data Movement Instructions (Continued)

Mnemonic	Operation Description
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

3.3.6 CONSTANT

The Constant instructions (Table 3-6) 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.

Table 3-6. Constant Instructions

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

3.3.7 FLOATING-POINT

The Floating-Point instructions (Table 3-7) provide operations on single-precision (32-bit) or double-precision (64-bit) floating-point data. In addition, they provide conversions between single-precision, double-precision, and integer number representations. In the current processor implementation, these instructions cause traps to routines which perform the floating-point operations.

Table 3-7. 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 (single-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

(Continued)

Table 3-7. Floating-Point Instructions (Continued)

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

3.3.8 BRANCH

The Branch instructions (Table 3-8) 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.

3.3.9 MISCELLANEOUS

The Miscellaneous instructions (Table 3-9) perform various operations that cannot be grouped into other instruction classes. In certain cases, these are control functions available only to Supervisor-mode programs.

Table 3-8. 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

Table 3-9. 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 to zeros
IRET	Perform an interrupt return sequence
IRETINV	Perform an interrupt return sequence, and reset all Valid bits in Branch Target Cache to zeros
HALT	Enter Halt mode on next cycle

3.3.10 RESERVED INSTRUCTIONS

Sixteen Am29000 operation codes are reserved for instruction emulation. These instructions cause traps and set the indirect pointers, much like the Floating-point instructions, but currently have no specified interpretation. The relevant operation codes, and the corresponding trap vectors, are:

Operation Codes (hexadecimal)	Trap Vector Numbers (decimal)
D8-DD	24-29
E7-E9	39-41
F8	56
FA-FF	58-63

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

3.4 DATA FORMATS AND HANDLING

This section describes the various data types supported by the Am29000, and the mechanisms for accessing data in external devices and memories. The Am29000 includes provisions for the external access of bytes, half-words, unaligned words, and unaligned half-words, as described in this section.

3.4.1 INTEGER DATA TYPES

Most Am29000 instructions deal directly with word-length integer data; integers may be either signed or unsigned, depending on the instruction. Some instructions (e.g., AND) treat word-length operands as strings of bits. In addition, there is support for character, half-word, and Boolean data types.

Byte Operations

The processor supports character data through load, store, extraction and insertion operations on word-length operands, and by a compare operation on byte-length fields within words. The format for unsigned and signed characters is shown in Figure 3-33; for signed characters, the sign bit is the most-significant bit of the character. For sequences of packed characters within words, bytes are ordered either left-to-right or right-to-left, depending on the BO bit of the Configuration Register (see Section 3.4.3).

If the Data Width Enable (DW) bit of the Configuration Register is 1, the Am29000 is enabled to load and store byte data. On a load, an external packed byte is converted to one of the character formats shown in Figure 3-33. On a store, the low-order byte of a word is packed into every byte of an external word. Section 3.4.3 describes external byte accesses in more detail.

The Extract Byte (EXBYTE) instruction replaces the low-order character of a destination word with an arbitrary byte-aligned character from a source word. For the EXBYTE instruction, the destination word can be a zero word, which effectively zero-extends the character from the source operand.

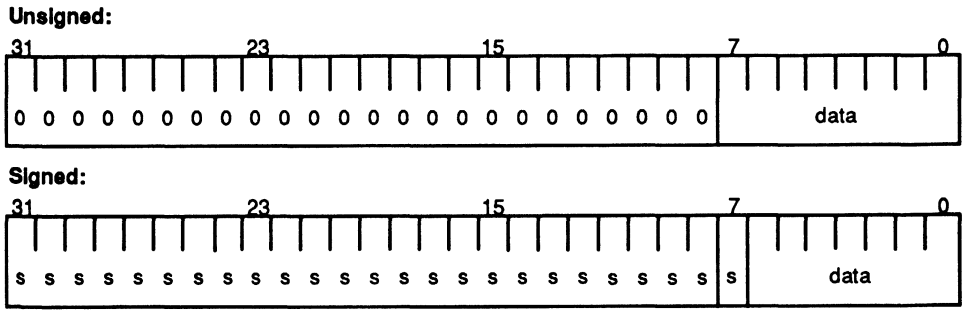


Figure 3-33. Character Format

The Insert Byte (INBYTE) instruction replaces an arbitrary byte-aligned character in a destination word with the low-order character of a source word. For the INBYTE instruction, the source operand can be a character constant specified by the instruction.

The Compare Bytes (CPBYTE) instruction compares two word-length operands and gives a result of TRUE if any corresponding bytes within the operands have equivalent values. This allows programs to detect characters within words without first having to extract individual characters, one at a time, from the word of interest.

Half-Word Operations

The processor supports half-word data through load, store, insertion and extraction operations on word-length operands. The format for unsigned and signed half-words is shown in Figure 3-34; for signed half-words, the sign bit is the most-significant bit of the half-word. For sequences of packed half-words within words, half-words are ordered either left-to-right or right-to-left, depending on the Byte Order (BO) bit of the Configuration Register (see Section 3.4.5).

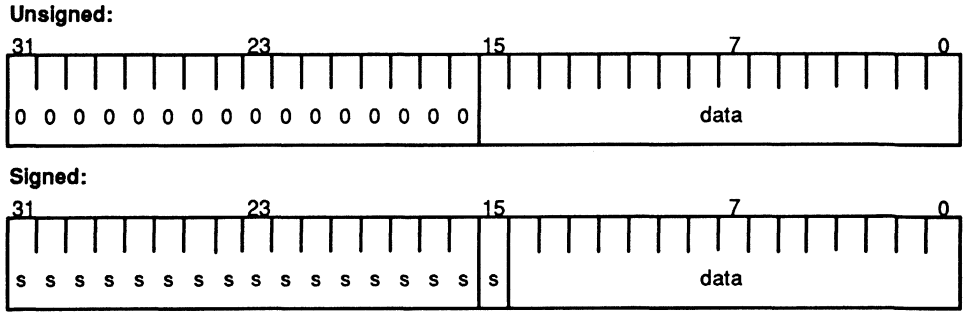


Figure 3-34. Half-Word Format

If the Data Width Enable (DW) bit of the Configuration Register is 1, the Am29000 is enabled to load and store half-word data. On a load, an external packed half-word is converted to one of the formats shown in Figure 3-34. On a store, the low-order half-word of a word is packed into every half-word of an external word. Section 3.4.4 describes external half-word accesses in more detail.

The Extract Half-Word (EXHW) instruction replaces the low-order half-word of a destination word with either the low-order or high-order half-word of a source word. For the EXHW instruction, the destination word can be a zero word, which effectively zero-extends the half-word from the source operand.

The Extract Half-Word, Sign-Extended (EXHWS) instruction is similar to the EXHW instruction, except that it sign-extends the half-word in the destination word (i.e., it replaces the most-significant 16 bits of the destination word with the most-significant bit of the source half-word).

The Insert Half-Word (INHW) instruction replaces either the low-order or high-order half-word in a destination word with the low-order half-word of a source word.

Boolean Data

Some instructions in the Compare class generate word-length Boolean results. Also, conditional branches are conditional upon Boolean operands. The Boolean format used by the processor is such that the Boolean values TRUE and FALSE are represented by a 1 or 0, respectively, in the most-significant bit of a word. The remaining bits are unimportant: for the compare instructions, they are reset. Note that two's-complement negative integers are indicated by the Boolean value TRUE in this encoding scheme.

3.4.2 FLOATING-POINT DATA TYPES

The Am29000 defines single- and double-precision floating-point formats that comply with the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std. 754-1985). These data types are not supported directly in processor hardware, but can be implemented by a virtual floating-point interface provided in the Am29000.

In this section, the following nomenclature is used to denote fields in a floating-point value:

- s: sign bit
- bexp: biased exponent
- frac: fraction
- sig: significand

Single-Precision Floating-Point

The format for a single-precision floating-point value is shown in Figure 3-35.

Typically, the value of a single-precision operand is expressed by:

$$(-1)^{**}s \ * \ 1.\text{frac} \ * \ 2^{**}(\text{bexp}-127)$$

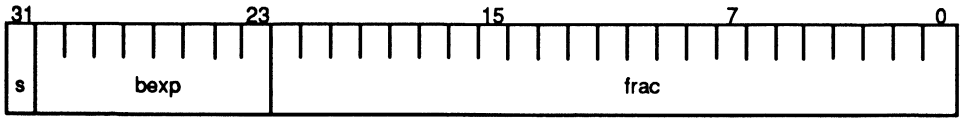


Figure 3-35. Single-Precision Floating-Point Format

The encoding of special floating-point values is given in Section 3.4.3.

Double-Precision Floating-Point

The format for a double-precision floating-point value is shown in Figure 3-36.

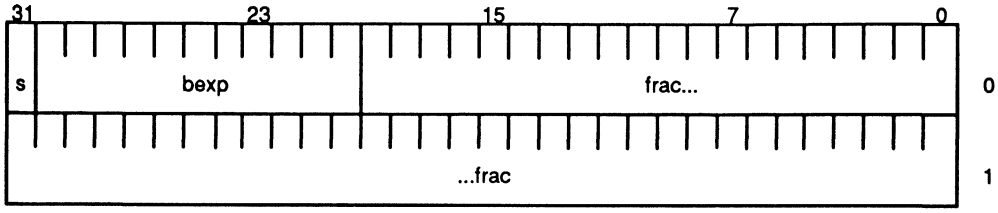


Figure 3-36. Double-Precision Floating-Point Format

Typically, the value of a double-precision operand is expressed by:

(–1)**s * 1.frac * 2**(bexp–1023)

The encoding of special floating-point values is given Section 3.4.3.

In order to be properly referenced by a floating-point instruction, a double-precision floating-point value must be double-word aligned. The absolute-register number of the register containing the first word (labeled "0" in Figure 3-36) must be even. The absolute-register number of the register containing the second word (labeled "1" in Figure 3-36) must be odd. If these conditions are not met, the results of the instruction are unpredictable. Note that the appropriate registers for a double-precision value in the local registers depends on the value of the Stack Pointer.

3.4.3 SPECIAL FLOATING-POINT VALUES

The Am29000 defines floating-point values which are encoded for special interpretation. The values are described in this section.

Not-a-Number

A Not-a-Number (NaN) is a symbolic value used to report certain floating-point exceptions. It also can be used to implement user-defined extensions to floating-point operations. A NaN comprises a floating-point number with maximum biased exponent and non-zero fraction. The sign bit can be

either 0 or 1, and has no significance. There are two types of NaN: signaling NaNs and quiet NaNs. A signaling NaN causes an Invalid Operation exception if used as an input operand to a floating-point operation; a quiet NaN does not cause an exception. The Am29000 distinguishes signaling and quiet NaNs by the most-significant bit of the fraction: a 1 indicates a quiet NaN, and a 0 indicates signaling NaN.

An operation never generates a signaling NaN as a result. A quiet NaN result can be generated in one of two ways:

- as the result of an invalid operation that cannot generate a reasonable result, or
- as the result of an operation for which one or more input operands are either signaling or quiet NaNs.

In either case, the Am29000 produces a quiet NaN having a fraction of 11000...0; that is, the two most-significant bits of the fraction are 11, and the remaining bits are 0. If desired, the Reserved Operand exception can be enabled to cause a Floating-Point Exception trap. The trap handler in this case can implement a scheme whereby user-defined NaN values appear to pass through operations as results, providing overall status for a series of operations.

Infinity

Infinity is an encoded value used to represent a value that is too large to be represented as a finite number in a given floating-point format. Infinity comprises a floating-point number with maximum biased exponent and zero fraction. The sign bit of an infinity distinguishes $+\infty$ from $-\infty$.

Denormalized Numbers

The IEEE Standard specifies that, wherever possible, a result that is too small to be represented as a normalized number be represented as a denormalized number. A denormalized number may be used as an input operand to any operation. For single- and double-precision formats, a denormalized number comprises a floating-point number with a biased exponent of zero and a non-zero fraction field; the sign bit can be either 1 or 0. The value of a denormalized number is expressed by:

```
(-1)**s  * 0.frac * 2**(-bias+1),
```

where "bias" is the exponent bias for the format in question.

Zero

A zero comprises a floating-point number with a biased exponent of zero and a zero fraction field. The sign bit of a zero can be either 0 or 1; however, positive and negative zero are both exactly zero, and are considered equal by comparison operations.

3.4.4 EXTERNAL DATA ACCESSES

All processor external accesses occur between general-purpose registers and external devices and memories. Accesses occur as the result of the execution of load and store instructions. The load and

store instructions specify which general-purpose register receives the data (for a load) or supplies the data (for a store). The format of the load and store instructions is shown in Figure 3-37.

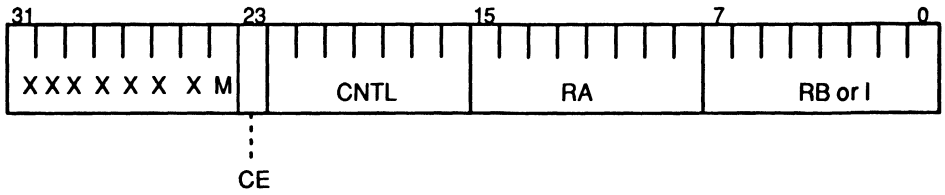


Figure 3-37. Load/Store Instruction Format

Addresses for accesses are given either by the content of a general-purpose register or by a constant value specified by the load or store instruction. The load and store instructions do not perform address computation directly. Any required address computations are performed explicitly by other instructions.

In the load or store instruction, the Coprocessor Enable (CE) bit (bit 23) determines whether or not the access is directed to the coprocessor. If the CE bit is 0, the access is directed to an external device or memory. If the CE bit is 1, data is transferred to or from the coprocessor. The CE bit affects the interpretation of the Control (CNTL) field as well as the channel protocol. Coprocessor accesses are discussed in Chapter 6. This section deals with all other external accesses.

The format of the instructions that do not perform coprocessor data transfers (i.e., in which the CE bit is 0) is shown in Figure 3-38.

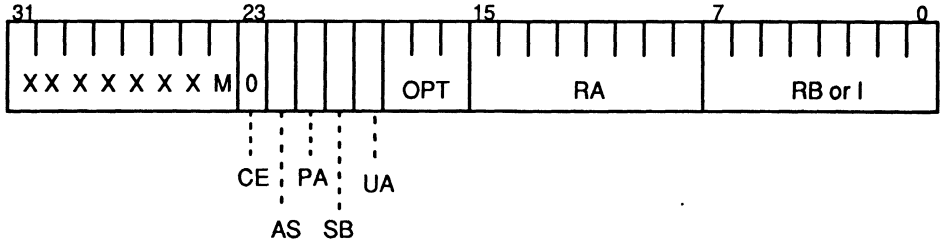


Figure 3-38. Non-Coprocessor Load/Store Format

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

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.

The definitions for other fields in the load or store instruction are given below:

Bit 23 : Coprocessor Enable (CE)—The CE bit is 0 for a non-coprocessor load or store.

Bit 22 : Address Space (AS)—If the AS bit is 0 for an untranslated load or store, the access is directed to instruction/data memory. If the AS bit is 1 for an untranslated load or store, the access is directed to input/output. The AS bit must be 0 for a translated load or store; if the AS bit is 1 for a translated load or store, a Protection Violation trap occurs. The address space for a translated load or store is determined by the Input/Output (IO) bit of the associated TLB entry.

Bit 21 : Physical Address (PA)—The PA bit may be used by a Supervisor-mode program to disable address translation for an access. If the PA bit is 1, then address translation is not performed for the access, regardless of the value of the Physical Addressing/Data (PD) bit in the Current Processor Status Register. If the PA bit is 0, address translation depends on the PD bit.

The PA bit may be 1 only for Supervisor-mode instructions. If it is 1 for a User-mode instruction, a Protection Violation trap occurs.

Bit 20 : Set Byte Pointer/Sign Bit (SB)—If the Data Width Enable (DW) bit of the Configuration Register is 0 and the SB bit is 1, the Byte Pointer Register is written with the two least-significant bits of the address for the access. These address bits can control subsequent character and half-word operations. If the BP bit is 0, the Byte Pointer Register is not affected.

If the Data Width Enable (DW) bit of the Configuration Register is 1 and 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. If the DW bit is 1 and the SB bit is 1 for either a load or store, then 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. 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 TLB protection-checking, the SUP/*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 below.

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. However, the address for the second operation may appear on the Address Bus if the first operation is to a device or memory that supports pipelined operations (see Section 5.2.8).

Load Operations

The processor provides 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 *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 *LOCK output is asserted during both the read and write access. Note that, if address translation is enabled for the LOADSET instruction, the TLB 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 loads a specified number of registers from sequential addresses, as explained below.

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.

Store Operations

The processor provides 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 *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 accesses until it completes.

Multiple Accesses

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.

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 four 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 processor increments 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 transition 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. This bit may be 1 even though the CR field has a value of zero (indicating that one transfer remains to be performed). 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 processor attempts to complete multiple accesses using the burst-mode capability of the channel (see Section 5.2.9). For this reason, multiple accesses of individual bytes and half-words is not supported. If the burst-mode access is preempted, the processor retransmits the address at the point of preemption. If the external device or memory cannot support burst-mode accesses, the processor transmits an address for every access. 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.

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	0	0	Instruction ROM access (as data)
0	1	0	1	Cache control
0	1	1	0	ADAPT29K accesses
	-all others -			reserved

Note that some of these encodings do not affect processor operation, and could have other interpretations in a particular system. For example, the OPT values 000, 001, and 010 affect processor operation only if the DW bit of the Configuration Register is 1. However, non-standard uses of the OPT field have an implication on the portability of software between different systems.

3.4.5 ADDRESSING AND ALIGNMENT

Address Spaces

External instructions and data are contained in one of four 32-bit address-spaces:

1) Instruction/Data Memory

2) Input/Output

3) Coprocessor

4) Instruction Read-Only Memory (Instruction ROM).

An address in the instruction/data memory address space may be treated as virtual or physical, as determined by the Current Processor Status Register. Address translation for data accesses is enabled separately from address translation for instruction accesses. A program in the Supervisor mode may temporarily disable address translation for individual loads and stores; this permits load-real and store-real operations.

It is possible to partition physical instruction and data addresses into two, separate physical address spaces. However, virtual instruction and data addresses appear in the same virtual address space (i.e., instruction/data memory).

The coprocessor address space is not an address space in the strictest sense. The coprocessor address space is defined so that transfers of operands and operation codes to the coprocessor do not interfere with other external devices and memories.

The processor does not directly support the access of the instruction ROM address space using loads and stores; this capability is defined as a system option requiring external hardware.

For untranslated data accesses, bits contained in load and store instructions distinguish between the instruction/data memory, input/output and coprocessor address spaces. For translated data

accesses, the Input/Output bit of the associated TLB entry distinguishes between the instruction/data memory and input/output address spaces.

For instruction fetches, the ROM Enable (RE) bit of the Current Processor Status Register distinguishes between the instruction/data and instruction ROM address spaces.

Byte and Half-Word Addressing

The Am29000 generates 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 are sufficient to select bytes packed within accessed words. For load and store operations, the processor provides means for using the least-significant address bits to access bytes and half-words within external words.

The selection of a byte within a 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 a word is determined by the next-to-least significant bit of an address and the BO bit. Figure 3-39 illustrates the addressing of bytes and half-words when the BO bit is 0 (big endian), and Figure 3-40 illustrates the addressing of bytes and half-words when the BO bit is 1 (little endian). In Figure 3-39 and Figure 3-40, addresses are represented in hexadecimal notation.

In the processor, the two least-significant bits of an external address can be reflected in the Byte Pointer (BP) field of the ALU Status Register when the DW bit of the Configuration Register is 0. Alternatively, the two least-significant bits of the address can be used to control byte and half-word accesses when the DW bit is 1. 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.

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.

Alignment of Words and Half-Words

Since only byte addressing is supported, it is possible that an address for the access of a word or half-word is not aligned to the desired word or half-word. The Am29000 either ignores or forces alignment in most cases. However, some systems may require that unaligned accesses be supported, for compatibility reasons. Because of this, the Am29000 provides an option that creates

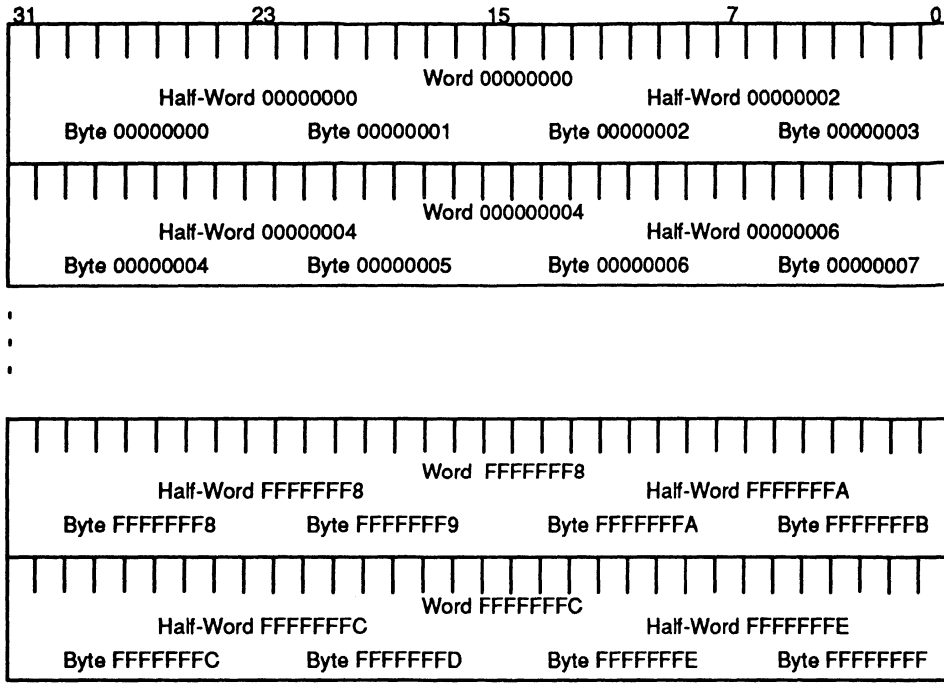


Figure 3-39. Byte and Half-Word Addressing with BO = 0 (Big Endian)

a 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 and coprocessor transfers.

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	
0	0	0	1	0	Unaligned
0	0	0	0	1	word access
0	0	0	1	1	
0	1	0	0	1	Unaligned
0	1	0	1	1	half-word access

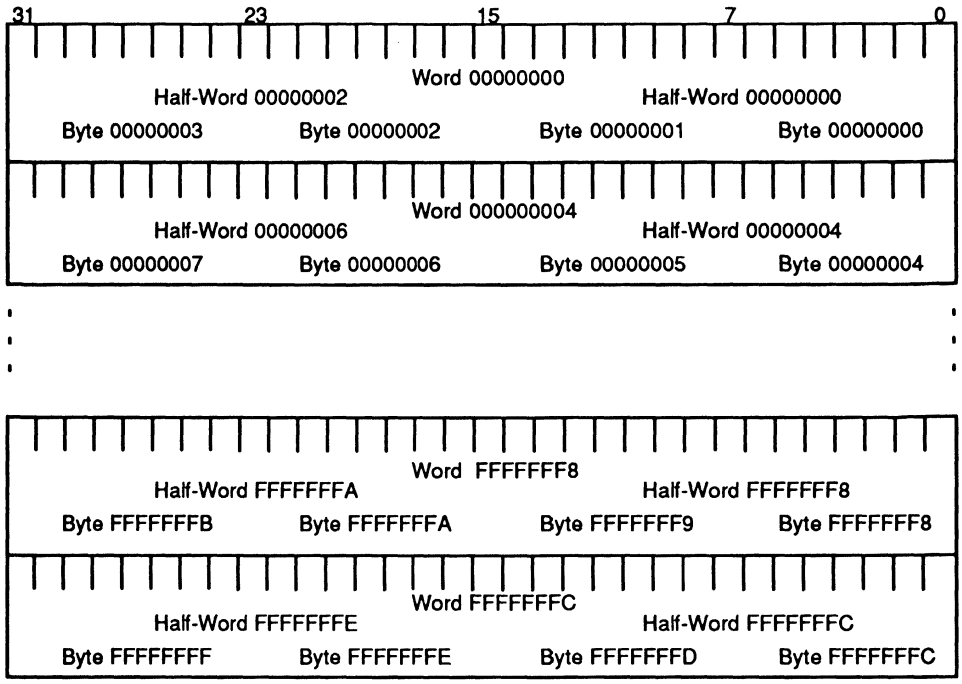


Figure 3-40. Byte and Half-Word Addressing with BO = 1 (Little Endian)

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 also may have to be considered.

Alignment of Instructions

In the Am29000, 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 it expects the system to force alignment (i.e., by interpreting the two least-significant address bits as 00, regardless of their values).

Accessing Instructions as Data

To aid the external access of instructions and data on separate buses, the processor distinguishes between instruction and data accesses. However, it does not support a logical distinction between instruction and data address spaces (except in the case of instruction read-only memory). In particular, address translation in the Memory Management Unit is in no way affected by this distinction (although memory protection is).

In systems where it is necessary to access instructions as data, this function should be performed via the shared address space. The OPT field provides a means for loads to access instructions in the

instruction read-only memory (ROM) address space. The Am29000 does not take any action to prevent a store to the instruction ROM address space.

3.4.6 BYTE AND HALF-WORD ACCESSES

The Am29000 can perform byte and half-word accesses in either software or hardware, under control of the Data Width Enable (DW) bit of the Configuration Register. Software byte and half-word accesses are selected by a DW bit of 0, and hardware byte and half-word accesses are selected by a DW bit of 1. Software byte and half-word accesses are less efficient than hardware byte and half-word accesses, but hardware accesses require that the system be able to selectively write individual byte and half-word positions within external devices and memories. The software-only technique is compatible with systems designed to provide hardware support for byte and half-word accesses.

This section describes the operation of both software and hardware byte and half-word accesses. Byte and half-word accesses operate as described here for memory and input/output accesses, but not for coprocessor transfers. Coprocessor transfers are unaffected by the DW bit.

The DW bit is cleared by a processor reset. It must explicitly be set to 1 by software before hardware byte and half-word accesses can be performed.

Software Byte and Half-Word Accesses

If the DW bit is 0, the Am29000 allows the Byte Pointer Register to be set with the least-significant bits of an address specified by any load or store instruction, except those that transfer information to and from the coprocessor. Insert and extract instructions can then be used to access the byte or half-word of interest, after the external word has been accessed. This provides a general-purpose mechanism for manipulating external byte and half-word data, without the need for external hardware support.

To load a byte or half-word, a word load first is performed. This load sets the BP field with the two least-significant bits of the address. A subsequent EXBYTE, EXHW or EXHWS instruction extracts the byte or half-word of interest from the accessed word.

To store a byte or half-word, a load is first performed, setting the BP field with the two least-significant bits of the address. A subsequent INBYTE or INHW instruction inserts the byte or half-word of interest into the accessed word, and the resulting word then is stored.

Software which relies on loads and stores setting the BP field cannot operate correctly when the Freeze (FZ) bit of the Current Processor Status Register is 1, because the ALU Status Register is frozen.

Hardware Byte and Half-Word Accesses

If the DW bit is 1 on a load, the Am29000 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.

If the DW bit is 1 on a store, the Am29000 replicates the low-order byte or half-word in the source register into every byte and half-word position of the stored word. The system is responsible for generating the appropriate byte and/or half-word strobes, based on the OPT(2:0) signals and the two least-significant bits of the address, to write the appropriate byte or half-word in the selected device or memory (the system byte order must also be considered). 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, and the DW bit is also 1, 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. Hardware byte and half-word accesses—in contrast to software byte and half-word accesses—can be performed when the FZ bit is 1, because these accesses do not rely on the BP field.

System Alternatives and Compatibility

The two mechanisms for performing byte and half-word accesses create the possibility of two types of systems. These are named for convenience:

- Type 1: simple, word-only accesses in external devices and memories; software byte and half-word accesses.

- Type 2: byte/half-word strobes in external devices and memories; hardware byte and half-word accesses by the Am29000.

The provision for hardware byte and half-word accesses encourages Type 2 systems. Software for Type 1 systems can execute on Type 2 systems, but the reverse is not true. Software compatibility is possible primarily because of the DW bit and because the Am29000 sets the BP field with an appropriate byte pointer even when it performs byte and half-word accesses with internal hardware. Also, the system must return a full word in either type of system, regardless of the access data-width. The DW bit must be 0 in Type 1 systems and must be 1 in Type 2 systems. To illustrate compatibility between systems, consider the following steps of an unsigned byte load compiled for a Type 1 system, but executing on a Type 2 system:

1) Perform a load with OPT=001 and SB=1.

   - Type 1 system: The addressed word is accessed and placed into the destination register. The BP field is set with the two least-significant bits of the address.

   - Type 2 system: The addressed byte is accessed, aligned, padded, and placed into the destination register. The BP field is set to point to the low-order byte,

reflecting the alignment that has been performed (the pointer depends on the value of the BO bit).

2) Perform a byte extract on the loaded word.

   - Type 1 system: The byte selected by the BP field is aligned to the low-order byte of the destination register and the remainder of the word is zero-extended. The selected byte may be in any byte position.

   - Type 2 system: The byte selected by the BP field (set to point to the low-order byte) is aligned to the low-order byte of the destination register and the remainder of the word is zero-extended. (Note that the selected byte was already in the low-order byte position. This operation does not change program state but merely allows software compatibility).

The recommended instruction sequences for all types of byte and half-word accesses and for both types of systems are enumerated below. Compatibility between these systems follows the above example, but for brevity, compatibility is not described in detail here.

Byte read, unsigned:

```
Type 1                      Comments
load 0,17,temp,addr;        ; OPT=001, SB=1
exbyte temp,temp,0          ; get byte

Type 2                      Comments
load 0,1,temp,addr;         ; OPT=001, SB=0
```

Byte read, signed:

```
Type 1                      Comments
load 0,17,temp,addr         ; OPT=001, SB=1
exbyte temp,temp,0          ; get byte
sll temp,temp,24            ; sign extend
sra temp,temp,24

Type 2                      Comments
load 0,17,temp,addr         ; OPT=001, SB=1 (sign extended)
```

Byte Write:

```
Type 1                      Comments
load 0,17,temp,addr         ; OPT=001, SB=1
inbyte temp,temp,data       ; insert byte
store 0,1,temp,addr         ; store
```

```
Type 2                      Comments
store 0,1,data,addr         ; OPT=001, SB=0
```

Half-word read, unsigned:

```
Type 1                      Comments
load 0,18,temp,addr         ; OPT=010, SB=1
exhw temp,temp,0            ; get half-word unsigned

Type 2                      Comments
load 0,2,temp,addr          ; OPT=010, SB=0
```

Half-word read, signed:

```
Type 1                      Comments
load 0,18,temp,addr         ; OPT=010, SB=1
exhws temp,temp             ; get half-word sign-extend

Type 2                      Comments
load 0,18,temp,addr         ; OPT=010, SB=1, (sign-extend)
```

Half-word write:

```
Type 1                      Comments
load 0,18,temp,addr         ; OPT=010, SB=1
inhw temp,temp,data         ; insert half-word
store 0,2,temp,addr         ; store

Type 2                      Comments
store 0,2,data,addr         ; OPT=010, SB=0
```

3.5 INTERRUPTS AND TRAPS

Interrupts and traps cause the Am29000 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.

Throughout this manual, 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.

3.5.1 INTERRUPTS

Interrupts are caused by signals applied to any of the external inputs *INTR(3:0), or by the Timer Facility (see Section 7.2.7). 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 and most traps. 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	*INTR0 enabled
01	*INTR(1:0) enabled
10	*INTR(2:0) enabled
11	*INTR(3:0) enabled

Note that the *INTR0 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 *INTR(3:0) is active, but that the corresponding interrupt is disabled due to the value of either DA, DI, or IM.

3.5.2 TRAPS

Traps are caused by signals applied to one of the inputs *TRAP(1:0), or by exceptional conditions such as protection violations. Except for the Instruction Access Exception, Data Access Exception, and Coprocessor 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.

3.5.3 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 the Reset mode (see Section 3.8) or a *WARN trap (see Section 3.5.6).

3.5.4 VECTOR AREA

Interrupt and trap processing relies on the existence of a user-managed Vector Area in external instruction/data memory or instruction read-only memory (instruction ROM). 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 24 routines for system operation and 40 routines for instruction emulation. The number and definition of the remaining 192 possible routines are system-dependent.

The Vector Area has one of two possible structures as determined by the Vector Fetch (VF) bit in the Configuration Register. The first structure, as described below, requires less external memory than the second, but imposes the performance penalty of the vector-table lookup.

If the VF bit is 1, 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 3-41. Each vector gives the beginning word-address of the associated interrupt or trap handling routine, and specifies, by the R bit, whether the routine is contained in instruction/data memory (R = 0) or instruction ROM (R = 1).

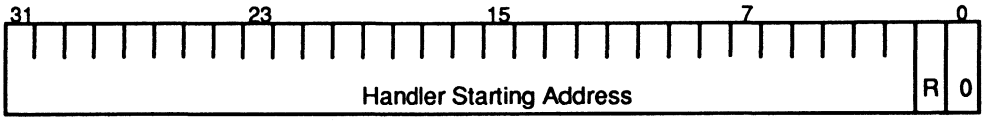


Figure 3-41. Vector Table Entry

If the VF bit is 0, the structure of the Vector Area is a segment of contiguous blocks of instructions in instruction/data memory or instruction ROM. The ROM Vector Area (RV) bit of the Configuration Register determines whether the Vector Area is in instruction/data memory (RV = 0) or instruction ROM (RV = 1). A 64-instruction block contains exactly one interrupt or trap handling routine, and blocks are aligned on 64-instruction address boundaries.

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 either the number of a vector table entry or the number of an instruction block, depending on the value of the VF bit. If the VF bit is 1, 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. If the VF bit is 0, the physical address of the first instruction of the handling routine is generated by replacing bits 15-8 of the value in the Vector Table Base Address Register with the vector number.

Vector numbers are either pre-defined, or specified by an instruction causing the trap. The assignment of vector numbers is shown in Table 3-10 (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.

Table 3-10. Vector Number Assignments

Number	Type of Trap or Interrupt	Cause
0	Illegal Opcode	executing undefined instruction*
1	Unaligned Access	access on unnatural boundary, TU=1
2	Out of Range	overflow or underflow
3	Coprocessor Not Present	coprocessor access, CP=0
4	Coprocessor Exception	coprocessor *DERR response
5	Protection Violation	invalid User-mode operation
6	Instruction Access Exception	*IERR response
7	Data Access Exception	*DERR response, not coprocessor
8	User-Mode Instruction TLB Miss	no TLB entry for translation
9	User-Mode Data TLB Miss	"
10	Supervisor-Mode Instruction TLB Miss	"
11	Supervisor-Mode Data TLB Miss	"
12	Instruction TLB Protection Violation	TLB UE/SE=0
13	Data TLB Protection Violation	TLB UR/SR=0, UW/SW=0 on write
14	Timer	Timer Facility
15	Trace	Trace Facility
16	*INTR0	*INTR0 input
17	*INTR1	*INTR1 input
18	*INTR2	*INTR2 input
19	*INTR3	*INTR3 input
20	*TRAP0	*TRAP0 input
21	*TRAP1	*TRAP1 input
22	Floating-Point Exception	unmasked floating-point exception
23	reserved	
24-29	reserved for instruction emulation (op codes D8-DD)	

* This vector number also results if an external device removes *INTR (3:0) or *TRAP (1:0) before the corresponding interrupt or trap is taken by the processor.

(Continued)

3.5.5 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.

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.

Table 3-10. Vector Number Assignments (Continued)

Number	Type of Trap or Interrupt	Cause
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 (op codes 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
54	FDIV	FDIV instruction
55	DDIV	DDIV instruction
56	reserved for instruction emulation (op code F8)	
57	FDMUL	FDMUL instruction
58-63	reserved for instruction emulation (op codes FA-FF)	
64-255	Assert and EMULATE instruction traps (vector number specified by instruction)	

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 3-42 (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.

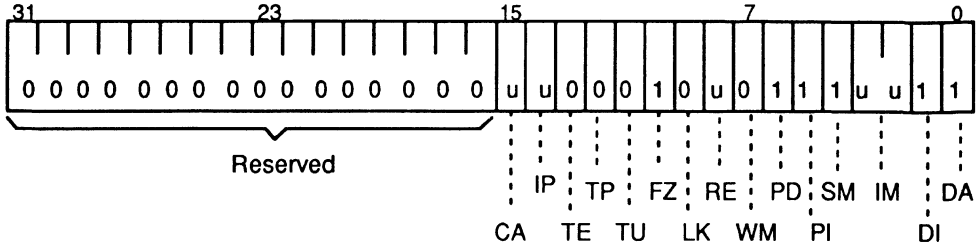


Figure 3-42. Current Processor Status After an Interrupt or Trap

6) The address of the first instruction of the interrupt or trap handler is determined. If the VF bit of the Configuration Register is 1, 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 channel as a data access, and the OPT(2:0) signals indicate a word-length access. If the VF bit is 0, the instruction address is given directly by the Vector Area Base Address Register and the vector number.

7) If the VF bit is 1, the R bit in the vector fetched in step 6 is copied into the RE bit of the Current Processor Status Register. If the VF bit is 0, the RV bit of the Configuration Register is copied into the RE bit. This step determines whether or not the first instruction of the interrupt handler is in instruction ROM.

8) 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 of traps are re-enabled, to allow program state to be reflected properly in processor registers if an interrupt or trap is taken.

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 Branch Target 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 3-43 (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 channel operations 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.

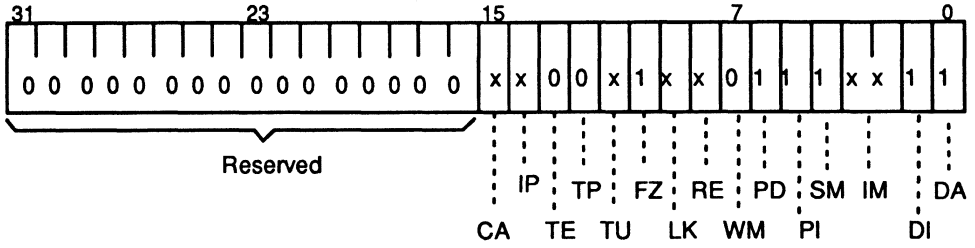


Figure 3-43. Current Processor Status Before Interrupt Return

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:

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 Branch Target Cache are reset.

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 Branch Target 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 Branch Target 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.

Fast 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.

The processor provides an additional benefit 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 in 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 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 3.5.9.

3.5.6 *WARN TRAP

The processor recognizes a special trap, caused by the activation of the *WARN input, which cannot be masked. The *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 *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, regardless of the VF bit of the Configuration Register, when the *WARN trap is taken. Instead, the ROM Enable (RE) bit in the Current

Processor Status is set, and instruction fetching begins immediately at address 16 in the instruction ROM. The trap handler executes directly from the instruction ROM without the need to access external (and possibly non-functional or invalid) instruction/data memory.

Note that *WARN trap may disrupt the state of the routine that is executing when it is taken, prohibiting this routine from being restarted.

3.5.7 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 3-11. In this table, interrupts and traps are listed in order of decreasing priority. This section discusses the first three columns of Table 3-11. The last two columns are discussed in Section 3.5.8.

In Table 3-11, 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 3-11 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 3-11 with priority 13 or 14 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 3) below.

3) The exceptions to 2) above are the Data Access Exception trap, the Coprocessor 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 and Coprocessor Exception traps, 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 3-11, except for the Unaligned Access trap, are not regenerated. These traps are mutually exclusive, and are given high priority because they

Table 3-11. Interrupt and Trap Priority Table

PRIORITY	TYPE OF INTERRUPT OR TRAP	INST/ASYNC	PC1	Channel Regs
1 (highest)	*WARN	async	next	see Note 1
2	User-Mode Data TLB Miss	inst	next	all
	Supervisor-Mode Data TLB Miss	inst	next	all
	Data TLB Protection Violation	inst	next	all
3	Unaligned Access	inst	next	all
	Coprocessor Not Present	inst	next	all
	Out of Range	inst	next	N/A
	Floating-Point Exceptions	inst	next	N/A
	Assert Instructions	inst	next	N/A
	Floating-Point Instructions	inst	next	N/A
	MULTIPLY	inst	next	N/A
	MULTM	inst	next	N/A
	DIVIDE	inst	next	N/A
	MULTIPLU	inst	next	N/A
	MULTMU	inst	next	N/A
	DIVIDU	inst	next	N/A
	EMULATE	inst	next	N/A
4	Data Access Exception	async	next	all
	Coprocessor Exception	async	next	all
5	*TRAP0	async	next	multiple
6	*TRAP1	async	next	multiple
7	*INTR0	async	next	multiple
8	*INTR1	async	next	multiple
9	*INTR2	async	next	multiple
10	*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 TLB Protection Violation	inst	curr	N/A
	Instruction Access Violation	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 *WARN trap only if an external access is in progress when the trap is taken.

cannot be regenerated; they must be taken if they occur. If one of these traps occurs at the same time as a reset or *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 4) above.

Note that the Channel Address, Channel Data, and Channel Control registers are set for a *WARN trap only if an external access is in progress when the trap is taken.

3.5.8 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 or coprocessor transfer 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 3-11 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.

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 Exception Opcode Register contains the operation code of the instruction causing the trap.

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 a non-sequential fetch occurs before the processor attempts the execution of the invalid instruction. This prevents the spurious indication of instruction exceptions.

Data Exceptions

The "Channel Regs" column of Table 3-11 indicates the cases for which the Channel Address, Channel Data, and Channel Control registers contain information related to an external access or

coprocessor transfer (these registers collectively are termed "channel registers" in the following discussion). For the cases indicated, the access or transfer did not complete because of some exceptional condition. Note that the Channel Data Register contains relevant information only in the case of a store.

For the *WARN trap, the channel registers are valid only if a load or store were in progress when the trap was taken. Recall that the *WARN trap does not wait for any in-progress access to complete.

For the traps with an "all" in the "Channel Regs" column of Table 3-11, the channel registers contain information relevant to the trap in all cases. These traps are associated with exceptional events during external accesses or coprocessor transfers.

For the traps with a "multiple" in the "Channel Regs" column, the channel registers might contain information for restarting an interrupted load-multiple or store-multiple operation. In these cases, the operation did not encounter an exception, but was simply canceled for latency considerations.

The information contained in the channel registers allows the processor to restart the related operation during an interrupt return sequence, without any special assistance by software. Software must only insure that the relevant information is retained in, or restored to, the channel registers before an interrupt return is executed.

3.5.9 ARITHMETIC EXCEPTIONS

Integer and floating-point instructions can cause Out of Range or Floating-Point Exception traps, respectively, if an exception is detected during the arithmetic operation. This section describes the conditions under which these traps occur and the additional operations performed beyond those described in Section 3.5.5.

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.

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 bit 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.

In addition to the operations described Section 3.5.5, the following operations are performed when an Out of Range trap is taken:

1) The operation code of the instruction causing the exception is placed in the IOP field of the Exception Opcode Register.

2) For the MULTIPLY, MULTIPLU, DIVIDE, and DIVIDU instructions, the absolute register numbers of the excepting instruction's source and destination registers are placed into the Indirect Pointer A, Indirect Pointer B, and Indirect Pointer C registers.

3) For the MULTIPLY, MULTIPLU, DIVIDE, and DIVIDU instructions, the destination register or registers are unchanged.

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.

In addition to the operations described Section 3.5.5, the following operations are performed when a Floating-Point Exception trap is taken:

1) The operation code of the instruction causing the exception is placed in the IOP field of the Exception Opcode Register.

2) The status of the trapping operation is written into the trap status bits of the Floating-Point Status Register. The status bits that are written do not depend on the values of the corresponding mask bits in the Floating-Point Environment Register.

3) The absolute-register numbers of the excepting instruction's source and destination registers are placed into the Indirect Pointer A, Indirect Pointer B, and Indirect Point C registers. If the RB or RC field specifies a function code, that code is transferred to the corresponding indirect pointer. Note that, if the most-significant bit of the this function code is one, the value of the Stack Pointer has been added to the RB field, and must be subtracted to recover the original field.

4) The destination register or registers are left unchanged.

3.5.10 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 assumed that these routines do not create many of the exceptions possible in most other processor routines, so most of these are ignored.

If the assumption of no exceptions is 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.

It is possible that errors reported via the *IERR and *DERR signals are associated with hardware errors, independent of any routine being executed. For this reason, the Instruction Access Exception, Data Access Exception, and Coprocessor Exception traps cannot be disabled by the DA bit, and the processor may take one of these traps even while handling another interrupt or trap.

If the processor does take an unmaskable trap while handling another interrupt or trap, and the state of the interrupt or trap handler is not reflected in processor registers, it is not possible to return to the point at which the unmaskable trap is taken. When the unmaskable trap is taken, the processor state saved is that state associated with the original interrupt or trap, not with the unmaskable trap; however, the Old Processor Status Register is modified to reflect the Current Processor Status Register of the interrupt or trap handler. This situation, indicated by the DA bit being 1 in the Old Processor Status Register, may not be recoverable.

3.6 MEMORY MANAGEMENT

The Am29000 incorporates a Memory Management Unit (MMU) for performing virtual-to-physical address translation and memory access protection. This section describes the logical operation of the Memory Management Unit. Related issues are discussed in Sections 7.2.4 and 7.2.5.

Address translation can be performed only for instruction/data memory accesses. No address translation is performed for instruction ROM, input/output, coprocessor or interrupt/trap vector accesses. However, an instruction/data memory access can be re-directed to input/output by the address-translation process.

3.6.1 TRANSLATION LOOK-ASIDE BUFFER

The MMU stores the most-recently performed address translations in a special cache, the Translation Look-Aside Buffer (TLB). All virtual addresses generated by the processor are translated by the TLB. Given a virtual address, the TLB determines the corresponding physical address.

The TLB reflects information in the processor 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 3-44. The TLB is a table of 64 entries, divided into two equal sets, called Set 0 and Set 1. Within each set, entries are numbered 0 to 31. Entries in different sets which have equivalent entry-numbers are grouped into a unit called a line; there are thus 32 lines 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 3.2.3.

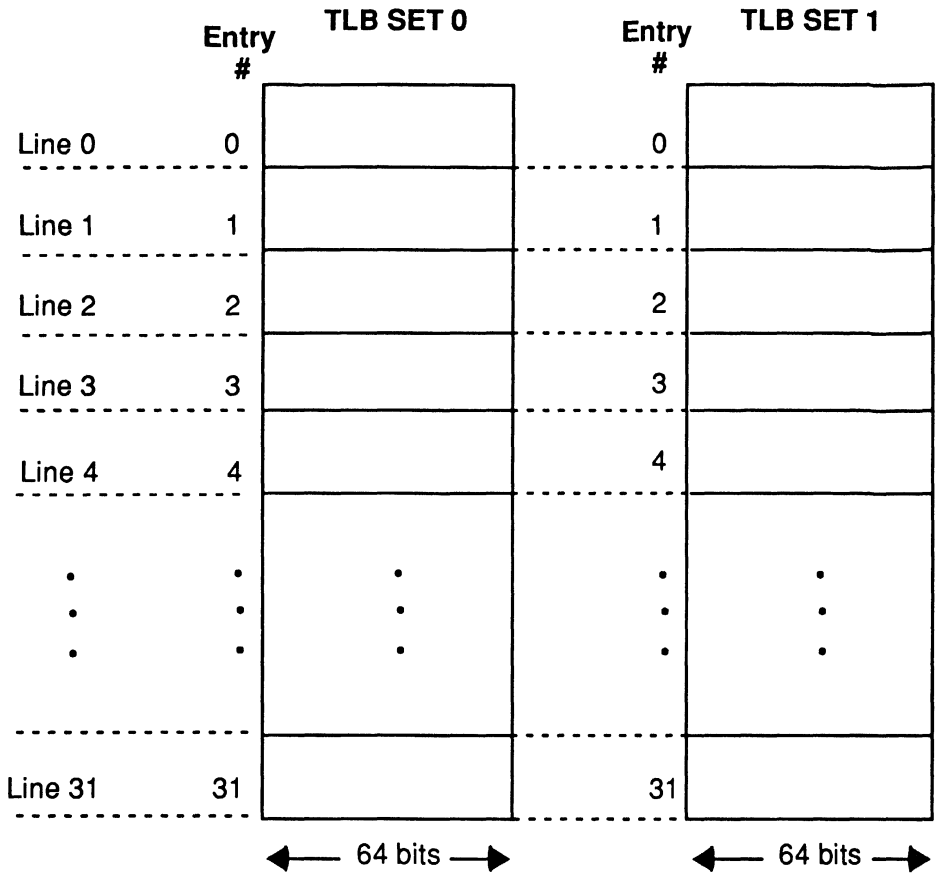


Figure 3-44. Translation Look-Aside Buffer Organization

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 set of the entry within a given line was least recently used to perform an address translation. Usage bits for two entries in the same line are equivalent.

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

3.6.2 ADDRESS TRANSLATION

For the purpose of address translation, the virtual instruction/data address-space of a process is partitioned into regions of fixed size, called pages. Pages are mapped by the address-translation

process 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.

Virtual addresses are partitioned into three fields for the address-translation process, as shown in Figure 3-45. The partitioning of the virtual address is based on the page size. Pages may be of size 1, 2, 4, or 8 Kbytes, as specified by the MMU Configuration Register. The fields shown in Figure 3-45 are described in the following discussion.

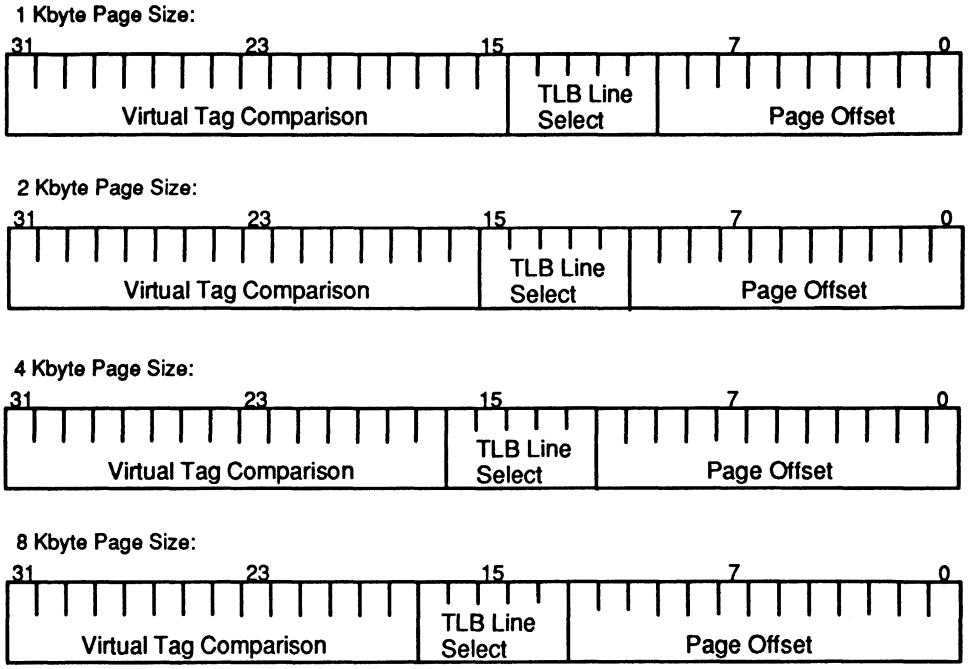


Figure 3-45. Virtual Address for 1, 2, 4, and 8 Kbyte Pages

Address Translation Controls

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

1) Instruction accesses, if the Physical Addressing/Instructions (PI) and ROM Enable (RE) bits of the Current Processor Status are both 0.

2) User-mode accesses to instruction/data memory if the Physical Addressing/Data (PD) bit of the Current Processor Status 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 Current Processor Status is 0.

Address translation also is controlled by the MMU Configuration Register. This register specifies the virtual page size, and contains an 8-bit Process Identifier (PID) field. The PID field specifies the process number associated with the currently running program, if this is a User-mode program. Supervisor-mode programs are assigned a fixed process number of zero. The process number is compared with Task Identifier (TID) fields of the TLB entries during address translation. The TID field of a TLB entry must match the process number for the translation to be valid.

Address Translation Process

The address-translation process is diagrammed in Figure 3-46. 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 line whose number is given by certain bits in the virtual address. The bits used depend on the page size as follows:

Page Size	Virtual Address Bits (for Line Access)
1 Kbyte	14-10
2 Kbyte	15-11
4 Kbyte	16-12
8 Kbyte	17-13

The accessed line 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
1 Kbyte	31-15	16-0
2 Kbyte	31-16	16-1
4 Kbyte	31-17	16-2
8 Kbyte	31-18	16-3

Certain bits of the VTAG field do not participate in the comparison for page sizes larger than 1 Kbyte. 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.

3) The VE bit in the TLB entry is 1.

4) Only one entry in the line 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.

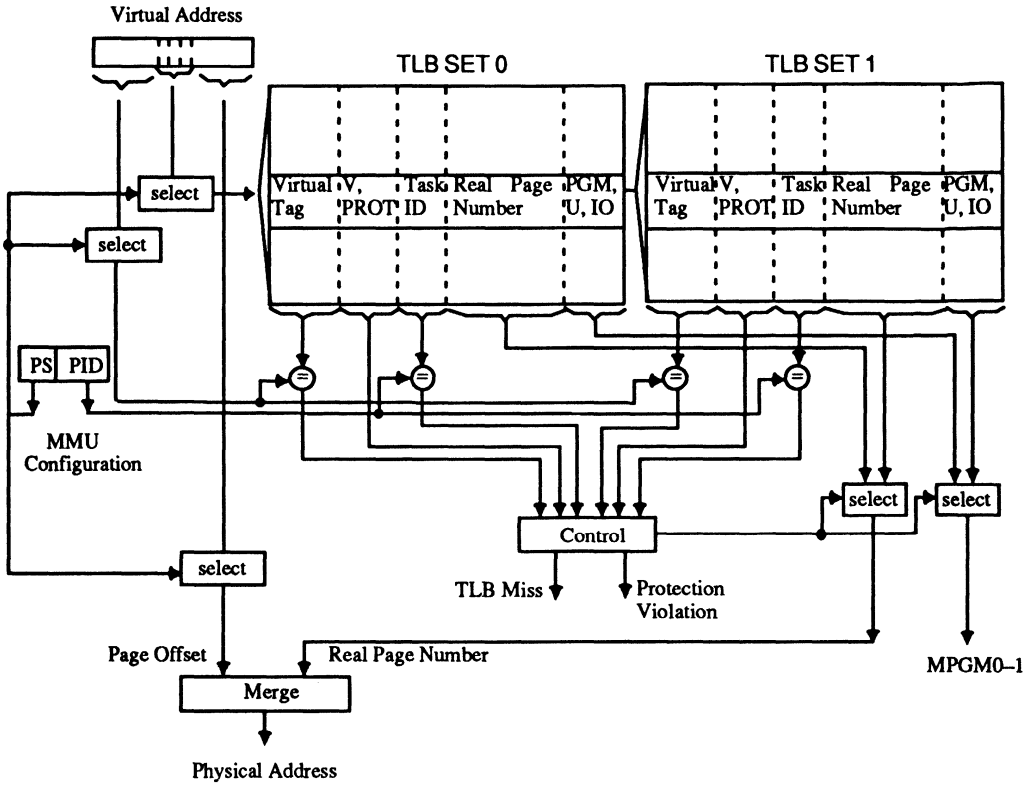


Figure 3-46. Address Translation Process

If the address translation is valid for one TLB entry in the selected line, 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 (because of byte addressing) within the virtual page. 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
1 Kbyte	21-0	9-0
2 Kbyte	21-1	10-0
4 Kbyte	21-2	11-0
8 Kbyte	21-3	12-0

Note: Certain bits of the RPN field are not used in forming the physical address for page sizes greater than 1 Kbyte. These bits of the RPN are required to be zero. In addition, for certain instruction accesses, the Page Offset is incremented by 16 as described in Section 4.2.3.

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.

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 3.6.5 describes 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. Depending on the state of the channel interface, the access request may be placed on the Address Bus with the signal *BINV asserted, even though the 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 TLB Protection Violation
13	Data TLB Protection Violation

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

3.6.3 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 is written with 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.

3.6.4 ENTRY INVALIDATION

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.

3.6.5 PROTECTION

If an address translation is performed successfully as described in Section 3.6.2, the TLB entry used in address translation is used to perform protection checking for the access. There are six bits in the TLB entry 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 3-12 (the value "x" is a don't care).

Table 3-12. TLB 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

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.

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

3.7 SERIALIZATION

Since the Am29000 overlaps an external data reference with other operations, it is necessary to restrict these other operations so that the data reference may be restarted if necessary.

The restriction is, in general, that the processor cannot perform any operation that changes the conditions under which the external access occurs. This is accomplished by serializing certain operations; that is, by having the processor enter the Pipeline Hold mode if an external access has not completed when the operation is attempted. Serialization is performed for the following operations:

1) The execution of one of the following instructions:
   - Move to Special Register
   - Move to Special Register Immediate
   - Move To TLB
   - Interrupt Return
   - Interrupt Return and Invalidate
   - Halt

2) The taking of an interrupt or trap, except for a *WARN trap.

If the processor is in the Pipeline Hold mode due to serialization, it enters the Executing mode once the external access completes. Note that the processor may immediately take a Data Access Exception or Coprocessor Exception trap instead of performing the operation that originally caused the serialization.

3.8 INITIALIZATION

When power is first applied to the processor, it is in an unknown 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 is invoked by activating the *RESET pin for at least four cycles. 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 3-47.

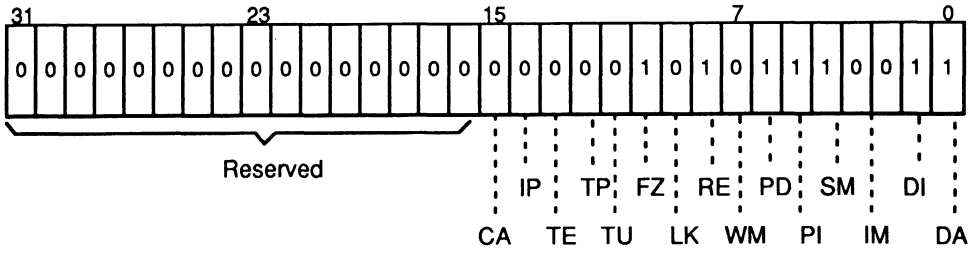


Figure 3-47. Current Processor Status Register In Reset Mode

5) The Cache Disable bit of the Configuration Register is set.

6) The Data Width Enable bit of the Configuration Register is reset.

7) The Contents Valid bit of the Channel Control Register is reset.

Except as previously noted, the contents of all general-purpose registers, special-purpose registers, and TLB registers are undefined. The contents of the Branch Target Cache are also undefined.

The Reset mode also configures the processor to initiate an instruction fetch using an address of 0. Since the ROM enable (RE) bit of the Current Processor Status is 1, this fetch is directed to external instruction read-only memory. This fetch occurs when the Reset mode is exited (i.e., when the *RESET input is de-asserted). Section 5.5 contains more information on this instruction fetch.

CHAPTER 4

HARDWARE FEATURES

This chapter describes the operation of the Am29000 pipeline, and the processor's three major functional units. The functional units are the Instruction Fetch Unit, the Execution Unit, and the Memory Management Unit. These units, which were shown in abstract form in Figure 2-2, are shown in detail in Figure 4-1.

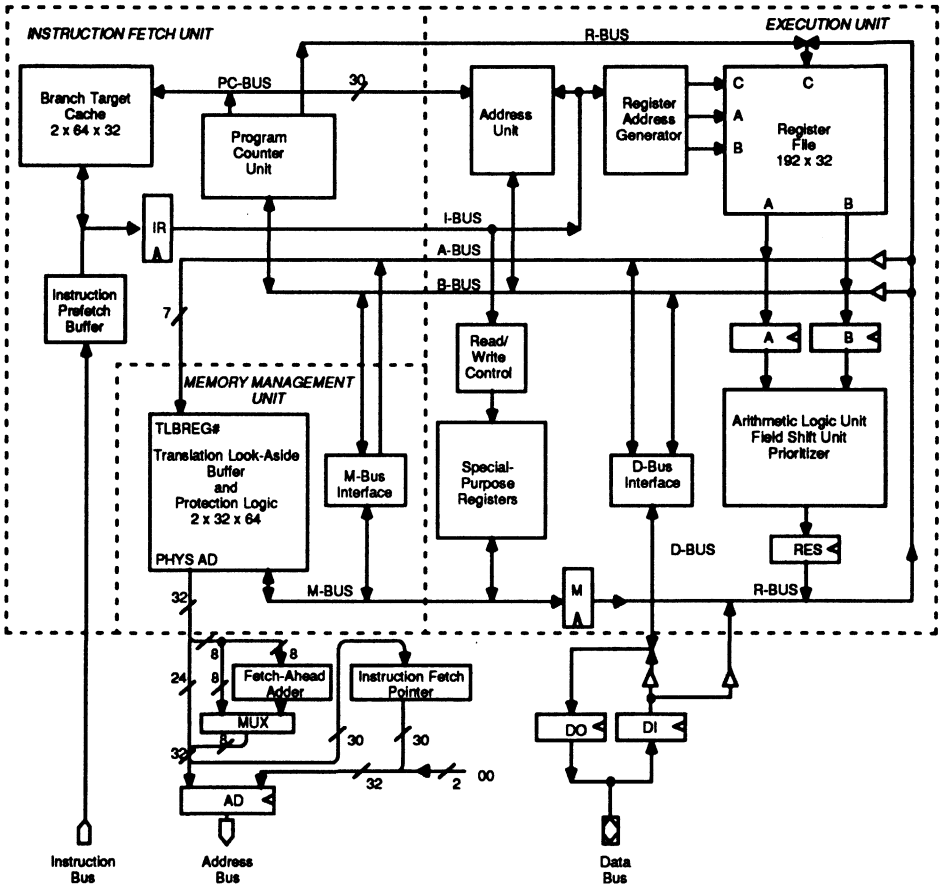


Figure 4-1. Am29000 Data Flow

The operation of the functional units is coordinated by the Pipeline Hold mode, which insures that operations are performed in the proper order. This chapter also describes the Pipeline Hold mode.

Since this chapter describes the internal operation of the Am29000, it provides information that may not be required by some users. However, it aids an understanding of the behavior of the Am29000 under certain conditions, especially the behavior of the system interfaces described in Chapter 5.

4.1 FOUR-STAGE PIPELINE

The Am29000 implements a four-stage pipeline for instruction execution. The four stages are fetch, decode, execute, and write-back. 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 (Section 4.2) 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 Branch Target Cache or an external instruction memory.

During the decode stage, the Execution Unit (Section 4.3) decodes the instruction selected during the fetch stage, and fetches and/or assembles the required operands. It also evaluates addresses for branches, loads, and stores.

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 (Section 4.4) 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 Am29000 (see Section 7.4).

4.2 INSTRUCTION FETCH UNIT

The Instruction Fetch Unit performs the functions required to keep the processor pipeline supplied with instructions. Since the processor can execute one instruction per cycle, instructions must be supplied at this rate if the execution stage is to perform at the maximum rate. To accomplish this, the Instruction Fetch Unit contains mechanisms for requesting instructions from instruction memory before they are required for execution, and for caching the most-recently executed branch target instructions.

The Instruction Fetch Unit also incorporates the logic necessary to calculate and sequence instruction addresses. The processor is word-oriented, but generates byte addresses for all external accesses. Since all processor instructions are word-length, and are aligned on word-address boundaries, the Instruction Fetch Unit deals only with 30-bit addresses. For external instruction accesses, these addresses are appended with 00 in the two least-significant bits to form the required 32-bit address (note that the two least-significant bits of an external instruction address may not be 00 for indirect jumps).

4.2.1 INSTRUCTION PREFETCH BUFFER

All instructions executed by the processor are fetched either from the Branch Target Cache or from external instruction memory (i.e., instruction/data memory or instruction read-only memory). When instructions are fetched from the external memory, they are requested in advance to assist the timing of instruction accesses. The processor attempts to initiate the fetch for any given instruction at least four cycles before it is required for execution.

Since instructions are requested in advance, 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 Instruction Fetch Unit contains a four-word Instruction Prefetch Buffer (IPB), as shown in Figure 4-1. The IPB is a circularly addressed buffer which acts as a first-in/first-out (FIFO) queue for instructions.

If instruction fetching is enabled, the processor requests an external instruction fetch on any cycle for which the IPB contains an available location. Instructions are stored in the IPB as they are returned from the external instruction memory. An instruction is stored into the IPB location whose number is given by bits 3-2 of the instruction address.

The instruction is held in the IPB until it is required for execution. When required, the instruction is sent to the decode stage, and the IPB location is freed to receive a subsequent instruction.

Instruction Prefetch Stream

An instruction prefetch stream is established whenever the processor performs a non-sequential instruction reference. Non-sequential references normally occur as the result of successful branches, but may also result either from the taking of an interrupt or trap (including the *WARN trap) or from an interrupt return.

A non-sequential instruction fetch is initiated by placing an instruction-fetch request on the Address Bus. Once the external instruction fetch has been initiated, the processor generates prefetches for subsequent instructions based on the availability of IPB locations, either by transmitting subsequent addresses, or by issuing burst-mode instruction requests.

The addresses for prefetched instructions are computed by a word-length register called the Instruction Fetch Pointer (IFP), which is maintained by the Instruction Fetch Unit. The IFP latches the physical instruction-address obtained from the Memory Management Unit whenever a non-sequential instruction reference occurs. Then, for instruction prefetches, an 8-bit incrementer associated with the IFP updates bits 9-2 of the IFP to point to sequential instructions in the prefetch stream. The incrementer is limited to eight bits because it increments physical addresses, and thus cannot increment beyond any possible virtual-page boundaries (recall that the minimum virtual

page size is 1 Kbyte). If the incrementer overflows, as indicated by a carry-out, prefetching is preempted. The prefetch stream is later re-established as described below.

The physical address in the IFP is always the address of the most-recently prefetched instruction, even though this address may not appear on the Address Bus for burst-mode fetches. If the burst is externally preempted, the IFP is used to re-establish the burst at the point of preemption.

Instruction Prefetch Buffer States

Four states are associated with each Instruction Prefetch Buffer location. The state-transition diagram for these states is shown in Figure 4-2.

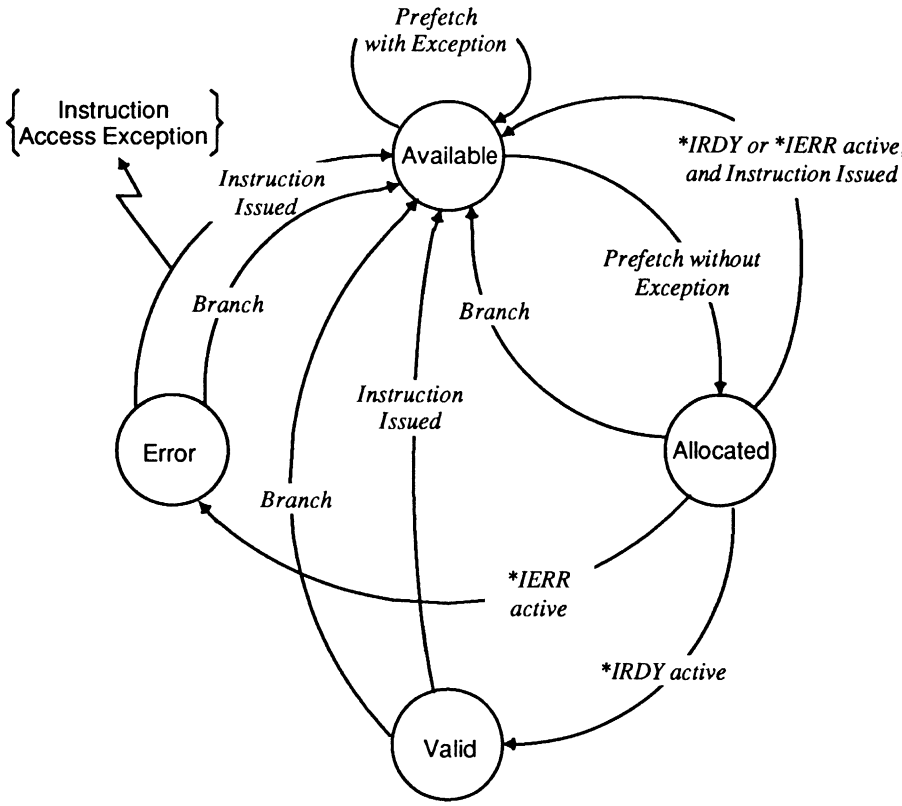


Figure 4-2. IPB State Transitions

Available: The IPB location is free for a new fetch. It contains no valid instruction, and is not due to receive any requested instruction.

Allocated: The IPB location has been scheduled to receive a requested instruction, which has not yet been returned from the external instruction memory.

Valid: The IPB location contains a valid instruction.

Error: The IPB location contains an instruction which was returned from the external memory with an *IERR indication.

If all internal conditions are such that an instruction fetch can occur, the IPB location given by bits 3-2 of the instruction address is set to the Allocated state, and the instruction is requested externally. Once this instruction is returned to the processor, it is stored in the IPB location. The location is set to the Valid or Error state (based on the *IERR input), unless the instruction is sent immediately to the decode stage, in which case the buffer is set to the Available state.

The instruction remains in the buffer until it is required for execution. When the instruction is required, it is issued to the decode stage, and the IPB location is set to the Available state. If the buffer were in the Error state, it is still set to the Available state, but an Instruction Access Exception trap occurs.

It is possible for all IPB locations to be in the Available or Valid states, but only one is allowed to be in the Allocated state at any given time. This restricts the number of unsatisfied instruction prefetches to one, reducing the amount of logic required to keep track of external fetches. It additionally restricts the number of apparent pipeline stages in the external prefetch mechanism to one stage (the other stages involved in the four-stage prefetch pipeline are the request stage and the processor's fetch and decode stages). Larger external prefetch pipelines may be implemented, but they are required to appear as single-stage pipelines; at most one instruction can be returned to the processor from the old instruction prefetch stream after a non-sequential fetch occurs.

When a non-sequential fetch occurs, all buffer locations are set to the Available state during the execute stage of the non-sequential fetch. All instruction requesting for the previous prefetch stream is terminated at this time. There is at most one instruction that will be returned to the processor after instruction fetches are terminated; this instruction is returned before any instruction associated with the new instruction stream is requested externally.

The Error state is provided only to handle errors reported via the *IERR input. However, there are many other situations in which the IPB does not contain a valid instruction. These situations arise because of errors, such as memory-management protection violations, and because instruction fetching is sometimes preempted, such as is the case when the IFP adder overflows. All of these cases are indicated by the fact that the IPB location is in the Available state when the instruction is required for execution (note that the location should, normally, at least be in the Allocated state when the instruction is required).

If the processor requires an instruction from an IPB location that is in the Available state, it initiates the fetch for the instruction using the current value of the Program Counter. This fetch resolves the exceptional condition. It either performs an address translation with the proper address, eliminating page-boundary-crossing problems, or re-creates an error condition, in which case a trap occurs.

4.2.2 BRANCH TARGET CACHE

The Branch Target Cache on the Am29000 allows fast access to instructions fetched non-sequentially. A branch instruction may execute in a single cycle, if the branch target is in the Branch Target Cache.

The target of a non-sequential fetch is in the Branch Target Cache if a similar fetch to the same target has occurred recently enough that it has neither been replaced by the target of another non-sequential fetch, nor invalidated by an INV or IRETINV instruction.

Branch Target Cache Organization

The organization of the Branch Target Cache is shown in Figure 4-3. To improve the ratio of the number of branch targets found in the cache, compared to the number of attempted cache accesses, two-way, set-associative mapping is used.

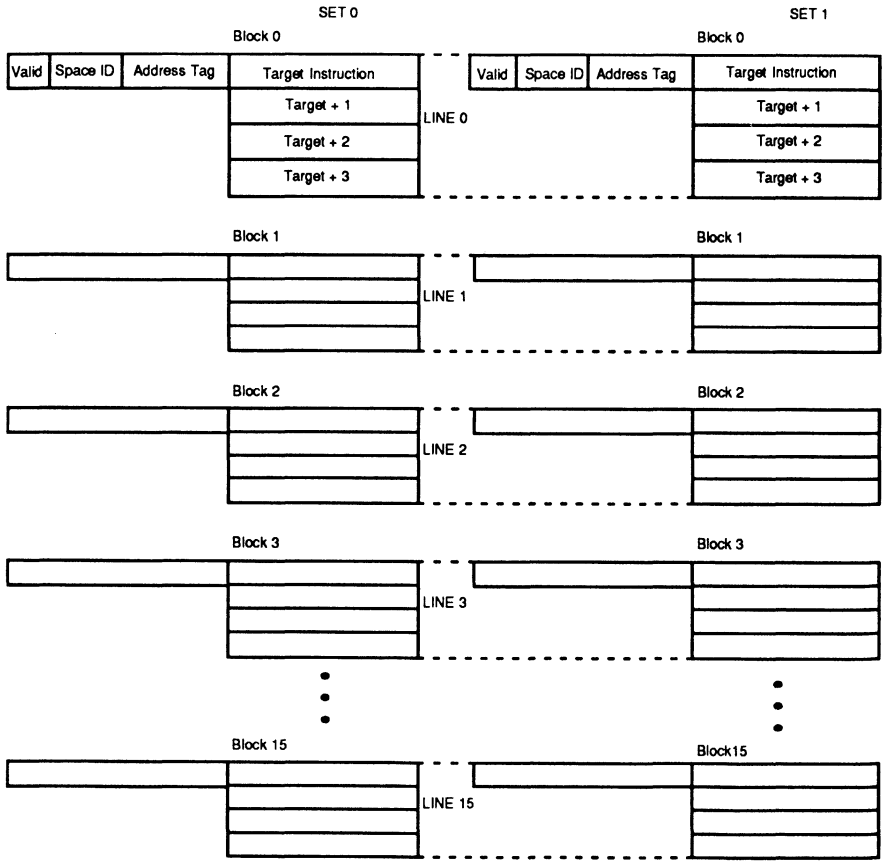


Figure 4-3. Branch Target Cache Organization

The Branch Target Cache is a 512-byte storage array divided into two sets each consisting of 64, 32-bit words (each instruction occupies a word). The sets are further divided into 16 blocks, numbered 0 to 15, which consist of 4 words each. Blocks in different sets with equivalent block numbers are organized into a unit called a line.

To eliminate fragmentation within the Branch Target Cache, each branch target entry is defined as a sequence of exactly four instructions, and is aligned on a cache-block boundary. A branch target sequence may occupy at most one block. This best utilizes the on-chip storage.

A 28-bit cache tag is associated with each four-word block. Of the 28 bits, 26 are derived from the address (possibly virtual) of the instructions in the block, and are called the Address Tag.

Note that the Address Tag is 26 bits in length, rather than 24 bits as might be implied by the organization of the Branch Target Cache. The reason for this is that branch target instruction sequences are aligned on cache-block boundaries, and cache blocks are not aligned with respect to memory addresses. Thus, two additional bits are required in the Address Tag than would be required if cache locations were mapped one-to-one to memory locations.

Three additional bits in the cache tag, called the Space Identification field (Space ID), indicate the instruction memory from which the instructions were fetched (instruction/data or read-only memory), whether the instructions were fetched from a virtual or physical address space, and the program mode under which the instructions were fetched (Supervisor or User). When instructions are placed into the Branch Target Cache, the Space ID bits are written with the values of the following bits of the Current Processor Status Register: ROM Enable (RE), Physical Addressing/Instructions (PI), and Supervisor Mode (SM).

A Valid bit associated with each cache word indicates that the word contains a valid instruction in the branch target sequence. There are thus four Valid bits for each cache block. Cache invalidation instructions make it possible to reset all Valid Bits in a single processor cycle. However, for the Invalidate instruction, the Valid bits are not reset until the next branch is executed.

Branch Target Cache Operation

It is possible to disable the operation of the Branch Target Cache via the Branch Target Cache Disable (CD) bit of the Configuration Register. If the CD bit is 1, all Branch Target Cache entries are made to appear invalid. If the CD bit is 0, there is no effect on Branch Target Cache entries. However, note that a change in the CD bit does not take effect until after the next non-sequential instruction fetch occurs.

When the Branch Target Cache is disabled, it continues to operate as described in this section. However, entries are made to appear invalid, even though they may be valid. If the Branch Target Cache is enabled after a period of being disabled, its contents reflect the most recent instruction execution, and it operates accordingly.

The Branch Target Cache lookup process is diagrammed in Figure 4-4. A given branch target sequence may be contained in one of two cache blocks, where these blocks are in the same line. The sequence is contained in the line whose number is given by bits 5-2 of the address of the first instruction of the sequence. A given branch target sequence is in a given cache block only if the following conditions are met:

1) Bits 31-6 of the address for the first instruction in the sequence match the corresponding bits in the Address Tag associated with the block.

2) The address of the first instruction in the block has a valid translation in the Memory Management Unit, if it is a virtual address.

3) The instruction address space as indicated by the Current Processor Status Register matches the Space ID.

4) The CD bit of the Configuration Register was 0 for the previous non-sequential instruction fetch.

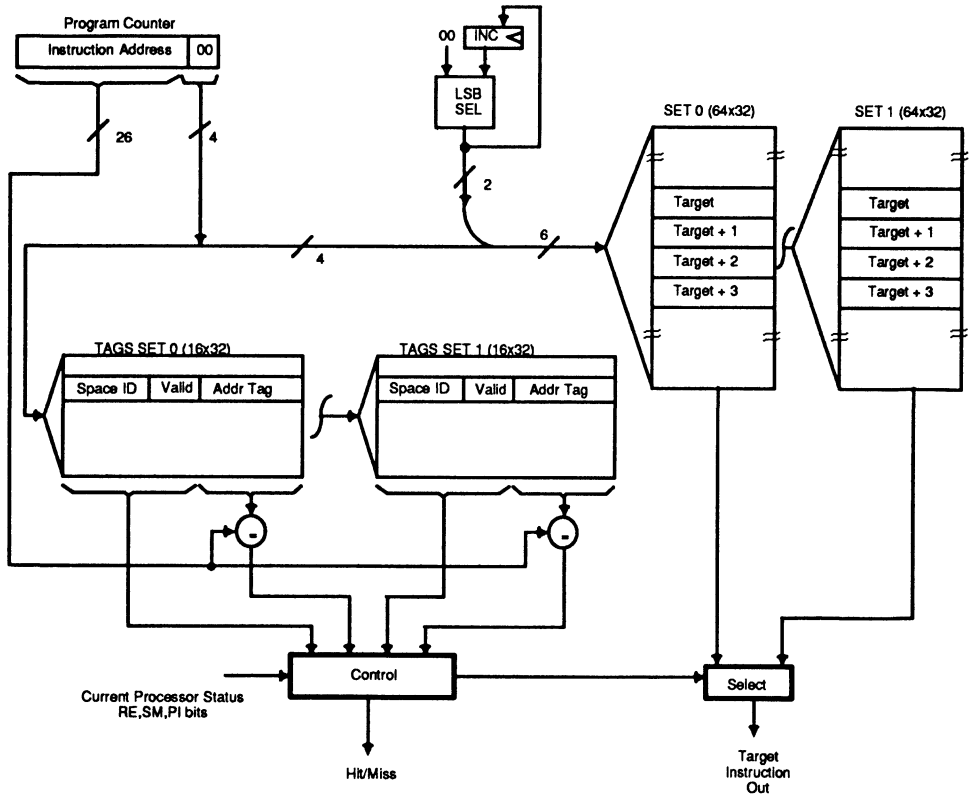


Figure 4-4. Branch Target Cache Lookup Process

In addition to the above requirements, the Valid bit must be 1 for any entry retrieved from the cache. Note that it is not required that all instructions in the sequence be present in the cache for the block to be considered valid.

Whenever a non-sequential fetch occurs (either for a branch instruction, an interrupt or a trap), the address for the fetch is presented to the Branch Target Cache at the same time that the address is translated by the Memory Management Unit. If the target instruction for the non-sequential fetch is in the cache, it is presented for decoding in the next cycle. This instruction is always the first instruction of the cache block, and its address matches the cache tag. Subsequent instructions in the cache are presented for decoding as required in subsequent cycles. However, their addresses do not necessarily match the Address Tag.

Branch Target Cache Replacement

On a non-sequential fetch, if the target instruction is not found in the Branch Target Cache, the address of the fetch selects a line to be used to store the instruction sequence of the new branch target. The replacement block within the line is selected at random, based on the processor clock. Random replacement has slightly better performance than least-recently used replacement, and has a simpler implementation.

To replace the selected entry, all Valid bits associated with the entry are reset, the Address Tag is set with the appropriate address bits of the first instruction in the new sequence, and the Space ID bits are set according to the Current Processor Status Register. Instructions from the new fetch stream are stored into the selected cache block as they are issued to the decode stage. The first instruction is stored into the first word of the block, the second instruction is stored into the second word, and so on up to a maximum of four instructions. The Valid bit for each word is set as the instruction is stored.

Special Cases of Branch Target Cache Entries

If a branch instruction appears as one of the first two instructions in a branch target sequence, the branch is executed before the Branch Target Cache block is filled. In this case, the cache block contains less than four valid instructions. The final valid instruction is the delay instruction of the branch.

When a block is only partially filled due to a branch within the block, the behavior of the cache during subsequent executions of the instructions in the block depends on the outcome of this branch.

If the branch is subsequently successful, then the instructions following the delay instruction of the branch are not needed, and the fact that they are not contained in the cache is irrelevant.

If the branch is subsequently unsuccessful, then the instructions following the delay instruction are required, and must be fetched externally. In this case, a required entry has a Valid bit of 0. When the invalid entry is encountered, the Program Counter is used to create an external instruction fetch for the missing instruction. When the fetch completes, the instruction is stored in the cache location that was previously invalid, and the Valid bit for this entry is set.

Since an instruction sequence in the four-word cache block is not necessarily aligned on a four-word address boundary, a virtual-page address boundary may be crossed for the sequence in the cache. The processor does not prefetch instructions beyond this boundary, so the cache block is only partially filled in this case. If the processor requires instructions beyond the boundary, it creates a fetch for them as described above for the case of a branch instruction in the cache block.

When a fetch is created for a page-boundary crossing, this fetch is treated as a non-sequential fetch; a new cache block is allocated, and the first four instructions at the boundary are placed into the new cache block as they are returned by the instruction memory. Subsequent references to the original cache block also encounter an invalid instruction at the page boundary, and also create a special fetch for this instruction. However, since the instructions beyond this boundary are in the Branch Target Cache, subsequent boundary crossings do not incur the instruction-fetch latency.

4.2.3 NON-SEQUENTIAL INSTRUCTION FETCHES

When a non-sequential instruction fetch occurs, the Memory Management Unit performs an address translation for target instruction, if address translation is enabled. If the address translation is valid, and the target of the fetch is not in the Branch Target Cache, an external instruction fetch is initiated. If there is a Translation Look-Aside Buffer (TLB) miss or memory-protection violation on this address, fetching is not initiated.

Instruction Fetch-Ahead

When a non-sequential fetch occurs, if the target of the fetch is found in the Branch Target Cache, the processor normally begins instruction fetching four instructions beyond the target. This behavior is termed fetch-ahead. The computation required to obtain the address for the fetch-ahead is performed in parallel with address translation, by a 6-bit adder called the Fetch-Ahead Adder (see Figure 4-1).

The Fetch-Ahead Adder is restricted to six bits so that the add cannot cause a page-boundary crossing (recall that the minimum virtual page size is 1 Kbyte and that all instructions are 32 bits in length). If the adder were larger, then the results of the add might affect the outcome of the address translation, and the add could not be performed in parallel with address translation.

Fetch-Ahead Disabling

When the target of a non-sequential fetch is in the Branch Target Cache, there are two cases for which a fetch-ahead is not initiated.

The first case occurs when the Fetch-Ahead Adder overflows during the address computation for the fetch-ahead, as indicated by a carry out of the Fetch-Ahead Adder. Here, a page boundary may have been crossed, making the address translation—which is performed concurrently—invalid.

The second case occurs when the Branch Target Cache block containing the target instruction does not have Valid bits set for all entries within the block. In this case, the processor may have to fetch instructions for these entries, so it does not immediately initiate prefetching beyond the block.

If fetch-ahead is not initiated for an instruction that the processor eventually requires, this fetch is restarted on the cycle in which the missing instruction is required. The Program Counter is used in both of these cases, guaranteeing that the proper instruction address is used.

4.2.4 PROGRAM COUNTER UNIT

The Program Counter Unit, shown in Figure 4-5, 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).

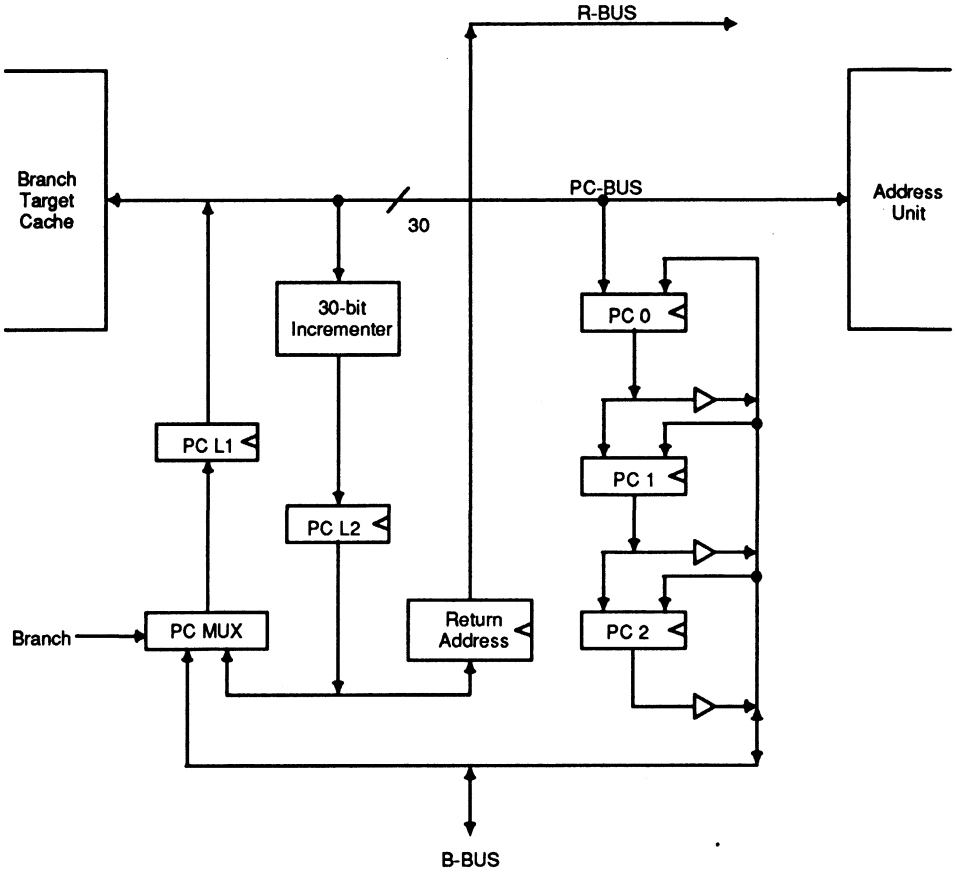


Figure 4-5. Program Counter Unit

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.

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.

4.3 EXECUTION UNIT

The Execution Unit performs most of the operations required for instruction execution. It incorporates the Register File, the Address Unit, the Arithmetic/Logic Unit, the Field Shift Unit, and the Prioritizer.

4.3.1 REGISTER FILE

The general-purpose registers are implemented by a triple-port, 192-location Register File. The Register File performs two read accesses and one write access in a single cycle. If a location is written and read in the same cycle, the data read is that written during the cycle.

The Register Address Generator, shown in Figure 4-6, computes register numbers for operands, detects pipeline data dependencies, and calculates register-number sequences for load-multiple and store-multiple operations.

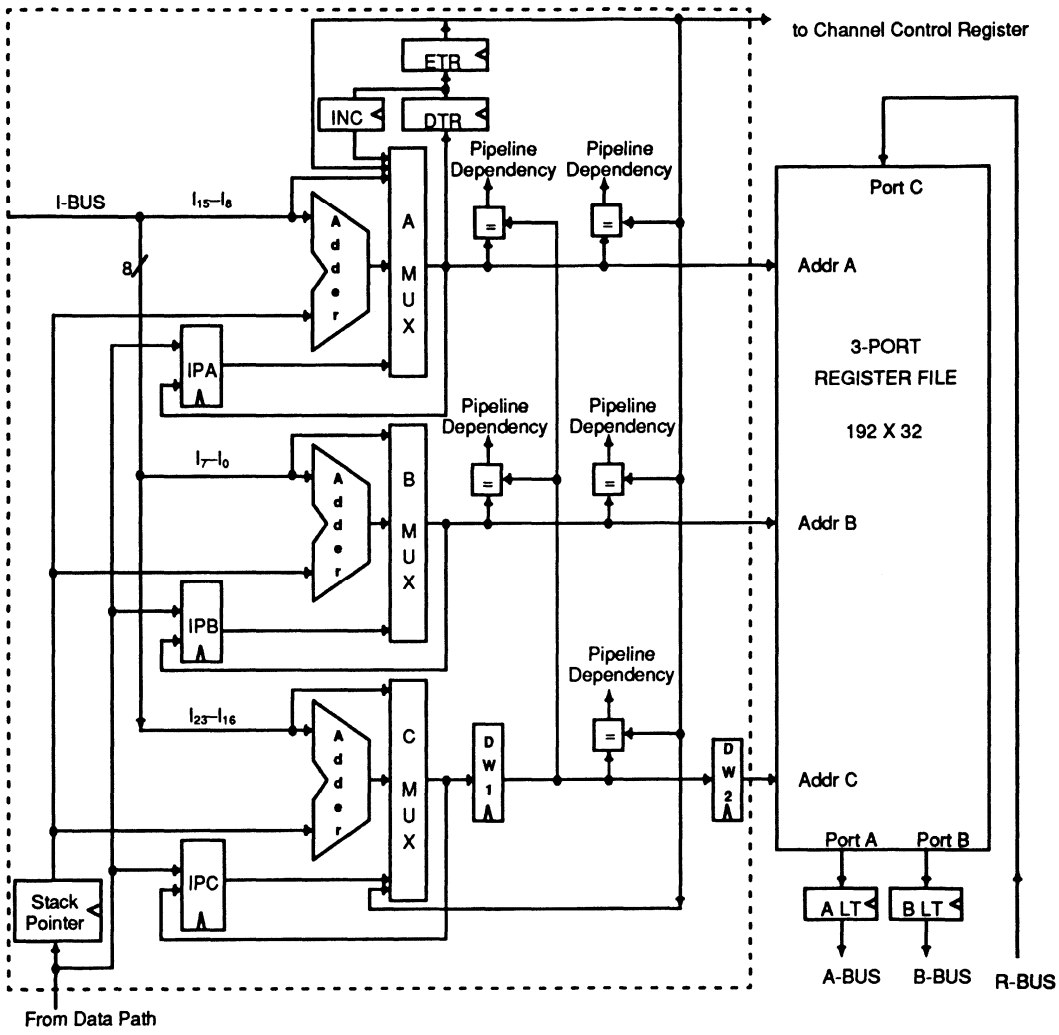


Figure 4-6. Register File and Register Address Generator

Register Addressing

Register numbers for instruction operands are computed during the decode stage. This computation is performed during the first half of a cycle, and the operands are read in the second half of a cycle. Three multiplexers select two source-operand register numbers and a single destination register number for any given instruction.

If the most-significant bit of a register number is 0, the global registers are selected, and the register number is used directly as a register address. If the most-significant bit of the register number is 1, the local registers are selected, and the lower seven bits of the register number are added to the Stack Pointer to form the desired local register address.

The Stack Pointer is a hardware shadow-copy of bits 8-2 of Global Register 1, and is updated whenever Global Register 1 is written with the result of an Arithmetic or Logical instruction. Global Register 1 is implemented as a full 32-bit register in the Register File; this register is distinct from the 192 locations that implement general-purpose registers.

If a register number is zero (i.e., if Global Register 0 is specified as an operand), the Register Address Generator selects the content of an indirect pointer as the register number. There are three indirect pointers, and each appears as a special-purpose register.

Pipeline Data-Dependencies

For the Register File, the pipeline delay in result write-back, compared to operand access, creates situations where a result from a previous operation may be required as an operand before it has been written into the register file. When one of these situations arises, a pipeline data dependency is said to exist.

The register numbers for the write-back of instruction results require two buffering registers, so that they are presented to the Register File during the write-back stage. In addition, the register numbers for uncompleted load operations are held until the load completes (these register numbers are held in the ETR Register shown in Figure 4-6).

Register read-address comparators detect pipeline data dependencies, and activate multiplexers to forward data directly to the required functional unit, without waiting for the data to be written to the register file. The comparators activate the forwarding multiplexers if they detect one of the following situations:

1) One of the source register numbers matches the destination register number of the immediately previous instruction.

2) One of the source register numbers matches the target register number (in the ETR) of an outstanding load.

In the first case listed above, the result of the execute stage is selected as an operand, instead of the output of the Register File port for which the forwarding condition is detected. In the second case, data from the channel is selected. The comparison may cause the processor to enter the Pipeline Hold mode if the load has not completed. However, data forwarding allows data from the Data Bus to be used immediately, in the cycle after it is returned on the Data Bus.

The content of the ETR is further compared to the register numbers supplied to the write-back stage. If the target register for a load is written with the result of an overlapped instruction, the Not Needed (NN) bit in the Channel Control Register is set. If the comparators determine that the NN bit should be set, they also inhibit the write-back of load data on the completion of the load. The NN bit inhibits the restarting of the load operation if an exception occurs.

Load-Multiple and Store-Multiple Sequences

During load-multiple and store-multiple operations, sequential register numbers are computed by an incrementer associated with the ETR/DTR pair shown in Figure 4-6. In the case of store multiple, the register numbers are supplied as read addresses to the Register File by the incrementer. The read addresses are latched by the DTR so that they may be incremented further. In the case of load multiple, target register numbers are held by the ETR as for any other load. However, the ETR is set with a sequence of incremented addresses in this case.

4.3.2 ADDRESS UNIT

The Address Unit, shown in Figure 4-7, computes addresses for branch target instructions, and load-multiple and store-multiple sequences. It also assembles instruction-immediate data and creates addresses for restarting terminated instruction prefetch streams.

The Address Unit consists of a 30-bit adder, the Decode PC Register, the ADRF Latch, and logic for formatting instruction-immediate data and generating the constants zero and one. The Decode PC Register holds the address of the instruction in the decode stage of the pipeline.

Branch Target Addresses

Branch target addresses are either fetched from the Register File or calculated by the Address Unit. The Address Unit calculates target addresses during the decode stage of branch instructions. These addresses are of two possible types:

1) PC relative: the current PC value is added to a sign-extended, 16-bit offset field from the branch instruction.

2) Absolute: a zero-extended, 16-bit field of the branch instruction is used directly as an instruction address.

For each of the above types of addresses, the 16-bit instruction field is aligned on a word address-boundary (i.e., it is shifted left by two bits).

To calculate the branch target address, the Address Unit formats the 16-bit instruction field as required and presents it to the 30-bit adder. This adder adds the formatted field either to the contents of the Decode PC Register or to zero, as required for PC-relative or absolute addresses, respectively.

Load-Multiple and Store-Multiple Addresses

During the execution of Load Multiple and Store Multiple instructions, addresses for the access sequence are held in the ADRF Latch. An address in the ADRF Latch is updated, as required for an access in the sequence, by the 30-bit adder in the Address Unit. The formatting logic creates a constant offset of one for the update. The updated address is presented to the Memory Management

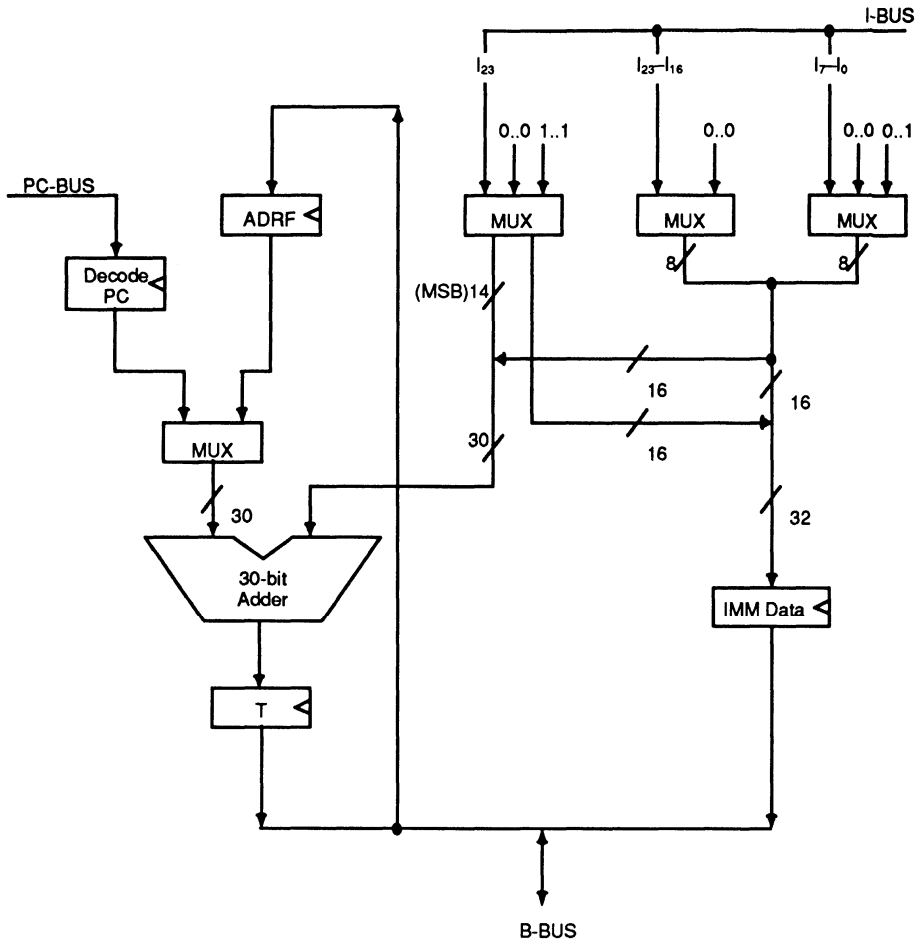


Figure 4-7. Address Unit

Unit for translation and protection checking, and is placed into the ADRF Latch for further address computations.

For load-multiple and store-multiple operations performed using burst-mode accesses, the physical address for each access does not appear on the Address Bus, but the addresses are maintained in the processor so that they may be used to restart the burst-mode access upon preemption.

Special Instruction Fetches

As discussed in Section 4.2, the processor must create special instruction fetches when it encounters an invalid instruction in the middle of a Branch Target Cache block, or when it attempts to fetch an instruction from an Instruction Prefetch Buffer location which is in the Available state. The Address Unit routes the address for this fetch in a manner similar to the routing of a branch target address. It passes the contents of the Decode PC (containing the required instruction address) through the

30-bit adder, adding it to zero. This address is presented to the Memory Management Unit for translation, and is used in the Instruction Fetch Unit to complete the fetch.

4.3.3 ARITHMETIC/LOGIC UNIT

The Arithmetic/Logic Unit (ALU) performs 32-bit arithmetic and logical operations. The arithmetic operations consist of addition, subtraction, addition with carry-in, subtraction with carry-in, and primitives for multiplication and division. Instructions specify whether or not a trap is generated on signed or unsigned arithmetic overflow.

The A and B operands may be complemented independently in the ALU; complementors for data into the ALU are controlled by instructions. This allows subtraction and reverse subtraction to be formed from addition, and allows certain logical operations (e.g., XNOR) to be formed from other basic operations (e.g., XOR). The carry-in to the ALU can be 0, 1, or the value of the Carry bit in the ALU Status Register. The carry-out of the ALU is used in overflow detection, unsigned comparisons, multiplication, and division. The ALU carry-out is stored in the ALU Status Register for multi-precision arithmetic.

The ALU also evaluates the relational expressions equal to, not equal to, less-than, less-than-or-equal-to, greater-than, and greater-than-or-equal-to. Each comparison computes a Boolean corresponding to the relation between two integer operands or creates a trap (possibly) based on this relation. The Boolean constants FALSE and TRUE are represented by a 0 and 1, respectively, in the most-significant bit of a word.

The relational operators may be applied to either signed or unsigned operands. For unsigned operands, these operators are implemented by recognizing that the ALU carry-out is the Boolean result of an unsigned comparison if the two numbers are subtracted and the carry-in is appropriately controlled. For comparison of signed numbers, the true sign of the result (i.e., the resulting sign exclusive-ORed with the overflow indication) gives the result of the compare. The relational operators equal-to and not-equal-to are independent of the data type. These operators are implemented by a 32-bit equal-to-zero comparator.

The ALU also supports the 32-bit logical operations AND, OR, NAND, NOR, AND-NOT, XOR, and XNOR.

4.3.4 FIELD SHIFT UNIT

The Field Shift Unit contains a Funnel Shifter, logic for performing word extracts, and logic for performing byte and half-word extracts and inserts.

The Funnel Shifter performs N-bit shifts, where N is an integer between 0 and 31, inclusive, given by a 5-bit shift count. The source of the shift count is specified by the shift instruction; the shift count is given either by a constant field in the shift instruction, bits 4-0 of a general-purpose register specified by the shift instruction, or by the 5-bit Funnel Shift Count field in the ALU Status Register.

Both arithmetic and logical shifts are supported, with the difference being the values stored into vacated bits: arithmetic shifts fill these bits with the sign bit of the operand, while logical shifts fill them with zero-bits. Arithmetic shifts are possible only for right shifts.

The Field Shift Unit operates on 32-bit words, 16-bit half-words, and 8-bit bytes. For byte operations, the position of a byte operand within a word is supplied by the 2-bit Byte Pointer (BP) field of the ALU Status Register. For half-word operations, the position of a half-word operand is given by the most-significant bit of the BP field; the least-significant bit is ignored. The processor supports either left-to-right or right-to-left byte and half-word ordering within a word.

4.3.5 PRIORITIZER

The prioritizer counts the number of leading zero-bits in an operand. The count of the number of zero-bits up to the leading 1 is stored in the specified destination register. If the operand does not contain a 1, the value stored is 32.

4.4 MEMORY MANAGEMENT UNIT

The Memory Management Unit (MMU) performs all memory-management functions described in Section 3.6. Address translation is performed during the execute stage of any load, store or branch instruction that requires address translation. Address translation also is performed whenever the processor requires an instruction that has not been prefetched; as discussed in Section 4.2, address translation is performed in this case to resolve certain exceptional events that occur during instruction prefetching.

Though the MMU is shared for instruction and data accesses, the processor pipeline is arranged so that there is no contention for the MMU. In general, this is the result of the instruction-set definition and the fact that instruction prefetch addresses are generated by the Instruction Fetch Pointer (see Section 4.2.1).

An instruction address is normally translated only when a branch is executed. Since neither a load nor a store is executed at the same time, there is no contention for the MMU. If the Instruction Fetch Pointer overflows, the address translation is deferred until the Instruction Fetch Unit determines that the processor requires the associated instruction. Since instruction execution cannot occur at this time, the MMU cannot be required for the translation of a load or store address, and again there is no contention.

When the processor performs load-multiple and store-multiple operations, the MMU translates the address associated with every access. Load-multiple and store-multiple address sequencing is performed in the virtual address space, rather than both the virtual and physical address spaces, so that only a single address incrementer is required. Since the execution of Load Multiple and Store Multiple instructions is not overlapped with the execution of other instructions, there is no penalty associated with using the MMU for every access.

The MMU performs address translation in a single cycle. If an address translation is valid, the results of the translation are placed on the Address Bus along with the instruction-access or data-access request. In many cases, the address appears on the Address Bus during the cycle immediately following address translation (it does not appear if the Address Bus is occupied with another access). This address appears regardless of the outcome of memory protection checking; this relaxes the timing constraints on protection checking, which can be performed only after address translation is complete. If a protection violation is detected, the processor activates the *BINV signal late in the first address cycle for the request.

4.5 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 access 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 decodes an instruction which modifies any Translation Look-Aside Buffer entry or special-purpose register, and there is a load or store in progress. This is required for the serialization operation described in Section 3.7.

5) 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.

6) 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.

7) 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 *WARN or *RESET input is asserted.

CHAPTER 5

SYSTEM INTERFACES

This chapter describes the attachment of the Am29000 to its hardware environment. It describes the channel, which allows the processor to communicate with external devices and memories. The Test/Development interface, provided for hardware development and testing, is also described. In addition, this chapter includes sections on external interrupts, traps, processor reset, clock generation, and master/slave checking.

In the signal descriptions of Section 5.1, certain outputs are described as being 3-state or bi-directional outputs. However, all outputs (except MSERR) may be placed in a high-impedance state by the Test mode. The 3-state and bi-directional terminology in this section is for those outputs (except SYSCLK) that are disabled when the processor grants the channel to another master.

5.1 SIGNAL DESCRIPTION

A(31:0) **Address Bus (3-state output, synchronous)**
The Address Bus transfers the byte address for all accesses except burst-mode accesses. For burst-mode accesses, it transfers the address for the first access in the sequence.

***BREQ** **Bus Request (input, synchronous)**
This input allows other masters to arbitrate for control of the processor channel.

***BGRT** **Bus Grant (output, synchronous)**
This output signals to an external master that the processor is relinquishing control of the channel in response to *BREQ.

***BINV** **Bus Invalid (output, synchronous)**
This output indicates that the Address Bus and related controls are invalid. It defines an idle cycle for the channel.

R/*W **Read/Write (3-state output, synchronous)**
This signal indicates whether data is being transferred from the processor to the system, or from the system to the processor.

SUP/*US **Supervisor/User Mode (3-state output, synchronous)**
This output indicates the program mode for an access.

***LOCK** **Lock (3-state output, synchronous)**
This output allows the implementation of various channel and device interlocks. It may be active only for the duration of an access, or active for an extended period of time under control of the Lock bit in the Current Processor Status.

The processor does not relinquish the channel (in response to *BREQ) when *LOCK is active.

MPGM(1:0) **MMU Programmable (3-state output, synchronous)**
These outputs reflect the value of 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.

***PEN** **Pipeline Enable (input, synchronous)**
This signal allows devices that can support pipelined accesses (i.e., that have input latches for the address and required controls) to signal that a second access may begin while the first completes.

I(31:0) **Instruction Bus (input, synchronous)**
The Instruction Bus transfers instructions to the processor.

***IREQ** **Instruction Request (3-state output, synchronous)**
This signal requests an instruction access. When it is active, the address for the access appears on the Address Bus.

IREQT **Instruction Request Type (3-state output, synchronous)**
This signal specifies the address space of an instruction request, when *IREQ is active:

IREQT	Meaning
0	Instruction/data memory access
1	Instruction read-only memory access

***IRDY** **Instruction Ready (input, synchronous)**
This input indicates that a valid instruction is on the Instruction Bus. The processor ignores this signal if there is no pending instruction access.

***IERR** **Instruction Error (input, synchronous)**
This input indicates that an error occurred during the current instruction access. The processor ignores the content of the Instruction Bus, and an Instruction Access Exception trap occurs if the processor attempts to execute the invalid instruction. The processor ignores this signal if there is no pending instruction access.

***IBREQ** **Instruction Burst Request (3-state output, synchronous)**
This signal is used to establish a burst-mode instruction access and to request instruction transfers during a burst-mode instruction access. *IBREQ may be active even though the Address Bus is being used for a data access. This signal becomes valid late in the cycle, with respect to *IREQ.

***IBACK** **Instruction Burst Acknowledge (input, synchronous)**
This input is active whenever a burst-mode instruction access has been established. It may be active even though no instructions currently are being accessed.

***PIA** **Pipelined Instruction Access (3-state output, synchronous)**
If *IREQ is not active, this output indicates that an instruction access is pipelined with another in-progress instruction access. The indicated access cannot complete until the first access is complete. The completion of the first access is signaled by the assertion of *IREQ.

D(31:0) **Data Bus (bi-directional, synchronous)**
The Data Bus transfers data to and from the processor, for load and store operations.

***DREQ** **Data Request (3-state output, synchronous)**
This signal requests a data access. When it is active, the address for the access appears on the Address Bus.

DREQT(1:0) **Data Request Type (3-state output, synchronous)**
These signals specify the address space of a data access, as follows (the value "x" is a don't care):

DREQT1	DREQT0	Meaning
0	0	Instruction/data memory access
0	1	Input/output access
1	x	Coprocessor transfer

An interrupt/trap vector request is indicated as a data-memory read. If required, the system can identify the vector fetch by the STAT(2:0) outputs.

***DRDY** **Data Ready (input, synchronous)**
For loads, this input indicates that valid data is on the Data Bus. For stores, it indicates that the access is complete, and that data need no longer be driven on the Data Bus. The processor ignores this signal if there is no pending data access.

***DERR** **Data Error (input, synchronous)**
This input indicates that an error occurred during the current data access. For a load, the processor ignores the content of the Data Bus. For a store, the access is terminated. In either case, a Data Access Exception trap occurs. The processor ignores this signal if there is no pending data access.

***DBREQ** **Data Burst Request (3-state output, synchronous)**
This signal is used to establish a burst-mode data access and to request data transfers during a burst-mode data access. *DBREQ may be active even though the Address Bus is being used for an instruction access. This signal becomes valid late in the cycle, with respect to *DREQ.

***DBACK** **Data Burst Acknowledge (input, synchronous)**
This input is active whenever a burst-mode data access has been established. It may be active even though no data are currently being accessed.

***PDA** **Pipelined Data Access (3-state output, synchronous)**
If *DREQ is not active, this output indicates that a data access is pipelined with another in-progress data access. The indicated access cannot complete until the first access is complete. The completion of the first access is signaled by the assertion of *DREQ.

OPT(2:0) **Option Control (3-state output, 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 (based on DREQT) are as follows (the value "x" is a don't care):

DREQT1	DREQT0	OPT2	OPT1	OPT0	Meaning
0	x	0	0	0	Word-length access
0	x	0	0	1	Byte access
0	x	0	1	0	Half-word access
0	0	1	0	0	Instruction ROM access (as data)
0	0	1	0	1	Cache control
0	0	1	1	0	ADAPT29K accesses
		-all others-			reserved

During an interrupt/trap vector fetch, the OPT(2:0) signals indicate a word-length access (000). Also, the system should return an entire, aligned word for a read, regardless of the indicated data length.

The Am29000 does not explicitly prevent a store to the instruction ROM.

***CDA** **Coprocessor Data Accept (input, synchronous)**
This signal allows the coprocessor to indicate the acceptance of operands or operation codes. For transfers to the coprocessor, the processor does not expect a *DRDY response; an active level on *CDA performs the function normally performed by *DRDY. *CDA may be active whenever the coprocessor is able to accept transfers.

***WARN** **Warn (input, asynchronous, edge-sensitive)**
A high-to-low transition on this input causes a non-maskable *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).

***INTR(3:0)** **Interrupt Request (input, asynchronous)**
These inputs generate prioritized interrupt requests. The interrupt caused by *INTR0 has the highest priority, and the interrupt caused by *INTR3 has the lowest priority. The interrupt requests are masked in prioritized order by the Interrupt Mask field in the Current Processor Status Register.

***TRAP(1:0)** **Trap Request (input, asynchronous)**
These inputs generate prioritized trap requests. The trap caused by *TRAP0 has the highest priority. These trap requests are disabled by the DA bit of the Current Processor Status Register.

STAT(2:0) **CPU Status (output, synchronous)**
These outputs indicate the state of the processor's execution stage on the previous cycle. 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 (input, asynchronous)**
These inputs control the processor mode:

CNTL1	CNTL0	Mode
0	0	Load Test Instruction
0	1	Step
1	0	Halt
1	1	Normal

***RESET** **Reset (input, asynchronous)**
This input places the processor in the Reset mode.

***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 of processor outputs with the signals provided internally to the off-chip drivers. If there is a difference for any enabled driver, this line is asserted.

SYSCLK **System Clock (bi-directional)**
This is either a clock output with a frequency that is half that of INCLK, or an input from an external clock generator at the processor's operating frequency.

INCLK **Input Clock (input)**
When the processor generates the clock for the system, this is an oscillator input to

the processor, at twice the processor's operating frequency. In systems where the clock is not generated by the processor, this signal must be tied High or Low, except in certain master/slave configurations as discussed in Section 5.8.

The following pins are not signal pins, but are named in Am29000 documentation because of their special role in the processor and system.

PWRCLK **Power Supply for SYSCLK Driver**
This pin is a power supply for the SYSCLK output driver. It isolates the SYSCLK driver, and is used to determine whether or not the Am29000 generates the clock for the system. If power (+5 volts) is applied to this pin, the Am29000 generates a clock on the SYSCLK output. If this pin is grounded, the Am29000 accepts a clock generated by the system on the SYSCLK input.

PIN169 **Alignment pin**
This pin is used to indicate proper pin-alignment of the Am29000. The ADAPT29K uses this pin to communicate its presence to the system.

5.2 CHANNEL DESCRIPTION

The processor channel provides the bandwidth required for performance, while permitting the connection of many different types of devices. This section describes the channel, and methods of connecting devices and memories to the processor.

The channel also is used for transfers to and from the coprocessor. Coprocessor transfers are described in Section 6.2.

Timing diagrams for operations described in this chapter appear in Appendix A.

5.2.1 CHANNEL OVERVIEW

The channel consists of three 32-bit synchronous buses with associated control and status signals: the Address Bus, Data Bus, and Instruction Bus. The Address Bus transfers addresses and control information to devices and memories. The Data Bus transfers data to and from devices and memories. The Instruction Bus transfers instructions to the processor from instruction memories. In addition, a set of signals allow control of the channel to be relinquished to an external master.

There are five logical groups of signals performing five distinct functions, as follows (since some signals perform more than one function, a signal may appear in more than one group):

1) Instruction Address Transfer and Instruction Access Requests: A(31:0), SUP/*US, MPGM(1:0), *PEN, *IREQ, IREQT, *PIA, *BINV.

2) Instruction Transfer: I(31:0), *IBREQ, *IRDY, *IERR, *IBACK.

3) Data Address Transfer and Data Access Requests: A(31:0), R/*W, SUP/*US, *LOCK, MPGM(1:0), *PEN, *DREQ, DREQT(1:0), OPT(2:0), *PDA, *BINV.

4) Data Transfer: D(31:0), *DBREQ, *DRDY, *DERR, *DBACK, *CDA.

5) Arbitration: *BREQ, *BGRT, *BINV.

5.2.2 USER-DEFINED SIGNALS

Two types of user-defined outputs on the processor to control devices and memories directly 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, and have a pre-defined interpretation for coprocessor data transfers.

Standard interpretations of OPT(2:0) are given in Section 5.1. 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.

Note that the standard interpretations of OPT(2:0) apply only to accesses to instruction/data memory and input/output. Other interpretations may be used for coprocessor transfers.

For interrupt and trap vector fetches, the MPGM(1:0) and OPT(2:0) outputs are all Low.

5.2.3 INSTRUCTION ACCESSES

Instruction accesses occur to one of two address spaces: instruction/data memory and instruction read-only memory (instruction ROM). The distinction between these address spaces is made by the IREQT signal, which is in turn derived from the ROM Enable (RE) bit of the Current Processor Status Register. These are truly distinct address spaces; each may be populated independently based on the needs of a particular system.

Instruction/data memory contains both instructions and data. Although the channel supports separate instruction and data memories, the Memory Management Unit does not. In certain systems, it may be required to access instructions via loads and stores, even though instructions may be contained in physically separate memories. For example, this requirement might be imposed because of the need to load instructions into memory. Note also that the OPT(2:0) signals may be used to allow the access of instructions in instruction ROM, using loads; the Am29000 does not prevent a store to the instruction ROM, and protection against stores to the instruction ROM must be provided externally, if required.

All processor instruction fetches are read accesses, and the R/*W signal is High for all instruction fetches.

5.2.4 DATA ACCESSES

Data accesses occur to one of three address spaces: instruction/data memory, input/output (I/O), and the coprocessor. The distinction between these spaces is made by the DREQT(1:0) signals, which are in turn determined by the load or store instruction which initiates a data access. Each of these address spaces is distinct from the others.

The protocol for data transfers to and from the coprocessor is slightly different than the protocol for instruction/data memory and I/O accesses. These transfers are described in Section 6.2.

Data accesses may occur either from a slave device or memory to the processor (for a load), or from the processor to a slave device or memory (for a store). The direction of transfer is determined by the R/*W signal. In the case of a load, the processor requires that data on the Data Bus be held valid only for a short time before the end of a cycle. In the case of a store, the processor drives the Data Bus as soon as the bus is available and holds the data valid until the slave device or memory signals that the access is complete.

5.2.5 REPORTING ERRORS

The successful completion of an instruction access is indicated by an active level on the *IRDY input, and the successful completion of a data access is indicated by an active level on the *DRDY input. If there are exceptional conditions for which an instruction or data access cannot complete successfully, the unsuccessful completion is indicated by an active level on the *IERR or *DERR input, as appropriate.

If the processor receives an *IERR or *DERR in response to an instruction or data access, it ignores the content of the Instruction or Data Bus and the value of *IRDY or *DRDY. An *IERR response causes an Instruction Access Exception trap, unless it is associated with an instruction that the processor does not ultimately execute (because of a non-sequential instruction fetch). A *DERR response always causes either a Data Access Exception trap or a Coprocessor 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.

5.2.6 ACCESS PROTOCOLS

Figure 5-1 shows a control flowchart for accesses performed by the Am29000. This control flow applies independently to both instruction and data accesses. Since the processor performs

concurrent instruction and data accesses, these accesses may be at different points in the control flow at any given point in time.

Note that the items on the flowchart of Figure 5-1 do not represent actual states, and have no particular relationship to processor cycles. The flowchart provides only a high-level understanding of the control flow. Also, exceptions and error conditions are not shown.

The channel supports three protocols for accesses: simple, pipelined, and burst-mode. These are described in the following sections. The various protocols are defined to accommodate minimum-latency accesses as well as maximum-transfer-rate accesses. The protocols allow an access to complete in a single cycle, although they support accesses requiring arbitrary numbers of cycles. Address transfers for accesses may be independent of instruction or data transfers.

5.2.7 SIMPLE ACCESSES

For a simple access, the processor holds the address valid throughout the entire access. This protocol is used for single-cycle accesses, and for accesses to simple devices and memories.

On any cycle before the completion of the access, a simple access may be converted to a pipelined access (by the assertion of *PEN) or to a burst-mode access (by the assertion of *IBACK or *DBACK, if the processor is asserting *IBREQ or *DBREQ). Thus, the protocol for simple accesses also may be used during the initial cycles of pipelined and/or burst-mode accesses. This is advantageous, for example, in cases where the slave device or memory either requires the address to be held for multiple cycles at the beginning of the pipelined or burst-mode access, or cannot respond to the pipelined or burst-mode request within one cycle.

5.2.8 PIPELINED ACCESSES

A pipelined access is one that starts before an earlier in-progress access completes. The in-progress access is called a primary access, and the second access is called a pipelined access. A pipelined access is of the same type as the primary access. For example, an instruction access that begins before the completion of a data access is not considered to be a pipelined access, whereas a second data access is.

The Am29000 allows only one pipelined access at any given time, and does not perform pipelined accesses for the Load Multiple and Store Multiple instructions.

Tradeoffs

For accesses that require more than one cycle to complete, pipelined accesses perform better than simple accesses, because they allow the overlap of portions of two accesses. In addition, the ability to latch addresses in support of pipelined accesses reduces utilization of the Address Bus, thereby reducing contention between instruction and data accesses. However, devices and memories that support pipelined accesses are somewhat more complex than devices and memories that support only simple accesses.

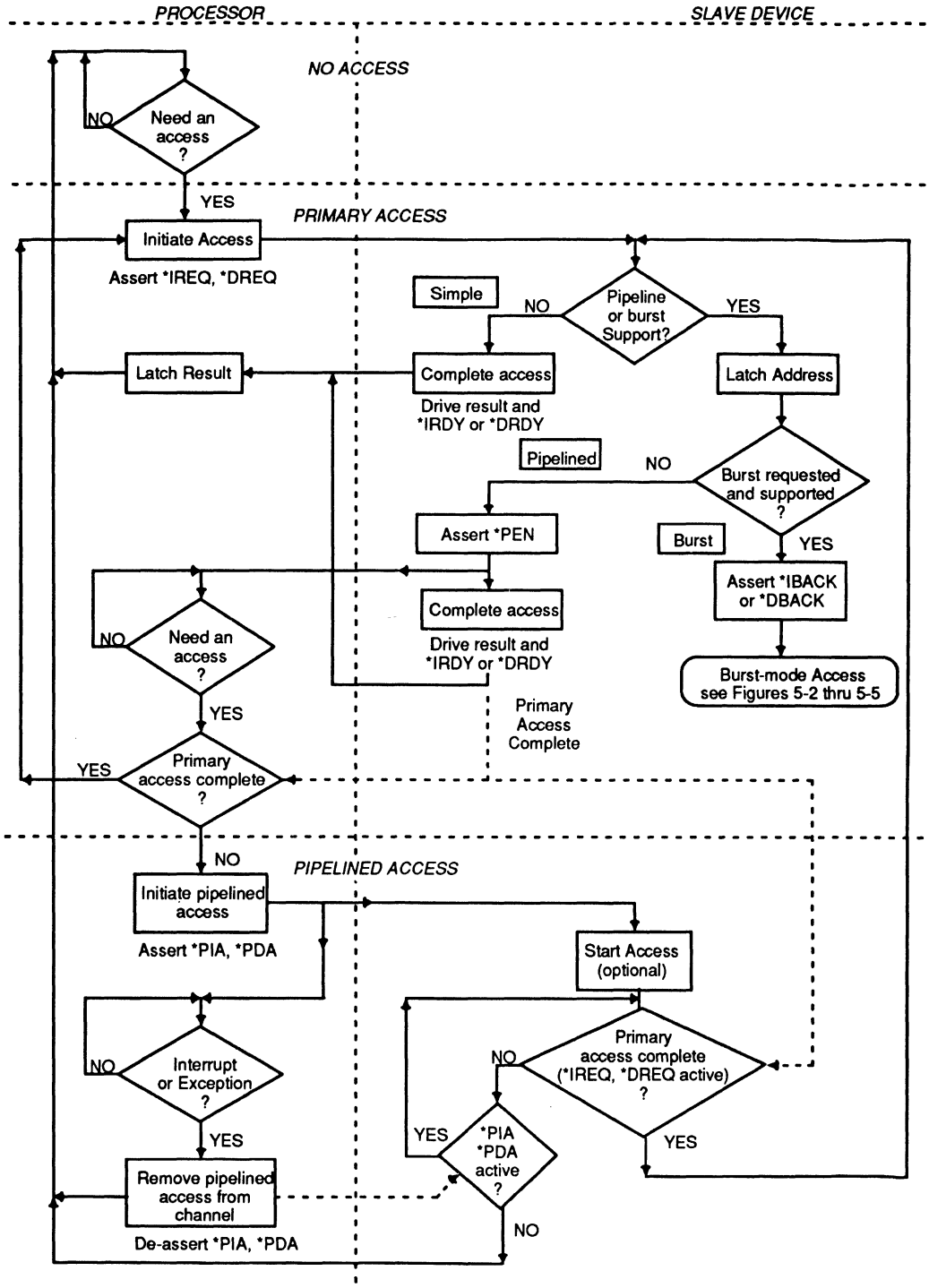


Figure 5-1. Channel Flowchart

Support for pipelined operations is required for both the primary access and the pipelined access. The slave performing the primary access must contain some means for storing the address and other information about the access. The slave performing the pipelined access must be able to restrict its use of the Instruction Bus or Data Bus, and must be prepared to cancel the access (as explained below).

Pipelined Operation

Pipelined accesses are controlled by the signals *PEN, *PIA, and *PDA. Because of internal data-flow constraints, the Am29000 does not perform a pipelined store operation while a load is in progress. However, the protocol does not restrict pipelined operations. Other channel masters may perform a pipelined store during a load.

Except as noted above, the processor attempts to perform pipelining for every access; the input *PEN indicates whether or not pipelining is supported for a given access. The *PEN input can be driven by individual devices, or can be tied active or inactive to enable or disable system-wide pipelined accesses. The processor ignores the value of *PEN unless it is performing an access.

The processor samples *PEN on every cycle during a primary access. If *PEN is active on any cycle, the processor ceases to drive the address and associated controls for the primary access in the next cycle. If the processor requires another access before the primary access completes, it drives the address and controls for the second access, asserting *PIA or *PDA to indicate that the second access is a pipelined access.

The output *IREQ or *DREQ, as appropriate, is not asserted for a pipelined access. Devices and memories that cannot support pipelined accesses should therefore ignore *PIA and/or *PDA, and base their operation upon *IREQ and/or *DREQ.

A device or memory that receives a request for a pipelined access may treat it as any other access, with one exception: the pipelined access cannot use the Instruction and Data buses nor the associated controls (e.g., *IRDY or *DRDY). In the case of a data read or instruction access, the results of the pipelined access cannot be driven on the appropriate bus. In the case of a data write, the data does not appear on the Data Bus. Any other operations for the access, such as address decoding, can occur.

When the primary access completes (as indicated by *IRDY or *DRDY), the pipelined access becomes a primary access. The processor indicates this by asserting *IREQ or *DREQ, depending on the type of access. The device or memory performing the pipelined access may complete the access as soon as *IREQ or *DREQ is asserted (possibly in the same cycle). When the access becomes a primary access, it controls the channel as any other primary access. For example, it may determine whether or not another pipelined access can be performed.

When the pipelined access becomes a primary access, the output *PIA or *PDA remains asserted for one cycle, to insure continuity of control within the slave device or memory. In the cycle after *IREQ or *DREQ is asserted, *PIA or *PDA is de-asserted, unless the processor initiates another pipelined access, in which case *PIA or *PDA remains asserted for the new access.

Cancellation of Pipelined Accesses

If the processor takes an interrupt or trap before a pipelined access becomes a primary access, the request for the pipelined access is removed from the channel. This may occur, for example, when *IERR or *DERR is signaled for the primary access.

If the pipelined access is removed from the channel, the slave device or memory does not receive an *IREQ or *DREQ for the pipelined access. Hence, the pipelined access does not become a primary access, and cannot complete. A pipelined access may be canceled in this manner at any time before it becomes a primary access. Because of this, a pipelined access should not change the state of a slave device or memory until the pipelined access becomes a primary access.

5.2.9 BURST-MODE ACCESSES

A burst-mode access allows multiple instructions or data words at sequential addresses to be accessed with a single address transfer. The number of accesses performed, and the timing of each access within the sequence, is controlled dynamically by the burst-mode protocol. Burst-mode accesses take advantage of sequential addressing patterns, and provide several benefits over simple and pipelined accesses:

1) Simultaneous instruction and data accesses. Burst-mode accesses reduce the utilization of the Address Bus. This is especially important for instruction accesses, which are normally sequential. Burst-mode instruction accesses eliminate most of the address transfers for instructions, allowing the Address Bus to be used for simultaneous data accesses.

2) Faster access times. By eliminating the address-transfer cycle, burst-mode accesses allow addresses to be generated in a manner which improves access times.

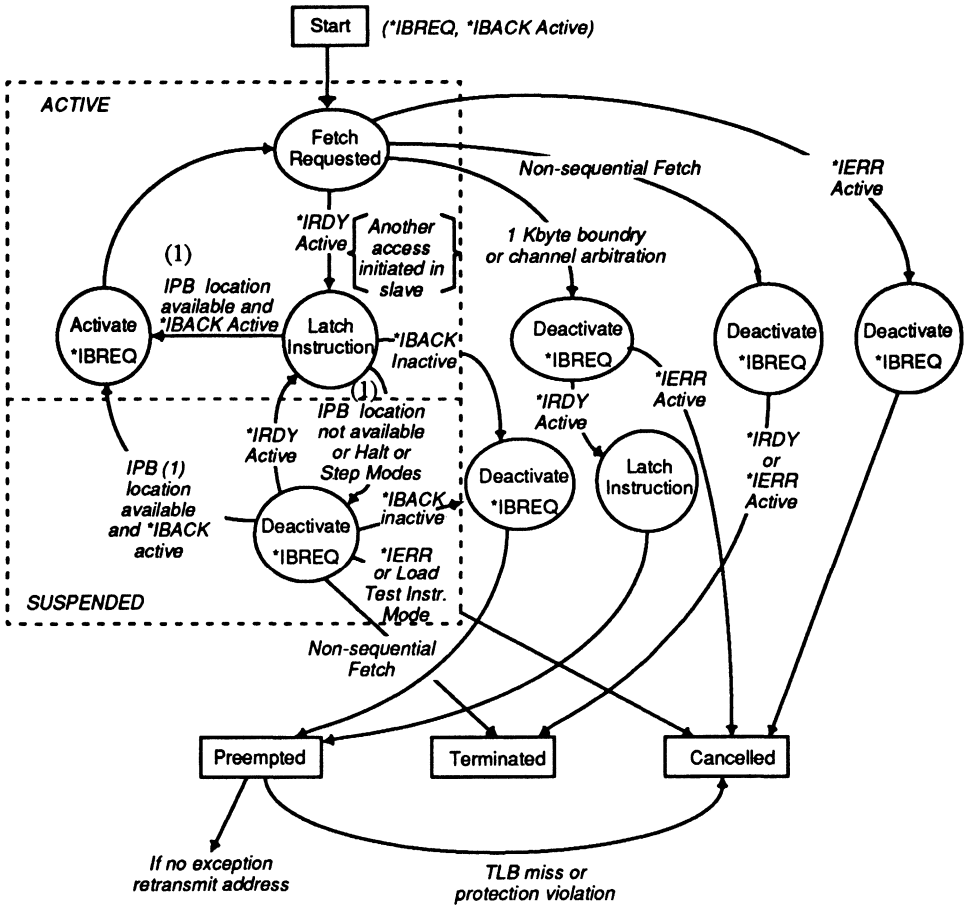
3) Faster memory access modes. Many memories have special high-bandwidth access modes (e.g., static-column page mode and nibble mode). These modes generally require a sequential addressing pattern, even though addresses may not be presented explicitly to the memory for all accesses. Burst-mode accesses allow the use of these access modes, without hardware to detect sequential addressing patterns.

Burst-Mode Overview

The control-flow diagrams in Figure 5-2 and Figure 5-3 illustrate the operation of the processor and an instruction memory during a burst-mode instruction access. The control-flow diagrams in Figure 5-4 and Figure 5-5 illustrate the operation of the processor and a data memory or device during a burst-mode data access. These diagrams are for illustration only: nodes on these diagrams do not necessarily correspond to processor or slave states, and transitions on these diagrams do not necessarily correspond to processor cycles.

A burst-mode access is in one of the following operational conditions at any given time:

1. Established: The processor and slave device have successfully initiated the burst-mode access. A burst-mode access that has been established is either active or



(1) IPB = Instruction Prefetch Buffer

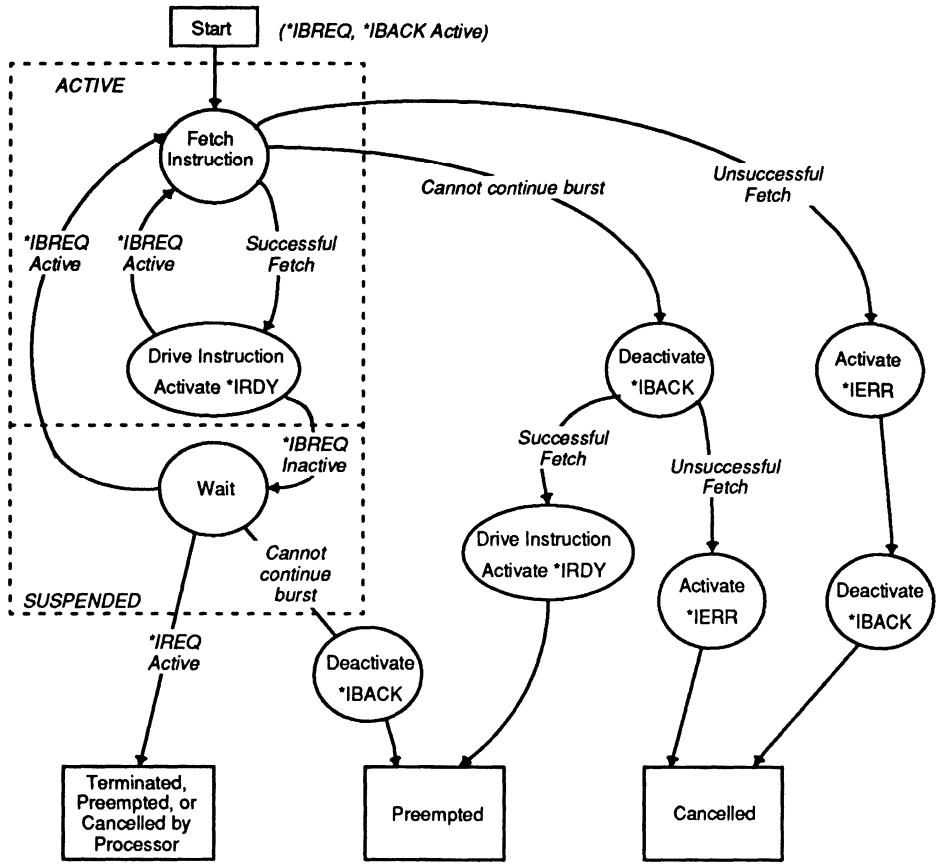
Figure 5-2. Processor Burst-Mode Instruction Accesses: Control Flow

suspended. An established burst-mode access may become preempted, terminated or canceled.

2. Active: Instruction or data accesses and transfers are being performed as the result of the burst-mode access. An active burst-mode access may become suspended.

3. Suspended: No accesses or transfers are being performed as the result of the burst-mode access, but the burst-mode access remains established. Additional accesses and transfers may occur at some later time (i.e., the burst-mode access may become active) without the re-transmission of the address for the access.

4. Preempted: The burst-mode access can no longer continue because of some condition, but the burst-mode access can be re-established within a short amount of time.



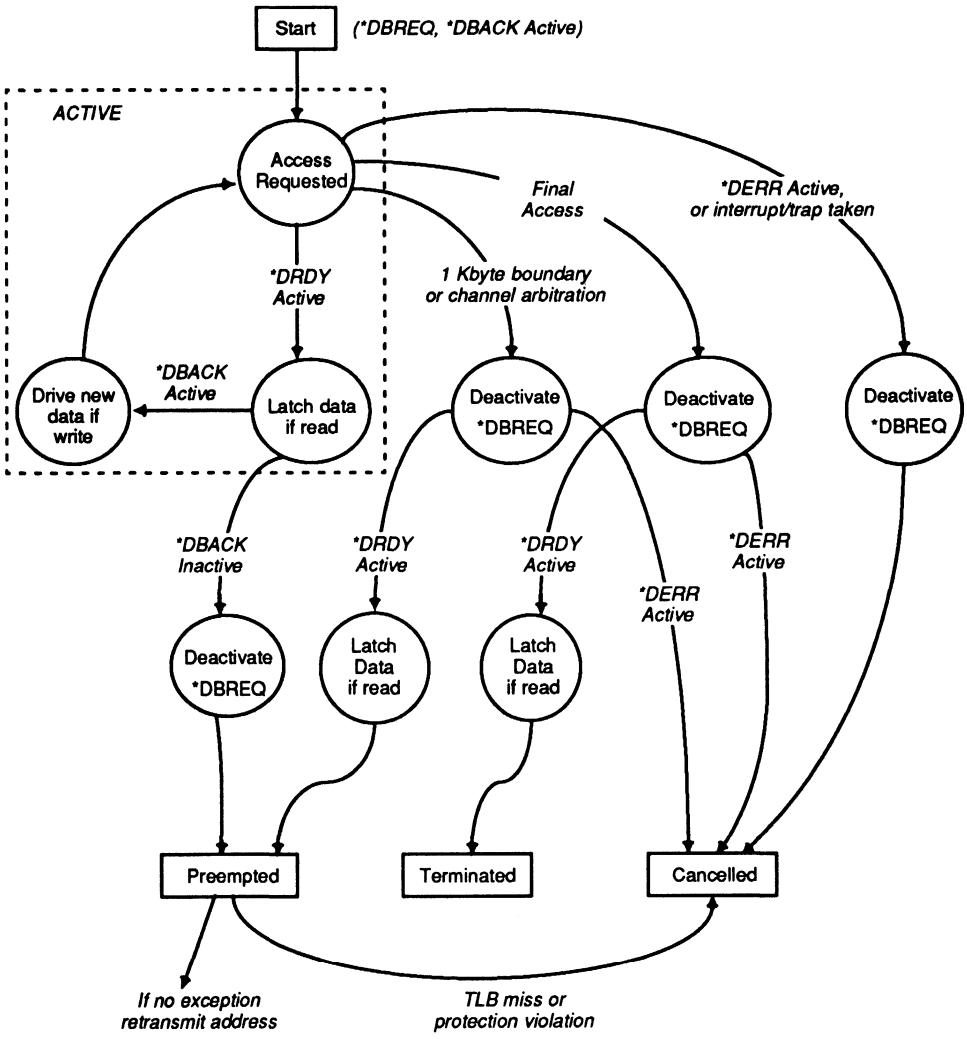
Note: A similar state transition may be used to support suspended burst-mode data accesses or a channel master other than the processor.

Figure 5-3. Slave Burst-Mode Instruction Accesses: Control Flow

5. Terminated: All required accesses have been performed.

6. Canceled: The burst-mode access can no longer continue because of some exceptional condition. The access may be re-established only after the exceptional condition has been corrected, if possible.

Each of the above conditions, except for the terminated condition, is under the control of both the processor and slave device or memory. The terminated condition is determined by the processor, since only the processor can determine that all required accesses have been performed. The following sections discuss each of the above conditions with respect to the burst-mode protocol.



Note: The Am29000 does not suspend burst-mode data accesses.

Figure 5-4. Processor Burst-Mode Data Accesses: Control Flow

Establishing Burst-Mode Accesses

The Am29000 attempts to perform all instruction prefetches using burst-mode accesses, except for instruction fetches at the last word before a 1-Kbyte address boundary. For data accesses, the processor attempts to perform load-multiple and store-multiple operations using burst-mode accesses. The processor indicates that it desires a burst-mode access by asserting *IBREQ or *DBREQ during the cycle in which the initial address is placed on the Address Bus (however, note that these signals become valid later in the cycle than the address).

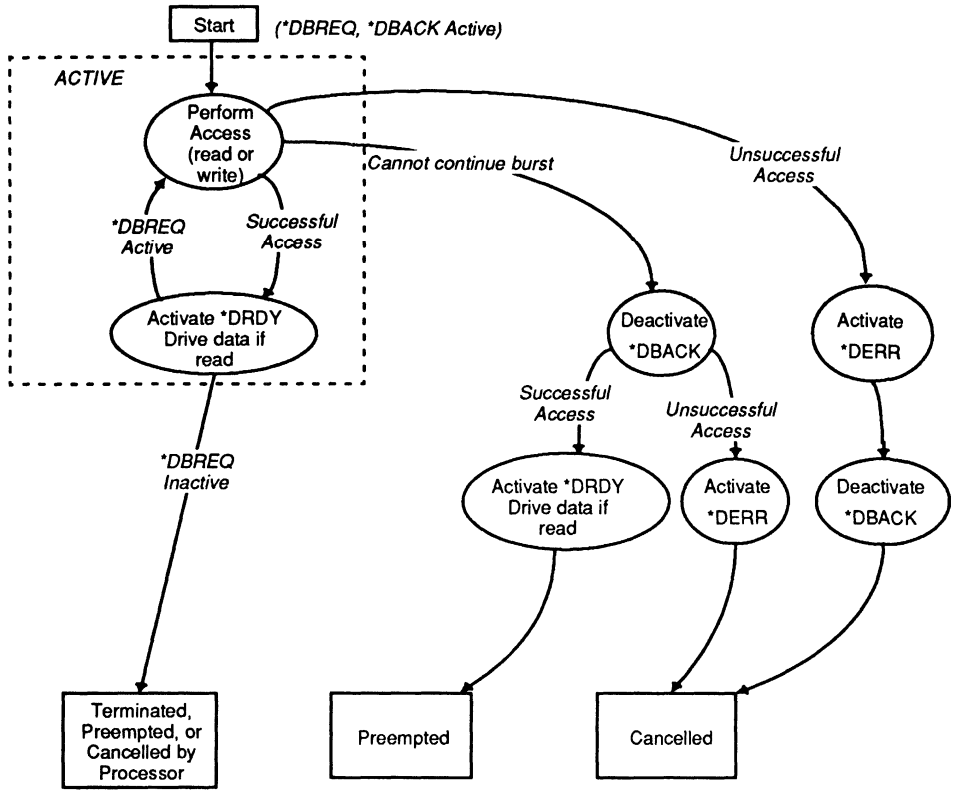


Figure 5-5. Slave Burst-Mode Data Accesses: Control Flow

The inputs *IBACK and *DBACK indicate that a requested burst-mode access is supported. The processor ignores the value of *IBACK unless *IBREQ is asserted, and it ignores the value of *DBACK unless *DBREQ is asserted.

When it desires a burst-mode access, the processor continues to drive *IBREQ or *DBREQ on every cycle for which the address is valid on the Address Bus. During this time, the device or memory involved in the access may assert *IBACK or *DBACK to indicate that it can perform the burst-mode access. If *IBACK or *DBACK (as appropriate) is asserted while the initial address appears on the Address Bus, the burst-mode access is established. In the following cycle, the processor removes the request address and de-asserts *IREQ or *DREQ. However, it continues to assert *IBREQ or *DBREQ. Please note, however, that the processor may de-assert *IBREQ or *DBREQ without receiving an acknowlegement of the requested burst access.

If the burst-mode access is not established on the first access, the processor attempts to establish a burst-mode access on each subsequent address transfer, as long as there are more accesses yet to be performed. During any subsequent access, the addressed device or memory may establish a burst-mode access by asserting *IBACK or *DBACK. If the burst-mode access is never established, the default behavior is to have the processor transmit an address for every access.

Active and Suspended Burst-Mode Accesses

After the burst-mode access is established, *IBREQ and *DBREQ are used during subsequent accesses to indicate that the processor requires at least one more access. If *IBREQ or *DBREQ is active at the end of the cycle in which an access successfully completes (i.e., when *IRDY or *DRDY is active), the processor requires another access. If the slave device or memory previously has not preempted the burst-mode access, and does not preempt (by de-asserting *IBACK or *DBACK) or cancel (by asserting *IERR or *DERR) the burst-mode access in the cycle that the access completes, the additional access must be performed.

The execution rate of instructions is known only dynamically, so that in certain situations, a burst-mode instruction access must be suspended. If *IBREQ is inactive during the cycle in which an instruction access completes, the burst-mode access is suspended (if it is neither preempted nor canceled at the same time). The burst-mode access remains suspended unless the processor requests a new instruction access (in which case *IREQ is asserted), or unless the instruction memory preempts the burst-mode access.

A suspended burst-mode instruction access becomes active whenever the processor can accept more instructions. The processor activates the burst-mode access by asserting *IBREQ. If the instruction memory does not preempt the burst-mode access during this cycle, an instruction access must be performed.

When a suspended burst-mode instruction access is activated, the resulting instruction access is not permitted to complete in the cycle in which *IBREQ is asserted, but may complete in the next cycle. The reason for this restriction is that the burst-mode protocol is defined such that the combination of an active level on *IBREQ and *IRDY causes an instruction access (as previously discussed). If the instruction access completes immediately in the cycle that a suspended burst-mode access is activated, there is an ambiguity in the protocol: it is possible to interpret a single-cycle assertion of *IBREQ as a request for two instructions.

The above ambiguity is resolved by delaying the instruction access resulting from a re-activated burst-mode access for a cycle. Since this restriction applies only when the Instruction Prefetch Buffer is full and the instruction memory is capable of a very fast access, the delayed instruction response has no performance impact.

The Am29000 does not suspend burst-mode data accesses, because the data transfers occur to and from general-purpose registers, which are always available. However, other channel masters may suspend burst-mode data accesses (during direct memory accesses, for example). The principles for suspending burst-mode accesses are the same as those for instruction accesses discussed above.

Processor Preemption, Termination, and Cancellation

The processor may preempt, terminate or cancel a burst-mode access by de-asserting *IBREQ or *DBREQ, and asserting *IREQ or *DREQ at some later point. Normally, the processor receives one more instruction or data word after *IBREQ or *DBREQ is de-asserted. However, this access may complete in the same cycle that *IBREQ or *DBREQ is de-asserted. During the period after

*IBREQ or *DBREQ is de-asserted and before *IREQ or *DREQ is asserted, the burst-mode access is in a suspended condition.

The slave device or memory cannot distinguish between preempted, terminated, and canceled burst-mode accesses, when these are caused by the processor, until the processor asserts *IREQ or *DREQ. If the slave continues to assert *IBACK or *DBACK after *IBREQ or *DBREQ is de-asserted, the slave should be prepared to accept any new request during the cycle that *IREQ or *DREQ is asserted to begin the new access. The reason for this is that the processor may attempt to establish a burst-mode access for the new access: if the slave is asserting *IBACK or *DBACK because of a previously preempted, terminated or canceled burst-mode access, the processor interprets the active *IBACK or *DBACK as establishing the new burst-mode access and removes the request in the following cycle.

The processor preempts a burst-mode access when an external channel master arbitrates for the channel, or when a burst-mode fetch crosses a potential virtual-page boundary. Since the minimum page size is 1 Kbyte, burst-mode instruction and data accesses are preempted whenever the address sequence crosses a 1-Kbyte address boundary. The burst 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.

Note that the preemption resulting from page boundaries is advantageous for devices or memories that require counters to follow the burst-mode address sequence. Since all burst-mode accesses are word accesses, and the processor re-transmits an address at every 1-Kbyte address boundary, an 8-bit counter in the slave device or memory is sufficient to follow the burst-mode address sequence. Additional address bits are simply latched.

The processor terminates a burst-mode access whenever all required instructions or data have been accessed. In the case of instruction accesses, the burst-mode access is terminated when a non-sequential fetch occurs. In the case of data accesses, the burst-mode access is terminated when the count indicates a single load or store remains. The last load or store is executed as a simple access.

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 when a Translation Look-Aside Buffer miss occurs on an address in the burst-mode sequence. 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 access is restarted at the point at which it was canceled, rather than at the beginning of the original address sequence.

Slave Preemption and Cancellation

The slave device or memory involved in a burst-mode access may preempt the access by de-asserting *IBACK or *DBACK. The processor samples *IBACK and *DBACK when *IRDY

and *DRDY are active, so that *IBACK and *DBACK may be de-asserted as the last supported access is completed. However, *IBACK and *DBACK also may be de-asserted in any cycle before the access completes; to preempt the access, *IBACK or *DBACK must remain inactive until *IRDY or *DRDY is asserted. If *IBACK or *DBACK is de-asserted when the processor is in a state where it expects an access, the access must be completed.

In general, the slave device or memory preempts the burst-mode access whenever it cannot support any further accesses in the burst-mode sequence. This normally occurs whenever an implementation-dependent address boundary is encountered (e.g., a cache-block boundary), but may occur for any reason. By preempting the burst-mode access, the slave receives a new request, with the address of the next instruction or data word required by the processor.

The slave device or memory may cancel a burst-mode access by asserting *IERR or *DERR in response to a requested access. The signals *IBACK or *DBACK need not be de-asserted at this time, but should be de-asserted in the next cycle.

Note that the *IERR and *DERR signals cause non-maskable traps, except in the case where *IERR is asserted for an instruction which the processor does not execute.

5.2.10 ARBITRATION

External masters can gain access to the Address, Data, and Instruction buses by asserting the *BREQ input. The processor completes any pending access, preempts any burst-mode access, and asserts the *BGRT output. At this time, the processor places all channel outputs associated with the Address, Data, and Instruction buses in the high-impedance state.

For the first cycle that *BGRT is asserted, the output *BINV is also asserted. If the external master cannot control the Address Bus and associated controls in the cycle that *BGRT is asserted, the active level on *BINV may be used to define an idle cycle for the channel (i.e. any spurious access requests are ignored). The *BINV signal is asserted only for a single cycle, so the external master must take control of the channel in the cycle after *BGRT is asserted.

While the *BREQ input remains asserted, the processor continues to assert *BGRT. The external master has control over the channel during this time.

To release the channel to the processor, the external master de-asserts *BREQ, but must continue to control the channel for the first cycle in which *BREQ is de-asserted. In the cycle after *BREQ is de-asserted, the processor asserts *BINV and de-asserts *BGRT; the external master should release control of the channel at this time. On the following cycle, the processor de-asserts *BINV, and is able to use the channel. The processor re-establishes any burst-mode access preempted by arbitration.

The processor does not relinquish the channel when the *LOCK signal is active. This prevents external masters from interfering with exclusive accesses.

5.2.11 USE OF *BINV TO CANCEL AN ACCESS

Besides using the *BINV signal to transfer control of the channel from one master to another, the Am29000 uses the *BINV signal to cancel accesses after they have been initiated. To cancel an

access, *BINV is asserted during a cycle in which *IREQ or *DREQ also is asserted. If an access is canceled, the accompanying response (using *IRDY, *IERR, *DRDY or *DERR) is ignored during the cycle that *BINV is asserted; thereafter, the system should not respond to the canceled access.

The *BINV signal is used to cancel an instruction access in the following situations:

- when an interrupt or trap is taken,
- when an instruction fetch-ahead is canceled because a target block is only partially present in the Branch Target Cache,
- when an instruction TLB miss or protection violation occurs on an instruction access,
- when a branch instruction is the delay instruction of another branch, and the targets of both branches are in the Branch Target Cache (in this case, the external fetch for the target of the first branch is not required), and
- when the processor enters the Load Test Instruction Mode, and there is an active instruction request on the channel.

The *BINV signal is used to cancel a data access in the following situations:

- when a data TLB miss or protection violation occurs on the data access, and
- when an interrupt or trap is taken in the cycle that a data access appears on the channel.

When a LOADSET instruction encounters a protection violation because store access is not permitted, the processor cancels the load access with *BINV.

5.2.12 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, if 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 electro-magnetic emissions.

The Am29000 channel is defined to prevent all situations where multiple drivers are driving a bus simultaneously. However, bus collisions may be allowed to occur, depending on the system design.

In the case of the Am29000 channel, arbitration for the channel prevents the processor from driving the Address and Data buses at the same time as another channel master. If there is more than one external master, the system design must include some means for insuring that only one external master gains control of the channel, and that no external master gains control of the channel at the same time as the processor.

When the processor relinquishes control of the channel to an external master, bus collisions may be prevented by not allowing the external master to drive any bus while *BINV is active. This insures that all processor outputs are disabled by the time the external master takes control of the channel. However, there is nothing in the channel protocol to prevent the external master from taking control as soon as *BGRT is asserted.

Slave devices and memories are prevented from simultaneously driving the Instruction Bus or Data Bus by allowing only the device or memory performing a primary access to drive the appropriate bus. When a pipelined access becomes a primary access, it may drive the Instruction or Data Bus immediately, so that there is a potential bus collision if the pipelined access is performed by a slave other than the slave performing the original primary access. This bus collision may be prevented by restricting all slaves to driving the Instruction and Data buses in the second half-cycle (using SYSCLK, for example). Since the processor samples data only at the end of a cycle, this restriction does not affect performance.

When the processor performs a store immediately following a load, it drives the Data Bus for the store in the second cycle following the cycle in which the data for the load appears on the 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 Data Bus until it receives a *DRDY or *DERR in response to the store; it ceases to drives the Data Bus in the cycle following the response.

5.2.13 CHANNEL BEHAVIOR FOR INTERRUPTS AND TRAPS

If an interrupt or trap is taken, any burst-mode accesses are canceled. If a request for a pipelined access is on the Address Bus, this request is removed. Any other accesses are completed, and no new accesses are started, other than those required for the interrupt or trap. Note that any accesses that the processor expects to complete must be completed, even though burst-mode and pipelined accesses are canceled.

When interrupt or trap processing is complete, any canceled burst-mode accesses transactions are re-established, using the address of the access that was to be performed next when the interrupt or trap was taken. Uncompleted pipelined accesses are restarted, either by the interrupt return sequence in the case of an instruction access, or by restarting the initiating instruction in the case of a data access.

Note that the restarting of a pipelined access is not performed by the Channel Address, Channel Data, and Channel Control registers, since these registers may be required to restart the primary access. The instruction initiating the pipelined access is not allowed to complete until the primary access completes, so that the Program Counter 1 (PC1) Register contains the address of the initiating instruction when a pipelined access is canceled. The address in PC1 can restart this instruction on interrupt return.

5.2.14 EFFECT OF THE *LOCK OUTPUT

The *LOCK output provides synchronization and exclusion of accesses in a multi-processor environment. *LOCK has no pre-defined effect for a system, other than the fact that the Am29000 does not grant the channel to an external master while *LOCK is active.

The *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. *LOCK may also 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).

*LOCK may be defined to provide any level of resource locking for a particular system. For example, it may lock the channel, 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.

5.3 TEST/DEVELOPMENT INTERFACE

The Test/Development Interface consists of the inputs CNTL(1:0) and *TEST, and the outputs STAT(2:0). The CNTL(1:0) inputs provide control of processor operation, and the STAT(2:0) outputs provide information about processor operation for external monitoring.

An ADAPT29K or other hardware-development system uses CNTL(1:0) and STAT(2:0) to control the processor for the purposes of processor and system debug.

A hardware tester uses the *TEST input to place all processor outputs in the high-impedance state. This allows the tester to check other system logic by driving processor outputs directly, without requiring that the processor be removed from the system.

5.3.1 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 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

On any given cycle, the STAT(2:0) signals reflect the state 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).

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. 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 *WARN trap (see Section 3.5.6 and Section 5.6).

5.3.2 CPU CONTROL INPUTS

Certain processor operational modes are under the control of the CNTL(1:0) inputs. These inputs have an effect on 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 5-6. The restriction on CNTL(1:0) transitions 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.

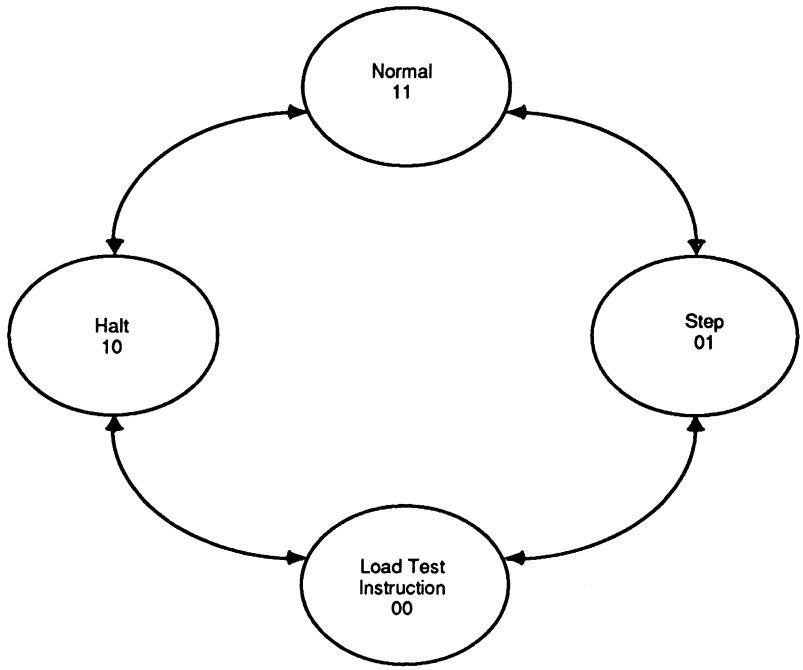


Figure 5-6. Valid Transitions on CNTL(1:0) Inputs

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.

5.3.3 HARDWARE DEVELOPMENT

The Halt, Step, and Load Test Instruction modes of operation are defined to support the debug of the processor system (both hardware and software) by a hardware-development system. 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.

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 is invoked by the application of 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 also is entered as the result of the execution of 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 *RESET inputs. If the instruction following the 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 disables protection checking for Halt instructions until the next processor reset.

In most cases, the STAT(2:0) lines 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 the Halt mode. However, the STAT(2:0) lines have a value 010 if the FZ bit of the Current Processor Status Register is 1 when the Halt mode is entered. This indicates that visible processor registers do not reflect the current program state.

If a burst-mode instruction access is established before the processor enters the Halt mode, it remains established when the processor enters the Halt mode, but is suspended.

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, and to the value 11, which causes the processor to resume normal execution.

Step Mode

The Step mode causes the Am29000 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) lines 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 the Step mode. However, the STAT(2:0) lines have a value 010 if the FZ bit of the Current Processor Status Register is 1 when the Step mode is entered. This indicates that visible processor registers do not reflect the current program state.

If a burst-mode instruction access is established before the processor enters the Step mode, it remains established when the processor enters the Step mode, but is suspended.

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 *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 Am29000 is placed in halt or step mode while either a LOADM or STOREM instruction is being executed, the STAT (2:0) lines indicate a halt/step mode (000) for one cycle and then indicate pipeline hold (001) for the number of cycles needed to complete the last access of the LOADM/STOREM before returning to halt/step mode (000). This is because the last access of a LOADM/STOREM is always performed as a simple access. A hardware-development system must therefore ignore any single-cycle halt/step indication of the STAT (2:0) lines as an indication that the processor is halted.

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 Fetch Unit. 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), except that it first preempts any established burst-mode instruction access. The Load Test Instruction mode can be entered only from the Halt or Step modes. Note that the burst-mode instruction access that is preempted here was previously suspended for the Halt or Step modes.

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 continually is storing the value on the Instruction 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 Bus. The value of *IRDY and *IERR 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 sub-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 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 Am29000 to ignore the *DRDY and *DERR responses for the access; the Am29000 completes the access at the end of the next stepped instruction, rather than upon the assertion of *DRDY. This eliminate the need for the hardware-development system to generate a synchronous *DRDY 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 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.

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 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).

5.3.4 ADAPT29K SYSTEM DESIGN CONSIDERATIONS

The ADAPT29K hardware-development system uses the Test/Development Interface to provide nearly all of the capabilities of an in-circuit emulator. To accommodate use of an ADAPT29K for hardware and/or software development, there are several guidelines that must be followed in the design of the system. These guidelines are listed below:

1) The Am29000's alignment pin (PIN169) is driven Low by the ADAPT29K to indicate that it is performing an operation. As an option, this pin can be used by the system to determine that an ADAPT29K operation is in progress. Suggested uses of PIN169 are given below.

2) It must be possible to place the system instruction memory in the high-impedance state. This allows the ADAPT29K to use the Load Test Instruction mode of the Am29000. To support this requirement, the system can either keep the Instruction Bus in the high-impedance state when there is no active instruction request (*IREQ and *IBREQ are both inactive) or can place the Instruction Bus in the high-impedance state when PIN169 is Low.

3) The data memory must ignore Am29000 data transfers to and from the ADAPT29K and must keep the Data Bus in a high-impedance state during these transfers. ADAPT29K transfers are indicated in two ways: by a *DREQ with DREQT(1:0) being 00 and OPT(2:0) being 110, and by a Low level on PIN169.

4) The ADAPT29K must be able to drive the *RESET and CNTL(2:0) signals without interference from the system. The system should provide pull-up resistors on these signals to provide an inactive level when the ADAPT29K is not driving them.

5) The layout of the Am29000 in the system should allow space for the ADAPT29K cable on the side of the Am29000 with pins A1–U1. The ADAPT29K cable has the same outline as the Am29000 and has all signals exiting on the A1–U1 row of pins. Obstructions such as the lever of the Am29000 socket or high-profile components should not be placed on this side of the Am29000.

6) The instruction and data memories should be mapped to a common address space. The ADAPT29K sets instruction breakpoints, downloads code, and displays instruction memory by accessing instruction memory via the Data Bus.

7) If the contents of the instruction read-only memory (ROM) are to be displayed by the ADAPT29K, the ADAPT29K must be able to access the instruction ROM via the Data Bus, using OPT(2:0) = 100. If breakpoints are to be set in instruction ROM, the ADAPT29K must be able to write the instruction ROM via the Data Bus (this capability may be provided only during the development of ROM-based code).

8) The ADAPT29K does not participate in channel arbitration (BREQ, BGRT). Other devices connected to the channel should not request the channel when the ADAPT29K is performing an operation.

9) The *CDA signal should be pulled down by a 33K to 68K ohm resistor. The ADAPT29K detects the presence or absence of the Am29027 Arithmetic Unit during initialization. If the Am29027 is not present and *CDA is not pulled down, the ADAPT29K will not operate correctly.

5.3.5 HARDWARE TESTING

The Test mode in the Am29000 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 *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 SYSCLK.

For some outputs, the transition to the high-impedance state that results from the Test mode may occur at a much slower rate than applies during normal system operation (for example, when the processor relinquishes the channel to another master). For this reason, the Test mode may not be appropriate for special user-defined purposes.

Note that SYSCLK 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 SYSCLK output does not have an adverse effect on the system. Furthermore, if SYSCLK is disabled, and a signal is not externally supplied, processor state may be lost.

5.4 EXTERNAL INTERRUPTS AND TRAPS

An external device causes an interrupt by asserting one of the *INTR(3:0) inputs, and causes a trap by asserting one of the *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 *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 *INTR0 has the highest priority, and the interrupt caused by *INTR3 has the lowest priority.

The interrupts caused by *INTR(3:0) may be masked by the Disable Interrupts (DI) or Disable All Interrupts and Traps (DA) bits of the Current Processor Status Register. In addition, the Interrupt Mask (IM) field of the Current Processor Status Register sets the priority of the processor with respect to these inputs. The IM field enables the *INTR(3:0) inputs as follows:

IM Value	Result
00	*INTR0 enabled
01	*INTR(1:0) enabled
10	*INTR(2:0) enabled
11	*INTR(3:0) enabled

Note that the interrupt caused by the *INTR0 input cannot be disabled by the IM field.

If one of the *INTR(3:0) inputs is active, and the resulting interrupt is disabled by the DA bit, DI bit or IM field, the Interrupt Pending (IP) bit of the Current Processor Status Register is set. The IP bit is reset if the interrupt is enabled, or if all disabled external interrupts are de-asserted.

The *TRAP(1:0) inputs are prioritized with respect to each other, so that the trap caused by *TRAP0 has priority over the trap caused by *TRAP1 when a conflict occurs. Both *TRAP0 and *TRAP1 have priority over the *INTR(3:0) inputs. The *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 *INTR(3:0) and *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 *INTR(3:0) and *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.

There is a three-cycle latency from the de-assertion of an *INTR(3:0) or *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.

5.5 PROCESSOR RESET

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 pre-defined state (see Section 3.8).

The Reset mode is invoked by asserting the *RESET input, and can be entered only if the SYSCLK pin is operating normally, whether or not the SYSCLK pin is being driven by the processor (see Section 5.7). The Reset mode is entered within four processor cycles after *RESET is asserted. The *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 (e.g., the Reset mode can be entered from the Halt mode). If the *RESET input is asserted at the time that power is first applied to the processor, the processor enters the Reset mode only after four cycles have occurred on the SYSCLK pin.

The Reset mode is exited when the *RESET input is de-asserted. Either three or four cycles after *RESET is de-asserted (depending on internal synchronization time), the processor performs an initial instruction access on the channel. The initial instruction access is directed to address 0 in the instruction read-only memory (instruction ROM). If instruction ROM is not implemented in a particular system, another device or memory must respond to this instruction fetch.

If the CNTL(1:0) inputs are 10 or 01 when *RESET is de–asserted, the processor enters the Halt or Step mode, respectively. If the processor enters the Halt mode immediately 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. The Load Test Instruction mode cannot be directly entered from the Reset mode. If the CNTL(1:0) inputs are 00 immediately after *RESET is de-asserted, the effect on processor operation is unpredictable. If the CNTL(1:0) inputs are 11, the processor enters the Executing mode.

The processor samples the STAT0 output internally when *RESET is asserted. A High level on STAT0 in this case is used to enable a special test configuration, and causes the processor to be inoperable. When *RESET is asserted, the processor drives STAT0 Low in order to disable this test configuration. However, if processor outputs are disabled by the Test mode, the processor is not able to drive STAT0. Thus, if *RESET is asserted when the processor is in the Test mode, the STAT0 pin must be driven Low externally. (In a master/slave configuration, as described in Section 5.8, STAT0 is driven Low by the master processor when *RESET is asserted).

5.6 *WARN INPUT

An inactive-to-active transition on the *WARN input causes a *WARN trap to be taken by the processor. The *WARN trap cannot be disabled; the processor responds to the *WARN input regardless of its internal condition, unless the *RESET input also is asserted. This 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 *WARN input is edge-sensitive, so that an active level on the *WARN input for long intervals does not cause the processor to take multiple *WARN traps. However, *WARN must be held active for at least 4 cycles in order to be properly recognized by the processor. The processor still takes the

*WARN trap if *WARN is de-asserted after four cycles. Another *WARN trap occurs if *WARN makes another inactive-to-active transition.

The processor enters the Executing mode when the *WARN input is asserted, regardless of its previous operational mode. Either seven or eight cycles after *WARN is asserted (depending on internal synchronization time), the processor performs a trap-handler instruction access on the channel. This instruction access is directed to address 16 in the instruction read-only memory (instruction ROM). If instruction ROM is not implemented in a particular system, another device or memory must respond to this instruction fetch.

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 *WARN trap is taken. If the CNTL(1:0) inputs are 00 immediately after *WARN is de-asserted, the effect on processor operation is unpredictable. If the CNTL(1:0) inputs are 11, the processor remains in the Executing mode.

5.7 CLOCKS

The Am29000 supports two methods of system-clock generation and distribution. In one arrangement, the processor generates a clock for the system at its operating frequency; this clock appears on the SYSCLK pin, and may be distributed externally to other system components. In the second arrangement, the system provides its own clock generation and distribution; in this case, the processor receives the externally generated clock on the SYSCLK pin.

In both arrangements, the circuits that generate and buffer SYSCLK are designed to minimize the apparent skew between internal processor clocks and external system clocks.

The processor provides a power-supply pin named PWRCLK for the SYSCLK driver that is independent of all other chip power distribution. The separate PWRCLK supply electrically isolates other processor circuits from noise which might be induced on the power supply by the SYSCLK driver. The PWRCLK pin also is used to decide between the two possible clocking arrangements.

5.7.1 PROCESSOR-GENERATED CLOCK

If power (i.e., +5 volts) is applied to the PWRCLK pin, the processor is configured to generate clocks for the system. In this case, the SYSCLK pin is an output, and the signal on INCLK is used to generate the system clock. The processor divides the INCLK signal by two in the generation of SYSCLK, so INCLK should be driven at twice the processor's operating frequency.

5.7.2 SYSTEM-GENERATED CLOCK

If the PWRCLK pin is grounded, the processor is configured to receive an externally generated clock. In this case, the SYSCLK pin is an input used directly as the processor clock. SYSCLK should be driven at the processor's operating frequency. In this configuration, the INCLK input should be tied High or Low, except in certain master/slave configurations as discussed in Section 5.8.

5.7.3 CLOCK SYNCHRONIZATION

The SYSCLK pin is at a High level during the first half of the processor cycle, and at a Low level during the second half of the processor cycle. Thus, a processor cycle begins on a Low-to-High transition of SYSCLK. The definition of the beginning of the processor cycle is independent of the clocking arrangement chosen for a particular system.

In some systems, it might be desirable to have two or more processors operate in lock-step synchronization, with each processor driven by a common INCLK signal. In this case, synchronization of the processors is achieved by the *RESET input. If the de-assertion of *RESET meets a specified set-up time with respect to the High-to-Low transition of INCLK, the SYSCLK output is guaranteed to be Low after the second following rising edge of INCLK. Thus, all processors may be synchronized as required.

5.7.4 ELECTRICAL SPECIFICATIONS

The electrical specifications for SYSCLK are different than the specifications for most other processor inputs and outputs. In order to reduce clock-skew effects, the SYSCLK pin is electrically compatible with the processor's CMOS circuits, rather than being compatible with transistor-transistor-logic (TTL) circuits.

Note that the SYSCLK pin is placed in the high-impedance state by the Test mode. If an externally generated clock is not supplied in this case, processor state may be lost.

5.8 MASTER/SLAVE CHECKING

Each Am29000 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.

5.8.1 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 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 channel, 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.

5.8.2 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 Am29000 design:

1) Unimplemented bits in processor registers that are reflected on processor outputs. This is avoided in the Am29000 design by having all unimplemented bits be read as 0.

2) Unpredictable values for channel signals. If a *DERR or *IERR response is asserted in response to an access, the Data Bus or Instruction 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 Am29000 avoids this problem by ignoring the instruction or data word returned with *DERR or *IERR.

3) Unpredictable power-up state that is reflected on processor outputs. The Am29000 avoids 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 SYSCLK, with the slave processor receiving SYSCLK as an input, or by driving both processors' SYSCLK inputs with the same externally generated clock.

However, the fact that both processors operate with the same 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., *WARN, *INTR(3:0), *TRAP(1:0), CNTL(1:0) and *RESET) must be externally synchronized to both the master and slave processors. Note that in the case of *RESET, only the active-to-inactive transition must be synchronized.

5.8.3 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 SYSCLK in this case, the slave processor also must generate SYSCLK when it becomes a master. Because of this, the INCLK signal must be supplied to both the master and slave processors, with both processors being configured to generate clocks.

In this master/slave configuration, the slave processor still receives SYSCLK from the master processor as described previously. The slave processor does not drive SYSCLK because of the Test mode. However, when the slave processor is taken out of the Test mode, it is able to drive SYSCLK as required.

Note that this processor-switching scheme may be generalized to more than two processors.

CHAPTER 6

COPROCESSOR INTERFACE

A coprocessor for the Am29000 is an off-chip extension of the processor's execution unit. The Am29000 communicates with the coprocessor using a mechanism that is very similar to the mechanism used to communicate with other external devices and memories. However, because the coprocessor extends the instruction-execution capabilities of the processor, transfers to and from the coprocessor are in terms of operands, operation codes, results, and status information. This is in contrast to address and data transfers that occur for other types of external accesses. This chapter describes the coprocessor interface, both from a software and a hardware point of view.

6.1 COPROCESSOR PROGRAMMING

6.1.1 OVERVIEW OF COPROCESSOR OPERATIONS

A program executes the following steps to perform a coprocessor operation. This sequence is intended only as a guide, since there are many possible variations:

1) Send operands to the coprocessor. The number of transfers to the coprocessor depends on the number of operands, and the length of each operand. As many as 64 bits of information can be transferred in a single cycle.

2) Send an operation code and other operation information to the coprocessor. The operation can be specified by as many as 64 bits of information.

3) Start the coprocessor operation. This can occur simultaneously with the operation-code transfer of step 2.

4) Read the coprocessor results. The number of transfers from the coprocessor depends on the number of results, and the length of each result.

The above sequence is defined so that coprocessor operations may be concurrent with other processor operations, including external accesses. This is possible because coprocessor operations are decoupled from the transfer of information to and from the coprocessor. Once the operation is started, in step 3, the processor may continue further execution, overlapped with coprocessor execution, until the coprocessor results are read.

Because the Am29000 implements overlapped loads, it can continue execution after attempting to read a coprocessor result. However, if the processor attempts to use the result before the operation is complete, the processor enters the Pipeline Hold mode until the operation is complete.

In certain circumstances, it may be desired to perform multiple coprocessor operations before any results are read. For example, certain array computations form a single result from more than one operation. In this case, steps 1 through 3 above may be repeated—in any combination desired and as many times as desired—before results are read. The coprocessor interface allows the coprocessor to prevent the transfer of operands and/or operation codes if it is not prepared to receive them.

6.1.2 COPROCESSOR TRANSFERS

All coprocessor transfers occur between general-purpose registers and the coprocessor. The transfers occur as the result of the execution of load and store instructions for which the Coprocessor Enable (CE) bit has a value 1. For a store, the information transferred to the coprocessor is given either by the contents of two general-purpose registers, or by the contents of a general-purpose register and an 8-bit constant. For a load, information is transferred into a single general-purpose register in the Am29000.

The coprocessor model includes no provision for addressing. Although it is possible to extend the coprocessor interface to include addressing, addressing is more appropriately handled by normal external accesses defined for the processor (such as input/output).

The format of the instructions that transfer information to and from a coprocessor is shown in Figure 6-1.

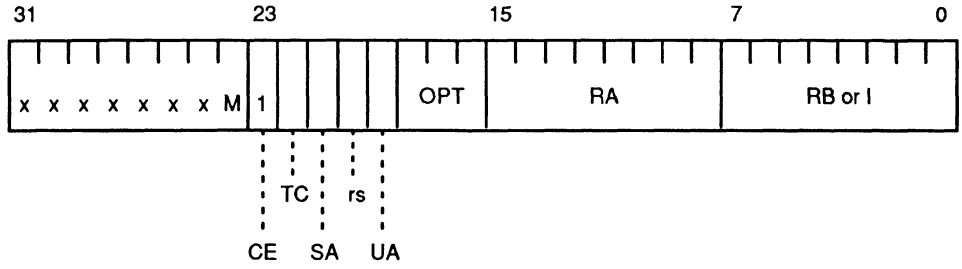


Figure 6-1. Coprocessor Load/Store Format

For coprocessor stores, the RA and "RB or I" fields specify the source of data to be transferred to the coprocessor. The RA field specifies a general-purpose register whose contents are transferred to the coprocessor. The "RB or I" field specifies either a general-purpose register whose contents are transferred to the coprocessor, or a zero-extended constant that is transferred to the coprocessor. For the latter, the M bit of the operation code (bit 24) determines whether the register or the constant is used, as with most instructions. Note that as many as 64 bits of information may be transferred to the coprocessor by a single store instruction.

For coprocessor loads, the data transferred from the coprocessor is written to the general-purpose register given by RA; the "RB or I" field is unused in this case (however, the contents of the specified register, or the zero-extended constant, appears on the Address Bus). In contrast to the coprocessor store, a load transfers only 32 bits of information from the coprocessor.

Other bits in the coprocessor load and store instructions are defined as follows:

Bit 22 : Transfer Control (TC)—This bit affects the behavior of the coprocessor for the transfer, depending on whether the transfer is for a load or store. The definition of this bit is by convention only, and is not enforced by the processor.

For transfers to the coprocessor (i.e., stores), a value of 1 for the TC bit causes a coprocessor operation to start. For transfers from the coprocessor (i.e., loads) a value of 1 for the TC bit causes the coprocessor to suppress exception reporting. In either case, a value of 0 for the TC bit has no special effect on the coprocessor.

Bit 21 : Set Coprocessor Active (SA)—This bit is provided to signal the beginning and end of a coprocessor operation, so that the proper action may be taken by software if the operation is interrupted.

An SA bit of 1 affects the Coprocessor Active (CA) bit in the Current Processor Status. If the SA bit is 1 for a store, the CA bit is set. If the SA bit is 1 for a load, the CA bit is reset. If the SA bit is 0, there is no effect on the CA bit.

Bit 20 : reserved.

Bit 19 : User Access (UA)—The UA bit allows programs executing in the Supervisor mode to emulate User-mode coprocessor transfers. This allows checking of the authorization of a transfer requested by a User-mode program. Note that this checking is performed externally, since the processor imposes no restriction on User-mode coprocessor transfers.

If the UA bit is 1, the coprocessor transfer is performed in the User mode, regardless of the value of the Supervisor Mode (SM) bit in the Current Processor Status. In this case, the User mode affects only the SUP/*US output; 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 transfer is controlled by the SM bit.

Bits 18-16 : Option (OPT)—The OPT field is placed on the OPT(2:0) outputs during the coprocessor transfer. 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 bits define the quantities being transferred to or from the coprocessor. For example, they can specify whether operands or operation codes are being transferred. The interpretation of the OPT field depends on the definition of a given coprocessor.

The transfer of data to or from the coprocessor may be caused by any load or store instruction defined for the processor; the operation of coprocessor transfers is very similar to the operation of external accesses.

Coprocessor transfers are overlapped with the execution of instructions that sequentially follow the coprocessor load or store instruction. However, only one load or store may be in progress in any given cycle, whether or not the load or store is directed to a coprocessor. The pipeline interlocks that apply to external accesses also apply to coprocessor transfers, except that coprocessor-transfer interlocks are determined by the time taken by the coprocessor to perform an operation, rather the time taken to perform an access.

Note that coprocessor transfers may be performed by Load Multiple and Store Multiple instructions. However, register RB has no defined interpretation for a Store Multiple to the coprocessor. For this

reason, Store Multiple is defined to transfer multiple, 32-bit quantities to the coprocessor. Similarly, a Load Multiple transfers multiple, 32-bit quantities from the coprocessor. Note, however, that the incrementing address sequence defined for Load Multiple and Store Multiple still appears on the Address Bus for coprocessor transfers.

6.1.3 COPROCESSOR EXCEPTIONS

A Coprocessor Exception trap occurs if the coprocessor reports an exception (using the *DERR signal) during a coprocessor transfer. The Coprocessor Exception may occur either for a coprocessor load or store.

In the case of a load that reads a coprocessor result, the Coprocessor Exception can be used to indicate that the result is incorrect because of some exceptional condition. In some cases, the Am29000 might be able to correct the results of the operation.

In the case of a store to the coprocessor, the Coprocessor Exception can be used to indicate that the coprocessor cannot accept the transfer because of some exceptional condition. For example, it may indicate an error in a stream of calculations, where intermediate results are not being read. As with a load, the Am29000 may be able to correct the exceptional condition.

As noted above, the trap handler that executes as the result of the Coprocessor Exception trap may attempt to correct the exceptional condition. In many cases, the trap handler must be able to read the intermediate results of the operation from the coprocessor, along with other information about the operation. When this information is read, it may be necessary to suppress further exception reporting, so that the trap handler does not create additional Coprocessor Exception traps. For this reason, the TC bit in the coprocessor load or store instruction allows the processor to read coprocessor results while suppressing exception reporting.

Additionally, the TC bit allows a program to read the result of a coprocessor operation regardless of any errors that may have occurred. This provides an optional trapping capability analogous to that provided for certain Am29000 arithmetic operations (e.g., Am29000 instructions allow an optional trap on arithmetic overflow).

6.1.4 COPROCESSOR AS A SYSTEM OPTION

When the coprocessor is a system option, coprocessor operations are performed by the processor when the coprocessor is not present.

The coprocessor may be designed as a system option by use of the Coprocessor Present (CP) bit of the Configuration Register. The CP bit is set during system initialization, based on the presence (CP = 1) or absence (CP = 0) of the coprocessor. If the CP bit is 0 when the processor attempts to execute a coprocessor load or store instruction, a Coprocessor Not Present trap occurs.

When a Coprocessor Not Present trap is taken, the Channel Address, Channel Data, and Channel Control registers contain information related to the coprocessor transfer. This information may be used by the trap handler to emulate the operation of the coprocessor.

6.1.5 INTERRUPTED COPROCESSOR OPERATIONS

The Coprocessor Active (CA) bit of the Current Processor Status Register may be used to indicate the duration of a coprocessor operation. The value 1 in the CA bit indicates that the coprocessor has begun an operation that has not completed (i.e., the final results have not been read).

The CA bit is affected by the Set Coprocessor Active (SA) bit in the coprocessor load and store instructions. If the SA bit is 1 for a store, the CA bit is set; if the SA bit is 1 for a load, the CA bit is reset. The routine that accesses the coprocessor is responsible for setting and resetting the CA bit appropriately.

If an interrupt or trap is taken during a coprocessor operation, and the CA bit has been properly managed, the CA bit of the Old Processor Status signals to an interrupt or trap handler that the interrupted routine had begun a coprocessor operation, but had not completed the operation before the interrupt or trap was taken. In this case, the coprocessor contains state information that must be preserved. This information may be saved and restored across the interrupt or trap, or, alternatively, kept in the coprocessor.

Upon an interrupt or trap, the state information contained in the coprocessor depends on both the operation being performed and the definition of the coprocessor. The methods used to determine what state information must be saved, and the methods used to transfer this information, are also dependent on the definition of the coprocessor.

Due to interrupt-latency considerations, it may be desirable to leave state information in the coprocessor upon interrupt, rather than require that it always be saved. A problem arises, however, when a routine other than the one that was originally interrupted attempts to use the coprocessor. The coprocessor may be protected from such use by resetting the CP bit in the Configuration Register. If another routine attempts to use the coprocessor in this case, a Coprocessor Not Present trap occurs. The trap handler for this trap may either save the coprocessor state and make the coprocessor available to the trapping routine, or return control to the routine that was originally using the coprocessor.

Certain coprocessor operations may not be interruptible. For these operations, interrupts may be disabled by the Disable Interrupts (DI) and/or Disable All Interrupts and Traps (DA) bits in the Current Processor Status Register. However, this disabling can be performed only by a program in the Supervisor mode. Any User-mode programs that perform non-interruptible coprocessor operations incur the overhead of a call to a Supervisor-mode program.

6.2 COPROCESSOR ATTACHMENT

Communication with the coprocessor occurs via the Am29000 channel. Figure 6-2 illustrates a typical coprocessor connection. For transfers to the coprocessor, 64 bits of data are transferred in a single cycle, using the Address Bus and Data Bus simultaneously. For transfers from the coprocessor, 32 bits of data are transferred in a cycle, using the Data Bus.

The width of transfers to the coprocessor is greater than the width of transfers from the coprocessor because the Am29000 is optimized for computations performed on two word-length operands, with a single word-length result. The operand/result data flow of the processor is reflected in the interface to the coprocessor.

The protocol for coprocessor transfers is nearly identical to the protocol for other external accesses on the channel. Minor differences result from the fact that there are no addresses for coprocessor transfers, and from the fact that the coprocessor is operation-oriented, rather than access-oriented.

6.2.1 SIGNAL DESCRIPTION

Coprocessor transfers are indicated on the channel by the DREQT1 output being High during a request. The DREQT0 output also affects the transfer, based on the R/*W signal, as follows:

R/*W	DREQT1	DREQT0	Meaning
0	1	0	Transfer to coprocessor
0	1	1	Transfer to coprocessor, start operation
1	1	0	Transfer from coprocessor
1	1	1	Transfer from coprocessor, suppress errors

Note that the interpretation of DREQT0 during a coprocessor transfer is by convention only.

The only signal unique to coprocessor transfers is the *CDA input. The coprocessor de-asserts this signal whenever it can accept no transfers from the processor (normally, this is because it is performing an operation).

The completion of a transfer to the coprocessor is indicated when the coprocessor asserts *CDA. The input *DRDY is not used in this case. The performance of transfers to the coprocessor is enhanced by the use of *CDA, since it eliminates the need for the coprocessor to decode a transfer request and respond with *DRDY and thereby eliminates the logic delay involved. Note that the coprocessor normally de-asserts *CDA when it starts an operation, so that *CDA can be independent of transfer requests.

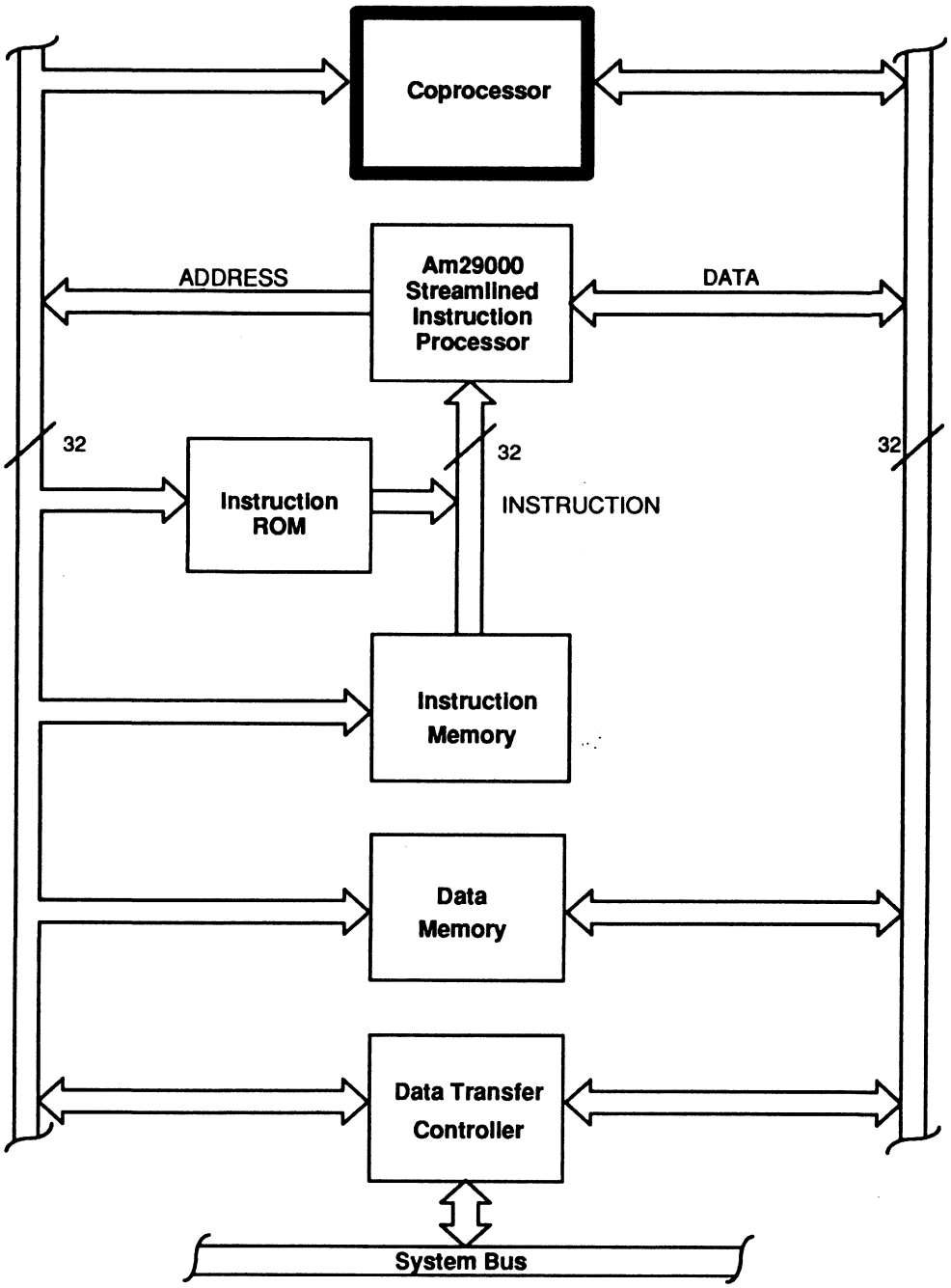


Figure 6-2. Coprocessor Attachment

6.2.2 COPROCESSOR COMMUNICATION

The Address Bus is used to transfer information to the coprocessor. Therefore, the addressing function of other devices and memories on the channel must be disabled during coprocessor transfers. Since DREQT1 is High for all coprocessor transfers, it should be used to inhibit the address-decoding function of channel devices and memories, as well as to indicate to the coprocessor that a transfer is occurring.

The OPT(2:0) outputs are used during coprocessor transfers to indicate the type of transfer, or to provide other controls for the coprocessor. The interpretation of the OPT(2:0) signals depends on the implementation of the coprocessor, and may also depend on the R/*W signal.

Coprocessor Transfer Protocols

The protocols available for coprocessor transfers are based on the protocols for simple, pipelined, and burst-mode data accesses discussed in Section 5.2. The protocols for write accesses are used for transfers to the coprocessor, and the protocols for read accesses are used for transfers from the coprocessor.

The protocol for coprocessor transfers differs in several respects from the protocol for external data accesses:

1) The *CDA signal consistently replaces the *DRDY for transfers to the coprocessor. An active level on *CDA, for transfers to the coprocessor, has an effect that is equivalent to the effect of an active level on *DRDY for normal store-operations. Note that *DRDY is still used for transfers from the coprocessor.

2) The Address Bus does not contain an address during a coprocessor transfer, but may contain data in the case of a transfer to the coprocessor. However, for transfers from the coprocessor, the Address Bus is still sequenced as described in Section 5.2, and the sequencing is determined by the same controls—except that *CDA replaces *DRDY for transfers to the coprocessor. The contents of the Address Bus are determined by the coprocessor load instruction, as for other load instructions.

3) For any coprocessor transfer, an active level on *DERR causes a Coprocessor Exception trap, rather than a Data Access Exception trap.

4) For burst-mode coprocessor transfers, the interpretation of sequential addressing is undefined. For this reason, burst-mode transfers are normally restricted to 32 bits of information for every transfer, regardless of whether the transfer is to or from the coprocessor. Note, however, that the incrementing address sequence is still present in the definition of a burst-mode coprocessor transfer, and may be useful in some cases.

Sequencing of *CDA

The coprocessor de-asserts *CDA whenever it cannot accept a transfer from the Am29000. An inactive level on *CDA prevents the Am29000 from transferring operands or operation codes to the coprocessor when these transfers might interfere with coprocessor operation.

Normally, the coprocessor de-asserts *CDA when it begins an operation. *CDA remains inactive until the coprocessor has completed the operation and can accept further transfers from the processor. For some operations, a result may have to be read before the coprocessor can assert *CDA.

Independent of the presence of the coprocessor, a pull–down resistor in the range of 33K to 68 K ohms on *CDA is necessary for ADAPT29K compatibility.

The coprocessor can acknowledge a transfer by asserting *CDA. However, it is generally more efficient for the coprocessor to hold *CDA active as long as it can accept transfers. In the latter case, multiple data transfers can occur at a high rate, without involving long logic delays. *CDA is related to the operation of the coprocessor in this case, rather than to the transfer of data.

Exception Reporting

The coprocessor reports exceptions by the activation of *DERR during any coprocessor transfer. This causes a Coprocessor Exception trap to occur. However, if the DREQT(1:0) signals have the value 11 for a transfer from the coprocessor, exception reporting should be suppressed, and *DERR should not be asserted. Note, however, that the Am29000 does not enforce the suppression of exception reporting.

CHAPTER 7

PROGRAMMING

This chapter discusses programming topics as they relate to the Am29000. It focuses on the use of processor resources that were more formally described in Chapter 3. 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.

This chapter is organized into four sections. The first section describes the run-time storage organization recommended for the Am29000 and the use of the local registers to improve the performance of procedure calls. The two subsequent sections discuss applications and systems programming for the processor. The final section discusses certain features of the Am29000 pipeline that are exposed to—and must be properly handled by—software which executes on the processor.

7.1 RUN-TIME STORAGE ORGANIZATION AND CALLING CONVENTION

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 Am29000 is designed to minimize the overhead of calling a procedure, and allows efficient passing of parameters to a procedure and returning of results from a procedure. This section describes the Am29000 run-time storage organization and procedure-calling conventions.

7.1.1 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 7-1 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 7-1, 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

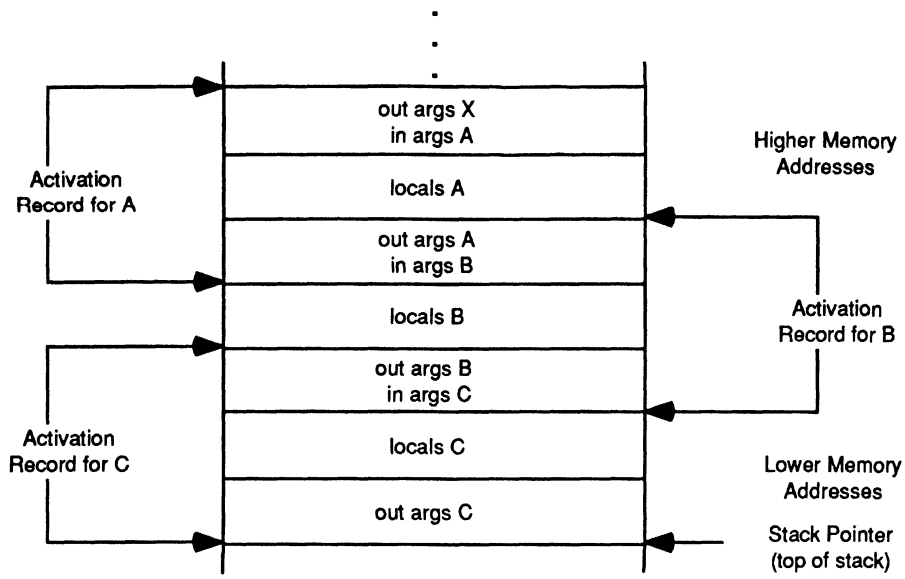


Figure 7-1. Run-time Stack Example

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.

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 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 may 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.

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.

The Am29000 run-time storage actually is 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 de-references). Both the Register Stack and the Memory Stack are stored in the external data memory. However, a portion of the Register Stack is normally also kept in the Am29000 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.

The Register Stack

The Register Stack contains activation records for active procedures (Figure 7-2). An activation record in the Register Stack stores the following information:

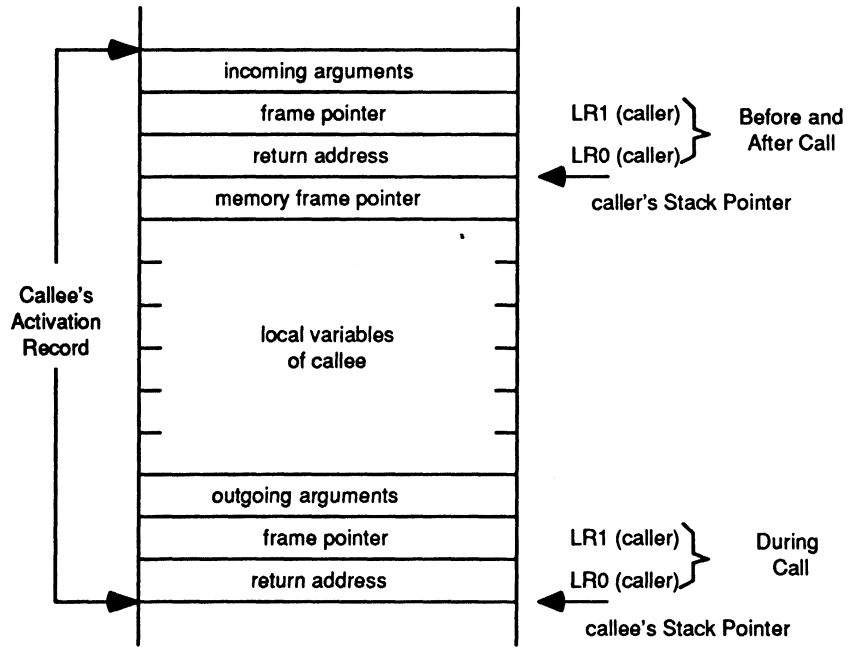


Figure 7-2. An Activation Record in the Register Stack

- 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 used when the called procedure calls another procedure.

Am29000 Local Registers as a Stack Cache

The Am29000 was designed for efficient implementation of the Register Stack. Specifically, the Am29000 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 7-3 shows the relationship 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 (*rfb, gr127*) pointer 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 (*rab, gr126*) pointer 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 7-4 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.

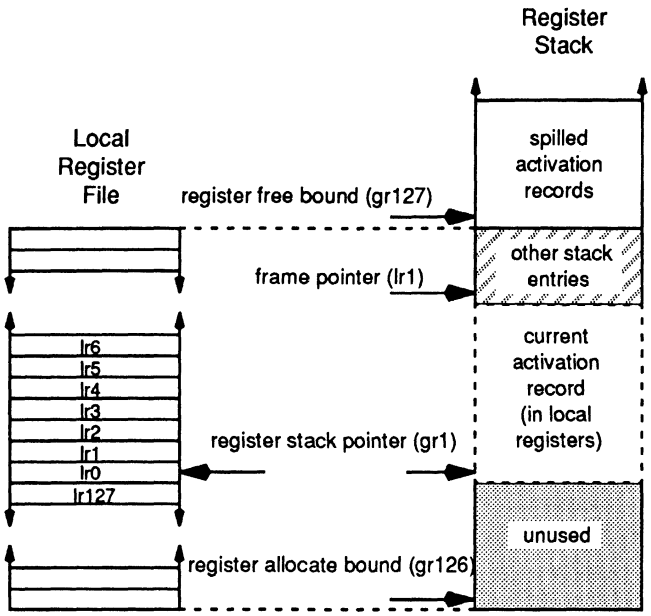


Figure 7-3. Relationship of Stack Cache and Register Stack

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 7-5 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 processor performs 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.

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.

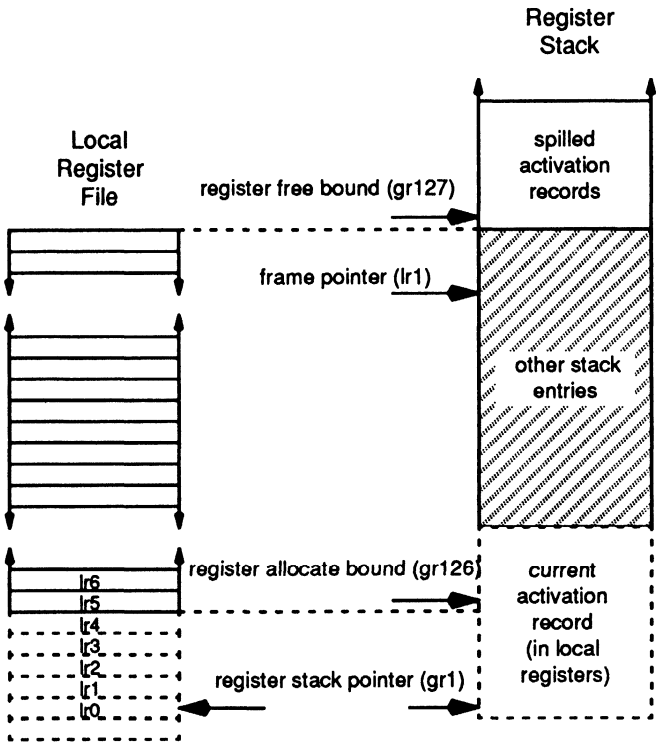


Figure 7-4. Stack Overflow

- 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.

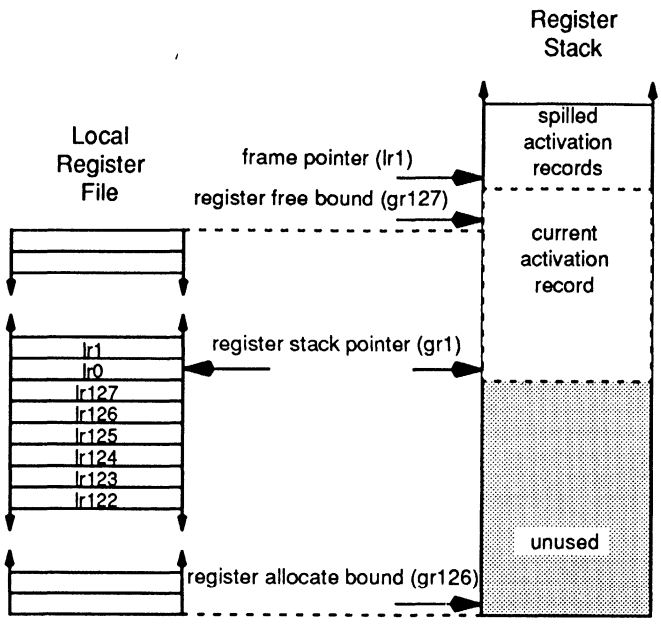


Figure 7-5. Stack Underflow

7.1.2 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 to the outgoing arguments area in 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.

- Store a return address and branch to the procedure.

- 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 Am29000 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.

Argument Passing

The linkage convention allows up to 16 word-length 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 must 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 is 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.

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 7-6). The value of *rsize* must be an even number given by the following formula:

rsize ≥ (size of local variable area) + (size of outgoing arguments area) + 2

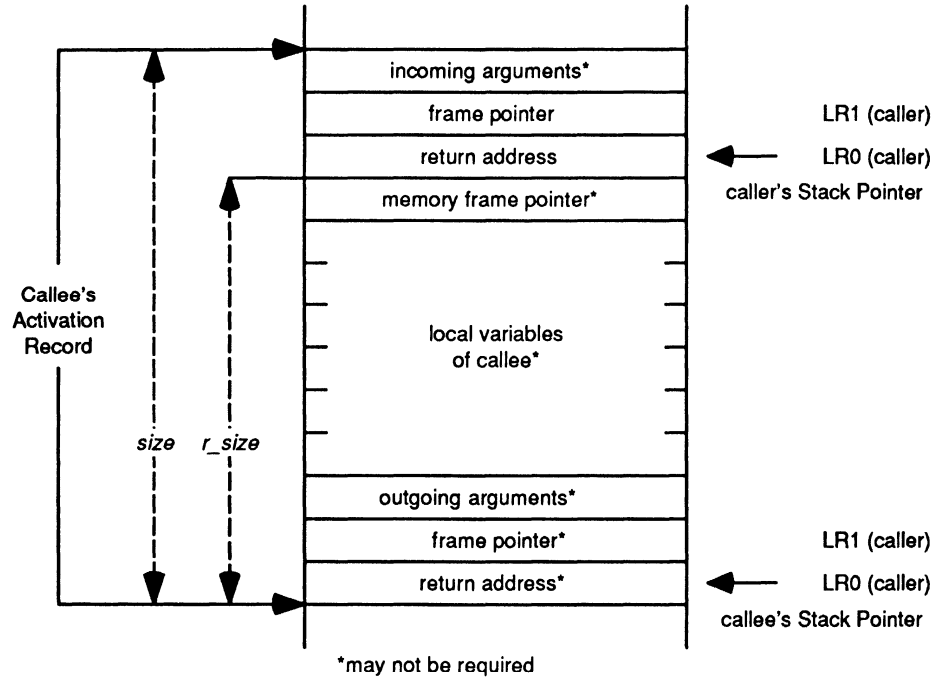


Figure 7-6. Definition of *size* and *rsize* Values

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).

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 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
```

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: lower bound – rsp
        sub     rab, rfb, tav                   ; adjust rfb pointer
        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
        storem  0, 0, lr0, rab                  ; spill
        sll     rfb, rab, 0                     ; adjust rfb pointer
        jmpi    tpc                             ; return to trapping procedure
        sll     rab, rsp, 0                     ; adjust rab
```

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.

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
```

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
        const   tav, 0x80<<2            ; 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           ; push up 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
        sll     tav, rfb, 0             ; save old rfb
        sll     rfb, fp, 0              ; push up the free bound
        jmpi    tpc                     ; return to trapping procedure
        loadm   0, 0, gr0, tav          ; fill
```

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 must allocate a frame on the Register Stack.

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 Am29000 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 may 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*). 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:

```
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
```

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 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 Am29000 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.

Floating-Point Registers

If Am29027 floating-point registers are kept in the Register-Stack Frame, they are placed at the bottom of the Register Stack frame, just before the memory frame pointer (Figure 7-7). The callee must save any of Am29027 Float Register 3 through Float Register 7 that it may alter, and must restore the saved registers upon return. The contents of Float Register 0 through Float Register 2 are allowed to be modified by the callee, and are not saved. Only those floating-point registers actually modified need be saved; floating-point registers which are not saved need not have space allocated for them.

Transparent Procedures

A transparent procedure is one that requires very little overhead for managing run-time storage. Transparent procedures are used in the Am29000 run-time organization primarily to implement compiler-specific support functions, such as integer multiply.

A transparent routine does not allocate any activation-record frames, and preserves all registers except for the *tav* and *tpc* registers (Global Register 121 and Global Register 122). 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.

7.1.3 REGISTER USAGE CONVENTION

The Am29000 run-time organization standardizes the uses of the local and global registers. This section summarizes register use and the nomenclature for register values:

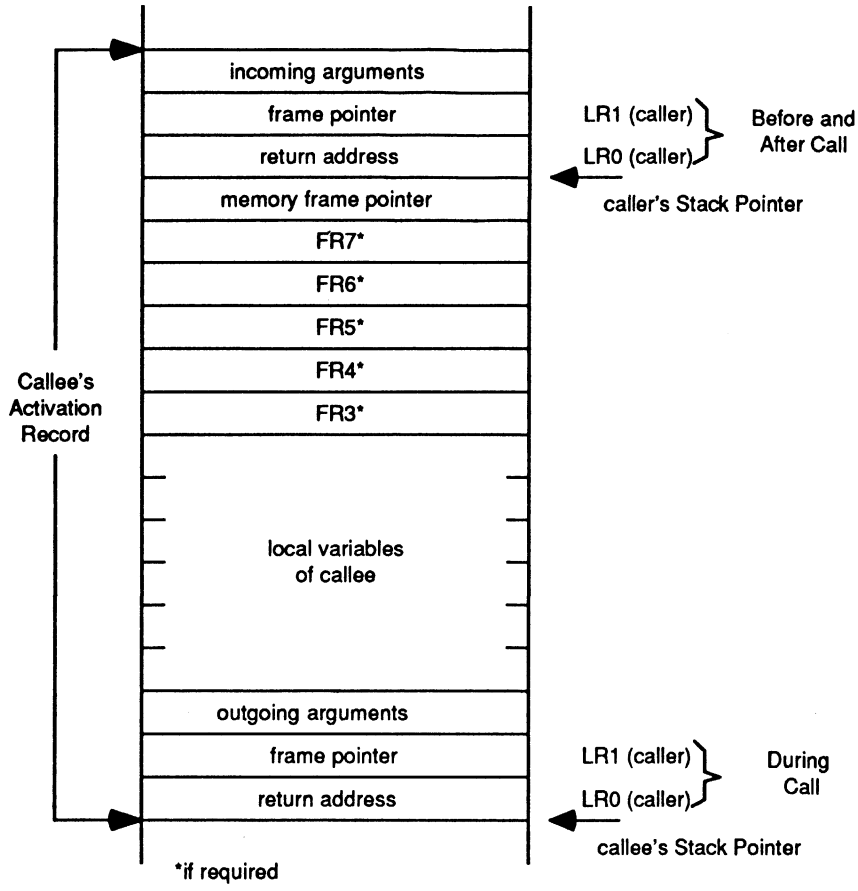


Figure 7-7. Am29027 Registers Saved in the Register Stack

- 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*).
- 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.

7.1.4 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, (128-2)*4              ; maximum register allocation
        sub     gr1, gr1, tav               ; allocate register frame
        asgeu   V_SPILL, gr1, rab
        const   tav,(128-2)*4+(2*4)         ; incoming arguments and overhead
        add     lr1,gr1,tav                 ; create frame pointer
        add     lr125,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, lr125,0                ; de-allocate memory frame
        const   tav,(128-2)*4               ; maximum allocation size
        add     gr1,gr1,tav                 ; de-allocate register frame
        const   gr96,1                      ; return value
        jmpi    lr0                         ; return to caller
        asleu   V_FILL, lr1,rfb             ; ensure caller's registers in frame
```

7.1.5 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 7-8 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 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.

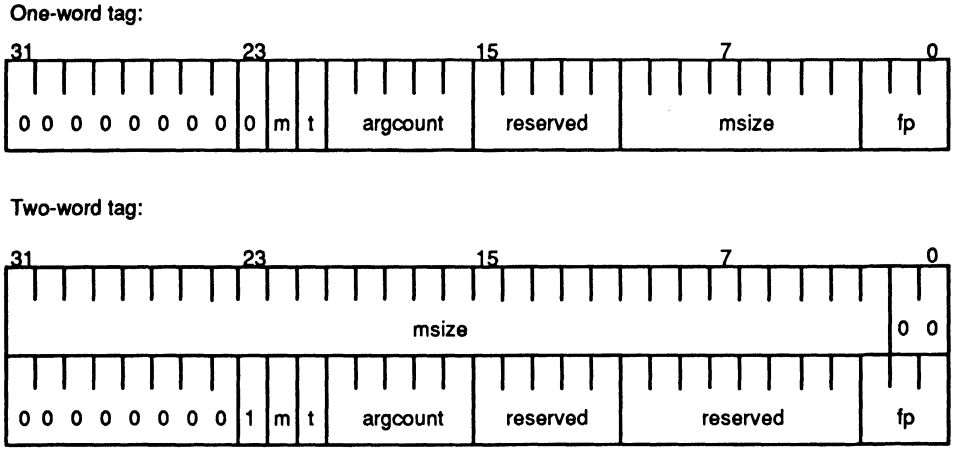


Figure 7-8. Trace-Back Tags

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	fp regs	number of floating-point registers saved for caller

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.

7.2 APPLICATIONS-PROGRAMMING CONSIDERATIONS

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

7.2.1 ADDRESSING GENERAL-PURPOSE REGISTERS INDIRECTLY

Registers in the processor usually are addressed directly by fields within instructions. However, indirect addressing of registers may be required in some situations, such as when a program pointer is known to point to a variable that is resident in the register file.

Three special registers—Indirect Pointers A, B, and C—are provided so that separate indirect register numbers can be set for each of the source and destination operands within an instruction.

Indirect Pointer C corresponds to the destination register RC, Indirect Pointer A corresponds to the RA operand register, and Indirect Pointer B corresponds to the RB operand register.

A given indirect pointer (the value in the corresponding register) is used to address the register file whenever Global Register 0 is specified as a source or destination register. For example, a value of 0 in the RA field of an instruction causes the content of the Indirect Pointer A Register to be used to access the RA operand.

The indirect pointers can be set by the Move To Special Register and Floating-Point instructions, and by the instructions EMULATE, MULTIPLY, MULTM, DIVIDE, MULTIPLU, MULTMU, DIVIDU, and Set Indirect Pointers (SETIP). The Move To Special Register instructions set the indirect pointers individually as special-purpose registers. Of the remaining instructions, all but the EMULATE instruction set all three indirect pointers simultaneously, deriving the values that are written into the pointers from the instruction fields RC, RA, and RB. The EMULATE instruction sets all three indirect pointers, but only the Indirect Pointer A and Indirect Pointer B registers are written with meaningful values.

When an indirect pointer is set by a Move To Special Register, bits 9-2 of the source operand are copied to corresponding bits in the indirect pointer. This allows the addressing of general-purpose registers, via the indirect pointers, to be consistent with the addressing of words in external memories and devices.

When the indirect pointers are set from instruction fields, the resulting values reflect the Stack-Pointer addition that is performed on local registers. In addition, register bank-protection checking is performed on the values that are loaded. A Protection Violation trap occurs if the values represent registers that cannot be accessed. The indirect pointers may thus be used to access exactly those operands that would be accessed by the instruction fields setting the indirect pointers. Consequently, a routine that emulates an instruction operation can access, with no overhead, the source and destination registers for the instruction being emulated. No copying of arguments and results needs to be done.

When using indirect register addressing, at least one cycle of delay must separate any instruction that sets an indirect pointer and any instruction which de-references that pointer. This restriction is the result of processor pipelining (see Section 7.4.3).

7.2.2 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.

7.2.3 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:

```
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.

7.2.4 MULTI-PRECISION INTEGER ADDITION AND SUBTRACTION

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.

7.2.5 INTEGER MULTIPLICATION

The processor performs integer multiplication by a series of multiply step instructions, rather than by a single instruction. 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 variables, 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. 64-bit 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 to be performed by doing some quick checks on 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 the 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 tav, 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 the indirect pointers:
            mtsrim  FC, 8                   ; useful only 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         ; perform initial multiply step

            .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 calling routine,
            extract gr0, Temp0, Temp1       ; extracting the result in the delay-slot of the jump
```

```
do16:
        const   Temp2, 0x7fff          ; less than 16 bits?
        cpleu   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 the 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 lower 16 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 calling routine
        extract gr0, Temp0, Temp1      ; extracting the result in the delay slot of the jump

do32:
        mulu    Temp0, Temp1, 0        ; perform the initial step

        .rep    31                     ; expand out 31 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                 ; copying the result to the result return register in the
                                       delay slot
```

7.2.6 INTEGER DIVISION

The processor performs integer division by a series of divide step instructions, rather than by a single instruction. 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 tav, 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
```

7.2.7 TRAPPING ARITHMETIC INSTRUCTIONS

The processor does not incorporate hardware to directly support floating-point operations, nor does it directly support full multiply and divide operations. However, instructions to perform these operations are included in the instruction set. These instructions are included in the anticipation of future processor implementations that may include hardware to perform these operations.

In applications programs that must be fully object-code compatible with future processor versions—while taking advantage of the performance of future versions—the defined instructions should be used to perform floating-point, multiplication, and division operations.

In the Am29000, the Floating-Point, MULTIPLY, MULTM, DIVIDE, MULTIPLU, MULTMU, and DIVIDU instructions simply 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. The trap handler can directly emulate the execution of the instruction, or can perform the instruction using an external coprocessor.

Note that interfacing to an external arithmetic coprocessor via the trapping arithmetic instructions simplifies the definition of the coprocessor as a system option. If the coprocessor is present, the trap handler uses the coprocessor to perform the arithmetic operations. If the coprocessor is not present, the trap handler emulates the operation by software.

7.2.8 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.

7.2.9 GENERATING LARGE CONSTANTS

Eight-bit constants are directly available to most instructions. Larger constants must be generated explicitly by instructions and placed into registers before they can be used as operands. The processor has three instructions for the generation of large data constants: Constant (CONST); Constant, High (CONSTH); and Constant, Negative (CONSTN).

The CONST instruction sets the least-significant 16 bits of a register with a field in the instruction; the most-significant 16 bits are zero-bits. This instruction allows a 32-bit positive constant to be generated with one instruction, when the constant lies in the range of 0 to 65535.

Any 32-bit constant may be generated with a combination of the CONST and CONSTH instructions. The CONSTH instruction sets the most-significant 16 bits of a register with a field in the instruction; the least-significant bits are set to the value of the corresponding bits in a source operand-register. Thus, to create a 32-bit constant in a register, the CONST instruction sets the least-significant 16 bits, and the CONSTH instruction sets the most-significant 16 bits.

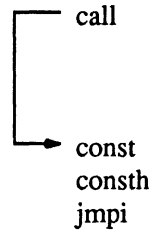
The CONSTN instruction sets the least-significant 16 bits of a register with a field in the instruction; the most-significant 16 bits are one-bits. This instruction allows a 32-bit, negative constant to be generated with one instruction, when the constant lies in the range of -65536 to -1.

7.2.10 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 re-targeted 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.

7.2.11 NO-OPS

When a NO-OP is required for proper operation (e.g., as described in Section 7.4.3), 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.

7.2.12 CHARACTER-STRING OPERATIONS

The need to perform operations on character strings arises frequently in many systems. The processor provides operations for manipulating character data, but these are frequently inefficient for dealing with character strings, since the processor is optimized for 32-bit data quantities.

It is much more efficient, in general, to perform character-string operations by operating on units of four bytes each. These four-byte units are more suited to the processor's data-flow organization. However, there are several things to be considered when dealing with four-byte units, as outlined in this section.

Alignment of Bytes Within Words

Character strings normally are not aligned with respect to 32-bit words. Thus, when word operations are used to perform character-string operations, alignment of the character strings must be taken into account.

For example, consider a character string aligned on the third byte of a word that is moved to a destination string aligned on the first byte of a word. If the movement is performed word-at-a-time, rather than byte-at-a-time, the move must involve shift and merge operations, since words in the destination character-string are split across word boundaries in the source character string.

The processor's Funnel Shifter can be used to perform the alignment operations required when character operations are performed in four-byte units. Though the Funnel Shifter supports general bit-aligned shift and merge operations, it easily is adapted to byte-aligned operations.

For byte-aligned shift and merge operations, it is only necessary to insure that the two most-significant bits of the Funnel Shift Count (FC) field of the ALU Status Register point to a byte within a word, and that the three least-significant bits of the FC field are 000.

Detection of Characters Within Words

Most character-string operations require the detection of a particular character within the string. For example, the end of a character string is identified by a special character in some character-string representations. In addition, character strings often are searched for a specific pattern. During such searches, the most-frequently executed operation is the search within the character string for the first character of the pattern.

The processor provides a Compare Bytes (CPBYTE) instruction, which directly supports the search for a character within a word. This instruction can provide a factor-of-four performance increase in character-search operations, since it allows a character string to be searched in four-byte units.

During the search, the words containing the character string are compared, a word at a time, to a search key. The search key has the character of interest in every byte position. The CPBYTE instruction then gives a result of TRUE if any character within the character-string word matches a byte in the search key.

7.2.13 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 normally much 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 large 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.

7.3 SYSTEMS-PROGRAMMING CONSIDERATIONS

This section discusses topics of general concern in the implementation of control programs and operating systems.

7.3.1 SYSTEM PROTECTION

The Am29000 provides protection of several different system resources. In general, this protection is based on the value of the Supervisor Mode (SM) bit in the Current Processor Status Register.

Memory Protection

Memory and input/output access protection is provided by the Memory Management Unit. Each Translation Look-Aside Buffer entry in the MMU contains protection bits which determine whether or not a given routine can access 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 User mode.

A Data TLB Protection Violation or Instruction TLB Protection Violation trap occurs if a data or instruction access, respectively, is attempted, but is not allowed because of the value of the protection bits.

Register Protection

General-purpose registers 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, protected from corruption by User-mode programs. Additionally, it allows processor registers to be partitioned among multiple tasks.

If a User-mode program attempts to access a protected general-purpose register, a Protection Violation trap occurs. Supervisor-mode programs may access any general-purpose register, regardless of protection.

The special-purpose registers 0 to 127 and all Translation Look-Aside Buffer registers are protected from User-mode access. Any attempted access of these registers by a User-mode program causes a Protection Violation trap. The special-purpose registers 160 to 255 (though not implemented) are protected from any access. Any attempted access of special-purpose registers 160 to 255, even in the Supervisor mode, causes a Protection Violation trap.

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/*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 *DERR input.

7.3.2 INTERRUPTS AND TRAPS

The Am29000 automatically saves only the Current Processor Status Register in the Old Processor Status Register when an interrupt or trap is taken. The processor does not automatically save any other state when an interrupt or trap is taken, but rather freezes the contents of the following registers:

1) Program Counters 0, 1, and 2.

2) Channel Address, Channel Data, and Channel Control.

3) ALU Status.

When these registers are frozen, they are allowed to be updated only by Move To Special Register instructions. The frozen condition is controlled directly by the Freeze (FZ) bit in the Current Processor Status Register.

Since the Channel Address, Channel Data, and Channel Control registers are frozen when an interrupt or trap is taken, the interrupt handler may perform any single-word loads and stores without interfering with the restart state of a channel operation in the interrupted routine. However, load-multiple and store-multiple operations have unpredictable results if performed while the FZ bit is 1, since these operations are sequenced by the Channel Control Registers.

Vector Area

As discussed in Section 3.5.4, interrupts and traps are dispatched through a 256-entry Vector Area, which directs the processor to a routine to handle a given interrupt or trap. Only 64 entries of this area are required for basic processor operation (or 22, if instruction emulation is not used).

The required number of Vector Area entries is system-dependent, as determined by the vector numbers that are specified in the assert and EMULATE instructions. The number of entries can be restricted to reduce the memory requirements for the Vector Area, which is especially important when the Vector Area is organized as a sequence of 64-instruction blocks. However, there is nothing to prevent an instruction from specifying a vector number in the range 64 to 255. For this reason, it may not be possible to reduce the size of the Vector Area, since erroneous instruction vector numbers might cause unpredictable results.

The Vector Area may be relocated by the Vector Area Base Address Register, and there may be multiple Vector Areas in the system, with the Vector Area Base Address Register pointing to the one that is currently active.

Interrupt Handling

For temporary program interruptions, such as for Translation Look-Aside Buffer reload, the basic processor interrupt mechanism is sufficient to eliminate the need for the interrupt or trap handler to save any state for the interrupted routine. This state may be left in the appropriate registers while the handler executes. An interrupt return returns immediately to the interrupted program.

Besides the direct performance advantage that results from not saving state for temporary program interruptions, there is an additional advantage provided by the processor. When the state of the interrupted routine remains in the appropriate registers, the processor can detect that the Program Counter 0 and Program Counter 1 registers contain sequential addresses. Instead of performing two non-sequential instruction fetches for the interrupt return in this case, the processor initiates only a single non-sequential fetch (the second fetch is performed as a sequential fetch). This reduces the overhead of the interrupt return for these routines.

Note that when the state of an interrupted program remains in the processor, the processor cannot be enabled to take any further interrupts until an interrupt return is executed. Therefore, this capability should be restricted to time-critical routines, where the execution time of the routine does not interfere with interrupt-latency considerations. (Note that the Interrupt Pending bit of the Current Processor Status Register may be used to detect the presence of external interrupts while these interrupts are disabled).

To support dynamically nested interrupts and traps, the interrupt or trap handler must save state as necessary for the application, using an appropriate data structure (such as an interrupt stack or program status area). Once the state has been saved (or, alternately, while it is being saved), the handler can load the state for a new program to be executed. An interrupt return then initiates the execution of the new program.

Interrupt Return

An interrupt return resumes the execution of a program whose processor state is contained in the following registers:

1) Old Processor Status.

2) Program Counters 0 and 1.

3) Channel Address, Channel Data, and Channel Control.

This state is most likely different from the state of the program executing the interrupt return. These registers must be set appropriately before an interrupt return is executed.

Note that the instruction sequence that sets these registers must have a Current Processor Status that is equivalent to that of an interrupt or trap handler; the FZ bit must be 1, and interrupts and traps must be disabled.

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. A Supervisor-mode assert instruction 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 *INTR1 interrupt may be simulated by an assert instruction that specifies a vector number of 17, without the activation of the *INTR1 signal.

7.3.3 FAST CONTEXT SWITCHING

The Am29000 allows general-purpose registers to be partitioned among multiple tasks, so that context switching can be very fast. However, in this configuration, fewer registers are available to each task, and the local registers cannot be used as a stack cache (see Section 7.1). Thus, task-switch time is minimized at the possible expense of an increase in procedure-call and procedure-execution time. Even so, this trade-off may be appropriate in some real-time applications.

The 128 local registers may be partitioned into eight banks of 16 registers each, with a each bank of registers allocated to one of eight tasks resident in the processor. Partitioning can be made

transparent to resident tasks (except that each task has a small number of registers), because the Stack Pointer can be set so that the first register in each bank is addressed as Local Register 0. Even when two or more banks of local registers are combined into larger banks, each of the larger banks can still start with Local Register 0. Registers within a given bank can be protected from access by other tasks by the Register Bank Protect Register.

Since the Stack Pointer does not affect the addressing of global registers, global registers cannot be partitioned among multiple tasks. A task cannot be made to reference the proper global registers unless the registers allocated to a given task are known before execution; this restriction is too severe in most cases. Because of this, the global registers should be restricted for use by the operating system, and protected from access by resident tasks. Given this restriction, it is best to use the global registers to contain the processor state of resident tasks.

The processor state that normally must be saved on a task switch consists of the contents of the following registers:

1) Old Processor Status.

2) Channel Address.

3) Channel Data.

4) Channel Control.

5) Program Counter 0.

6) Program Counter 1.

7) Q.

8) ALU Status.

Thus, the processor state can be saved in eight general-purpose registers, and the state for eight resident tasks can be saved in the 64 global registers. This state is protected from access (by resident tasks) by the Register Bank Protect Register.

In summary, the general-purpose registers can accommodate as many as eight tasks resident in the processor simultaneously. In this configuration, the global registers contain the processor state for the tasks, and the local registers contain the program state (e.g., variables) for the tasks. Each task has 16 general-purpose registers, numbered 0 to 15, which are used in the normal way.

The above configuration allows a complete context switch to be performed by the adjustment of the Stack Pointer and the movement of processor state to and from general-purpose registers. These operations can be completed within 17 processor cycles.

7.3.4 MEMORY MANAGEMENT

This section discusses various issues involved in memory management as they relate to an operating system. The focus is on virtual-addressing issues.

Virtual Page Size

The MMU Configuration Register determines the size of a virtual page mapped by the Memory Management Unit. The choices for page size are 1, 2, 4, and 8 Kbytes. 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 8 Kbytes 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).

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, a TLB 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 then 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/data memory.

Monitoring Critical Areas of Memory

In certain fault-tolerant systems, it is necessary to detect changes to critical areas of memory, so that these changes may be reflected immediately on a non-volatile storage device. To monitor critical memory areas, the TLB protection bits can be set so that any change to the area causes a Data TLB Protection Violation trap. This trap signals that the area is being modified.

In this use of the protection bits, the trap handler does not set the bits to allow the access. Rather, the trap handler must emulate the access, using the Channel Address, Channel Data, and Channel Control registers. The Contents Valid (CV) bit of the Channel Control Register is reset before the trapping routine is restarted, so that the trap does not re-occur.

TLB Miss Handling

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 7.3.5).

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 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).

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 its TLB contents. When the task 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 perform all TLB reloads individually.

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 changed often so that it affects more than one process, warm start may not be advantageous.

Minimum Number of Resident Pages

In any processor that supports demand-paging, there is 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 Am29000, two pages are required for a process to make progress through the system. The reason for this requirement is that the Am29000, on interrupt return, restarts an interrupted Load Multiple or Store Multiple only after fetching two instructions (see Section 3.5.5). 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.

Branch Target Cache Considerations

The Branch Target Cache is accessed with virtual as well as physical addresses, depending on whether address translation is enabled for instruction accesses. Because of this, the Branch Target 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 Branch Target Cache tags, so the tags do not necessarily perform valid comparisons. Also, the Branch Target Cache does not differentiate between virtual and physical addresses, so that it may perform an invalid comparison after address translation for instructions is enabled or disabled.

If a TLB miss occurs during the address translation for a branch target instruction, the processor considers the contents of the Branch Target 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 Branch Target Cache still may perform an invalid comparison.

To avoid the above problem, the contents of the Branch Target 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 Branch Target Cache to become invalid (after the next successful branch). 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 Branch Target 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 it changes address translation. At the point of interrupt return, the contents of the Branch Target Cache are most likely not of much use anyway.

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

The Branch Target Cache distinguishes between virtual and physical addresses, between the instruction/data memory and instruction read-only memory (ROM) address spaces, and between User-mode and Supervisor-mode addresses. Thus, the Branch Target Cache does not have to be invalidated on transitions between these address spaces. This improves the performance of applications that make heavy use of ROM-based and/or operating-system routines in either a virtual or physical address space.

7.3.5 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 is not detectable to the program causing the trap.

Additionally, in the Am29000, 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 Am29000 overlaps 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 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. 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.

Please note that the exception handler for the Data Access Exception must clear the Transaction Faulted (TF) bit in the Channel Control Register. Failure to clear the TF bit will result in the Am29000 taking the trap again once the exception handler returns, causing an infinite sequence 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 its 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. Interrupted load-multiple and store-multiple operations are restarted as if they had faulted. In this case, the operation resumes from the point of interruption, not the beginning of the sequence.

7.3.6 MULTI-PROCESSING

The Am29000 provides 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 atomically 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 *LOCK output during the external access. Since the processor cannot control the behavior of external devices and memories directly, system hardware must support locking, if required.

Note that the protocol for the locking and unlocking of 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 by the locking process 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. This bit activates the *LOCK pin, and prevents the processor from relinquishing the channel to another channel 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 channel 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 also must be disabled for true exclusion, this may have a negative impact on system performance if used improperly.

7.3.7 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. These registers implement timing functions independent of program execution.

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. The reloading of 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 described above. 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 also is set. The OV bit indicates that a Timer interrupt may have occurred before a previous interrupt was serviced.

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 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 is normally 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.

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.

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 separately specified.

The Timer Facility can be used to generate time intervals for collecting virtual page usage information (see Section 7.3.4). 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.

7.3.8 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 always is copied into the TP bit when an instruction enters the write-back stage. 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 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—that is 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.

7.4 PIPELINE FEATURES EXPOSED TO SOFTWARE

In certain cases, the Am29000 pipeline is exposed during instruction execution, in that the execution of certain instructions are dependent on the execution of previous instructions. This section discusses the cases where the pipeline is exposed to software, and the resulting effect on instruction execution.

7.4.1 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 immediately following instruction is called the delay instruction.

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 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 preempts 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)
        sub  r  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.

7.4.2 OVERLAPPED LOADS AND STORES

The processor allows an external access to be overlapped with instruction execution. This means that, while the access is being performed, the processor continues to execute instructions, as long as the instructions and data required for execution are available.

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 JMPF is taken—and 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.

7.4.3 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 7.1.2). 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.

CHAPTER 8

INSTRUCTION SET

This chapter provides a specification of the Am29000 instruction set. Sections 8.1 through 8.3 describe the terminology used, the setting of the ALU Status Register by instructions, and the instruction formats. Section 8.4 describes each instruction in detail; instructions are presented alphabetically by assembler mnemonic. Finally, Section 8.5 gives an index of instructions by operation code.

8.1 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.

8.1.1 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. This terminology also is used for coprocessor words, except that the address n either has no pre-defined interpretation or is a data item transferred to the coprocessor.

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 8.3).
SPDEST	The special-purpose register that is the destination of an instruction.
SPECIAL	The content 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 8.3).
SRCA SRCB	The contents of general-purpose registers, used as instruction operands.
SRCA.BYTEn SRCB.BYTEn	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.

8.1.2 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.

8.1.3 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 8.3).

8.1.4 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:

ce	Determines the Coprocessor Enable (CE) bit of a load or store instruction.
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.

8.2 ARITHMETIC/LOGIC STATUS RESULTS OF INSTRUCTIONS

8.2.1 ARITHMETIC/LOGIC STATUS BITS

The arithmetic/logic status bits of the ALU Status Register are:

V	Overflow
N	Negative
Z	Zero
C	Carry

The C bit is used in extended arithmetic operations (i.e., on operands greater than 32 bits in length), and the N bit is used in divide step operations. Other than these uses, the status bits are not involved in instruction operations. In particular, they are not used to determine the outcome of conditional jump instructions: Boolean values in registers are used instead for this purpose. The status bits are primarily informational.

Except for instructions that explicitly modify the ALU Status Register, the status bits are modified only by the execution of instructions in the Arithmetic and Logical classes. The Arithmetic and Logical instructions affect the status bits differently. The following two sections describe the setting of the status bits by Arithmetic and Logical instructions.

When the Freeze (FZ) bit of the Current Processor Status Register is 1, the ALU Status Register is not modified except by the Move To Special Register instruction.

8.2.2 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.

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.

8.2.3 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.

8.2.4 FLOATING-POINT STATUS

The floating-point instructions check for a number of exceptional conditions, and report these exceptions by setting bits of the Floating-Point Status Register (see Section 3.2.2). 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 Am29000 sets 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.

Since a floating-point exception may affect either a trap status bit, a sticky status bit, or both, the description of status results for floating-point instructions in this section indicates the exceptions that may be detected, rather than which status bits are set. The following terminology is used:

fpD Divide By Zero. The processor determines whether a divide operation has a zero divisor and a non-zero, finite dividend. If so, the DT and/or DS bits of the Floating-Point Status Register are set.

fpX Inexact Result. If the result of the associated floating-point operation is not equal to the infinitely-precise result, the XT and/or XS bits of the Floating-Point Status Register are set.

fpU Underflow. If the result of the associated floating-point operation is too small to be expressed in the destination format, the UT and/or US bits of the Floating-Point Status Register are set.

fpV Overflow. If the result of the associated floating-point operation is too large to be expressed in the destination format, the VT and/or VS bits of the Floating-Point Status Register are set.

fpR Reserved Operand. If one or more input operands to the associated floating-point operation is a reserved value, or if the result of this floating-point operation is a reserved value, the RT and/or RS bits of the Floating-Point Status Register are set.

fpN Invalid Operation. If the input operands to the associated floating-point operation produce an indeterminate result, the NT and/or NS bits of the Floating-Point Status Register are set.

8.3 INSTRUCTION FORMATS

All instructions for the Am29000 are 32 bits in length, and are divided into four fields, as shown in Figure 8-1. 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.

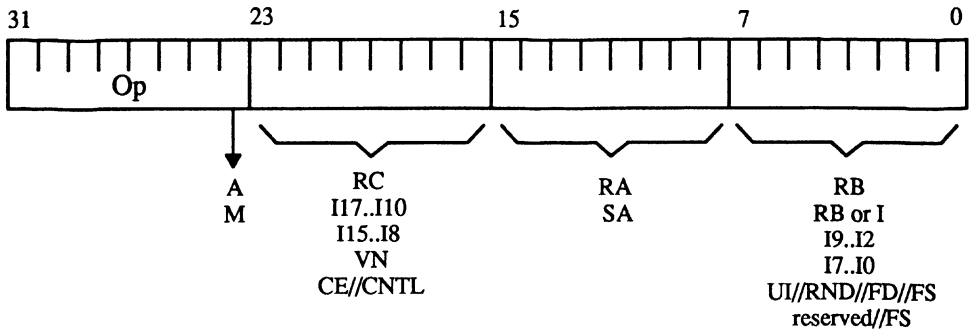


Figure 8-1. Instruction Format

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.

CE//CNTL — This field controls a load or store access, as described in Sections 3.4.4 and 6.1.2.

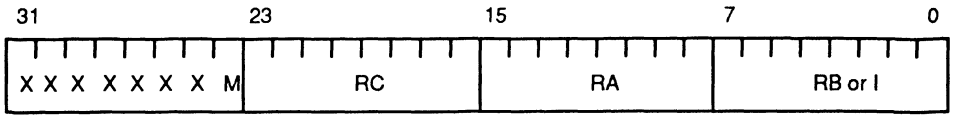
Bits 15-8

RA — The RA field contains a global or local register number.

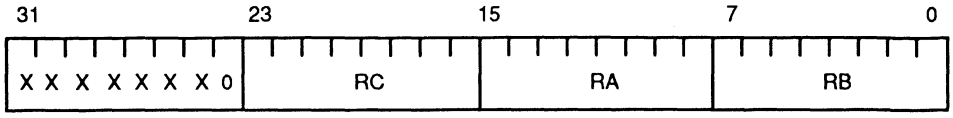
SA	The SA field contains a special-purpose register number.
Bits 7-0	
RB	The RB field contains a global or local register number.
RB or I	This field contains either a global or local register number, or an 8-bit instruction constant, depending on the value of the M bit of the operation code.
I9..I2	This 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 8-2.

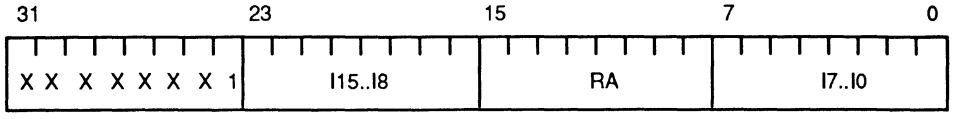
Three operands, with possible 8-bit constant:



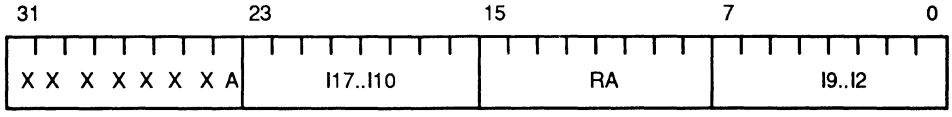
Three operands, without constant:



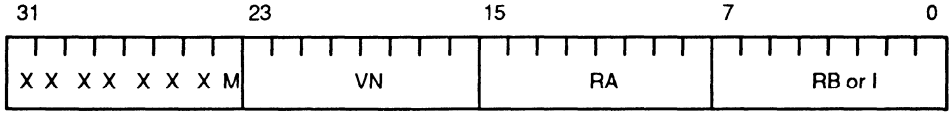
One register operand, with 16-bit constant:



Jumps and calls with 16-bit instruction address:



Two operands with trap vector number:



Loads and stores:

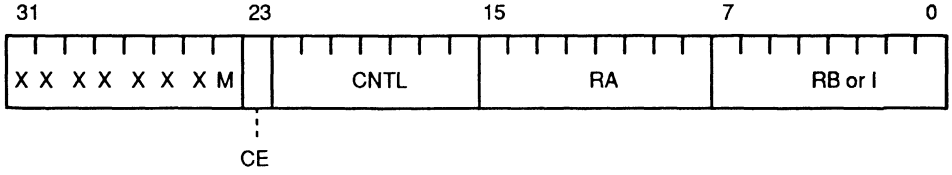


Figure 8-2. Frequently Occurring Instruction Field Uses

8.4 INSTRUCTION DESCRIPTION

This section describes each Am29000 instruction in detail. Figure 8-3 illustrates the layout of the information given for each description.

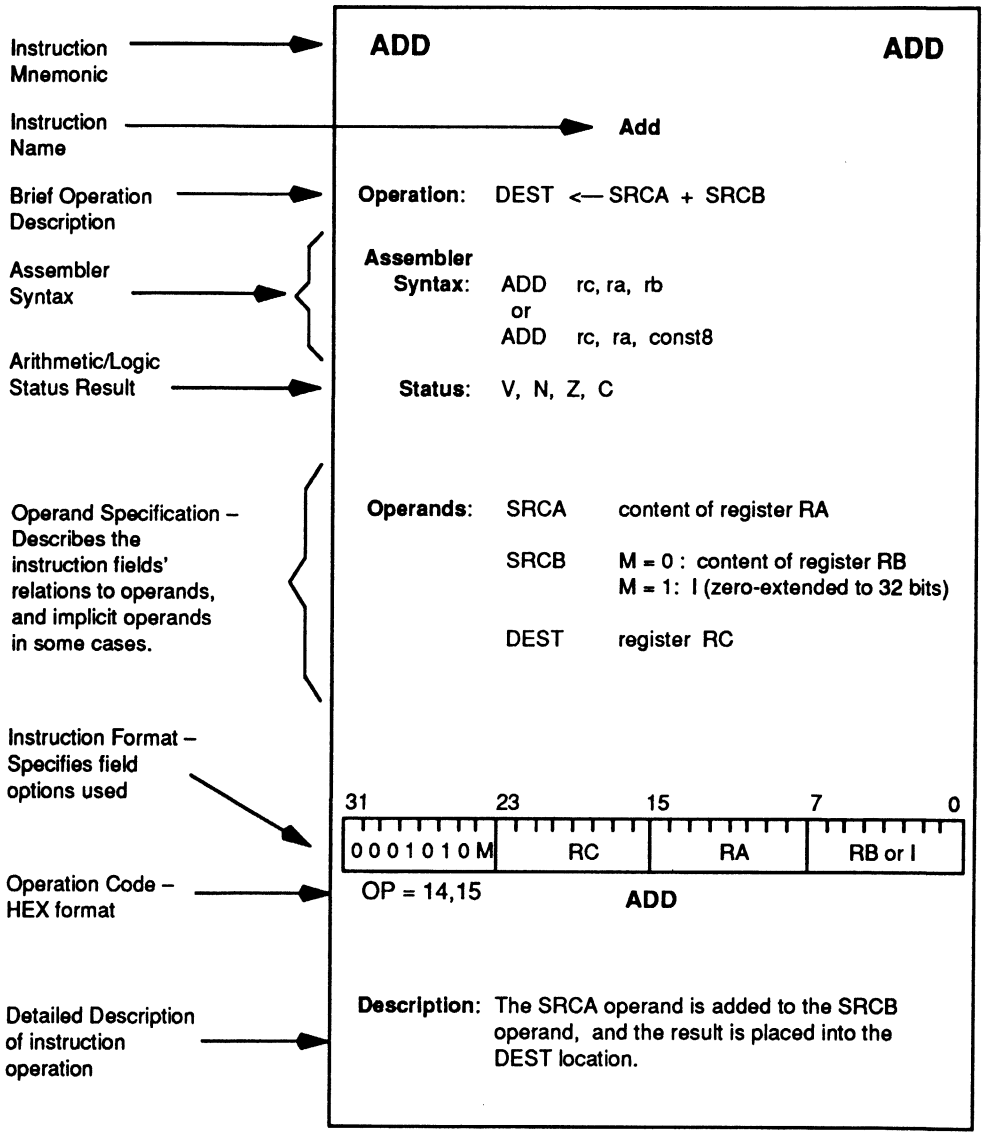


Figure 8-3. Instruction-Description Format

Add

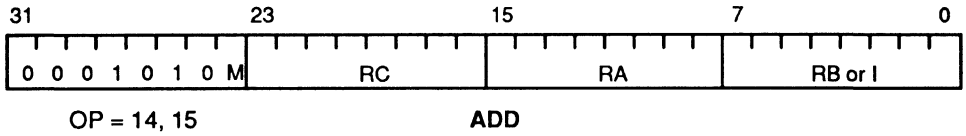
Operation: DEST <— SRCA + SRCB

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

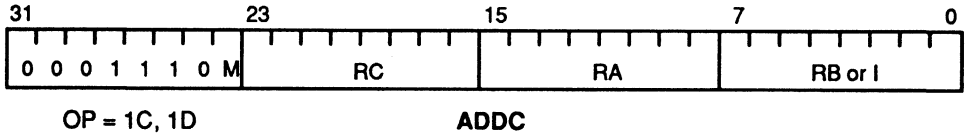
Operation: DEST <— SRCA + SRCB + C

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



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

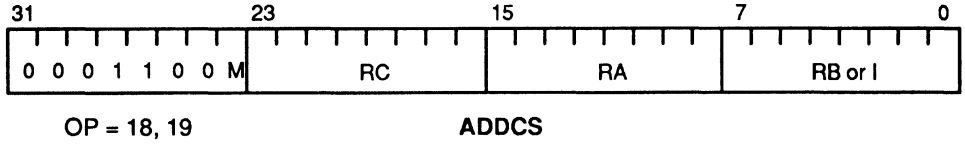
Operation: DEST <— SRCA + SRCB + C,
IF signed overflow THEN Trap (Out of Range)

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



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

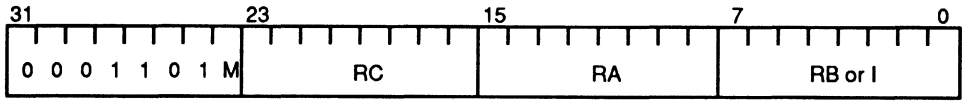
Operation: DEST <— SRCA + SRCB + C,
IF unsigned overflow THEN Trap (Out of Range)

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

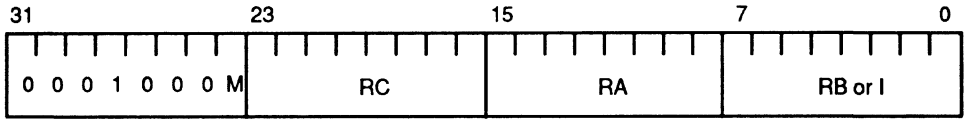
Operation: DEST <— SRCA + SRCB
IF signed overflow THEN Trap (Out of Range)

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

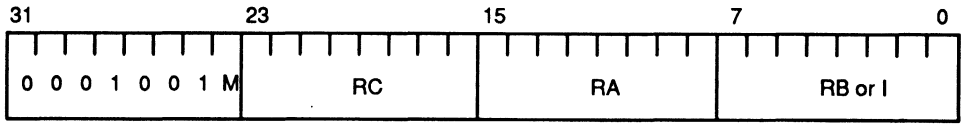
Operation: DEST <— SRCA + SRCB
IF unsigned overflow THEN Trap (Out of Range)

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

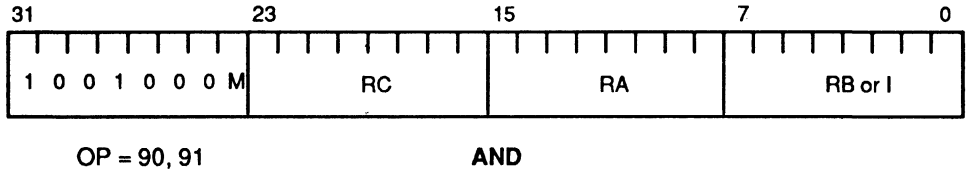
Operation: DEST <— SRCA & SRCB

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



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

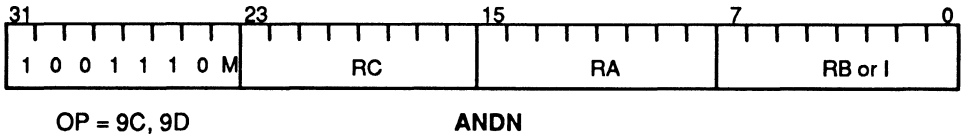
Operation: DEST <— SRCA & ~SRCB

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



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

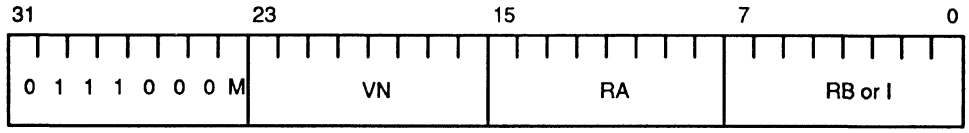
Operation: IF SRCA = SRCB THEN Continue
ELSE Trap (VN)

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

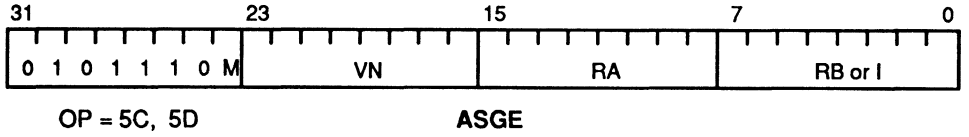
Operation: IF SRCA >= SRCB THEN Continue
ELSE Trap (VN)

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



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

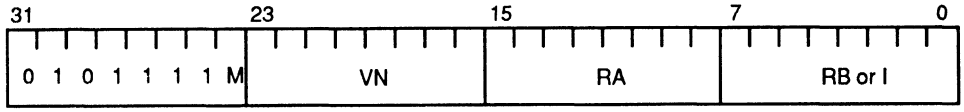
Operation: IF SRCA >= SRCB (unsigned) THEN Continue
ELSE Trap (VN)

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

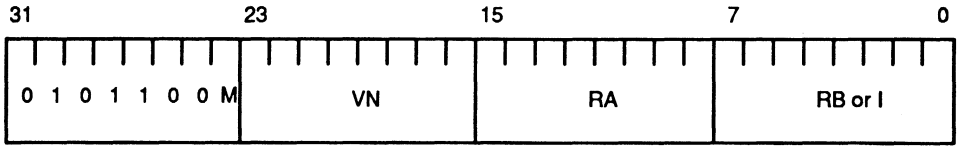
Operation: IF SRCA > SRCB THEN Continue
ELSE Trap (VN)

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

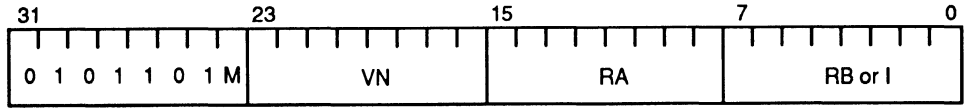
Operation: IF SRCA > SRCB (unsigned) THEN Continue
ELSE Trap (VN)

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

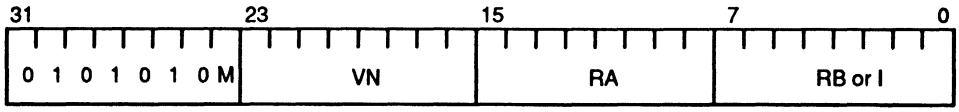
Operation: IF SRCA <= SRCB THEN Continue
ELSE Trap (VN)

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

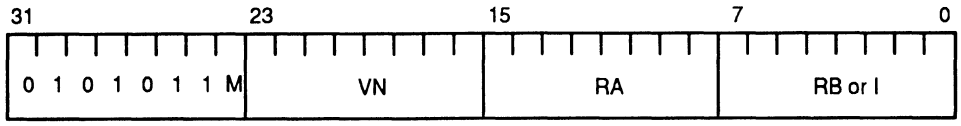
Operation: IF SRCA <= SRCB (unsigned) THEN Continue
ELSE Trap (VN)

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

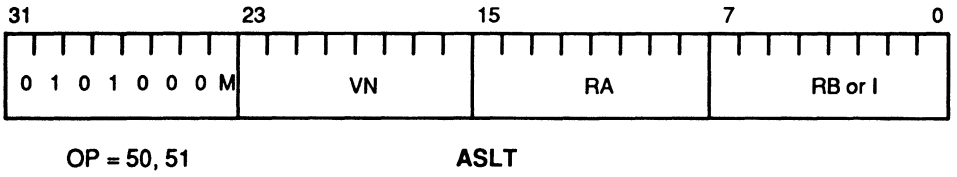
Operation: IF SRCA < SRCB THEN Continue
ELSE Trap(VN)

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



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

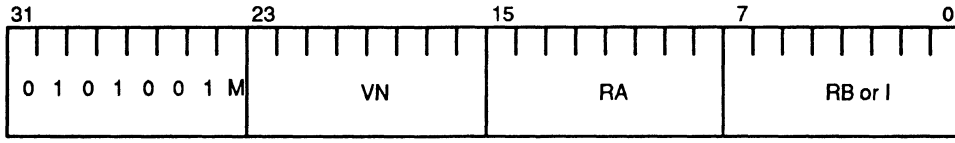
Operation: IF SRCA < SRCB (unsigned) THEN Continue
ELSE Trap (VN)

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

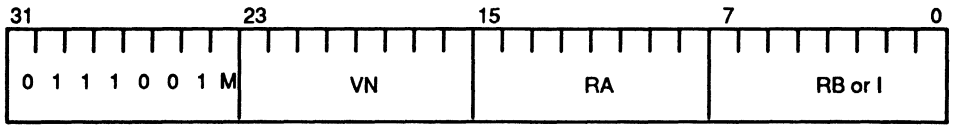
Operation: IF SRCA <> SRCB THEN Continue
ELSE Trap (VN)

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

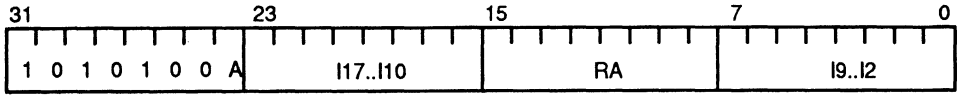
Operation: DEST <— PC//00 + 8
PC <— TARGET
Execute delay instruction

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



OP =A8, A9 **CALL**

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

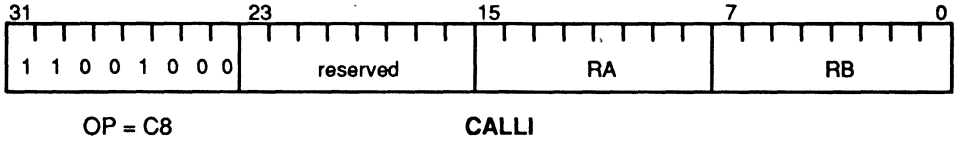
Operation: DEST <— PC//00 + 8
PC <— SRCB
Execute delay instruction

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

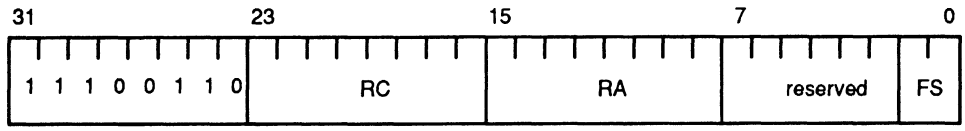
Status: None

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

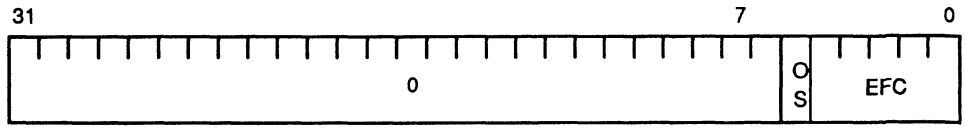
Control:

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 ("Max" is the largest biased exponent that can be used to represent a finite number. This exponent is 254 for the single-precision format and 2,046 for the double-precision format):

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: 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

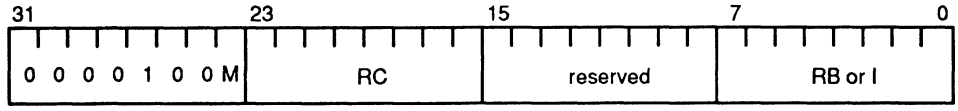
Operation: Determine number of leading zeros in a word

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

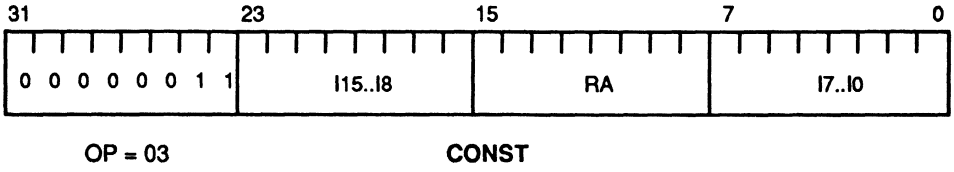
Operation: DEST <— 0I16

Assembler Syntax: CONST ra, const16

Status: Not affected

Operands:

0I16	I15..I8//I7..I0	(zero-extended to 32 bits)
DEST	register RA	



Description: The 0I16 operand is placed into the DEST location.

Constant, High

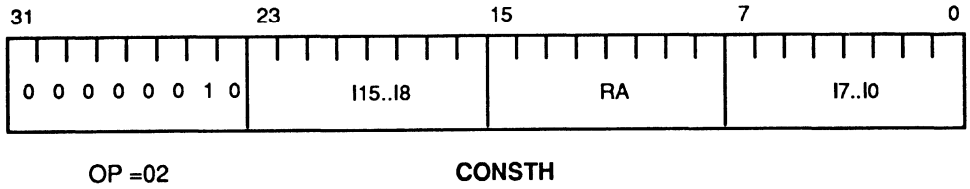
Operation: Replace high-order half-word of SRCA by I16

Assembler Syntax: CONSTH ra, const16

Status: Not affected

Operands:

SRCA	content of register RA
I16	I15..I8//I7..I0
DEST	register RA



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

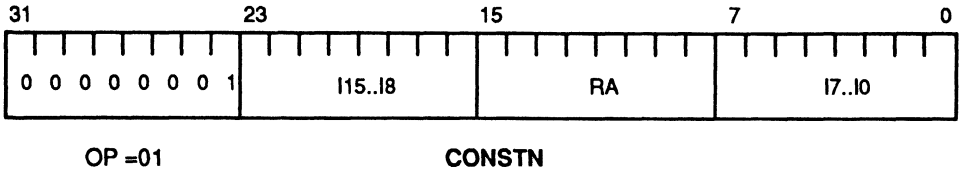
Operation: DEST <— 1I16

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

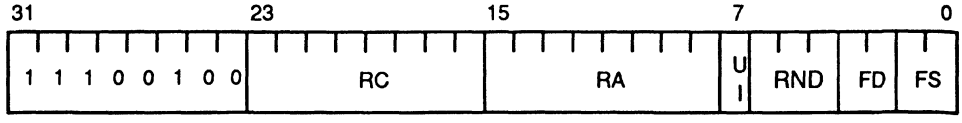
Status: fpX, fpU, fpV, fpR, fpN

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
```

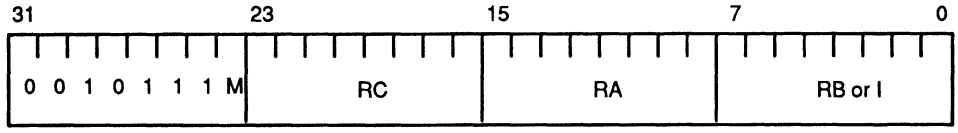
Assembler Syntax:

```
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

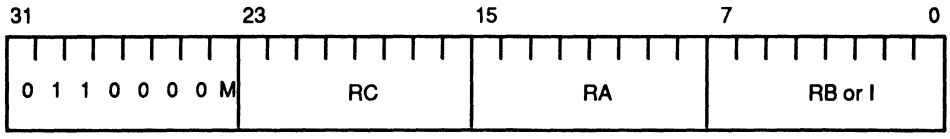
Operation: IF SRCA = SRCB THEN DEST <— TRUE
ELSE DEST <— FALSE

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

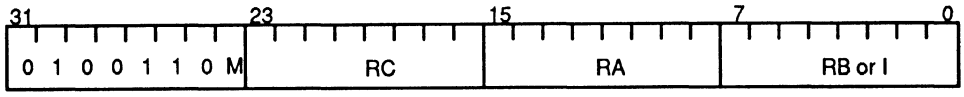
Operation: IF SRCA >= SRCB THEN DEST <— TRUE
ELSE DEST <— FALSE

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

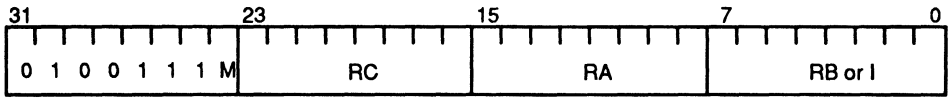
Operation: IF SRCA >= SRCB (unsigned) THEN DEST <— TRUE
ELSE DEST <— FALSE

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

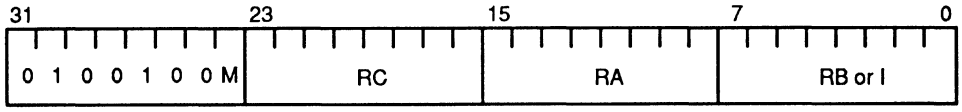
Operation: IF SRCA > SRCB THEN DEST <— TRUE
ELSE DEST <— FALSE

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

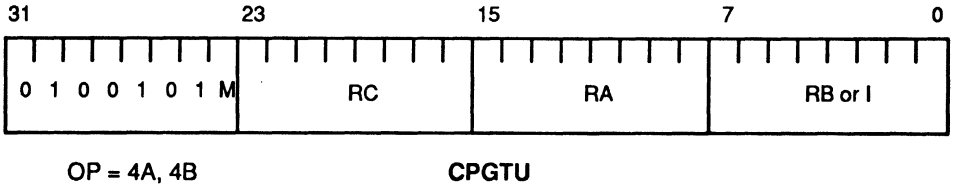
Operation: IF SRCA > SRCB (unsigned) THEN DEST <— TRUE
ELSE DEST <— FALSE

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

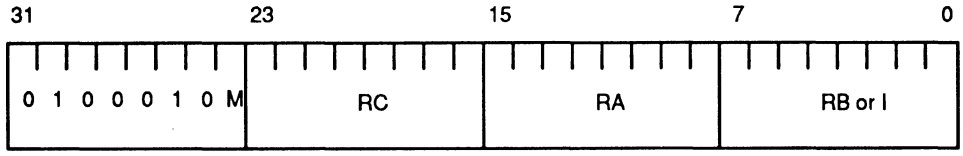
Operation: IF SRCA <= SRCB THEN DEST <— TRUE
ELSE DEST <— FALSE

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

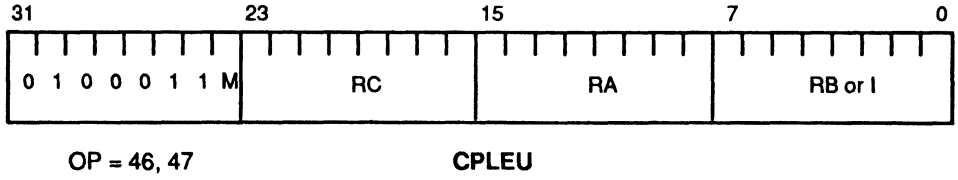
Operation: IF SRCA <= SRCB (unsigned) THEN DEST <— TRUE
ELSE DEST <— FALSE

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

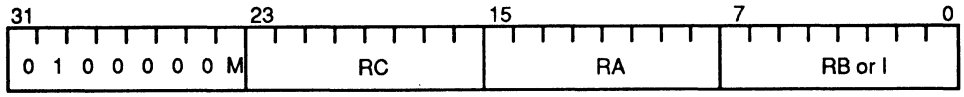
Operation: IF SRCA < SRCB THEN DEST <— TRUE
ELSE DEST <— FALSE

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



OP = 40, 41 **CPLT**

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

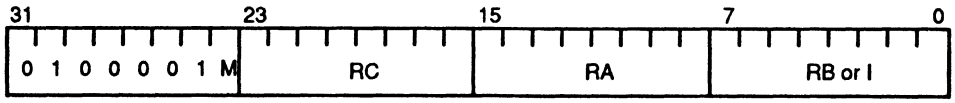
Operation: IF SRCA < SRCB (unsigned) THEN DEST <— TRUE
ELSE DEST <— FALSE

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

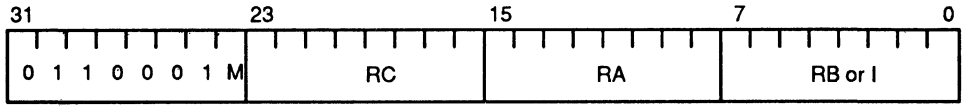
Operation: IF SRCA <> SRCB THEN DEST <— TRUE
ELSE DEST <— FALSE

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

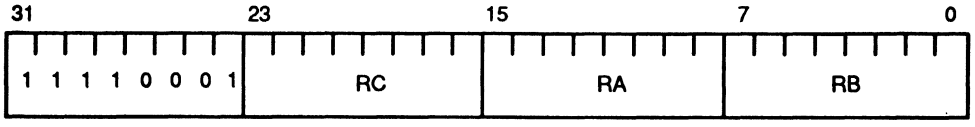
Operation: DEST (double-precision) <— SRCA (double-precision) + SRCB (double-precision)

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

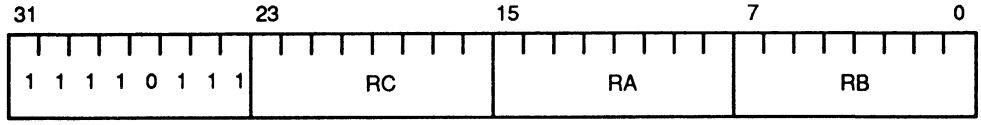
Operation: DEST (double-precision) <— SRCA (double-precision) / SRCB (double-precision)

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

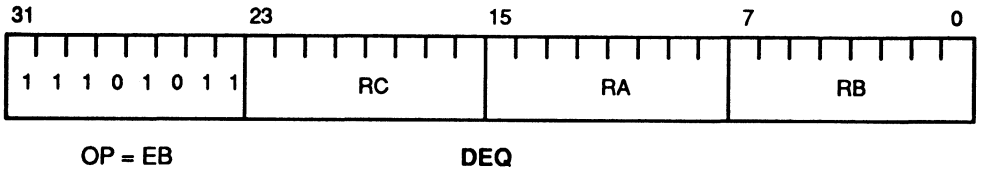
Operation: IF SRCA (double-precision) = SRCB (double-precision)
THEN DEST <— TRUE
ELSE DEST <— FALSE

Assembler Syntax: DEQ rc, ra, rb

Status: 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



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.

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

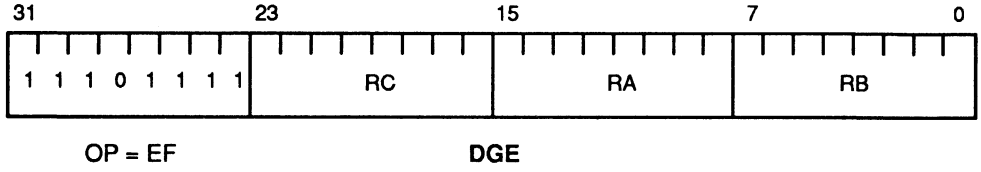
Operation: IF SRCA (double-precision) >= SRCB (double-precision)
THEN DEST <— TRUE
ELSE DEST <— FALSE

Assembler Syntax: DGE rc, ra, rb

Status: 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



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.

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

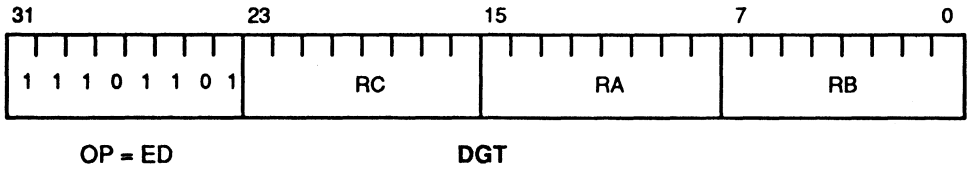
Operation: IF SRCA (double-precision) > SRCB (double-precision)
THEN DEST <— TRUE
ELSE DEST <— FALSE

Assembler Syntax: DGT rc, ra, rb

Status: 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



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.

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

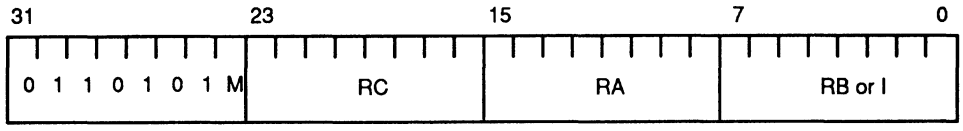
Operation: Perform one-bit step of a divide operation (unsigned)

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 7.2.6.

Divide Initialize

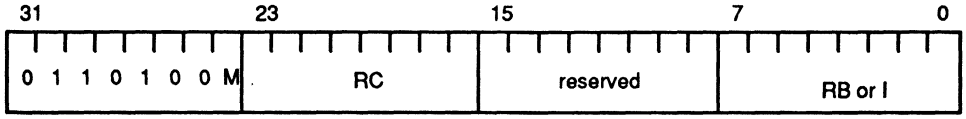
Operation: Initialize for a sequence of divide steps (unsigned)

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 7.2.6.

Integer Divide, Signed

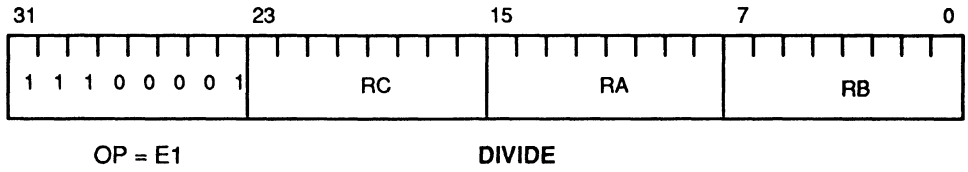
Operation: DEST <— (Q//SRCA) / SRCB (signed)
Q <— Remainder

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.

This instruction does not check for a divide overflow condition. Checking for divide overflow must occur before the instruction is executed.

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

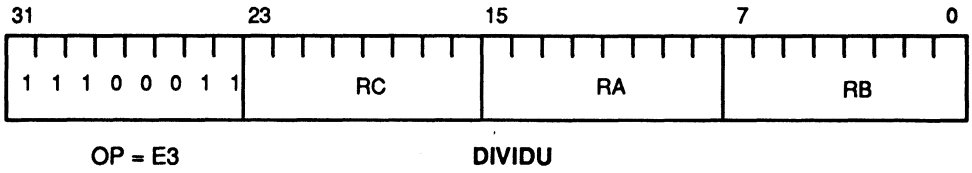
Operation: DEST <— (Q//SRCA) / SRCB (unsigned)
Q <— Remainder

Assember 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

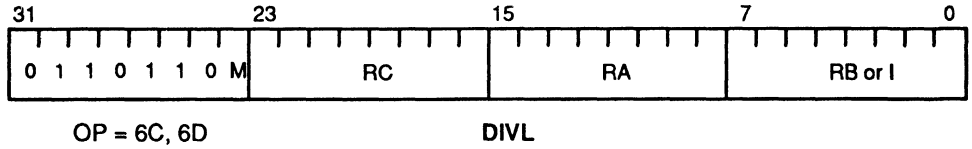
Operation: Complete a sequence of divide steps (unsigned)

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



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 7.2.6.

Divide Remainder

Operation: Generate remainder for divide operation (unsigned)

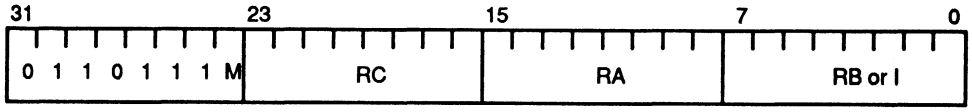
Assembler Syntax:
DIVREM rc, ra, rb
or
DIVREM rc, ra, const8

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 7.2.6.

Floating-Point Multiply, Double-Precision

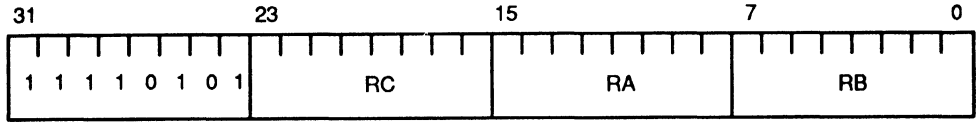
Operation: DEST (double-precision) <— SRCA (double-precision) * SRCB (double-precision)

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

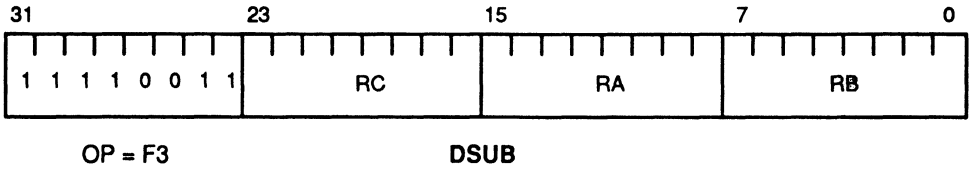
Operation: DEST (double-precision) <— SRCA (double-precision)
SRCB (double-precision)

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



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

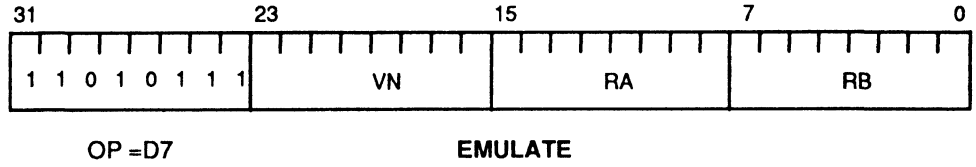
Operation: Load IPA and IPB registers with operand register-numbers and Trap (VN)

Assembler Syntax: EMULATE vn, ra, rb

Status: Not affected

Operands: Absolute-register numbers for registers RA and RB

VN Trap vector number



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

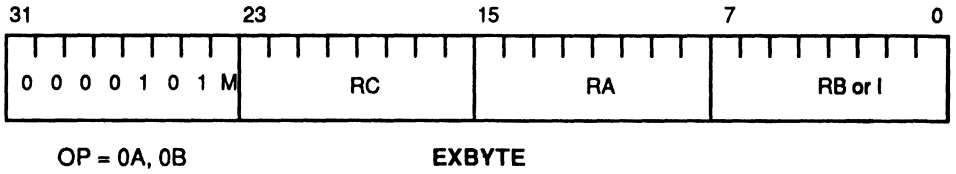
Assembler Syntax:
EXBYTE rc, ra ,rb
or
EXBYTE rc, ra, const8

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



Description: A byte in the SRCA operand is selected by the Byte Position field of the ALU Status Register and the Byte Order 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 7.2.6.

Extract Half-Word

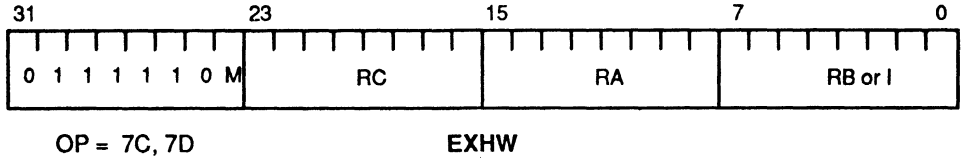
Operation: DEST <— SRCB, with low-order half-word replaced by half-word in SRCA selected by BP

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 Position field of the ALU Status Register and the Byte Order 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 3.4.5.

Extract Half-Word, Sign-Extended

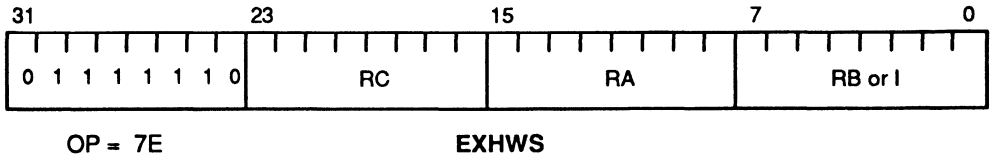
Operation: DEST <— half-word in SRCA selected by BP, sign-extended to 32 bits

Assembler Syntax: EXHWS rc, ra

Status: Not affected

Operands: SRCA content of register RA

DEST register RC



Description: A half-word in the SRCA operand is selected by theByte Position field of the ALU Status Register and the Byte Order 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 3.4.5.

Extract Word, Bit-Aligned

Operation: DEST <— high-order word of (SRCA//SRCB << FC)

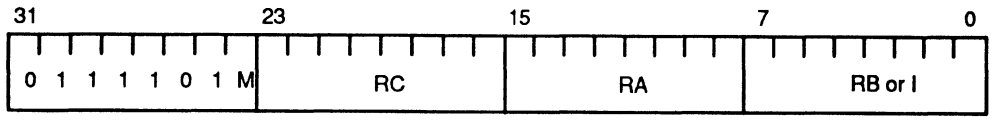
Assembler Syntax:
EXTRACT rc, ra ,rb
or
EXTRACT rc, ra, const8

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

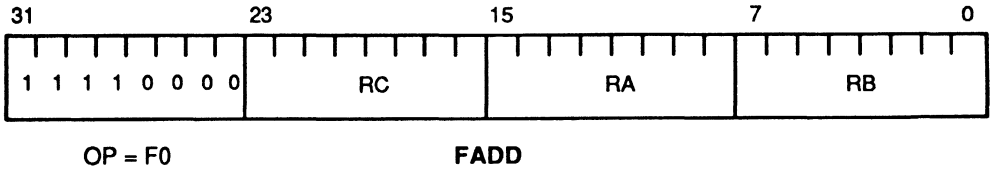
Operation: DEST (single-precision) <— SRCA (single-precision) – SRCB (single-precision)

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



Description: The SRCB operand is added to 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 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

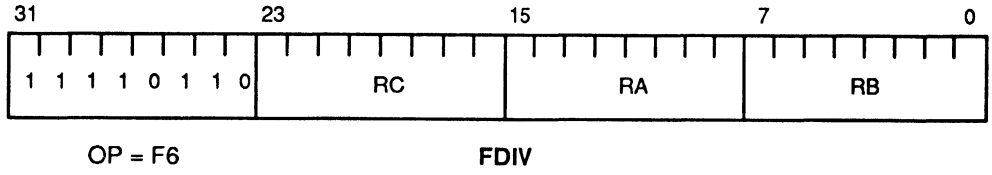
Operation: DEST (single-precision) <— SRCA (single-precision) / SRCB (single-precision)

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



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 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

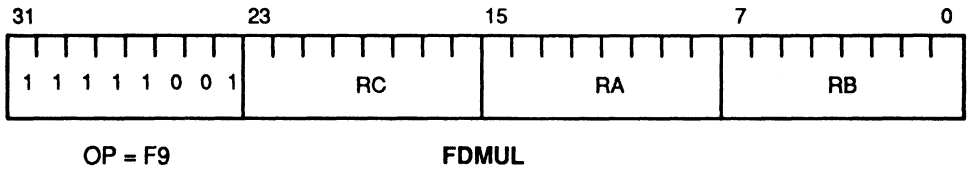
Operation: DEST (double-precision) <— SRCA (single-precision) * SRCB (single-precision)

Assembler Syntax: FDMUL rc, ra, rb

Status: fpR, fpN

Operands:

SRCA	content of register RA
SRCB	content of register RB
DEST	register RC



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

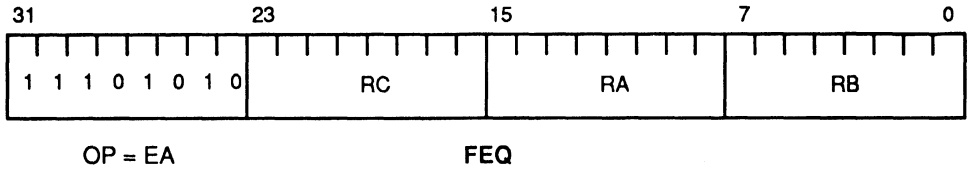
Operation: IF SRCA (single-precision) = SRCB (single-precision)
THEN DEST <— TRUE
ELSE DEST <— FALSE

Assembler Syntax: FEQ rc, ra, rb

Status: fpN

Operands:

SRCA	content of register RA
SRCB	content of register RB
DEST	register RC



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.

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

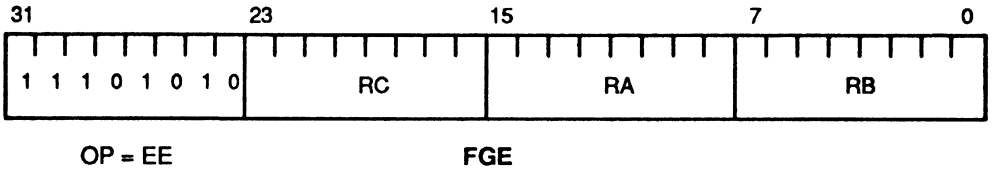
Operation: IF SRCA (single-precision) >= SRCB (single-precision)
THEN DEST <— TRUE
ELSE DEST <— FALSE

Assembler Syntax: FGE rc, ra, rb

Status: fpN

Operands:

SRCA	content of register RA
SRCB	content of register RB
DEST	register RC



)escription: 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.

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

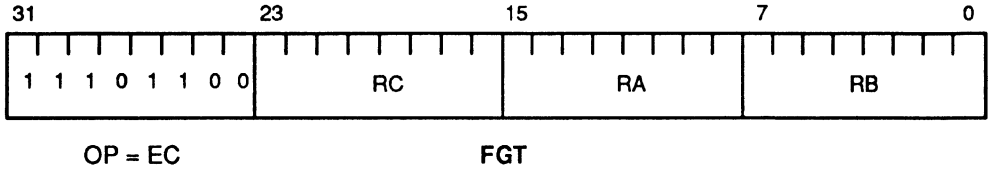
Operation: IF SRCA (single-precision) > SRCB (single-precision)
THEN DEST <— TRUE
ELSE DEST <— FALSE

Assembler Syntax: FGT rc, ra, rb

Status: fpN

Operands:

SRCA	content of register RA
SRCB	content of register RB
DEST	register RC



Description: If the SRCA operand is greater than the SRCB operand, a Boolean TRUE is place into the DEST location; otherwise, a Boolean FALSE is placed into the DES location. SRCA and SRCB are single-precision floating-point numbers.

Note: This instruction is not supported directly in processor hardware. In th current implementation, this instruction causes an FGT trap. When the tra occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, an DEST.

Floating-Point Multiply, Single-Precision

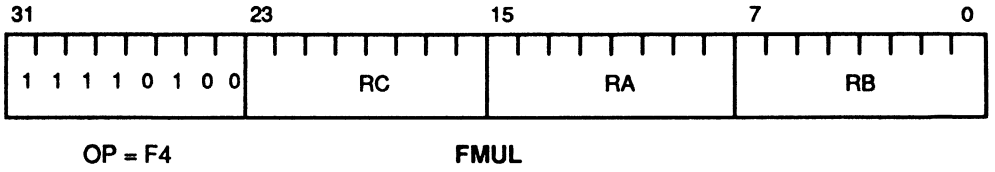
Operation: DEST (single-precision) <— SRCA (single-precision) * SRCB (single-precision)

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



Description: The SRCA operand is multiplied 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 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

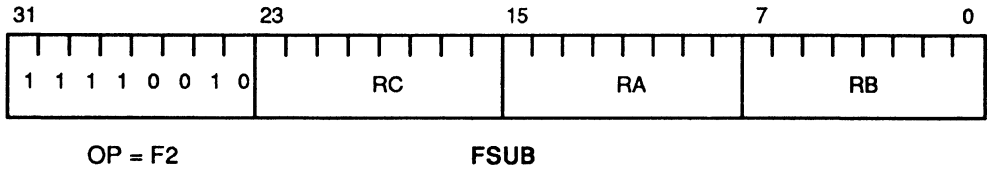
Operation: DEST (single-precision) <— SRCA (single-precision) – SRCB (single-precision)

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



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 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.

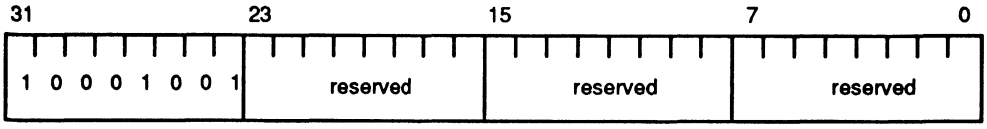
Enter Halt Mode

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.

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

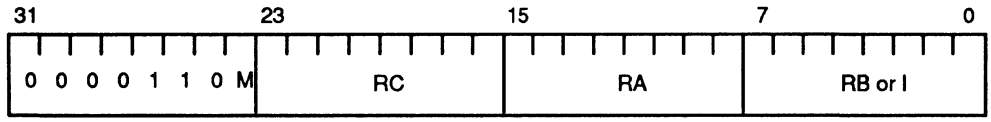
Operation: DEST <— SRCA, with byte selected by BP replaced by low-order byte of SRCB

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 Position field of the ALU Status Register and the Byte Order 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 3.4.5.

Insert Half-Word

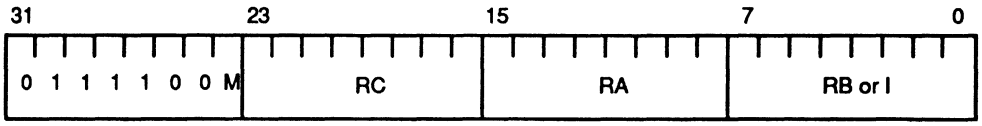
Operation: DEST <— SRCA, with half-word selected by BP replaced by low-order half-word of SRCB

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 Position field of the ALU Status Register and the Byte Order 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 3.4.5.

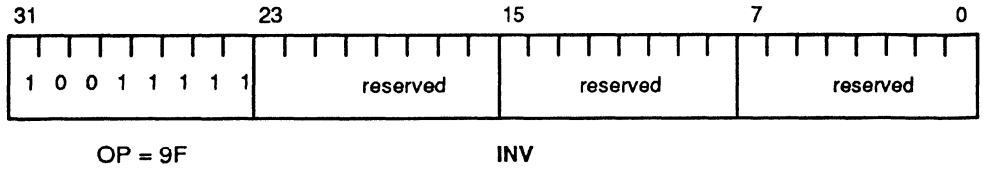
Invalidate

Operation: Reset all Valid bits in Branch Target Cache

Assembler Syntax: INV

Status: Not affected

Operands: Not applicable



Description: This instruction causes all Branch Target Cache Valid bits to be reset, on the execution of the next successful branch. This causes all Branch Target Cache locations 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.

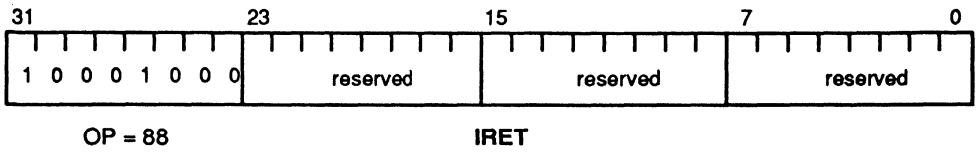
Interrupt Return

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 3.5.5.

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.

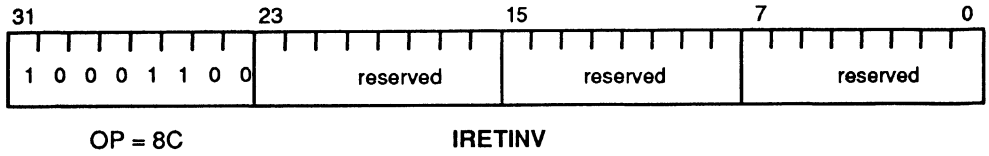
Interrupt Return and Invalidate

Operation: Perform an interrupt return sequence, and reset all valid bits in Branch Target Cache

Assembler Syntax: IRETINV

Status: Not affected

Operands: Not applicable



Description: This instruction performs the interrupt return sequence described in Section 3.5.5. When the sequence begins, all Branch Target Cache Valid bits are reset to zeros. This causes all Branch Target Cache locations 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.

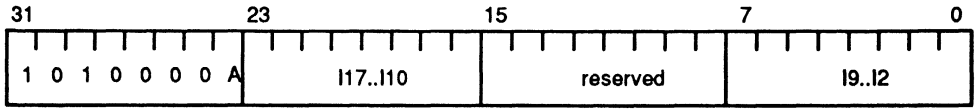
Jump

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

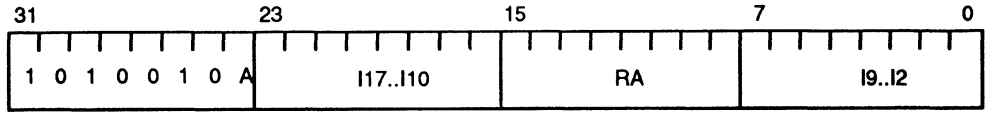
Operation: IF SRCA = FALSE THEN PC <— TARGET
Execute delay instruction

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

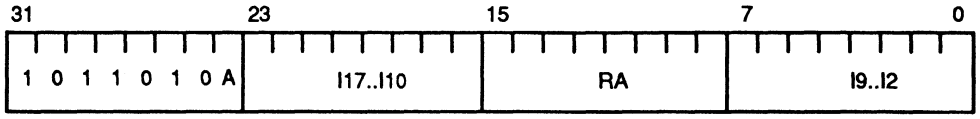
Operation:
```
IF SRCA = FALSE THEN
      SRCA <— SRCA - 1
      PC <— TARGET
ELSE
      SRCA <— SRCA - 1
Execute delay instruction
```

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

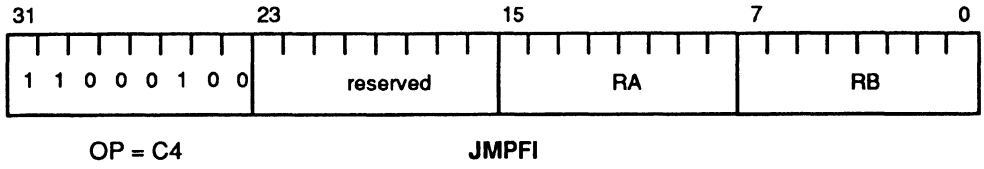
Operation: IF SRCA = FALSE THEN PC <— SRCB
Execute delay instruction

Assembler Syntax: JMPFI ra, rb

Status: Not affected

Operands: SRCA content of register RA

SRCB content of register RB



Description: 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.

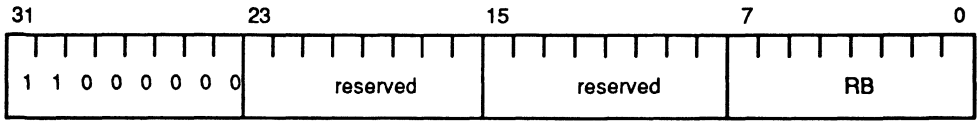
Jump Indirect

Operation: PC <— SRCB
Execute delay instruction

Assembler Syntax: JMPI rb

Status: Not affected

Operands: SRCB content of register RB



OP = C0 **JMPI**

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

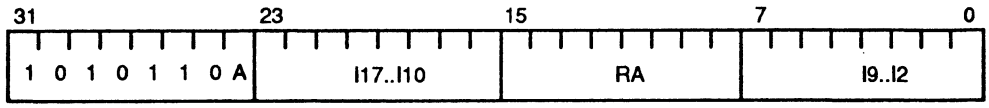
Operation: IF SRCA = TRUE THEN PC <— TARGET
Execute delay instruction

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

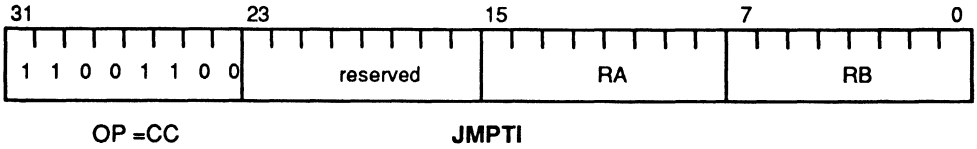
Operation: IF SRCA = TRUE THEN PC <— SRCB
Execute delay instruction

Assembler Syntax: JMPTI ra, rb

Status: Not affected

Operands: SRCA content of register RA

SRCB content of register RB



Description: 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

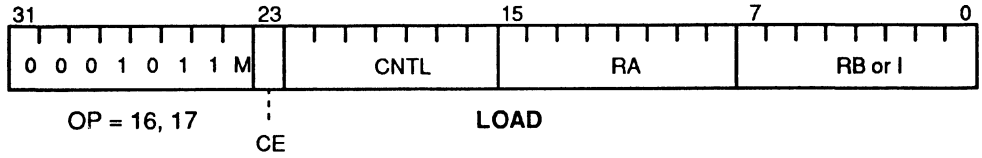
Operation: DEST <— EXTERNAL WORD [SRCB]

Assembler Syntax:
LOAD ce, cntl, ra, rb
or
LOAD ce, 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: If the CE bit is 0, the external word addressed by the SRCB operand is placed into the DEST location.

If the CE bit is 1, a word is transferred from the coprocessor into the DEST location. The SRCB operand has no pre-defined interpretation in this case, though it appears on the address bus.

The CNTL field of the LOAD instruction affects the access or transfer as described in Sections 3.4.4 and 6.1.2.

Load and Lock

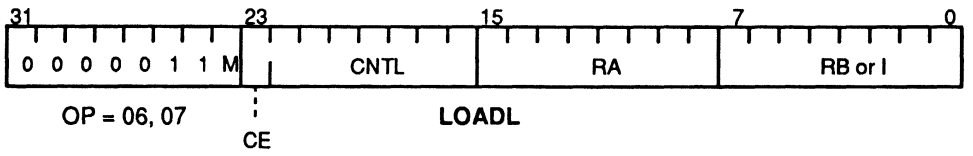
Operation: DEST <— EXTERNAL WORD [SRCB],
assert *LOCK output during access

Assembler Syntax:
LOADL ce, cntl, ra, rb
or
LOADL ce, 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: If the CE bit is 0, the external word addressed by the SRCB operand is placed into the DEST location.

If the CE bit is 1, a word is transferred from the coprocessor into the DEST location. The SRCB operand has no pre-defined interpretation in this case, though it appears on the address bus.

The CNTL field of the LOADL instruction affects the access or transfer as described in Sections 3.4.4 and 6.1.2.

The *LOCK output is asserted during the access or transfer.

Load Multiple

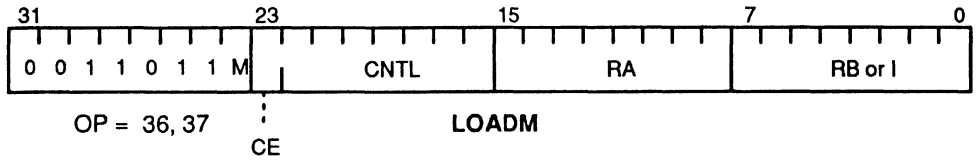
Operation: DEST ... DEST+COUNT <— EXTERNAL WORD [SRCB] ...
EXTERNAL WORD [SRCB+(COUNT* 4)]

Assembler Syntax:
LOADM ce, cntl, ra, rb
or
LOADM ce, 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: If the CE bit is 0, 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.

If the CE bit is 1, multiple words are transferred from the coprocessor into consecutive registers, beginning with the DEST location. The SRCB operand has no pre-defined interpretation in this case.

The total number of words accessed or transferred 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 access. The total number of words is the value of the CR field plus one. The CNTL field of the LOADM instruction affects the access or transfer as described in Sections 3.4.4 and 6.1.2.

Note: The address and register-number sequences for the LOADM instruction are specified in Section 3.4.4.

Load and Set

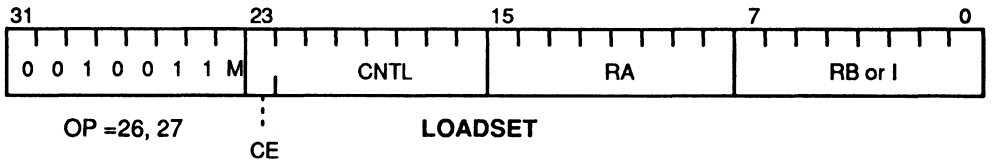
Operation: DEST <— EXTERNAL WORD [SRCB]
EXTERNAL WORD [SRCB] <— h'FFFFFFFF',
assert *LOCK output during access

Assembler Syntax:
LOADSET ce, cntl, ra, rb
or
LOADSET ce, 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: If the CE bit is 0, 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.

If the CE bit is 1, a word is transferred from the coprocessor into the DEST location. The SRCB operand has no pre-defined interpretation in this case, though it appears on the Address Bus. After the DEST location is altered, a word consisting of a 1 in every bit position is transferred, atomically, to the coprocessor.

The CNTL field of the LOADSET instruction affects the access or transfer as described in Sections 3.4.4 and 6.1.2.

The *LOCK output is asserted throughout the LOADSET operation.

Move from Special Register

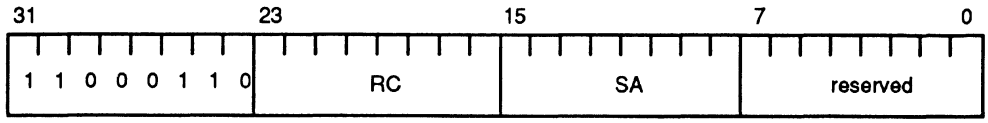
Operation: DEST <— SPECIAL

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

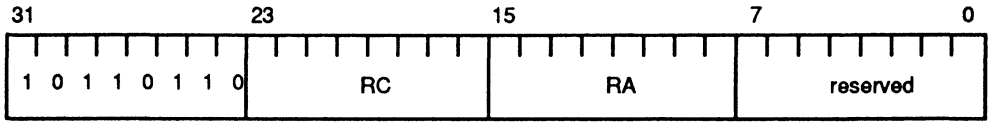
Operation: DEST <— TLB [SRCA]

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 SCRA 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

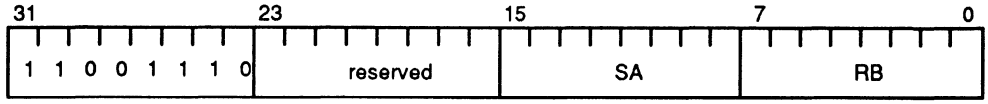
Operation: SPDEST <— SRCB

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



OP = CE **MTSR**

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

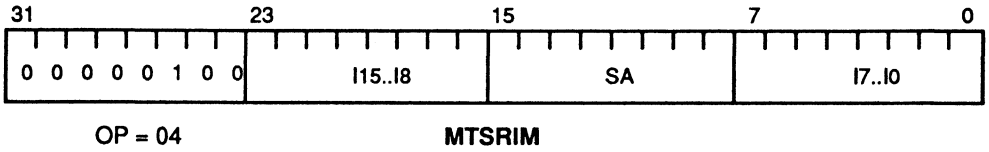
Operation: SPDEST <— 0I16

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.

Multiply Last Step

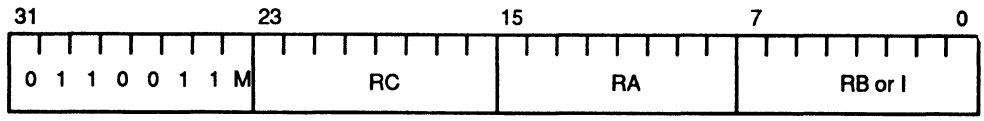
Operation: Complete a sequence of multiply steps (for signed multiply)

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 7.2.5.

Integer Multiply, Unsigned

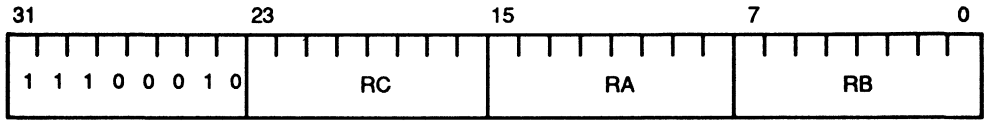
Operation: DEST <— SRCA * SRCB

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.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes a MULTIPLU trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Integer Multiply, Signed

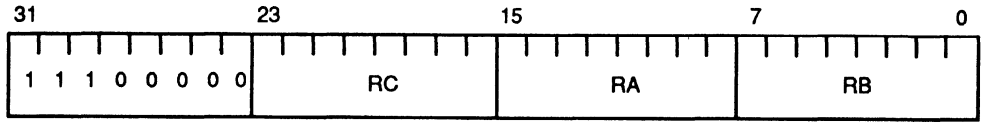
Operation: DEST <— SRCA * SRCB

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.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes a 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

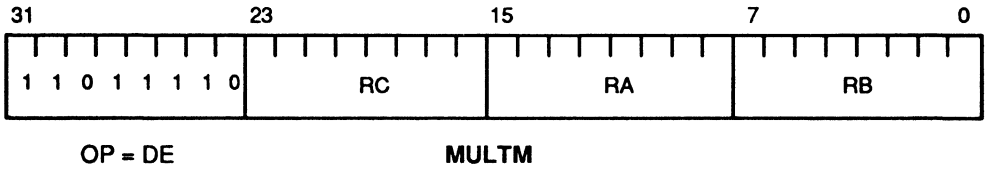
Operation: DEST <— SRCA * SRCB

Assembler Syntax: MULTM rc, ra, rb

Status: None

Operands:

SRCA	content of register RA
SRCB	content of register RB
DEST	register RC



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.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes a 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

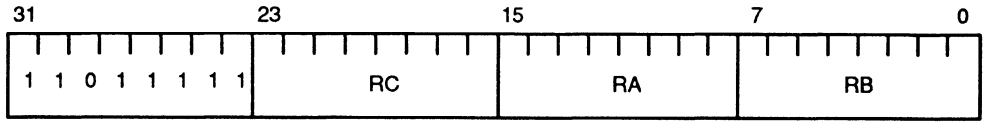
Operation: DEST <— SRCA * SRCB

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.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes a MULTMU trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Multiply Step, Unsigned

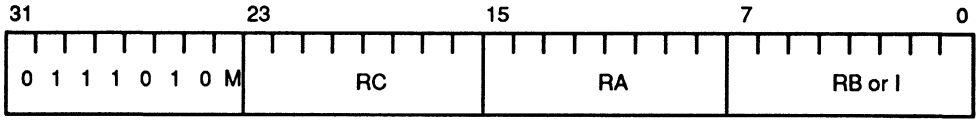
Operation: Perform one-bit step of a multiply operation (unsigned)

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.

Examples of integer multiply operations appear in Section 7.2.5.

NAND Logical

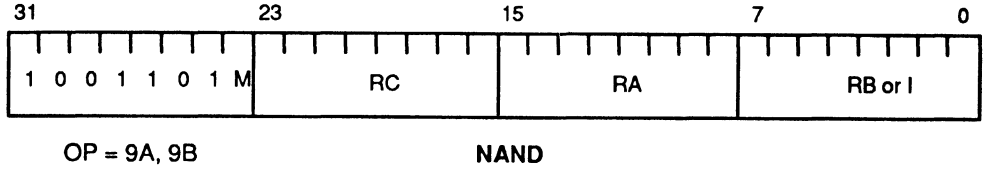
Operation: DEST <— ~(SRCA & SRCB)

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



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

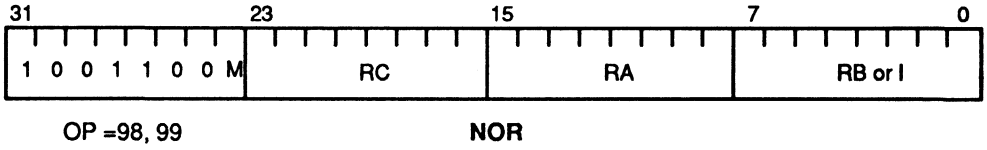
Operation: DEST <— ~(SRCA | SRCB)

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



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

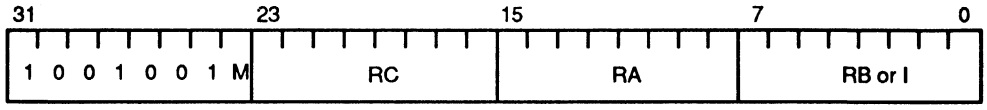
Operation: DEST <— SRCA | SRCB

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.

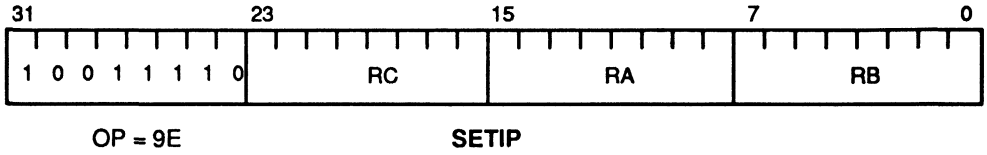
Set Indirect Pointers

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.

Shift Left Logical

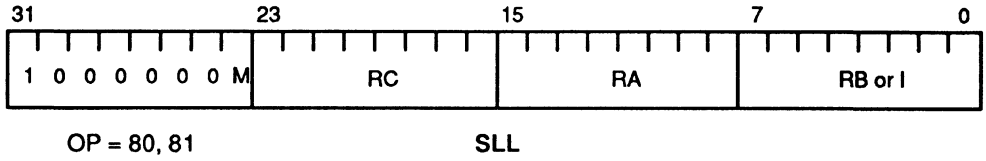
Operation: DEST <— SRCA << SRCB (zero fill)

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



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.

Square Root

Operation: DEST <— SQRT(SRCA)

Assembler Syntax: SQRT rc, ra, FS

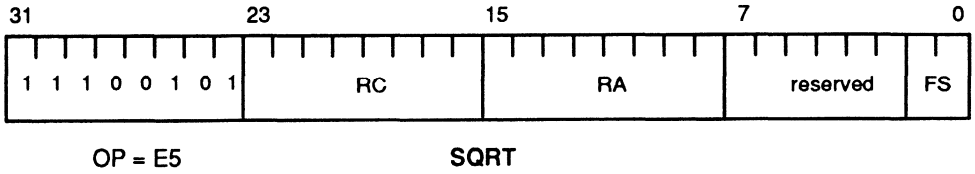
Status: fpX, fpR, fpN

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



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

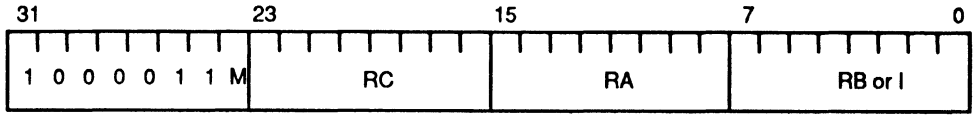
Operation: DEST <— SRCA >> SRCB (sign fill)

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

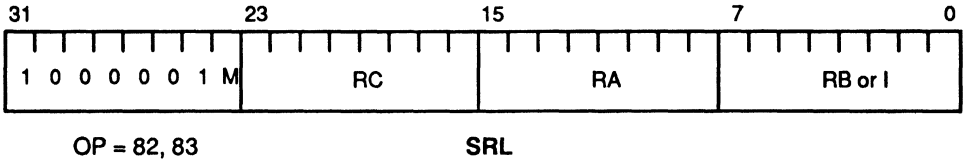
Operation: DEST <— SRCA >> SRCB (zero fill)

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



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

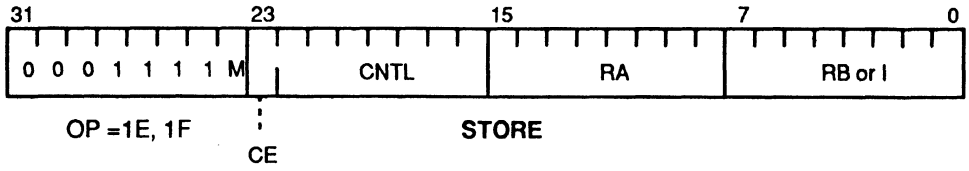
Operation: EXTERNAL WORD [SRCB] <— SRCA

Assembler Syntax:
STORE ce, cntl, ra, rb
or
STORE ce, 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: If the CE bit is 0, the SRCA operand is placed into the external word addressed by the SRCB operand.

If the CE bit is 1, the SRCA and SRCB operands are transferred to the coprocessor.

The CNTL field of the STORE instruction affects the access or transfer as described in Sections 3.4.4 and 6.1.2.

Store and Lock

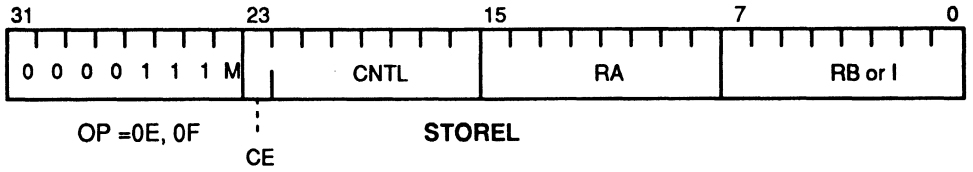
Operation: EXTERNAL WORD [SRCB] <— SRCA,
assert *LOCK output during access

Assembler Syntax:
STOREL ce, cntl, ra, rb
or
STOREL ce, 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: If the CE bit is 0, the SRCA operand is placed into the external word addressed by the SRCB operand.

If the CE bit is 1, the SRCA and SRCB operands are transferred to the coprocessor.

The CNTL field of the STOREL instruction affects the access or transfer as described in Sections 3.4.4 and 6.1.2.

The *LOCK output is asserted during the access or transfer.

Store Multiple

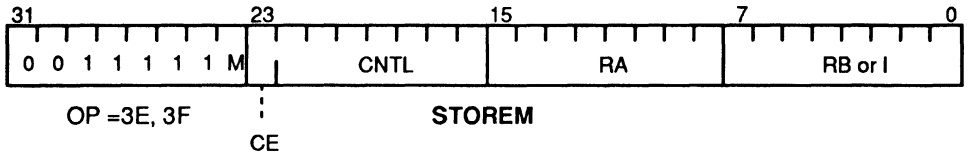
Operation: EXTERNAL WORD [SRCB] ... EXTERNAL WORD [SRCB+(COUNT*4)] <— SRCA ... SRCA+COUNT

Assembler Syntax:
STOREM ce, cntl, ra, rb
or
STOREM ce, 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: If the CE bit is 0, 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.

If the CE bit is 1, the contents of consecutive registers, beginning with the SRCA operand, are transferred to the coprocessor. The SRCB operand has no pre-defined interpretation in this case.

The total number of words accessed or transferred in the sequence is specified by the Count Remaining field (CR) of the Channel Control Register (which also appears in the Load/Store Count Remaining Register) at the beginning of the 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 or transfer as described in Sections 3.4.4 and 6.1.2.

Note: The address and register-number sequences for the STOREM instruction are specified in Section 3.4.4.

Subtract

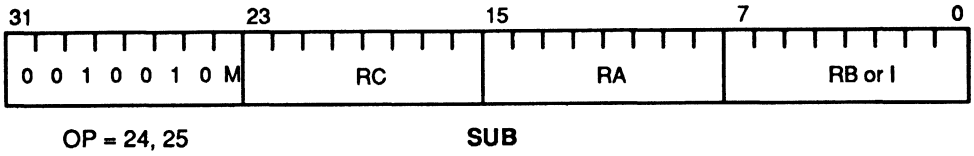
Operation: DEST <— SRCA – SRCB

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



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

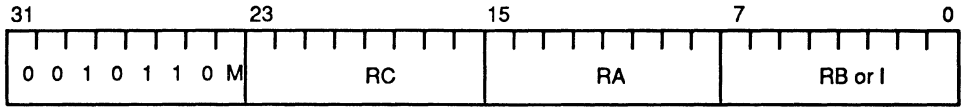
Operation: DEST <— SRCA – SRCB – 1 + C

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

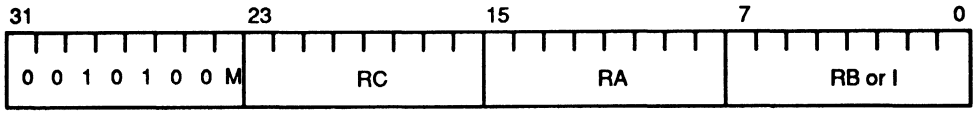
Operation: DEST <— SRCA – SRCB – 1 + C
IF signed overflow THEN Trap (Out of Range)

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

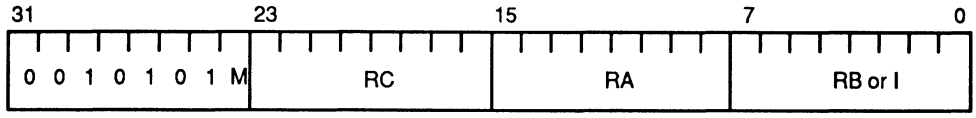
Operation: DEST <— SRCA – SRCB – 1 + C
IF unsigned underflow THEN Trap (Out of Range)

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

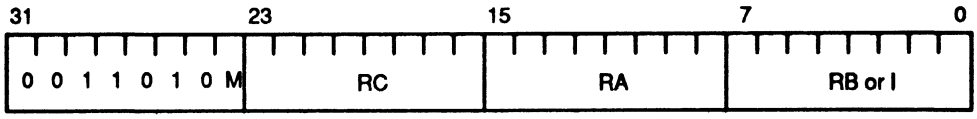
Operation: DEST <— SRCB – SRCA

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

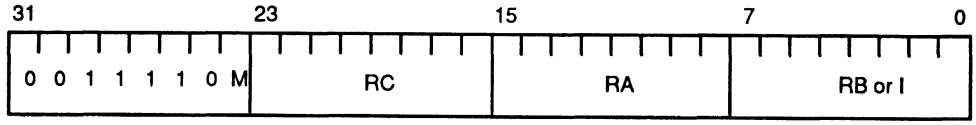
Operation: DEST <— SRCB – SRCA – 1 + C

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

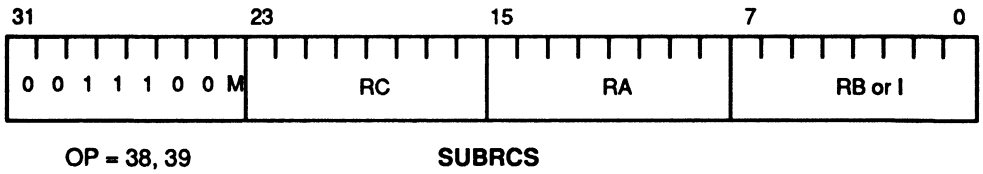
Operation: DEST <— SRCB – SRCA – 1 + C
IF signed overflow THEN Trap (Out of Range)

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



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

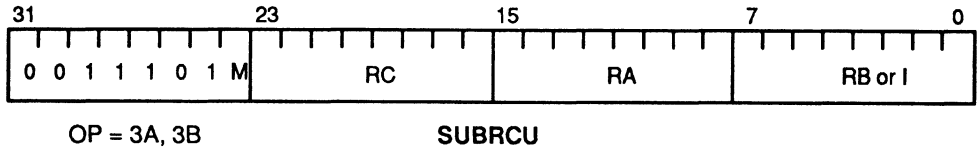
Operation: DEST <— SRCB – SRCA – 1 + C
IF unsigned underflow THEN Trap (Out of Range)

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



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

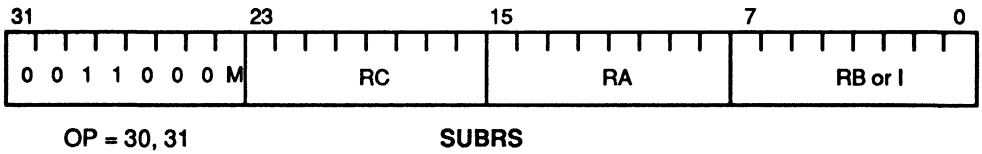
Operation: DEST <— SRCB – SRCA
IF signed overflow THEN Trap (Out of Range)

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



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

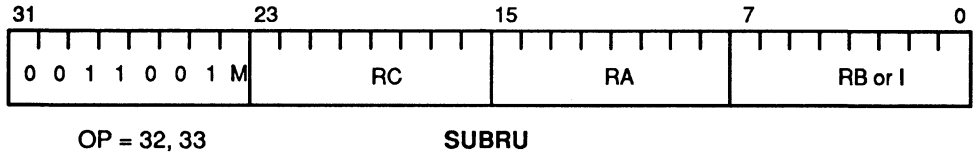
Operation: DEST <— SRCB – SRCA
IF unsigned underflow THEN Trap (Out of Range)

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



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

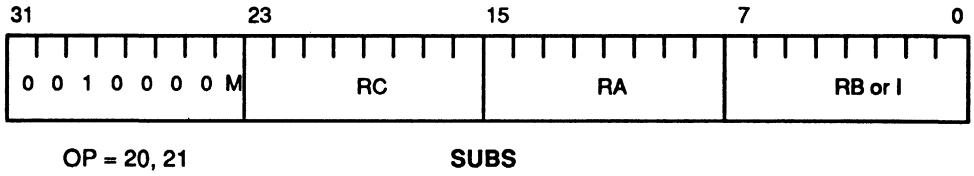
Operation: DEST <— SRCA – SRCB
IF signed overflow THEN Trap (Out of Range)

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



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

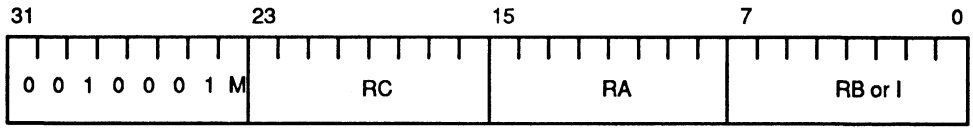
Operation: DEST <— SRCA – SRCB
IF unsigned underflow THEN Trap (Out of Range)

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

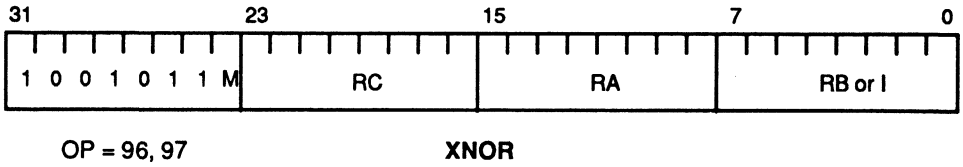
Operation: DEST <— ~(SRCA ^ SRCB)

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



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

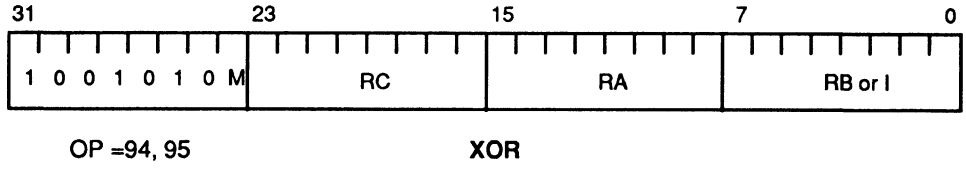
Operation: DEST <— SRCA ^ SRCB

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



Description: The SRCA operand is logically exclusive-ORed, bit-by-bit, with the SRCB operand, and the result is placed into the DEST location.

8.5 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,OF	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 numbers 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)	

Appendices

APPENDIX A. CHANNEL OPERATION TIMING

Table A-1. Signal Summary

Signal Name	Signal Function	Type (1)	Synch Async
A0–A31	Address Bus	3-state output	synch
*BGRT	Bus Grant	output	synch
*BINV	Bus Invalid	output	synch
*BREQ	Bus Request	input	synch
*CDA	Coprocessor Data Accept	input	synch
CNTL0–CNTL1	CPU Control	input	async
D0–D31	Data Bus	bi-directional	synch
*DBACK	Data Burst Acknowledge	input	synch
*DBREQ	Data Burst Request	3-state output	synch
*DERR	Data Error	input	synch
*DRDY	Data Ready	input	synch
*DREQ	Data Request	3-state output	synch
DREQT0–DREQT1	Data Request Type	3-state output	synch
I0–I31	Instruction Bus	input	synch
*IBACK	Instruction Burst Acknowledge	input	synch
*IBREQ	Instruction Burst Request	3-state output	synch
*IERR	Instruction Error	input	synch
INCLK	Input Clock	input	N/A
*INTR0–*INTR3	Interrupt Request	input	async
*IRDY	Instruction Ready	input	synch
*IREQ	Instruction Request	3-state output	synch
IREQT	Instruction Request Type	3-state output	synch

(Table continued)

(1) The signals labeled "3-state output" and "bi-directional" (except SYSCLK) are disabled when the channel is granted to an external master. All outputs (except MSERR) may be disabled by asserting the *TEST input.

Table A-1. Signal Summary (continued)

Signal Name	Signal Function	Type (1)	Synch Async
*LOCK	Lock	3-state output	synch
MPGM0–MPGM1	MMU Programmable	3-state output	synch
MSERR	Master/Slave Error	output	synch
OPT0–OPT2	Option Control	3-state output	synch
*PDA	Pipelined Data Access	3-state output	synch
*PEN	Pipeline Enable	input	synch
PIN169	ADAPT29K use	alignment	n/a
*PIA	Pipelined Instruction Access	3-state output	synch
PWRCLK	n/a	SYSCLK power	n/a
R/*W	Read/Write	3-state output	synch
*RESET	Reset	input	async
STAT0–STAT2	CPU Status	output	synch
SUP/*US	Supervisor/User Mode	3-state output	synch
SYSCLK	System Clock	bi-directional	N/A
*TEST	Test Mode	input	async
*TRAP0–*TRAP1	Trap Request	input	async
*WARN	Warn	edge-sensitive input	async

(1) The signals labeled "3-state output" and "bi-directional" (except SYSCLK) are disabled when the channel is granted to an external master. All outputs (except MSERR) may be disabled by asserting the *TEST input.

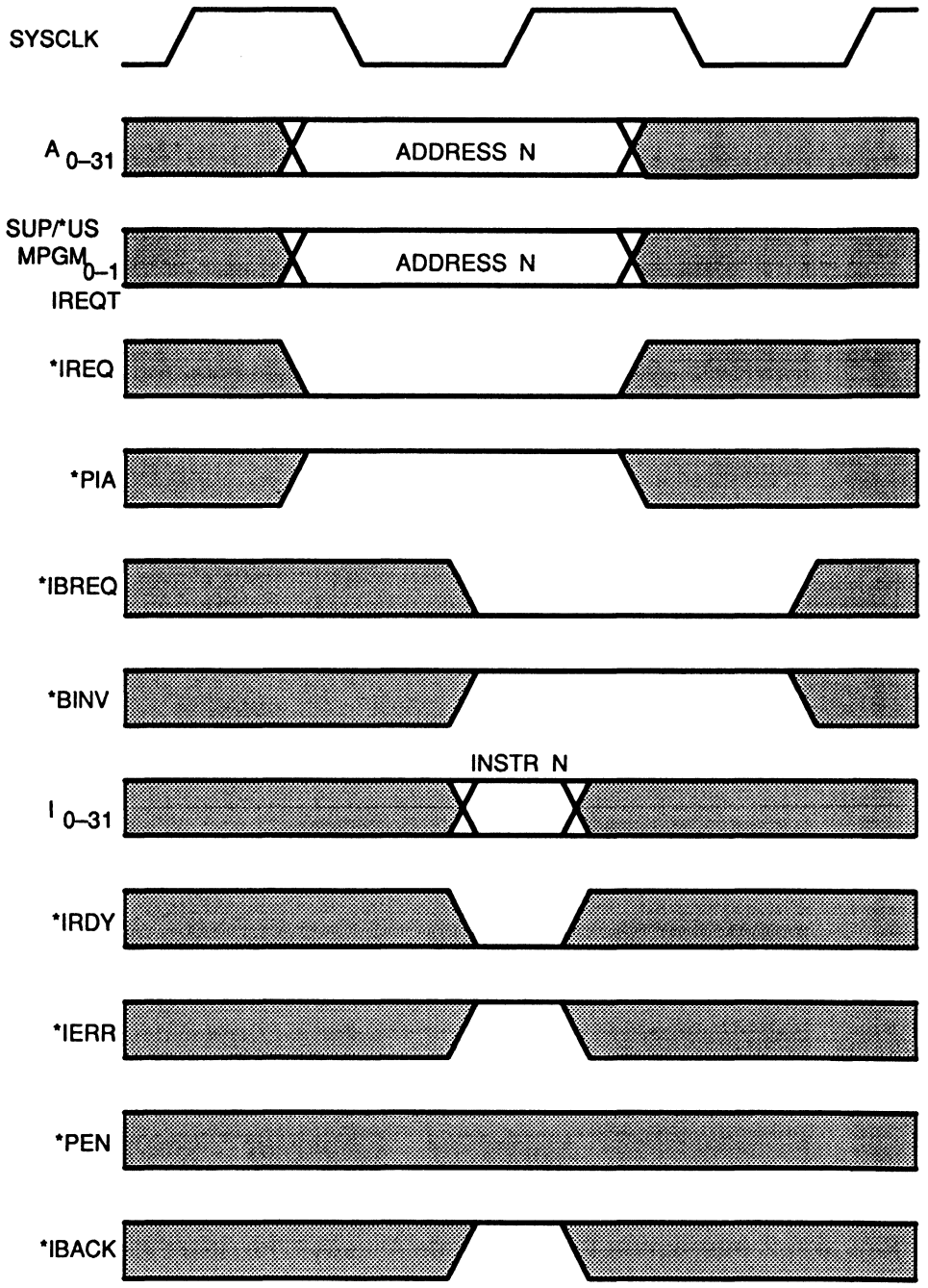


Figure A-1. Instruction Read—Simple Access

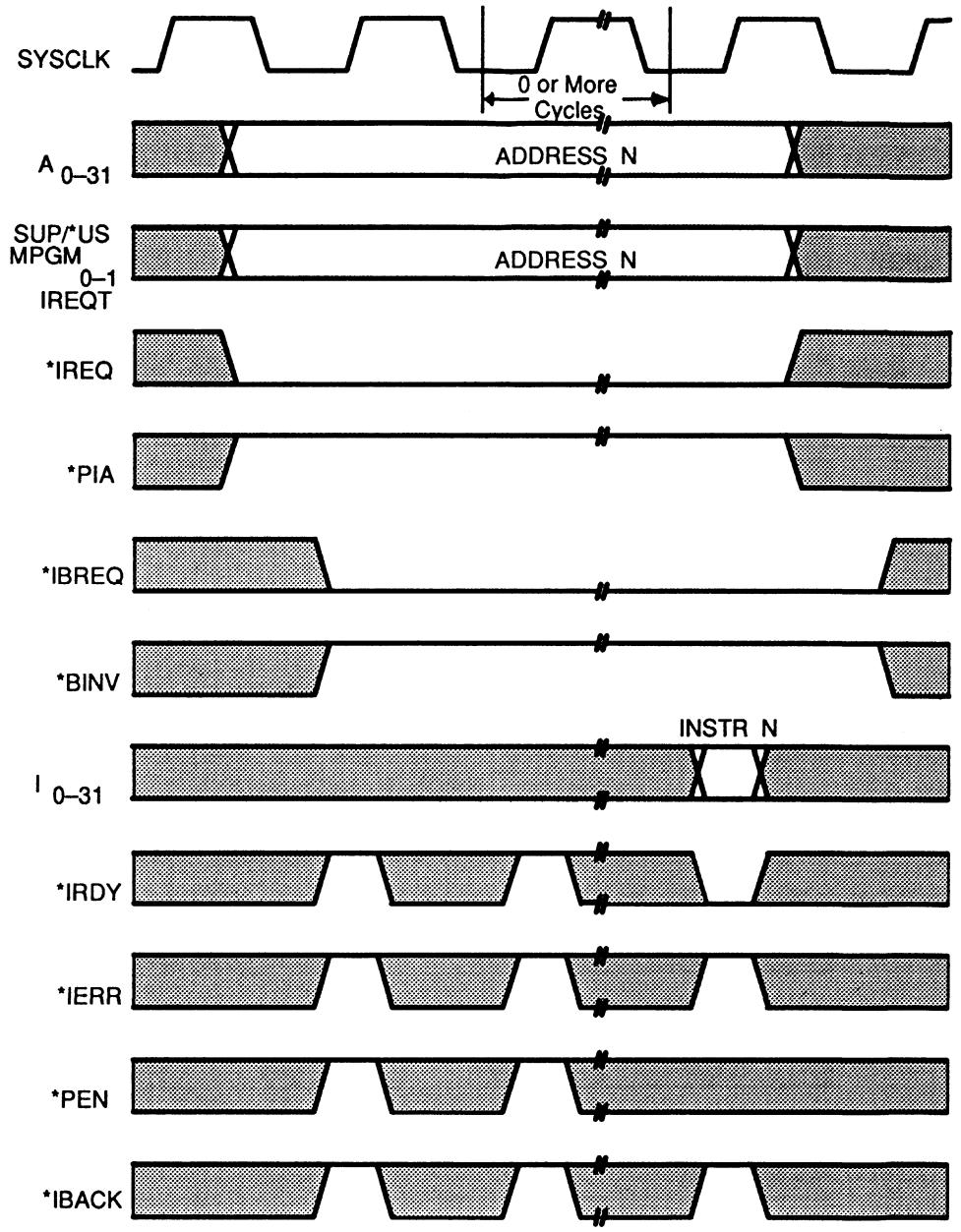


Figure A-2. Instruction Read—Simple Access With *IRDY Delayed

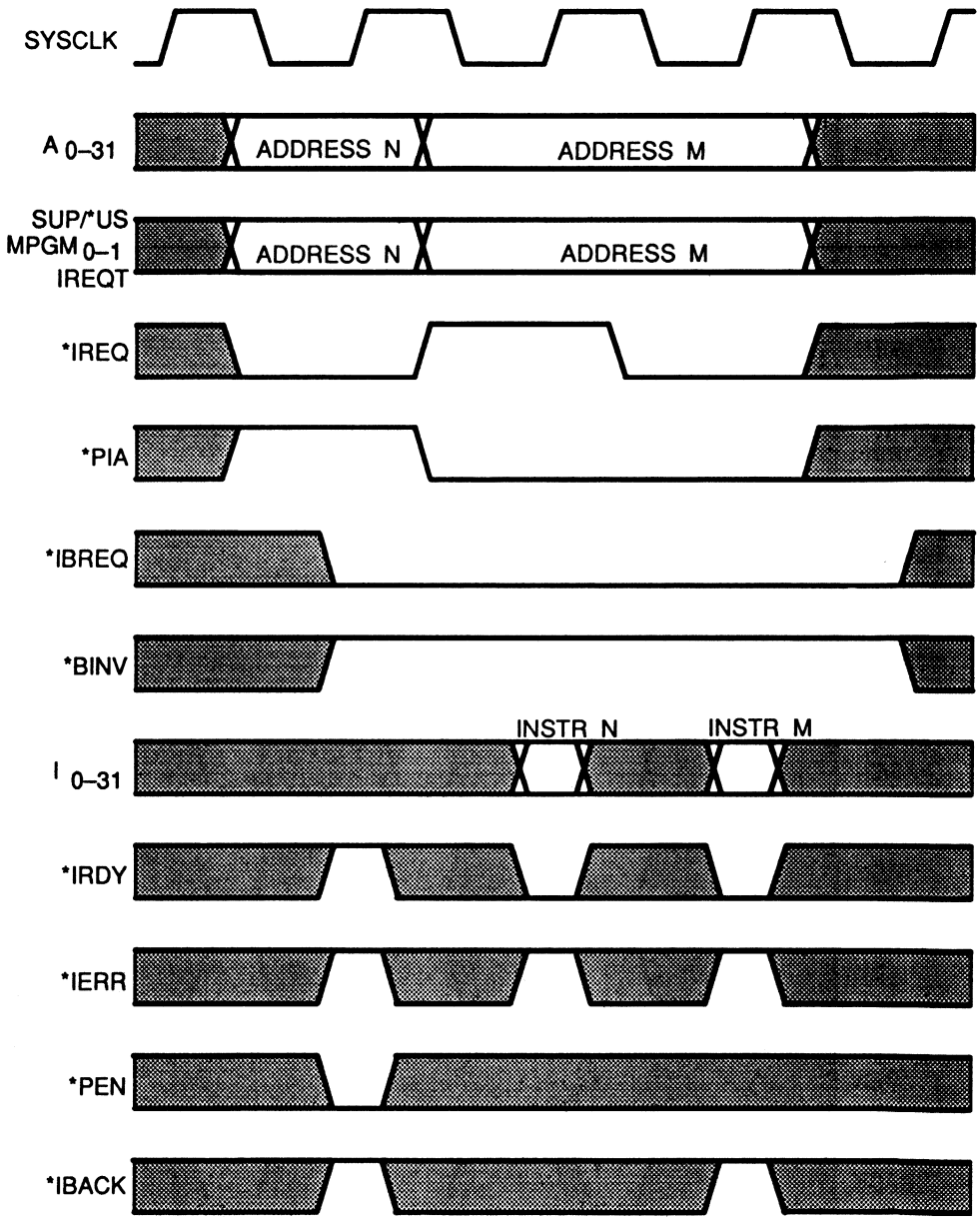


Figure A-3. Instruction Read—Pipelined Access

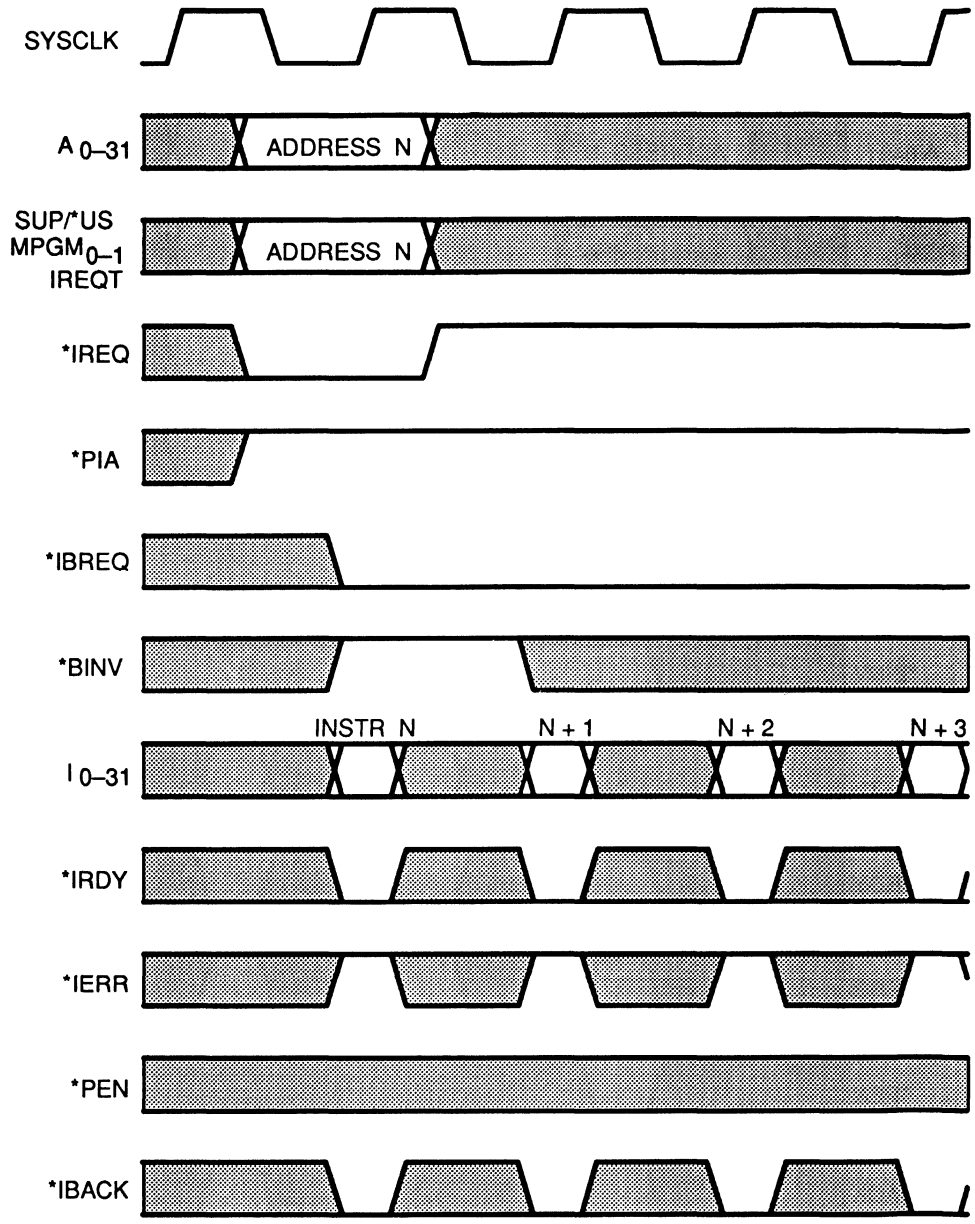


Figure A-4. Instruction Read—Establishing Burst-mode Access

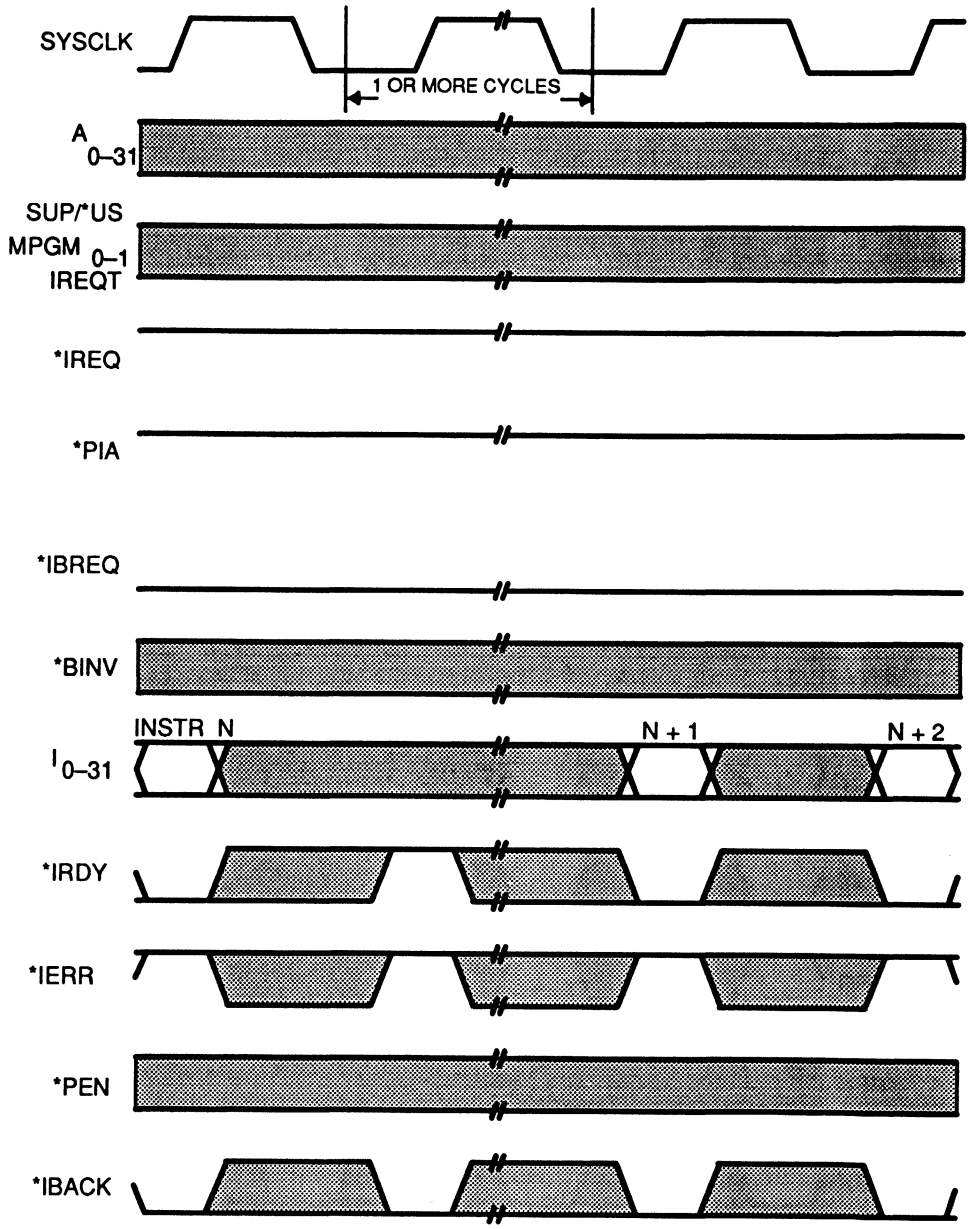


Figure A-5. Instruction Read—Burst-mode Access Suspended By Slave

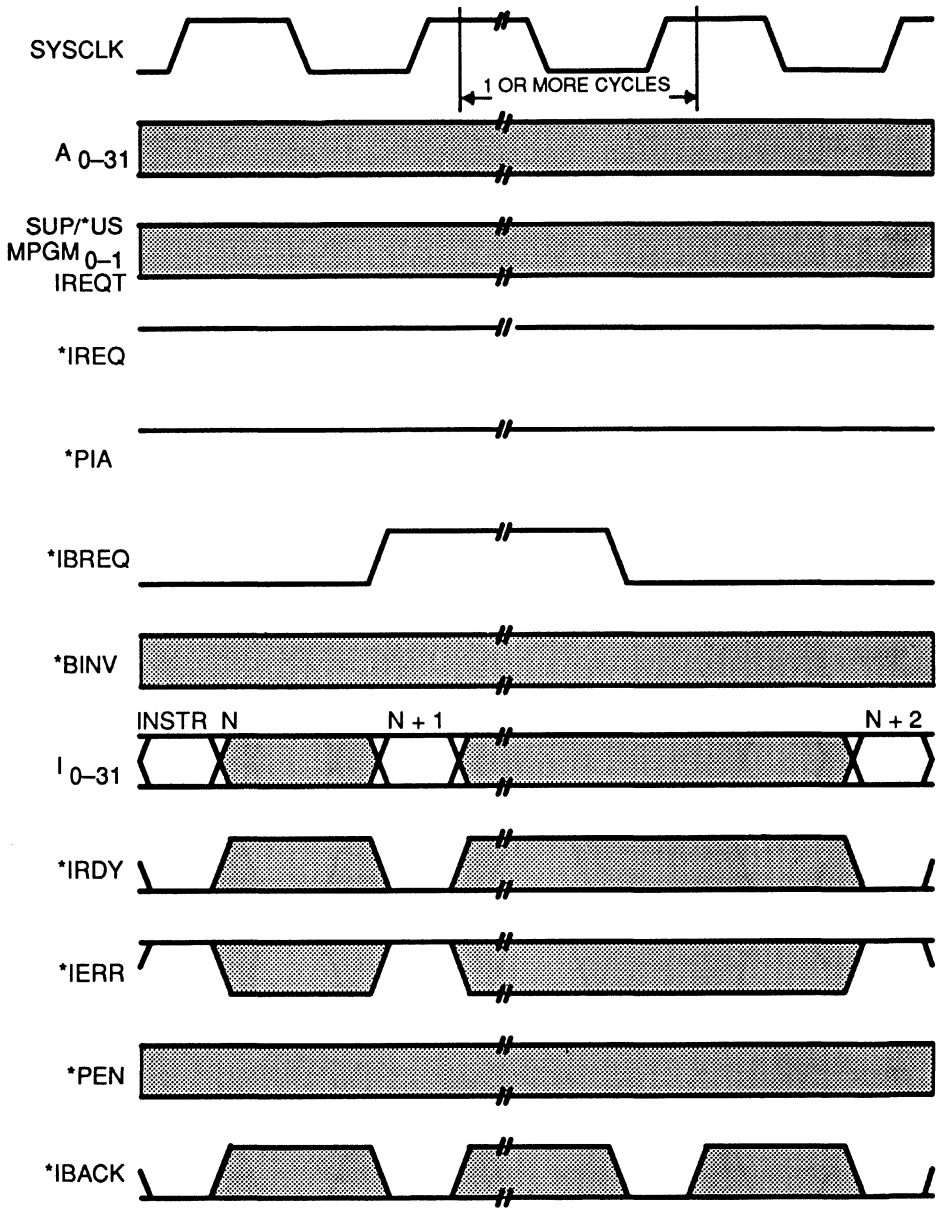


Figure A-6. Instruction Read—Burst-mode Access Suspended By Master

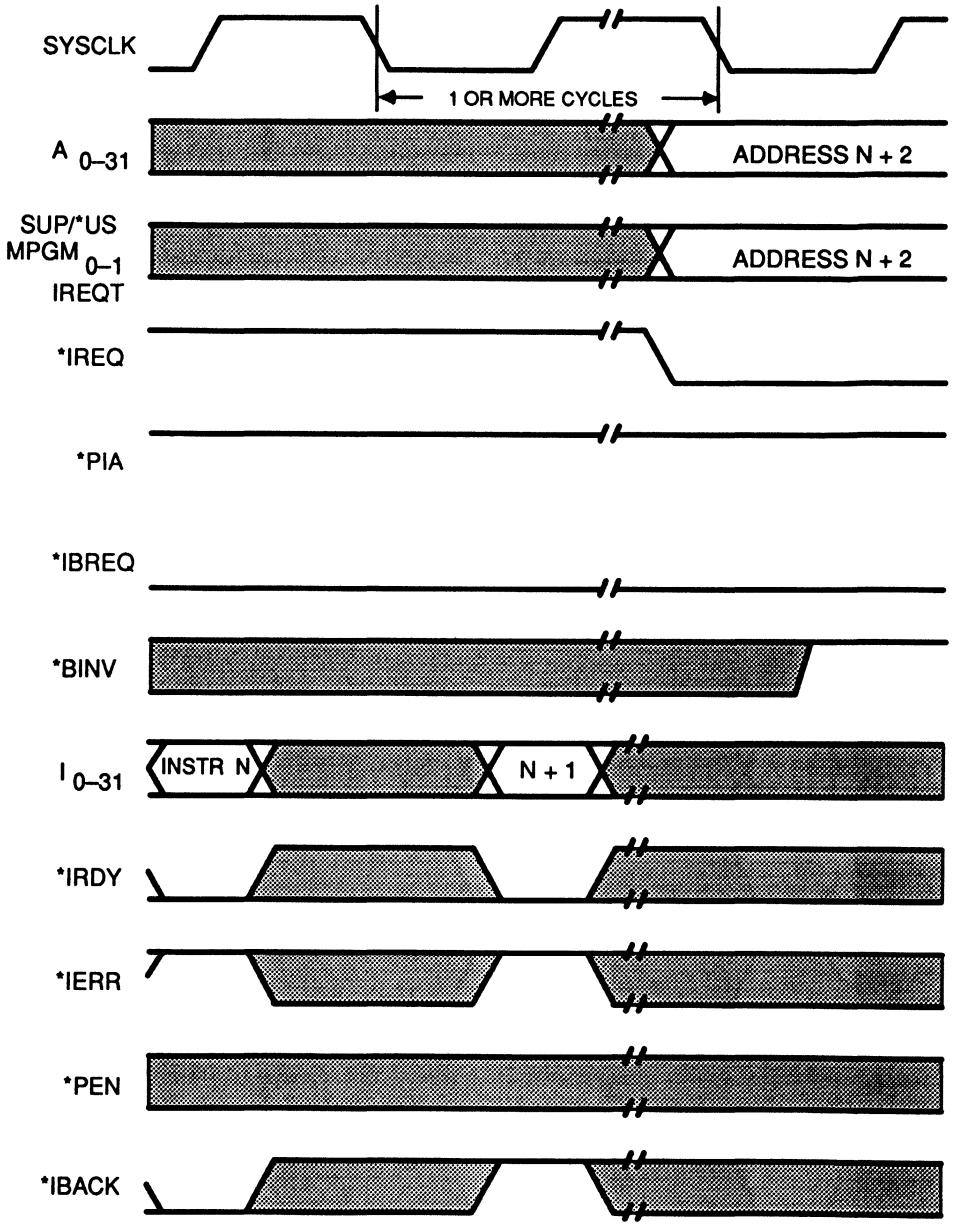


Figure A-7. Instruction Read—Burst-mode Access Preempted By Slave

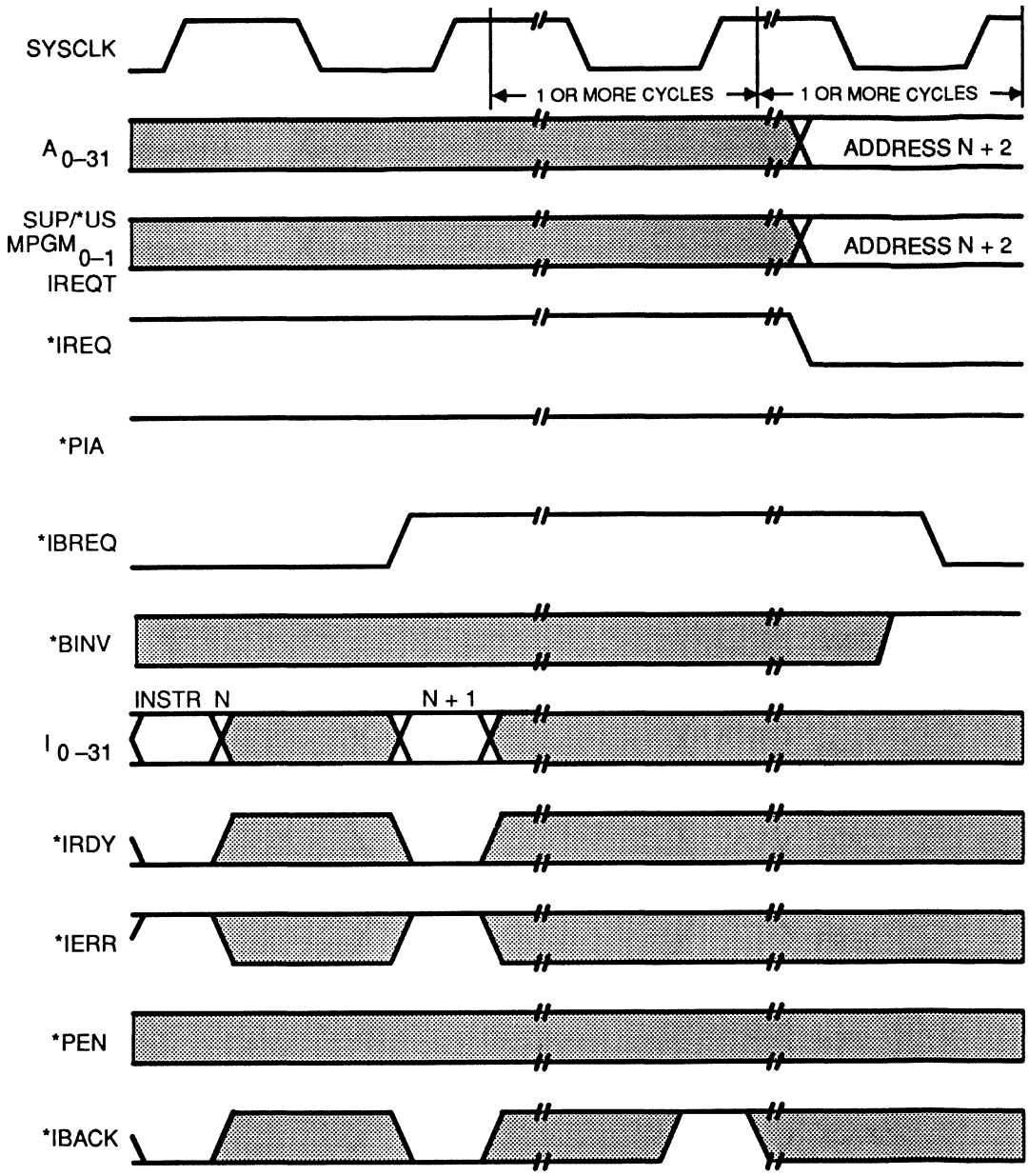


Figure A-8. Instruction Read—Burst-mode Access Suspended By Master And Later Preempted By Slave

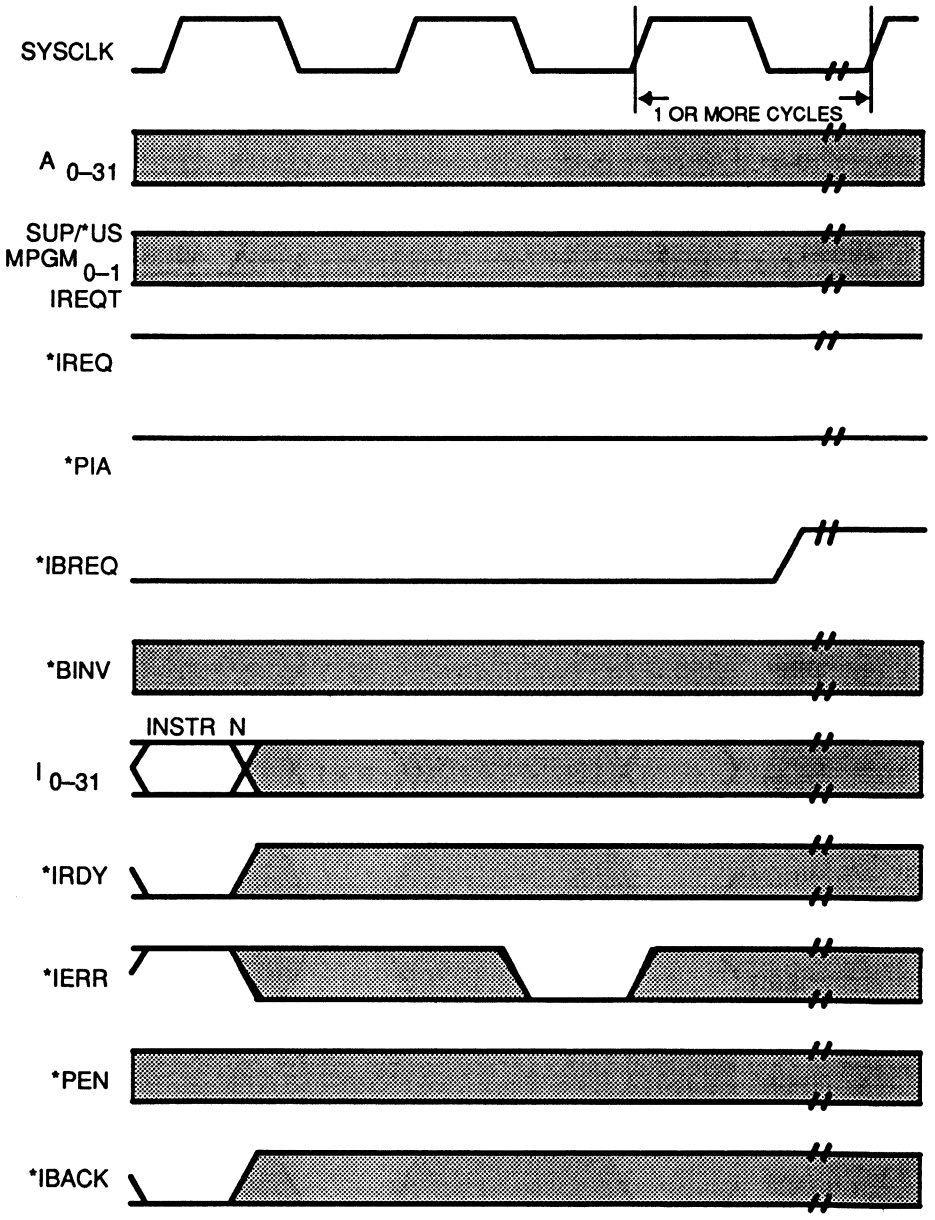


Figure A-9. Instruction Read—Burst-mode Access Cancelled By Slave

Note: This results in a trap.

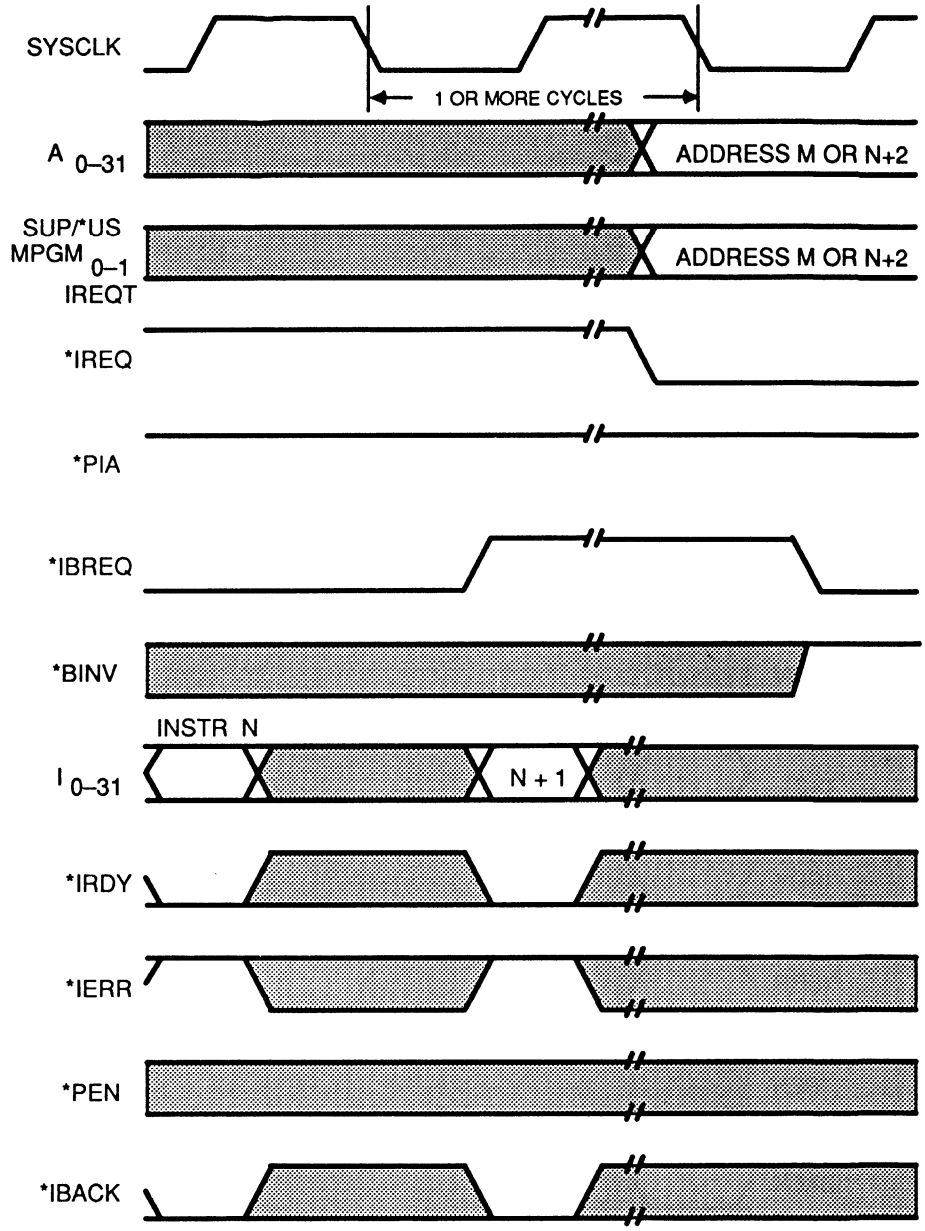


Figure A-10. Instruction Read—Burst-mode Access Ended By Master (Preempted, Terminated Or Cancelled)

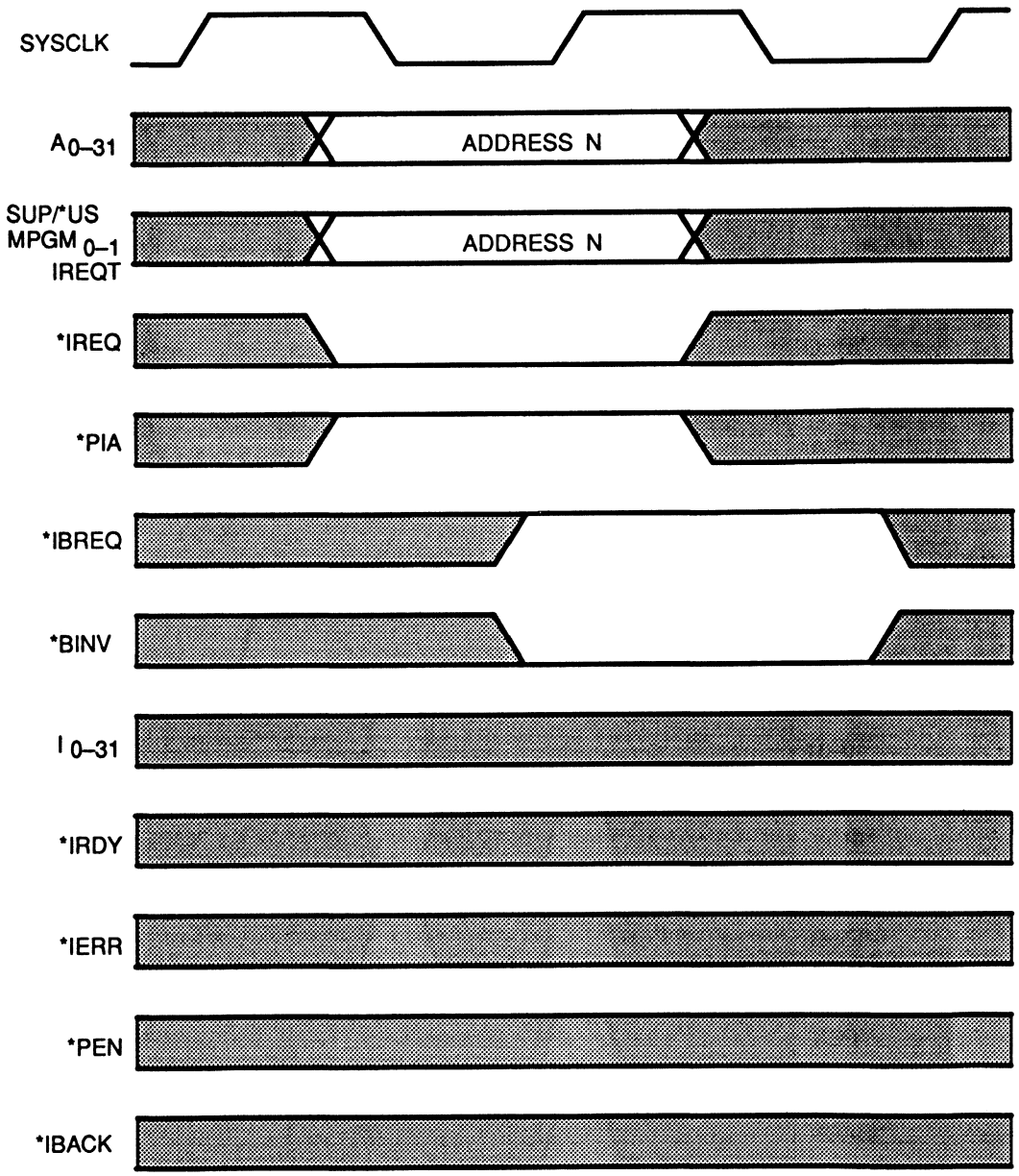


Figure A-11. Instruction Read—TLB Miss Or Protection Violation

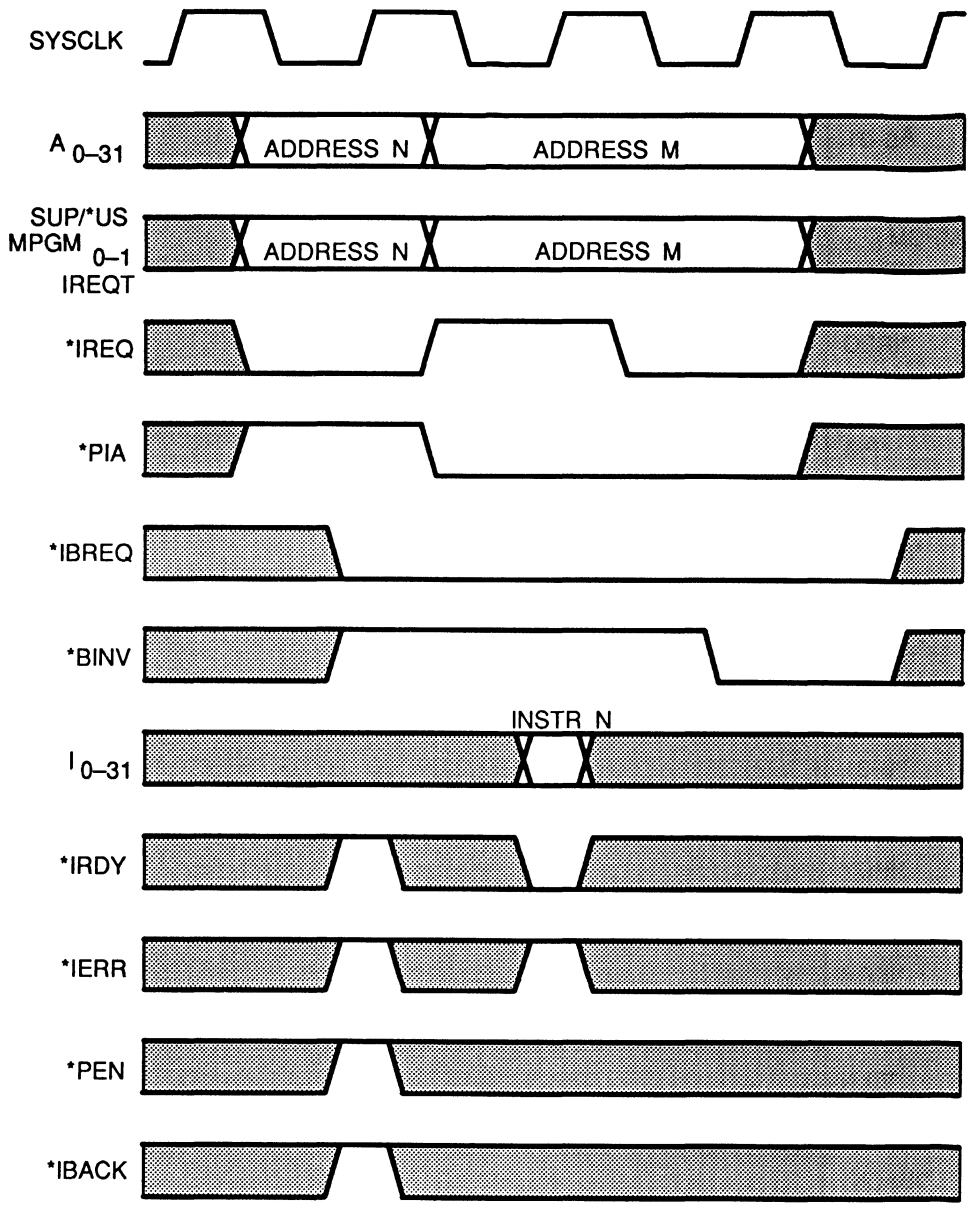


Figure A-12. Instruction Read—Pipelined Access With TLB Miss Or Protection Violation

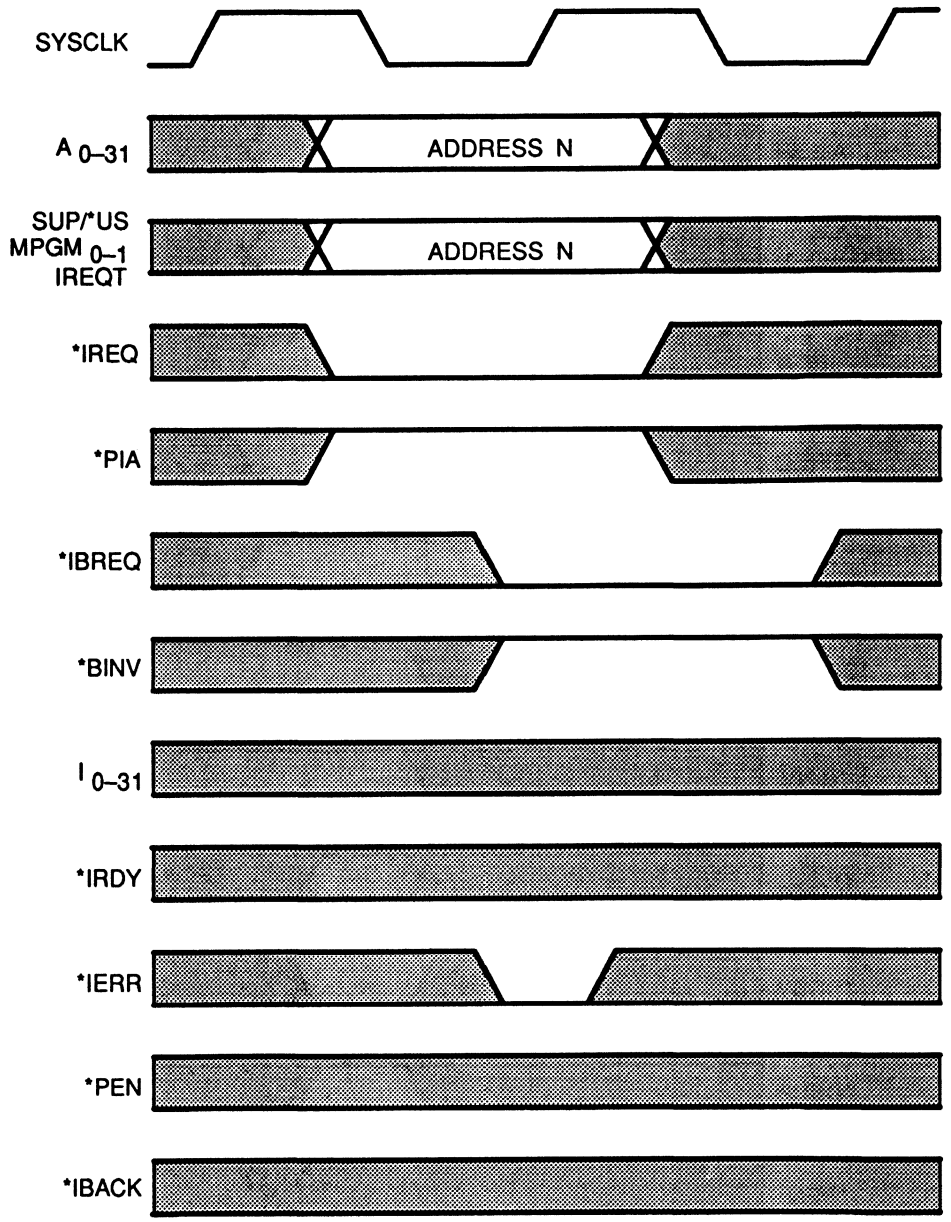


Figure A-13. Instruction Read—Error Detected By Slave

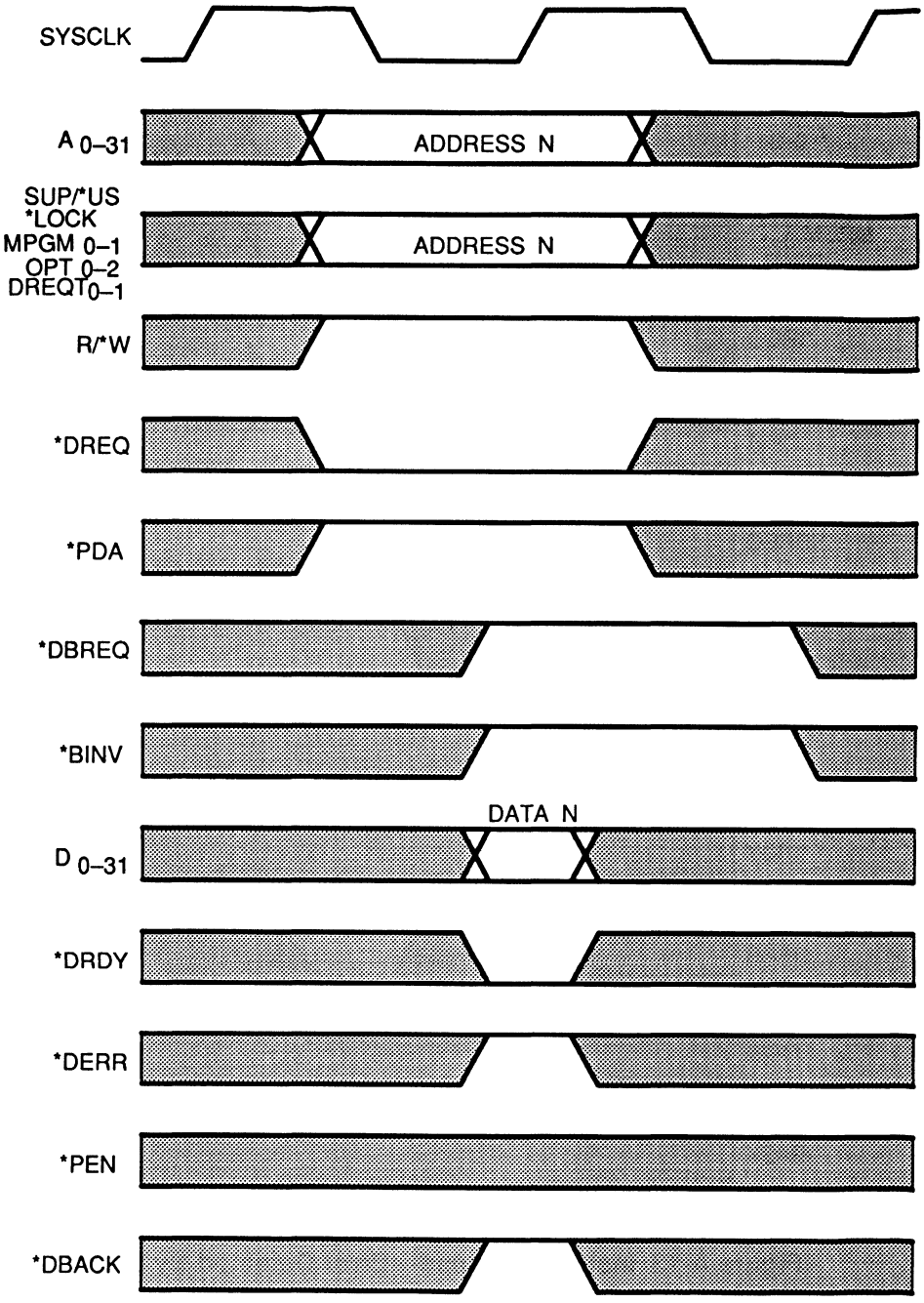


Figure A-14. Data Read—Simple Access

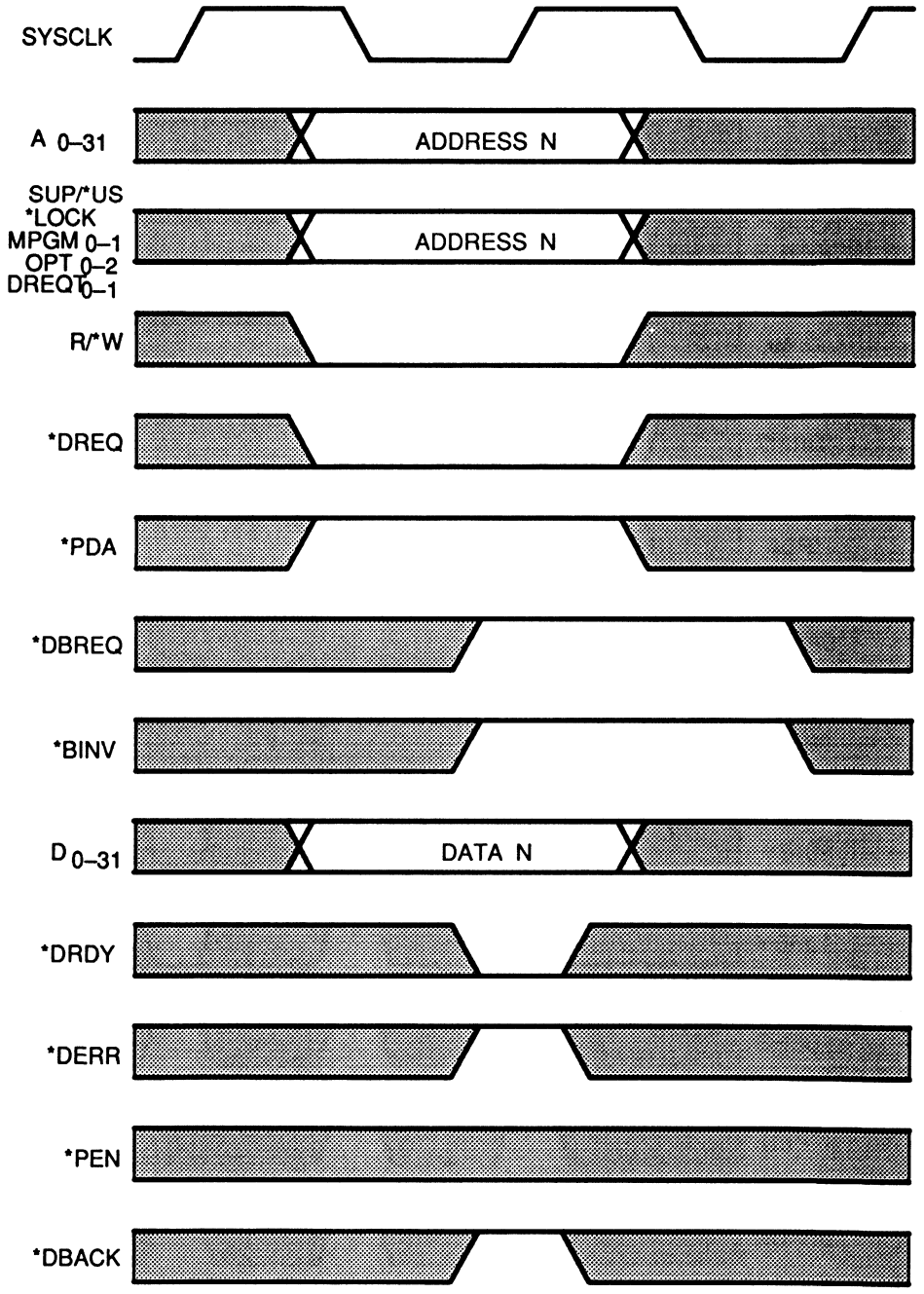


Figure A-15. Data Write—Simple Access

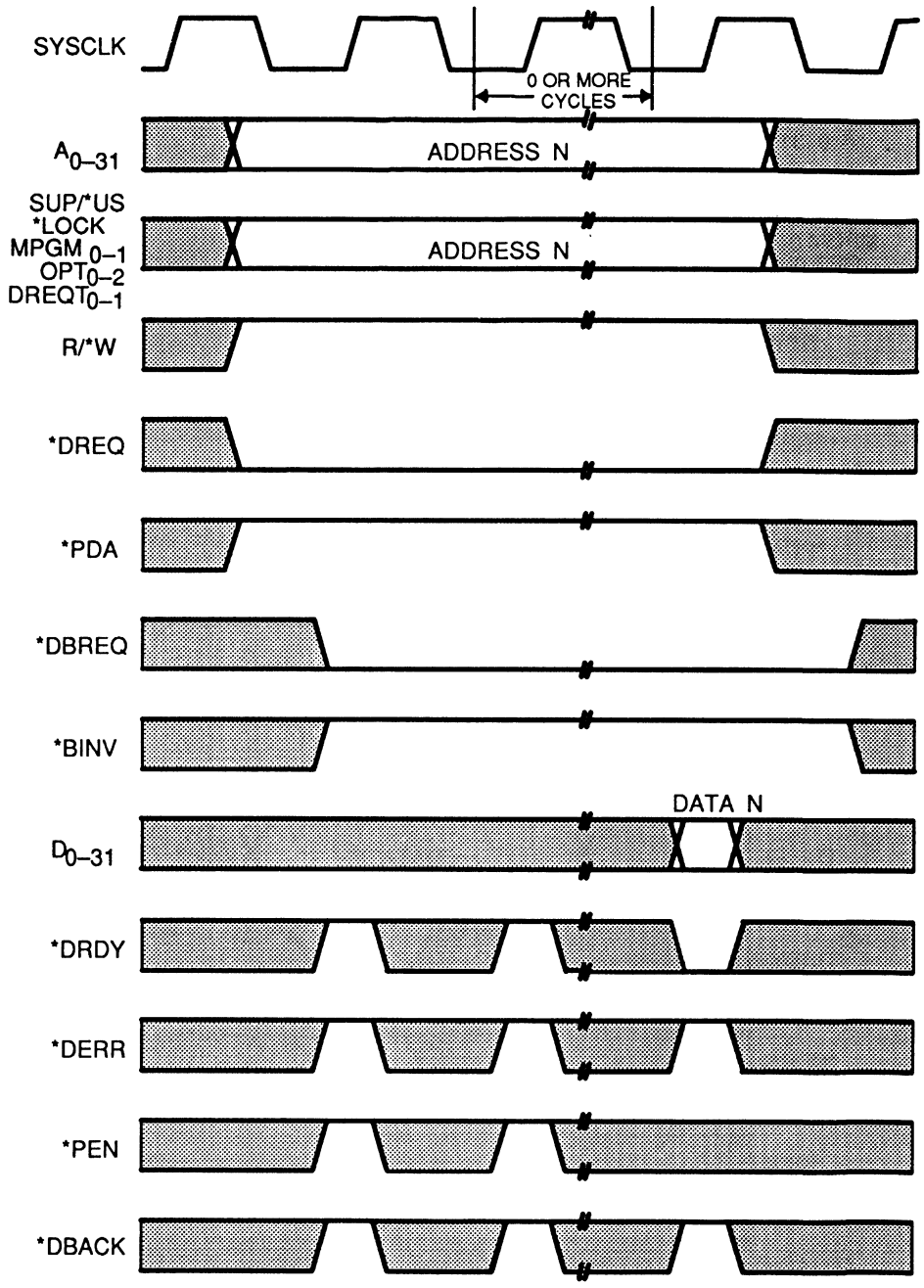


Figure A-16. Data Read—Simple Access With *DRDY Delayed

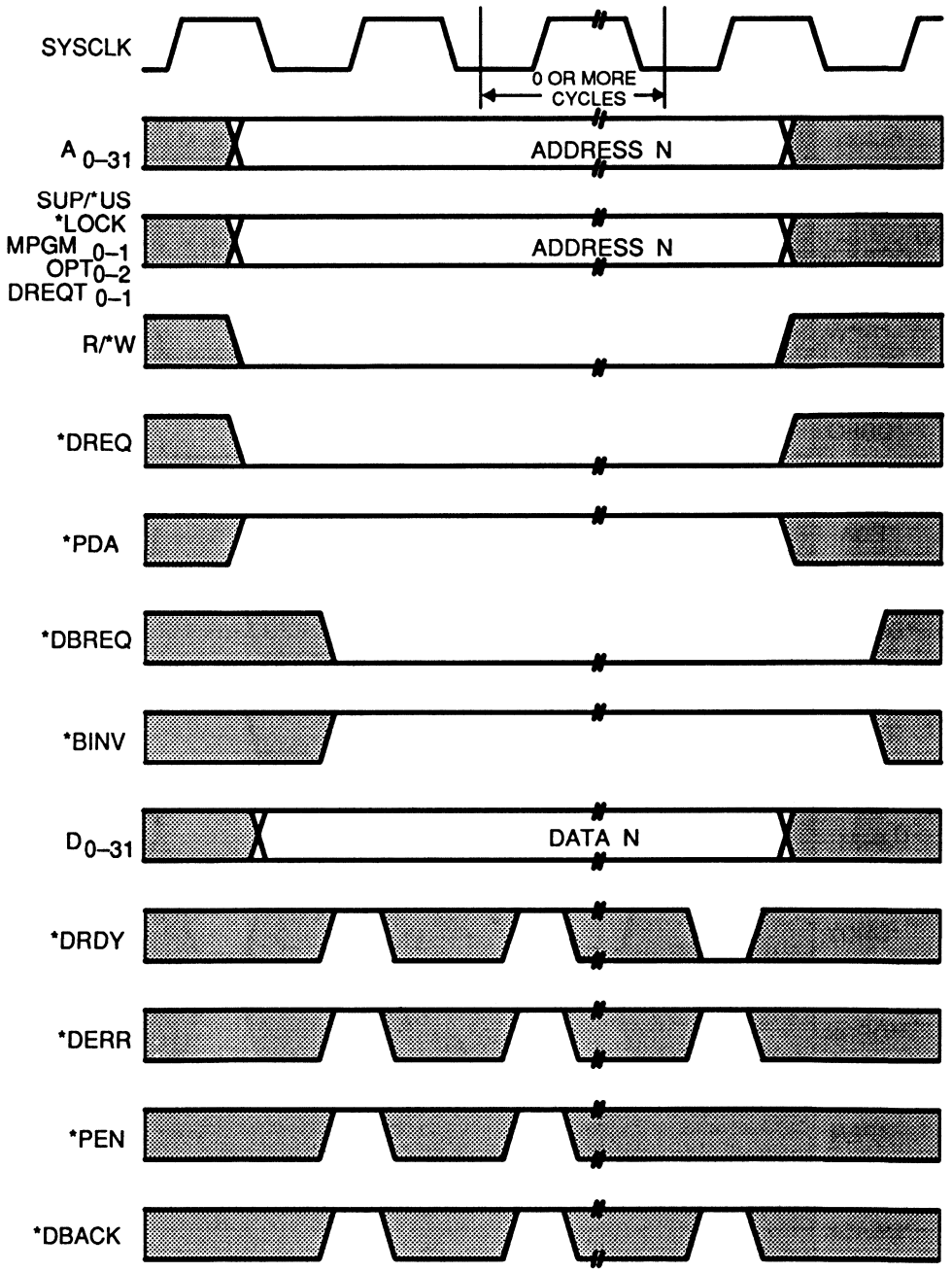


Figure A-17. Data Write—Simple Access With *DRDY Delayed

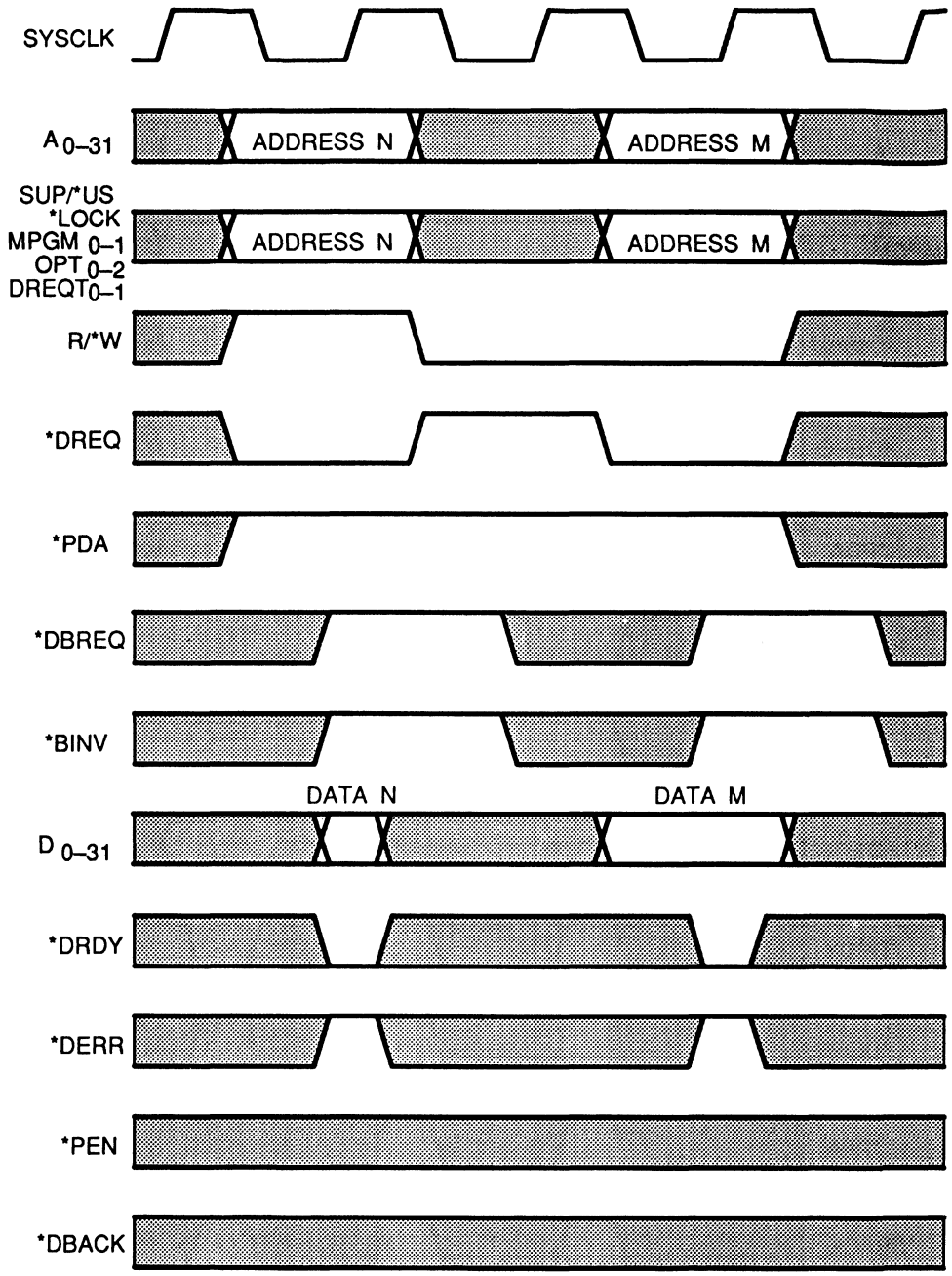


Figure A-18. Data Read Followed By Data Write—Simple Access

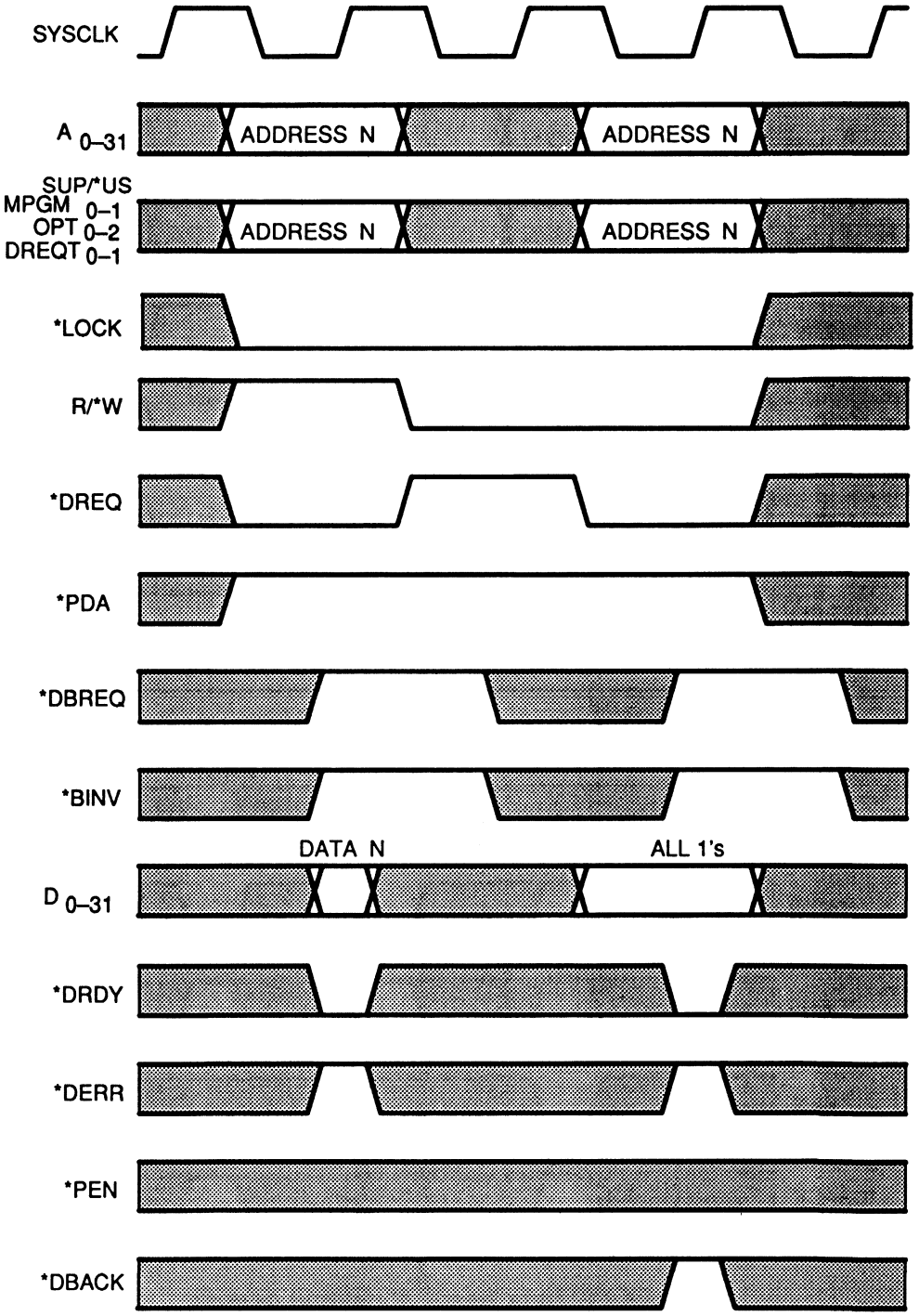


Figure A-19. Load And Set Instruction

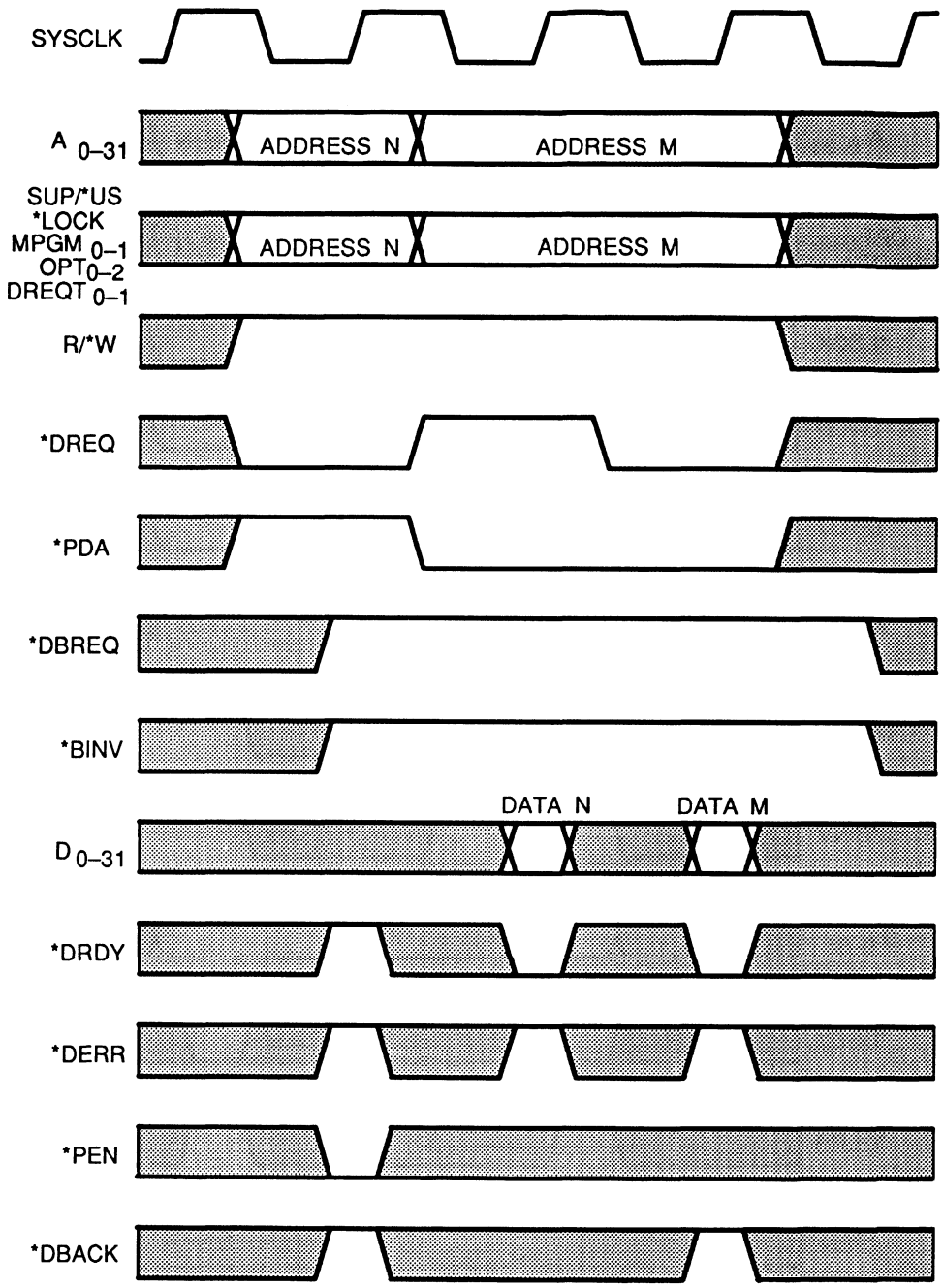


Figure A-20. Data Read—Pipelined Access

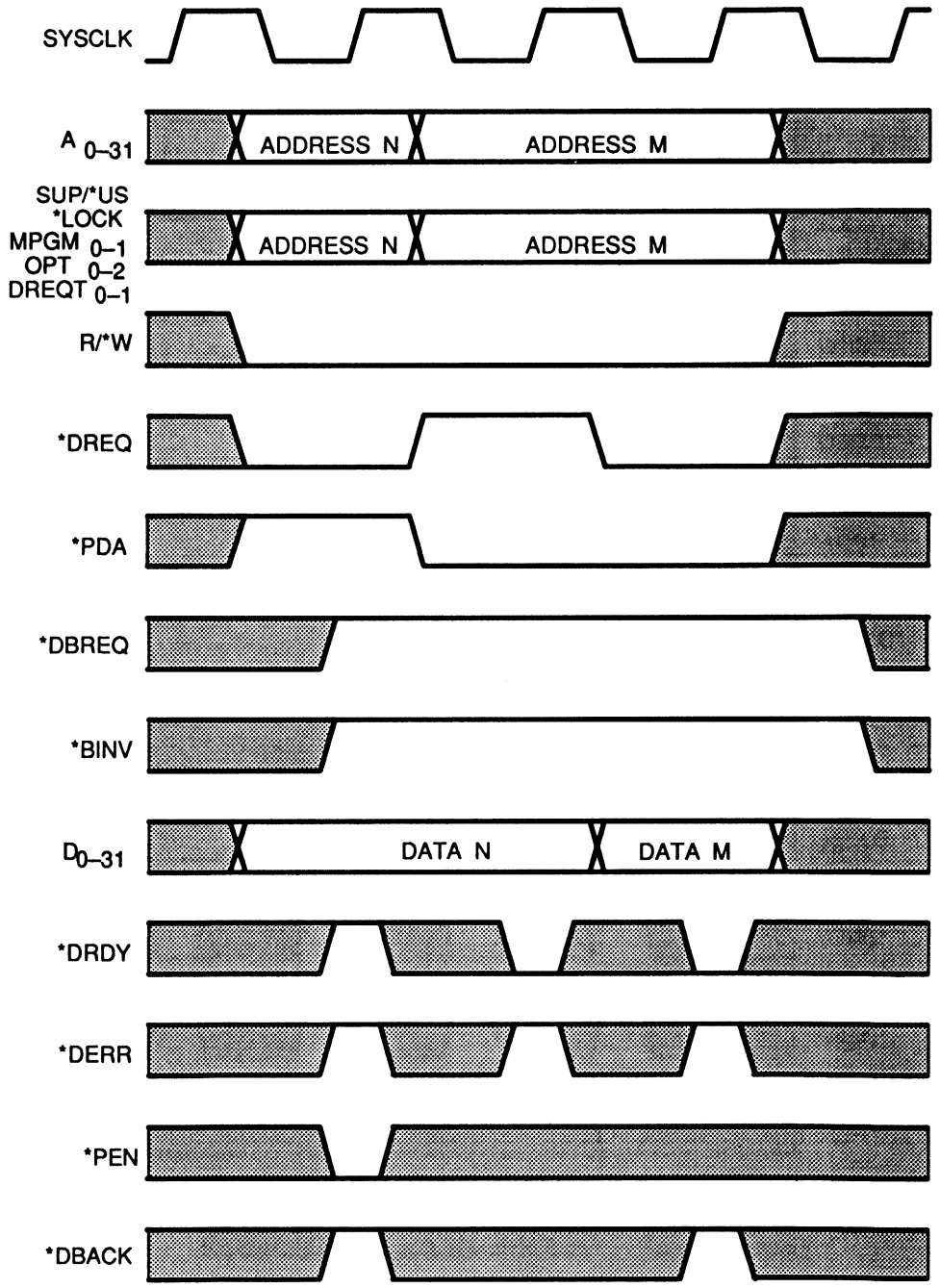


Figure A-21. Data Write—Pipelined Access

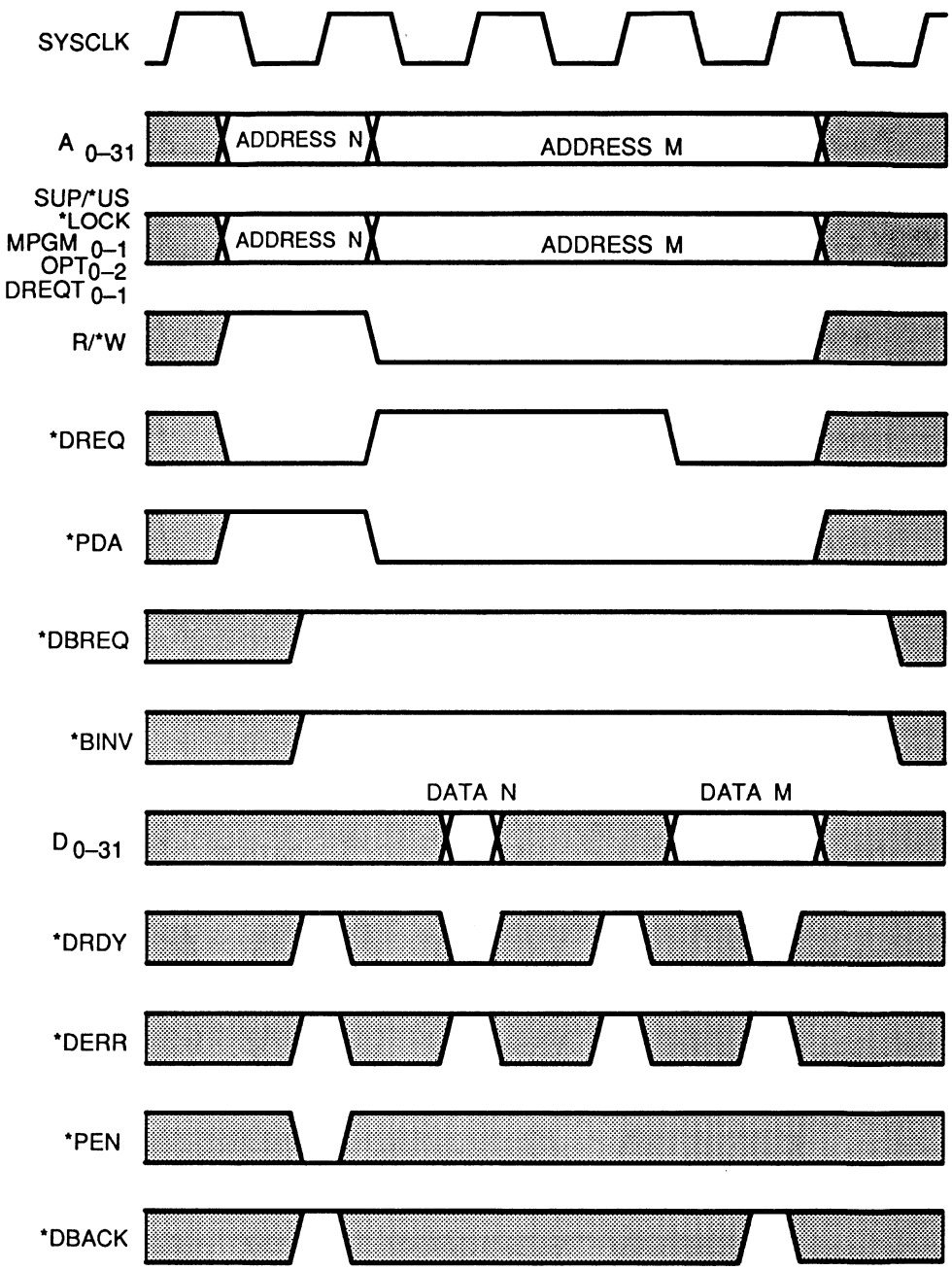


Figure A-22. Data Read Followed By Data Write—Pipelined Access (Not Used By Processor)

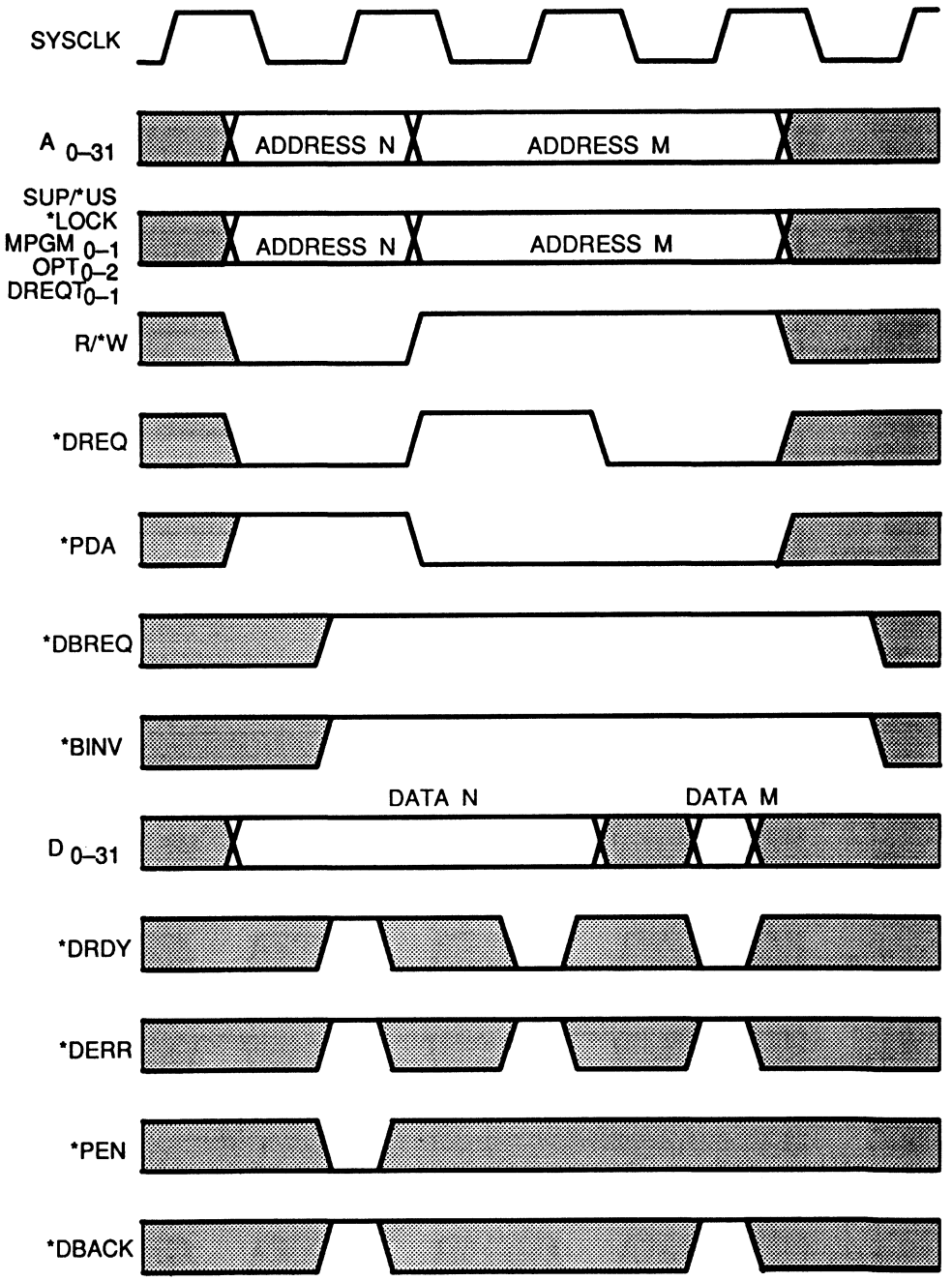


Figure A-23. Data Write Followed By Data Read—Pipelined Access

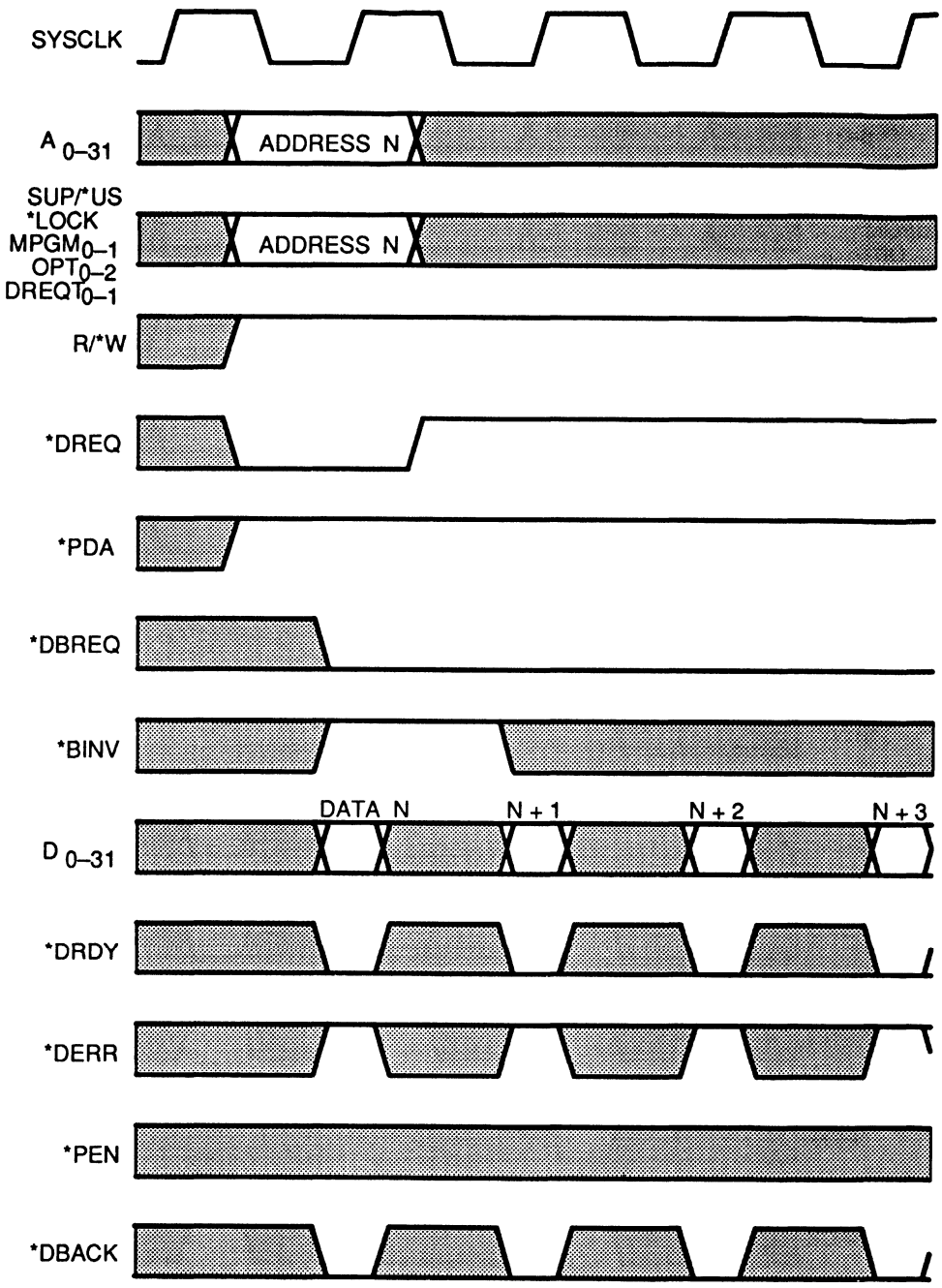


Figure A-24. Data Read—Establishing Burst-mode Access

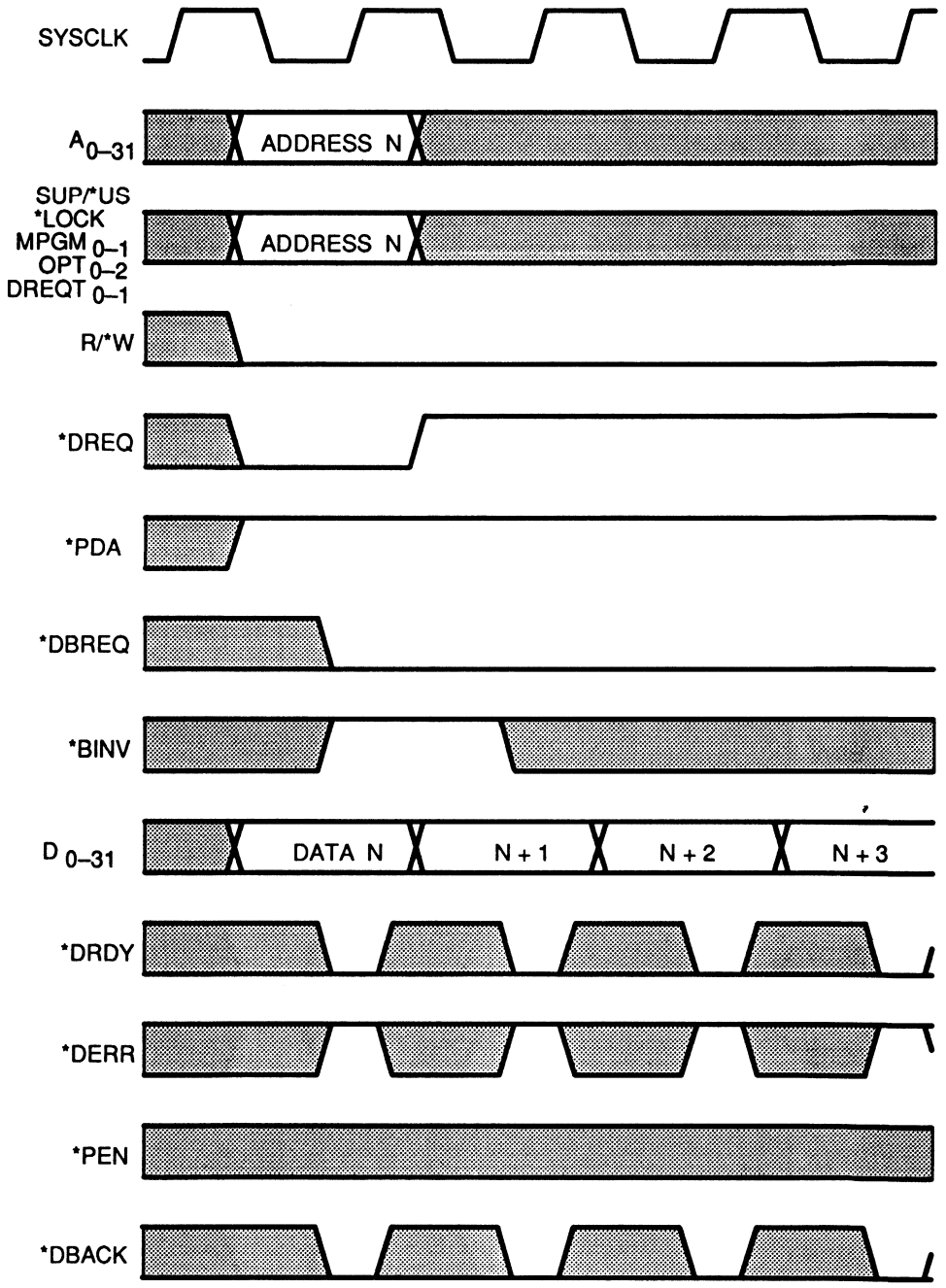


Figure A-25. Data Write—Establishing Burst-mode Access

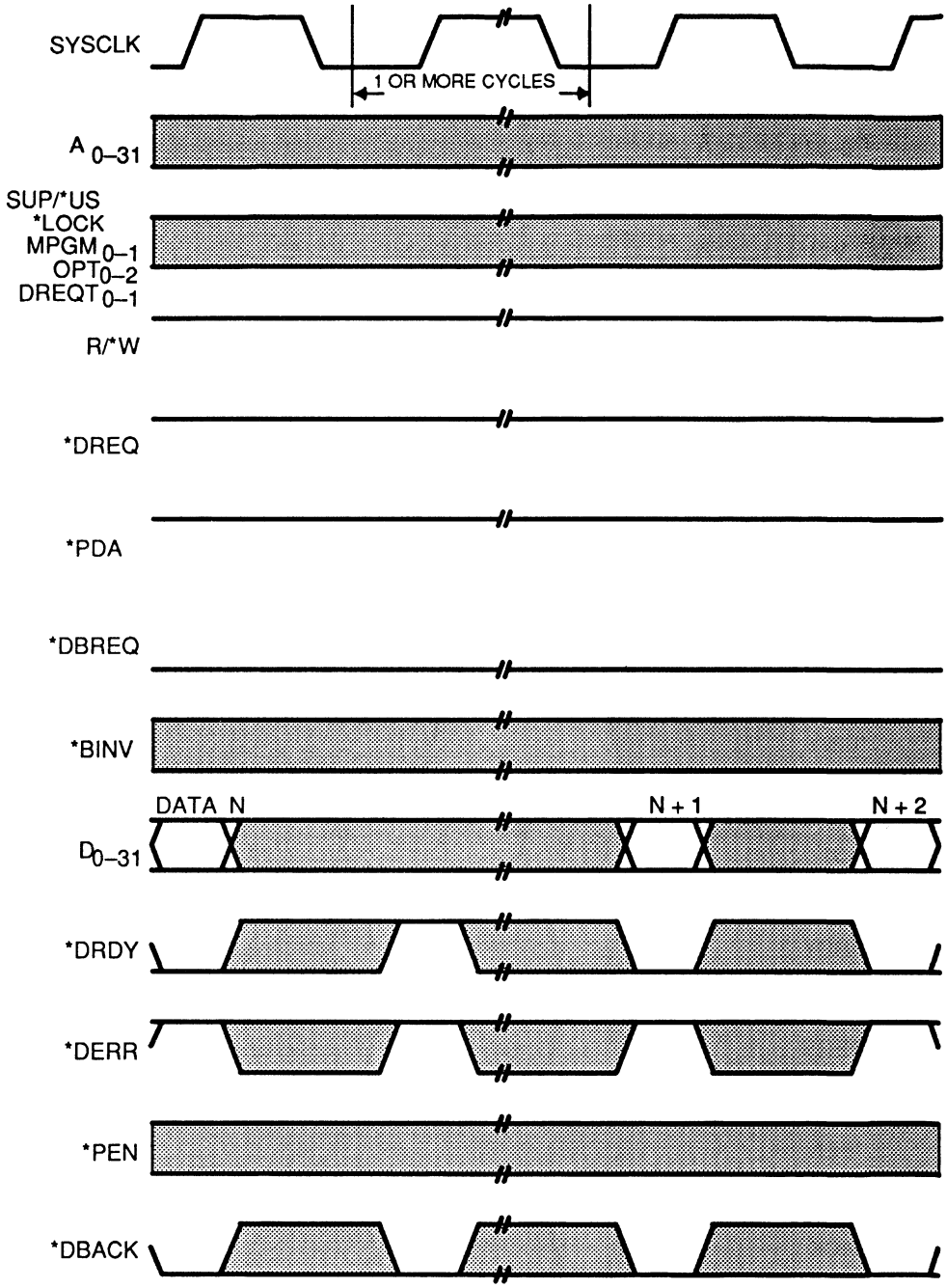


Figure A-26. Data Read—Burst-mode Access Suspended By Slave

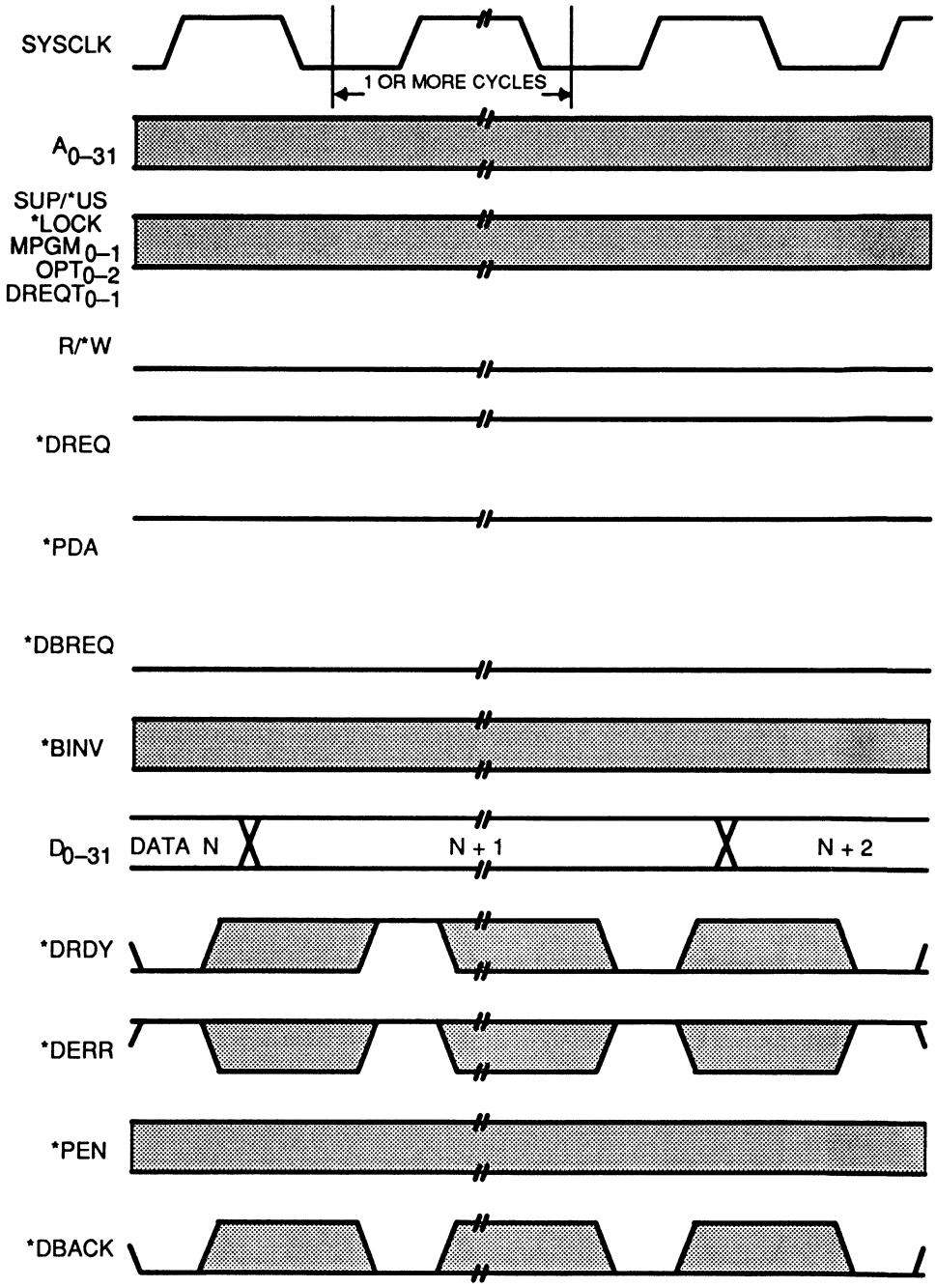


Figure A-27. Data Write—Burst-mode Access Suspended By Slave

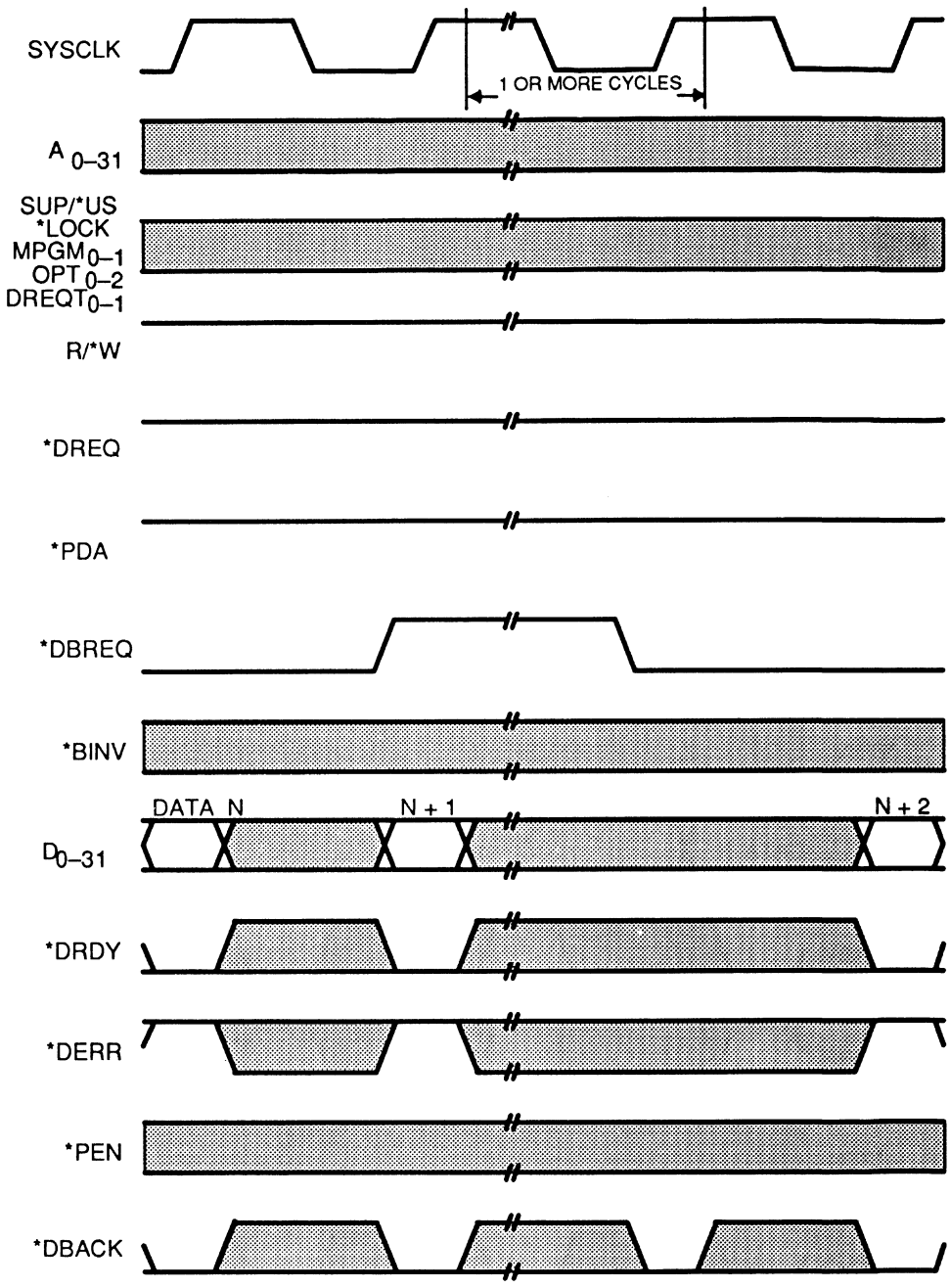


Figure A-28. Data Read—Burst-mode Access Suspended By Master (Not Used By Processor)

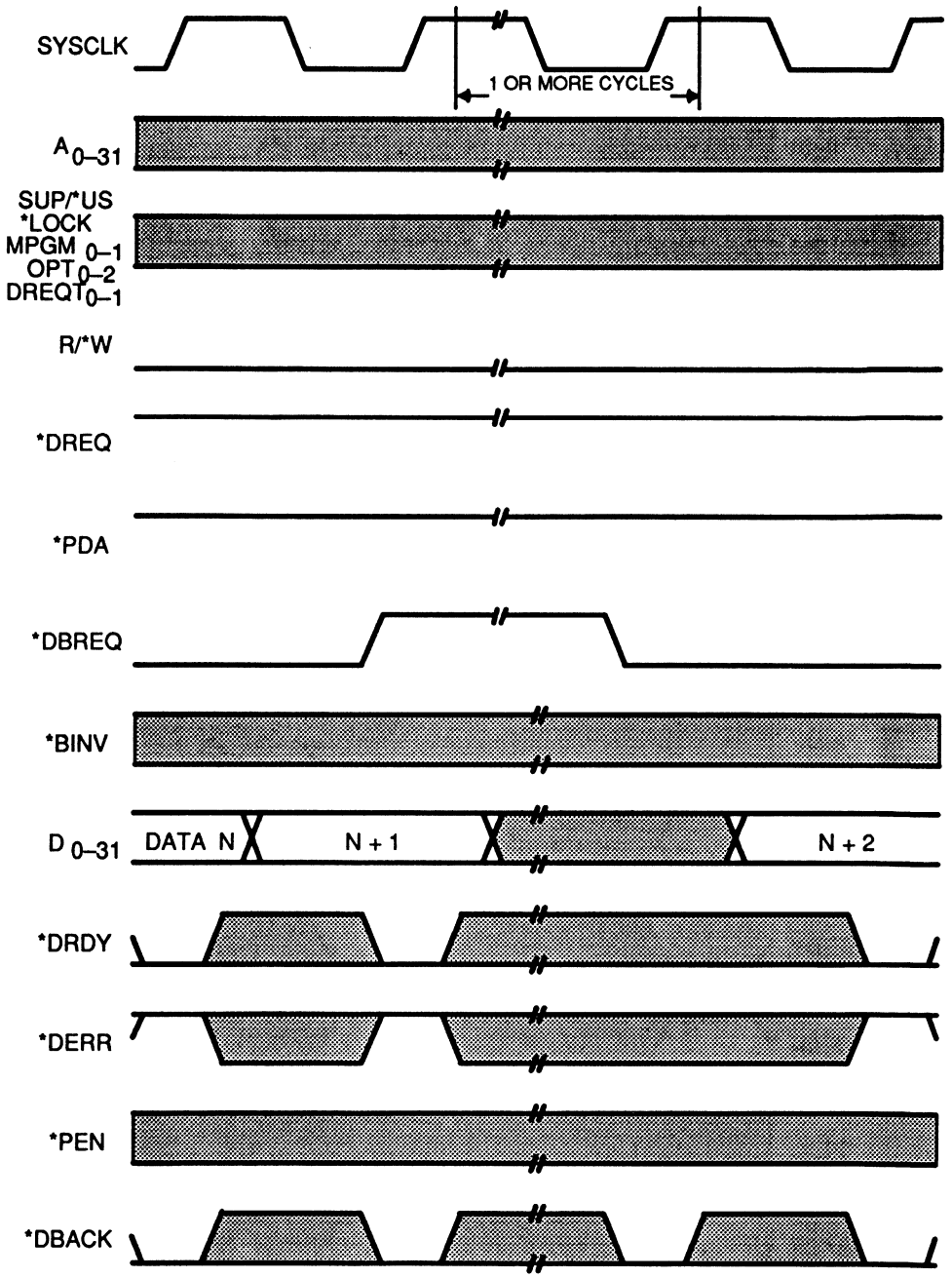


Figure A-29. Data Write—Burst-mode Access Suspended By Master (Not Used By Processor)

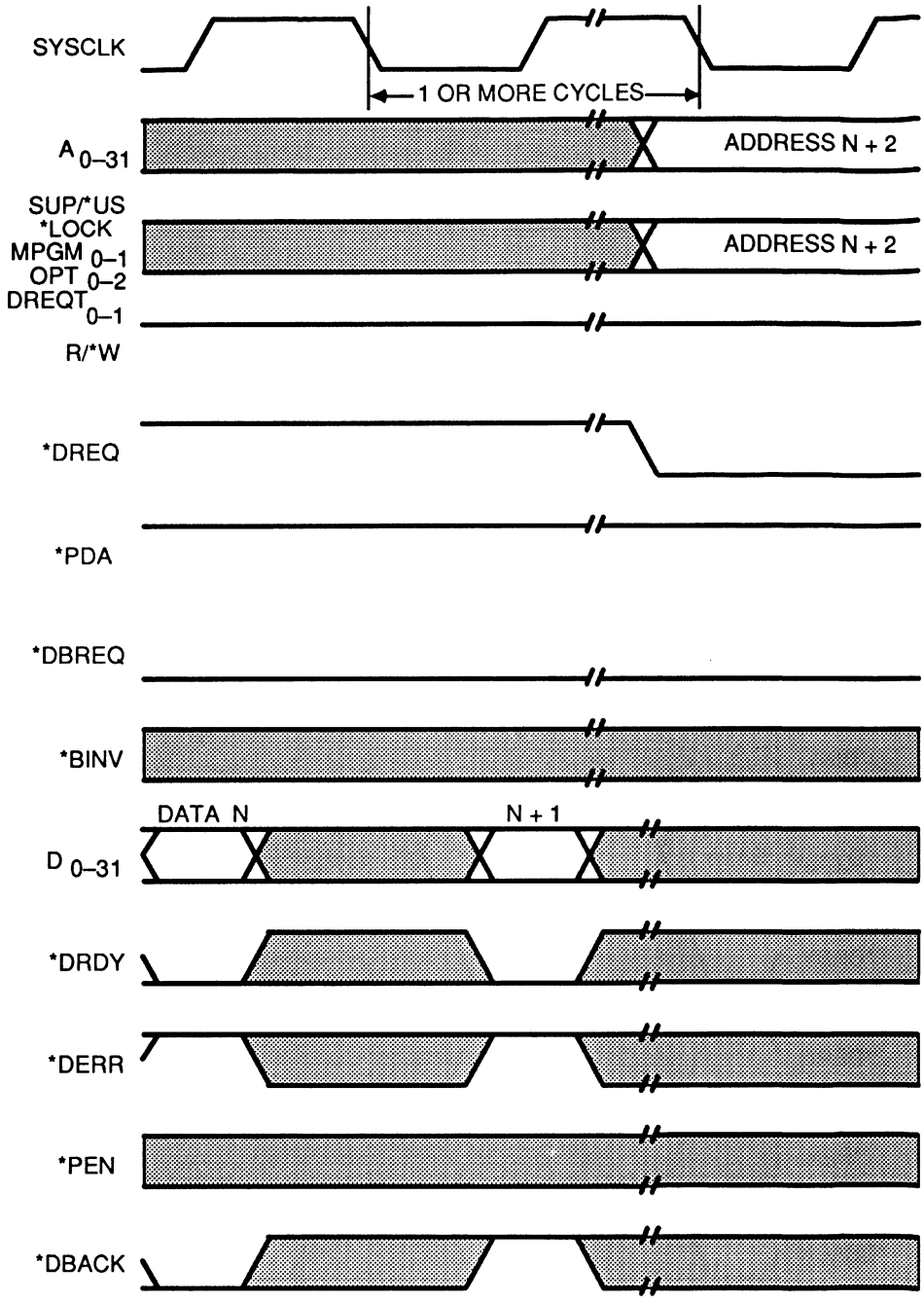


Figure A-30. Data Read—Burst-mode Access Preempted By Slave

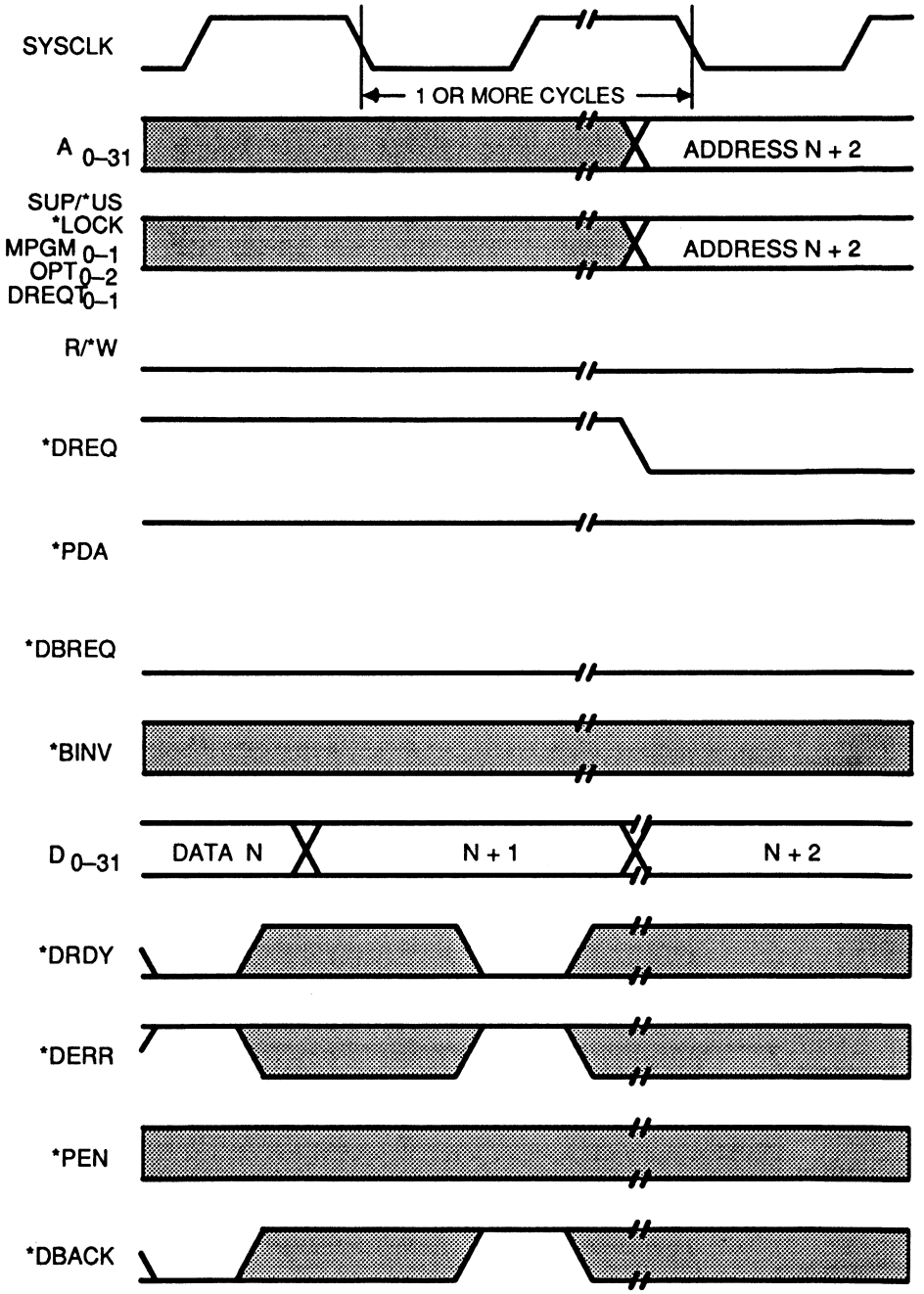


Figure A-31. Data Write—Burst-mode Access Preempted By Slave

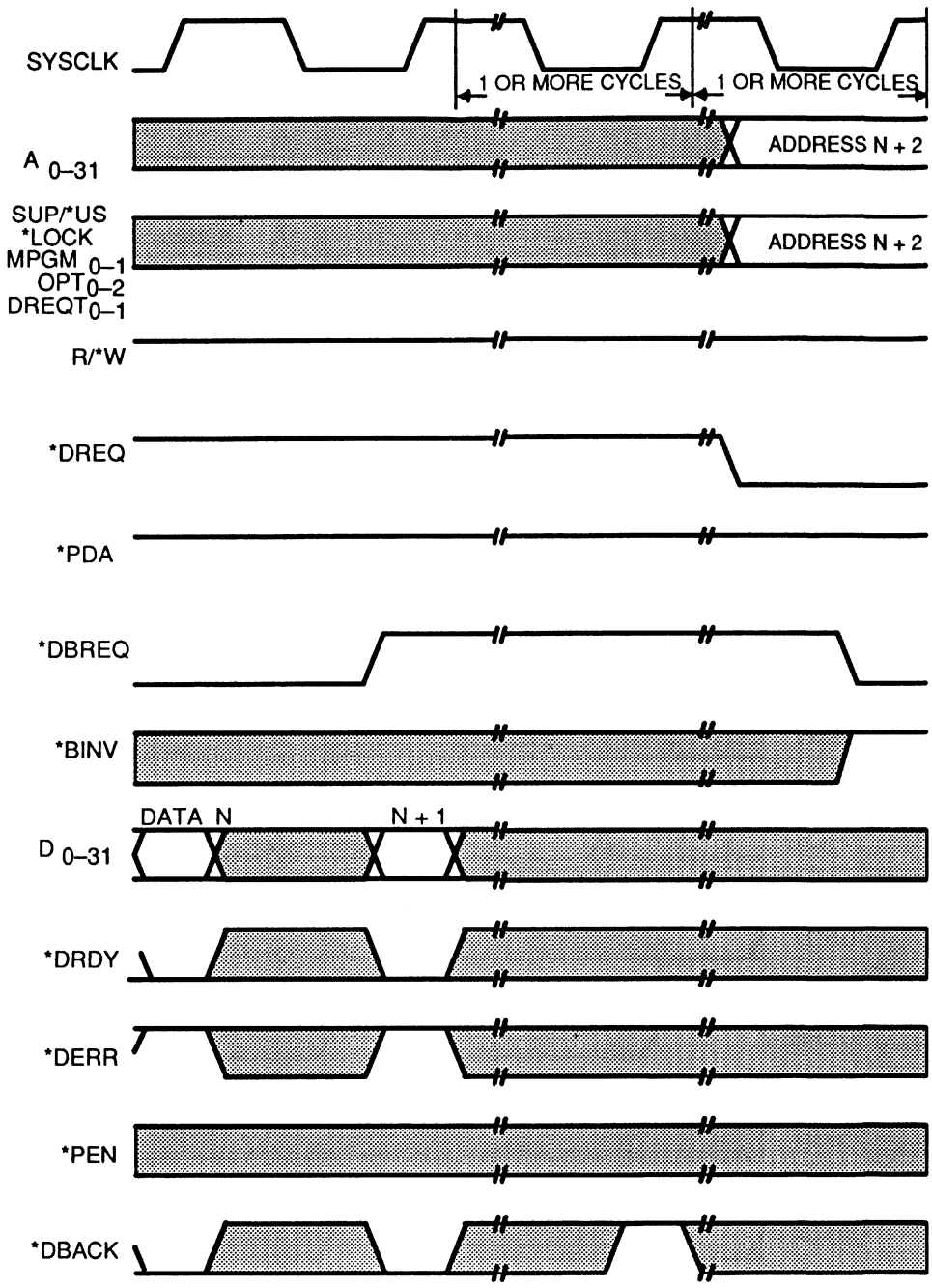


Figure A-32. Data Read—Burst-mode Access Suspended By Master And Later Preempted By Slave (Not Used By Processor)

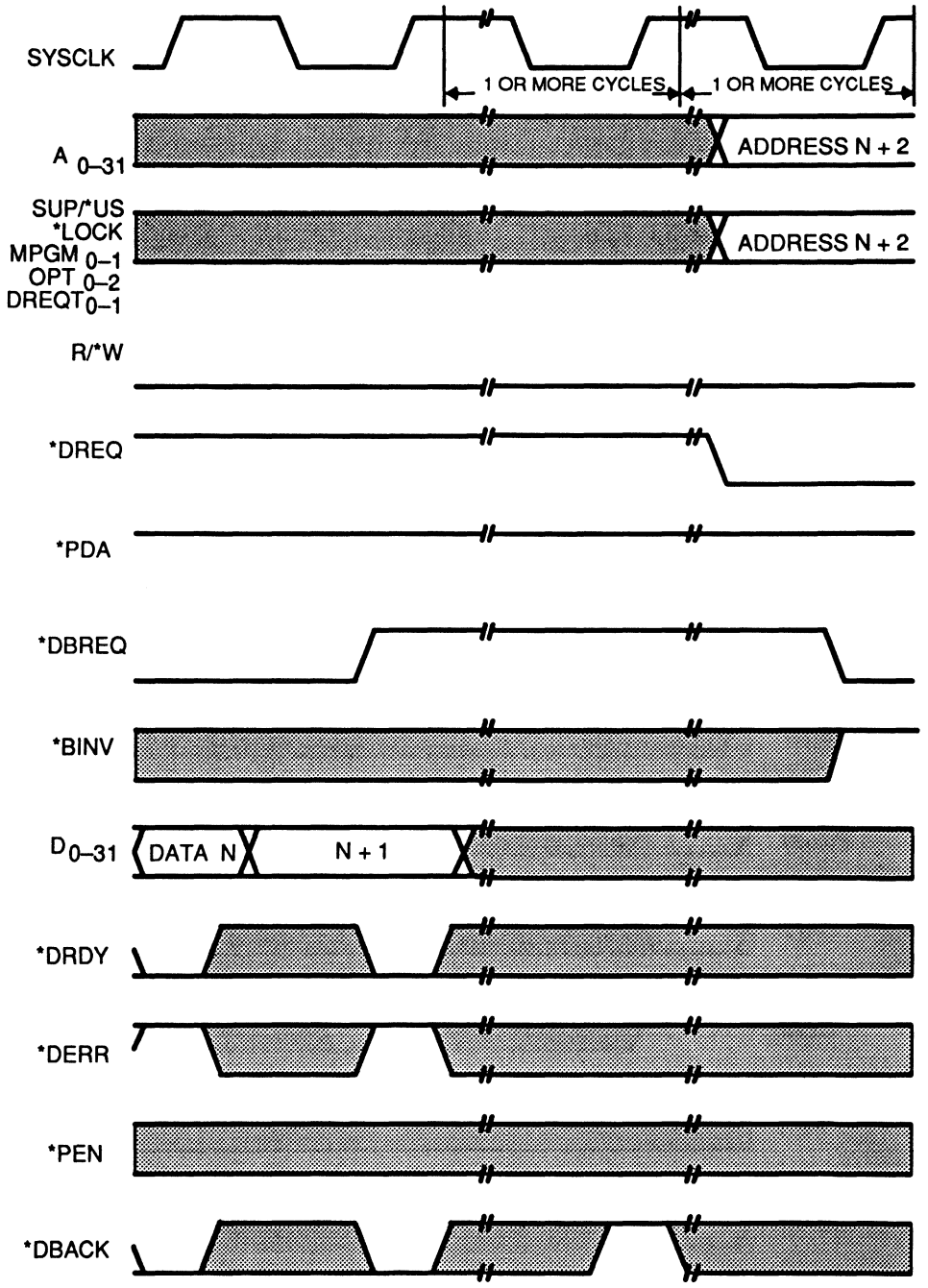


Figure A-33. Data Write—Burst-mode Access Suspended By Master And Later Preempted By Slave (Not Used By Processor)

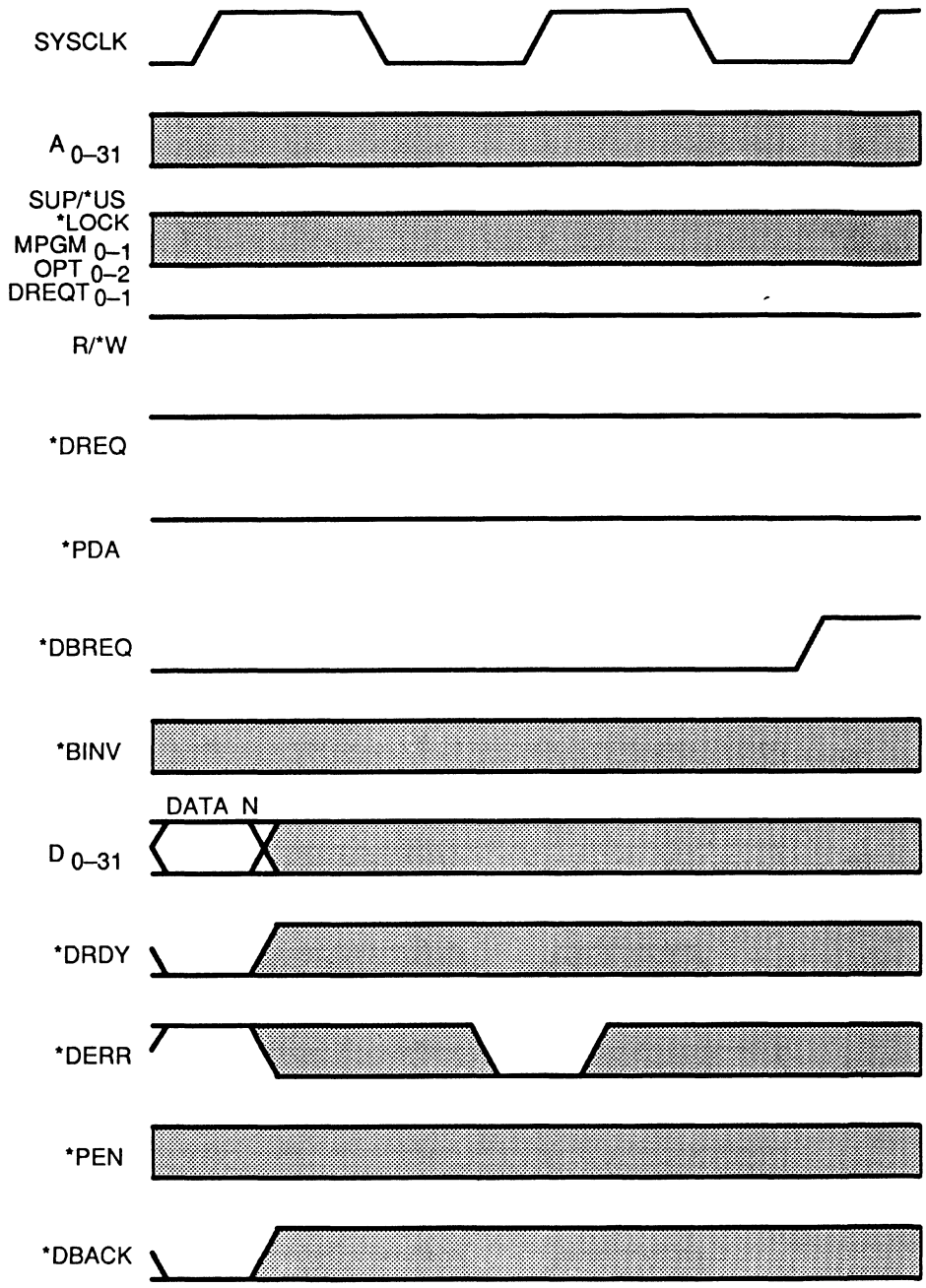


Figure A-34. Data Read—Burst-mode Access Cancelled By Slave

Note: This results in a trap.

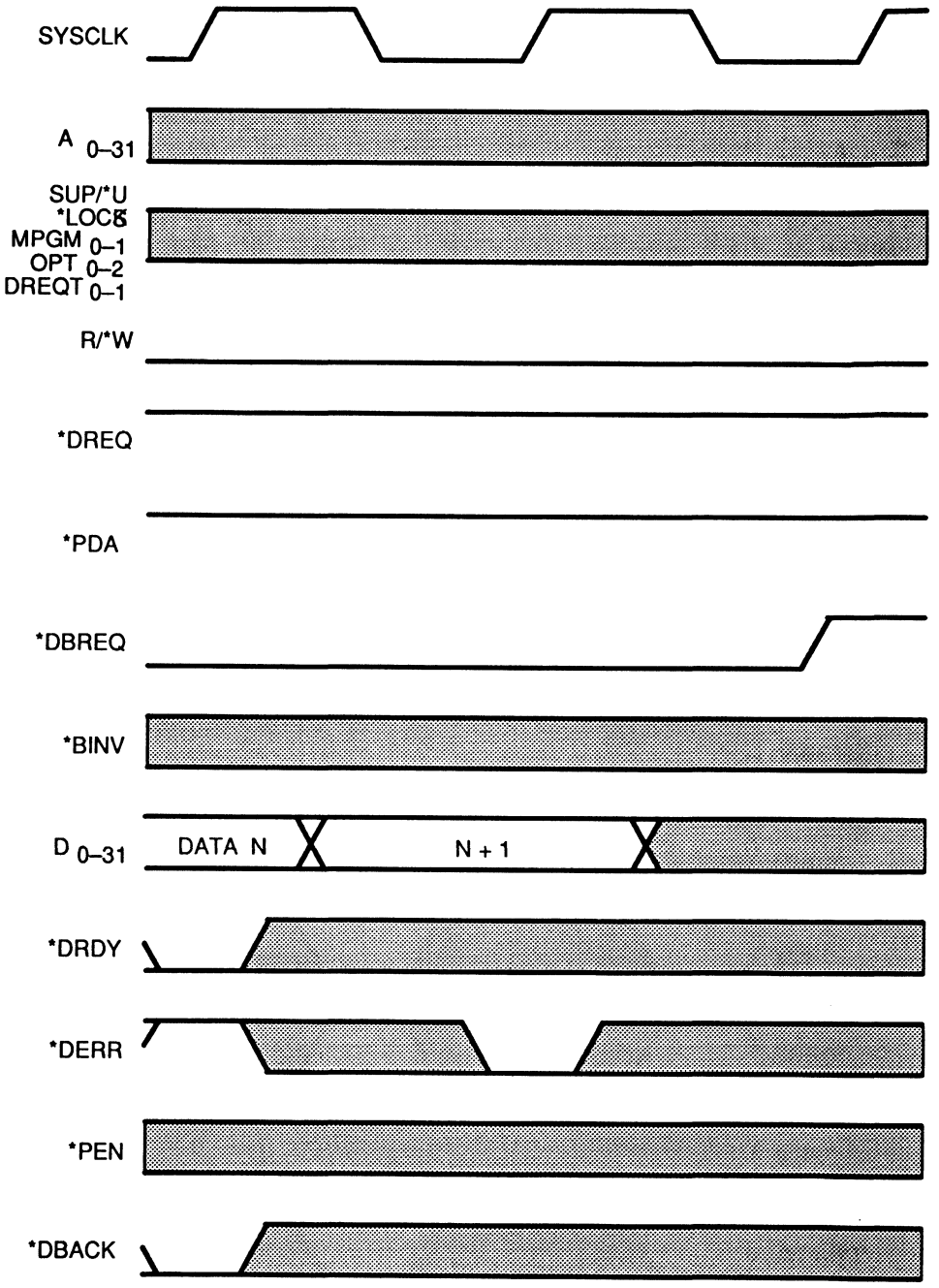


Figure A-35. Data Write—Burst-mode Access Cancelled By Slave

Note: This results in a trap.

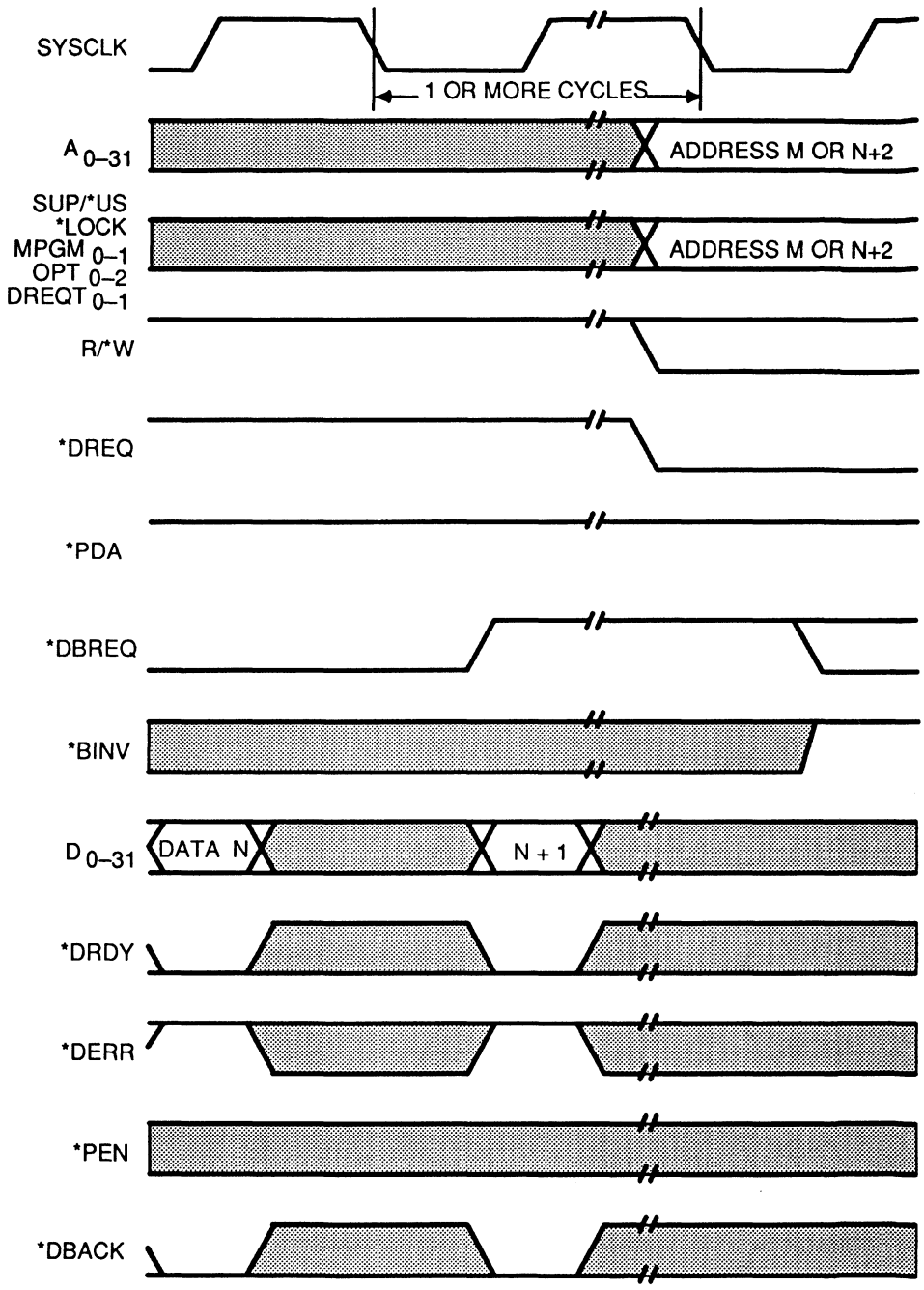


Figure A-36. Data Read—Burst-mode Access Ended By Master (Preempted, Terminated Or Cancelled)

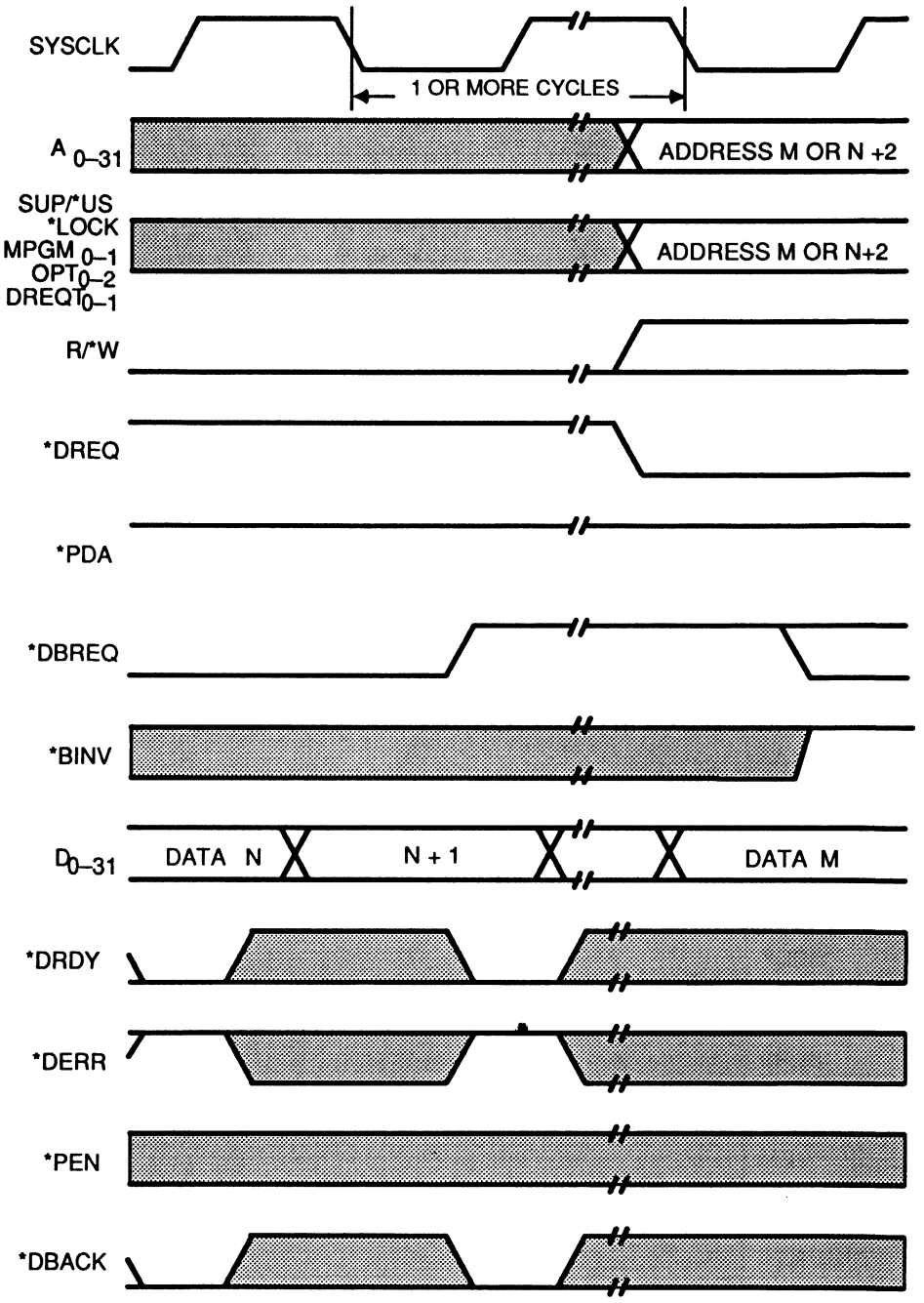


Figure A-37. Data Write—Burst-mode Access Ended By Master (Preempted, Terminated Or Cancelled)

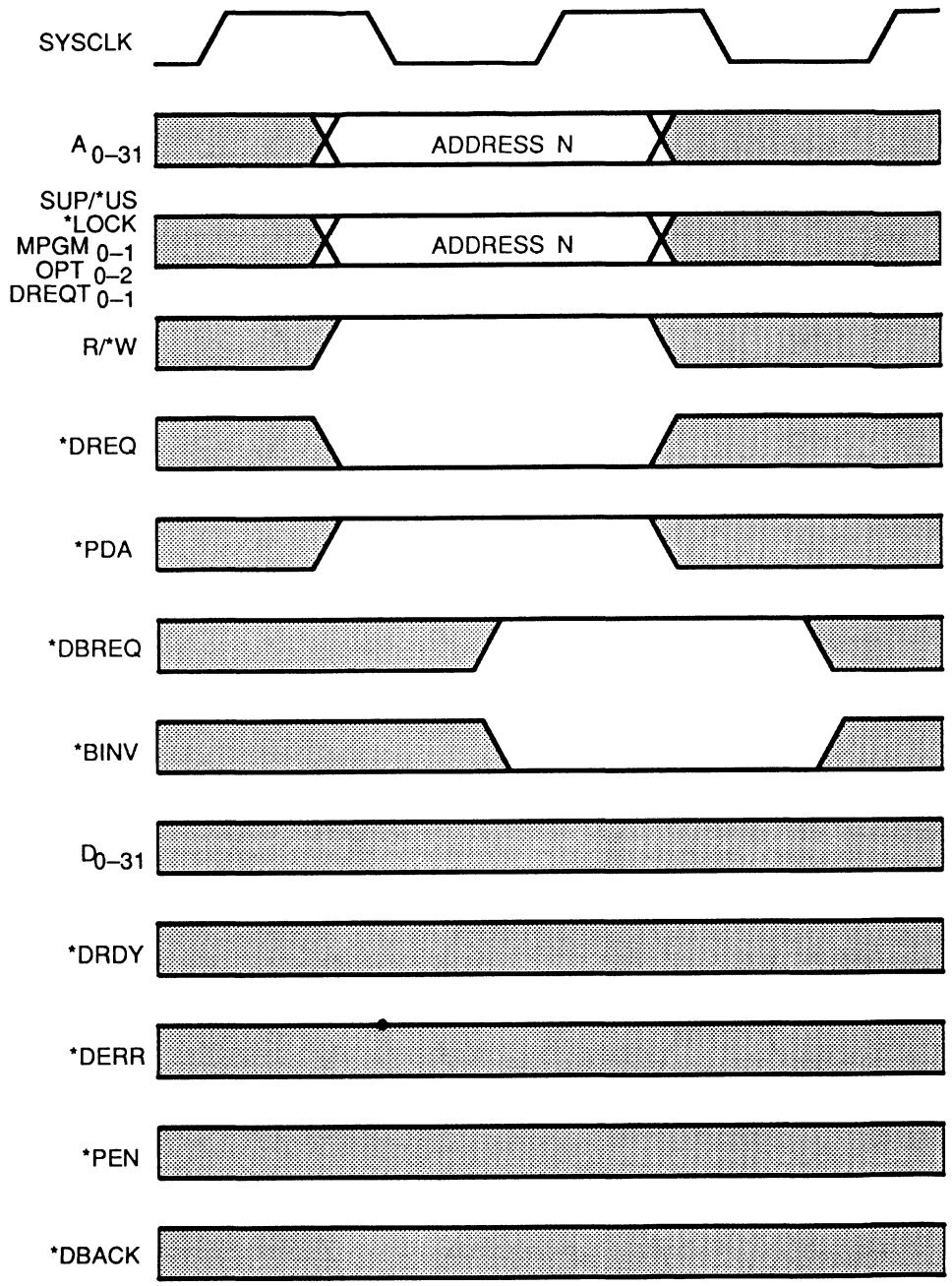


Figure A-38. Data Read—TLB Miss Or Protection Violation

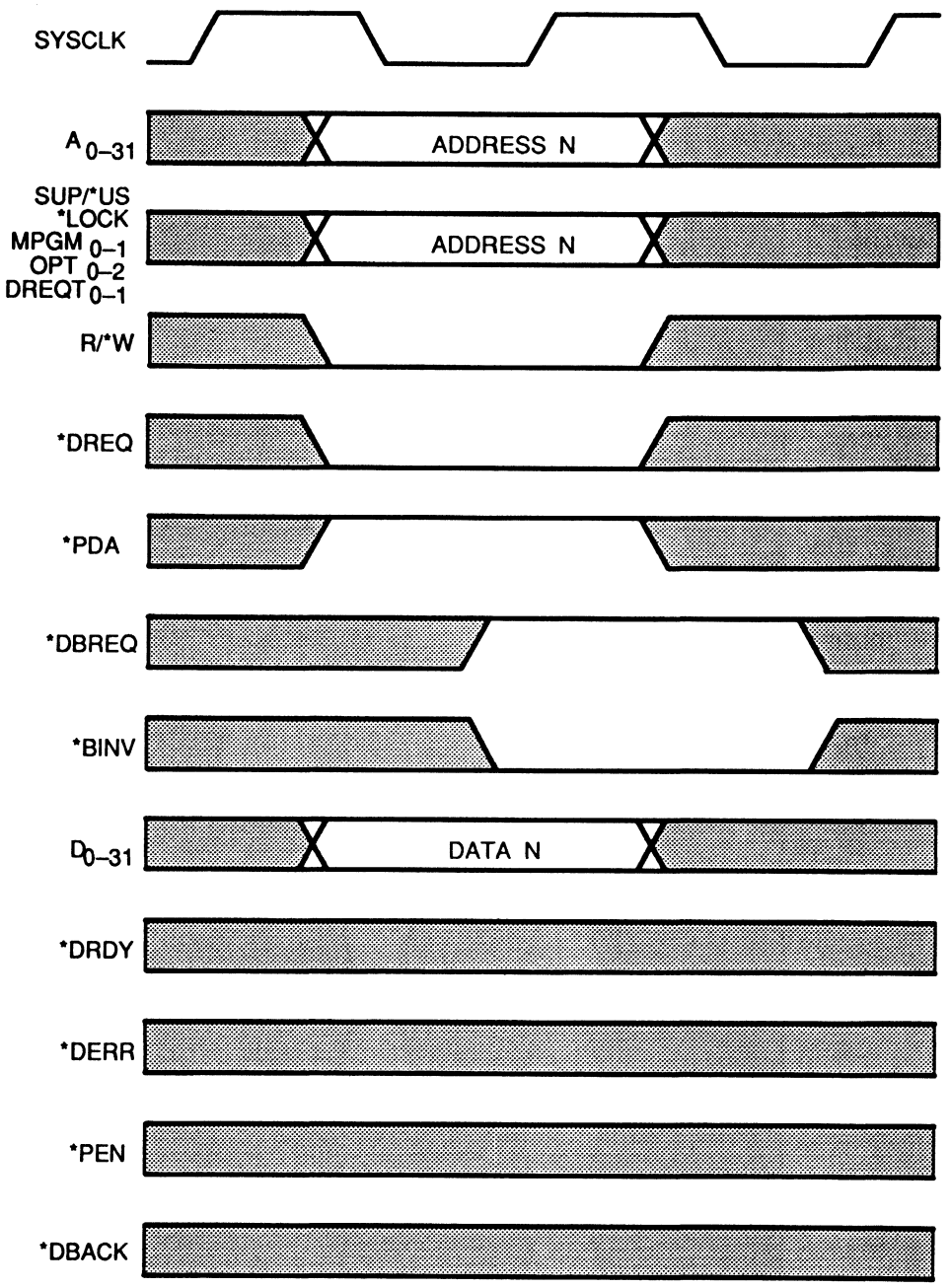


Figure A-39. Data Write—TLB Miss Or Protection Violation

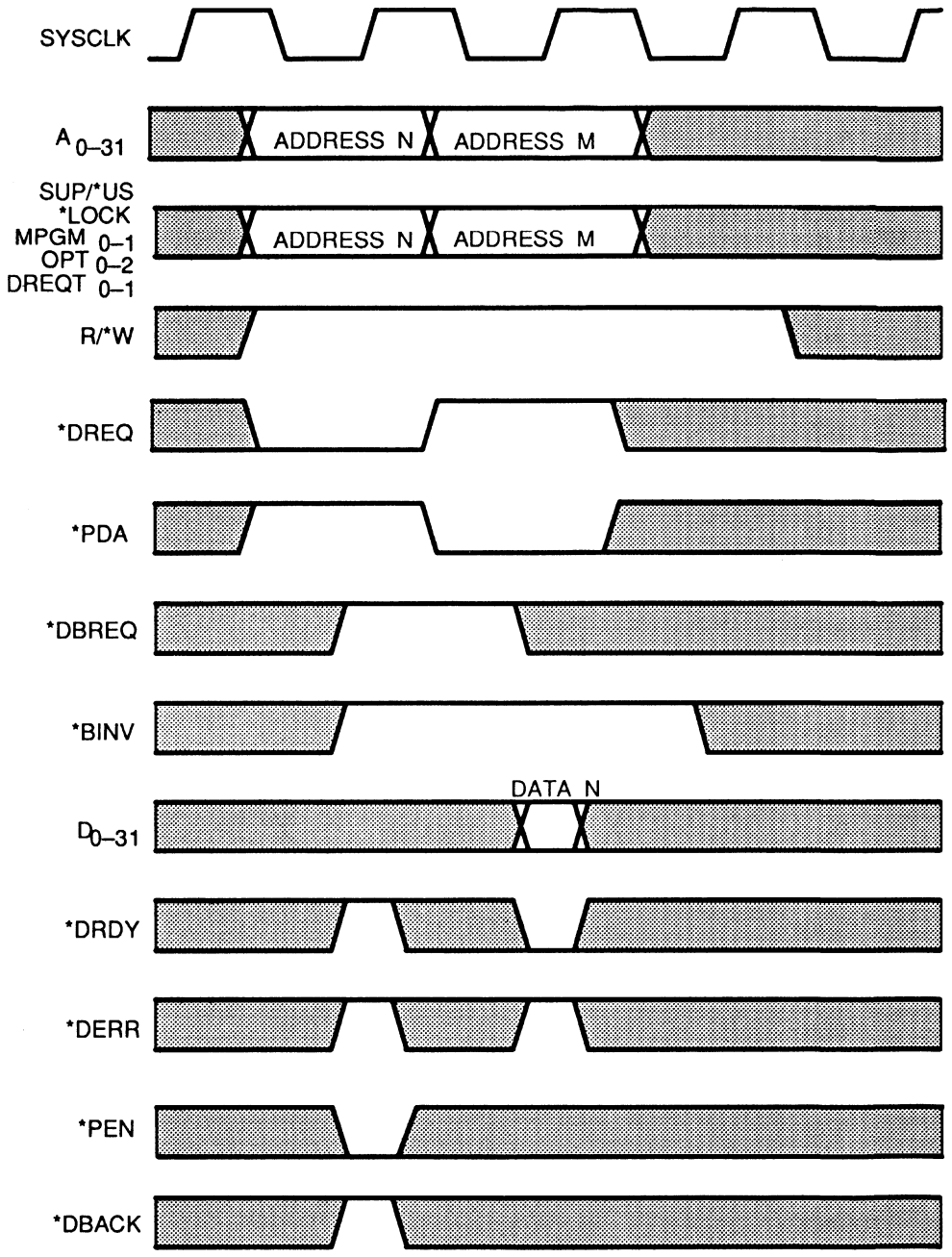


Figure A-40. Data Read—Pipelined Access With TLB Miss Or Protection Violation

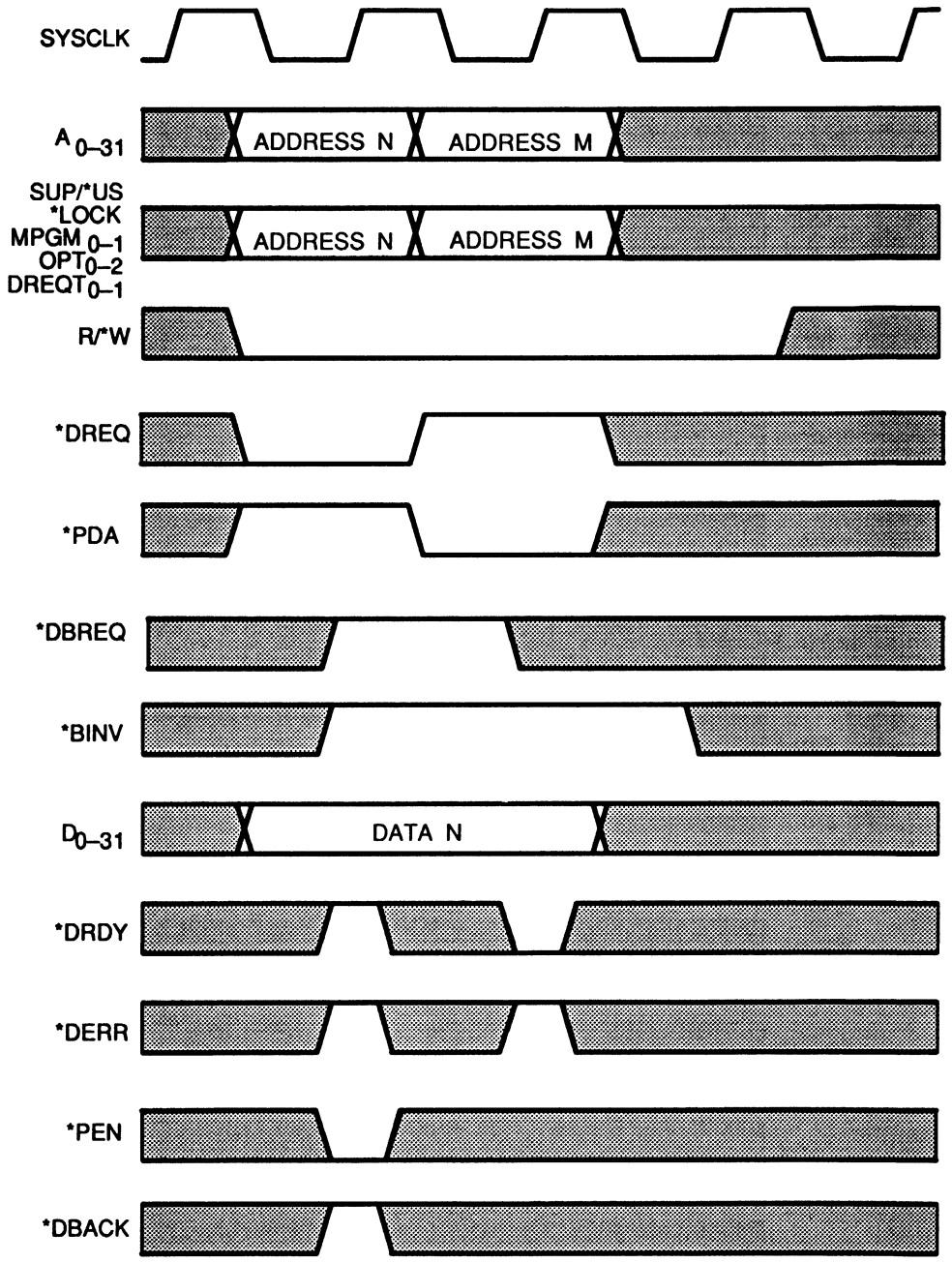


Figure A-41. Data Write—Pipelined Access With TLB Miss Or Protection Violation

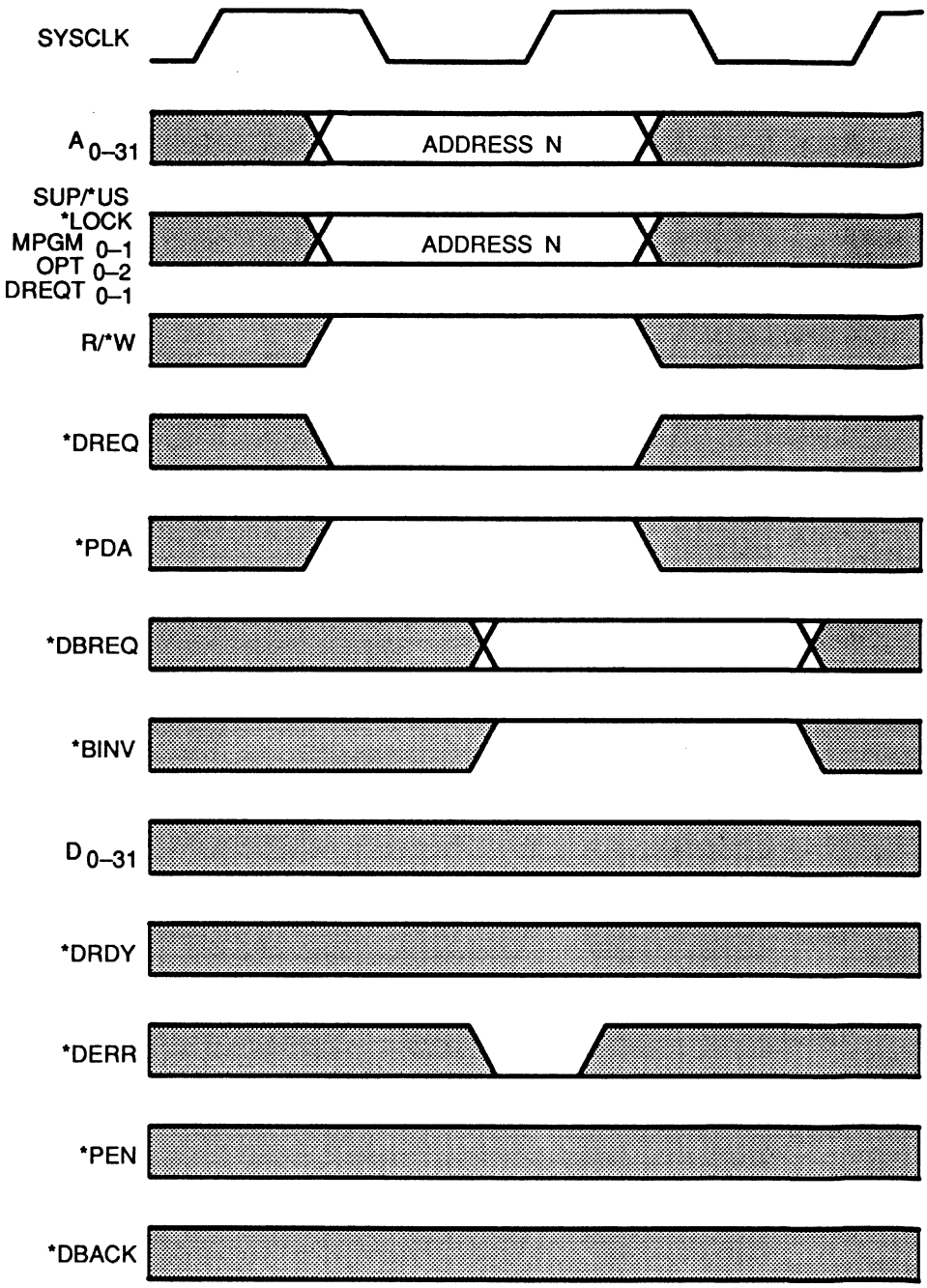


Figure A-42. Data Read—Error Detected By Slave

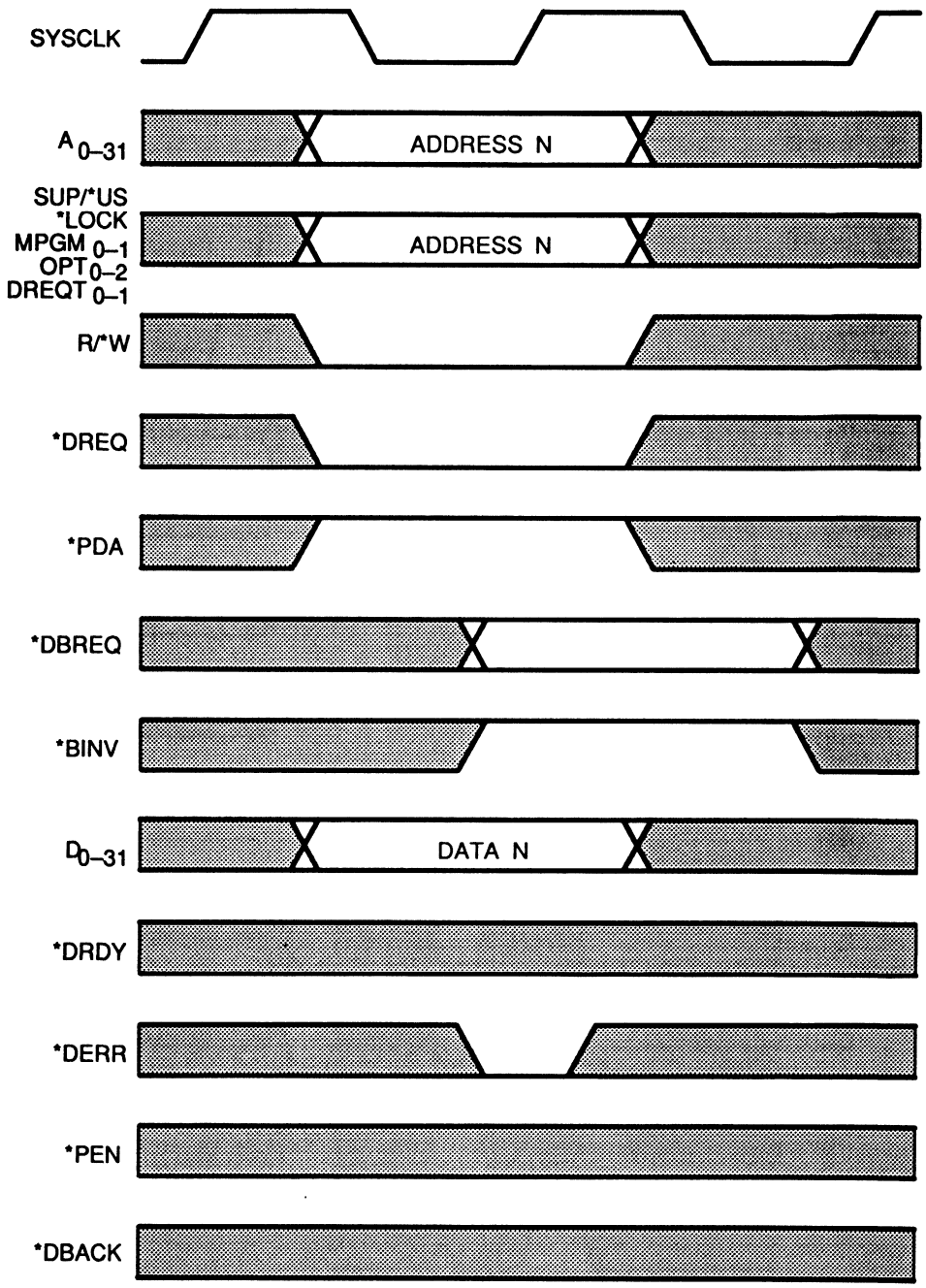


Figure A-43. Data Write—Error Detected By Slave

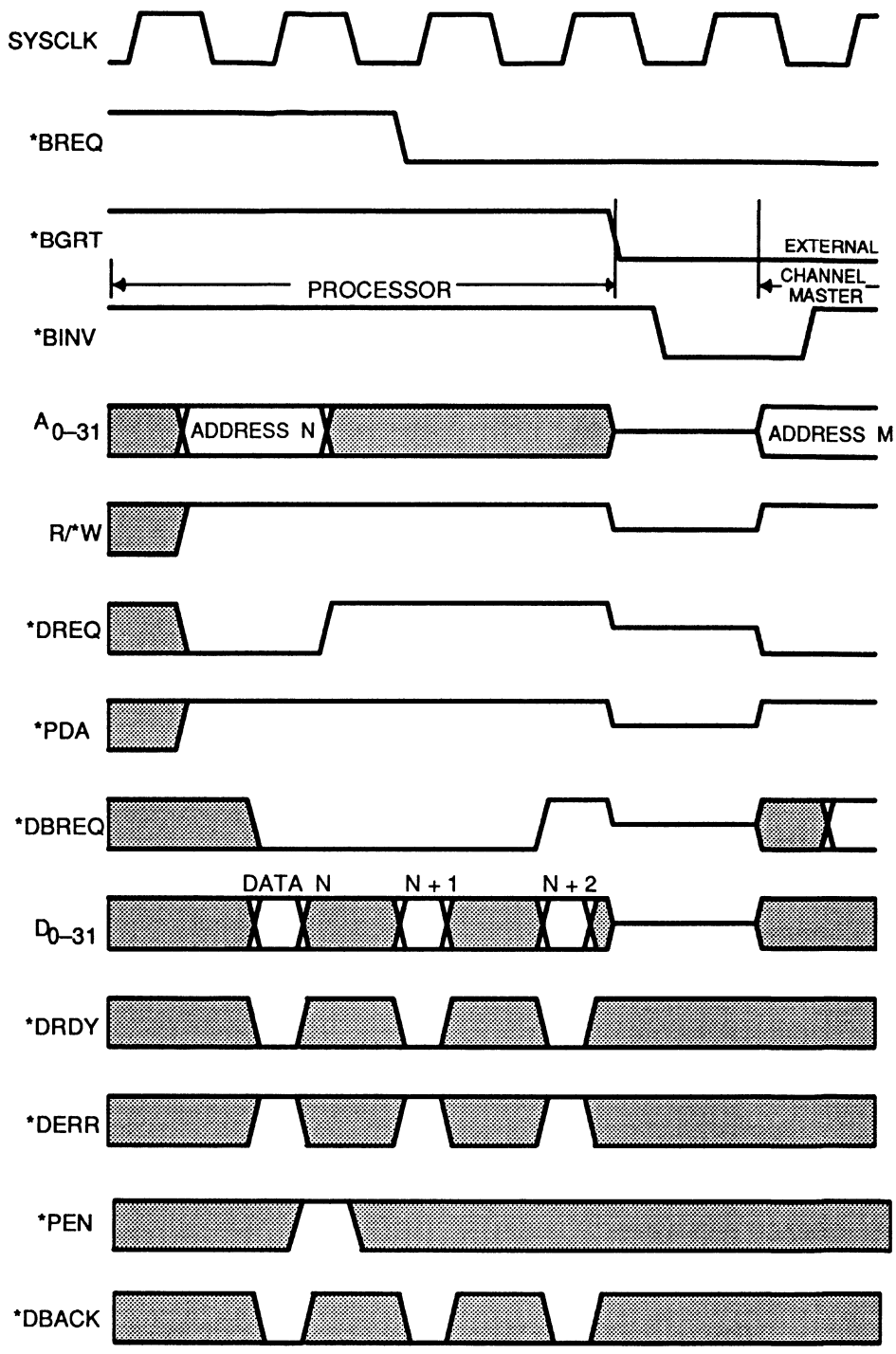


Figure A-44. Channel Transfer From Processor To External Master

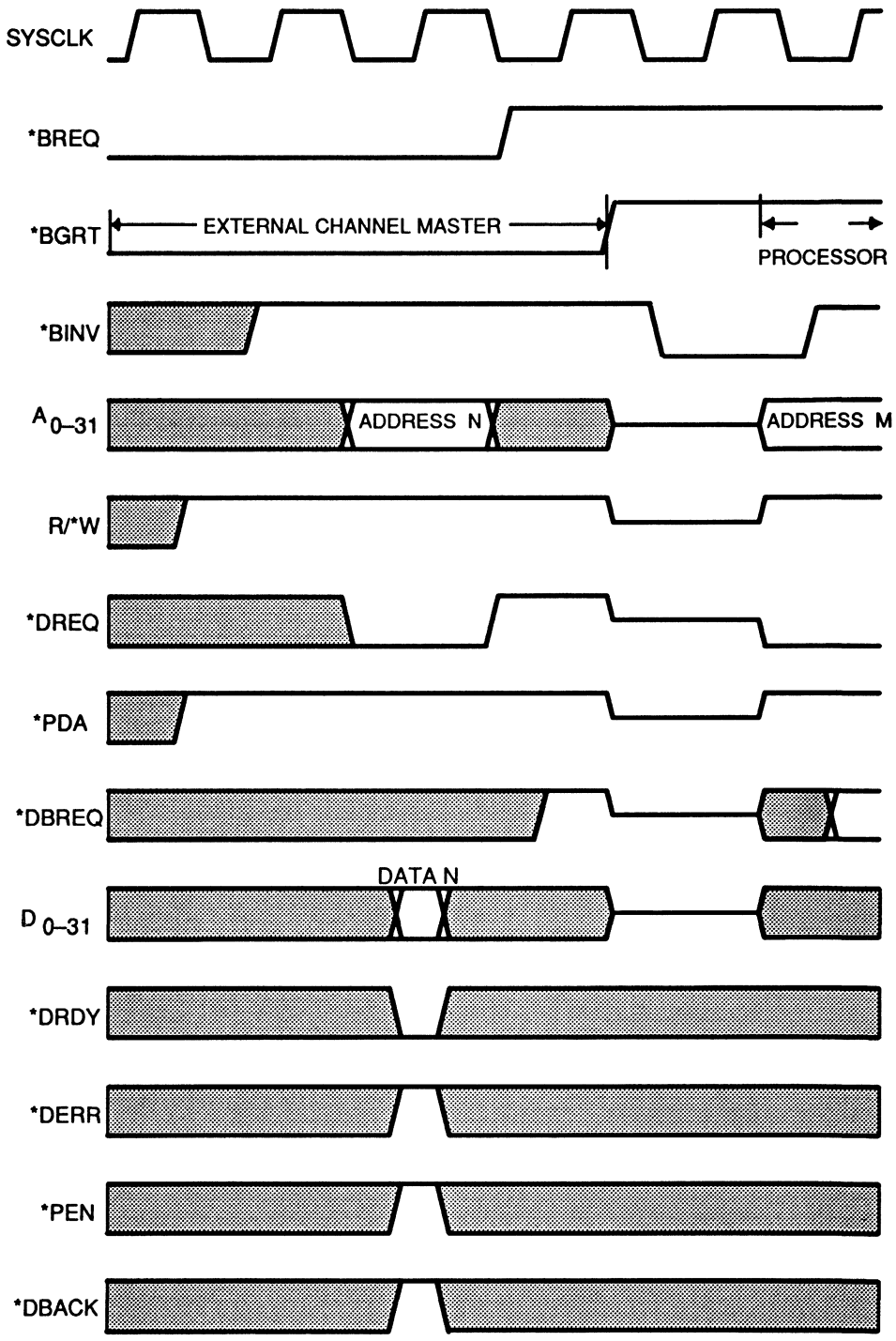


Figure A-45. Channel Transfer From External Master To Processor

APPENDIX B. REGISTER SUMMARY

Absolute REG #	GENERAL-PURPOSE REGISTER
0	Indirect Pointer Access
1	Stack Pointer

2 THRU 63	not implemented

GLOBAL REGISTERS

64	GLOBAL REGISTER	64
65	GLOBAL REGISTER	65
66	GLOBAL REGISTER	66
•	•	
•	•	
•	•	
126	GLOBAL REGISTER	126
127	GLOBAL REGISTER	127

LOCAL REGISTERS

128	LOCAL REGISTER	125
129	LOCAL REGISTER	126
130	LOCAL REGISTER	127
131	LOCAL REGISTER	0
132	LOCAL REGISTER	1
•	•	
•	•	
•	•	
254	LOCAL REGISTER	123
255	LOCAL REGISTER	124

STACK POINTER =131 (example)

Figure B-1. General-Purpose Register 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-2. Register Bank Organization

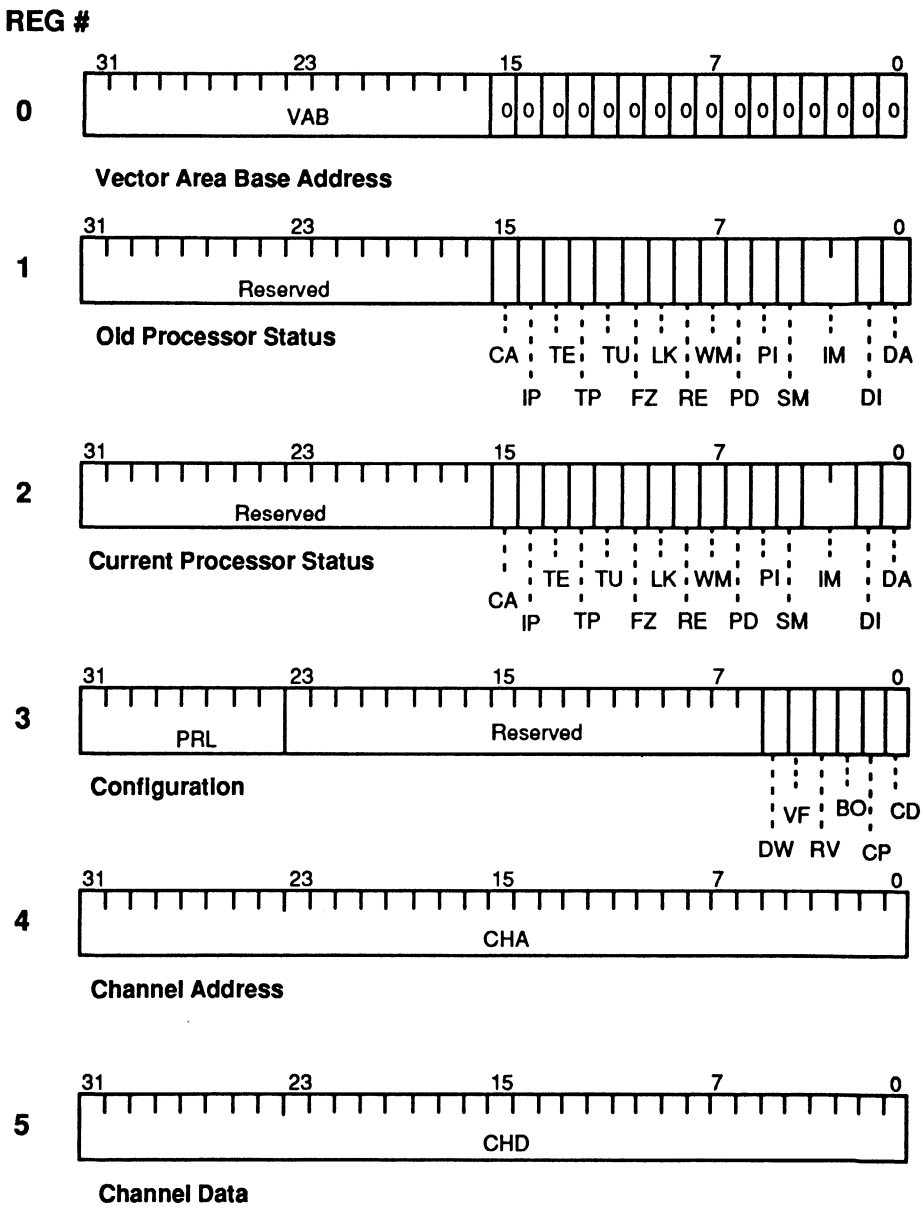


Figure B-3. Special Purpose Registers

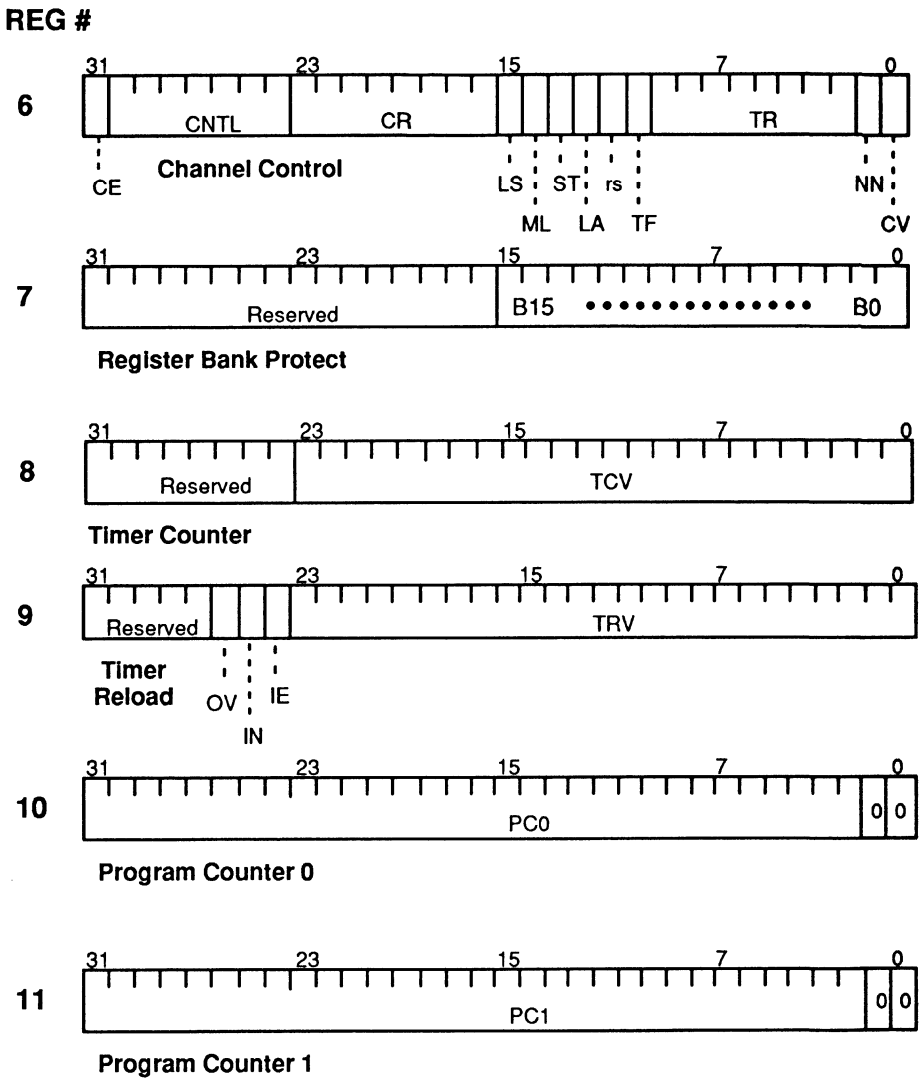
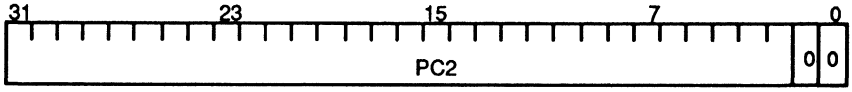


Figure B-3. Special Purpose Registers (continued)

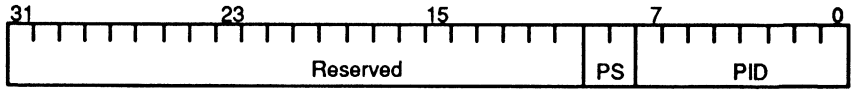
REG #

12



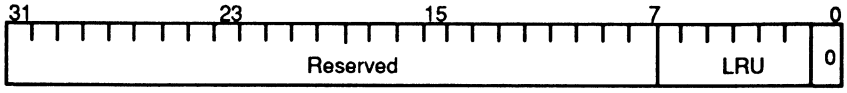
Program Counter 2

13



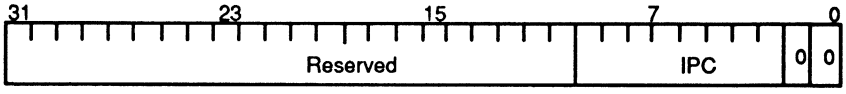
MMU Configuration

14



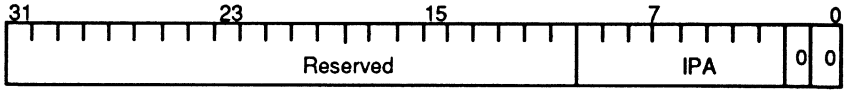
LRU Recommendation

128



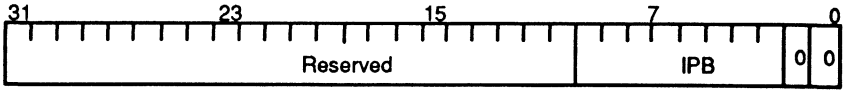
Indirect Pointer C

129



Indirect Pointer A

130



Indirect Pointer B

Figure B-3. Special Purpose Registers (continued)

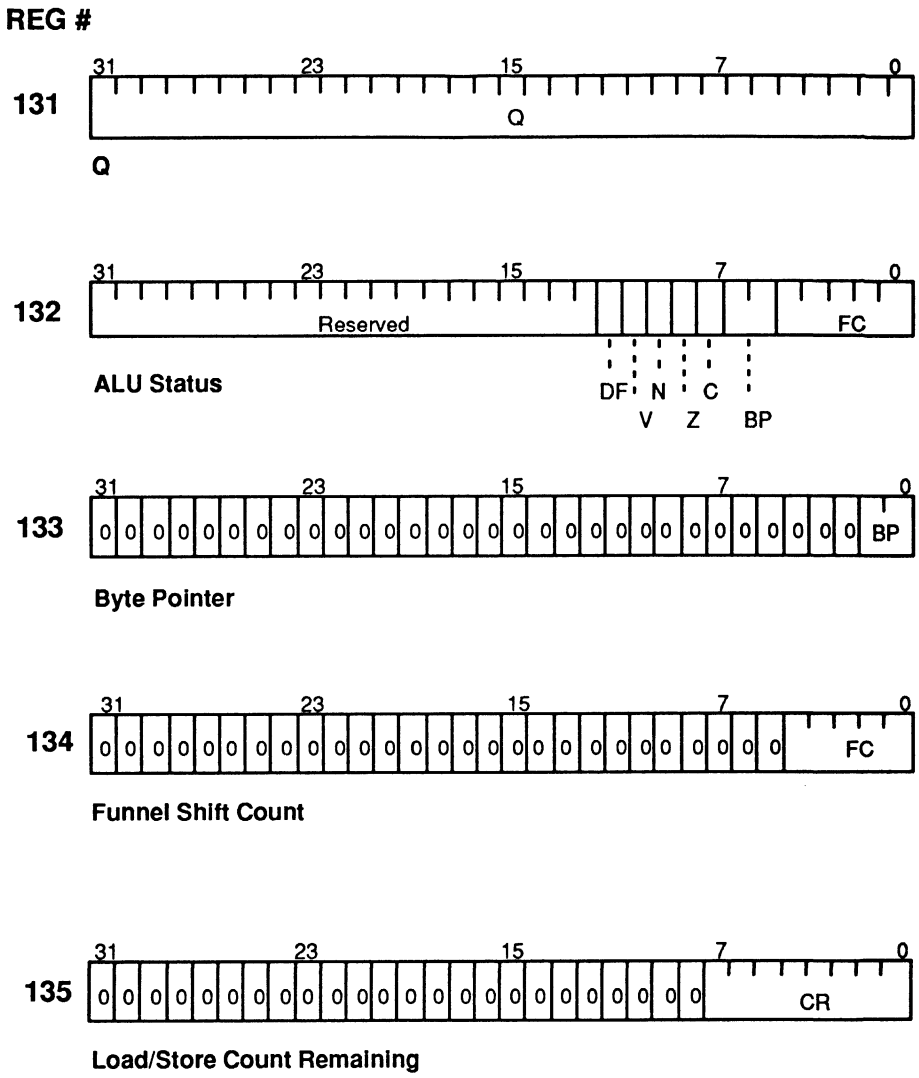


Figure B-3. Special Purpose Registers (continued)

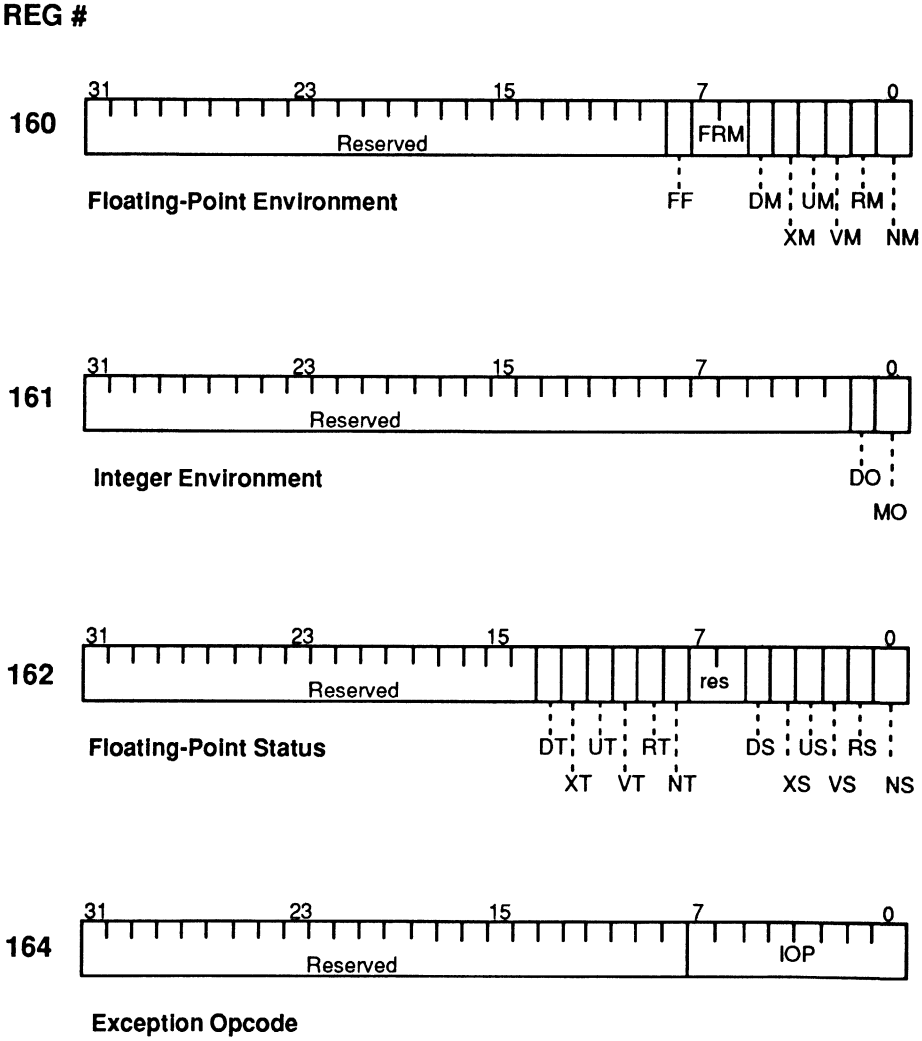


Figure B-3. Special Purpose Registers (continued)

REG #	TLB Set 0
0	TLB Entry Line 0 Word 0
1	TLB Entry Line 0 Word 1
2	TLB Entry Line 1 Word 0
3	TLB Entry Line 1 Word 1
60	TLB Entry Line 30 Word 0
61	TLB Entry Line 30 Word 1
62	TLB Entry Line 31 Word 0
63	TLB Entry Line 31 Word 1
	TLB Set 1
64	TLB Entry Line 0 Word 0
65	TLB Entry Line 0 Word 1
66	TLB Entry Line 1 Word 0
67	TLB Entry Line 1 Word 1
124	TLB Entry Line 30 Word 0
125	TLB Entry Line 30 Word 1
126	TLB Entry Line 31 Word 0
127	TLB Entry Line 31 Word 1

Figure B-4. Translation Look-Aside Buffer Registers

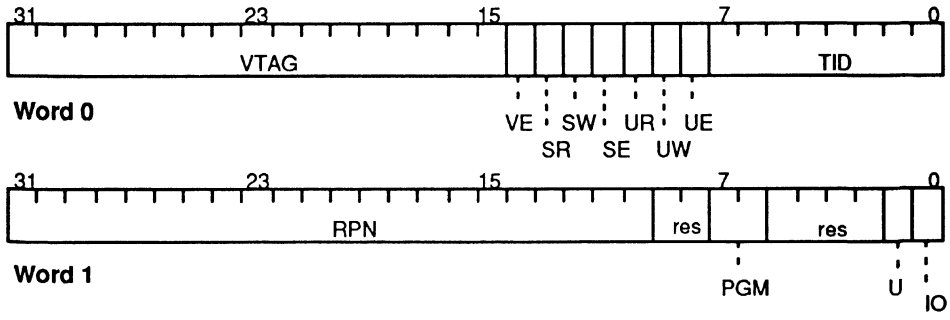


Figure B-5. Translation Look-Aside Buffer Entries

Table B-1. Register Field Summary

Label	Field Name	Register	Bit
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
CA	Coprocessor Active	Current Processor Status Old Processor Status	15 15
CD	Branch Target Cache Disable	Configuration	0
CE	Coprocessor Enable	Channel Control	31
CHA	Channel Address	Channel Address	31–0
CHD	Channel Data	Channel Data	31–0
CNTL	Control	Channel Control	30–24
CP	Coprocessor Present	Configuration	1
CR	Load/Store Count Remaining	Channel Control Load/Store Count Remaining	23–16 7–0
CV	Contents Valid	Channel Control	0
DA	Disable All Interrupts and Traps	Current Processor Status Old Processor Status	0 0

(Table continued)

Table B-1. Register Field Summary (continued)

Label	Field Name	Register	Bit
DF	Divide Flag	ALU Status	11
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
DW	Data Width Enable	Configuration	5
FF	Fast Floating-Point Select	Floating-Point Environment	8
FRM	Floating-Point Round Mode	Floating-Point Environment	7–6
FC	Funnel Shift Count	ALU Status Funnel Shift Count	4–0 4–0
FZ	Freeze	Current Processor Status Old Processor Status	10 10
IE	Interrupt Enable	Timer Reload	24
IM	Interrupt Mask	Current Processor Status Old Processor Status	3–2 3–2
IN	Interrupt	Timer Reload	25
IO	Input/Output	TLB Entry Word 1	0
IOP	Instruction Opcode	Exception Opcode	7–0
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

(Table continued)

Table B-1. Register Field Summary (continued)

Label	Field Name	Register	Bit
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
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
PRL	Processor Release Level	Configuration	31–24
PS	Page Size	MMU Configuration	9–8
Q	Quotient/Multiplier	Q Register	31–0
RE	ROM Enable	Current Processor Status Old Processor Status	8 8
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
RV	ROM Vector Area	Configuration	3
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

(Table continued)

Table B-1. Register Field Summary (continued)

Label	Field Name	Register	Bit
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
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	31–16
VE	Valid Entry	TLB Entry Word 0	14
VF	Vector Fetch	Configuration	4
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

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Sales Offices

North American

ALABAMA		(205) 882-9122
ARIZONA		(602) 242-4400
CALIFORNIA,		
Culver City		(213) 645-1524
Newport Beach		(714) 752-6262
Roseville		(916) 786-6700
San Diego		(619) 560-7030
San Jose		(408) 452-0500
Woodland Hills		(818) 992-4155
CANADA, Ontario,		
Kanata		(613) 592-0060
Willowdale		(416) 224-5193
COLORADO		(303) 741-2900
CONNECTICUT		(203) 264-7800
FLORIDA,		
Clearwater		(813) 530-9971
Ft. Lauderdale		(305) 776-2001
Orlando (Casselberry)		(407) 830-8100
GEORGIA		(404) 449-7920
ILLINOIS,		
Chicago (Itasca)		(312) 773-4422
Naperville		(312) 505-9517
KANSAS		(913) 451-3115
MARYLAND		(301) 796-9310
MASSACHUSETTS		(617) 273-3970
MICHIGAN		(313) 347-1522
MINNESOTA		(612) 938-0001
NEW JERSEY,		
Cherry Hill		(609) 662-2900
Parsippany		(201) 299-0002
NEW YORK,		
Liverpool		(315) 457-5400
Poughkeepsie		(914) 471-8180
Rochester		(716) 272-9020
NORTH CAROLINA		(919) 878-8111
OHIO,		
Columbus (Westerville)		(614) 891-6455
OREGON		(503) 245-0080
PENNSYLVANIA		(215) 398-8006
SOUTH CAROLINA		(803) 772-6760
TEXAS,		
Austin		(512) 346-7830
Dallas		(214) 934-9099
Houston		(713) 785-9001
UTAH		(801) 264-2900

International

BELGIUM, Bruxelles	TEL	(02) 771-91-42
	FAX	(02) 762-37-12
	TLX	846-61028
FRANCE, Paris	TEL	(1) 49-75-10-10
	FAX	(1) 49-75-10-13
	TLX	263282F
WEST GERMANY,		
Hannover area	TEL	(0511) 736085
	FAX	(0511) 721254
	TLX	922850
München	TEL	(089) 4114-0
	FAX	(089) 406490
	TLX	523883
Stuttgart	TEL	(0711) 62 33 77
	FAX	(0711) 625187
	TLX	721882
HONG KONG,	TEL	852-5-8654525
Wanchai	FAX	852-5-8654335
	TLX	67955AMDAPHX
ITALY, Milan	TEL	(02) 3390541
		(02) 3533241
	FAX	(02) 3498000
	TLX	843-315286
JAPAN,		
Kanagawa	TEL	462-47-2911
	FAX	462-47-1729
Tokyo	TEL	(03) 346-7550
	FAX	(03) 342-5196
	TLX	J24064AMDTKOJ
Osaka	TEL	06-243-3250
	FAX	06-243-3253

International (Continued)

KOREA, Seoul	TEL	822-784-0030
	FAX	822-784-8014
LATIN AMERICA,		
Ft. Lauderdale	TEL	(305) 484-8600
	FAX	(305) 485-9736
	TLX	5109554261 AMDFTL
NORWAY, Hovik	TEL	(03) 010156
	FAX	(02) 591959
	TLX	79079HBCN
SINGAPORE	TEL	65-3481188
	FAX	65-3480161
	TLX	55650 AMDMMI
SWEDEN,		
Stockholm	TEL	(08) 733 03 50
(Sundbyberg)	FAX	(08) 733 22 85
	TLX	11602
TAIWAN	TEL	886-2-7213393
	FAX	886-2-7723422
	TLX	886-2-7122066
UNITED KINGDOM,		
Manchester area	TEL	(0925) 828008
(Warrington)	FAX	(0925) 827693
	TLX	851-628524
London area	TEL	(0483) 740440
(Woking)	FAX	(0483) 756196
	TLX	851-859103

North American Representatives

CANADA
Burnaby, B.C.
DAVETEK MARKETING (604) 430-3680
Calgary, Alberta
DAVETEK MARKETING (403) 291-4984
Kanata, Ontario
VITEL ELECTRONICS (613) 592-0060
Mississauga, Ontario
VITEL ELECTRONICS (416) 676-9720
Lachine, Quebec
VITEL ELECTRONICS (514) 636-5951
IDAHO
INTERMOUNTAIN TECH MKTG, INC (208) 888-6071
ILLINOIS
HEARTLAND TECH MKTG, INC (312) 577-9222
INDIANA
Huntington - ELECTRONIC MARKETING CONSULTANTS, INC (317) 921-3450
Indianapolis - ELECTRONIC MARKETING CONSULTANTS, INC (317) 921-3450
IOWA
LORENZ SALES (319) 377-4666
KANSAS
Merriam – LORENZ SALES (913) 384-6556
Wichita – LORENZ SALES (316) 721-0500
KENTUCKY
ELECTRONIC MARKETING CONSULTANTS, INC (317) 921-3452
MICHIGAN
Birmingham - MIKE RAICK ASSOCIATES (313) 644-5040
Holland – COM-TEK SALES, INC (616) 392-7100
Novi – COM-TEK SALES, INC (313) 344-1409
MISSOURI
LORENZ SALES (314) 997-4558
NEBRASKA
LORENZ SALES (402) 475-4660
NEW MEXICO
THORSON DESERT STATES (505) 293-8555
NEW YORK
East Syracuse – NYCOM, INC (315) 437-8343
Woodbury – COMPONENT CONSULTANTS, INC (516) 364-8020
OHIO
Centerville – DOLFUSS ROOT & CO (513) 433-6776
Columbus – DOLFUSS ROOT & CO (614) 885-4844
Strongsville – DOLFUSS ROOT & CO (216) 238-0300
PENNSYLVANIA
DOLFUSS ROOT & CO (412) 221-4420
PUERTO RICO
COMP REP ASSOC, INC (809) 746-6550
UTAH, R² MARKETING (801) 595-0631
WASHINGTON
ELECTRA TECHNICAL SALES (206) 821-7442
WISCONSIN
HEARTLAND TECH MKTG, INC (414) 792-0920

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 TOLL FREE: (800) 222-9323 • (408) 749-5703

2/2/90
BTA/16.5M/5-90/0 Printed in USA

ADVANCED MICRO DEVICES, INC.
901 Thompson Place
P.O. Box 3453
Sunnyvale,
California 94088-3453
(408) 732-2400
TWX: 910-339-9280
TELEX: 34-6306
TOLL-FREE
(800) 538-8450

APPLICATIONS HOTLINE
(800) 222-9323
(408) 749-5703

SUPPORT PRODUCTS ENGINEERING HOTLINE
USA (800) 2929-AMD
UK 0-800-89-1131
JAPAN 0-031-11-1129

Printed in USA
10620C