Am29200™
RISC Microcontroller
User's Manual
and Data Sheet

ADVANCED MICRO DEVICES

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION AND OVERVIEW

THE Am29200™ RISC MICROCONTROLLER

The Am29200 microprocessor is the first in a new series of 32-bit processors employing submicron circuits to increase the degree of system integration, yielding very low system cost. Dense circuitry and a large number of on-chip peripherals minimize the number of components required to implement embedded systems, while providing performance superior to that of complex-instruction-set (CISC) microprocessors. New systems implemented with the Am29200 microprocessor can achieve higher performance at lower cost than existing systems. The Am29200 microprocessor is software compatible with all other members of the 29K™ Family, further broadening the cost/performance range of the 29K Family.

The Am29200 microprocessor was designed expressly to meet the requirements of embedded applications such as laser printers, graphics processing, application program interface (API) accelerators, X terminals and servers, and scanners. Such applications make the following demands on system design:

- Performance at low cost: A processor must interface with memory and peripherals with a minimum number of external components.
- Design flexibility: One basic design must be extensible to an entire product line.
- Reduced time-to-market: A complete suite of development, debug, and benchmarking tools is critical for reducing product development time.
- A rational, easy upgrade path: The processor family must provide bus-, pin-, and software-compatibility so processor upgrades are transparent to both hardware and software.

The Am29200 microprocessor is optimized for any embedded application that requires better-than-CISC performance at minimal system cost. The electronic components for many systems, such as personal laser printers, amount to little more than the Am29200 microprocessor, ROM, DRAM, and electrical buffering.

DESIGN PHILOSOPHY

The 29K Family of processors 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, while considered alone may increase one aspect of processor performance, may actually decrease the performance of the total system, because of the burden 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 the terms that contribute to the product; in fact, this is generally not possible due to the interaction of the terms.

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

OPTIMUM PERFORMANCE

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

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

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

PERFORMANCE LEVERAGE

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

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

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

CONCLUSION

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

The Am29200 microprocessor was designed with the above effects in mind. Maximum performance is obtained by the optimization of the product of the number of instructions per task, the number of cycles per instruction, and the cycle time, not by minimizing one factor at the expense of the others. This is accomplished by careful definition of all processor components. In particular:

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

PURPOSE OF THIS MANUAL

This manual describes the technical features, programming interface, on-chip peripherals, and complete instruction set of the Am29200 microprocessor.

INTENDED AUDIENCE

This manual is intended for system hardware and software architects and system engineers who are designing or are considering designing systems based on the Am29200 microprocessor.

Am29200 MICROPROCESSOR USER'S MANUAL OVERVIEW

This manual contains information on the Am29200 microprocessor that is essential for system hardware and software architects and design engineers. Additional information is available in the form of data sheets, application notes, and other documentation provided with software products and hardware-development tools.

The information in this manual is organized into eighteen chapters:

Chapter 1 introduces the features and performance aspects of the Am29200 microprocessor.

Chapter 2 describes the programmer's model of the Am29200 microprocessor, including the instruction set and register model.

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

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

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

Chapter 6 describes the system-protection features provided by the Am29200 microprocessor.

Chapter 7 provides an overview of the processor's system interfaces and the system components that are integrated on-chip.

Chapter 8 describes the ROM Interface.

Chapter 9 describes the DRAM Interface.

Chapter 10 describes the Peripheral Interface Adapter, which is used for glueless attachment of a number of peripheral components.

Chapter 11 describes the DMA Controller.

Chapter 12 describes the Programmable I/O Port.

Chapter 13 describes the Parallel Port.

Chapter 14 describes the Serial Port.

Chapter 15 describes the Video Interface.

Chapter 16 provides a description of the interrupt and trap mechanism and the handling of interrupts and traps, including the operation of the on-chip Interrupt Controller.

Chapter 17 describes the software and hardware facilities for debugging and testing.

Chapter 18 provides a detailed description of the instruction set.

For users already familiar with other 29K Family processors, Chapters 7 through 15 describe the on-chip peripherals and system functions unique to the Am29200 microprocessor.

For those readers desiring only a brief overview of the Am29200 microprocessor, Chapter 1 identifies the outstanding features of the processor. This chapter addresses the basic software and hardware concerns. Chapters 2, 3, and 5 are recommended reading for both hardware and software developers.

For software architects and system programmers interested mainly in software-related issues, Chapters 4, 6, and 16 provide the necessary information. Chapters 17 and 18 also provide related information.

For hardware architects and systems hardware designers interested mainly in hardware-related issues, Chapters 7 through 15, Chapter 17, and Appendix C provide most of the required information. Chapters 5 and 18 also provide related information.

29K FAMILY DOCUMENTATION

ORDER NO.	TITLE
10620	**29K User's Manual** Describes the Am29000™ microprocessor's technical features, programming interface, and complete instruction set.
11426	**Fusion29K℠ Catalog** Provides information on more than 100 tools that speed a 29K Family embedded product to market. Includes products from over 50 expert suppliers of embedded development solutions. Design solution chapters include: laser printer and OCR solutions, graphics solutions, and networking solutions.
12990	**Fusion29K Newsletter** Contains quarterly updates on developments in the 29K Family.
14779	**Am29050™ User's Manual** Describes the Am29050 microprocessor's technical features, programming interface, and complete instruction set.
15723	**Am29030™ and Am29035™ User's Manual** Describes the Am29030 and Am29035 microprocessors' technical features, programming interface, and complete instruction set.
15176	**29K Laser Printer Solutions Brochure** Reviews how the 29K Family of microprocessors fits into the laser printer marketplace. Includes a description of AMD's PCL and PostScript® Laser29K™ Low-Cost Raster Image Processor demonstration boards.
12175	**29K Family Data Book** Provides a comprehensive collection of data sheets for the Am29000 microprocessor, Am29027™ arithmetic accelerator, HighC29K™ Cross Development Toolkit, and XRAY29K™ Source-Level Debugger. It also includes application notes to help shorten designers' learning curves and hardware and software development time.
10626	**XRAY29K Data Sheet**
10957	**High C 29K Data Sheet**
13089	**Am29005™ Data Sheet**
14721	**EB29K™ Data Sheet**
15039	**Am29050 Data Sheet**

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

RELATED PUBLICATIONS

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

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

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

CHAPTER 1

FEATURES AND PERFORMANCE

This chapter provides an evaluation of the Am29200 microprocessor as an aid in considering a particular application. A detailed technical description of the Am29200 microprocessor is contained in subsequent chapters. This chapter informally describes the features of the processor, concentrating on features which distinguish the Am29200 microprocessor from other available processors and describing how these features enhance system performance and cost-effectiveness. This chapter consists of the following sections:

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

1.1 DISTINCTIVE CHARACTERISTICS

- Completely integrated system for embedded applications
- Full 32-bit architecture
- CMOS technology/TTL-compatible
- 16-MHz operational frequency
- Cost/performance flexibility. Support for several low-cost memory configurations allows performance points of 8, 6, 5, and 3 million instructions per second sustained
- 4-GB effective address space, 304 Mbytes implemented
- 192 general-purpose registers
- Three-address instruction architecture
- Fully pipelined
- Glueless system interfaces with on-chip wait-state control
- 8-,16-, or 32-bit ROM interface
- 16- or 32-bit DRAM interface
- Burst-mode and page-mode access support
- DRAM mapping on-chip
- Two-channel DMA controller
- Six-port peripheral interface adapter
- 16-line programmable I/O port
- Bi-directional video interface
- Serial port (UART)

- Bi-directional parallel port for IBM-compatible personal computers
- Interrupt controller
- On-chip timer
- Enhanced debugging support
- IEEE Std.1149.1-1990 (JTAG) compliant Standard Test Access Port and Boundary-Scan Architecture implementation

1.2 KEY FEATURES AND BENEFITS

The Am29200 RISC microcontroller is a highly integrated, 32-bit embedded processor implemented in complementary metal-oxide semiconductor (CMOS) technology. It is targeted primarily at printer, imaging, and graphics applications, using a flexible architecture, a complete set of common system peripherals on-chip, and glueless interfacing to external memories and peripherals.

The Am29200 microprocessor also establishes a new line of RISC microcontrollers based on the 29K architecture. This RISC microcontroller product line will allow users who do not require the very high performance of other 29K Family members to capitalize on the very low system cost made possible by the integration of processor and peripherals.

The Am29200 microprocessor expands the cost/performance range of systems that can be built with the 29K Family. The Am29200 microprocessor is fully software compatible with the Am29000, Am29005™, Am29030, Am29035, and Am29050 microprocessors. It can be used in most existing Am29000 microprocessor applications without software modifications.

A block diagram of the Am29200 microprocessor is shown in Figure 1-1.

1.2.1 Complete Set of Common System Peripherals

The Am29200 microprocessor minimizes system cost by incorporating a complete set of system facilities commonly found in embedded applications, eliminating the cost of additional components. The on-chip functions include: a ROM Controller, a DRAM Controller, a Peripheral Interface Adapter, a DMA Controller, a Programmable I/O Port, a Parallel Port, a Serial Port, and an Interrupt Controller. A Video Interface is also included for printer, scanner, and other imaging applications. These facilities allow many simple systems to be built using only the Am29200 microprocessor and external ROM and/or DRAM memory.

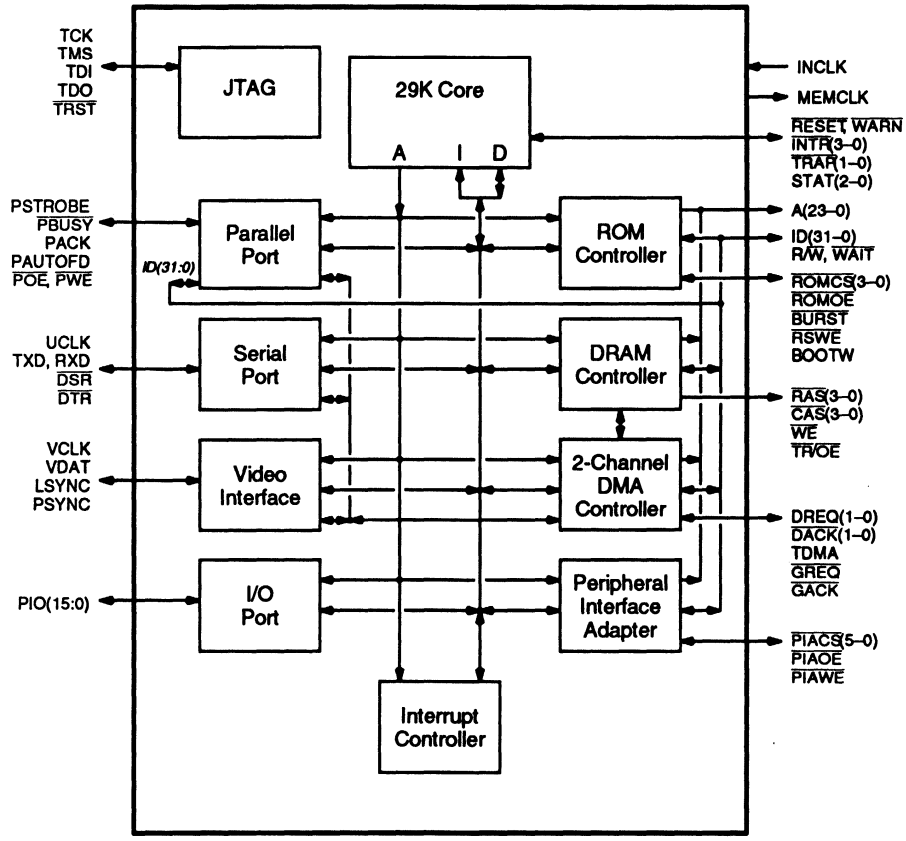
1.2.1.1 ROM CONTROLLER (Chapter 8)

The ROM Controller supports four individual banks of ROM or other static memory, each with its own timing characteristics. Each ROM bank may be of a different size and may be either 8, 16, or 32 bits wide. The ROM banks appear as a contiguous memory area of up to 64 Mb in size. The ROM Controller also supports writes to the ROM memory space, for devices such as flash EPROMs and SRAMs.

1.2.1.2 DRAM CONTROLLER (Chapter 9)

The DRAM Controller supports four separate banks of dynamic memory. Each bank may be a different size and may be either 16 or 32 bits wide. The DRAM banks appear as a contiguous memory area of up to 64 Mb in size. Four, 64-Kb regions of the DRAM can be mapped into a 16-Mb virtual address space, supporting system functions such as on-the-fly data compression and decompression.

Figure 1-1 Am29200 Microprocessor Block Diagram



1.2.1.3 DMA CONTROLLER (Chapter 11)

The DMA controller provides two channels for transfer of data between the DRAM and internal or external peripherals. One of the DMA channels is double buffered to relax the constraints on the reload time.

1.2.1.4 PERIPHERAL INTERFACE ADAPTER (Chapter 10)

The Peripheral Interface Adapter (PIA) permits glueless interfacing to as many as six external peripheral chips. The PIA allows for additional system features implemented by external peripheral chips.

1.2.1.5 INTERRUPT CONTROLLER (Section 16.8)

The Interrupt Controller generates and reports the status of interrupts caused by on-chip peripherals.

1.2.1.6 I/O PORT (Chapter 12)

The I/O Port permits direct access to 16 individually programmable external input/output signals. Eight of these signals can be configured to cause interrupts.

1.2.1.7 VIDEO INTERFACE (Chapter 15)

The Video Interface permits direct connection to a number of laser marking engines, video displays, or raster input devices such as scanners.

1.2.1.8 SERIAL PORT (Chapter 14)

The Serial Port implements a full-duplex UART.

1.2.1.9 PARALLEL PORT (Chapter 13)

The Parallel Port implements a bidirectional IBM PC-compatible parallel interface to a host processor.

1.2.2 Wide Range of Price/Performance Points

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

The Am29200 microprocessor provides this capability through programmable memory widths, programmable wait states, burst-mode and page-mode access support, and hardware and 29K software compatibility. A system can be upgraded without hardware and software redesign using various memory architectures.

The ROM controller accommodates memories that are either 8, 16, or 32 bits wide, and the DRAM controller accommodates dynamic memories that are either 16 or 32 bits wide. This unique feature provides a flexible interface to low-cost memory as well as a convenient, flexible upgrade path. For example, a system can start with a 16-bit memory design and can subsequently improve performance by migrating to a 32-bit memory design. One particular advantage is the ability to add memory in half-megabyte increments. This provides significant cost savings for applications that do not require larger memory upgrades.

The Am29200 microprocessor allows users to address a wide range of cost performance points, with higher performance and lower cost than existing designs based on CISC microprocessors.

1.2.3 Glueless System Interfaces

The Am29200 microprocessor also minimizes system cost by providing a glueless attachment to external ROMs, DRAMs, and other peripheral components. Processor outputs have edge-rate control that allows them to drive a wide range of load capacitances with low noise and ringing. This eliminates the cost of external logic and buffering.

1.2.4 Pin-, Bus-, and Software-Compatibility

Compatibility within a processor family is critical for achieving a rational, easy upgrade path. The Am29200 microprocessor is the first member of a pin-compatible series of RISC microcontrollers. Future members of this family will improve in price and performance and system capabilities without requiring that users redesign their

system hardware or software. Pin compatibility ensures a convenient upgrade path for future systems.

Moreover, the Am29200 microprocessor is software compatible with existing members of the 29K Family (the Am29000, Am29005, Am29030, Am29035, and Am29050 microprocessors). The Am29200 microprocessor provides a migration path to low cost, highly integrated systems from other 29K Family members, without requiring expensive rewrites of applications software.

1.2.5 Complete Development and Support Environment

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

Advanced Micro Devices offers a complete set of hardware and software tools for design, integration, debugging, and benchmarking. These tools, which are available now for the 29K Family, include the following:

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

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

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

1.3 PERFORMANCE OVERVIEW

The Am29200 microprocessor offers a significant margin of performance over CISC microprocessors in existing embedded designs, since the majority of processor features were defined for the maximum achievable performance at a very low cost. This section describes the features of the Am29200 microprocessor from the point of view of system performance.

1.3.1 Instruction Timing (Section 2.1)

The Am29200 microprocessor 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.

The performance degradation of load and store operations is minimized in the Am29200 microprocessor by overlapping them with instruction execution, by taking advantage of pipelining, and by organizing the flow of external data into the processor so that the impact of external accesses is minimized.

1.3.2 Pipelining (Section 5.1)

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

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

1.3.3 Burst-Mode and Page-Mode Memories (Sections 8.2.4, 9.2.6)

The Am29200 microprocessor directly supports burst-mode memories such as AMD's 27B010 burst-mode EPROM. The burst-mode memory supplies instructions at the maximum bandwidth, without the complexity of an external cache or the performance degradation due to cache misses.

The Am29200 microprocessor also can use the page-mode capability of common DRAMs to improve the access time in cases where page-mode accesses can be used. This is particularly useful in very low-cost systems with 16-bit-wide DRAMs, where the DRAM must be accessed twice for each 32-bit operand.

1.3.4 Instruction Set Overview (Chapter 2)

The Am29200 microprocessor employs a three-address instruction set architecture. The compiler or assembly-language programmer is given complete freedom to allocate register usage. There are 192 general-purpose registers, allowing the retention of intermediate calculations and avoiding needless data destruction. Instruction operands may be contained in any of the general-purpose registers, and the results may be stored into any of the general-purpose registers.

The Am29200 instruction set contains 117 instructions which are divided into nine classes. These classes are integer arithmetic, compare, logical, shift, data movement, constant, floating-point, branch, and miscellaneous. The floating-point instructions are not executed directly, but are emulated by trap handlers.

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

1.3.5 Data Formats (Chapter 3)

The Am29200 microprocessor defines a word as 32 bits of data, a half-word as 16 bits, and a byte as 8 bits. The hardware provides direct support for word-integer (signed and unsigned), word-logical, word-boolean, half-word integer (signed and unsigned), and character data (signed and unsigned).

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

Other data formats, such as character strings, are supported by instruction sequences. Floating-point formats (single and double precision) are defined for the processor; however, there is no direct hardware support for these formats in the Am29200 microprocessor.

1.3.6 Protection (Chapter 6)

The Am29200 microprocessor offers two mutually exclusive modes of execution, the User and Supervisor modes, which restrict or permit accesses to certain processor registers and external storage locations.

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

1.3.7 DRAM Mapping (Section 9.2.4)

The Am29200 microprocessor provides a 16-Mb region of virtual memory that is mapped to one of four 64-Kb blocks in the physical DRAM memory. This supports system functions such as on-the-fly data compression and decompression, allowing a large data structure such as a frame buffer to be stored in a compressed format while the application software operates on a region of the structure that is decompressed. Using a mechanism that is analogous to demand paging, system software moves data between the compressed and decompressed formats in a way that is invisible to the applications software. This feature can greatly reduce the amount of memory required for printing, imaging, and graphics applications.

1.3.8 Interrupts and Traps (Chapter 16)

When an Am29200 microprocessor takes an interrupt or trap, it does not automatically save its current state information in memory. This lightweight interrupt and trap facility greatly improves the performance of temporary interruptions such as simple operating-system calls which require no saving of state information.

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

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

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

1.4 DEBUGGING AND TESTING (Chapter 17)

The Am29200 microprocessor provides debugging and testing features at both the software and hardware levels.

Software debugging is facilitated by the instruction trace facility and instruction breakpoints. Instruction tracing is accomplished by forcing the processor to trap after each instruction has been executed. Instruction breakpoints are implemented by the HALT instruction or by a software trap.

The processor provides two additional features to assist system debugging and testing. The first feature, the Test/Development Interface, is composed of a group of pins that indicate the state of the processor and control the operation of the processor. The second feature is an IEEE Std. 1149.1-1990 (JTAG) compliant Standard Test Access

Port and Boundary-Scan Architecture. The Test Access Port provides a scan interface for testing system hardware in a production environment, and contains extensions that allow a hardware-development system to control and observe the processor without interposing hardware between the processor and system.

CHAPTER 2

PROGRAMMING

This chapter focuses on programming the Am29200 microprocessor. First, this chapter presents an instruction set overview. It then describes the register model, emphasizing the general- and special-purpose registers. This chapter also describes certain special-purpose registers that deal directly with instruction execution. Finally, this chapter describes general considerations related to applications programming.

2.1 INSTRUCTION SET

The Am29200 microprocessor recognizes 117 instructions. All instructions execute in a single cycle, except for IRET, IRETINV, LOADM, STOREM, and certain arithmetic instructions such as floating-point instructions. Floating-point and integer multiply and divide instructions are not implemented directly in hardware, but are implemented by a virtual arithmetic interface invoked using instruction traps (see Section 2.8).

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

This section describes the nine instruction classes in the Am29200 microprocessor, and provides a brief summary of instruction operations. A detailed instruction specification is contained in Chapter 18. Section 19.1 describes the nomenclature used here.

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

2.1.1 Integer Arithmetic

The Integer Arithmetic instructions perform add, subtract, multiply, and divide operations on word-length integers. Certain instructions in this class cause traps if signed or unsigned overflow occurs during the execution of the instruction. There is support for multiprecision arithmetic on operands whose lengths are multiples of words. All instructions in this class set the ALU Status Register. The integer arithmetic instructions are shown in Table 2-1. In the Am29200 implementation, the integer multiply and divide instructions cause traps to routines which perform the floating-point operations.

2.1.2 Compare

The Compare instructions (Table 2-2) 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 17.2).

Table 2-1 Integer Arithmetic Instructions

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

Table 2-2 Compare Instructions

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

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

2.1.3 Logical

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

2.1.4 Shift

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

2.1.5 Data Movement

The Data Movement instructions (Table 2-5) move bytes, half-words, and words between processor registers. In addition, they move data between general-purpose registers and external devices, and memories. The instructions LOADL and STOREL are provided for compatibility with other 29K processors and are treated as LOAD and STORE instructions. Similarly, the instructions MFTLB and MTTLB perform no operation, except that both are privileged instructions.

Table 2-3 Logical Instructions

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

Table 2-4 Shift Instructions

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

Table 2-5 Data Movement Instructions

Mnemonic	Operation Description
LOAD	DEST ← EXTERNAL WORD [SRCB]
LOADL	DEST ← EXTERNAL WORD [SRCB]
LOADSET	DEST ← EXTERNAL WORD [SRCB] EXTERNAL WORD [SRCB] ← h'FFFFFFFF'
LOADM	DEST.. DEST + COUNT ← EXTERNAL WORD [SRCB] .. EXTERNAL WORD [SRCB + COUNT · 4]
STORE	EXTERNAL WORD [SRCB] ← SRCA
STOREL	EXTERNAL WORD [SRCB] ← SRCA
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	no operation (privileged)
MTSR	SPDEST ← SRCB
MTSRIM	SPDEST ← 0|16
MTTLB	no operation (privileged)

2.1.6 Constant

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

Table 2-6 Constant Instructions

Mnemonic	Operation Description
CONST	DEST ← 0|16
CONSTH	Replace high-order half-word of SRCA by I16
CONSTN	DEST ← 1|16

2.1.7 Floating-Point

The Floating-Point instructions (Table 2-7) provide operations on single-precision (32-bit) or double-precision (64-bit) floating-point data. They also provide conversions between single-precision, double-precision, and integer number representations. In the Am29200 processor implementation, these instructions cause traps to routines which perform the floating-point operations.

2.1.8 Branch

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

2.1.9 Miscellaneous

The Miscellaneous instructions (Table 2-9) perform various operations that cannot be grouped into other instruction classes. In certain cases, these are control functions available only to Supervisor-mode programs. The instructions INV and IRETINV are provided for compatibility with other 29K processors. INV performs no operation, and IRETINV performs the same operations as IRET. Both are privileged instructions.

2.1.10 Reserved Instructions

Sixteen Am29200 operation codes are reserved for instruction emulation. Each of these instructions causes a trap and sets the indirect pointers IPC, IPA, and IPB. The relevant operation codes, and the corresponding trap vectors, are as follows:

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

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

2.2 REGISTER MODEL

The Am29200 microprocessor has two classes of registers that are accessible by instructions. These are the general-purpose registers and the special-purpose registers. Any operation available to the Am29200 microprocessor can be performed on the general-purpose registers, while special-purpose registers are accessed only by the instructions MTSR, MTSRIM, and MFSR. This section describes the general-purpose and special-purpose registers.

Table 2-7 Floating-Point Instructions

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

Table 2-8 Branch Instructions

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

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

Table 2-9 Miscellaneous Instructions

Mnemonic	Operation Description
CLZ	Determine number of leading zeros in a word
SETIP	Set IPA, IPB, and IPC with operand register numbers
EMULATE	Load IPA and IPB with operand register numbers, and Trap (VN)
INV	No operation
IRET	Perform an interrupt return sequence
IRETINV	Perform an interrupt return sequence
HALT	Enter Halt mode

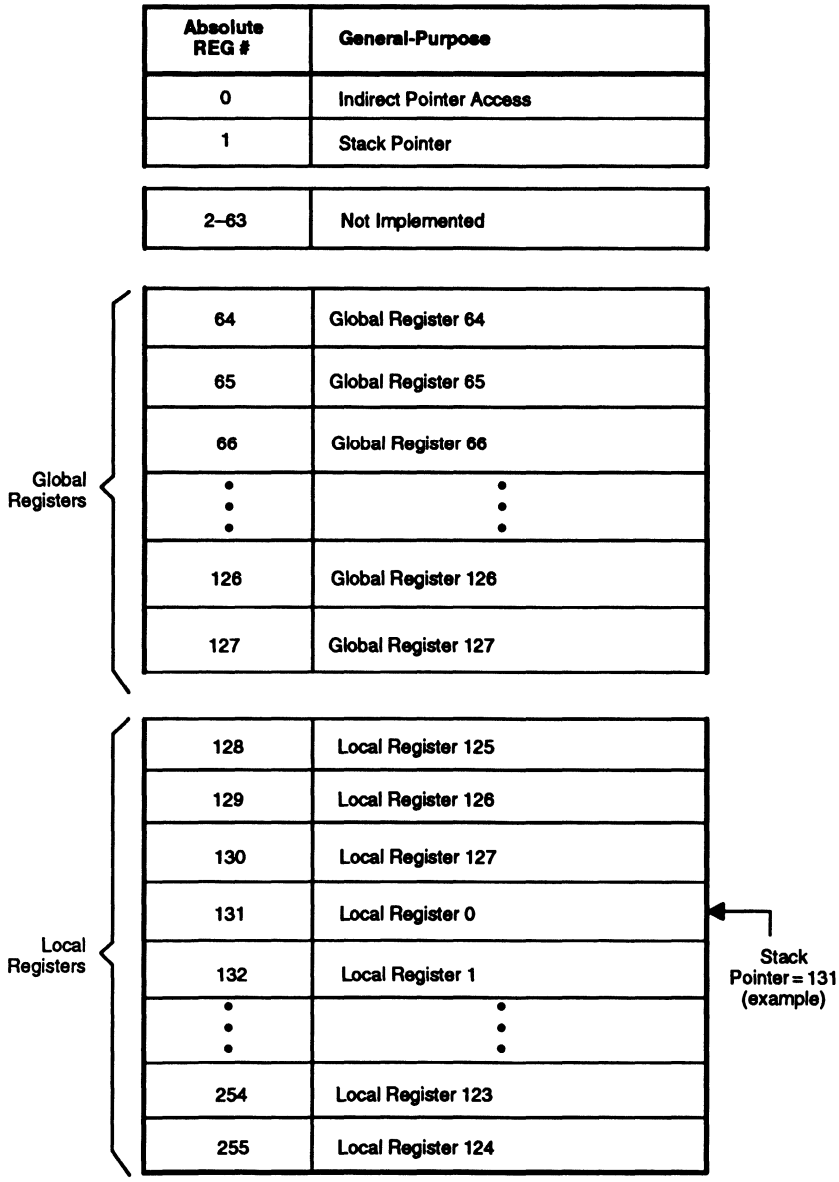
2.2.1 General-Purpose Registers

The Am29200 microprocessor incorporates 192 general-purpose registers. The organization of the general-purpose registers is diagrammed in Figure 2-1.

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

1. 32-bit addresses
2. 32-bit signed or unsigned integers

Figure 2-1 General-Purpose Register Organization



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.

Am29200 instructions can specify two general-purpose registers for source operands, and one general-purpose register for storing the instruction result. These registers are specified by three 8-bit instruction fields containing register numbers. A register may be specified directly by the instruction, or indirectly by one of three special-purpose registers.

2.2.1.1 REGISTER ADDRESSING

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

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

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

2.2.1.2 GLOBAL REGISTERS

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

Global registers 2 through 63 are not implemented. An attempt to access these registers yields unpredictable results; however, they may be protected from User-mode access by the Register Bank Protect Register (see Section 7.2.1).

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

2.2.1.3 LOCAL REGISTERS

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

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

2.2.1.4 LOCAL-REGISTER STACK POINTER

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

A modification of the Stack Pointer has a delayed effect on the addressing of local registers, as discussed in Section 6.6.

2.2.2 Special-Purpose Registers

The Am29200 microprocessor contains 24 special-purpose registers. The organization of the special-purpose registers is shown in Figure 2-2.

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

Some special-purpose registers have fields 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, except for bits 5 and 6 of the Current Processor Status Register. These bits are used to disable address translation in other 29K processors and may be written with 1 in the Am29200 microprocessor.

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

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

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

Figure 2-2 Special-Purpose Registers

Register Number	Protected Registers	Mnemonic
0	Vector Area Base Address	VAB
1	Old Processor Status	OPS
2	Current Processor Status	CPS
3	Configuration	CFG
4	Channel Address	CHA
5	Channel Data	CHD
6	Channel Control	CHC
7	Register Bank Protect	RBP
8	Timer Counter	TMC
9	Timer Reload	TMR
10	Program Counter 0	PC0
11	Program Counter 1	PC1
12	Program Counter 2	PC2
	Unprotected Registers	
128	Indirect Pointer C	IPC
129	Indirect Pointer A	IPA
130	Indirect Pointer B	IPB
131	Q	Q
132	ALU Status	ALU
133	Byte Pointer	BP
134	Funnel Shift Count	FC
135	Load/Store Count Remaining	CR
•		•
•		•
•		•
160	Floating-Point Environment (virtual)	FPE
161	Integer Environment (virtual)	INTE
162	Floating-Point Status (virtual)	FPS

numbers are reserved for virtual registers in the arithmetic architecture, and any attempted access causes a Protection Violation trap.

The Floating-Point Environment Register, Integer Environment Register, and Floating-Point Status Register are not implemented in processor hardware. These registers are implemented via the virtual arithmetic interface provided on the Am29200 microprocessor (see Section 2.8).

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

2.3 ADDRESSING REGISTERS INDIRECTLY

Specifying Global Register 0 as an instruction operand register or result register causes an indirect access to the general-purpose registers. In this case, the

absolute-register number is provided by an indirect pointer contained in a special-purpose register.

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

The indirect pointers are set by the MTSR, MTSRIM, SETIP, and EMULATE instructions and by floating-point, MULTIPLY, MULTM, MULTIPLU, MULTMU, DIVIDE, and DIVIDU instructions.

For a move-to-special-register instruction, an indirect pointer is set with bits 9–2 of the 32-bit source operand. This provides consistency between the addressing of words in general-purpose registers and the addressing of words in external devices or memories. A modification of an indirect pointer using a move-to-special-register instruction has a delayed effect on the addressing of general-purpose registers, as discussed in Section 6.6.

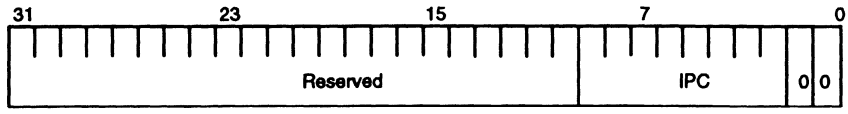
For the remaining instructions, all three indirect pointers are set simultaneously with the absolute-register numbers derived from the register numbers specified by the instruction. For any local registers selected by the instruction, the Stack-Pointer addition is applied to the register numbers before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

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

2.3.1 Indirect Pointer C (IPC, Register 128)

This unprotected special-purpose register (Figure 2-3) provides the RC-operand register number (see Section 19.3) when an instruction RC field has the value zero (i.e., when Global Register 0 is specified).

Figure 2-3 Indirect Pointer C Register



Bits 31–10: Reserved.

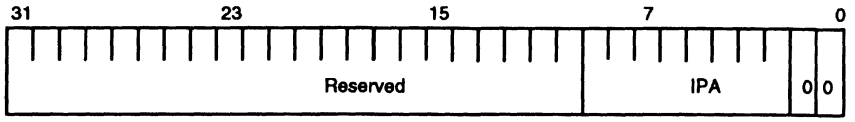
Bits 9–2: Indirect Pointer C (IPC)—The 8-bit IPC field contains an absolute-register number for a general-purpose register. This number directly selects a register (Stack-Pointer addition is not performed in the case of local registers).

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

2.3.2 Indirect Pointer A (IPA, Register 129)

This unprotected special-purpose register (Figure 2-4) provides the RA-operand register number (see Section 19.3) when an instruction RA field has the value zero (i.e., when Global Register 0 is specified).

Figure 2-4 **Indirect Pointer A Register**



Bits 31–10: Reserved.

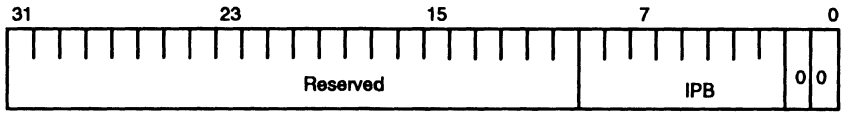
Bits 9–2: Indirect Pointer A (IPA)—The 8-bit IPA field contains an absolute-register number for either a general-purpose register or a local register. This number directly selects a register (Stack-Pointer addition is not performed in the case of local registers).

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

2.3.3 Indirect Pointer B (IPB, Register 130)

This unprotected special-purpose register (Figure 2-5) provides the RB-operand register number (see Section 19.3) when an instruction RB field has the value zero (i.e., when Global Register 0 is specified).

Figure 2-5 **Indirect Pointer B Register**



Bits 31–10: Reserved.

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

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

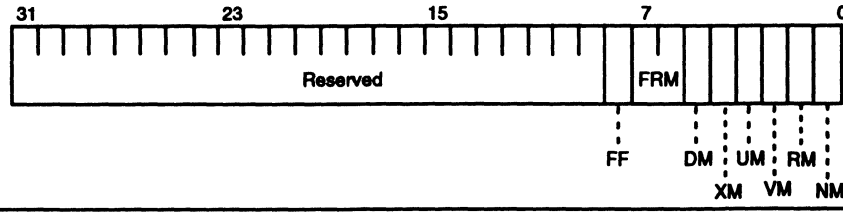
2.4 INSTRUCTION ENVIRONMENT

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

2.4.1 Floating-Point Environment (FPE, Register 160)

This unprotected special-purpose register (Figure 2-6) contains control bits that affect the execution of floating-point operations. This register is not implemented directly by processor hardware, but is implemented by the virtual arithmetic interface.

Figure 2-6 **Floating-Point Environment Register**



Bits 31–9: Reserved.

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

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

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

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

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

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

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

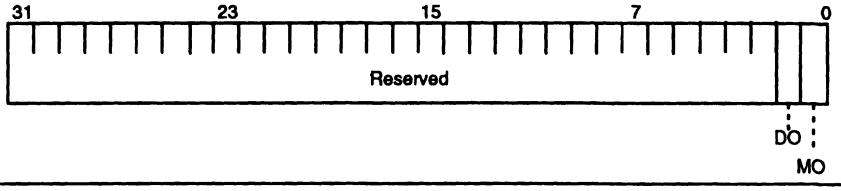
Bit 1: Floating-Point Reserved Operand Mask (RM)—If the RM bit is 0, a Floating-Point Exception trap occurs when one or more input operands to a floating-point operation is a reserved value, or when the result of a floating-point operation is a reserved value. If the RM bit is 1, a Floating-Point Exception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

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

2.4.2 Integer Environment (INTE, Register 161)

This unprotected special-purpose register (Figure 2-7) contains control bits which affect the execution of integer multiplication and division operations. This register is not implemented directly by processor hardware, but is implemented by the virtual arithmetic interface.

Figure 2-7 Integer Environment Register



Bits 31–2: Reserved.

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

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

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

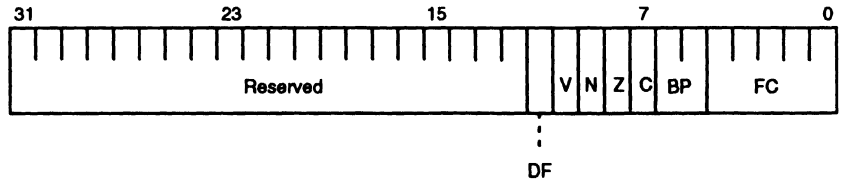
2.5 STATUS RESULTS OF INSTRUCTIONS

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

2.5.1 ALU Status (ALU, Register 132)

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

Figure 2-8 ALU Status Register



Bits 31–12: Reserved.

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

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

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

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

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

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

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

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

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

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

2.5.2 Arithmetic Operation Status Results

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

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

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

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

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

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

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

2.5.3 Logical Operation Status Results

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

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

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

2.5.4 Floating-Point Status Results

The floating-point instructions check for a number of exceptional conditions, and report these exceptions by setting bits of the Floating-Point Status Register. The exceptional conditions may also 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 virtual arithmetic processor on the Am29200 microprocessor 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.

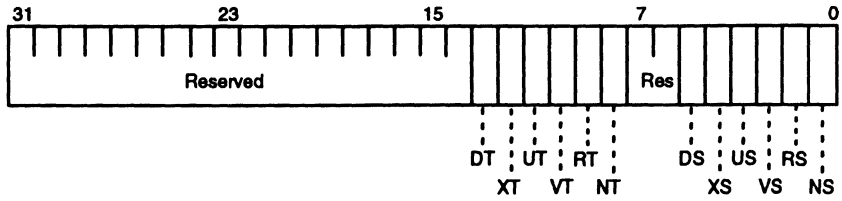
2.5.5 Floating-Point Status (FPS, Register 162)

This unprotected special-purpose register (Figure 2-9) contains status bits indicating the outcome of floating-point operations. This register is not implemented directly by processor hardware, but is implemented by the virtual arithmetic interface.

The floating-point status bits are divided into two groups. The first group consists of the sticky status bits (DS, XS, US, VS, RS, and NS), which, once set, remain set until explicitly cleared by a Move-to-Special-Register (MTSR) or Move-to-Special-Register-Immediate (MTSRIM) instruction. Only those sticky status bits corresponding to masked exceptions are updated. The update occurs at the end of instruction execution.

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

Figure 2-9 **Floating-Point Status**



Bits 31–14: Reserved.

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

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

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

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

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

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

Bits 7–6: Reserved.

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

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

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

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

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

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

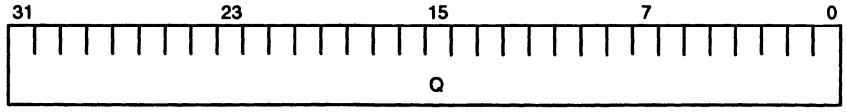
2.6 INTEGER MULTIPLICATION AND DIVISION

The Am29200 microprocessor does not directly support the instructions MULTIPLU, MULTMU, MULTIPLY, MULTM, DIVIDE, and DIVIDU. The processor is capable of performing these instructions as a sequence of multiply- or divide-steps, which are directly supported by hardware. A special register, Q, is used in conjunction with the SRCA and SRCB operands to execute the multiply- or divide-step. This section describes the Q register and discusses the general method for multiplication and division.

2.6.1 Q (Q, Register 131)

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

Figure 2-10 Q Register



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

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

2.6.2 Multiplication

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

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

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

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

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

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

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

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


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

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

```
; next 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 it jumps, Temp1 holds 0, so this copies
                                        ; the value of the IPB register

        const   Temp0,0                 ; swap the operands
        or      Temp0,Temp0,gr0
        or      Temp1,gr0,0

do8:
        cpleu   Temp2,Temp1,0x7f        ; less than 8 bits?
        jmpf    Temp2,do16              ; no, check for 16 bits
        mtsr    Q,Temp0
        mulu    Temp0,Temp1,0

        .rep    7                       ; expand out 7 copies of the next instruction
                                        ; in-line
        mulu    Temp0,Temp1,Temp0       ; total of 7 more multiply steps
        .endr

; the top 24 bits of the result are in the lower 24 bits of Temp0, and the bottom 8 bits are in the
; top of Q
        mfsr    Temp1,Q
        jmpi    tpc                     ; return to the calling routine
        extract gr0,Temp0,Temp1         ; extract the result in the delay-slot of the
                                        ; jump

do16:
        const   Temp2,0x7fff            ; less than 16 bits?
        cplequ  Temp2,Temp0,Temp2
        jmpf    Temp2,do32              ; no, perform all 32 steps
        mulu    Temp0,Temp1,0           ; perform initial multiply-step

        .rep    15                      ; expand out 15 copies of next instruction
                                        ; in-line
        mulu    Temp0,Temp1,Temp0       ; total of 15 more multiply-steps
        .endr

; the top 16 bits of the result will be in the 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 the calling routine
        extract gr0,Temp0,Temp1         ; extracting the result in the delay-slot of the
                                        ; jump

do32:
        mulu    temp0,Temp1,0           ; perform initial step

        .rep    31                      ; expand out 32 copies of the next instruction
                                        ; in-line
        mulu    Temp0,Temp1,Temp0       ; total of 31 more multiply steps
        .endr

        jmpi    tpc                     ; return to calling routine
        mfsr    gr0,Q                   ; copy the result to the return register in the
                                        ; delay slot
```

2.6.3 Division

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

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

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

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

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

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

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

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

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

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

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

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

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

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

;           call    tpc, SDiv32                 ; call the divide routine
;           setip   dst_reg, src1_reg, src2_reg
                                                ; passing pointers to the operand
                                                ; registers in the delay slot
; Input:    dividend and divisor in the registers pointed to by the indirect-pointer
;           registers IPA and IPB
; Output:   result quotient in the register pointed to by IPC, remainder left in Temp0
; Used:     return address in tpc, special register Q
; Destroyed: previous contents of registers tpc, Temp0-Temp2
; Symbolic register names:
            .reg    Temp0, gr116
            .reg    Temp1, gr119
            .reg    Temp2, gr120
            .reg    tpc, gr122
            .word   0x00200000              ; Debugger tag word
SDiv32:
            const   Temp1, 0
            asneq   V_DIVBYZERO, Temp1, gr0
                                            ; check for divide by zero with an assert
            add     Temp0, gr0, 0           ; get dividend from indirect pointer
            jmpf    Temp0, pdividend        ; is it negative? (jmpf is also "jmppos")
            add     Temp2, Temp1, gr0       ; get divisor from indirect pointer
            const   Temp1, 3                ; set negative result and remainder flags
            subr    Temp0, Temp0, 0         ; make dividend positive
pdividend:
            jmpf    Temp2, pdivisor         ; is divisor negative?
            mtsr    Q, Temp0                ; copy dividend to Q register in delay slot
                                            ; of the jump
            xor     Temp1, Temp1, 1         ; turn off negative result flag
            subr    Temp2, Temp2, 0         ; make divisor positive
pdivisor:
            div0    Temp0, 0                ; initialize

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

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

2.7 I NEED AN INSTRUCTION TO...

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

2.7.1 Run-Time Checking

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

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

The assert instructions specify a vector number for the trap. However, only vector numbers 64 through 255 (inclusive) may be specified by User-mode programs. If a User-mode assert instruction causes a trap, and the vector number is between 0 and 63 inclusive, a Protection Violation trap occurs, instead of the specified trap.

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

2.7.2 Operating-System Calls

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

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

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

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

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

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

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

2.7.3 Multiprecision Integer Operations

The processor allows the Carry (C) bit of the ALU Status Register to be used as an operand for add and subtract instructions. This provides for the addition and

subtraction of operands which are greater than 32 bits in length. For example, the following code implements a 96-bit addition with signed overflow detection.

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

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

2.7.4 Complementing a Boolean

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

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

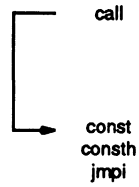
The Boolean is in GR96 in this example. This instruction is based on the observation that a Boolean TRUE is a negative integer, since the Boolean bit coincides with the integer sign bit. If the operand of this instruction is a negative integer (i.e., TRUE), the result is the Boolean FALSE. If the operand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

2.7.5 Large Jump and Call Ranges

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

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

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



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

2.7.6 NO-OPs

When a NO-OP is required for proper operation (e.g., as described in Section 6.6), it is important that the selected instruction not perform any operation, regardless of program operating conditions. For example, the NO-OP cannot access general-purpose registers because a register may be protected from access in some situations. The suggested NO-OP is:

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

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

2.8 VIRTUAL ARITHMETIC PROCESSOR

In order to be object-code compatible with present and future implementations of the 29K Family of microprocessors, the Am29200 microprocessor provides a virtual arithmetic interface. A virtual interface is the means by which a processor appears to perform functions that it does not actually perform. In the case of the Am29200 processor's virtual arithmetic interface, the processor defines arithmetic instructions, control, and status which are not directly supported by hardware, but which are implemented by system software.

2.8.1 Trapping Arithmetic Instructions

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

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

In the Am29200 microprocessor, the Floating-Point, CLASS, CONVERT, MULTIPLY, MULTM, MULTIPLU, MULTMU, DIVIDE, DIVIDU, and SQRT instructions cause traps. The indirect pointers are set at the time the trap occurs, so a trap handler can gain access to the operands of the instruction and can determine where the result is to be stored. A trap handler can directly emulate the execution of the instruction.

2.8.2 Virtual Registers

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

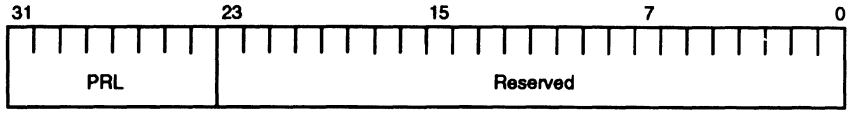
2.9 PROCESSOR INITIALIZATION

When power is first applied to the processor, it is in an indeterminate state and must be placed in a known state. Also, under certain circumstances, it may be necessary to place the processor in a defined state. This is accomplished by the Reset mode, which places the processor into a predefined state.

2.9.1 Configuration (CFG, Register 3)

This protected special-purpose register (Figure 2-11) controls certain processor and system options. The Configuration Register is defined as follows:

Figure 2-11 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 23–0: Reserved.

2.9.2 Reset Mode

The Reset mode is invoked by asserting the $\overline{\text{RESET}}$ input. The Reset mode is entered within four processor cycles after $\overline{\text{RESET}}$ is asserted. The $\overline{\text{RESET}}$ input must be asserted for at least four processor cycles to accomplish a processor reset.

The Reset mode can be entered at any point during operation. If the $\overline{\text{RESET}}$ input is asserted at the time power is first applied to the processor, the processor enters the Reset mode only after four cycles have occurred on the MEMCLK pin.

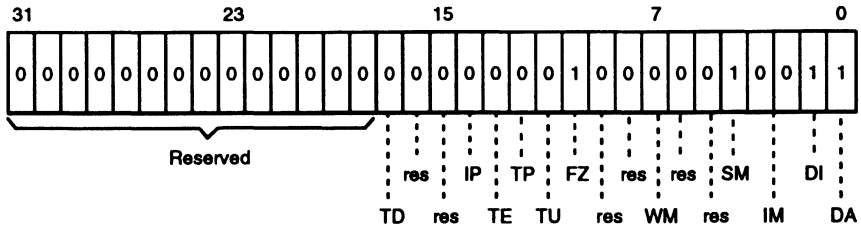
The Reset mode configures the processor state as follows:

1. Instruction execution is suspended.
2. Instruction fetching is suspended.
3. Any interrupt or trap conditions are ignored.
4. The Current Processor Status Register is set as shown in Figure 2-12.
5. The Contents Valid (CV) bit of the Channel Control Register is reset.

Except as previously noted, the contents of all general-purpose registers and special-purpose registers are undefined.

The Reset mode is exited when the $\overline{\text{RESET}}$ input is de-asserted. Either four or five cycles after $\overline{\text{RESET}}$ is de-asserted (depending on internal synchronization time), the processor performs an initial instruction access on the external interface. The initial instruction access is directed to address 0, which is in ROM Bank 0 after a reset. The characteristics of the ROM in Bank 0 are set by the BOOTW signal during reset (see Section 9.1.3).

Figure 2-12 Current Processor Status Register In Reset Mode



A processor reset configures the internal peripherals as follows:

1. In the ROM Controller, ROM Bank 0 is configured by the BOOTW signal and the other banks are set so as not to interfere with accesses to ROM Bank 0.
2. The DRAM configuration is not set by a processor reset, DRAM mapping is disabled, and the refresh rate is set to the slowest possible value (refresh every 511 MEMCLK cycles).
3. The configuration of the Peripheral Interface Adapter is not set by a processor reset.
4. The DMA Controller is disabled, and all state machines are reset.
5. All I/O Port signals are disabled as outputs.
6. The Parallel Port is disabled, and all state machines are reset.
7. The Serial Port is disabled, and all state machines are reset.
8. The Video Interface is disabled, and all state machines are reset. All signals that may be either inputs or outputs are configured as inputs.

CHAPTER 3

DATA FORMATS AND HANDLING

This section describes the various data types supported by the Am29200 microprocessor and the mechanisms for accessing data in external devices and memories. The Am29200 microprocessor includes provisions for the external access of words, bytes, half-words, unaligned words, and unaligned half-words, as described in this section.

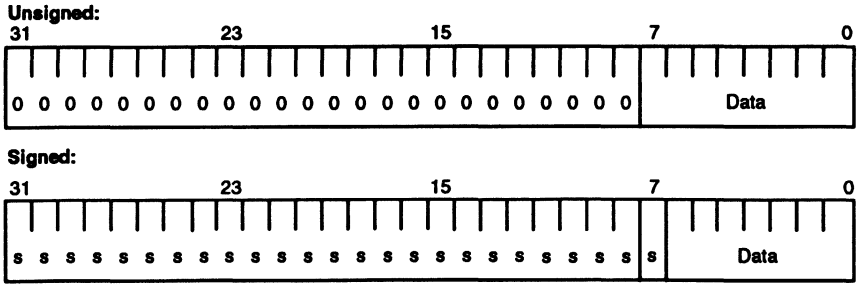
3.1 INTEGER DATA TYPES

Most 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.

3.1.1 Character Data

The processor supports character data through load, store, extraction, and insertion operations, and by a compare operation on byte-length fields within words. The format of unsigned and signed characters is shown in Figure 3-1; for signed characters, the sign bit is the most significant bit of the character. For sequences of packed characters within words, bytes are ordered left-to-right (that is, "big-endian").

Figure 3-1 Character Format



On a byte load, an external packed byte is converted to one of the character formats shown in Figure 3-1. On a byte store, the low-order byte of a word is packed into a selected byte of an external word.

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.

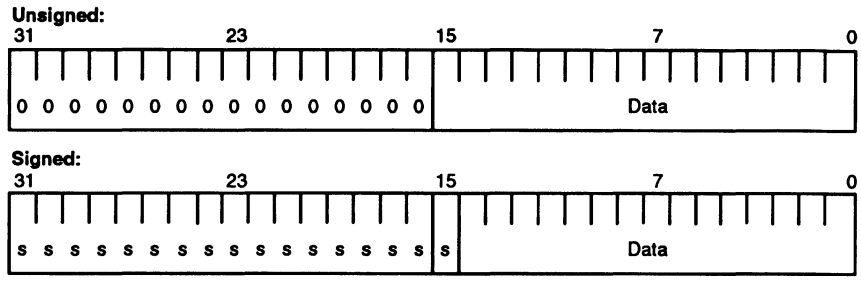
The Insert Byte (INBYTE) instruction replaces an arbitrary byte-aligned character in a destination word with the low-order character of a source word. For the INBYTE instruction, the source operand can be a character constant specified by the instruction.

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.

3.1.2 Half-Word Operations

The processor supports half-word data through load, store, insertion, and extraction operations. The format of unsigned and signed half-words is shown in Figure 3-2. 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 left-to-right (that is, "big-endian").

Figure 3-2 Half-Word Format



On a half-word load, an external packed half-word is converted to one of the formats shown in Figure 3-2. On a half-word store, the low-order half-word of a word is packed into a selected half-word of an external word.

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.

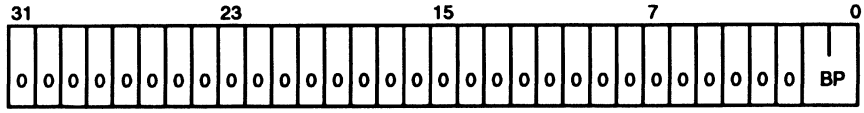
The Extract Half-Word, Sign-Extended (EXHWS) instruction is similar to the EXHW instruction, except that it sign-extends the half-word in the destination word (i.e., it replaces the most significant 16 bits of the destination word with the most significant bit of the source half-word).

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.

3.1.3 Byte Pointer (BP, Register 133)

This unprotected special-purpose register (Figure 3-3) provides an alternate access to the BP field in the ALU Status Register (see Section 2.5.1). For the Extract Byte (EXBYTE) and Insert Byte (INBYTE) instructions, the character is selected via the Byte Pointer field. For the Extract Half-Word (EXHW), Extract Half-Word Signed (EXHWS), and Insert Half-Word (INHW) instructions, the half-word is selected by the most significant bit of the Byte Pointer field.

Figure 3-3 **Byte Pointer Register**



Bits 31–2: Zeros.

Bits 1–0: 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 most significant bit of the BP field is used to determine the position of a half-word within a word for the following three instructions; Insert Half-Word, Extract Half-Word, and Extract Half-Word Sign-Extended instructions.

The BP field is set by a Move To Special Register instruction with either the ALU Status Register or the Byte Pointer Register as the destination. It is also set by a load or store instruction if the Set Byte Pointer (SB) bit in the instruction is 1. A load or store sets the BP field with 11.

This field allows a program to change the BP field without affecting other fields in the ALU Status Register.

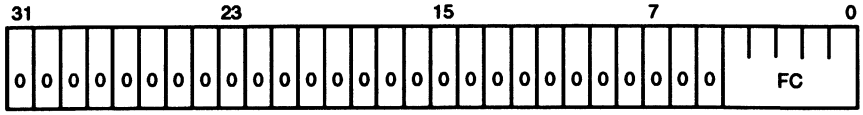
3.1.4 Bit Strings

Graphics and imaging applications often require that a data region be collectively shifted by a specific number of bits. The Am29200 microprocessor supports such an operation through the Extract (EXTRACT) instruction. The Extract instruction concatenates two 32-bit values, producing a 64-bit source operand, and then shifts this value left by an arbitrary number of bits to produce a 32-bit result. The shift amount is determined by the value in the Funnel Shift Count Register. The Funnel Shift Count Register is set before executing the Extract instruction.

3.1.4.1 FUNNEL SHIFT COUNT (FC, Register 134)

This unprotected special-purpose register (Figure 3-4) provides an alternate access to the FC field in the ALU Status Register.

Figure 3-4 **Funnel Shift Count Register**



Bits 31–5: Zeros.

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.

This field allows a program to change the FC field without affecting other fields in the ALU Status Register.

3.1.5 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.

In general, it is much more efficient 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, as outlined in this section, there are several things to be considered when dealing with four-byte units.

3.1.5.1 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 is easily 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.

3.1.5.2 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 the corresponding byte in the search key.

3.1.6 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.

3.1.7 Instruction 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 set to zero. 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 can 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 not modified. 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 set to one. 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.

3.2 FLOATING-POINT DATA TYPES

The Am29200 microprocessor 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 directly supported in processor hardware, but can be implemented using the virtual arithmetic interface provided on the Am29200 microprocessor.

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

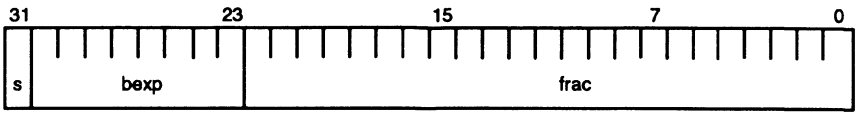
3.2.1 Single-Precision Floating-Point Values

The format for a single-precision floating-point value is shown in Figure 3-5. Typically, the value of a single-precision operand is expressed by:

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

The encoding of special floating-point values is given in Section 3.2.3.

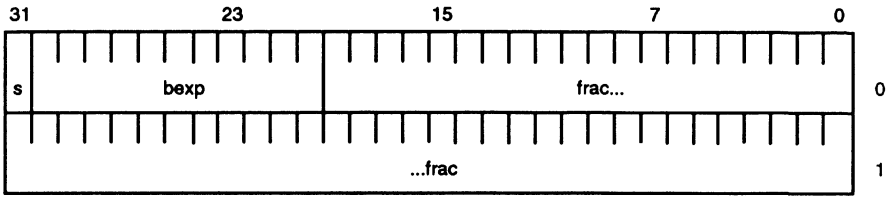
Figure 3-5 **Single-Precision Floating-Point Format**



3.2.2 Double-Precision Floating-Point Values

The format for a double-precision floating-point value is shown in Figure 3-6.

Figure 3-6 **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 in Section 3.2.3.

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 3-6) must be even. The absolute-register number of the register containing the second word (labeled 1 in Figure 3-6) 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.

3.2.3 Special Floating-Point Values

The Am29200 microprocessor defines floating-point values encoded for special interpretation. The values are described in this section.

3.2.3.1 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 (SNaNs) and quiet NaNs (QNaNs). A SNaN causes an Invalid Operation exception if used as an input operand to a floating-point operation; a QNaN does not cause an exception. The Am29200 microprocessor distinguishes SNaNs and QNaNs by the most significant bit of the fraction: a 1 indicates a QNaN and a 0 indicates a SNaN.

An operation never generates a SNaN as a result. A QNaN 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 SNaNs or QNaNs.

In either case, the Am29200 microprocessor produces a QNaN 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.

3.2.3.2 INFINITY

Infinity is an encoded value used to represent a value 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 plus infinity ($+\infty$) from minus infinity ($-\infty$).

3.2.3.3 DENORMALIZED NUMBERS

The IEEE Standard specifies that, wherever possible, a result 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 is 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:

$$(-1)^{**}s * 0.\text{frac} * 2^{**}(-\text{bias}+1),$$

where *bias* is the exponent bias for the format in question (127 for single precision, 1023 for double precision).

3.2.3.4 ZERO

A zero is 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.

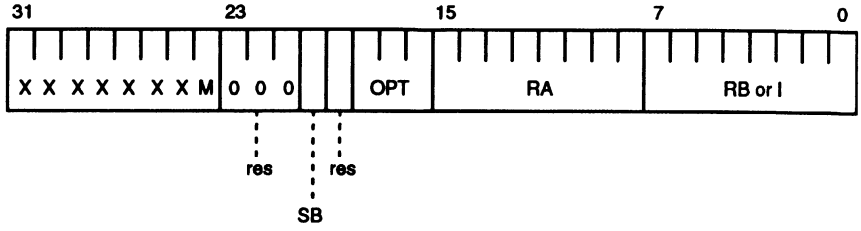
3.3 EXTERNAL DATA ACCESSES

This section discusses external data accesses supported by load and store operations on the Am29200 microprocessor.

3.3.1 Load/Store Instruction Format

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 3-7.

Figure 3-7 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 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 an immediate 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:

Bits 31–24: Opcode.

Bits 23-21: Reserved.

Bit 20: Set Byte Pointer/Sign Bit (SB)—If the SB bit is 1 for a load, the loaded byte or half-word is sign-extended in the destination register; if the SB bit is 0, the byte or half-word is zero-extended. When the SB bit is 1 for either a load or store, the Byte Pointer Register is written with 11. The Byte Pointer Register is set in this case to provide software compatibility across different types of memory systems and 29K Family processors. If the SB bit is 0, the Byte Pointer Register is not affected.

Bit 19: Reserved.

Bits 18–16: Option (OPT)—This field indicates the width of the data access and controls certain system functions, as follows:

OPT Value	Access Width or Type
000	32-bit (word) access
001	8-bit (byte) access
010	16-bit (half-word) access
110	Hardware-Development System access
—all others—	Reserved

The value OPT=110 is used by a hardware-development system to inspect and alter processor internal state. It prevents a data access from appearing externally, although the access does appear at the boundary-scan interface (see Section 18.6.4).

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

Bits 7–0: (RB or I)—In load and store instructions, the RB or I field specifies the address for the access. The address is either the content of a general-purpose

register with register number RB, or a constant value I (zero-extended to 32 bits). The M bit of the operation code (bit 24) determines whether the register or the constant is used.

Load and store operations are overlapped with the execution of instructions that follow the load or store instruction. Only one load or store may be in progress on any given cycle. If a load or store instruction is encountered while another load or store operation is in progress, the processor enters the Pipeline Hold mode until the first operation completes (see Section 6.2).

3.3.2 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 a memory or a peripheral (internal or external) into one or more general-purpose registers.

The LOADL instruction in other 29K Family processors supports the implementation of device and memory interlocks in a multiprocessor configuration. In the Am29200 microprocessor, LOADL is provided for compatibility and is identical to a LOAD.

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 LOADM instruction loads a specified number of registers from sequential addresses, as explained below in Section 3.3.4.

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

3.3.3 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 a memory or a peripheral (internal or external).

The STOREL instruction in other 29K Family processors supports the implementation of device and memory interlocks in a multiprocessor configuration. In the Am29200 microprocessor, STOREL is provided for compatibility and is identical to a STORE.

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

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

3.3.4 Multiple Accesses

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

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

Before a LOADM or STOREM is executed, the CR field is set by a Move To Special Register. A LOADM or STOREM uses the most-recently written value of the CR field. If an attempt is made to alter the CR field, and the Channel Control Register contains information for an external access that has not yet completed, the processor enters the Pipeline Hold mode until the access completes. Note that since the CR is set independently of the LOADM and STOREM, the CR field may represent valid state of an interrupted program even if the Contents Valid (CV) bit of the Channel Control Register is 0 (see also Section 17.6.2).

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

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

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

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

The 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 move to the local registers, but a sequence that begins in the local registers remains in the local registers. Also, note that the local registers are addressed circularly.

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

Intermediate addresses are stored in the Channel Address Register, and register numbers are stored in the Target Register (TR) field of the Channel Control Register. For the STOREM instruction, the data for every access is stored in the Channel Data Register (this register also is set during the execution of the LOADM instruction, but

has no interpretation in this case). The CR field is updated on the completion of every access, so that it indicates the number of accesses remaining in the sequence.

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

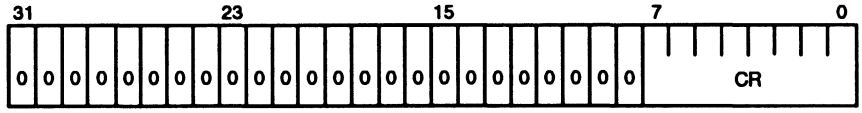
While a multiple load or store is executing, the processor is in the Pipeline Hold mode, suspending any subsequent instruction execution until the multiple access completes. If an interrupt or trap is taken, the Channel Address, Channel Data, and Channel Control registers contain the state of the multiple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

The processor performs multiple accesses using the burst-mode capability of the ROM or the page-mode capability of the DRAM, if possible. Multiple accesses of individual bytes and half-words is not supported. If the memory cannot support burst-mode accesses, a sequence of simple single accesses are performed.

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

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

Figure 3-8 Load/Store Count Remaining Register



Bits 31–8: Zeros.

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

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

3.3.4.2 MOVEMENT OF LARGE DATA BLOCKS

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

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

3.3.5 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 is as follows:

OPT Value	Access Width or Type
000	32-bit (word) access
001	8-bit (byte) access
010	16-bit (half-word) access
110	Hardware-Development System access
—all others—	Reserved

3.3.6 Addressing and Alignment

3.3.6.1 BYTE AND HALF-WORD ADDRESSING

The Am29200 microprocessor 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 permit byte selection within accessed words. For load and store operations, the processor provides for using the least significant address bits to access bytes and half-words within external words.

For all external byte and half-word accesses, the selection of a byte within an external word is determined by the two least significant bits of an address. The selection of a half-word within an external word is determined by the next-to-least significant bit of an address. Figure 3-9 illustrates the addressing of bytes and half-words. In Figure 3-9, addresses are represented in hexadecimal notation.

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

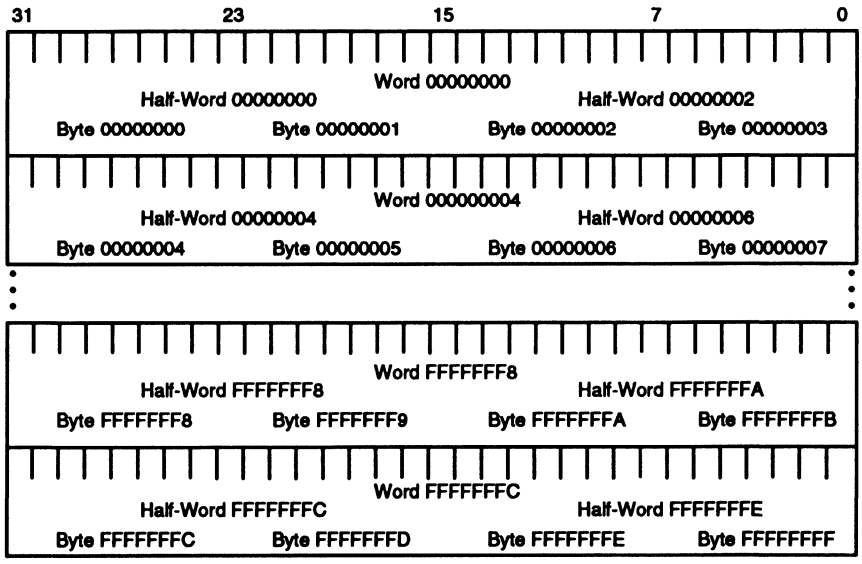
Bytes are ordered within words such that a 00 in the BP field or in the two least significant address bits selects the high-order byte of a word, and a 11 selects the low-order byte. A 00 in the BP field or in the two least significant address bits selects the low-order byte of a word, and a 11 selects the high-order byte.

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

3.3.6.2 BYTE AND HALF-WORD ACCESSES

During a load, the processor selects a byte or half-word from the loaded word depending on the Option (OPT) bits of the load instruction 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

Figure 3-9 **Byte and Half-Word Addressing (Big Endian)**



register is zero-extended. If the SB bit is 1, the remainder of the destination register is sign-extended with the sign bit of the selected byte or half-word.

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

If the SB bit is 1 for either a load or store, the BP field is set to 11 when the load or store is executed. This does not directly affect the load or store access, but supports compatibility for software developed for word-write-only systems and other 29K Family processors.

3.3.6.3 ALIGNMENT OF WORDS AND HALF-WORDS

Since byte addressing is supported, it is possible that the address for an access of a word or half-word is not aligned to the desired word or half-word. The Am29200 microprocessor 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 Am29200 microprocessor provides an option to trap when a non-aligned access is attempted. This trap allows software emulation of the non-aligned accesses, in a manner 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.

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:

OPT Value	A1	A0	Meaning
000	1	0	Unaligned Word Access
000	0	1	Unaligned Word Access
000	1	1	Unaligned Word Access
010	0	1	Unaligned Half-Word Access
010	1	1	Unaligned Half-Word Access

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

3.3.6.4 ALIGNMENT OF INSTRUCTIONS

In the Am29200 microprocessor, all instructions are 32 bits in length and are aligned on word-address boundaries. The processor's Program Counter is 30 bits in length, and the least significant two bits of processor-generated instruction addresses are always 00. An unaligned address can be generated by indirect jumps and calls. However, alignment is ignored by the processor in this case, and the processor expects the system to force alignment (i.e., by interpreting the two least significant address bits as 00, regardless of their values).

CHAPTER 4

PROCEDURE LINKAGE

This chapter describes the run-time storage organization recommended for the Am29200 microprocessor and describes the use of the local registers to improve the performance of procedure calls. The presentation in this chapter is intended to be used as a guide in the implementation of software systems for the processor, not necessarily as a strict definition of how these systems must be implemented.

Programming languages that use recursive procedures, such as C, generally use a stack to store data objects 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 Am29200 microprocessor is designed to minimize the overhead of calling a procedure, passing parameters to a procedure, and returning results from a procedure. This chapter describes the run-time storage organization and procedure-calling conventions.

4.1 RUN-TIME STACK ORGANIZATION AND USE

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

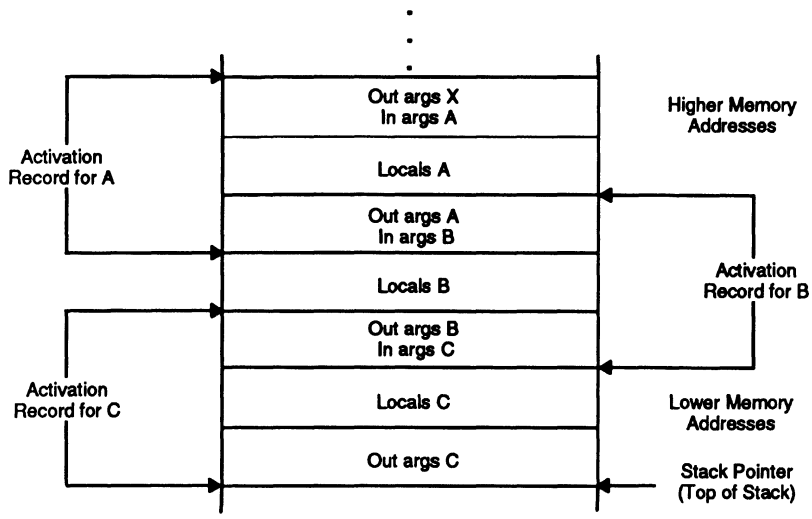
As an example, Figure 4-1 shows three activation records on a run-time stack. This stack configuration was generated by procedure A calling procedure B, which in turn called procedure C. The fact that procedure C is the currently active procedure is reflected by its activation record being on the top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

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

4.1.1 Management Of The Run-Time Stack

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

Figure 4-1 Run-Time Stack Example



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

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

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

In some cases storage is not required for one or more of the above items. Also, the incoming arguments area, though 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 subtracted during allocation.

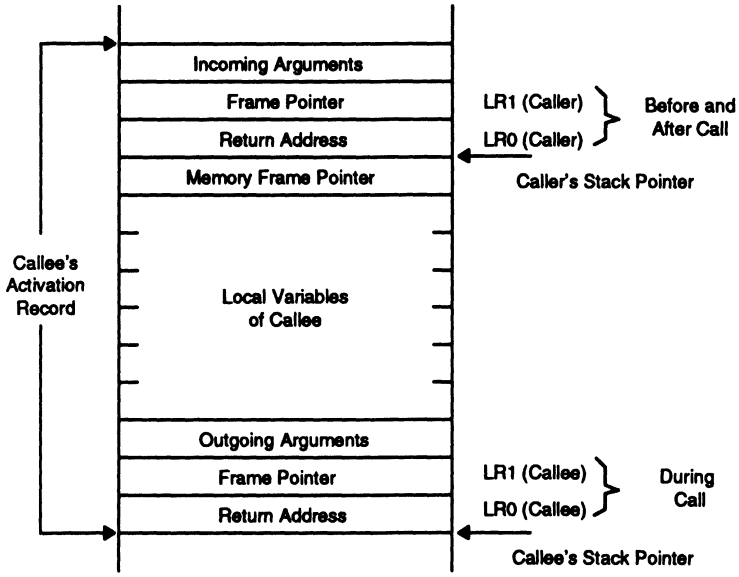
In the Am29200 microprocessor, run-time storage is actually implemented as two stacks: the Register Stack and the Memory Stack. Storage is allocated and de-allocated on these stacks at the same time. The Register Stack stores activation records associated with all active procedures (except leaf routines, as described later). The Memory Stack stores activation-record information that does not fit into the Register Stack or that must be kept in memory for other reasons (e.g., because of pointer dereferences). Both the Register Stack and the Memory Stack are stored in the external data memory. However, a portion of the Register Stack is kept in the processor's local registers for performance. The term *stack cache* in this section refers to the use of the local registers to contain a portion of the Register Stack.

4.1.2 The Register Stack

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

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

Figure 4-2 An Activation Record in the Register Stack



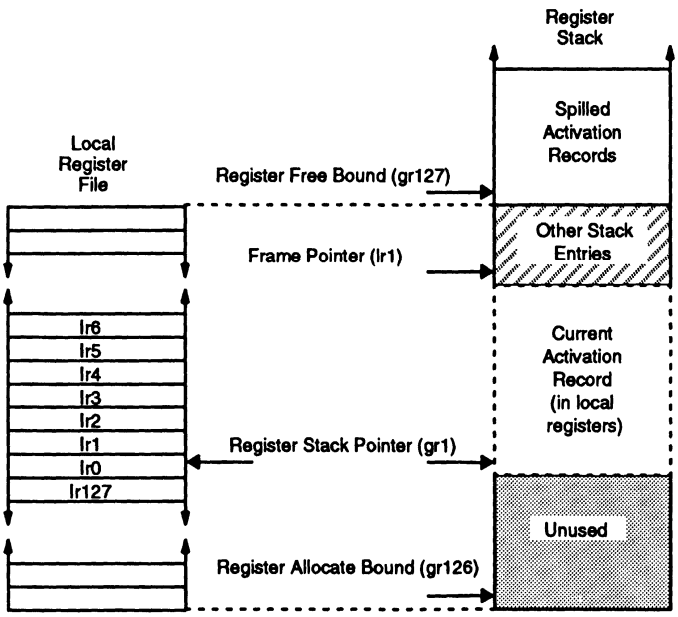
4.1.3 Local Registers As A Stack Cache

The Am29200 microprocessor is designed for efficient implementation of the Register Stack. Specifically, the Am29200 microprocessor can use the large number of relatively addressed local registers to cache portions of the Register Stack, yielding a significant gain in performance. Allocation and de-allocation of activation records occurs largely within the confines of the high-speed local registers, and most procedure calls occur without external references. Furthermore, during procedure execution, most data accesses occur without external references, because activation-record data are referenced most frequently. The principle of locality of reference, which allows any cache to be effective, also applies to the stack cache. The entries in the stack cache are likely to remain there for re-use, because the size of the Register Stack does not change very much over long intervals of program execution. Activation records are typically small, so the 128 locations in the local register file can hold many activation records.

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

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

Figure 4-3 Relationship of Stack Cache and Register Stack



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

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

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

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

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

The 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.

Figure 4-4 Stack Overflow

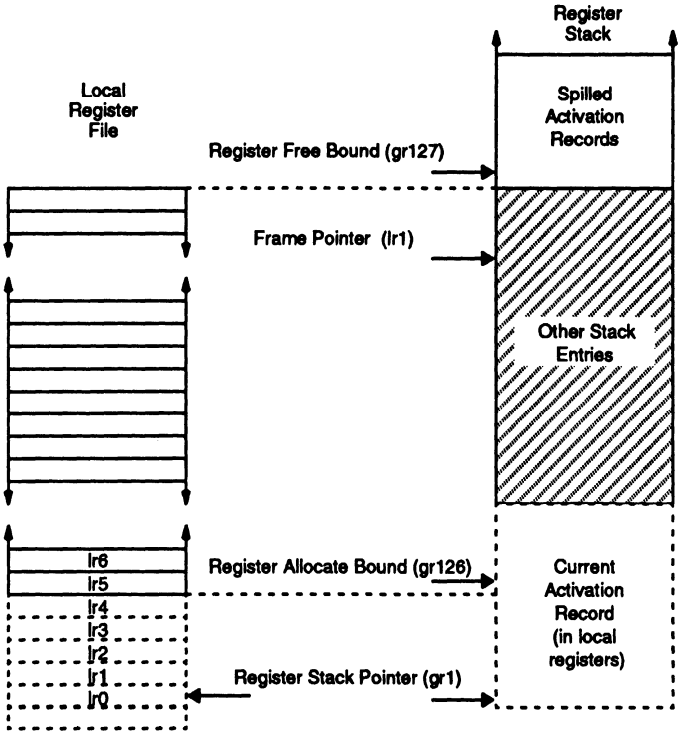
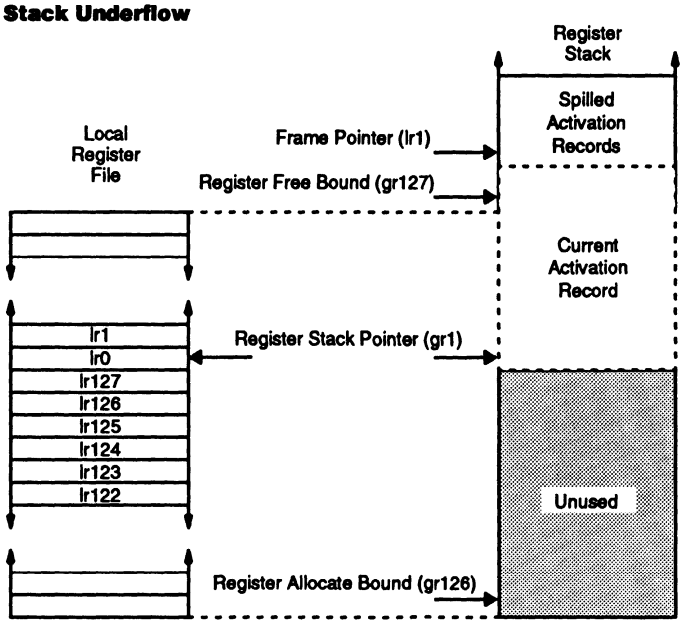


Figure 4-5 Stack Underflow



4.1.4 The Memory Stack

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

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

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

4.2 PROCEDURE LINKAGE CONVENTIONS

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

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

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

4.2.1 Argument Passing

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

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

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

No argument is aligned to anything 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). It may be necessary to copy such arguments to an aligned memory or register area before use.

4.2.2 Procedure Prologue

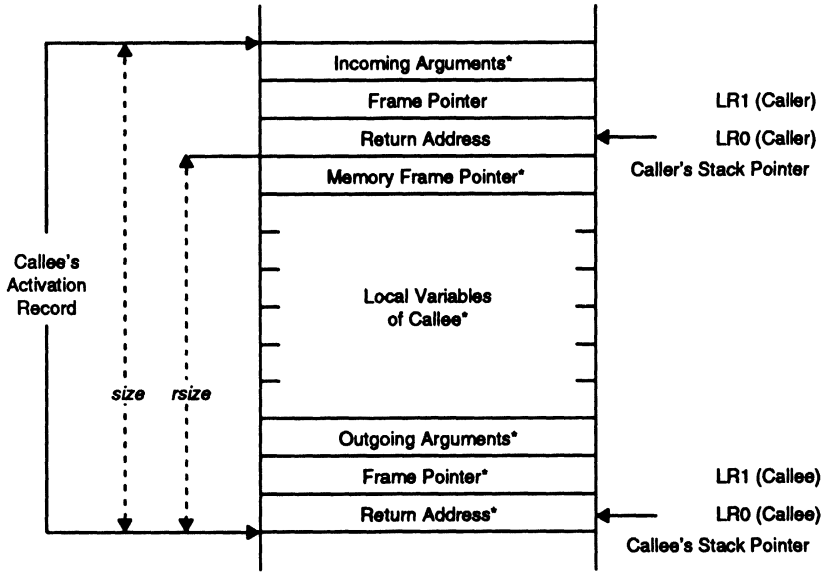
When a procedure is called, and the procedure may call another procedure, the callee must allocate a frame for itself on the Register Stack (this is not required for *leaf* procedures that do not call other procedures, as described later). A frame is allocated by decrementing the register stack pointer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

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

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

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

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



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* + (size of the incoming arguments area) + 2

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

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

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

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

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

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

4.2.3 Spill Handler

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

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

The complete spill handler is:

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

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

4.2.4 Return Values

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

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

4.2.5 Procedure Epilogue

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

The complete procedure epilogue is:

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

4.2.6 Fill Handlers

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

The complete fill handler is:

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

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

4.2.7 The Register Stack Leaf Frame

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

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

4.2.8 Local Variables And Memory-Stack Frames

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

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

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

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

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

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

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

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

4.2.9 Static Link Pointer

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

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

4.2.10 Transparent Procedures

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

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

4.3 REGISTER USAGE CONVENTION

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

- GR1: Register stack pointer (*rsp*).
- GR2–GR63: Unimplemented.
- GR64–GR95: Reserved for operating-system use.
- GR96–GR111: Procedure return values. Lower-numbered registers are used before higher-numbered registers. If more than 16 words are needed, the additional

words are stored in the Memory Stack (see GR123, large return pointer). These registers are also used for temporary values that are destroyed upon a procedure call.

- GR112–GR115: Reserved for programmer. These registers are not used by the compiler, except as directed by the programmer.
- GR116–GR120: Compiler temporaries.
- GR121: Trap handler argument/temporary (*tav*)—This register is used to communicate arguments to a software-invoked trap routine. It can be destroyed by the trap, but not by other traps and interrupts not explicitly generated by the program (for example, a Timer trap).
- GR122: Trap handler return address/temporary (*tpc*). This register is also used by software-invoked traps. It can be destroyed by the trap, but not by other traps and interrupts not explicitly generated by the program (for example, a Timer trap).
- GR123: Large return pointer/temporary (*lrp*).
- GR124: Static link pointer/temporary (*slp*).
- GR125: Memory stack pointer (*msp*).
- GR126: Register allocate bound (*rab*).
- GR127: Register free bound (*rfb*).
- LR0: Return address.
- LR1: Frame pointer.

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

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

4.4 EXAMPLE OF A COMPLEX PROCEDURE CALL

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

caller:

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

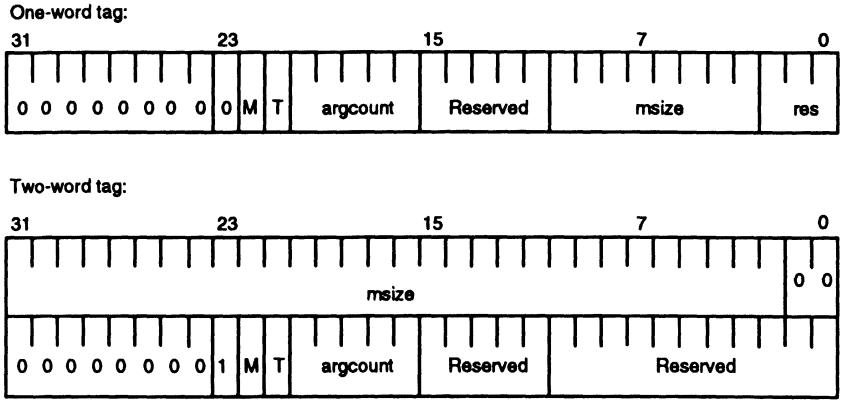
```
callee:
        const   tav, (126–2)*4              ; giant register allocation
        sub     rsp, rsp, tav               ; allocate register frame
        asgeu   V_SPILL, rsp, rab
        const   tav, (126–2)*4 + (3*4)      ; incoming arguments and overhead
        add     fp, rsp, tav      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

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

4.5 TRACE-BACK TAGS

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

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

Figure 4-7 Trace-Back Tags



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.

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

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

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

CHAPTER 5

PIPELINING AND INSTRUCTION SCHEDULING

This chapter describes the operation of the Am29200 pipeline. A description of the Am29200 pipeline is presented only to offer the reader a general overview of the internal operation of this pipeline, with the intent to aid understanding of the effects the pipeline has on program execution and on the behavior of the microprocessor under certain conditions.

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

5.1 FOUR-STAGE PIPELINE

The Am29200 microprocessor implements a four-stage pipeline for instruction execution. The four stages are fetch, decode, execute, and write-back. For operations, the pipeline is organized so the effective instruction-execution rate may be as high as one instruction per cycle.

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

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

During the execute stage, the Execution Unit performs the operation specified by the instruction.

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, an address is transmitted to a memory or a peripheral.

Most pipeline dependencies 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 Am29200 microprocessor (see Sections 5.4, 5.5, and 5.6).

5.2 PIPELINE HOLD MODE

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

The processor places itself in the Pipeline Hold mode in the following situations:

1. The processor requires an instruction that has either not been fetched or not been returned by the external instruction memory.
2. The processor requires data from an in-progress load and the operation has not completed.
3. The processor attempts to execute a load or store instruction while another load or store is in progress.
4. The processor must perform a serialization operation as described in Section 5.3.
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 $\overline{\text{WARN}}$ or $\overline{\text{RESET}}$ input is asserted.

5.3 SERIALIZATION

The Am29200 microprocessor overlaps external data references with other operations. When an external data reference might have to be restarted, however, the processor context must be the same as when the operation was first attempted. To insure this, certain operations are serialized.

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

1. An external access is not yet completed, and one of the following instructions is encountered:
   Move to Special Register (MTSR)
   Move to Special Register Immediate (MTSRIM)
   Move to TLB (MTTLB)—even though this performs no operation
   Interrupt Return (IRET)
   Interrupt Return and Invalidate (IRETINV)
   Halt (HALT)
2. An external access is not yet completed, and an interrupt or trap, other than a $\overline{\text{WARN}}$ trap, is taken.

If the processor is in the Pipeline Hold mode due to serialization, it enters the Executing mode once the external access is completed.

5.4 DELAYED BRANCH

The effect of jump and call instructions is delayed by one cycle to allow the processor pipeline to achieve maximum throughput. When one of these branches is successful, the instruction immediately following the jump or call is executed before the target

instruction of the jump or call is executed. Jump and call instructions collectively are referred to as delayed branches, and the instruction immediately following is called the delay instruction (sometimes referred to as a delay slot).

For example, in the following code fragment:

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

The sub instruction (3) is executed regardless of the outcome of the jmpf instruction (2). Of course, if the jmpf is not successful, the const instruction (4) is also executed. If the jmpf is successful, then the instruction sequence is: (2), (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 15% of all instructions are taken branches, then a processor without delayed branches would take at least two cycles for 15% of its instructions, leading to $0.85(1) + 0.15(2) = 1.15$ cycles per instruction, on average. This represents a 15% performance degradation compared to a processor with delayed branches (assuming, for this simple example, the delay instruction is always useful).

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

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

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

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

5.5 OVERLAPPED LOADS AND STORES

The Am29200 microprocessor overlaps external data references with other operations. Certain programming practices are necessary to exploit this parallelism to improve program performance.

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

```
loop:       .
            .
            sll         gr121, gr119, 2     (1)
            add         gr121, gr120, gr121 (2)
            load        0, 0, gr121, gr121  (3)
            add         gr96, gr96, gr121   (4)
            sub         gr98, gr98, 3       (5)
            add         gr119, gr119, 1     (6)
            cplt        gr122, gr119, lr2   (7)
            jmpt        gr122, loop         (8)
            nop                             (9)
            .
            .
```

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

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

```
loop:   .
        .
        sll     gr121, gr119, 2         (1)
        add     gr121, gr120, gr121     (2)
        load    0, 0, gr121, gr121      (3)
        sub     gr98, gr98, 3           (4)
        add     gr119, gr119, 1         (5)
        cplt    gr122, gr119, lr2       (6)
        jmpt    gr122, loop             (7)
        add     gr96, gr96, gr121       (8)
        .
        .
```

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

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

5.6 DELAYED EFFECTS OF REGISTERS

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

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

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

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

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 6

SYSTEM PROTECTION

The Am29200 microprocessor provides protection for general-purpose registers and special-purpose registers. Certain processor operations are also protected. This chapter describes the processor's protection mechanisms.

6.1 USER AND SUPERVISOR MODES

At any given time, the Am29200 microprocessor operates in one of two mutually exclusive program modes: the Supervisor mode or the User mode. All system-protection features of the Am29200 microprocessor are based on the difference between these two modes.

6.1.1 Supervisor Mode

The processor operates in the Supervisor mode whenever the Supervisor Mode (SM) bit of the Current Processor Status Register is 1 (see Section 16.1.1). In the Supervisor mode, executing programs have access to all processor resources.

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

6.1.2 User Mode

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

1. An attempted access of any general-purpose register for which a bit in the Register Bank Protect Register is 1 (see Section 6.2).
2. An attempted execution of one of the following instructions: Interrupt Return, Interrupt Return and Invalidate, Invalidate, or Halt. However, a hardware-development system can disable protection checking for the Halt instruction, so this instruction may be used to implement instruction breakpoints in User-mode programs (see Sections 17.2 and 17.6.5).
3. An attempted access of special-purpose register in the range of 0 to 127 or 160 to 255.
4. An attempted execution of an assert or Emulate instruction which specifies a vector number between 0 and 63, inclusive (see Section 16.2.2).

6.2 REGISTER PROTECTION

General-purpose registers are divided into register banks and are protected by the Register Bank Protection Register. The Register Bank Protection Register allows parameters for the operating system to be kept in general-purpose registers and

protected from corruption by User-mode programs. Register banks consist of 16 registers (except for Bank 0, which contains Registers 2 through 15) and are partitioned according to absolute-register numbers, as shown in Figure 6-1.

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

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

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

6.2.1 Register Bank Protect (RBP, Register 7)

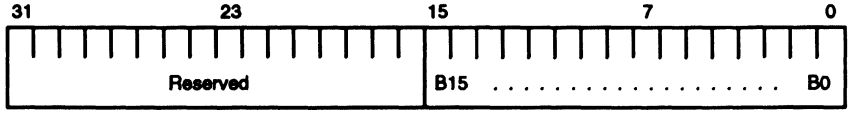
This protected special-purpose register (Figure 6-2) protects banks of general-purpose registers from User-mode program accesses.

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

Figure 6-1 Register Bank Organization

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

Figure 6-2 **Register Bank Protect Register**



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).

CHAPTER 7

SYSTEM OVERVIEW

The Am29200 microprocessor significantly reduces system cost because it integrates many system functions onto a single chip. This chapter overviews the system interfaces and on-chip peripherals of the Am29200 microprocessor.

7.1 SIGNAL DESCRIPTION

The Am29200 microprocessor uses 140 pins for signal inputs and outputs. It uses 28 pins for power and ground.

Note: If the Serial Port is not used, the UCLK signal must be tied High.

7.1.1 Clocks

INCLK — **Input Clock (input)**
This is an oscillator input at twice the processor and system operating frequency. It can be driven at TTL levels.

MEMCLK — **Memory Clock (output)**
This is a clock output at one-half of the frequency of INCLK. Most processor outputs, and many inputs, are synchronous to MEMCLK. MEMCLK is driven with CMOS levels.

7.1.2 Processor Signals

A(23–0) — **Address Bus (output, synchronous)**
The Address Bus supplies the byte address for all accesses, except for DRAM accesses. For DRAM accesses, multiplexed row and column addresses are provided on A(14–1). A(2–0) are also used to provide a clock to an optional burst-mode EPROM.

ID(31–0) — **Instruction/Data Bus (bi-directional, synchronous)**
The Instruction/Data Bus (ID Bus) transfers instructions to, and data to and from the processor.

$\overline{\text{WAIT}}$ — **Add Wait States (input, synchronous)**
External accesses are normally timed by the Am29200 microprocessor. However, the $\overline{\text{WAIT}}$ signal may be asserted during a PIA, ROM, or DMA access to extend the access indefinitely.

R/$\overline{\text{W}}$ — **Read/Write (output, synchronous)**
During an external ROM, DRAM, DMA, or PIA access, this signal indicates the direction of transfer: High for a read and Low for a write.

$\overline{\text{RESET}}$ — **Reset (input, asynchronous)**
This input places the processor in the Reset mode. This signal has special hardening against metastable states, allowing it to be driven with a slow-rise-time signal.

$\overline{\text{WARN}}$ **Warn (input, asynchronous, edge-sensitive)**
A High-to-Low transition on this input causes a non-maskable $\overline{\text{WARN}}$ trap to occur. This trap bypasses the normal trap vector fetch sequence, and is useful in situations where the vector fetch may not work (e.g., when data memory is faulty). This signal has special hardening against metastable states, allowing it to be driven with a slow-transition-time signal.

$\overline{\text{INTR}}$(3–0) **Interrupt Requests 3-0 (input, asynchronous)**
These inputs generate prioritized interrupt requests. The interrupt caused by $\overline{\text{INTR0}}$ has the highest priority, and the interrupt caused by $\overline{\text{INTR3}}$ has the lowest priority. The interrupt requests are masked in prioritized order by the Interrupt Mask field in the Current Processor Status Register and are disabled by the DA and DI bits of the Current Processor Status Register. These signals have special hardening against metastable states, allowing them to be driven with slow-transition-time signals.

$\overline{\text{TRAP}}$(1–0) **Trap Requests 1-0 (input, asynchronous)**
These inputs generate prioritized trap requests. The trap caused by $\overline{\text{TRAP0}}$ has the highest priority. These trap requests are disabled by the DA bit of the Current Processor Status Register. These signals have special hardening against metastable states, allowing them to be driven with slow-transition-time signals.

STAT(2–0) **CPU Status (output, synchronous)**
These outputs indicate information about the processor or the current access for the purposes of hardware debug. They are encoded as follows:

STAT2	STAT1	STAT0	Condition
0	0	0	Halt or Step Modes
0	0	1	Reserved
0	1	0	Load Test Instruction Mode, Halt/Freeze
0	1	1	Wait Mode
1	0	0	External data access (data valid)
1	0	1	External instruction access (instruction valid)
1	1	0	Internal data access (data valid)
1	1	1	Idle or data/instruction not valid

These signals are described in Section 17.3.

7.1.3 ROM Interface

$\overline{\text{ROMCS}}$(3–0) **ROM Chip Selects, Banks 3-0 (output, synchronous)**
A Low level on one of these signals selects the memory devices in the corresponding ROM bank. $\overline{\text{ROMCS}}$ 3 selects devices in ROM Bank 3, and so on. The timing and access parameters of each bank are individually programmable.

$\overline{\text{ROMOE}}$ **ROM Output Enable (output, synchronous)**
This signal enables the selected ROM Bank to drive the ID bus. It is used to prevent bus contention when switching between different ROM banks or switching between a ROM bank and another device or DRAM bank.

$\overline{\text{BURST}}$ **Burst-Mode Access (output, synchronous)**
This signal is asserted to perform sequential accesses from a burst-mode device such as the Am27B010 burst-mode EPROM.

$\overline{\text{RSWE}}$ **ROM Space Write Enable (output, synchronous)**
This signal is used to write an alterable memory in a ROM bank (such as an SRAM or Flash EPROM).

BOOTW **Boot ROM Width (input, asynchronous)**
This input configures the width of ROM Bank 0, so the ROM can be accessed before the ROM configuration has been set by the system initialization software. The BOOTW signal is sampled during and after a processor reset. If BOOTW is High before and after reset (tied High), the boot ROM is 32 bits wide. If BOOTW is Low before and after reset (tied Low), the boot ROM is 16 bits wide. If BOOTW is Low before reset and High after reset (tied to $\overline{\text{RESET}}$), the boot ROM is 8 bits wide. This signal has special hardening against metastable states, allowing it to be driven with a slow-rise-time signal and permitting it to be tied to $\overline{\text{RESET}}$.

7.1.4 DRAM Interface

$\overline{\text{RAS}}$(3–0) **Row Address Strobe, Banks 3-0 (output, synchronous)**
A High-to-Low transition on one of these signals causes a DRAM in the corresponding bank to latch the row address and begin an access. $\overline{\text{RAS}}$3 starts an access in DRAM Bank 3, and so on. These signals also are used in other special DRAM cycles.

$\overline{\text{CAS}}$(3–0) **Column Address Strobes, Byte 3-0 (output, synchronous)**
A High-to-Low transition on these signals causes the DRAM selected by $\overline{\text{RAS}}$(3–0) to latch the column address and complete the access. To support byte and half-word writes, column address strobes are provided for individual DRAM bytes. $\overline{\text{CAS}}$3 is the column address strobe for the DRAMs, in all banks, attached to ID(31–24). $\overline{\text{CAS}}$2 is for the DRAMs attached to ID(23–16), and so on. These signals are also used in other special DRAM cycles.

$\overline{\text{WE}}$ **Write Enable (output, synchronous)**
This signal is used to write the selected DRAM bank. "Early write" cycles are used so the DRAM data inputs and outputs can be tied to the common ID Bus.

$\overline{\text{TR/OE}}$ **Video DRAM Transfer/Output Enable (output, synchronous)**
This signal is used with video DRAMs to transfer data to the video shift register. It is also used as an output enable in normal video DRAM read cycles.

7.1.5 Peripheral Interface Adapter (PIA)

$\overline{\text{PIACS}}$(5–0) **Peripheral Chip Selects, Regions 5-0 (output, synchronous)**
These signals are used to select individual peripheral devices. DMA Channel 0 may be programmed to use $\overline{\text{PIACS}}$0 during an external peripheral access, and DMA Channel 1 may be programmed to use $\overline{\text{PIACS}}$1.

$\overline{\text{PIAOE}}$ **Peripheral Output Enable (output, synchronous)**
This signal enables the selected peripheral device to drive the ID bus.

$\overline{\text{PIAWE}}$ **Peripheral Write Enable (output, synchronous)**
This signal causes data on the ID bus to be written into the selected peripheral.

7.1.6 DMA Controller

DREQ(1–0) **DMA Request, Channels 1-0 (input, asynchronous)**
These signals request an external transfer on DMA Channel 0 (DREQ0) or DMA Channel 1 (DREQ1). These requests are individually programmable to be either level- or edge-sensitive for either polarity of level or edge. DMA transfers can occur to and from internal peripherals independent of these requests.

$\overline{\text{DACK}}$**(1–0)** **DMA Acknowledge, Channels 1-0 (output, synchronous)**
These signals acknowledge an external transfer on DMA Channel 0 (DREQ0) or DMA Channel 1 (DREQ1). DMA transfers can occur to and from internal peripherals independent of these acknowledgments.

TDMA **Terminate DMA (input, synchronous)**
This signal can be asserted during an external DMA transfer to terminate the transfer after the current access.

$\overline{\text{GREQ}}$ **External Memory Grant Request (input, synchronous)**
This signal is used by an external device to request an access to the Am29200 microprocessor's ROM or DRAM. To perform this access, the external device supplies an address to the Am29200 ROM Controller or DRAM Controller.

To support a hardware-development system, GREQ should be either tied High or held at a high-impedance state during a processor reset.

$\overline{\text{GACK}}$ **External Memory Grant Acknowledge (output, synchronous)**
This signal indicates to an external device that it has been granted an access to the Am29200 microprocessor's ROM or DRAM, and that the device should provide an address.

7.1.7 I/O Port

PIO(15–0) **Programmable Input/Output (input/output, asynchronous)**
These signals are available for direct software control and inspection. PIO(15–8) may be individually programmed to cause processor interrupts. These signals have special hardening against metastable states, allowing them to be driven with slow-transition-time signals.

7.1.8 Parallel Port

PSTROBE **Parallel Port Strobe (input, asynchronous)**
This signal is used by the host to indicate that data is on the Parallel Port or to acknowledge a transfer from the Am29200 microprocessor.

$\overline{\text{PBUSY}}$ **Parallel Port Busy (output, synchronous)**
This indicates to the host that the Parallel Port is busy and cannot accept a data transfer.

PACK **Parallel Port Acknowledge (output, synchronous)**
This signal is used by the Am29200 microprocessor to acknowledge a transfer from the host or to indicate to the host that data has been placed on the port.

PAUTOFD **Parallel Port Autofeed (input, asynchronous)**
This signal is used by the host to indicate how line feeds should be performed or is used to indicate that the host is busy and cannot accept a data transfer.

$\overline{\text{POE}}$ **Parallel Port Output Enable (output, synchronous)**
This signal enables a data buffer containing data from the host to drive the ID Bus.

$\overline{\text{PWE}}$ **Parallel Port Write Enable (output, synchronous)**
This signal writes a buffer with data on the ID Bus. The buffer in turn drives data to the host.

7.1.9 Serial Port

UCLK **UART Clock (input)**
This is an oscillator input for generating the UART (Serial Port) clock. To generate the UART clock, the oscillator frequency may be divided by any amount up to 65,536. The UART clock operates at 16 times the Serial Port's baud rate. As an option, UCLK may be driven with MEMCLK or INCLK. It can be driven with TTL levels.

TXD **Transmit Data (output, asynchronous)**
This output is used to transmit serial data.

RXD **Receive Data (input, asynchronous)**
This input is used to receive serial data.

$\overline{\text{DSR}}$ **Data Set Ready (output, synchronous)**
This indicates to the host that the serial port is ready to transmit or receive data.

$\overline{\text{DTR}}$ **Data Terminal Ready (input, asynchronous)**
This indicates to the Am29200 microprocessor that the host is ready to transmit or receive data.

7.1.10 Video Interface

VCLK **Video Clock (input, asynchronous)**
This clock is used to synchronize the transfer of video data. As an option, VCLK may be driven with MEMCLK or INCLK. It can be driven with TTL levels.

VDAT **Video Data (input/output, synchronous to VCLK)**
This is serial data to or from the video device.

LSYNC **Line Synchronization (input, asynchronous)**
This signal indicates the start of a raster line.

PSYNC **Page Synchronization (input/output, asynchronous)**
This signal indicates the beginning of a raster page.

7.1.11 JTAG 1149.1 Boundary Scan Interface

TCK — **Test Clock Input (asynchronous input)**
This input is used to operate the Test Access Port. The state of the Test Access Port must be held if this clock is held either High or Low. This clock is internally synchronized to MEMCLK for certain operations of the Test Access Port controller, so signals internally driven and sampled by the Test Access Port are synchronous to processor internal clocks.

TMS — **Test Mode Select (input, synchronous to TCK)**
This input is used to control the Test Access Port. If it is not driven, it appears High internally.

TDI — **Test Data Input (input, synchronous to TCK)**
This input supplies data to the test logic from an external source. It is sampled on the rising edge of TCK. If it is not driven, it appears High internally.

TDO — **Test Data Output (3-state output, synchronous to TCK)**
This input supplies data from the test logic to an external destination. It changes on the falling edge of TCK. It is in the high-impedance state except when scanning is in progress.

$\overline{\text{TRST}}$ — **Test Reset Input (asynchronous input)**
This input asynchronously resets the Test Access Port. If $\overline{\text{TRST}}$ is not driven, it appears High internally.

7.2 ACCESS PRIORITY

Many of the processor interface signals are shared between various types of accesses. If more than one access request occurs at the same time, the requests are prioritized as follows, in decreasing order of priority:

1. "Panic mode" DRAM Refresh (see Section 9.2.7)
2. DMA Channel 0 transfer
3. DMA Channel 1 transfer
4. Memory access request by an external device (see Section 11.4)
5. Processor DRAM, PIA, or ROM access for data
6. Processor DRAM or ROM access for an instruction

External DMA transfers require two accesses: one to read the data from a peripheral or the DRAM, and another to write the data to a peripheral or DRAM. The two accesses are performed back-to-back, without interruption by another access.

Some processor accesses to narrow memories (a narrow memory is 8 or 16 bits wide) require two or four accesses; for example, reading 32 bits from an 8-bit-wide ROM requires four reads. These accesses are also performed back-to-back, without interruption.

DRAM refresh cycles are normally overlapped with other, non-DRAM accesses. Because normal refresh cycles are performed when there is no conflict with other accesses, these cycles are not prioritized in the above list.

7.3 SYSTEM ADDRESS PARTITION

All addresses are in the processor's instruction/data memory address space. The processor's address space is partitioned as shown in Table 7-1.

Table 7-1 Internal Peripheral Address Assignments

Address Range (hexadecimal)	Selection
00000000–03FFFFFF	ROM Banks (all)
40000000–43FFFFFF	DRAM Banks (all)
50000000–50FFFFFF	Mapped DRAM Banks (all)
60000000–63FFFFFF	VDRAM transfer cycles
80000000–800000FC	Internal peripherals/controllers
90000000–90FFFFFF	PIA Region 0 ($\overline{PCS0}$)
91000000–91FFFFFF	PIA Region 1 ($\overline{PCS1}$)
92000000–92FFFFFF	PIA Region 2 ($\overline{PCS2}$)
93000000–93FFFFFF	PIA Region 3 ($\overline{PCS3}$)
94000000–94FFFFFF	PIA Region 4 ($\overline{PCS4}$)
95000000–95FFFFFF	PIA Region 5 ($\overline{PCS5}$)
—all others—	Reserved

An access to any unimplemented address or address range has an unpredictable effect on processor operation.

7.4 INTERNAL PERIPHERALS AND CONTROLLERS

Internal peripherals are accessed via interface registers selected by offsets from address 80000000 (hexadecimal). The assignment of registers to offsets is shown in Table 7-2. Nearly all registers are read/write and are 32 bits in length (a few register bits are read only, bits in the Interrupt Control Register are reset-only, and the DMA0 Address Tail Register and DMA0 Count Tail Register are both write-only). It is not possible to perform writes on individual bytes or halfwords of any register. Unimplemented register bits are read as zeros and should be written with zeros to insure compatibility with future processor versions.

Table 7-2 Internal Peripheral Address Assignments

Peripheral	Address (hex)	Register
ROM Controller	80000000	ROM Control Register
	80000004	ROM Configuration Register
DRAM Controller	80000008	DRAM Control Register
	8000000C	DRAM Configuration Register
DRAM Mapping Unit	80000010	DRAM Mapping Register 0
	80000014	DRAM Mapping Register 1
	80000018	DRAM Mapping Register 2
	8000001C	DRAM Mapping Register 3
Peripheral Interface Adapter	80000020	PIA Control Register 0
	80000024	PIA Control Register 1
Interrupt Controller	80000028	Interrupt Control Register
DMA Channel 0	80000030	DMA0 Control Register
	80000034	DMA0 Address Register
	80000036	DMA0 Address Tail Register
	80000038	DMA0 Count Register
	8000003A	DMA0 Count Tail Register
DMA Channel 1	80000040	DMA1 Control Register
	80000044	DMA1 Address Register
	80000048	DMA1 Count Register
Serial Port	80000080	Serial Port Control Register
	80000084	Serial Port Status Register
	80000088	Serial Port Transmit Holding Register
	8000008C	Serial Port Receive Buffer Register
	80000090	Baud Rate Divisor Register
Parallel Port	800000C0	Parallel Port Control Register
	800000C1	Parallel Port Status Register
	800000C4	Parallel Port Data Register
Programmable I/O Port	800000D0	PIO Control Register
	800000D4	PIO Input Register
	800000D8	PIO Output Register
	800000DC	PIO Output Enable Register
Video Interface	800000E0	Video Control Register
	800000E4	Top Margin Register
	800000E8	Side Margin Register
	800000EC	Video Data Holding Register
	—all others—	Reserved

CHAPTER 8

ROM CONTROLLER

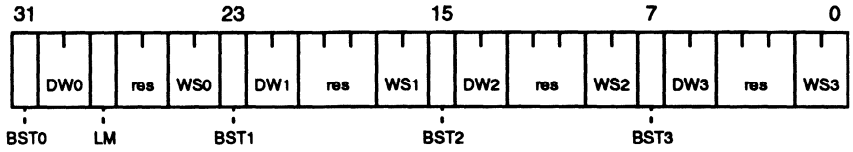
The ROM Interface accommodates up to four banks of ROM that appear as a contiguous memory. Each bank is individually configurable in width and timing. This chapter describes the operation of the ROM controller.

8.1 PROGRAMMABLE REGISTERS

8.1.1 ROM Control Register (RMCT, Address 80000000)

The ROM Control Register (Figure 8-1) controls the access of ROM Banks 0 through 3.

Figure 8-1 ROM Control Register



Bit 31: Burst-Mode ROM, Bank 0 (BST0)—When this bit is 1, ROM Bank 0 is accessed using the burst-mode protocol, in which sequential accesses are completed at the rate of one access per cycle. When this bit is 0, the burst-mode protocol is not used.

Bits 30-29: Data Width, Bank 0 (DW0)—This field indicates the width of the ROM in Bank 0, as follows:

DW0	ROM Width
00	32 bits
01	8 bits
10	16 bits
11	Reserved

Bit 28: Large Memory (LM)—This bit controls the size of the ROM banks and the total size of the ROM address space. If the LM bit is 0, each ROM bank is up to 4 Mbytes in size, for a total of 16 Mbytes. If the LM bit is 1, each ROM bank is up to 16 Mbytes in size, for a total of 64 Mbytes.

Bits 27-26: Reserved.

Bits 25-24: Wait States, Bank 0 (WS0)—This field specifies the number of wait states in a ROM access: that is, the number of cycles in addition to one cycle required

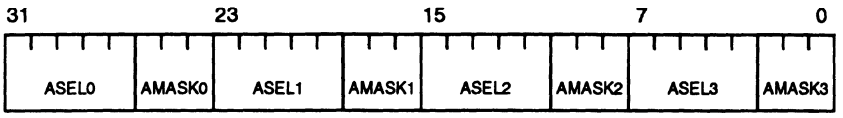
to access the ROM. Zero-wait-state cycles are supported only for non-burst-mode ROM reads. Writes to the ROM address space and burst-mode ROMs require at least one wait state.

Other bits of this register have a definition similar to BST0 and DW0 for ROM Banks 1 through 3.

8.1.2 ROM Configuration Register (RMCF, Address 80000004)

The ROM Configuration Register (Figure 8-2) controls the selection of ROM Banks 0 through 3. In most systems, this register should be set by software to cause the four banks of ROM to appear as a single, contiguous region of memory.

Figure 8-2 ROM Configuration Register



Bits 31-27: Address Select, Bank 0 (ASEL0)—On a load, store, or instruction access, this field is compared against bits of the access address, with the comparisons possibly masked by the AMASK0 field. The unmasked bits of the ASEL0 field must match the corresponding bits of the address for ROM bank 0 to be accessed.

Bits 26-24: Address Mask, Bank 0 (AMASK0)—This field masks the comparison of the ASEL0 field with bits of the address on an access, to permit various sizes of memories and memory chips in ROM Bank 0 ("*ad*(x:y)" represents a field of address bits x through y, inclusive).

AMASK0 Value	Address Comparison (LM=0)	Address Comparison (LM=1)
000	ASEL0(4:0) to *ad*(23:19)	ASEL0(4:0) to *ad*(25:21)
001	ASEL0(4:1) to *ad*(23:20)	ASEL0(4:1) to *ad*(25:22)
011	ASEL0(4:2) to *ad*(23:21)	ASEL0(4:2) to *ad*(25:23)
111	ASEL0(4:3) to *ad*(23:22)	ASEL0(4:3) to *ad*(25:24)

Only the AMASK0 values shown in the above table are valid. The AMASK0 field permits various sizes of memories and memory chips in ROM Bank 0 that are independent of the sizes in the other banks.

Other bits of this register have a definition similar to ASEL0 and AMASK0 for ROM banks 1 through 3.

8.1.3 Initialization

ROM Bank 0 is used as the boot ROM containing the initialization code for the processor and peripherals. The width of this ROM is set by the BOOTW signal, which is sampled during and after a processor reset. If BOOTW is High before and after reset (tied High), the boot ROM is 32 bits wide. If BOOTW is Low before and after reset (tied Low), the boot ROM is 16 bits wide. If BOOTW is Low before reset and High

after reset (tied to $\overline{\text{RESET}}$), the boot ROM is 8 bits wide. The BOOTW signal is used to set the DW0 field before the boot ROM is accessed. The boot ROM defaults to a non-burst-mode ROM with three wait states until the ROM Control Register and ROM Configuration Register are set with the correct configuration. The LM bit is reset to 0. The ASEL0 and AMASK0 fields are both set to zero by a processor reset.

To prevent bank conflicts during initialization, the ASEL and AMASK fields for ROM banks 1 through 3 are set to all 1s. The configuration of ROM banks 1 through 3, if present, must be set by software before the respective bank is accessed.

8.2 ROM ACCESSES

8.2.1 ROM Address Mapping

The ASEL and AMASK fields allow the four ROM banks to appear as a contiguous region of ROM, with the restriction that a bank of a certain size must fit on the natural address boundary for that size. For example, a 2-Mbyte ROM must be placed on a 2-Mbyte address boundary. For this reason, ROM banks must appear in the address space in order of decreasing bank size. Note that to achieve a contiguous memory, the various ROM banks need not appear in sequence in the address space. For example, ROM Bank 3 may appear in an address range below the address range for ROM Bank 1 or 2. The only restriction in the placement of ROM banks is that ROM Bank 0 is used for the initial instruction fetches after a processor reset, starting at address 00000000, hexadecimal.

8.2.2 Simple ROM Accesses

Figure 8-3 shows the timing of a simple ROM read cycle. The number of cycles is controlled by the WSx field in the ROM Control Register ("x" represents one of ROM Banks 0 through 3). The WSx field specifies the number of waits states: that is, the number of cycles beyond one cycle required to access the ROM. Figure 8-4 shows the timing of a zero-wait-state ROM read (WSx = 00). In this case, the $\overline{\text{ROMOE}}$ signal is asserted at the midpoint of the cycle rather than at the beginning of the second cycle (since there is no second cycle).

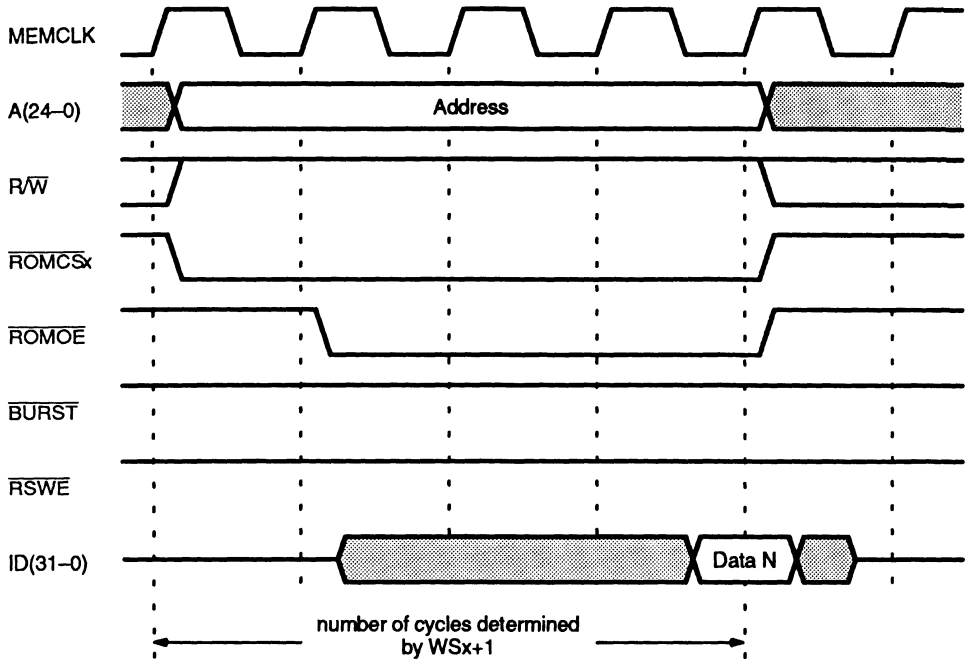
8.2.3 Writes to the ROM Space

Figure 8-5 shows the timing of a write to the ROM address space. This cycle is provided for alterable memories in the ROM space, such as SRAMs or Flash EPROMs. Zero-wait-state cycles are not supported for writes. Because there are no individual byte write enables and because of processor limitations, the ROM must be at least 16 bits wide to support writes (see Section 8.2.5). Also, the width of data written must be at least as wide as the ROM, so that 16-bit data cannot be written into a 32-bit-wide ROM. If 32-bit data is written into a 16-bit-wide ROM, the processor performs two simple writes to write the entire 32 bits.

8.2.4 Burst-Mode ROM Accesses

Figure 8-6 shows the timing of a burst-mode ROM access, for direct connection to burst-mode devices such as AMD's 27B010 burst-mode EPROM. Burst-mode

Figure 8-3 Simple ROM Read Cycle



accesses require at least one wait state for the initial access. Burst-mode writes are not supported.

8.2.5 Narrow ROM Accesses

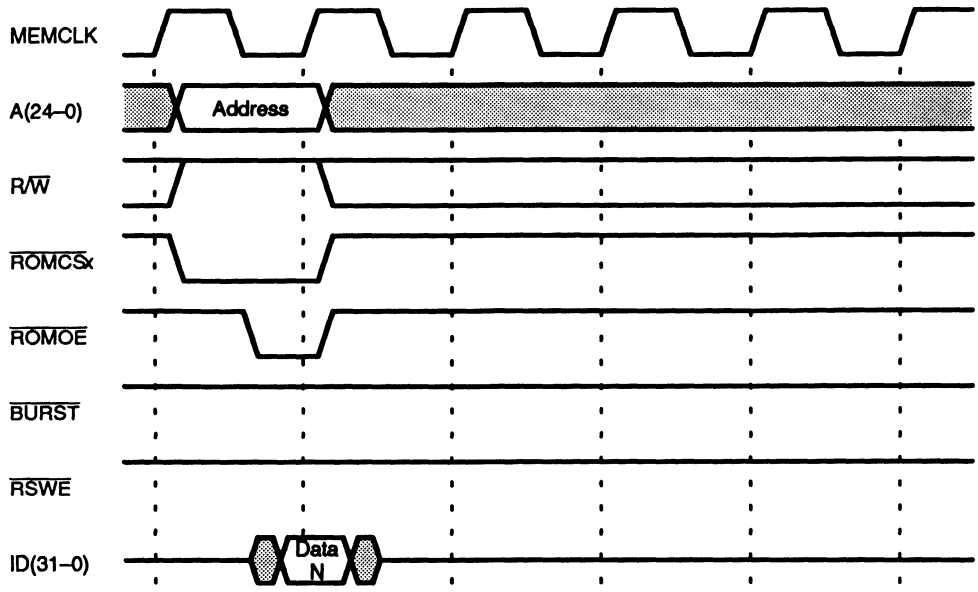
A narrow ROM is one that is less than 32 bits wide. The Am29200 microprocessor supports 8- and 16-bit-wide ROMs in any bank, as determined by the DWx field in the ROM Control Register. An 8-bit-wide ROM is attached to ID(31–24). A 16-bit-wide ROM is attached to ID(31–16) and ignores A0. A 32-bit ROM is attached to ID(31–0) and ignores A(1–0). A narrow ROM can respond to any read access, but the ROM must be at least 16 bits wide to respond to writes, and the data written must be as wide as or wider than the ROM. Thus, only 32-bit data may be written into a 32-bit-wide ROM, and only 16- and 32-bit data may be written into a 16-bit-wide ROM.

8.2.5.1 8-BIT NARROW ACCESSES

If the processor expects a half-word or a word on a read (that is, if the access is not a byte read), and a narrow ROM is 8 bits wide, the Am29200 microprocessor generates one (for a half-word) or three (for a word) requests immediately following the first access. No other intervening accesses are performed. The address for each subsequent access is the same as the address for the first access, except that A(1–0) are incremented by one for each access. A burst-mode access may be performed for the subsequent bytes if the ROM permits such an access.

The Am29200 microprocessor assembles the final word or half-word by placing the first received byte in the high-order byte position of the word or half-word. The

Figure 8-4 Simple ROM Read Cycle—Zero Wait States



second received byte is placed in the next-lower-order byte position and so on until the entire word or half-word is assembled.

If the read access is a byte access, the processor performs only one access.

If software generates an unaligned half-word or word read, the narrow ROM does not permit the implementation of the unaligned read. The address sequence generated to assemble the half-word or word wraps within the half-word or word.

8.2.5.2 16-BIT NARROW ACCESSES

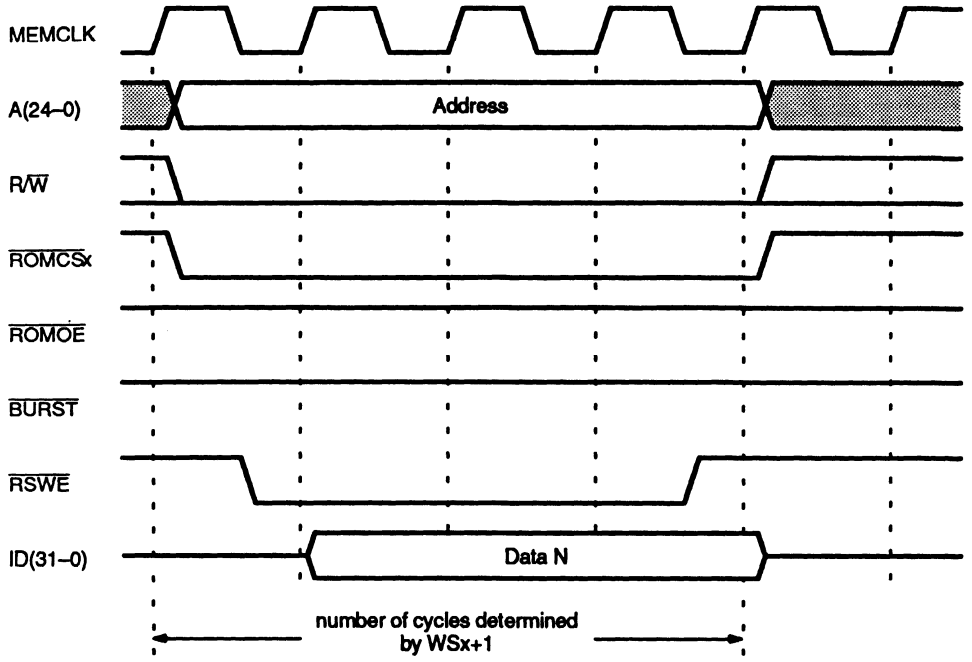
If the processor expects a word on a read, and a narrow ROM is 16 bits wide, the Am29200 microprocessor generates one more request immediately following the first access. No other intervening accesses are performed. The address for the second access is the same as the address for the first access, except that A(1–0) are incremented by two for the second access. A burst-mode access may be performed for the second 16 bits if the ROM permits such an access.

The Am29200 microprocessor assembles the final word by placing the first received half-word in the high-order half-word position of the word, and the second received half-word in the low-order half-word position.

If the read access is a byte or half-word access, the processor performs only one access.

If software generates an unaligned word read, the narrow ROM does not permit the implementation of the unaligned read. The address sequence generated to assemble the word wraps within the word.

Figure 8-5 **Simple Write to ROM Bank (for alterable memories in the ROM address space)**



8.2.6 Use of WAIT to Extend ROM Cycles

If the WAIT signal is asserted before the end of a ROM access (that is, before the cycle in which ROMCSx is deasserted), the processor extends the ROM access until WAIT is deasserted. This permits the system to extend the ROM access indefinitely. The access ends on the cycle after WAIT is deasserted, both for reads (Figure 8-7) and for writes (Figure 8-8).

The WAIT signal can also be used to extend individual accesses in a sequence of burst-mode accesses. For each access, the processor does not consider the data to be valid until the cycle after WAIT is High.

Figure 8-6 Burst-Mode ROM Read

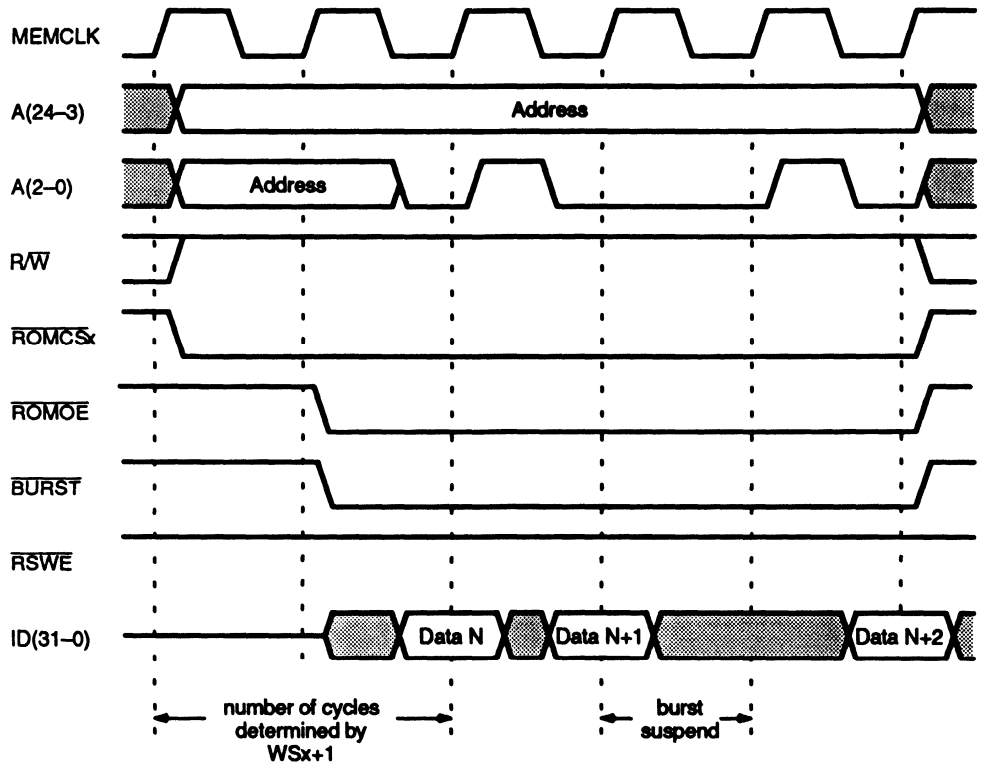


Figure 8-7 Extending a ROM Read Cycle with $\overline{WAIT}$

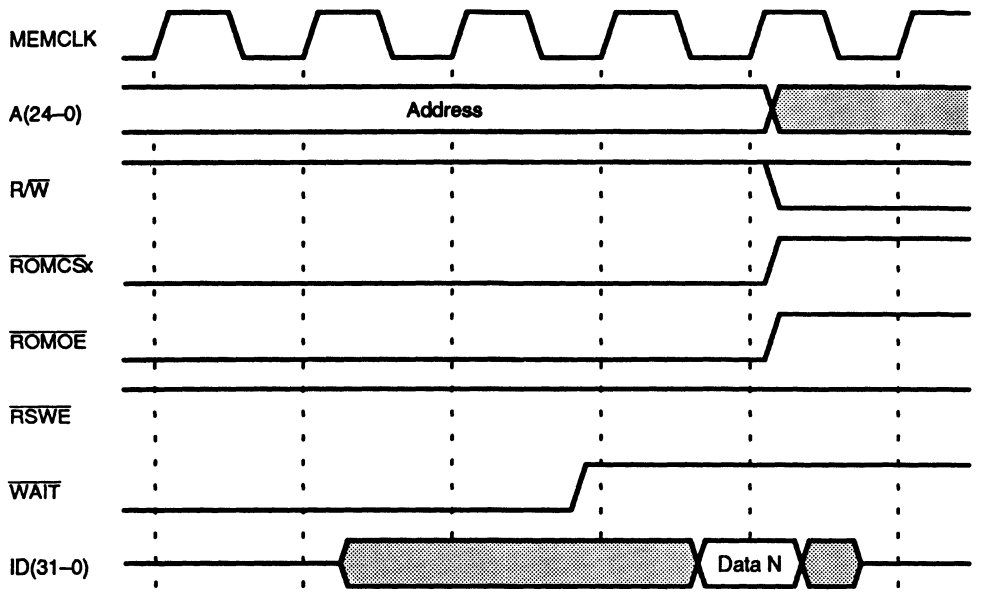
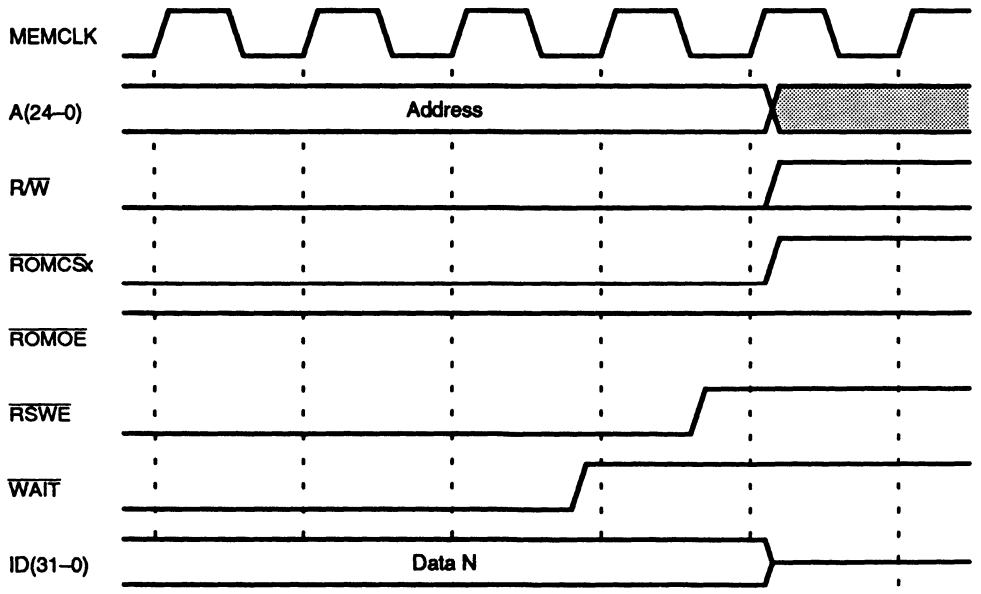


Figure 8-8 Extending a ROM Write Cycle with $\overline{WAIT}$



CHAPTER 9

DRAM CONTROLLER

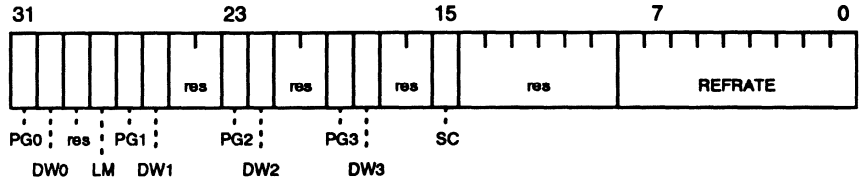
The DRAM Interface accommodates up to four banks of DRAM that appear as a contiguous memory. Each bank is individually configurable in width. Four, 64-Kbyte regions of the DRAM can be mapped into a 16-Mbyte virtual address space.

9.1 PROGRAMMABLE REGISTERS

9.1.1 DRAM Control Register (DRCT, Address 80000008)

The DRAM Control Register (Figure 9-1) controls the access to and refresh of DRAM Banks 0 through 3.

Figure 9-1 DRAM Control Register



Bit 31: Page-Mode DRAM, Bank 0 (PG0)—When this bit is 1, burst-mode accesses to DRAM Bank 0 are performed using page-mode accesses for all but the first access. When this bit is 0, page-mode accesses are not performed.

Bit 30: Data Width, Bank 0 (DW0)—This field indicates the width of the DRAM in Bank 0, as follows:

DW Value	DRAM Width
0	32 bits
1	16 bits

Bit 29: Reserved.

Bit 28: Large Memory (LM)—This bit controls the size of the DRAM banks and the total size of the DRAM address space. If the LM bit is 0, each DRAM bank is up to 4 Mbytes in size, for a total of 16 Mbytes. If the LM bit is 1, each DRAM bank is up to 16 Mbytes in size, for a total of 64 Mbytes.

PG1, DW1, and so on perform functions similar to PG0 and DW0 for DRAM Banks 1 through 3.

Bit 15: Static-Column DRAM (SC)—When this bit is 1, page-mode accesses to the DRAM are performed using static-column accesses. Static column accesses differ from page-mode cycles only in that $\overline{CAS}$(3–0) are held Low throughout a read access. The timing of the access is not affected, and write accesses are not affected. When this bit is 0, normal page-mode accesses are performed, if enabled.

Bits 14-9: Reserved.

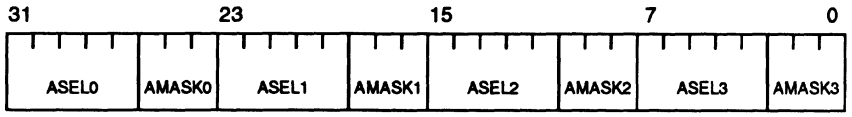
Bits 8-0: Refresh Rate (REFRATE)—This field indicates the number of MEMCLK cycles between DRAM refresh cycles. "CAS before RAS" cycles are performed, overlapped in the background with other non-DRAM accesses when possible. If one or more banks have not been refreshed in the background when the REFRATE interval expires, the processor forces refresh of the unrefreshed banks.

A zero in the REFRATE field disables refresh. Upon reset, this field is initialized to the value 1ff, hexadecimal.

9.1.2 DRAM Configuration Register (DRCF, Address 8000000C)

The DRAM Configuration Register (Figure 9-2) controls the selection of DRAM Banks 0 through 3. In most systems, this register should be set by software to cause the four banks of DRAM to appear as a single, contiguous region of memory.

Figure 9-2 DRAM Configuration Register



Bits 31-27: Address Select, Bank 0 (ASEL0)—On a load, store, or instruction access, this field is compared against bits of the access address, with the comparisons possibly masked by the AMASK0 field. The unmasked bits of the ASEL0 field must match the corresponding bits of the address for DRAM bank 0 to be accessed.

Bits 26-24: Address Mask, Bank 0 (AMASK0)—This field masks the comparison of the ASEL0 field with bits of the address on an access, to permit various sizes of memories and memory chips in DRAM Bank 0 ("*ad*(x:y)" represents a field of address bits x through y, inclusive).

AMASK0 Value	Address Comparison (LM=0)	Address Comparison (LM=1)
000	ASEL0(4:0) to *ad*(23:19)	ASEL0(4:0) to *ad*(25:21)
001	ASEL0(4:1) to *ad*(23:20)	ASEL0(4:1) to *ad*(25:22)
011	ASEL0(4:2) to *ad*(23:21)	ASEL0(4:2) to *ad*(25:23)
111	ASEL0(4:3) to *ad*(23:22)	ASEL0(4:3) to *ad*(25:24)

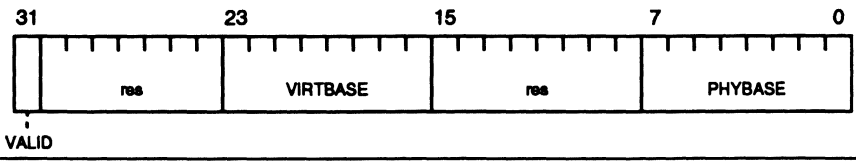
Only the AMASK0 values shown in the above table are valid.

Other bits of this register have a definition similar to ASEL0 and AMASK0 for DRAM Banks 1 through 3.

9.1.3 DRAM Mapping Register 0 (DRM0, Address 80000010)

This register (Figure 9-3) specifies one of four possible mappings of a mapped DRAM access.

Figure 9-3 DRAM Mapping Register 0



Bit 31: Valid Mapping (VALID)—This bit, when 1, indicates that the mapping specified by the VIRTBASE and PHYBASE fields is valid.

Bits 30-24: Reserved.

Bits 23-16: Virtual Base Address (VIRTBASE)—This field specifies the virtual base address of the mapped region. On a mapped DRAM access, it is compared against bits 23-16 of the address generated by the load or store instruction. The comparison must match for the mapping to be performed.

Bits 15-8: Reserved.

Bits 7-0: Physical Base Address (PHYBASE)—This field specifies the physical base address of the mapped region. On a mapped DRAM access, if the comparison of the virtual base address yields a match and the VALID bit is 1, the PHYBASE field replaces bits 23-16 of the address.

9.1.4 DRAM Mapping Register 1 (DRM1, Address 80000014)

This register is identical in layout and definition to the DRAM Mapping Register 0. It specifies the second of the four possible mappings.

9.1.5 DRAM Mapping Register 2 (DRM2, Address 80000018)

This register is identical in layout and definition to the DRAM Mapping Register 0. It specifies the third of the four possible mappings.

9.1.6 DRAM Mapping Register 3 (DRM3, Address 8000001C)

This register is identical in layout and definition to the DRAM Mapping Register 0. It specifies the the fourth of the four possible mappings.

9.1.7 Initialization

The configuration of DRAM banks, if present, must be set by software before normal DRAM accesses are performed (the DRAM may be accessed using default parameters that are set by software to determine the configuration of the DRAM). The DRAM Mapping registers are not initialized by a processor reset, and must be set by software before a mapped DRAM access occurs. The REFRATE field is initialized on reset to the value 1ff, hexadecimal.

9.2 DRAM ACCESSES

9.2.1 DRAM Address Mapping

The ASEL and AMASK fields allow the four DRAM banks to appear as a contiguous region of DRAM, with the restriction that a bank of a certain size must fit on the natural address boundary for that size. For example, a 2-Mbyte DRAM must be placed on a 2-Mbyte address boundary. For this reason, DRAM banks must appear in the address space in order of decreasing bank size. Note that to achieve a contiguous memory, the various DRAM banks need not appear in sequence in the address space. For example, DRAM Bank 3 may appear in an address range below the address range for DRAM Bank 1 or 2. This provides flexibility in meeting the restriction that DRAM banks appear in the address space in order of decreasing size.

9.2.2 Address Multiplexing

The address multiplexing for the DRAMs is performed directly by the Am29200 microprocessor on the A(14–1) pins, and no external multiplexing is required. Address multiplexing is performed as follows ("*ax*" represents address bit x):

When	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1
$\overline{RAS}$ asserted	*a22*	*a21*	*a20*	*a19*	*a19*	*a18*	*a17*	*a16*	*a15*	*a14*	*a13*	*a12*	*a11*	*a10*
$\overline{CAS}$ asserted	*a23*	*a22*	*a21*	*a20*	*a10*	*a9*	*a8*	*a7*	*a6*	*a5*	*a4*	*a3*	*a2*	*a1*

Table 9-1 shows how this multiplexing of addresses supports various configuration of memory densities and memory widths, assuming the individual DRAMs are 4 bits wide. The addresses shown in Table 9-1 are the address bits for an access. Table 9-2 shows how the various memories should be connected to the processor's address pins to realize this address multiplexing, again assuming the individual DRAMs are 4 bits wide.

Table 9-1 DRAM Address Multiplexing (by-4 DRAMs)

DRAM density	DRAM width	Portion of cycle	DRAM multiplexed address bits										
			10	9	8	7	6	5	4	3	2	1	0
1 Mbit	16 bits	RAS			a18	a17	a16	a15	a14	a13	a12	a11	a10
		CAS			a9	a8	a7	a6	a5	a4	a3	a2	a1
	32 bits	RAS			a19	a18	a17	a16	a15	a14	a13	a12	a11
		CAS			a10	a9	a8	a7	a6	a5	a4	a3	a2
4 Mbit	16 bits	RAS		a19	a18	a17	a16	a15	a14	a13	a12	a11	a10
		CAS		a20	a9	a8	a7	a6	a5	a4	a3	a2	a1
	32 bits	RAS		a20	a19	a18	a17	a16	a15	a14	a13	a12	a11
		CAS		a21	a10	a9	a8	a7	a6	a5	a4	a3	a2
16 Mbit	16 bits	RAS	a21	a19	a18	a17	a16	a15	a14	a13	a12	a11	a10
		CAS	a22	a20	a9	a8	a7	a6	a5	a4	a3	a2	a1
	32 bits	RAS	a22	a20	a19	a18	a17	a16	a15	a14	a13	a12	a11
		CAS	a23	a21	a10	a9	a8	a7	a6	a5	a4	a3	a2

Table 9-2 DRAM Address Connections to Am29200 Microprocessor (by-4 DRAMs)

DRAM density	DRAM width	DRAM multiplexed address bits										
		10	9	8	7	6	5	4	3	2	1	0
1 Mbit	16 bits			A9	A8	A7	A6	A5	A4	A3	A2	A1
	32 bits			A10	A9	A8	A7	A6	A5	A4	A3	A2
4 Mbit	16 bits		A11	A9	A8	A7	A6	A5	A4	A3	A2	A1
	32 bits		A12	A10	A9	A8	A7	A6	A5	A4	A3	A2
16 Mbit	16 bits	A13	A11	A9	A8	A7	A6	A5	A4	A3	A2	A1
	32 bits	A14	A12	A10	A9	A8	A7	A6	A5	A4	A3	A2

Sequential accesses can use page-mode accesses, even though not all CAS address bits are contiguous address bits, because the processor does not generate a page-mode access across a 1 Kbyte address boundary. Thus the processor will not change any address bits other than *a*(9:1) during a page-mode access.

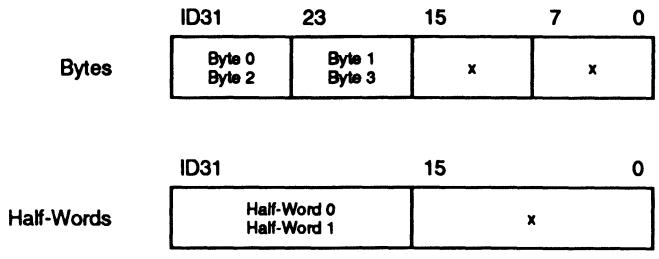
9.2.3 Sixteen-Bit DRAM Width

For a data access, the width of each DRAM bank can be programmed to be either 16 or 32 bits by the DRAM Control Register. If the DRAM is 16 bits wide, only ID(31–16) are used to transfer data to and from the processor, and the processor performs two accesses to read or write a full word.

To read a 32-bit word from a 16-bit DRAM bank, the processor first reads the high-order 16 bits of the word, then generates a second access to read the low-order 16 bits of the word. The address is incremented by two for the second access. To read an 8-bit byte or 16-bit half-word from a 16-bit DRAM, the processor performs only a single access. Alignment and sign extension are performed as usual, except the required byte or half-word is received on ID(31–16). Figure 9-4 shows the location of bytes and half-words from a 16-bit DRAM bank. In Figure 9-4, bytes and half-words are numbered as they are numbered in a word.

To write a 32-bit word into a 16-bit DRAM bank, the processor first writes the high-order 16 bits of the word, then generates a second access to write the low-order 16 bits of the word. The address is incremented by two for the second access, and the low-order bits of the word appear on ID(31–16). To write an 8-bit byte or 16-bit half-word on a 16-bit bus, the processor performs only a single access. For a byte write, the appropriate byte is replicated on both ID(31–24) and ID(23–16). For a half-word

Figure 9-4 Location of Bytes and Half-Words on a 16-Bit Bus



write, the appropriate half-word appears on ID(31–16). The $\overline{CAS}$(3–0) signals are asserted as follows (the value "0" is Low, "1" is High, and "x" is a don't care):

Data width	A(1–0)	$\overline{CAS}$(3–0) (on write)
8 bits	00	0111
8 bits	01	1011
8 bits	10	0111
8 bits	11	1011
16 bits	0x	0011
16 bits	1x	0011
—all other writes (two cycles)—		0011

9.2.4 Mapped DRAM Accesses

Processor DRAM accesses in the 16-Mbyte address range 50000000 – 50FFFFFF are mapped to one of four 64-Kbyte regions of the DRAM. This provides a virtual memory region supporting functions such as image compression and decompression that yield lower overall memory requirements and thus lower system cost. Only processor DRAM accesses can be mapped. DRAM accesses by a DMA channel cannot be mapped.

DRAM Mapping Registers 0 through 3 each specify a DRAM mapping. Before an access to a DRAM location having an address in the range 50000000 – 50FFFFFF, bits 23-16 of the address are compared to the VIRTBASE fields in each of the DRAM Mapping registers. If the address bits match the VIRTBASE field in one of the registers, and the associated VALID bit is 1, then the PHYBASE field replaces bits 23-16 of the address before the access is performed. If more than one valid comparison occurs, the mapping specified by DRAM Mapping Register 0 has the highest priority, and the mapping specified by DRAM Mapping Register 3 has the lowest priority. If no valid comparison is detected, the processor's User- or Supervisor-mode Instruction or Data Mapping Miss occurs, depending on the program mode and type of access.

9.2.5 Normal Access Timing

Figure 9-5 shows the timing for a normal DRAM read cycle. Figure 9-6 shows the timing for a normal DRAM write cycle. DRAM cycles are fixed at four cycles and cannot be extended with $\overline{WAIT}$. An additional cycle is taken after the data is read or written to permit time for $\overline{RAS}$ precharge. The rising edge of $\overline{RAS}$ occurs on the third rising edge of MEMCLK after the beginning of the cycle.

9.2.6 Page-Mode Access Timing

Page-mode accesses can be enabled for each bank to reduce the average access time for a sequence of accesses. If enabled, page-mode accesses are performed for instruction accesses and for the LOADM and STOREM instructions. Page-mode accesses permit an access time of two cycles for all but the first access. When the DRAM bank is 16 bits wide, two accesses are required to obtain a 32-bit word. Page-mode accesses are performed to access the second 16 bits in this case if page-mode accesses are enabled.

Figure 9-7 shows the timing for a page-mode DRAM read cycle. Figure 9-8 shows the timing for a page-mode DRAM write cycle. Static-column accesses are performed

Figure 9-5 DRAM Read Cycle

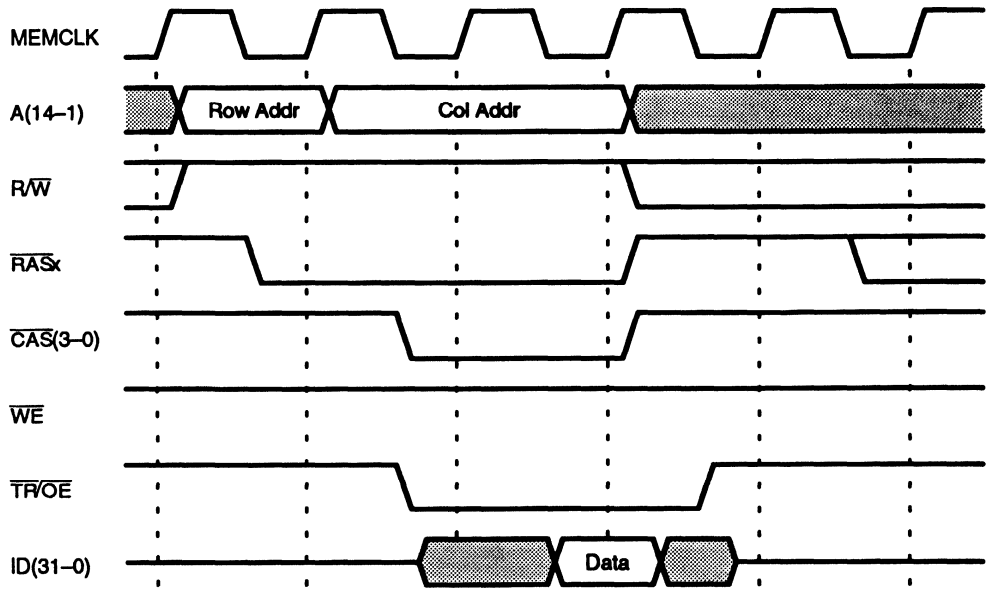


Figure 9-6 DRAM Write Cycle

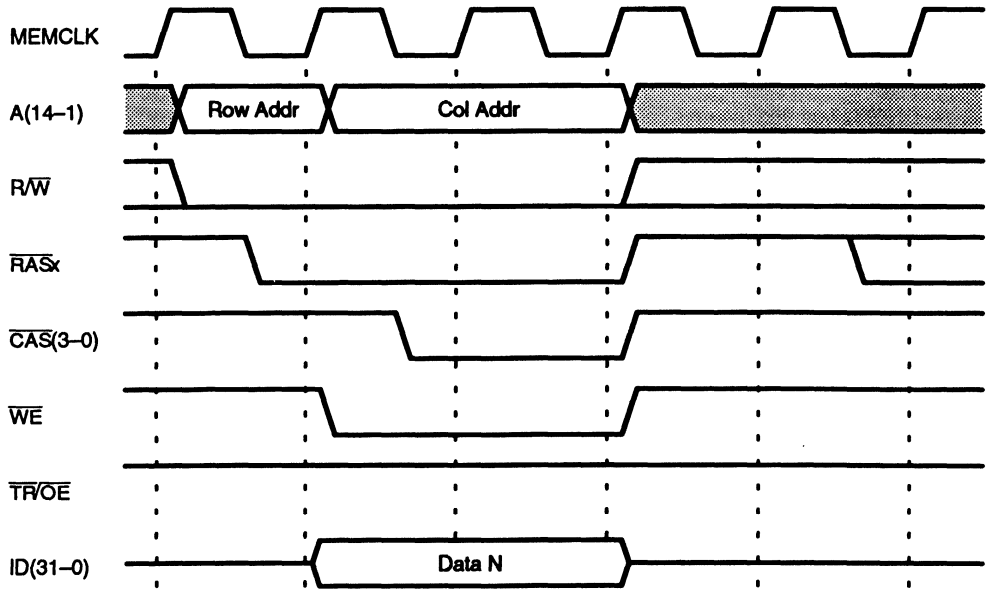
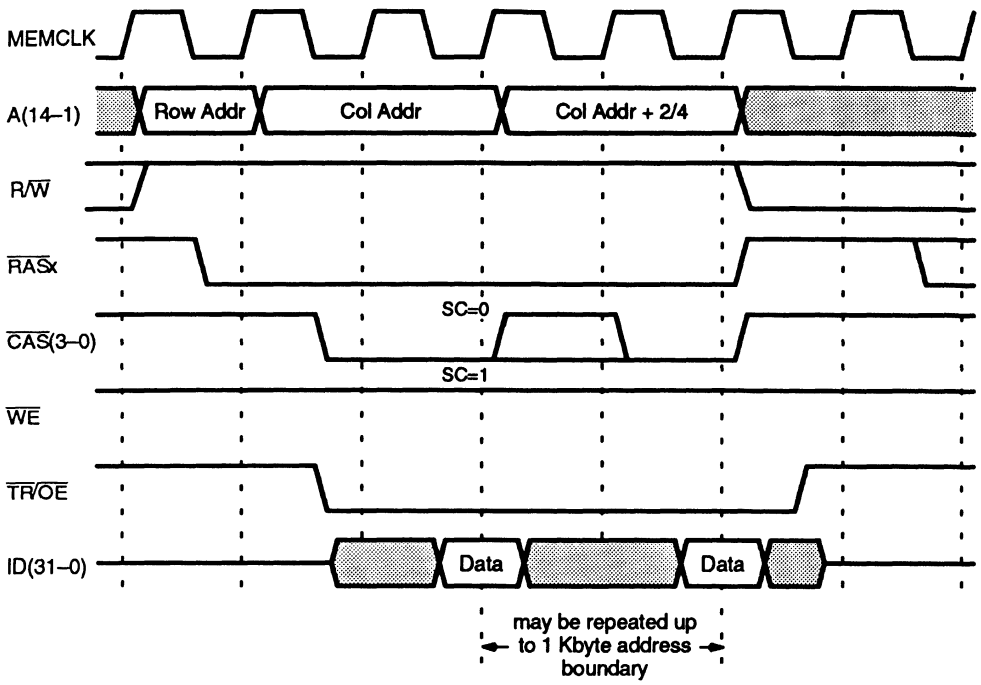


Figure 9-7 DRAM Page-Mode Read Cycle



if SC=1 in the DRAM Control Register. Static-column accesses differ from page-mode accesses only in that CAS(3–0) remain Low throughout the access.

9.2.7 DRAM Refresh

"CAS before RAS" refresh cycles are performed periodically, as determined by the REFRATE field of the DRAM Control Register. The REFRATE field specifies the number of MEMCLK cycles in a refresh interval; a zero in this field disables refresh. The Am29200 microprocessor insures that one row of each DRAM bank is refreshed in every interval. Each bank is refreshed separately to distribute the demand placed on the DRAM power supplies by the individual banks.

Figure 9-9 shows the timing of a refresh cycle. Because refresh cycles use only the RAS(3–0) and CAS(3–0) signals, the processor attempts to perform refresh in the background, refreshing each bank in the cycles that the DRAM is not being used, possibly overlapped with ROM and PIA accesses. Background refresh incurs very little overhead. The average penalty of refresh is about 2 cycles per refresh interval. This penalty arises because the processor sometimes attempts to access the DRAM after a refresh cycle has been started. If one or more banks has not been refreshed by the end of a refresh interval, the DRAM controller performs "panic mode" refresh cycles to refresh the remaining banks. Panic mode refresh cycles take priority over all other processor accesses.

Figure 9-8 DRAM Page-Mode Write Cycle

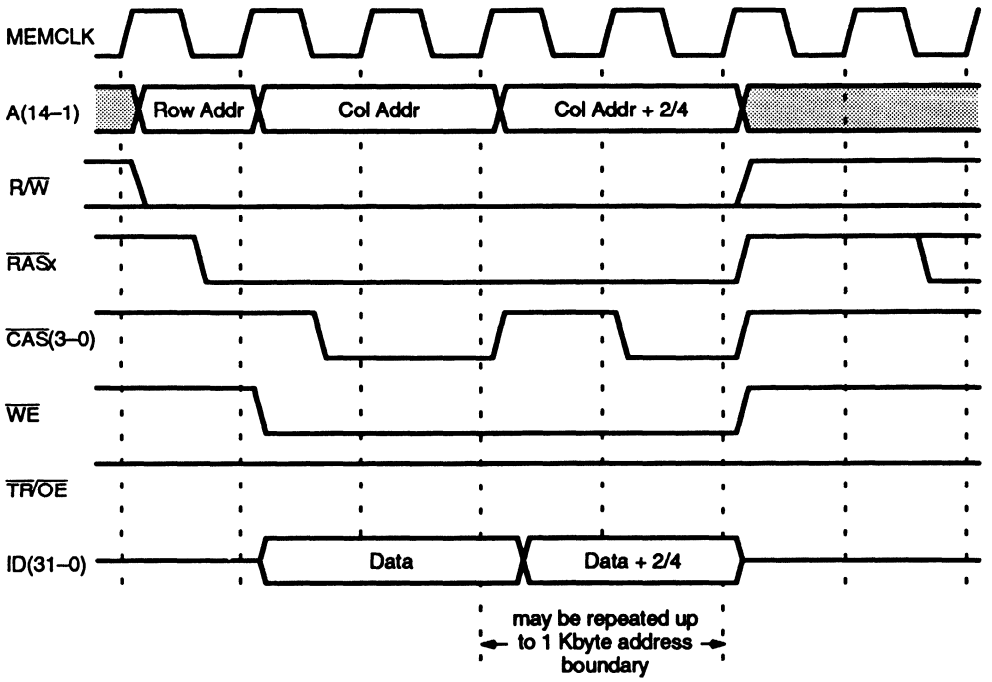
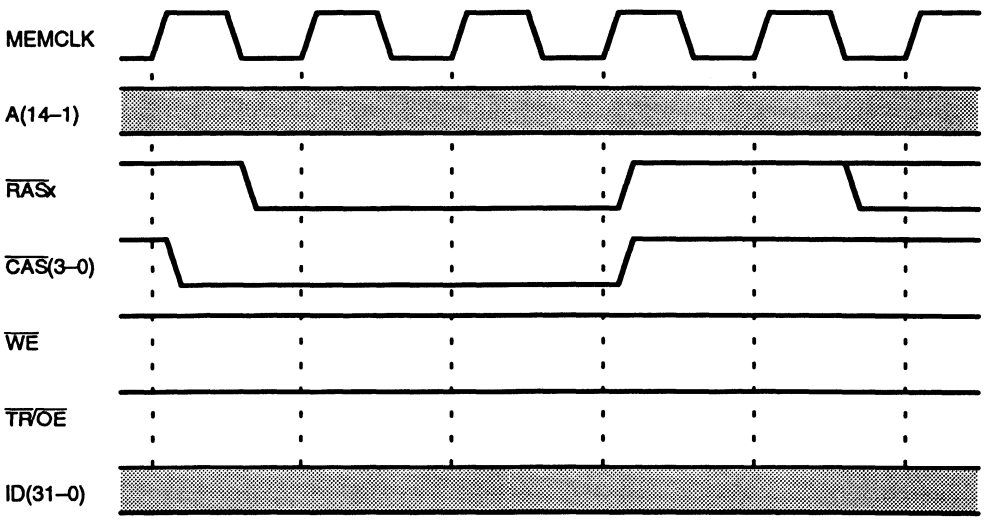


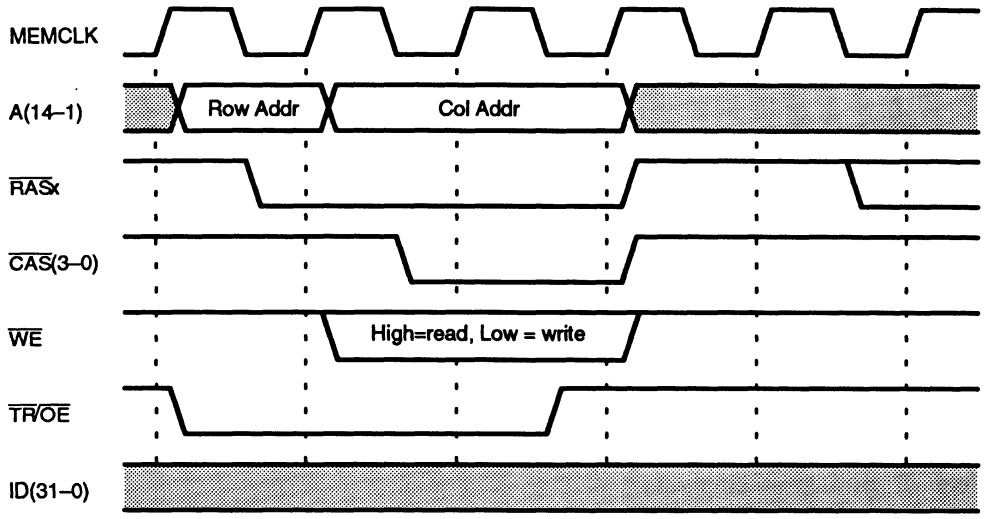
Figure 9-9 DRAM Refresh Cycle



9.2.8 Video DRAM Interface

A video DRAM (VDRAM) transfer cycle is performed during accesses in the range 60000000 – 63FFFFFF (hexadecimal). These cycles permit the transfer of data to a VDRAM shift register in graphics applications. Figure 9-10 shows the timing of a VDRAM transfer cycle. This cycle differs from a normal DRAM cycle because the signal $\overline{TR/OE}$ is asserted with different timing.

Figure 9-10 VDRAM Transfer Cycle



CHAPTER 10

PERIPHERAL INTERFACE ADAPTER

The Peripheral Interface Adapter (PIA) permits direct attachment of up to six peripheral devices, each with its own 24-bit address space.

10.1 PROGRAMMABLE REGISTERS

10.1.1 PIA Control Register 0/1

The PIA Control Registers (Figure 10-1 and Figure 10-2) control the access to PIA Regions 0 through 5.

Figure 10-1 PIA Control Register 0 (PICT0, Address 80000020)

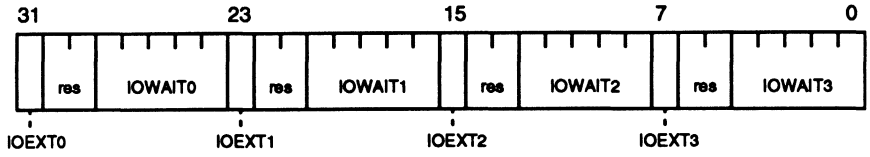
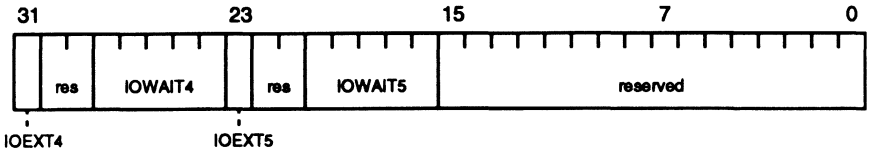


Figure 10-2 PIA Control Register 1 (PICT1, Address 80000024)



Bit 31: Input/Output Extend, Region 0 (IOEXT0)—If this bit is one, the end of a PIA access is extended by one cycle after $\overline{POE}$ is deasserted or by two cycles after $\overline{PWE}$ is deasserted. This provides one additional cycle of output disable time or data hold time for reads and writes, respectively.

Bits 30-29: Reserved.

Bits 28-24: Input/Output Wait States, Region 0 (IOWAIT0)—This field specifies the number of wait states taken by an access to PIA Region 0. An I/O read cycle takes at least three cycles (two wait states), and an I/O write cycle takes at least four cycles (three wait states). If the IOWAIT0 field specifies an insufficient number of wait states for an access (for example, IOWAIT0 = 00010 for a write), the processor takes the required minimum number of wait states instead of the specified number.

Other bits perform similar functions to IOEXT0 and IOWAIT0 for PIA Regions 1 through 5.

10.1.2 Initialization

The configuration of PIA regions, if present, must be set by software before PIA accesses are performed. Peripherals may be accessed using default parameters set by software to determine the presence and/or configuration of the peripherals.

10.2 PIA ACCESSES

PIA accesses are performed as a result of load and store instructions with an address within the range of PIA Region 0 (addresses 90000000 – 90FFFFFF) through PIA Region 5 (addresses 95000000 – 95FFFFFF). The PIA region number determines which of $\overline{\text{PIACS}}$(5–0) is asserted during the access. $\overline{\text{PIACS5}}$ is asserted for an access to PIA Region 5, and so on. The data width of the load or store determines the width of the access. An 8-bit device must be attached to ID(7–0), and a 16-bit device must be attached to ID(15–0). LOADM and STOREM instructions (possible only for 32-bit accesses) are performed as a series of simple loads or stores.

Instruction fetching from a PIA region is not supported.

10.2.1 Normal Access Timing

Figure 10-3 shows the timing of a PIA read cycle. The address is driven in the first cycle, the $\overline{\text{PIACS}}$x signal is asserted in the second cycle to allow for address setup, and the $\overline{\text{PIAOE}}$ signal is asserted in the third cycle to allow for chip select setup. The data must be valid after the number of cycles specified by IOWAITx+1. After sampling the data, the Am29200 microprocessor deasserts $\overline{\text{PIACS}}$x and $\overline{\text{PIAOE}}$. The interface operates such that the processor allows at least one cycle before it drives ID(31–0) for a new access (though a new address may be driven on A(23–0) immediately), providing one cycle for the peripheral to disable its drivers. If this cycle is insufficient, setting the IOEXTx bit for the region causes the processor to insert an additional cycle after the read before starting a new access.

Figure 10-4 shows timing of a PIA write cycle. The $\overline{\text{PIAOE}}$ signal is not asserted. Instead, the processor drives data in the second cycle and asserts the $\overline{\text{PIAWE}}$ signal in the third cycle to allow for address, data, and chip select setup. The $\overline{\text{PIAWE}}$ signal is deasserted one cycle before the final cycle to provide data hold time for the write. If one cycle of hold time is insufficient, setting the IOEXTx bit for the region causes the processor to insert an additional cycle of data hold time.

10.2.2 Use of $\overline{\text{WAIT}}$ to Extend I/O Cycles

The $\overline{\text{WAIT}}$ signal is used to extend the number of wait states beyond the number determined by the IOWAITx field. $\overline{\text{WAIT}}$ can be asserted during a read at any time before $\overline{\text{PIAOE}}$ is deasserted, and can be asserted during a write at any time before $\overline{\text{PIAWE}}$ is deasserted. In response to $\overline{\text{WAIT}}$, the processor extends the access until $\overline{\text{WAIT}}$ is deasserted. If $\overline{\text{WAIT}}$ is asserted within the appropriate amount of time, a normal read access ends on the cycle after $\overline{\text{WAIT}}$ is deasserted (Figure 10-5), and a normal write access ends on the second cycle after $\overline{\text{WAIT}}$ is deasserted, to provide data hold time (Figure 10-6). If IOEXTx=1, the processor waits one more cycle after a read access to begin a new access, and inserts one more cycle of data hold time after a write access.

Figure 10-3 PIA Read Cycle

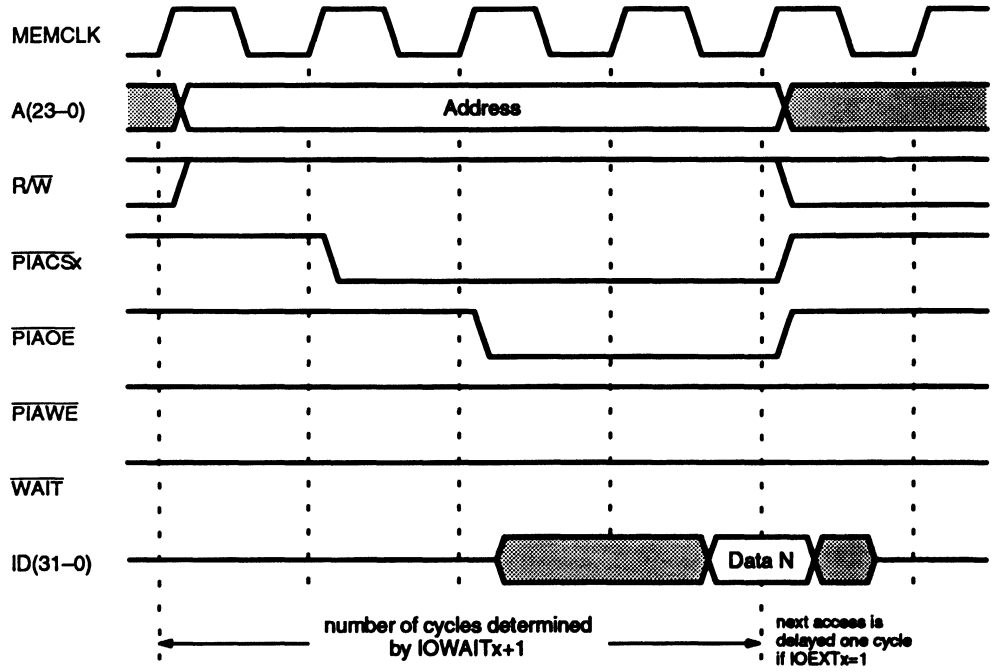


Figure 10-4 PIA Write Cycle

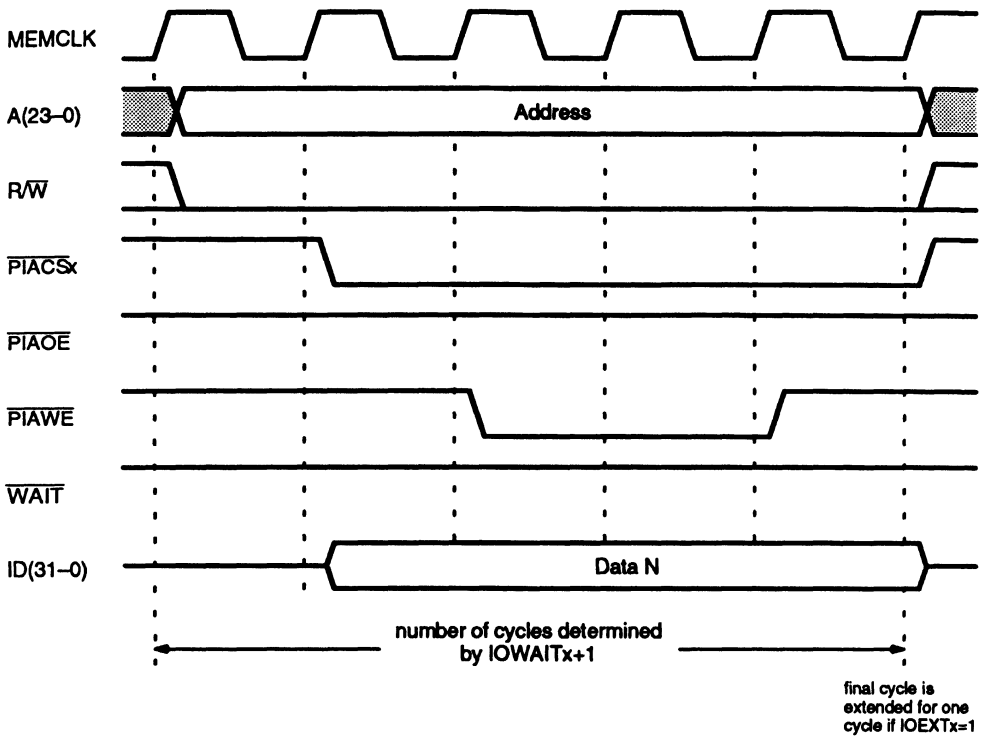


Figure 10-5 Extending a PIA Read Cycle with $\overline{\text{WAIT}}$

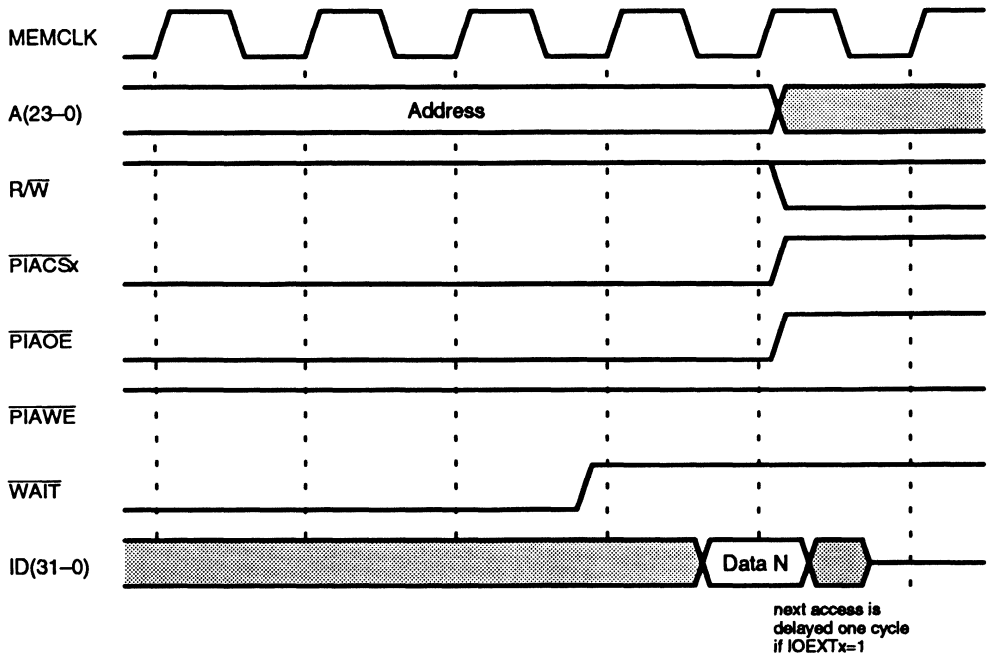
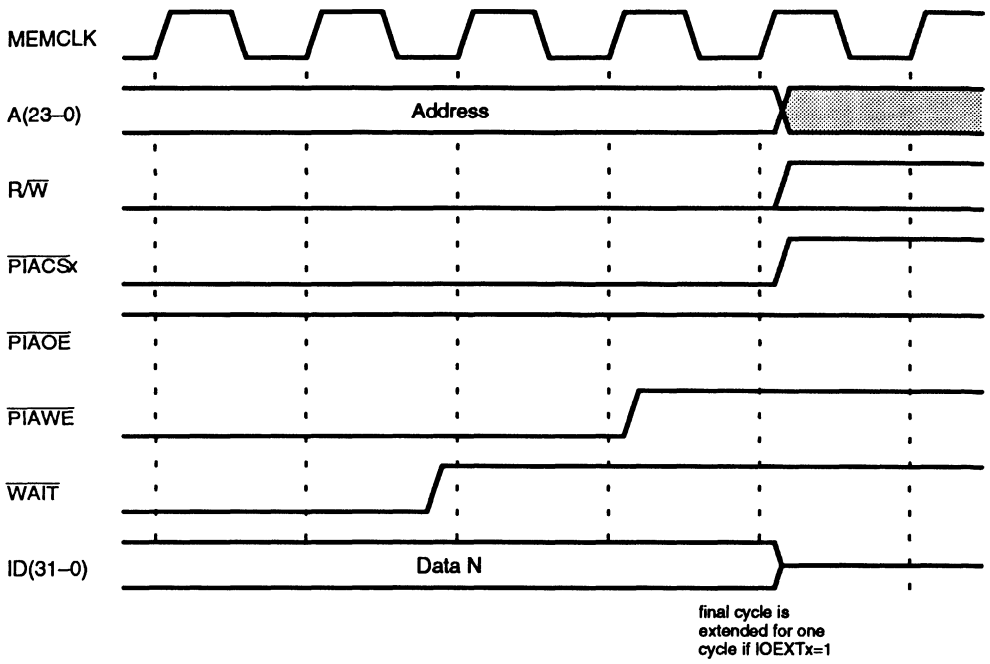


Figure 10-6 Extending a PIA Write Cycle with $\overline{WAIT}$



CHAPTER 11

DMA CONTROLLER

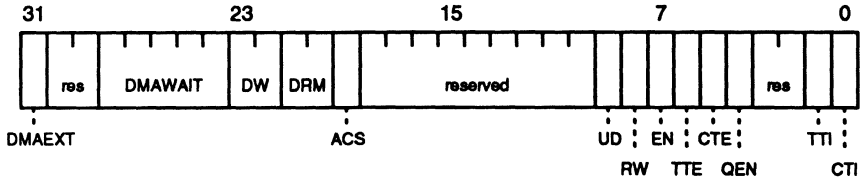
The Am29200 microprocessor has two DMA channels, each capable of performing either external or internal DMA. An external DMA transfers data between an external peripheral and DRAM. An internal DMA transfers data between an on-chip peripheral and DRAM. One of the DMA channels supports queued transfers. The DMA controller also supports direct DRAM access by an external device such as an external DMA controller.

11.1 PROGRAMMABLE REGISTERS

11.1.1 DMA0 Control Register (DMCT0, Address 80000030)

The DMA0 Control Register (Figure 11-1) controls DMA Channel 0.

Figure 11-1 DMA0 Control Register



Bit 31: DMA Extend (DMAEXT)—The DMAEXT bit serves a function very similar to the IOEXTx bits in the PIA Control registers. This bit is set to provide an additional cycle of output disable time for a read or an additional cycle of data hold time for a write.

Bits 30-29: Reserved.

Bits 28-24: DMA Wait States (DMAWAIT)—This field specifies the number of wait states taken by an external access by DMA Channel 0. An external DMA read cycle takes at least three cycles (two wait states) and an external DMA write cycles takes at least four cycles (three wait states). If the DMAWAIT field specifies an insufficient number of wait states for an access (for example, DMAWAIT = 00010 for a write), the processor takes the required minimum number of wait states instead of the specified number.

Bits 23-22: Data Width (DW)—This field indicates the width of the data transferred by the DMA Channel, as follows:

DW Value	DMA Transfer Width
00	32 bits
01	8 bits
10	16 bits
11	32 bits, address unchanged

The value DW=11 is used to repeatedly transfer a fixed pattern from a single DRAM location to a peripheral. For example, it can be used to transfer to a blank area of a printed page without requiring that a memory buffer be allocated for the blank area.

Bits 21-20: DMA Request Mode (DRM)—This field indicates how external DMA requests are signaled by DREQ0, as follows:

DRM Value	DREQ0 Request
00	Active Low
01	Active High
10	High-to-Low transition
11	Low-to-High transition

Bit 19: Assert Chip Select (ACS)—This bit controls whether the DMA channel asserts $\overline{\text{PIACS0}}$ during an external peripheral access. If the ACS bit is 1, the DMA channel asserts $\overline{\text{PIACS0}}$; if the ACS bit is 0, the DMA channel does not assert $\overline{\text{PIACS0}}$.

Bits 18-10: Reserved.

Bit 9: Transfer Up/Down (UD)—This bit controls the addressing of memory for the series of DMA transfers. If the UD bit is 1, the DMA address (in the DMA0 Address Register) is incremented after each transfer. If the UD bit is 0, the DMA address is decremented after each transfer. The amount by which the address is incremented or decremented is determined by the width of the transfer, as follows:

DW Value	Address Incr/Decr
00 (32 bits)	+/–4
01 (8 bits)	+/–1
10 (16 bits)	+/–2
11 (32 bits)	+/–0

Bit 8: Read/Write (RW)—This bit controls whether the DMA transfer is to or from the DRAM. If the RW bit is 1, the DMA channel transfers data from the DRAM to the peripheral. If the RW bit is 0, the DMA channel transfers data from the peripheral to the DRAM.

Bit 7: Enable (EN)—This bit enables the DMA channel to perform transfers. A 1 enables transfers, and a 0 disables transfers.

Bit 6: TDMA Terminate Enable (TTE)—This bit, when 1, causes the DMA channel to sample the TDMA signal during an external DMA transfer and to terminate the transfer if TDMA is asserted. TDMA does not apply to an internal transfer. If this bit is 0, the TDMA signal is ignored.

Bit 5: Count Terminate Enable (CTE)—This bit, when 1, causes the DMA channel to terminate the transfer when the DMACNT field of the DMA Count Register decrements past zero. If this bit is 0, the COUNT field does not terminate the DMA transfer, though the DMA channel still decrements the count after every transfer.

Bit 4: Queue Enable (QEN)—This bit, when 1, enables the DMA queuing feature (which is implemented only on DMA Channel 0). DMA Queuing allows the DMA0 Address Register and DMA0 Count Register to be reloaded automatically at the end of a DMA transfer from the DMA0 Address Tail Register and the DMA0 Count Tail Register, respectively. Queuing permits a second transfer to start immediately after a first transfer has terminated, greatly reducing the response-time requirement for software to set up and start the second transfer. When this bit is 0, DMA queuing is disabled, and the DMA0 Address Register and DMA0 Count Register are set directly to initiate a transfer.

Bits 3-2: Reserved.

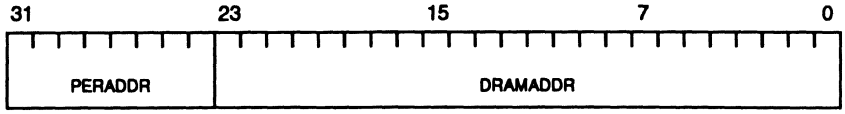
Bit 1: TDMA Terminate Interrupt (TTI)—The TTI bit is used to report that the DMA channel has generated an interrupt because of TDMA termination. If the TTE bit is one and the TDMA signal is asserted during an external DMA transfer, the TTI bit is set and a processor interrupt occurs.

Bit 0: Count Terminate Interrupt (CTI)—The CTI bit is used to report that the DMA channel has generated an interrupt because of count termination. If the CTE bit is one and the DMACNT field decrements past zero, the CTI bit is set and a processor interrupt occurs.

11.1.2 DMA0 Address Register (DMAD0, Address 80000034)

The DMA0 Address Register (Figure 11-2) contains the addresses for a transfer by DMA Channel 0.

Figure 11-2 DMA0 Address Register



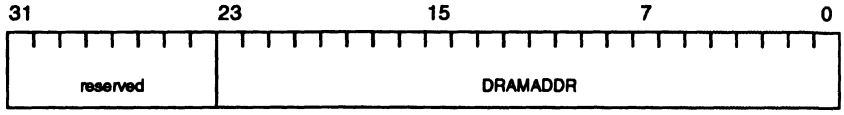
Bits 31-24: Peripheral Address (PERADDR)—This field specifies eight bits that are driven on A(7–0) during an external peripheral access by the DMA channel. A(23–8) are driven Low during the transfer.

Bits 23-0: DRAM Address (DRAMADDR)—This field contains the DRAM address for the next DMA transfer to or from the DRAM. The DRAMADDR field is incremented or decremented (based on the UD bit) by an amount determined by the width of the DMA transfer. The increment or decrement amount is 1 for a byte transfer, 2 for a halfword transfer, and 4 for a word transfer. To support repeated transfers from the same word, the address can be left unchanged. The DRAMADDR field wraps from the value 000000 to FFFFFF (hexadecimal) when decremented and from FFFFFF to 000000 when incremented.

11.1.3 DMA0 Address Tail Register (TAD0, Address 80000036)

This write-only register (Figure 11-3) is the tail of the DMA Channel 0 address queue, and is used to write the address of a queued transfer when the QEN bit is 1.

Figure 11-3 DMA0 Address Tail Register



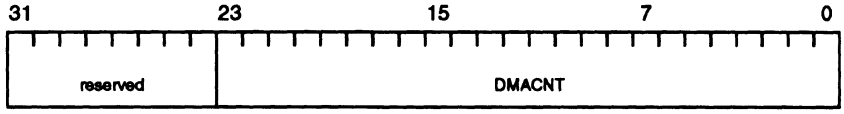
Bits 31-24: Reserved.

Bits 23-0: DRAM Address (DRAMADDR)—This field is written with the beginning DRAM address for a queued DMA transfer, if queuing is enabled.

11.1.4 DMA0 Count Register (DMCN0, Address 80000038)

The DMA0 Count Register (Figure 11-4) specifies the number of transfers remaining to be performed by DMA Channel 0.

Figure 11-4 DMA0 Count Register



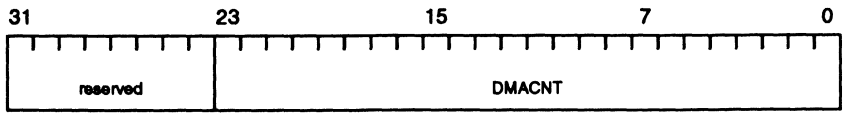
Bits 31-24: Reserved.

Bit 23-0: DMA Count (DMACNT)—This field normally specifies the number of transfers remaining to be performed on the DMA channel. The count is zero-based: a count of zero indicates one transfer, a count of one indicates two transfers, and so on. The DMA channel decrements the DMACNT field after every transfer. If the CTE bit is 1, the DMA channel generates an interrupt when the DMACNT field is decremented past zero. However, if the CTE bit is not 1, the DMACNT field is still decremented after every transfer and can be used to determine how many transfers have been performed when the DMA channel terminates because of the TDMA signal.

11.1.5 DMA0 Count Tail Register (TCN0, Address 8000003A)

This write-only register (Figure 11-5) is the tail of the DMA Channel 0 count queue, and is used to write the transfer count of a queued transfer when the QEN bit is 1.

Figure 11-5 DMA0 Count Tail Register



Bits 31-24: Reserved.

Bits 23-0: DMA Count (DMACNT)—This field is written with the zero-based number of transfers to be performed by a queued DMA transfer, if queuing is enabled.

11.1.6 DMA1 Control Register (DMCT1, Address 80000040)

The DMA1 Control Register controls DMA Channel 1. It is identical in layout and definition to the DMA0 Control Register, except that the ACS bit controls whether or not $\overline{\text{PIACS1}}$ is asserted and there is no QEN bit, since queuing is not implemented on DMA Channel 1.

11.1.7 DMA1 Address Register (DMAD1, Address 80000044)

The DMA1 Address Register contains the addresses for a transfer by DMA Channel 1. It is identical in layout and definition to the DMA0 Address Register.

11.1.8 DMA1 Count Register (DMCN1, Address 80000048)

The DMA1 Count Register specifies the number of transfers remaining to be performed by DMA Channel 1. It is identical in layout and definition to the DMA0 Count Register.

11.1.9 Initialization

The EN bits of both DMA Channel 0 and DMA Channel 1 are reset to 0 by a processor reset. The DMA channels must be configured by software before they are used.

11.2 DMA TRANSFERS

A DMA transfer is performed as a result of a DMA request. The DMA request may be generated either by an internal peripheral, or by an external device using DREQ(1–0). The direction of a DMA transfer is determined by the RW bit of the DMA Control Register.

If the RW bit is 0, the DMA channel transfers data from the peripheral to the DRAM. The DMA channel first performs an access to read the data from the peripheral and then performs a DRAM write to store the data into the DRAM. Both accesses occur without interruption: there is no other intervening access.

If the RW bit is 1, the DMA channel transfers data from the DRAM to the peripheral. The DMA channel first performs a DRAM read to access the data and then performs an internal or external access to write the data to the peripheral. Both accesses occur without interruption: there is no other intervening access.

The details of DMA transfers to and from the internal peripherals are unimportant to users. External DMA transfers appear very much like PIA accesses, except the DMA acknowledge signals $\overline{\text{DACK}}$(1–0) are asserted during the transfer as well as, optionally, $\overline{\text{PIACS}}$(1–0). The address bus is driven with an address derived from the DMA Address Register. Bits 23–8 of the address are all 0s, and bits 7–0 are driven with the PERADDR field. It is possible to use the $\overline{\text{DACK}}$(1–0) signals as chip selects to the DMA peripherals. The signals $\overline{\text{PIAOE}}$, $\overline{\text{PIAWE}}$, and $\overline{\text{WAIT}}$ are used as they are during a PIA access. The DMAWAIT field is used to determine the number of wait states, much as the IOWAITx field is used during a PIA access.

If the DRAM is 16 bits wide, a 32-bit DMA DRAM access appears as two 16-bit accesses on ID(31–16). If the peripheral is 8 or 16 bits wide, a DMA peripheral access appears as a single access on ID(7–0) or ID(15–0), respectively. The peripheral must have the same width as the transfer.

Figure 11-6 shows the timing of a DMA read cycle (performed when the RW bit is 0). The $\overline{\text{DACK}}$x signal (and, optionally, the $\overline{\text{PIACS}}$x signal) is asserted in the second cycle, and the $\overline{\text{PIAOE}}$ signal is asserted in the third cycle. The data must be valid after the number of cycles determined by DMAWAIT. If DMAEXT=1, the processor waits one more cycle after the read access to begin a new access.

Figure 11-7 shows timing of a DMA write cycle (performed when the RW bit is 1). The $\overline{\text{PIAOE}}$ signal is not asserted. Instead, the processor drives data in the second cycle and asserts the $\overline{\text{PIAWE}}$ signal in the third cycle. The $\overline{\text{PIAWE}}$ signal is deasserted one cycle before the final cycle (the number of cycles is determined by DMAWAIT) to provide data hold time. If DMAEXT=1, the processor inserts one more cycle of data hold time after a write access.

Figure 11-6 DMA Read Cycle

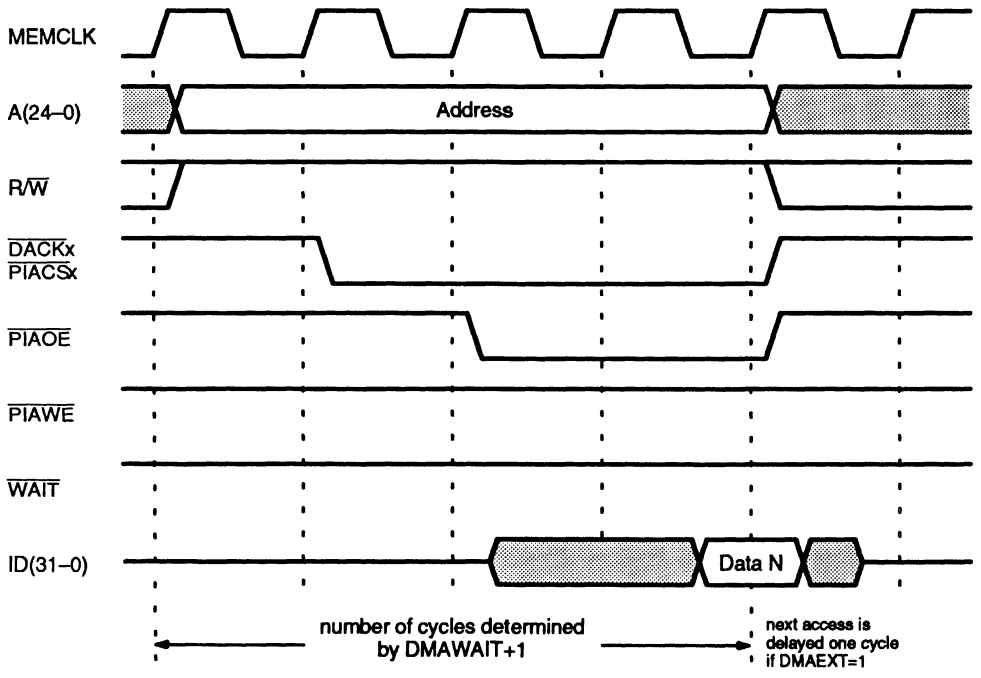
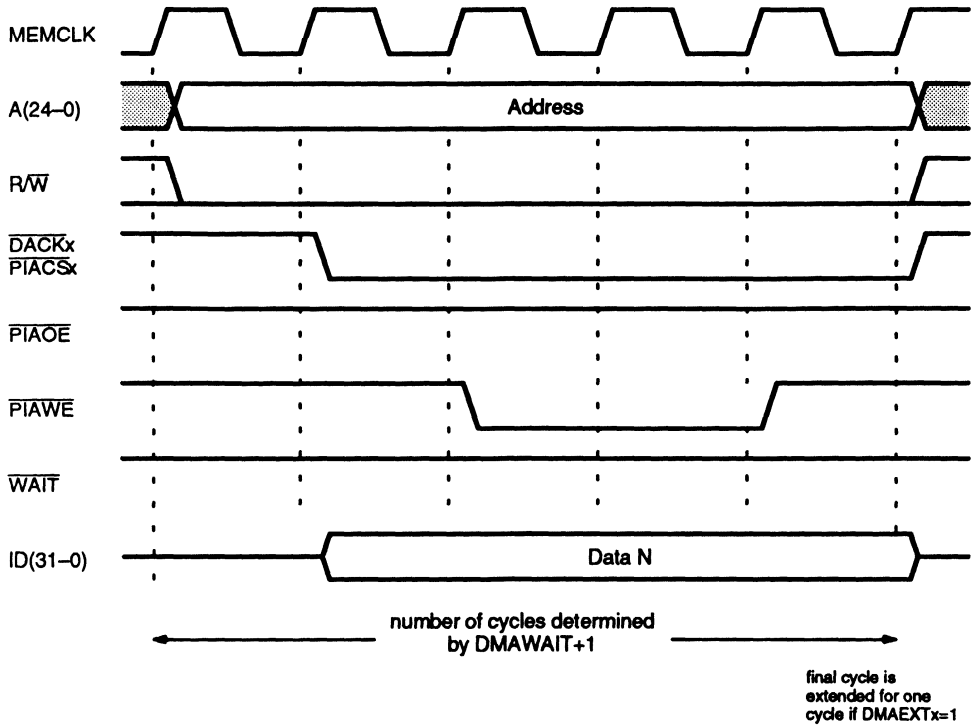


Figure 11-7 DMA Write Cycle



If the DMA channel's TTE bit is 1, an external peripheral can assert TDMA at any time while DACKx is asserted to terminate the transfer after the current access; in this case, the current access is completed as usual. As with PIA accesses, the peripheral can use WAIT to extend the access.

The generation of DMA requests by the DREQ(1–0) signals is controlled by the DRM field of the DMA control register. The DMA requests can be programmed individually to be edge- or level-sensitive for either polarity of edge or level.

If the DMA request is edge-sensitive, the DMA request signal must remain at the appropriate level for at least four cycles after the active edge to insure that the DMA channel detects the request. An active edge that occurs during an in-progress transfer (that is, while DACKx is asserted) is ignored. The DREQx signal must be Low (rising-edge-triggered) or High (falling-edge-triggered) for four cycles before a new active edge can be recognized.

If the DMA request is level-sensitive, the request may be deasserted at any time while DACKx is asserted, and must be deasserted during the cycle in which DACKx is deasserted unless it is desired to generate a subsequent DMA request.

The DMA channel continues to perform transfers until the count expires or the TDMA input is asserted (depending on the CTE and TTE bits). When the transfer terminates, the EN bit is reset unless there is an active queued transfer, as explained in Section 11.3.

11.3 DMA QUEUING (DMA CHANNEL 0 ONLY)

The address and count registers for DMA Channel 0 each consist of a two-entry queue, with each entry of the queue separately addressable for loading a new transfer. The DMA0 Address Register and DMA0 Count Register are at the head of the queue. The DMA0 Address Tail Register and DMA0 Count Tail Register are at the tail of the queue, and are write-only registers. A DMA transfer queued behind an active transfer can start as soon as the first transfer is complete. This reduces the response-time requirement for software to load a new transfer: software has the entire transfer time of the second transfer to load the next transfer at the tail of the queue.

DMA queuing is enabled by writing the appropriate address and count values at the head of the queue, then setting the DMA0 Control Register appropriately, with EN=1, QEN=0, and CTE/TTE=1.

A transfer is loaded into the tail of the queue by first loading the DMA0 Count Tail Register, then loading the DMA0 Address Tail Register (note that the PERADDR field cannot be changed by a queued transfer). Writing the tail address causes the QEN bit to be set. Whenever a DMA transfer terminates at the head of the queue and the QEN bit is 1, the transfer at the tail of the queue advances to the head of the queue and begins immediately. When the queued transfer advances to the head of the queue, the QEN bit is reset, the EN bit remains set, and the CTI/TTI bit is set (note that the automatic queue advance makes it impossible to inspect the count of the former transfer after a TTI interrupt in order to discover how many transfers were performed by that transfer).

The CTI/TTI interrupt handler need not clear the CTI/TTI bit, in fact, it is unsafe to write the DMA0 Control Register at this point because the termination of the current transfer (the transfer that was formerly queued) may be lost. The interrupt handler need only place the count and address of the next transfer at the tail of the queue (again, the tail address should be loaded after the count, because writing the tail address sets the QEN bit and enables the queue to advance). The CTI/TTI bit is automatically reset when the tail address is written.

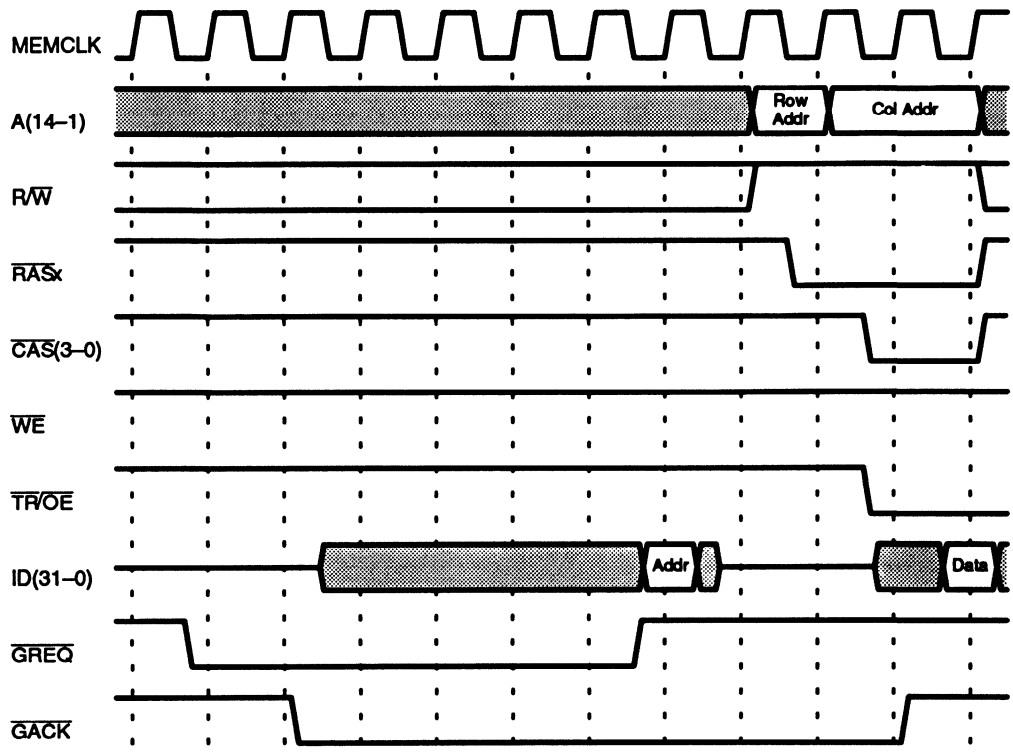
Queue underflow occurs if the transfer at the head of the queue terminates before the next transfer is loaded at the tail of the queue. Software can detect that underflow has occurred by examining the EN bit after setting up the next transfer. If the EN bit is 0, underflow has occurred, because a successful start of a queued transfer causes the EN bit to remain set when the termination interrupt is generated.

11.4 RANDOM DIRECT MEMORY ACCESS BY EXTERNAL DEVICES

The Am29200 microprocessor is designed primarily for single-controller applications, and it has no provision for other bus masters to control the address and data buses in the traditional sense. However, the DMA controller does provide a mechanism for an external device to access the ROM or DRAM using addresses provided by the device rather than by a DMA channel. External devices use the $\overline{\text{GREQ}}$ and $\overline{\text{GACK}}$ signals to perform a random memory access via the Am29200 DRAM or ROM controller.

Figure 11-8 shows the timing for a memory read using $\overline{\text{GREQ}}$ and $\overline{\text{GACK}}$. The external device indicates that it wants to perform a memory access by asserting $\overline{\text{GREQ}}$. As soon as the processor can perform the access, it asserts $\overline{\text{GACK}}$. The external device can place the memory address on ID(31–0) during any cycle following the assertion of $\overline{\text{GACK}}$: the device indicates that the address is valid by deasserting $\overline{\text{GREQ}}$. The processor uses this address to determine whether the access is to ROM or DRAM (according to the normal address allocation) and performs the required access. Figure 11-8 shows an access to DRAM, as an example. The processor

Figure 11-8 External Random DRAM Read Cycle



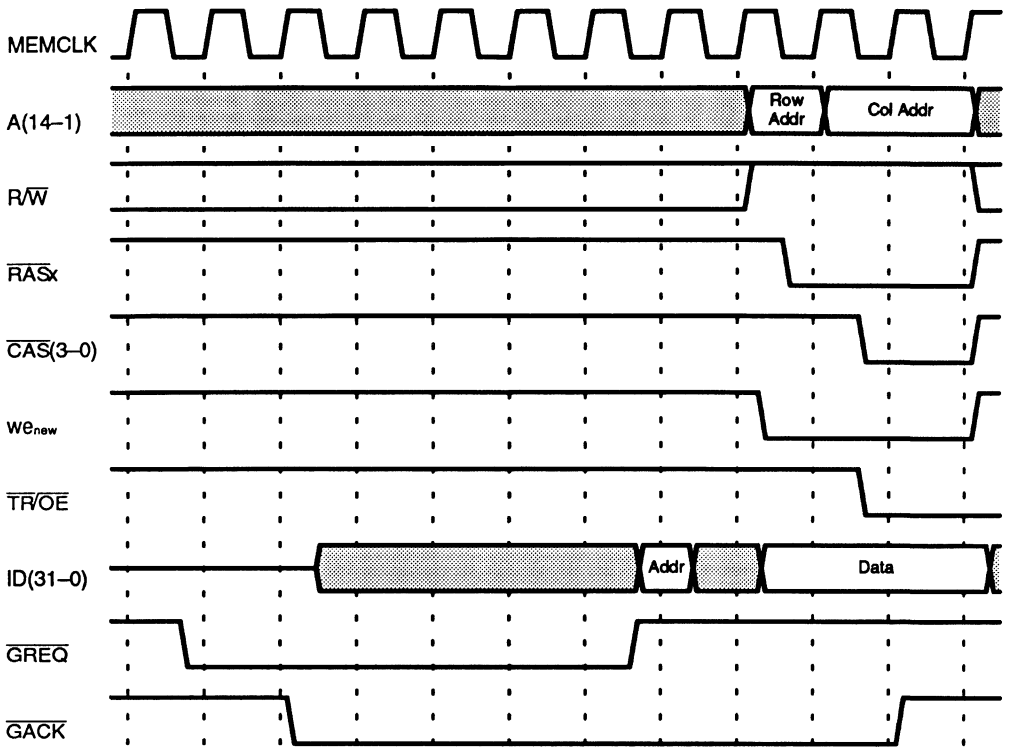
deasserts GACK at the beginning of the cycle in which the data is valid on ID(31–0). The deassertion of GACK completes the access.

Figure 11-9 illustrates how the GREQ/GACK protocol can be used to perform a memory write. In this case, the external device supplies the address upon the deassertion of GREQ and then provides the write data on ID(31–0). The processor does not distinguish between a read and a write, allowing the ID Bus to be available to the device for the transfer of both address and data. The distinction between reads and writes must be made by external logic (which, for example, forms the signal WE_{new} in Figure 11-9) in a way that meets the memory timing requirements. For example, an AND gate can be used to form the negative OR of the processor's WE signal and the write enable from the external device.

To summarize the use of GREQ and GACK:

1. The external device asserts GREQ to request an access.
2. Following the assertion of GACK, the device places the address on ID(31–0) and deasserts GREQ to indicate that the address is valid.
3. For a read, the device must be able to latch data from ID(31–0) at the end of the cycle in which GACK is deasserted. For a write, the device must be prepared to drive data on ID(31–0) on the second cycle following the address transfer and must hold the data valid until the cycle following the deassertion of GACK, at which time it must stop driving. The device must also supply a write enable signal that satisfies the timing requirements of the memory. In either case, the processor deasserts GACK based on the access timing of the ROM or DRAM.

Figure 11-9 External Random DRAM Write Cycle

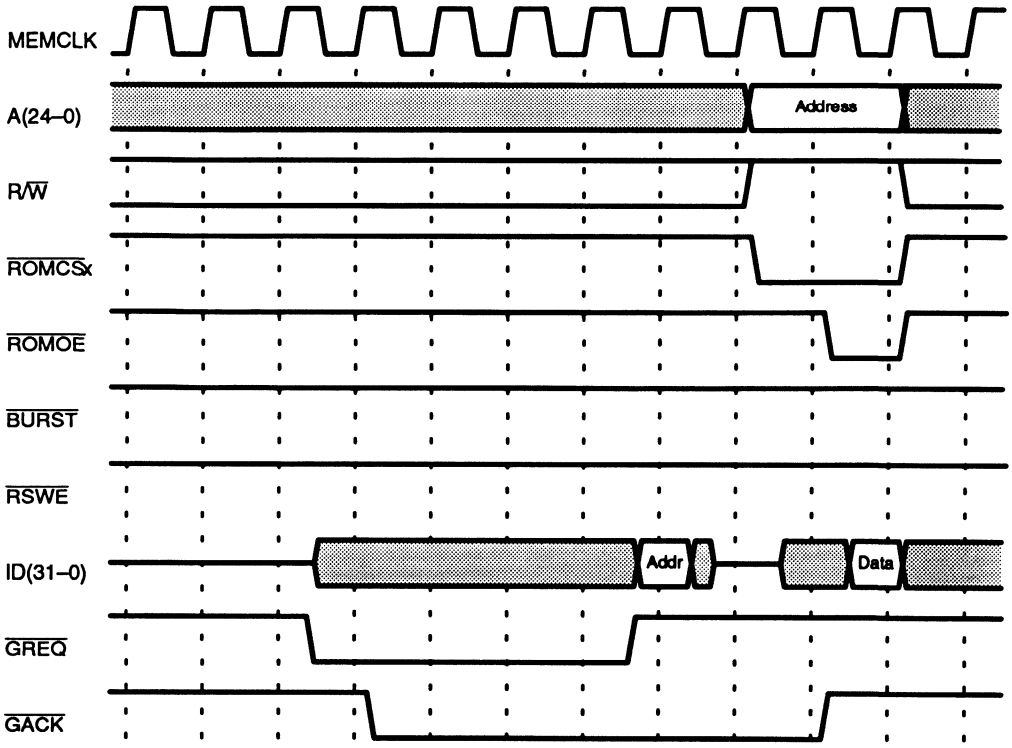


To further clarify the use of GREQ and GACK, Figure 11-10 shows example timing for a ROM read. Writes to the ROM space are more difficult to implement than DRAM writes because the processor always asserts the ROMOE signal.

Memory accesses using GREQ and GACK are restricted to 32-bit accesses: 8- and 16-bit accesses are not supported. Zero-wait-state accesses are also not supported. Furthermore, the ROM and/or DRAM bank must be 32 bits wide. Although the GREQ/GACK protocol supports full 32-bit addressing, the addresses supplied must be within the range of ROM or DRAM addresses. DRAM mapping cannot be performed.

During a processor reset, the GREQ input may be used by a hardware-development system to force processor outputs to the high-impedance state. To prevent driver conflicts, the system should keep GREQ in a high-impedance state during a processor reset.

Figure 11-10 External Random ROM Read Cycle



CHAPTER 12

PROGRAMMABLE I/O PORT

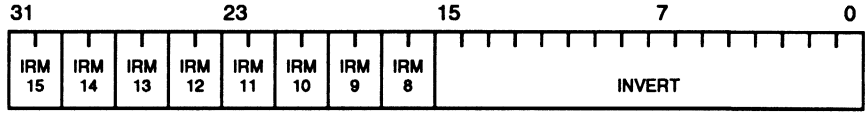
The I/O Port permits direct programmable access to the sixteen external signals, PIO(15–0), as either inputs, outputs, or open-drain signals. Eight of these signals, PIO(15–8), can be programmed to cause interrupts.

12.1 PROGRAMMABLE REGISTERS

12.1.1 PIO Control Register (POCT, Address 800000D0)

The PIO Control Register (Figure 12-1) controls interrupt generation and determines the polarity of PIO(15–0).

Figure 12-1 PIO Control Register



Bits 31-30: Interrupt Request Mode, PIO15 (IRM15)—This field enables PIO15 to generate an interrupt equivalent to a request on the processor's $\overline{\text{INTR3}}$ input, and indicates whether PIO15 is level- or edge-sensitive in generating the interrupt. The IRM15 field controls PIO15 as follows:

IRM15 Value	PIO15 Interrupt
00	Interrupt disabled
01	Level-sensitive
10	Edge-sensitive
11	IRM15 only – see below

The INVERT field (see below) further conditions interrupt generation. If the INVERT bit for PIO15 is 0, an interrupt, if enabled, is generated by a High level on PIO15 (level-sensitive) or on a Low-to-High transition (edge-sensitive) of PIO15. If the INVERT bit for PIO15 is 1, an interrupt, if enabled, is generated by a Low level on PIO15 (level-sensitive) or on a High-to-Low transition (edge-sensitive) of PIO15.

For IRM15, the value 11 causes PIO15 to generate an edge-triggered interrupt and to also set the FBUSY bit in the Parallel Port Control Register (see Section 13.1.1), causing the $\overline{\text{PBUSY}}$ output to be asserted. This can be used to support certain system-specific features of the Parallel Port. Note that this value may cause a spurious setting of FBUSY during a reset, depending on the activity on PIO15 after a reset.

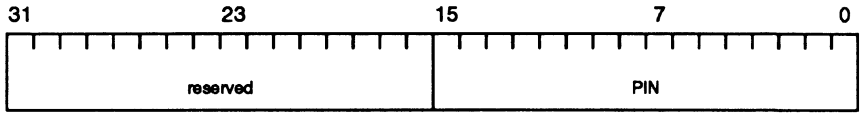
Bits 29-16: IRM14 through IRM8—The IRM14 through IRM8 fields enable interrupts and specify level- or edge-sensitivity for PIO14 through PIO8, respectively. These fields are identical in definition to IRM15, except that the value 11 is reserved.

Bits 15-0: PIO Inversion (INVERT)—This field determines how the level on each PIO signal is reflected in the PIO Input and PIO Output Registers, and how interrupts are generated. The most significant bit of the INVERT field determines the sense of PIO15, the next bit determines the sense of PIO14, and so on. A 0 in this field causes the internal and external sense of the respective PIO signal to be noninverted; a High external level is reflected as a 1 internally, and a Low is reflected as a 0 internally. A 1 in this field causes the internal and external sense of the respective PIO signal to be inverted; a High external level is reflected as a 0 internally, and a Low is reflected as a 1 internally.

12.1.2 PIO Input Register (PIN, Address 800000D4)

The PIO Input Register (Figure 12-2) reflects the external levels on the PIO(15–0) signals.

Figure 12-2 PIO Input Register



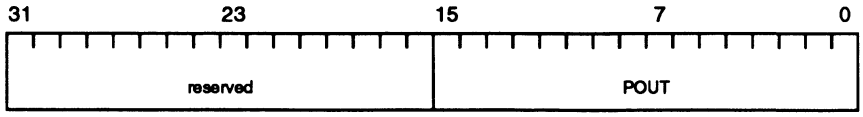
Bits 31-16: Reserved.

Bits 15-0: PIO Input (PIN)—This field reflects the levels on each PIO signal. The most significant bit of the PIN field reflects the level on PIO15, the next bit reflects the level on PIO14, and so on. The correspondence between levels and bits in this register is controlled by the INVERT field.

12.1.3 PIO Output Register (POUT, Address 800000D8)

The PIO Output Register (Figure 12-3) determines the levels driven on the PIO(15–0) signals, for those signals enabled to be driven by the PIO Output Enable Register.

Figure 12-3 PIO Output Register



Bits 31-16: Reserved.

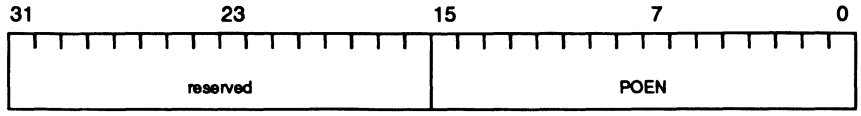
Bits 15-0: PIO Output (POUT)—This field determines the levels on each PIO signal, if so enabled by the PIO Output Enable Register. The most significant bit of the POUT field determines the level on PIO15, the next bit determines the level on PIO14,

and so on. The correspondence between levels and bits in this register is controlled by the INVERT field.

12.1.4 PIO Output Enable Register (POEN, Address 800000DC)

The PIO Output Enable Register (Figure 12-4) determines whether or not the PIO(15–0) signals are driven as outputs.

Figure 12-4 PIO Output Enable Register



Bits 31-16: Reserved.

Bits 15-0: PIO Output Enable (POEN)—This field determines whether each PIO signal is driven as an output. The most significant bit of the POEN field determines whether PIO15 is driven, the next bit determines whether PIO14 is driven, and so on. A 1 in a bit position enables the respective signal to be driven according to the associated POUT and INVERT bits, and a 0 disables the signal as an output.

12.1.5 Initialization

During a processor reset, all bits of the PIO Output Enable Register are reset to 0, disabling all PIO signals as outputs. The I/O Port must be initialized by software before the I/O Port is used.

12.2 OPERATING THE I/O PORT

The PIO(15–0) signals are asynchronous to the processor. A change on PIO(15–0) is reflected in the PIO Input Register a maximum of four MEMCLK cycles after the change occurs. A level-sensitive interrupt occurs four cycles after the change, and an edge-sensitive interrupt occurs five cycles after the change. When driven as an output, a change to the PIO Output Register is reflected on PIO(15–0) a maximum of one cycle after the change occurs. The PIO(15–0) signals have additional metastable hardening, allowing them to be driven with slow-transition-time signals.

The PIO Output Enable Register permits the PIO signals to be operated as open-drain outputs. This is accomplished by keeping the appropriate POUT bits constant and writing data into the POEN field, so the output is either driving Low or is disabled, depending on the data.

CHAPTER 13

PARALLEL PORT

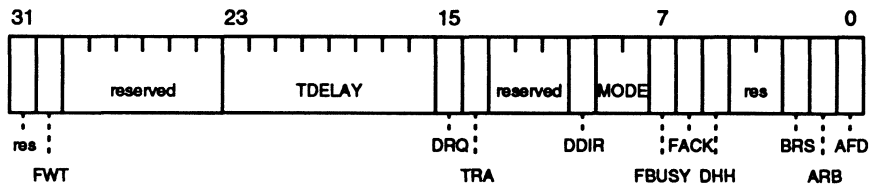
The Parallel Port connects a host processor to the Am29200 microprocessor. It supports data transfers from the host to the Am29200 microprocessor or from the Am29200 microprocessor to the host.

13.1 PROGRAMMABLE REGISTERS

13.1.1 Parallel Port Control Register (PPCT, Address 800000C0)

The Parallel Port Control Register (Figure 13-1) controls the Parallel Port.

Figure 13-1 Parallel Port Control Register



Bit 31: Reserved.

Bit 30: Full Word Transfer (FWT)—The Parallel Port is normally configured to transfer 8 bits at a time from the Parallel Port Data Register, and FWT is normally 0. When the FWT bit is 1, the Parallel Port is configured to transfer 32-bit words from the Parallel Port Data Register, reducing the demand the Parallel Port places on the processor. An FWT value of 1 causes the Parallel Port to generate an interrupt or DMA request for every fourth handshake. For proper transfer of data, external logic must assemble bytes from the parallel-port interface into the 32-bit external latch that implements the Parallel Port Data Register. The DMA transfer or load instruction that reads the Parallel Port Data Register must indicate a data width of 32 bits. Full word transfers are implemented only for transfers from the host.

Bits 29-24: Reserved.

Bits 23-16: Transfer Delay (TDELAY)—During a transfer from the host, this field controls the duration of the assertion of PACK (and possibly $\overline{\text{PBUSY}}$). During a transfer to the host, it controls the duration of data setup, PACK assertion, and data hold times. The TDELAY field specifies the number of MEMCLK cycles in the duration interval.

Bit 15: Data Request (DRQ)—This bit is set to indicate that the Parallel Port is ready for data to be read from or written to the Parallel Port Data Register. If so enabled by the MODE field, this bit being 1 generates an interrupt or DMA request to read or write data. This bit is reset when the Parallel Port Data Register is read or written. The

DRQ bit is read-only, allowing other bits of the Parallel Port Control Register to be set (for example, the FACK bit) without interfering with the data request.

Bit 14: Transfer Active (TRA)—This bit is set at the beginning of a transfer on the Parallel Port and reset at the end of a transfer. It is read-only, so that setting other bits of the Parallel Port Control Register do not interfere with the indication of an active request. The TRA bit can be inspected by software to detect that a transfer is hung.

Bits 13-11: Reserved.

Bit 10: Data Direction (DDIR)—This bit controls the direction of data transfer on the Parallel Port. If the DDIR bit is 0 (the default), data is received on the Parallel Port. If the DDIR bit is 1, data is transmitted on the Parallel Port. The MODE field must be 00 when the DDIR bit is changed.

Bits 9-8: Parallel Port Mode (MODE)—This field enables the Parallel Port and controls the operational mode of the Parallel Port, as follows:

MODE Value	Effect on Parallel Port
00	Disabled
01	Generate interrupt requests for service
10	Generate DMA Channel 0 requests
11	Generate DMA Channel 1 requests

Requests for service are requests to read or write the Parallel Port Data Register. Placing the Parallel Port into the disabled state causes all internal state machines to be reset, forces PACK Low, and holds the Parallel Port in an idle state. Parallel Port programmable registers are not affected when the port is disabled.

Bit 7: Force Busy (FBUSY)—A 1 in this bit forces an active level on the $\overline{\text{PBUSY}}$ output. A 0 allows the $\overline{\text{PBUSY}}$ signal to operate normally.

Bit 6: Force ACK (FACK)—A 1 in this bit forces an active level on the PACK output for one TDELAY interval. At the end of the interval, the FACK bit is reset and PACK is deasserted.

Bit 5: Disable Hardware Handshake (DHH)—A 1 in this bit prevents the Parallel Port interface logic from controlling PACK or $\overline{\text{PBUSY}}$. A 0 in this bit permits normal handshaking with PACK and $\overline{\text{PBUSY}}$. FACK and FBUSY may be used by software to control PACK and $\overline{\text{PBUSY}}$ regardless of the DHH bit.

Bits 4-3: Reserved.

Bit 2: BUSY Relationship to STROBE (BRS)—This bit controls the relative timing of the $\overline{\text{PBUSY}}$ and PSTROBE hardware handshaking when the Parallel Port is receiving data. If BRS=0, $\overline{\text{PBUSY}}$ is asserted on the Low-to-High transition (leading edge) of PSTROBE. If BRS=1, $\overline{\text{PBUSY}}$ is asserted on the High-to-Low transition (trailing edge) of PSTROBE. The Parallel Port does not respond to PSTROBE until $\overline{\text{PBUSY}}$ is asserted, except that the TRA bit is always set on the leading edge of PSTROBE.

Bit 1: ACK Relationship to BUSY (ARB)—This bit controls the relative timing of the PACK and $\overline{\text{PBUSY}}$ handshaking when the Parallel Port is receiving data.

If ARB=0, $\overline{\text{PBUSY}}$ and PACK are asserted and deasserted at the same time (except for output driver skew). Both PACK and $\overline{\text{PBUSY}}$ are asserted at either the leading or trailing edge of PSTROBE, as controlled by the BRS bit. Both are deasserted together at the end of a transfer, which is usually at the end of a TDELAY interval.

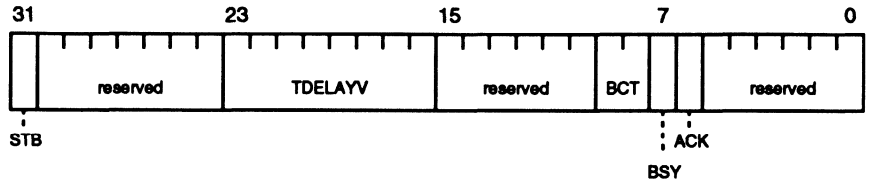
If ARB=1, the PACK pulse follows the $\overline{\text{PBUSY}}$ pulse in time. $\overline{\text{PBUSY}}$ is asserted in response to an assertion of PSTROBE and is deasserted when the Parallel Port Data Register has been read and PSTROBE is Low. PACK is asserted at the same time $\overline{\text{PBUSY}}$ is deasserted and is deasserted at the end of a TDELAY interval.

Bit 0: Autofeed (AFD)—This bit reflects the level on the PAUTOFD input. A 1 indicates PAUTOFD is active (High), and a 0 indicates PAUTOFD is inactive (Low).

13.1.2 Parallel Port Status Register (PPST, Address 800000C1)

The Parallel Port Control Register (Figure 13-2) controls the Parallel Port.

Figure 13-2 Parallel Port Status Register



Bit 31: PSTROBE Level (STB)—This bit indicates the level on the PSTROBE signal. If PSTROBE is Low, this bit is 0; if PSTROBE is High, this bit is 1.

Bits 30-24: Reserved.

Bits 23-16: TDELAY Counter Value (TDELAYV)—This field indicates the current value of the TDELAY counter used to time transitions of the handshaking signals. This value changes as the TDELAY interval is being timed.

Bits 15-10: Reserved.

Bits 9-8: Byte Count (BCT)—When the FWT bit is 1, this field indicates the number of bytes (that is, the number of complete handshakes) received on the Parallel Port since the most recent data request. This information is useful for handling partial-word transfers at the end of a block transfer.

Bit 7: $\overline{\text{PBUSY}}$ Level (BSY)—This bit indicates the level on the $\overline{\text{PBUSY}}$ signal. If $\overline{\text{PBUSY}}$ is Low, this bit is 0; if $\overline{\text{PBUSY}}$ is High, this bit is 1.

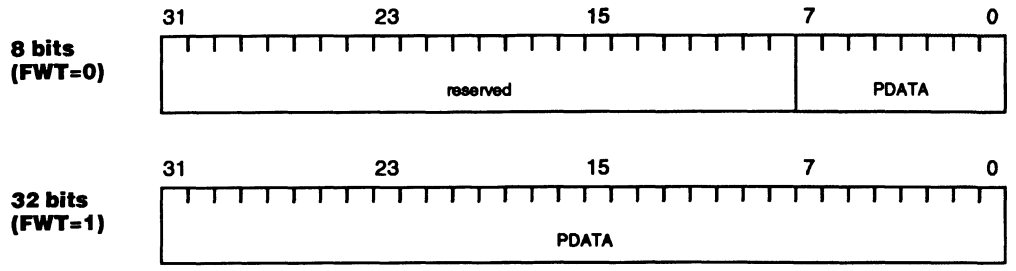
Bit 6: PACK Level (ACK)—This bit indicates the level on the PACK signal. If PACK is Low, this bit is 0; if PACK is High, this bit is 1.

Bits 5-0: Reserved.

13.1.3 Parallel Port Data Register (PPDT, Address 800000C4)

The Parallel Port Data Register (Figure 13-3) is used to read from and write data to the Parallel Port. This register is not implemented directly on the processor, but rather is implemented by an external data buffer connected to the parallel-port interface cable. The processor converts an access of this register into an external access of the data buffer. This access is similar to a PIA access, except the timing is fixed (see Section 13.2) and the access uses the signals $\overline{\text{POE}}$ and $\overline{\text{PWE}}$ to read and write the buffer.

Figure 13-3 Parallel Port Data Register



Bits 7-0 (8-bit transfers) or

Bits 31-0 (32-bit transfers): Parallel Port Data (PDATA)—This field contains the data being transferred to the Am29200 microprocessor or to the host over the Parallel Port. For transfers from the host, the width of this field depends on the setting of the FWT bit in the Parallel Port Control Register; however, the instruction or DMA channel that reads the Parallel Port must also specify the correct data width to properly read the Parallel Port Data Register.

13.1.4 Initialization

During a processor reset, the MODE field of the Parallel Port Control Register is reset to 00. The Parallel Port must be configured by software before the Parallel Port is enabled.

Writing the value 00 into the MODE field resets the Parallel Port, forces PACK Low, and forces $\overline{\text{PBUSY}}$ High (unless FBUSY is set).

The I/O Port signal PIO15 may be used by the host to signal a change in the configuration of the Parallel Port. If the IRM15 field of the PIO Control Register has the value 11 (see Section 12.1.1), PIO15 causes an edge-triggered interrupt and causes the FBUSY bit to be set. Setting the FBUSY bit causes the Parallel Port to appear busy to the host while the port's configuration is changed. The FBUSY bit must be reset by software (if required) once configuration is complete.

13.2 PARALLEL PORT TRANSFERS

The Parallel Port does not attach directly to the Am29200 microprocessor, but is attached to the interface cable via buffers. Data must be latched in the interface, using a three-state latch such as a 74LS374. The handshaking signals, PSTROBE, PAUTOFD, PACK, and $\overline{\text{PBUSY}}$, are connected to the Am29200 microprocessor via simple interface circuits. The inputs PSTROBE and PAUTOFD should be connected to the processor via a Schmitt-trigger inverter such as a 74HCT14, and the outputs PACK and $\overline{\text{PBUSY}}$ should be connected to the host via an open-collector inverter such as a 7406.

The hardware handshaking described in this section can be disabled by setting the DHH bit. If the DHH bit is 1, handshaking can be accomplished by software using the FACK and FBUSY bits.

13.2.1 Transfers from the Host

Figure 13-4 shows the state-transition diagram for transferring data from the host to the Am29200 microprocessor over the Parallel Port. Figure 13-5 through Figure 13-8 show the timing diagrams for these transfers. The timing diagrams differ in the settings of the BRS and ARB bits. The timing diagrams also show the signals as they appear at the processor interface, and do not reflect the inversions in the buffers to the parallel-port connector.

The host begins the transfer by placing data on the interface and asserting the PSTROBE signal. The data is latched in the interface on the rising edge of PSTROBE if BRS=0, and can be latched by either edge if BRS=1. The TRA bit is set on the leading edge of PSTROBE.

Figure 13-4 State Transistions for Transfers From the Host

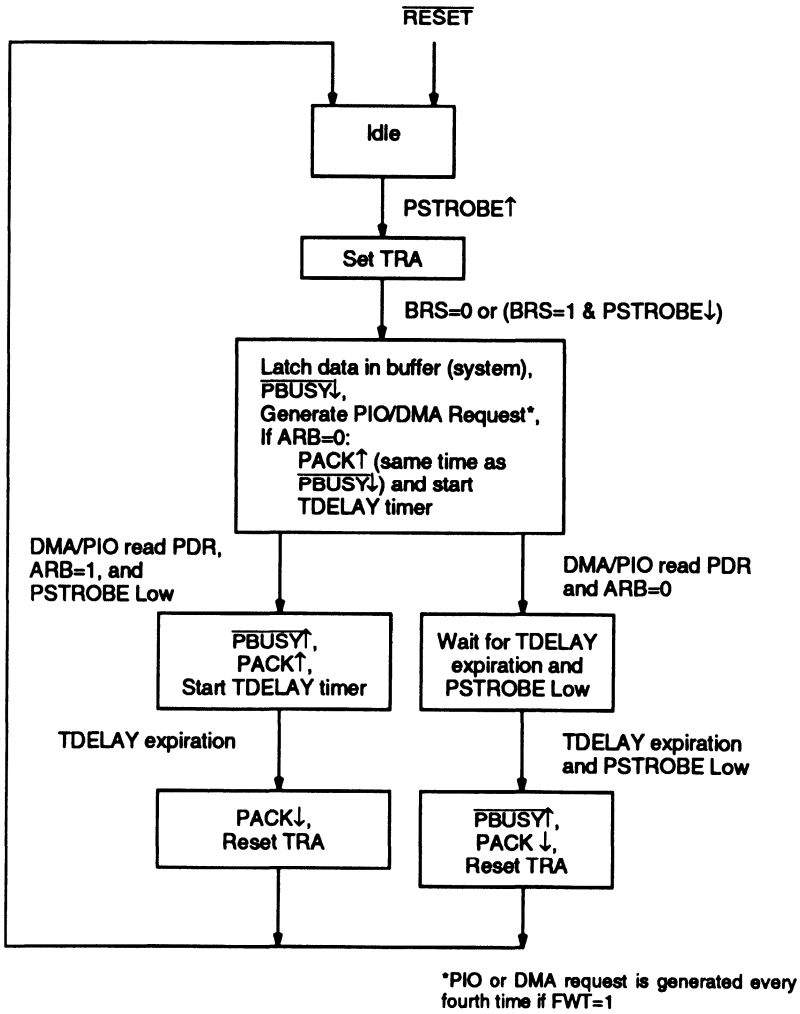


Figure 13-5 Transfer from the Host on the Parallel Port (BRS=0, ARB=0)

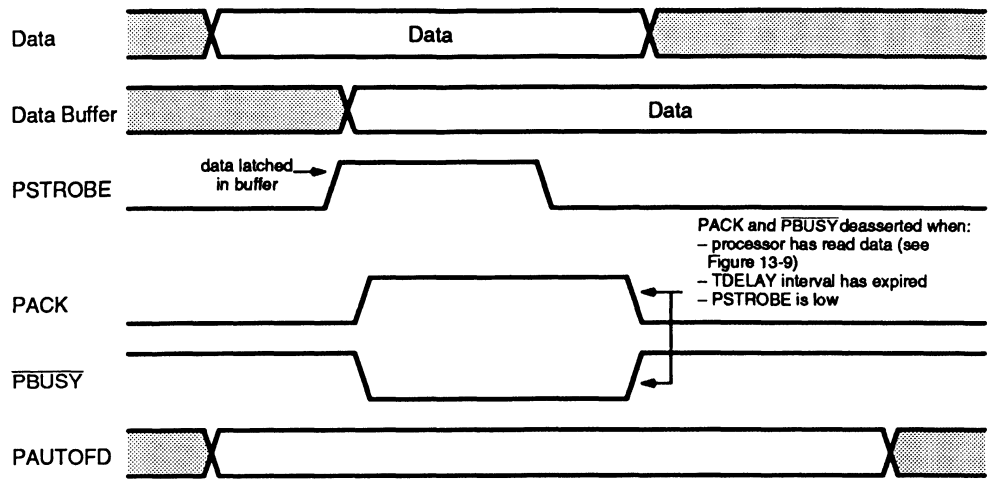
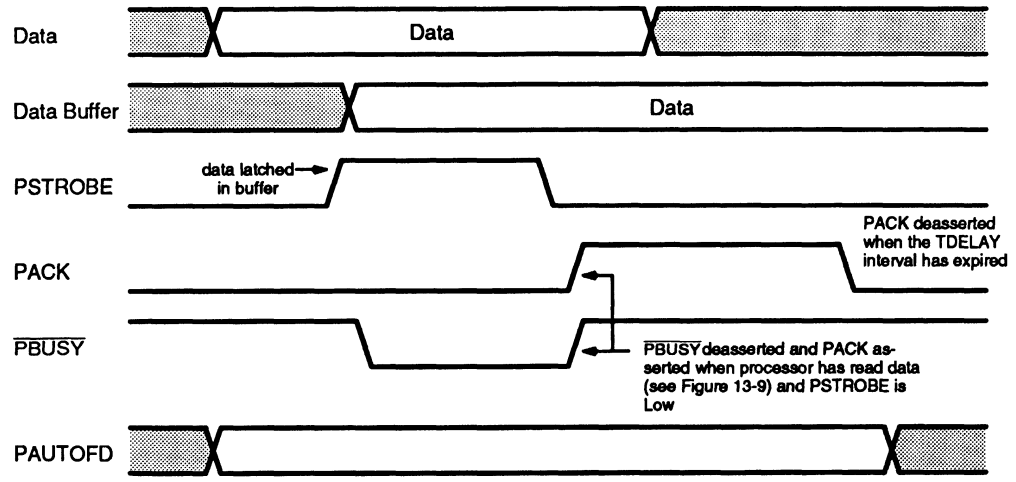


Figure 13-6 Transfer From the Host on the Parallel Port (BRS=0, ARB=1)



The Am29200 microprocessor asserts PBUSY within three MEMCLK cycles after the leading edge of PSTROBE (BRS=0) or within three MEMCLK cycles after the trailing edge of PSTROBE (BRS=1). The Am29200 microprocessor asserts PACK at the same time as PBUSY if ARB=0. The Parallel Port then generates either an interrupt request or a DMA request, as controlled by the MODE field, so the data can be read. If ARB=0, both PBUSY and PACK are deasserted once the TDELAY interval has expired, the Parallel Port Data Register (PDR) has been read, and the host has deasserted PSTROBE. If ARB=1, PBUSY is deasserted and PACK is asserted when the PDR has been read and PSTROBE is Low. PACK remains active until the TDELAY interval has expired. In any case, the TRA bit is reset when PACK is deasserted.

Figure 13-7 Transfer From the Host on the Parallel Port (BRS=1, ARB=0)

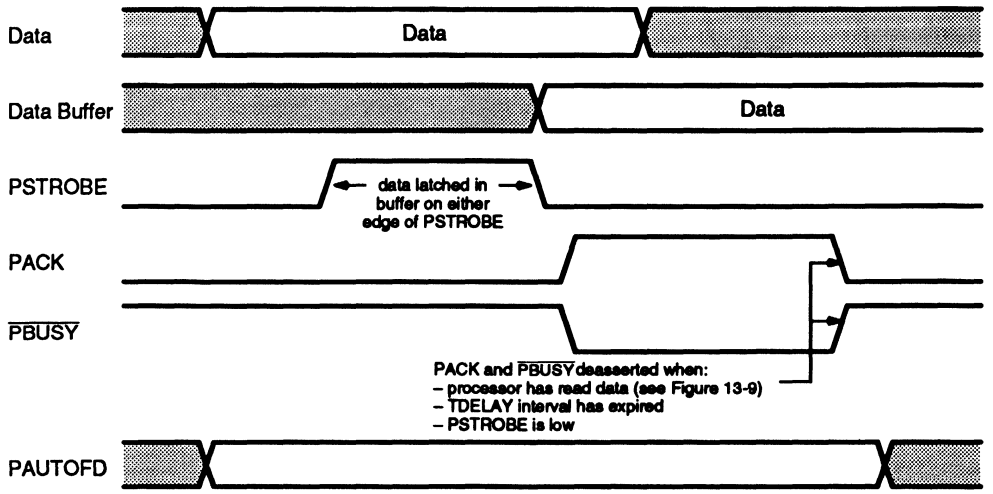
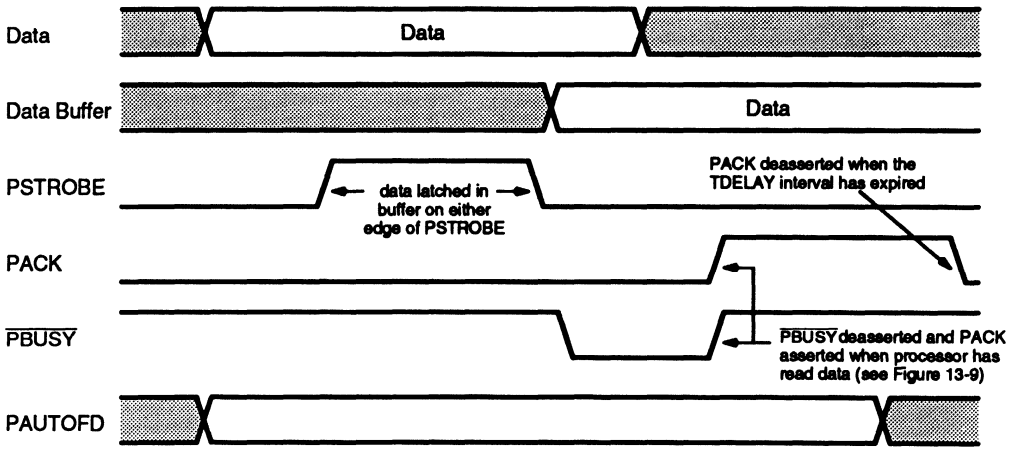


Figure 13-8 Transfer From the Host on the Parallel Port (BRS=1, ARB=1)



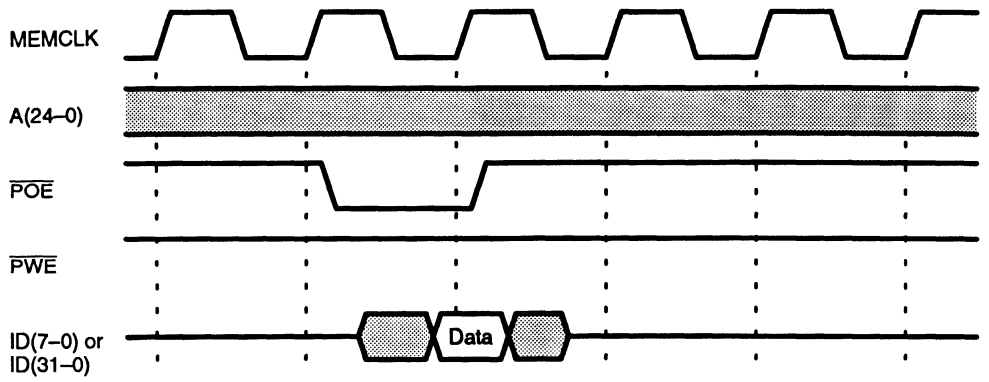
The PDR is mapped to the external buffer register. Figure 13-9 shows the timing of the external access. This external access is treated as either a DMA access or a processor PIA access for the purpose of prioritization with other accesses.

The PAUTOFD signal is used for software control during a transfer from the host. Software can detect the level on PAUTOFD by reading the AFD bit in the Parallel Port Control Register.

13.2.2 Transfers to the Host

Figure 13-10 shows the state transition diagram for transferring data from the Am29200 microprocessor to the host over the Parallel Port. Figure 13-11 shows the

Figure 13-9 Parallel Port Buffer Read Cycle



timing for this transfer. Transfers to the host are enabled by the host, using a system-dependent software protocol. This type of transfer is enabled in the processor by setting the DDIR bit in the Parallel Port Control Register. Setting the DDIR bit forces the PBUSY output active, preventing the host from transferring data to the Am29200 microprocessor. The MODE bit must be 00 when the DDIR bit is set or reset.

The Am29200 microprocessor begins the transfer by writing data to the external buffer. Figure 13-12 shows the timing for a buffer write. The buffer is written by either software writing the Parallel Port Data Register or a DMA transfer that writes the Parallel Port Data Register. Setting the DDIR bit causes the Parallel Port to generate the first DMA or interrupt request to write the data. Thereafter, the Parallel Port generates a DMA or interrupt request after it completes each transfer to the host.

During a transfer to the host, the PAUTOFD signal is used to indicate that the host is busy and cannot accept data. PAUTOFD has the same polarity as PBUSY for this purpose. After the data buffer has been written, the Parallel Port waits for one TDELAY interval and then asserts PACK as soon as PAUTOFD is High and PSTROBE is Low (these signal conditions may hold before the interval expires). The TDELAY interval is used to provide data setup time for the host. PACK is active for one TDELAY interval, then is deasserted.

In response to PACK, the host acknowledges the transfer by asserting PSTROBE, which resets the TRA bit. PSTROBE has no fixed relationship to PACK. The host may also assert PAUTOFD before the end of the transfer to indicate it is not ready for a subsequent transfer. Following the deassertion of PACK or the assertion of PSTROBE (whichever is later), the Parallel Port waits one TDELAY interval to provide data hold time to the host. At the end of the interval, the Parallel Port generates a new DMA or interrupt request to have the data buffer written again, starting a new transfer. Software or the DMA channel may determine that all transfers have been made, and a new transfer does not start in this case.

Figure 13-10 State Transitions for Transfers to the Host

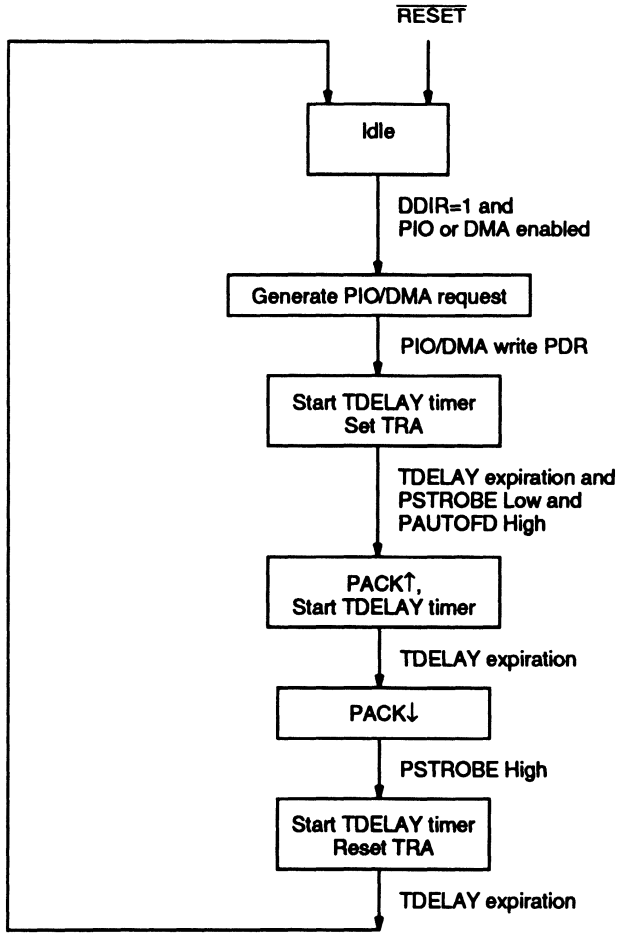


Figure 13-11 Transfer to the Host on the Parallel Port

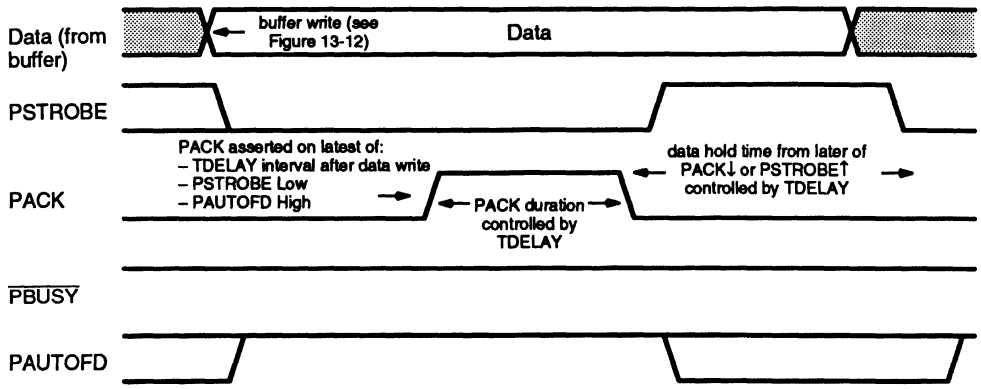
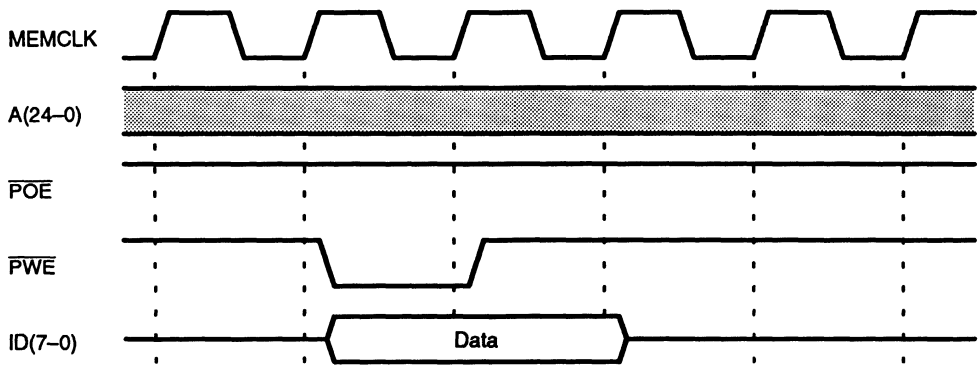


Figure 13-12 Parallel Port Buffer Write Cycle



CHAPTER 14

SERIAL PORT

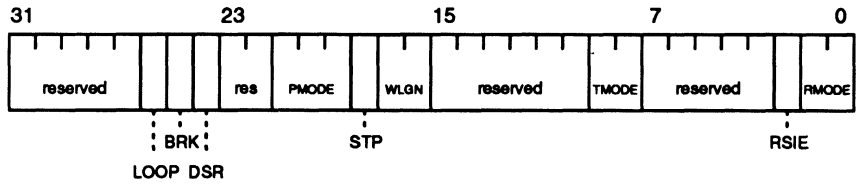
The Serial Port permits full-duplex, bi-directional data transfer using the RS-232 protocol.

14.1 PROGRAMMABLE REGISTERS

14.1.1 Serial Port Control Register (SPCT, Address 80000080)

The Serial Port Control Register (Figure 14-1) controls both the transmit and receive sections of the Serial Port.

Figure 14-1 Serial Port Control Register



Bits 31-27: Reserved.

Bit 26: Loopback (LOOP)—Setting this bit places the Serial Port in the loopback mode. In this mode, the TXD output is set High and the Transmit Shift Register is connected to the Receive Shift Register. Data transmitted by the transmit section is immediately received by the receive section. The loopback mode is provided for testing the Serial Port.

Bit 25: Send Break (BRK)—Setting this bit causes the Serial Port to send a break, which is a continuous Low level on the TXD output for a duration of more than one frame transmission time. The transmitter can be used to time the frame by setting the BRK bit when the transmitter is empty (indicated by the TEMT bit of the Serial Port Status Register), writing the Serial Port Transmit Holding Register with data to be transmitted, and then waiting until the TEMT bit is set again before resetting the BRK bit.

Bit 24: Data Set Ready (DSR)—Setting this bit causes the $\overline{\text{DSR}}$ output to be asserted. Resetting this bit causes the $\overline{\text{DSR}}$ output to be deasserted.

Bits 23-22: Reserved.

Bits 21-19: Parity Mode (PMODE)—This field specifies how parity generation and checking are performed during transmission and reception (the value "x" is a don't care):

PMODE Value	Parity Generation and Checking
0xx	No parity bit in frame
100	Odd parity (odd number of 1s in frame)
101	Even parity (even number of 1s in frame)
110	Parity forced/checked as 1
111	Parity forced/checked as 0

Bit 18: Stop Bits (STP)—A 0 in this bit specifies that one stop bit is used to signify the end of a frame. A 1 in this bit specifies that 2 stop bits are used to signify the end of a frame.

Bits 17-16: Word Length (WLGN)—This field indicates the number of data bits transmitted or received in a frame, as follows:

WLGN Value	Word Length
00	5 bits
01	6 bits
10	7 bits
11	8 bits

Data words of less than eight bits are right-justified in the Transmit Holding Register and Receive Buffer Register.

Bits 15-10: Reserved.

Bits 9-8: Transmit Mode (TMODE)—This field enables data transmission and controls the operational mode of the Serial Port for the transmission of data, as follows:

TMODE Value	Effect on Transmit Section
00	Disabled
01	Generate interrupt requests for service
10	Generate DMA Channel 0 requests
11	Generate DMA Channel 1 requests

Requests for service are requests to write the Transmit Holding Register with data to be transmitted. Placing the transmit section into the disabled state causes all internal state machines to be reset and holds the transmit section in an idle state with TXD High. Serial Port programmable registers are not affected when the transmit section is disabled.

Bits 7-3: Reserved.

Bit 2: Receive Status Interrupt Enable (RSIE)—This bit enables the Serial Port to generate an interrupt because of an exception during reception. If this bit is 1 and the Serial Port receives a break or experiences a framing error, parity error, or overrun error, the Serial Port generates a Receive Status interrupt.

Bits 1-0: Receive Mode (RMODE)—This field enables data reception and controls the operational mode of the Serial Port for the reception of data:

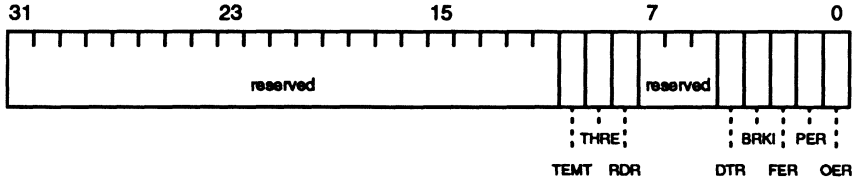
RMODE Value	Effect on Receive Section
00	Disabled
01	Generate interrupt requests for service
10	Generate DMA Channel 0 requests
11	Generate DMA Channel 1 requests

Requests for service are requests to read data from the Receive Buffer Register. Placing the receive section into the disabled state causes all internal state machines to be reset and holds the receive section in an idle state. Serial Port programmable registers are not affected when the receive section is disabled.

14.1.2 Serial Port Status Register (SPST, Address 80000084)

The Serial Port Status Register (Figure 14-2) indicates the status of the transmit and receive sections of the Serial Port.

Figure 14-2 Serial Port Status Register



Bits 31-11: Reserved.

Bit 10: Transmitter Empty (TEMT)—This bit is 1 when the transmitter has no data to transmit and the Transmit Shift Register is empty. This indicates to software it is safe to disable the transmit section.

Bit 9: Transmit Holding Register Empty (THRE)—When the THRE bit is 1, the Transmit Holding Register does not contain valid data and can be written with data to be transmitted. When the THRE bit is 0, the Transmit Holding Register contains valid data not yet copied to the Transmit Shift Register for transmission, and cannot be written. If so enabled by the TMODE field, the THRE bit causes an interrupt or DMA request when it is set. The THRE bit is reset automatically by writing the Transmit Holding Register. This bit is read-only, allowing other bits of the Serial Port Status Register to be written (for example, resetting the BRKI bit) without interfering with the data request.

Bit 8: Receive Data Ready (RDR)—When the RDR bit is 1, the Receive Buffer Register contains data that has been received on the serial port, and can be read to obtain the data. When the RDR bit is 0, the Receive Buffer Register does not contain valid data. If so enabled by the RMODE field, the RDR bit causes an interrupt or DMA request when it is set. The RDR bit is reset automatically by reading the Receive Buffer Register.

Bits 7-5: Reserved.

Bit 4: Data Terminal Ready (DTR)—The DTR bit indicates the level on the $\overline{DTR}$ pin. The DTR bit is 1 when the $\overline{DTR}$ pin is active, and the DTR bit is 0 when the $\overline{DTR}$ pin is inactive.

Bit 3: Break Interrupt (BRKI)—The BRKI bit is set to indicate that a break has been received. If the RSIE bit is 1, the BRKI bit being set causes a Receive Status interrupt. The BRKI bit should be reset by the Receive Status interrupt handler.

Bit 2: Framing Error (FER)—This bit is set to indicate that a framing error occurred during reception of data. If the RSIE bit is 1, the FER bit being set causes a Receive Status interrupt. The FER bit should be reset by the Receive Status interrupt handler.

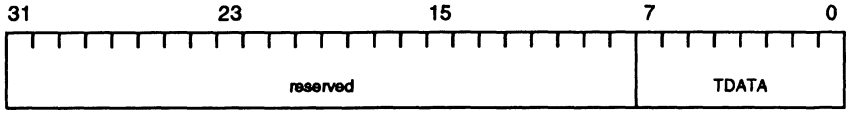
Bit 1: Parity Error (PER)—This bit is set to indicate that a parity error occurred during reception of data. If the RSIE bit is 1, the PER bit being set causes a Receive Status interrupt. The PER bit should be reset by the Receive Status interrupt handler.

Bit 0: Overrun Error (OER)—This bit is set to indicate that an overrun error occurred during reception of data. If the RSIE bit is 1, the OER bit being set causes a Receive Status interrupt. The OER bit should be reset by the Receive Status interrupt handler.

14.1.3 Serial Port Transmit Holding Register (SPTH, Address 80000088)

The processor writes this register (Figure 14-3) with data to be transmitted on the Serial Port. The transmitter is double-buffered, and the transmit section copies data from the Transmit Holding Register to the Transmit Shift Register (which is not accessible to software) before transmitting the data.

Figure 14-3 Serial Port Transmit Holding Register



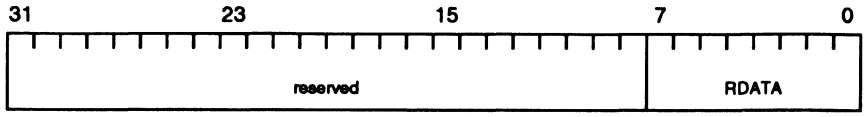
Bits 31-8: Reserved.

Bits 7-0: Transmit Data (TDATA)—This field is written with data to be transmitted on the Serial Port. The THRE bit of the Serial Port Status Register should be 1 when this register is written, to avoid overwriting data already in the register. Writing this register causes the THRE bit to be reset.

14.1.4 Serial Port Receive Buffer Register (SPRB, Address 8000008C)

This register (Figure 14-4) contains data received over the Serial Port. The receiver is double-buffered, and the receive section can be receiving a subsequent frame of data in the Receive Shift Register (which is not accessible to software) while the Receive Buffer is being read by software or by a DMA channel.

Figure 14-4 Serial Port Receive Buffer Register



Bits 31-8: Reserved.

Bits 7-0: Receive Data (RDATA)—This field contains data received on the Serial Port. The RDR bit of the Serial Port Status Register should be 1 when this register is read, to avoid reading invalid data. Reading this register causes the RDR bit to be reset.

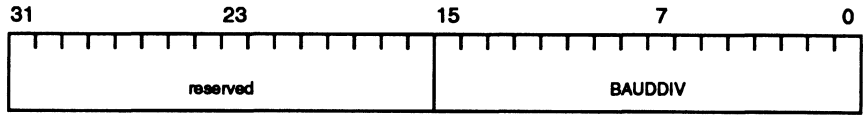
14.1.5 Baud Rate Divisor Register (BAUD, Address 80000090)

This register specifies a clock divisor for the generation of a serial clock that controls the Serial Port. The serial clock rate is 16 times the rate of transmission or reception of data. The Baud Rate Divisor Register specifies the zero-based number of UCLK cycles in one phase (half period) of the 16x serial clock. The formula for the baud rate is thus:

$$\text{Baud Rate} = (\text{Frequency of UCLK}) \div (\text{BAUDDIV}+1) \div 32$$

The maximum baud rate is 1/32 of INCLK, and is achieved by tying UCLK to INCLK with BAUDDIV=0000, hexadecimal.

Figure 14-5 Baud Rate Divisor Register



Bits 31-16: Reserved.

Bit 1: Baud Rate Divisor (BAUDDIV)—This field specifies the amount by which the UCLK input is divided to generate one phase of the serial clock. The serial clock operates at 16 times the rate of transmission or reception of data. The BAUDDIV value is zero-based. For example, a value of two specifies a divisor of three.

14.1.6 Serial Port Initialization

During a processor reset, both the TMODE and RMODE fields of the Serial Port Control Register are reset to 00, disabling the transmit and receive sections of the Serial Port. Software must initialize the Serial Port before it is enabled.

CHAPTER 15

VIDEO INTERFACE

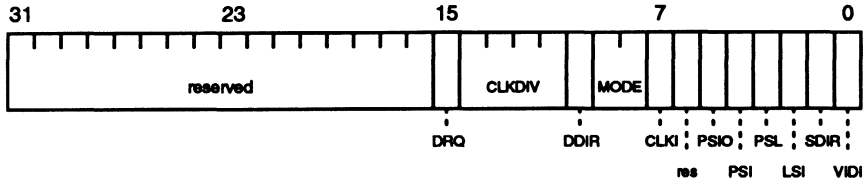
The Video Interface provides direct connection to a number of laser-beam marking engines. It may also be used to receive data from a raster input device such as a scanner or to serialize/deserialize a data stream.

15.1 PROGRAMMABLE REGISTERS

15.1.1 Video Control Register (VCT, Address 800000E0)

This register (see Figure 15-1) controls the operation of the Video Interface.

Figure 15-1 Video Control Register



Bits 31–16: Reserved.

Bit 15: Data Request (DRQ)—This bit is set to indicate that the Video Interface is ready for data to be written to or read from the Video Data Holding Register. If so enabled by the MODE field, this bit being set generates an interrupt or DMA request to write or read data. This bit is reset when the Video Data Holding Register is read or written. This bit is read-only, to allow other bits of the Video Control Register to be set (for example, the PSL bit) without interfering with the data request.

Bits 14–11: Clock Divide (CLKDIV)—This fields contains the divisor of the VCLK input used to generate the internal video clock. It specifies the number of VCLK periods in one phase (half period) of the internal video clock. For example, a value of 0001 indicates that one VCLK period constitutes one phase of the internal video clock—a divide by two. A value of 0000 causes VCLK to be used directly as the video clock. At the beginning of a video raster line, the clock divider is initialized so that, in the line, the first period of the internal clock is the indicated number of VCLK periods.

Bit 10: Data Direction (DDIR)—This bit controls the direction of video data. If the DDIR bit is 0, data is transmitted on the video interface. If the DDIR bit is 1, data is received on the video interface.

Bits 9-8: Video Interface Mode (MODE)—This field enables the Video Interface and controls the operational mode of the Video Interface, as follows:

MODE Value	Effect on Video Interface
00	Disabled
01	Generate interrupt requests for service
10	Generate DMA Channel 0 requests
11	Generate DMA Channel 1 requests

Requests for service are requests to read or write the Video Data Holding Register. Placing the Video Interface into the disabled state causes all internal state machines to be reset and holds the Video Interface in an idle state. Video Interface programmable registers are not affected when the interface is disabled.

Bit 7: Clock Invert (CLKI)—If this bit is 0, the VDAT, PSYNC, and LSYNC pins are driven or sampled on the Low-to-High transition of the VCLK input. If this bit is 1, the VDAT, PSYNC, and LSYNC pins are driven or sampled on the High-to-Low transition of the VCLK input.

Bit 6: Reserved.

Bit 5: Page Sync Input/Output (PSIO)—This bit determines whether or not PSYNC is an input or output. If this bit is 0, PSYNC is an input. If this bit is 1, PSYNC is an output.

Bit 4: Page Sync Invert (PSI)—If this bit is 0 and PSYNC is an input, a Low-to-High transition of the PSYNC input indicates the beginning of a page. If this bit is 1 and PSYNC is an input, a High-to-Low transition of the PSYNC input indicates the beginning of a page.

If this bit is 0 and PSYNC is an output, PSYNC is noninverted with respect to the PSL bit. A PSL bit of 0 is reflected as a Low level, a PSL bit of 1 is reflected as a High level, and a page starts on a Low-to-High transition. If this bit is 1 and PSYNC is an output, PSYNC is inverted with respect to the PSL bit. A PSL bit of 0 is reflected as a High level, a PSL bit of 1 is reflected as a Low level, and a page starts on a High-to-Low transition.

Bit 3: Page Sync Level (PSL)—When PSYNC is an input, this bit reflects the level on PSYNC. When PSYNC is an output, this bit determines the level on PSYNC. If PSI=0, a 0 in this bit corresponds to a Low level on PSYNC and a 1 in this bit corresponds to a High level on PSYNC. If PSI=1, a 0 in this bit corresponds to a High level on PSYNC and a 1 in this bit corresponds to a Low level on PSYNC.

Bit 2: Line Sync Invert (LSI)—If this bit is 0, a Low-to-High transition of the LSYNC input indicates the beginning of a line. If this bit is 1, a High-to-Low transition of the LSYNC input indicates the beginning of a line.

Bit 1: Shift Direction (SDIR)—When this bit is 0, the Video Data Shift Register is shifted right to transfer data, with video data being shifted out of the least significant bit of the register (corresponding to bit 0 of the Video Data Holding Register) or into the most significant bit (corresponding to bit 31 of the Video Data Holding Register). When this bit is 1, the Video Data Shift Register is shifted left to transfer data, with video data being shifted out of the most significant bit of the register or into the least significant bit.

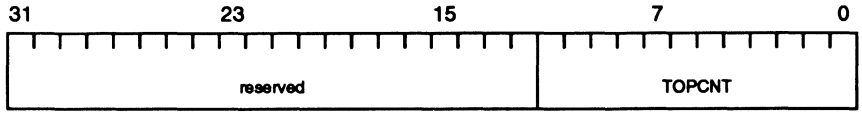
Bit 0: Video Invert (VIDI)—When this bit is 0, a 1 in the Video Data Shift Register corresponds to a High level on VDAT and a 0 in the Video Data Shift Register corresponds to a Low level on VDAT. When this bit is 1, a 1 in the Video Data Shift

Register corresponds to a Low level on VDAT and a 0 in the Video Data Shift Register corresponds to a High level on VDAT.

15.1.2 Top Margin Register (TOP, Address 800000E4)

This register (Figure 15-2) specifies the number of lines in the top margin of a page.

Figure 15-2 Top Margin Register



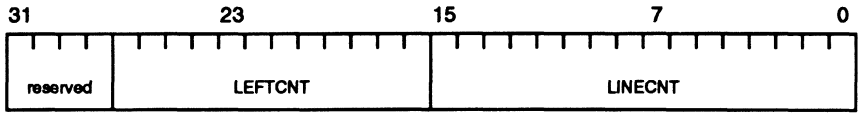
Bits 31–12: Reserved.

Bits 11-0: Top Margin Count (TOPCNT)—This field specifies the number of lines in the top margin.

15.1.3 Side Margin Register (SIDE, Address 800000E8)

This register (Figure 15-3) specifies the number of data bits in the left margin of a page and the number of bits in a raster line of video data. Together, this information sets the right and left margins of a page.

Figure 15-3 Side Margin Register



Bits 31–28: Reserved.

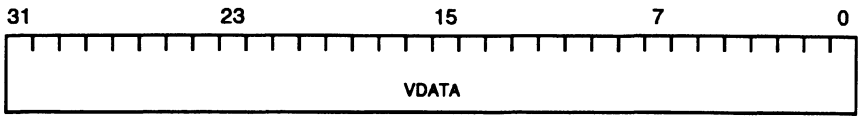
Bits 27-16: Left Margin Count (LEFTCNT)—This field specifies the number of data bit equivalents in the left margin of a page.

Bits 15-0: Line Count (LINECNT)—This field specifies the number of data bits in a raster line of video data.

15.1.4 Video Data Holding Register (VDT, Address 800000EC)

This register (Figure 15-4) contains data to be transmitted on or received from the video interface. Video data is double-buffered so data can be written to or read from the Video Data Holding Register while other data is transmitted from or received into the Video Data Shift Register.

Figure 15-4 Video Data Holding Register



Bits 31-0: Video Data (VDATA)—This field is written or read to transmit or receive data on the video interface.

15.1.5 Initialization

During a processor reset, the MODE field of the Video Control Register is reset to 00. Software must configure the Video Interface before it is enabled. To prevent possible driver conflicts during reset, the PSIO bit is reset and the DDIR bit is set so both PSYNC and VDAT are inputs. To allow time for the interface signals to settle, the inputs and outputs should be configured before the interface is enabled.

15.2 VIDEO INTERFACE OPERATION

The operation of the Video Interface is synchronous to the VCLK input, which either clocks the Video Interface directly or at a frequency multiple specified by the CLKDIV field. The CLKDIV field specifies the number of VCLK periods in one phase (half period) of the internal video clock. If the CLKDIV field has the value 0000, the VCLK input is used directly. The clock divider circuit is initialized when the Video Interface is disabled, and does not operate until the interface is enabled by the MODE field. This circuit is also initialized by the transition of LSYNC that indicates the beginning of a line. Initializing the clock divider with LSYNC insures that the first internal clock period in the line is the indicated number of VCLK periods. The maximum frequency of VCLK is the same as the maximum frequency of INCLK. The maximum operating frequency of the Video Interface is the frequency of INCLK if the interface is used to output data. The maximum operating frequency is one-eighth of the frequency of INCLK if the interface is used to input data.

The PSYNC, LSYNC, and VDAT pins are driven and/or sampled during either the Low-to-High (CLKI=0) or High-to-Low (CLKI=1) transition of the VCLK input. The clock divider sequences on the same transition. If the clock is not divided down, new data can be driven or sampled on every active transition of VCLK. If the clock is divided down, new data can be driven or sampled on every CLKDIV-times-2 active transition of VCLK.

15.2.1 Transmitting Data on the Video Interface

Before the Video Interface is enabled to transmit, the Video Control Register should be set to configure the interface, and the Top Margin and Side Margin registers should be set with the appropriate counts. When the DDIR bit is 0 (VDAT is an output) and the Video Interface is disabled or is not transferring data, the VDAT output is held at a level corresponding to a 0 data bit (Low if VIDI=0 or High if VIDI=1). Once the Video Interface has been configured, it is enabled via the MODE field.

Enabling the Video Interface with DDIR=0 causes the interface to set the DRQ bit, generating an interrupt or DMA request to write the Video Data Holding Register.

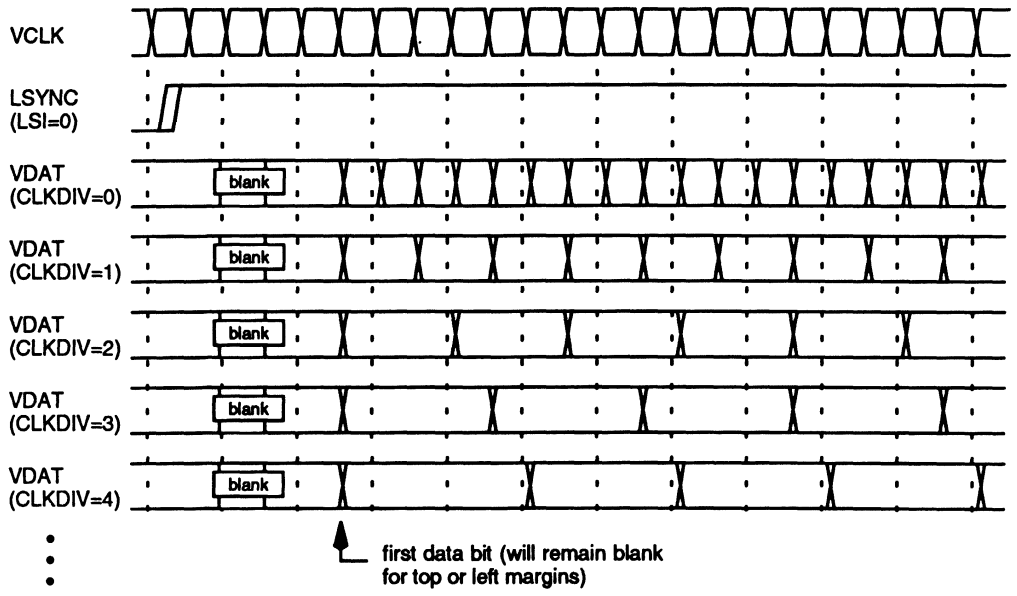
Writing data into the Video Data Holding Register resets the DRQ bit. Data is transferred from the Video Data Holding Register to the Video Data Shift Register whenever the Video Data Shift Register is empty. After the transfer, the DRQ bit is set to request more data. Thus, the DRQ bit may be set very soon after the first data word is written. Thereafter, however, the DRQ bit will be set only as data is transmitted on the interface.

A page cycle begins by an active transition of PSYNC, either as an input or output. At the beginning of a page cycle, three count-down registers are loaded from the TOPCNT, LEFTCNT, and LINECNT fields. The TOPCNT counter enables the transmission of the first raster line when it counts down to zero. The LEFTCNT counter enables the transmission of raster data on a line when it counts down to zero. The LINECNT counter enables the transmission of raster data as long as it is nonzero.

After the page cycle begins, the counter registers are not enabled to count until the first active transition of LSYNC. An active transition of LSYNC indicates the beginning of a line. Because of internal synchronization delay, the Video Interface does not respond to LSYNC until five VCLK cycles have elapsed (see Figure 15-5). If the Video Data Shift Register is not empty, an active transition on LSYNC causes the TOPCNT counter to decrement by one (the TOPCNT field is unaffected). The TOPCNT counter continues to decrement by one on each active transition of LSYNC until it reaches zero. Note that if the TOPCNT field contains zero at the beginning of a page, the Video Interface begins transmitting on the first active transition of LSYNC.

When the TOPCNT counter reaches zero, the interface is enabled to transmit the first raster line. At the beginning of the line, the LEFTCNT counter decrements on each active transition of the interface clock, beginning five VCLK cycles after the active edge of LSYNC, until the counter reaches zero. When the LEFTCNT counter reaches zero, the data in the selected end of the Video Data Shift Register is enabled to drive the VDAT output, and the LINECNT counter is enabled to count. The LEFTCNT counter is reloaded from the LEFTCNT field but does not count until the next active transition of LSYNC. If the LEFTCNT field contains zero at the beginning of a line,

Figure 15-5 VCLK, LSYNC, and VDAT Relationships (LSI=0 for example only)



video data is driven and the LINECNT counter is enabled to count immediately on the fifth VCLK cycle after the first active transition of LSYNC, after the TOPCNT counter reaches zero.

The first bit of video data is driven for a period of the interface clock, during the cycle in which the LEFTCNT counter reaches zero. On the next active transition of the clock, the Video Data Shift Register is shifted right (SDIR=0) or left (SDIR=1) by one bit and the new data driven on the VDAT output. Also, the LINECNT counter is decremented by one. When the last bit in the Video Data Shift Register has been transmitted, new data is loaded from the Video Data Holding Register and the DRQ bit is set to request more data. Data transmission continues until the LINECNT counter reaches zero. When the LINECNT counter reaches zero, the VDAT output is driven to correspond to a 0 data bit and the Video Data Shift Register is cleared. The LINECNT counter is reloaded but is not enabled to count until a new line begins and the LEFTCNT counter reaches zero once more. The VDAT output is held at a 0 data level and the Video Shift Register does not shift until the next line is transmitted. Clearing the Video Data Shift Register at the end of a line enables it to be reloaded with new data from the Video Data Holding Register as soon as this data is available.

On each subsequent active transition of LSYNC, a subsequent line of data is transmitted. Each line begins with a synchronization period of five VCLK cycles, then a countdown of the LEFTCNT counter until it reaches zero, followed by data transmission and shifting until the LINECNT counter reaches zero. On any active transition of LSYNC, if the Video Data Shift Register is empty, the page cycle ends and the Video Interface waits for the next active transition of PSYNC.

15.2.2 Receiving Data on the Video Interface

When the Video Interface is configured to receive data, the TOPCNT and LEFTCNT fields are not used, and the PSYNC pin is not used. Data reception is controlled by LSYNC, VCLK, and the LINECNT field.

On the active edge of LSYNC, the LINECNT counter is loaded with the contents of the LINECNT field. On the fifth active edge of VCLK following the active edge of LSYNC (for synchronization), data is sampled into the selected end of the Video Data Shift Register, the register is shifted in the selected direction, and the LINECNT counter is decremented by one. When the Video Data Shift Register has received 32 bits, the contents of the register are transferred into the Video Data Holding Register and the DRQ bit is set to request that the data be read. Data sampling and shifting continue until the LINECNT counter reaches zero. To clear the data at the end of a line after the LINECNT counter reaches zero, the data in the Video Data Shift Register is transferred into the Video Data Holding Register as soon as the holding register is available, and the DRQ bit is set. The interface waits for the next active transition of LSYNC before it accepts a new line of data.

CHAPTER 16

INTERRUPTS AND TRAPS

16.1 OVERVIEW

Interrupts and traps cause the Am29200 microprocessor to suspend the execution of an instruction sequence and to begin the execution of a new sequence. The processor may or may not later resume the execution of the original instruction sequence.

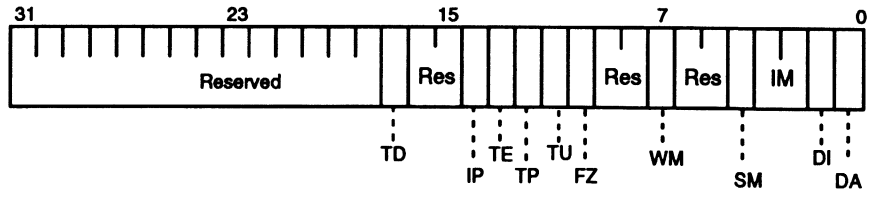
The distinction between interrupts and traps is largely one of causation and enabling. Interrupts allow external devices and the Timer Facility to control processor execution and are always asynchronous to program execution. Traps are intended to be used for certain exceptional events that occur during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

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

16.1.1 Current Processor Status (CPS, Register 2)

This protected special-purpose register (see Figure 16-1) controls the behavior of the processor and its ability to recognize exceptional events.

Figure 16-1 Current Processor Status Register



Bits 31–18: Reserved.

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

Bit 16–15: Reserved.

Bit 14: Interrupt Pending (IP)—This bit allows software to detect the presence of interrupts while the interrupts are disabled. The IP bit is set if an interrupt request 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 interrupt requests are subsequently deactivated 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 17.1 describes the use of these bits in more detail.

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

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

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

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

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

Bit 9–8: Reserved.

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

Bit 6–5: Reserved.

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

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

Bits 3–2: Interrupt Mask (IM)—The IM field is an encoding of the processor priority with respect to external interrupts. The interpretation of the interrupt mask is specified in Section 16.1.2.

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

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

16.1.2 Interrupts

Interrupts are caused by signals applied to any of the external inputs $\overline{\text{INTR}}$(3–0), by the Timer Facility (see Section 16.7), or by internal peripherals (see Section 16.8). 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. Timer interrupts may be disabled by the Timer Disable (TD) bit of the Current Processor Status Register.

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

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

Note that the $\overline{\text{INTR}}$0 interrupt cannot be disabled by the IM field. Also, 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 interrupt requests is active, but the corresponding interrupt is disabled due to the value of either DA, DI, or IM.

16.1.3 Traps

Traps are caused by signals applied to one of the inputs $\overline{\text{TRAP}}$(1–0), or by exceptional conditions such as protection violations. 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.

16.1.4 External Interrupts And Traps

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

The $\overline{\text{INTR}}$(3–0) inputs are prioritized with respect to each other and with respect to the processor. To resolve conflicts between these inputs, the inputs are prioritized in order, so the interrupt caused by $\overline{\text{INTR0}}$ has the highest priority, and the interrupt caused by $\overline{\text{INTR3}}$ has the lowest priority.

The $\overline{\text{TRAP}}$(1–0) inputs are prioritized with respect to each other, so the trap caused by $\overline{\text{TRAP0}}$ has priority over the trap caused by $\overline{\text{TRAP1}}$ when a conflict occurs. Both $\overline{\text{TRAP0}}$ and $\overline{\text{TRAP1}}$ have priority over the $\overline{\text{INTR}}$(3–0) inputs. The $\overline{\text{TRAP}}$(1–0) inputs cannot be disabled selectively. Both traps, however, can be disabled by the DA bit in the Current Processor Status Register.

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

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

There is a five-cycle latency from the de-assertion of an $\overline{\text{INTR}}$(3–0) or $\overline{\text{TRAP}}$(1–0) input to the time the corresponding interrupt or trap is actually not recognized by the processor. The latency is due to the metastability hardening that allows these signals to be driven with slow-transition-time signals. The de-assertion must be timed so the processor is not recognizing the interrupt or trap by the time the corresponding mask is reset. Otherwise, a spurious interrupt or trap may occur.

16.1.5 Wait Mode

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

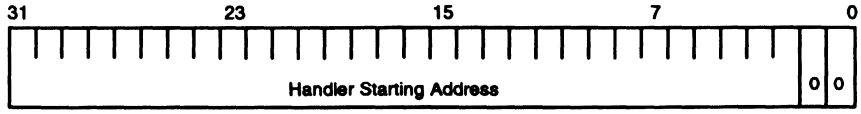
The processor can take only those interrupts or traps for which it is enabled, even in the Wait mode. For example, if the processor is in the Wait mode with a DA bit of 1, it can leave the Wait mode only via a processor reset (see Section 2.9.2) or a $\overline{\text{WARN}}$ trap (see Section 16.4).

16.2 VECTOR AREA

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

The structure of the Vector Area is a table of vectors in instruction/data memory. The layout of a single vector is shown in Figure 16-2. Each vector gives the beginning word-address of the associated interrupt or trap handling routine.

Figure 16-2 Vector Table Entry

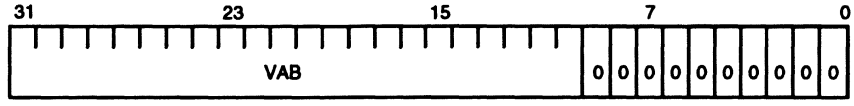


16.2.1 Vector Area Base Address (VAB, Register 0)

This protected special-purpose register (Figure 16-3) specifies the beginning address of the interrupt/trap Vector Area. The Vector Area is a table of 256 vectors which point to interrupt and trap handling routines.

When an interrupt or trap is taken, the vector number for the interrupt or trap (see Section 16.2.2) replaces bits 9–2 of the value in the Vector Area Base Address Register to generate the physical address for a vector contained in instruction/data memory.

Figure 16-3 Vector Area Base Address Register



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

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

16.2.2 Vector Numbers

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

Vector numbers are either predefined or specified by an instruction causing the trap. The assignment of vector numbers is shown in Table 16-1 (vector numbers are in decimal notation). Vector numbers 64 to 255 are for use by trapping instructions; the definition of the routines associated with these numbers is system dependent.

16.3 INTERRUPT AND TRAP HANDLING

Interrupt and trap handling consists of two distinct operations: taking the interrupt or trap and returning from the interrupt or trap handler. If the interrupt or trap handler

Table 16-1 Vector Number Assignments

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

1. This vector number also results if an external device removes $\overline{\text{INTR3}}$–$\overline{\text{INTR0}}$ or $\overline{\text{TRAP1}}$–$\overline{\text{TRAP0}}$ before the corresponding interrupt or trap is taken by the processor.
2. Some Supervisor-mode operations cause Protection Violations to facilitate virtualization of certain operations.
3. The Floating-Point Exception trap is not generated by the processor hardware. It is generated by the software that implements the virtual arithmetic interface (see Section 2.8).

Table 16-1 Vector Number Assignments (continued)

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

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

returns directly to the interrupted routine, the interrupt or trap handler need not save and restore processor state.

16.3.1 Old Processor Status (OPS, Register 1)

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

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

16.3.2 The Program Counter Stack

The Program Counter Unit, shown in Figure 16-4, forms and sequences instruction addresses for the Instruction Fetch Unit. It contains the Program Counter (PC), the Program-Counter Multiplexer (PC MUX), the Return Address Latch, and the Program-Counter Buffer (PC Buffer).

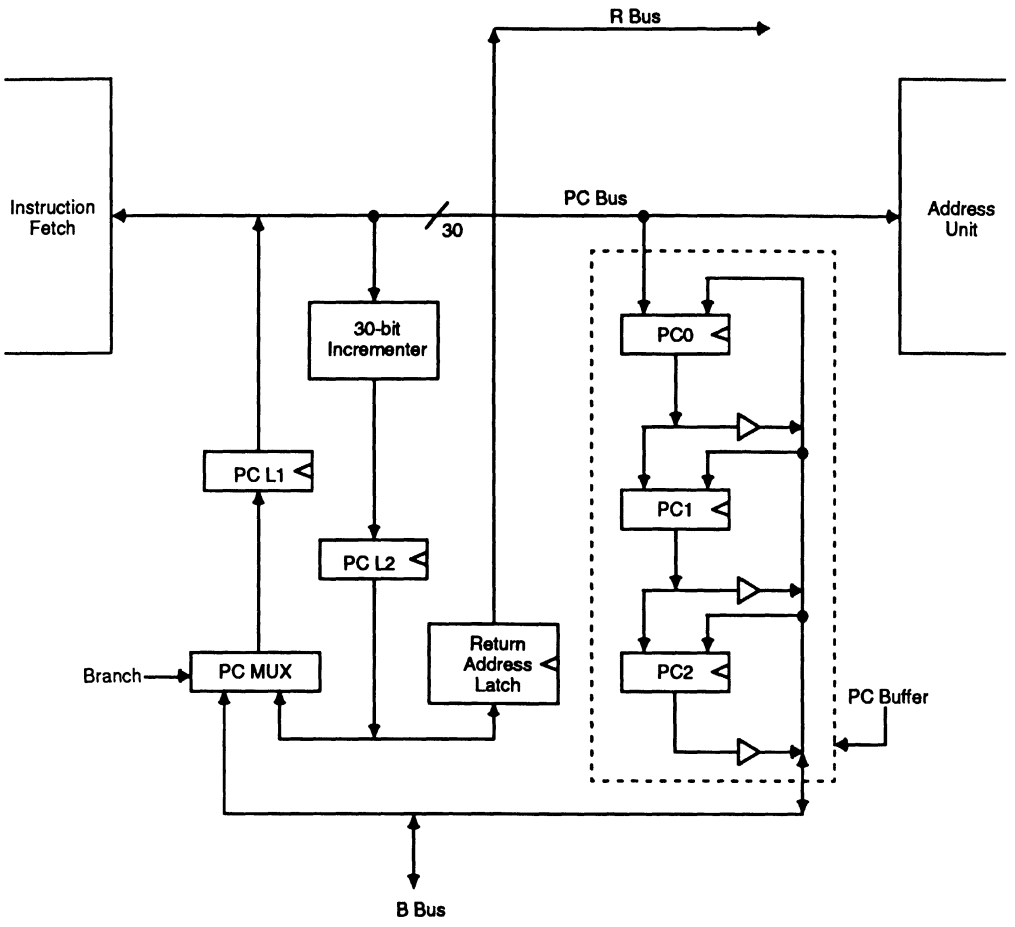
The PC forms addresses for sequential instructions executed by the processor. The master of the PC Register, PC L1, contains the address of the instruction being fetched in the Instruction Fetch Unit. The slave of the PC Register, PC L2, contains the next sequential address, which may be fetched by the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

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

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

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

Figure 16-4 Program Counter Unit



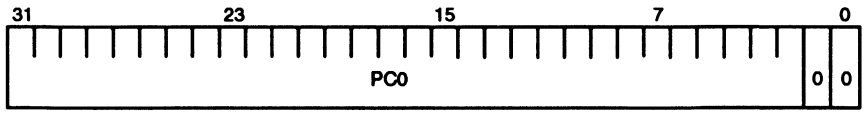
Upon the execution of an interrupt return, the target instruction stream is restarted using the instruction addresses in PC0 and PC1. Two registers are required here because the processor implements delayed branches. An interrupt or trap may be taken when the processor is executing the delay instruction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

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

16.3.2.1 PROGRAM COUNTER 0 (PC0, Register 10)

This protected special-purpose register (Figure 16-5) is used on an interrupt return to restart the instruction in the decode stage when the original interrupt or trap was taken.

Figure 16-5 Program Counter 0 Register



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

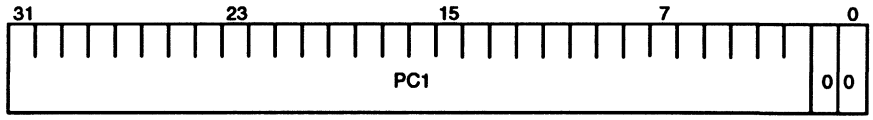
When an interrupt or trap is taken, the PC0 field contains the word-address of the instruction in the decode stage. The interrupt or trap has prevented this instruction from executing. The processor uses the PC0 field to restart this instruction on an interrupt return.

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

16.3.2.2 PROGRAM COUNTER 1 (PC1, Register 11)

This protected special-purpose register (Figure 16-6) is used on an interrupt return to restart the instruction in the execute stage when the original interrupt or trap was taken.

Figure 16-6 Program Counter 1 Register



Bits 31–2: Program Counter 1 (PC1)—This field captures the word-address of an instruction as it enters the execute stage of the processor pipeline, unless the Freeze (FZ) bit of the Current Processor Status Register is 1. If the FZ bit is 1, PC1 holds its value.

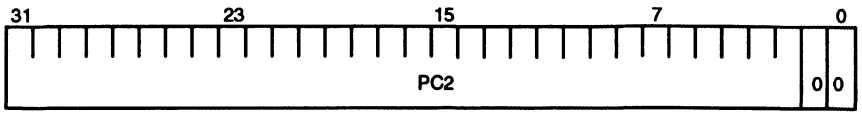
When an interrupt or trap is taken, the PC1 field contains the word-address of the instruction in the execute stage; the interrupt or trap has prevented this instruction from completing execution. The processor uses the PC1 field to restart this instruction on an interrupt return.

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

16.3.2.3 PROGRAM COUNTER 2 (PC2, Register 12)

This protected special-purpose register (Figure 16-7) reports the address of certain instructions causing traps.

Figure 16-7 **Program Counter 2 Register**



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

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

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

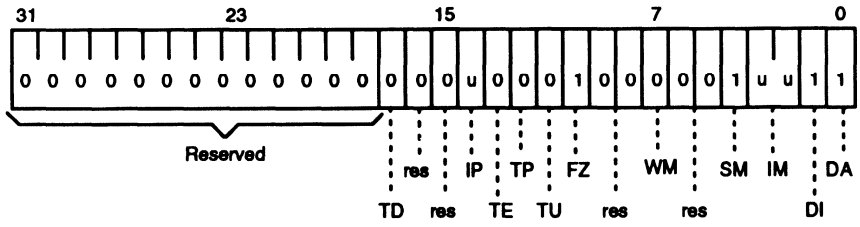
16.3.3 Taking An Interrupt Or Trap

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

1. Instruction execution is suspended.
2. Instruction fetching is suspended.
3. Any in-progress load or store operation is completed. Any additional operations are canceled in the case of load multiple and store multiple.
4. The contents of the Current Processor Status Register are copied into the Old Processor Status Register.
5. The Current Processor Status register is modified as shown in Figure 16-8 (the value *u* means unaffected). Note that setting the Freeze (FZ) bit freezes the Channel Address, Channel Data, Channel Control, Program Counter 0, Program Counter 1, Program Counter 2, and ALU Status Registers.
6. The address of the first instruction of the interrupt or trap handler is determined. The address is obtained by accessing a vector from instruction/data memory, using the physical address obtained from the Vector Area Base Address Register and the vector number. This is a 32-bit access.
7. An instruction fetch is initiated using the instruction address determined in step 6. At this point, normal instruction execution resumes.

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

Figure 16-8 Current Processor Status After an Interrupt or Trap



16.3.4 Returning From An Interrupt Or Trap

Two instructions are used to resume the execution of an interrupted program: Interrupt Return (IRET), and Interrupt Return and Invalidate (IRETINV). These instructions are identical in the Am29200 microprocessor; in other 29K Family processors, the IRETINV instruction resets all Valid bits in an instruction cache, whereas the IRET instruction does not affect the Valid bits.

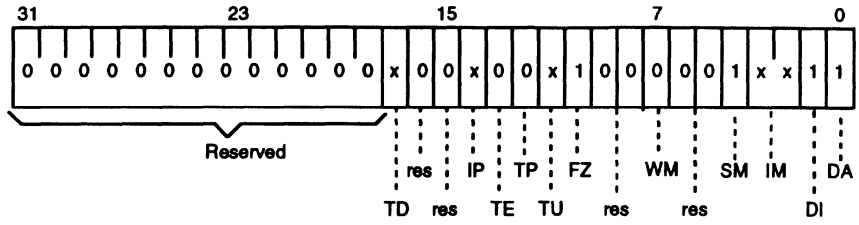
In some situations, the processor state must be set properly by software before the interrupt return is executed. The following is a list of operations normally performed in such cases:

1. The Current Processor Status is configured as shown in Figure 16-9 (the value *x* is a *don't care*). Note that setting the FZ bit freezes the registers listed below so they may be set for the interrupt return.
2. The Old Processor Status is set to the value of the Current Processor Status for the target routine.
3. The Channel Address, Channel Data, and Channel Control registers are set to restart or resume uncompleted external accesses of the target routine.
4. The Program Counter 1 and Program Counter 0 registers are set to the addresses of the first and second instructions, respectively, to be executed in the target routine.
5. Other registers are set as required. These may include registers such as the ALU Status, Q, and so forth, depending on the particular situation. Some of these registers are unaffected by the FZ bit, so they must be set in such a manner that they are not modified unintentionally before the interrupt return.

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

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. 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

Figure 16-9 Current Processor Status Before Interrupt Return



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).

4. 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 usual. Load-multiple and store-multiple operations are not restarted at this point.
5. 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 processor pipeline.
6. The instruction fetched in step 6 enters the decode stage of the pipeline.
7. 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 processor pipeline.
8. The instruction fetched in step 6 enters the execute stage of the pipeline, and the instruction fetched in step 8 enters the decode stage.
9. 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.
10. Interrupts and traps are enabled per the appropriate bits in the Current Processor Status Register.
11. The processor resumes normal operation.

16.3.5 Lightweight Interrupt Processing

The registers affected by the FZ bit of the Current Processor Status Register are those 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 affected by the FZ bit. This permits the implementation of light-weight interrupt handlers that do not have all of the overhead normally associated with interrupt handlers.

The processor provides an additional benefit to lightweight interrupts if the Program Counter 0 and Program Counter 1 Registers are not modified by the interrupt or trap handler. If Program Counters 0 and 1 contain the addresses of sequential instructions when an interrupt or trap is taken, and if they are not modified before an interrupt return is executed, step 7 of the interrupt return sequence in Section 16.3.4 occurs as a sequential fetch—instead of a branch—for the interrupt return. The performance impact of a sequential fetch is normally less than that of a branch.

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

1. Load Multiple and Store Multiple. The Channel Address, Channel Data, and Channel Control registers are used to sequence load-multiple and store-multiple operations, so these instructions cannot be executed while the registers are frozen. However, 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 SB bit 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, assert instructions, illegal operation codes, arithmetic overflow, etc.), since these are disabled. There are certain cases, however, where traps are unavoidable. Special considerations for these cases are discussed in Section 16.6.6.

16.3.6 Simulation Of Interrupts And Traps

Assert instructions may be used by a Supervisor-mode program to simulate the occurrence of various interrupts and traps defined for the processor. Only an assert instruction executed in Supervisor mode can specify a vector number between 0 and 63. If this instruction causes a trap, the effect is to create an interrupt or trap similar to that associated with the specified vector number.

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

16.4 $\overline{\text{WARN}}$ TRAP

The processor recognizes a special trap, caused by the activation of the $\overline{\text{WARN}}$ input, which cannot be masked. The $\overline{\text{WARN}}$ trap is intended to be used for severe system-

error or deadlock conditions. It allows the processor to be placed in a known, operable state, while preserving much of its original state for error reporting and possible recovery. Therefore, it shares some features in common with the Reset mode as well as features common to other traps described in this section.

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

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

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

16.4.1 $\overline{\text{WARN}}$ Input

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

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

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

16.5 SEQUENCING OF INTERRUPTS AND TRAPS

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

To resolve conflicts, interrupts and traps are taken according to the priority shown in Table 16-2. In this table, interrupts and traps are listed in order of decreasing priority. This section discusses the first three columns of Table 16-2. The last two columns are discussed in Section 16.6.

In Table 16-2, interrupts and traps fall into one of two categories depending on the timing of their occurrence relative to instruction execution. These categories are indicated in the third column of Table 16-2 by the labels *Inst* and *Async*. These labels have the following meaning:

1. Inst—Generated by the execution or attempted execution of an instruction.

Table 16-2 Interrupt and Trap Priority Table

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

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

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 either remain active until they can be taken or they 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 16-2 with priority 13 through 15 are regenerated by the re-execution of the causing instruction.
2. Most of the interrupts and traps of priority 4 through 12 must be held by external hardware until they are taken. The exceptions to this are listed in item 3.
3. The exceptions to item 2 are 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 two relevant bits are 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 16-2, except for the Unaligned Access trap, are not regenerated. These traps are mutually exclusive and are given high priority because they cannot be regenerated; they must be taken if they occur. If one of these traps occurs at the same time as a reset or $\overline{\text{WARN}}$ trap, it is not taken and its occurrence is lost.
5. The Unaligned Access trap is regenerated internally when an external access is restarted by the Channel Address, Channel Data, and Channel Control registers. Note this trap is not necessarily exclusive to the traps discussed in item 4 above.

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

16.6 EXCEPTION REPORTING AND RESTARTING

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

The *PC1* column in Table 16-2 describes the value held in the Program Counter 1 Register (PC1) when the interrupt or trap is taken. For traps in the *Inst* category, PC1 contains either the address of the instruction causing the trap, indicated by *Curr*, or the address of the instruction following the instruction causing the trap, indicated by *Next*.

For interrupts and traps in the *Async* category, PC1 contains the address of the first instruction 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.

16.6.1 Instruction Exceptions

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

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

16.6.2 Restarting Mapped DRAM Accesses

DRAM mapping is provided to support application needs such as on-the-fly data compression and decompression. In such applications, programs operate on large, compressed data structures by decompressing data into a smaller region of memory, operating on the data, and then compressing back into the large compressed structure. The ability to store the data in a compressed format reduces system memory requirements, while the ability to operate on the data in a decompressed format simplifies the application software.

For generality, mapped DRAM accesses allow the mapping configuration to be changed on demand. In other words, the DRAM mapping is performed by a system routine that changes the mapping as needed by the application program. This allows applications written with no knowledge of DRAM mapping to operate in a system that uses DRAM mapping. Since the DRAM mapping trap is part of normal system operation and does not represent an error, the access that causes the trap must be restarted—once the trapping condition is remedied—in a manner that cannot be detected by the program causing the trap.

The Am29200 microprocessor 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, but after the access began, may have altered the conditions of the access, such as by altering the address source register.

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

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

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

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

The mechanism used to restart trapping accesses has the additional benefit of allowing a fast interrupt-response time when the processor is performing a load-multiple or

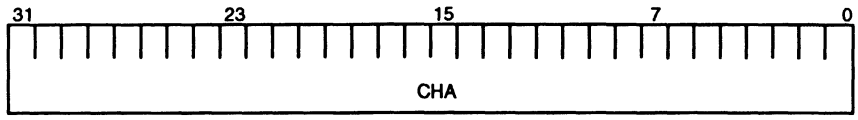
store-multiple operation. An interrupted load-multiple or store-multiple is restarted as if it had faulted. In this case, the operation resumes from the point of interruption, not from the beginning of the sequence.

16.6.2.1 CHANNEL ADDRESS (CHA, Register 4)

This protected special-purpose register (Figure 16-10) is used to report exceptions during external accesses. It also is used to restart interrupted load-multiple and store-multiple operations and to restart other external accesses when possible (e.g., after DRAM mapping misses are serviced).

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

Figure 16-10 Channel Address Register



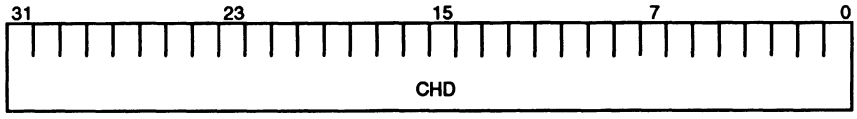
Bits 31–0: Channel Address (CHA)—This field contains the address of the current access (if the FZ bit of the Current Processor Status Register is 0).

16.6.2.2 CHANNEL DATA (CHD, Register 5)

This protected special-purpose register (Figure 16-11) is used to report exceptions during external accesses. 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 DRAM mapping misses are serviced).

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

Figure 16-11 Channel Data Register



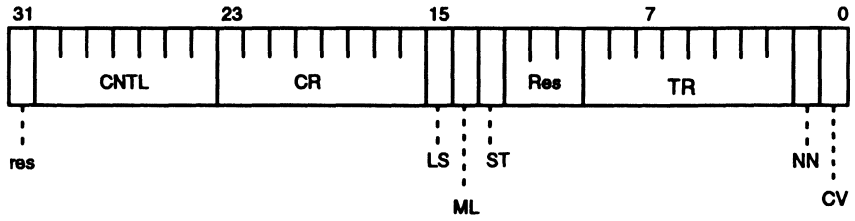
Bits 31–0: Channel Data (CHD)—This field contains the data (if any) associated with the current access (if the FZ bit of the Current Processor Status Register is 0). If the current access is not a store, the value of this field is irrelevant.

16.6.2.3 CHANNEL CONTROL (CHC, Register 6)

This protected special-purpose register (Figure 16-12) is used to report exceptions during external accesses. It is also used to restart interrupted load-multiple and store-multiple operations and to restart other external accesses when possible (e.g., after DRAM mapping misses are serviced).

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

Figure 16-12 Channel Control Register



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

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

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

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

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

Bit 12–10: Reserved.

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

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

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

16.6.3 Integer Exceptions

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

Two integer multiply instructions—MULTIPLY and MULTIPLU—cause an Out-of-Range trap upon overflow of a 32-bit signed or unsigned result, respectively, if the MO bit of the Integer Environment Register is 0. If the MO bit is 1, these multiply

instructions cannot cause an Out-of-Range trap. Since the processor does not contain hardware to directly support these instructions, the Out-of-Range trap must be generated by the software that implements the virtual arithmetic interface (see Section 2.8).

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

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

16.6.4 Floating-Point Exceptions

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

In addition to the operations described in Section 16.3.3, the following operations are performed when a Floating-Point Exception trap is taken:

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

16.6.5 Correcting Out-of-Range Results

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

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

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

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

16.6.6 Exceptions During Interrupt and Trap Handling

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

If these assumptions are not valid for a particular interrupt or trap handler, the handler must save the state of the processor and reset the FZ bit of the Current Processor Status, so 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.

16.7 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—accessible only to Supervisor-mode programs. Also, the Current Processor Status Register contains a control bit as part of the timer facility. These registers implement timing functions independent of program execution.

16.7.1 Timer Facility Operation

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

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

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

16.7.2 Timer Facility Initialization

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

1. Set the TCV field with the desired interval count for the first timing interval. This interval must be sufficiently large to allow the execution of the next step before the TCV field decrements to zero (this normally is the case).

2. Set the TRV field with the desired interval count for the second timing interval. The OV and IN bits are reset and the IE bit is set as desired. The second timing interval may be equivalent to the first timing interval.

16.7.3 Handling Timer Interrupts

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

1. Read the Timer Reload register into a general-purpose register.
2. Reset the IN bit in the general-purpose register.
3. Set the TRV field in the general-purpose register to the desired value for the next timing interval. Note that at this time the Timer Counter is timing the current interval. 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.

16.7.4 Timer Facility Uses

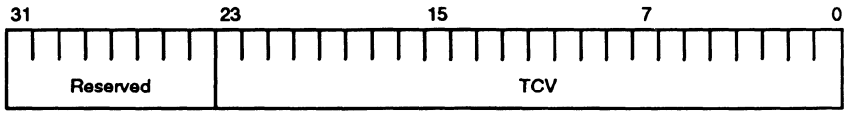
Since the Timer Facility has a resolution of a single processor cycle, it may be used to perform precise timing of system events. For example, it may be used to determine an exact measurement of the number of cycles between two events in the system or to perform precise time-critical control functions.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

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.

16.7.5 Timer Counter (TMC, Register 8)

This protected special-purpose register (Figure 16-13) contains the counter for the Timer Facility.

Figure 16-13 Timer Counter Register



Bits 31–24: Reserved.

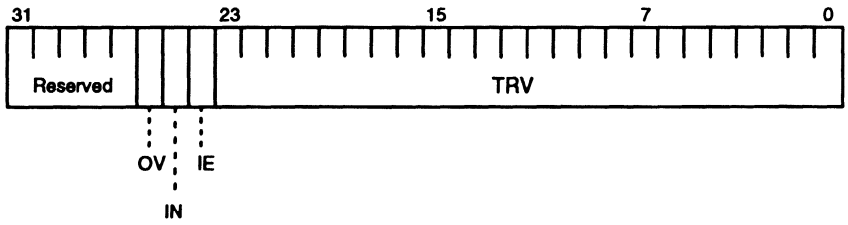
Bits 23–0: Timer Count Value (TCV)—The 24-bit TCV field decrements by one on each processor clock. When the TCV field decrements to zero, it is reloaded with the content of the Timer Reload Value field in the Timer Reload Register. At this time, the Interrupt bit in the Timer Reload Register is set.

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

16.7.6 Timer Reload (TMR, Register 9)

This protected special-purpose register (Figure 16-14) maintains synchronization of the Timer Counter Register, enables Timer interrupts, and maintains Timer Facility status information.

Figure 16-14 Timer Reload Register



Bits 31–27: Reserved.

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

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

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

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

16.8 INTERNAL INTERRUPT CONTROLLER

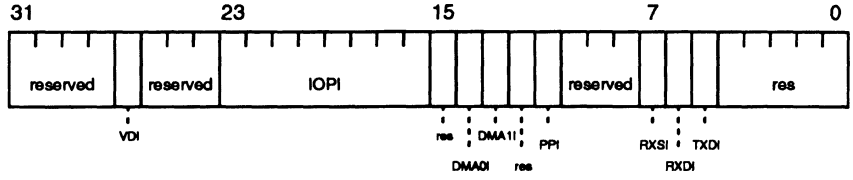
The various peripherals and controllers on the Am29200 microprocessor can cause interrupts having the same effect on the processor as asserting the processor's $\overline{\text{INTR3}}$ input. The Interrupt Controller provides a central location for generating interrupts, indicating which interrupts are active and permitting software to reset the interrupts independent of servicing the interrupting peripheral.

16.8.1 Interrupt Control Register (ICT, Address 80000028)

Bits of the Interrupt Control Register (Figure 16-15) are set at the leading edge of an interrupt condition, except for the bits related to the I/O Port (in the IOPI field), since I/O Port signals are independently configurable to generate edge-triggered interrupts. For example, the DMA0I bit is set when the CTI bit transitions from 0 to 1 in the DMA0 Control Register. When a bit in this register is 1, it causes an internal assertion of the processor's $\overline{INTR3}$ input (there is no external indication of this on $\overline{INTR3}$). Software can inspect this register to determine the source of the interrupt and can reset bits in this register to clear the interrupt.

Bits in the Interrupt Control Register are reset-only. Writing a 1 into a bit position causes the bit to be reset, unless an interrupting condition becomes active at the same time, in which case the bit remains set. Writing a bit with 0 does not affect the bit, and the bit may be set by an interrupting condition at the same time the bit is written with 0.

Figure 16-15 Interrupt Control Register



Bits 31-28: Reserved.

Bit 27: Video Interrupt (VDI)—A 1 in this bit indicates the Video Interface has generated an interrupt request.

Bits 26-24: Reserved.

Bits 23-16: I/O Port Interrupt (IOPI)—A 1 in this field indicates the respective PIO signal has generated an interrupt request. A 1 in the most significant bit of the IOPI field indicates PIO15 has caused an interrupt, the next bit indicates PIO14 has caused an interrupt, and so on.

Bit 15: Reserved.

Bit 14: DMA Channel 0 Interrupt (DMA0I)—A 1 in this bit indicates DMA Channel 0 has generated an interrupt request.

Bit 13: DMA Channel 1 Interrupt (DMA1I)—A 1 in this bit indicates DMA Channel 1 has generated an interrupt request.

Bit 12: Reserved.

Bit 11: Parallel Port Interrupt (PPI)—A 1 in this bit indicates the Parallel Port has generated an interrupt request.

Bits 10-8: Reserved.

Bit 7: Serial Port Receive Status Interrupt (RXSI)—A 1 in this bit indicates the Serial Port has generated an interrupt request because of the status of the receive logic.

Bit 6: Serial Port Receive Data Interrupt (RXDI)—A 1 in this bit indicates the Serial Port has generated an interrupt request because receive data is ready.

Bit 5: Serial Port Transmit Data Interrupt (TXDI)—A 1 in this bit indicates the Serial Port has generated an interrupt request because the Transmit Holding Register is empty.

Bits 4-0: Reserved.

16.8.2 Interrupt Controller Initialization

Processor interrupts are disabled by a processor reset, but the Interrupt Control Register is not affected by a reset. To prevent spurious interrupts, software should reset all bits of the Interrupt Control Register to 0 before processor interrupts are enabled.

16.8.3 Servicing Internal Interrupts

The Interrupt Control Register allows software to determine the source of an internal interrupt. Software can prioritize these interrupts using the processor's Count Leading Zeros instruction.

Software clears an interrupt by writing a 1 into the bit that is causing the interrupt (normally, the leading 1-bit in the Interrupt Control Register). For level-sensitive I/O Port interrupts, the interrupting condition must be cleared and the corresponding PIO signal be in an inactive state before the Interrupt Control Register bit is cleared, otherwise another interrupt will be generated. For other types of interrupts, the condition causing the interrupt can be cleared in the interrupting peripheral independent of resetting the bit in the Interrupt Control Register, because the leading edge of the condition must be detected again before another interrupt can occur (however, the interrupt should not be cleared in a way that might lose the occurrence of a newly generated interrupt). Because the Interrupt Control Register is reset-only and because resetting a bit takes lower precedence than setting a bit, bits can be reset without interfering with other interrupts or with the detection of a new interrupt of the type being cleared.

CHAPTER 17

DEBUGGING AND TESTING

This chapter details the features of the Am29200 microprocessor that support debugging and testing. The chapter first describes the Trace Facility and instruction breakpoints which aid in software debugging. Next, the Test/Development Interface is described. Finally, the Test Access Port and the Boundary Scan Architecture is discussed.

17.1 TRACE FACILITY

Software debug is supported by the Trace Facility. The Trace Facility guarantees exactly one trap after the execution of any instruction in a program being tested. This allows a debug routine to follow the execution of instructions and to determine the state of the processor and system at the end of each instruction.

Tracing is controlled by the Trace Enable (TE) and Trace Pending (TP) bits of the Current Processor Status Register. The value of the TE bit is always copied into the TP bit when an instruction enters the write-back stage of the processor pipeline. A Trace trap occurs whenever the TP bit is 1. As with most traps, the Trace trap can be disabled only by the DA bit of the Current Processor Status Register.

In order to trace the execution of a program, the debug routine performs an interrupt return to cause the program to begin or resume execution. However, before the interrupt return is executed, the TE and TP bits of the Old Processor Status are set with the values 1 and 0, respectively. The interrupt return causes these bits to be copied into the TE and TP bits of the Current Processor Status.

When the target instruction of the interrupt return (whose address is contained in the Program Counter 1 Register when the interrupt return is executed) enters the write-back stage, the processor copies the value of the TE bit into the TP bit. Since the TP bit is a 1, a Trace trap occurs. This trap prevents any further instruction execution in the target routine until the interrupt is taken and the routine is resumed with an interrupt return. When the Trace trap is taken, the TE and TP bits are both reset automatically, preventing any further Trace traps.

Since the Trace Facility is managed by the Old and Current Processor Status registers, it operates properly in the event the processor takes an interrupt or trap—unrelated to the Trace Facility—before the above trace sequence completes. When the unrelated interrupt or trap is taken, the state of the Trace Facility (i.e., the values of the TE and TP bits) is copied into the Old Processor Status from the Current Processor Status. The Trace Facility then resumes operation when the interrupted routine is restarted by an interrupt return.

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.

17.2 INSTRUCTION BREAKPOINTS

The HALT instruction can be used as an instruction breakpoint by a 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 as outlined in Section 17.6.5

The Assert class of instructions and the Illegal Opcode trap can be used by software to implement instruction breakpoints. An instruction breakpoint is set by replacing an instruction with the Assert instruction or an illegal opcode in the program under test. When the breakpoint instruction is encountered, the instruction breakpoint causes a trap. The illegal opcode is preferred since the Program Counter 1 (PC1) points to the illegal opcode when the trap is taken, whereas PC1 points to the instruction following the breakpoint if an Assert is used.

17.3 PROCESSOR STATUS OUTPUTS

The STAT(2–0) outputs indicate certain information about processor modes along with 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	Reserved
0	1	0	Load Test Instruction Mode, Halt/Freeze
0	1	1	Wait Mode
1	0	0	External data access (data valid)
1	0	1	External instruction access (instruction valid)
1	1	0	Internal data access (data valid)
1	1	1	Idle or data/instruction not valid

The STAT(2–0) outputs are a delayed indication of a mode or condition, so a mode or condition on a given cycle is reflected on STAT(2–0) on the second following cycle. For example, STAT(2–0)=101 indicates an instruction was valid on ID(31–0) at the end of the second previous cycle. The R/$\overline{W}$ output indicates the direction of an access, adding information about the access indicated on STAT(2–0) (the R/$\overline{W}$ signal appears with the access and is not delayed as are STAT(2–0)). If an access is extended by $\overline{WAIT}$, the appropriate status is shown for every additional cycle until the access does complete. The address always appears on A(23–0), whether the access is a read or a write and whether the access is external or internal (that is, to an internal peripheral). The data appears on ID(31–0), except on a read of an internal peripheral.

17.4 CONTROL SIGNALS IN SCAN PATH

The Am29200 microprocessor incorporates a boundary scan interface is compatible with the IEEE 1149.1 JTAG specification. This interface permits access to testing and debug features of the processor core not visible on the external interface (see Section 17.5.2).

A two-bit CNTL field appears on the boundary-scan register (for readers familiar with the 29K Family, the CNTL field corresponds to the CNTL(1–0) pins that appear on other family processors). This field can be used to halt, step, and start the processor,

as well as force the processor to execute instructions for the purpose of testing and debugging. The CNTL field affects processor operations as follows:

CNTL Value	Mode
00	Load Test Instruction
01	Step
10	Halt
11	Normal

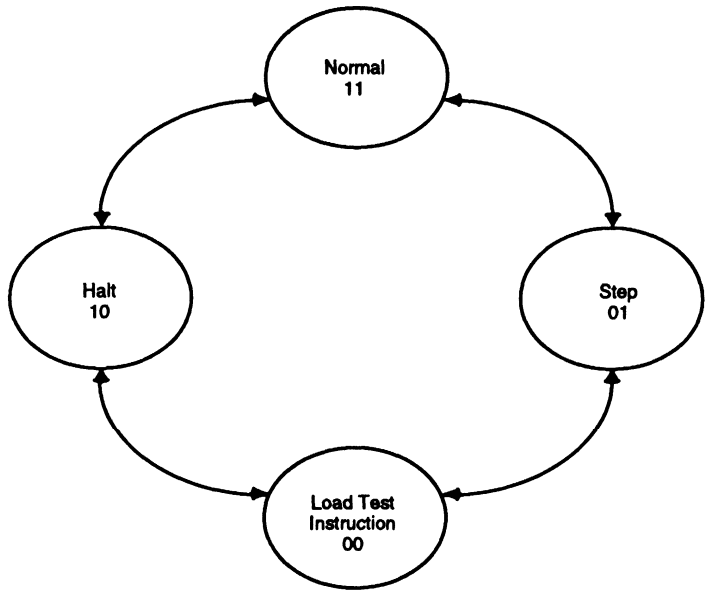
Changes to the CNTL field are restricted so only one bit of the CNTL field may change at any given time. The allowed transitions are shown in Figure 17-1. If these restrictions are violated, processor operation is unpredictable, and a processor reset is required to resume predictable operation.

Because of the restrictions just described, it is not possible to transition directly between all possible modes controlled by the CNTL field. For example, the processor cannot go from the Load Test Instruction mode to Normal operation without first entering the Halt or Step modes.

17.5 TEST ACCESS PORT

The Am29200 microprocessor implements the Standard Test Access Port (TAP) and Boundary-Scan Architecture as specified by the IEEE Specification 1149.1–1990 (JTAG), with the exception that the INCLK pin is not part of the boundary-scan register. The IEEE 1149.1–1990 Specification includes many details omitted from the discussion in this section and is included by reference. The following description discusses Am29200 microprocessor-specific considerations.

Figure 17-1 Valid Transitions for CNTL Field



17.5.1 Boundary Scan Cells

The Test Access Port can access, affect, and sample the processor inputs and outputs because a Boundary Scan Register (BSR) and Parallel Data Register (PDR) are incorporated into the design of the input and output cells. The Boundary Scan Register allows serial data to be loaded into or read out of the processor input/output boundary. The Parallel Data Register holds data stable at inputs and outputs during scanning, so system signals are not adversely affected during scanning.

An input or output cell incorporating a BSR and PDR register bit is referred to as a boundary scan cell. This section describes the implementation of the Am29200 boundary scan cells.

Figure 17-2 shows the design of an input boundary scan cell, and Figure 17-3 shows the design of an output boundary scan cell. Bidirectional signals use both of these designs in the same cell. Multiplexor selects, when active, select the lower multiplexor input.

The Shift and Update clocks, when used to sample or drive processor and system signals, are synchronized to the processor internal clocks so all signals (except the TAP signals) are sampled or driven synchronously to system clocks. However, the Shift and Update clocks still satisfy the JTAG constraints that inputs are sampled after the rising edge of TCK, outputs change after the falling edge of TCK, and TCK is the only control needed to affect sampling and driving.

The IEEE 1149.1–1990 Specification requires that it be possible to force the processor three-state outputs to be enabled. This is accomplished by cells that have no associated pin. The outputs of these cells force groups of output drivers to be enabled. Some outputs can be disabled by these cells even though the outputs cannot be disabled during normal operation (for example, the A(23–0) outputs can be disabled).

The boundary scan cells for the CNTL(1–0) field and STAT(2–0) outputs are part of the BSR and are accessible by scanning the BSR. However, they can also be scanned individually using the *ICTEST1* instruction (see Section 17.5.2).

Figure 17-2 Input Boundary-Scan Cell

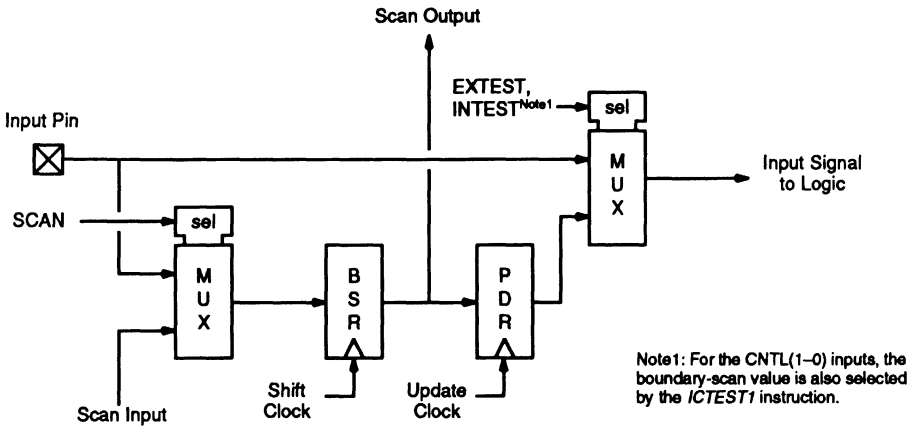
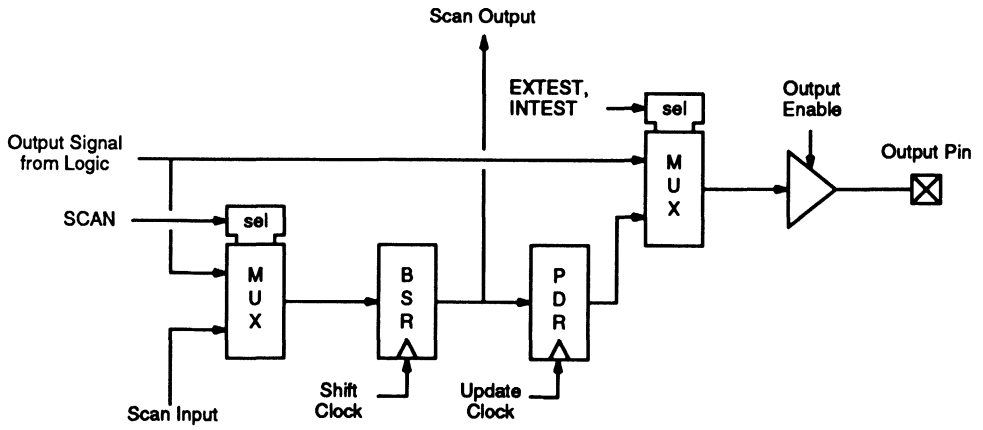


Figure 17-3 Output Boundary-Scan Cell



If the *ICTEST1* instruction is active, no other boundary scan cell is scanned. However, the contents of the other scan cells are undefined after this operation.

The INCLK input is not a boundary scan cell. The clocks to the processor must continue to operate even if the Test Access Port is active. However, a fault on this input is readily visible in the operation of the Test Access Port.

The MEMCLK pin has an output boundary scan cell. The *EXTEST* and *INTEST* instructions hold the MEMCLK signal at a fixed logic level for the duration of the instruction.

17.5.2 Instruction Register and Implemented Instructions

The Instruction Register (IREG) of the Test Access Port is a 3-bit register. The least significant bit (IREG0) is the bit nearest the TDO output. Instructions are encoded as follows:

IREG2	IREG1	IREG0	Instruction
0	0	0	EXTEST
0	0	1	Preloaded value (acts like BYPASS)
0	1	0	ICTEST2
0	1	1	Reserved (acts like BYPASS)
1	0	0	INTEST
1	0	1	SAMPLE
1	1	0	ICTEST1
1	1	1	BYPASS

The *EXTEST*, *BYPASS*, *INTEST*, and *SAMPLE* instructions are specified by the 1149.1–1990 Specification. Reserved instructions behave as *BYPASS* instructions to conform to the specification. *ICTEST1 and ICTEST2* are AMD® public instructions.

Most of these instructions are described in detail in the IEEE 1149.1–1990 Specification. Below is a brief description of the special considerations in the Am29200 implementation.

17.5.2.1 EXTEST

The *EXTEST* instruction is provided for external continuity and logic tests. It allows the Test Access Port to drive outputs and sample inputs.

EXTEST selects the BSR for scanning. During execution:

1. Processor outputs are driven from the PDR.
2. Processor internal output signals are sampled into the BSR. This is default behavior.
3. Processor inputs are sampled into the BSR.
4. Processor internal input signals are driven from the PDR. This prevents internal logic from seeing invalid combinations of input signals possibly received from other chips during the test.

17.5.2.2 INTEST

The *INTEST* instruction is provided to test the processor's internal logic. Its primary value is to allow a hardware-development system to drive the processor's Test Interface without a direct electrical connection to all pins of the package.

INTEST selects the BSR for scanning. During execution:

1. Processor outputs are driven from the PDR. This prevents external logic from seeing invalid combinations of output signals.
2. Processor internal output signals are sampled into the BSR.
3. Processor inputs are sampled into the BSR. This is default behavior.
4. Processor internal input signals are driven from the PDR.

The *INTEST* instruction allows the hardware-development system to alter and inspect internal registers, using processor load and store instructions, without having the external system see any bus activity.

17.5.2.3 SAMPLE

The *SAMPLE* instruction is provided to inspect the processor's external signals without interfering with system operations.

SAMPLE selects the BSR for scanning. During execution:

1. Processor outputs are driven by the processor.
2. Processor internal output signals are sampled into the BSR.
3. Processor inputs are sampled into the BSR.
4. Processor internal input signals are driven from the processor inputs.

17.5.2.4 ICTEST1

The *ICTEST1* instruction is defined for AMD processors using the extension mechanisms permitted by the IEEE 1149.1–1990 Specification. It is provided to drive the CNTL field and sample the STAT(2–0) outputs while leaving other inputs and outputs in their normal system connection. This allows a hardware-development system to control the processor and system using the Test Access Port.

ICTEST1 selects a subset of the BSR for scanning. During execution:

1. Processor outputs are driven by the processor.

2. Processor internal output signals are sampled into the BSR. This is default behavior for most signals, but allows the sampling of STAT(2–0).
3. Processor input signals are sampled into the BSR. This is default behavior.
4. The processor CNTL field is driven by the PDR. Processor internal inputs are driven from the processor inputs.

17.5.2.5 ICTEST2

The *ICTEST2* instruction is defined for AMD processors using the extension mechanisms permitted by IEEE 1149.1–1990. *ICTEST2* is similar to *EXTEST* with the exception that the scan path for *ICTEST2* excludes most of the processor outputs so the system is not disrupted (for example, by interfering with refresh). This allows a hardware-development system to access and modify processor internal state without disrupting the system.

1. Processor ID(31–0) and STAT(2–0) outputs are driven from the PDR. The output enable for the ID Bus is controlled by the PDR. Other processor outputs are controlled by the processor.
2. Processor internal output signals for ID(31–0) and STAT(2–0) are sampled into the BSR. This allows a hardware-development system to sample the processor's status and data driven by the processor.
3. Processor internal input signals for ID(31–0) are driven from the PDR. This allows a hardware-development system to provide data to the processor independent of system controls.

17.5.2.6 BYPASS

The *BYPASS* instruction is provided to bypass the BSR and shorten access times to other devices at the board level.

BYPASS selects the Bypass Register for scanning. The processor is not otherwise affected.

17.5.3 Order of Scan Cells in Boundary Scan Path

This section documents the scan paths and the order of scan cells in the paths. The cells are listed in order from TDI to TDO. In the Am29200 microprocessor, there are five scan paths from TDI to TDO: 1) the instruction path, 2) the bypass path, 3) the main data path, 4) the *ICTEST1* path, and 5) the *ICTEST2* path.

17.5.3.1 INSTRUCTION PATH

This is a 3-cell path which is used to scan into the Instruction Register. When the instruction path is selected the captured data is always IREG(2–0) = 001 and the instruction is set by scanning. The preloaded pattern 001 is used to test for faults in the boundary scan connections at the board level. The instructions are specified in Section 17.5.2.

Bit	Cell Name
1	IREG2
2	IREG1
3	IREG0

17.5.3.2 BYPASS PATH

This is a one-cell path which is used to bypass the processor and shorten access to other devices at the board level. When the bypass path is selected, the captured data is always 0 and the scan in data has no effect on the processor.

17.5.3.3 MAIN DATA PATH

This is a 188-cell path used to access the processor pins. This path is divided into five sets of cells. Where applicable, each set has a cell which enables the outputs of the set to be driven on the processor's pins. These cells are not connected to a processor pin. For convenience, the drive enable cells are shown in boldface. Some of these cells affect outputs not normally enabled and disabled during normal system operation. The sets of cells are divided logically as follows: 1) clocks, requests, and reset, 2) miscellaneous peripheral control signals, 3) memory and peripheral controls, 4) instruction/data bus.

Bit	Cell Name	Comments
1	MEMCLK	The MEMCLK scan cell is an output scan cell: it captures processor internal MEMCLK and substitutes the scanned value for the output.
2	$\overline{\text{RESET}}$	
3	LSYNC	
4	VCLK	
5	$\overline{\text{WARN}}$	
6	$\overline{\text{INTR3}}$	
7	$\overline{\text{INTR2}}$	
8	$\overline{\text{INTR1}}$	
9	$\overline{\text{INTR0}}$	
10	$\overline{\text{TRAP1}}$	
11	$\overline{\text{TRAP0}}$	
12	TDMA	
13	DREQ0	
14	DREQ1	
15	$\overline{\text{GREQ}}$	
16	**TOPDRV**	Enables the drivers for PSYNC through PWE
17	PSYNCI	PSYNC input
18	PSYNCO	PSYNC output
19	VDATI	VDAT input
20	VDATO	VDAT output
21	STAT0	
22	STAT1	
23	STAT2	
24	PIOI0	PIO0 input
25	PIOO0	PIO0 output
26	PIOI1	PIO1 input
27	PIOO1	PIO1 output
.	.	
.	.	
54	PIOI15	PIO15 input
55	PIOO15	PIO15 output
56	$\overline{\text{PBUSY}}$	
57	$\overline{\text{PACK}}$	
58	$\overline{\text{POE}}$	
59	$\overline{\text{PWE}}$	
60	PSTROBE	
61	PAUTOFD	
62	$\overline{\text{WAIT}}$	
63	BOOTW	

Bit	Cell Name	Comments
64	**ABIDRV**	Enables the driving of the A(23–0) outputs
65	A0	
66	A1	
.		
.		
88	A23	
89	**BOTDRV**	Enables the drivers for $\overline{\text{DACK0}}$ through $\overline{\text{DSR}}$
90	$\overline{\text{DACK0}}$	
91	$\overline{\text{DACK1}}$	
92	R/$\overline{\text{W}}$	
93	$\overline{\text{PIAOE}}$	
94	$\overline{\text{PIAWE}}$	
95	$\overline{\text{PIACS0}}$	
96	$\overline{\text{PIACS1}}$	
97	$\overline{\text{PIACS2}}$	
98	$\overline{\text{PIACS3}}$	
99	$\overline{\text{PIACS4}}$	
100	$\overline{\text{PIACS5}}$	
101	$\overline{\text{GACK}}$	
102	$\overline{\text{WE}}$	
103	$\overline{\text{TR}}$	
104	$\overline{\text{CAS0}}$	
105	$\overline{\text{CAS1}}$	
106	$\overline{\text{CAS2}}$	
107	$\overline{\text{CAS3}}$	
108	$\overline{\text{RAS0}}$	
109	$\overline{\text{RAS1}}$	
110	$\overline{\text{RAS2}}$	
111	$\overline{\text{RAS3}}$	
112	$\overline{\text{ROMOE}}$	
113	$\overline{\text{RSWE}}$	
114	$\overline{\text{BURST}}$	
115	$\overline{\text{ROMCS0}}$	
116	$\overline{\text{ROMCS1}}$	
117	$\overline{\text{ROMCS2}}$	
118	$\overline{\text{ROMCS3}}$	
119	$\overline{\text{TXD}}$	
120	$\overline{\text{DSR}}$	
121	UCLK	
122	$\overline{\text{RXD}}$	
123	$\overline{\text{DTR}}$	
124	**DBIDRV**	Enables the ID Bus drivers
125	IDI0	ID0 input
126	IDO0	ID0 output
127	IDI1	ID1 input
128	IDO1	ID1 output
187	IDI31	ID31 input
188	IDO31	ID31 output

17.5.3.4 ICTEST1 PATH

This is a 5-bit path used to provide quick access to the CNTL field and the STAT(2–0) output signals while keeping other inputs and outputs in their normal system connection.

Bit	Cell Name	Comments
1	CNTL0	Internal control field only
2	CNTL1	
3	STAT0	Outputs: These signals are scanned out and are shown on the TDO pin. The scan in values do not replace the processor output values. In ICTEST1, the processor outputs STAT(2–0) continue to reflect the internal processor signals.
4	STAT1	
5	STAT2	

If the *ICTEST1* path is scanned, the contents of the shift register bits in the other scan cells become undefined. This occurs because all scan paths share the same shift clocks

17.5.3.5 ICTEST2 PATH

The *ICTEST2* path includes only the ID Bus, the CNTL field, and the STAT(2–0) signals. It is provided so a hardware-development system can access the processor without disrupting the system.

Bit	Cell Name	Comments
1	CNTL0	Internal control field only
2	CNTL1	
3	STAT0	Outputs: These signals are scanned out and are shown on the TDO pin. The scan in values do not replace the processor output values. In ICTEST1, the processor outputs STAT(2–0) continue to reflect the internal processor signals.
4	STAT1	
5	STAT2	
6	*DBIDRV*	Enables the ID Bus drivers
7	IDI0	ID0 input
8	IDO0	ID0 output
9	IDI1	ID1 input
10	IDO1	ID1 output
69	IDI31	ID31 input
70	IDO31	ID31 output

17.6 IMPLEMENTING A HARDWARE-DEVELOPMENT SYSTEM

The Halt, Step, and Load Test Instruction modes of operation, invoked using the CNTL field in the boundary scan path, are defined to support the debugging of the processor system by a hardware-development system (both hardware and software debug). This section describes the use of these modes during debug and describes the corresponding activity on the CNTL field and STAT(2–0) pins.

17.6.1 Halt Mode

The Halt mode allows the hardware-development system to stop processor operation while preserving its internal state. The Halt mode is defined so normal operation may resume from the point the processor enters the Halt mode. All external accesses are

completed before the Halt mode is entered, so a minimum amount of system logic is required to support the Halt mode.

The Halt mode can be invoked by applying a value of 10 to the CNTL field. The processor enters the Halt mode within two or three cycles after the CNTL field is changed (depending on synchronization time), except it first completes any external data access in progress.

The Halt mode can also be entered as the result of executing a HALT instruction. When a HALT instruction is executed, the processor enters the Halt mode on the next cycle, except it completes any external data accesses in progress. In this case, the processor remains in the Halt mode even though the CNTL field is 11. However, the processor cannot exit the Halt mode except as the result of the CNTL field or $\overline{\text{RESET}}$ input. If the instruction following a Halt instruction has an exception (e.g., instruction mapping miss), the trap associated with the exception is taken before the processor enters the Halt mode.

The Halt instruction is designed as an instruction breakpoint by the hardware-development system. However, the Halt instruction is normally 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 as described in Section 17.6.5.

In most cases, the STAT(2–0) outputs have a value of 000 whenever the processor is in the Halt mode. These outputs can be used to verify the processor is in Halt mode. However, the STAT(2–0) outputs have a value of 010 if the Freeze (FZ) bit of the Current Processor Status Register is 1 when the Halt mode is entered. This indicates the visible registers do not reflect the current program state.

While in the Halt mode, the processor does not execute instructions and performs no external accesses. The Timer Facility does not operate (i.e., the Timer Counter Register does not change).

The Halt mode is exited when the Reset mode is entered or the CNTL field places the processor into another mode. The only valid transitions on the CNTL field from the value of 10 are to the value 00, which places the processor into the Load Test Instruction mode, or to the value 11, which causes the processor to resume normal execution.

17.6.2 Step Mode

The Step mode causes the Am29200 microprocessor 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 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 value of 01 in the CNTL field. The processor enters the Step mode within two or three cycles after the CNTL field is changed (depending on synchronization time), except it first completes any external data access in progress.

In most cases, the STAT(2–0) outputs have a value of 000 whenever the processor is in the Step mode; these outputs can be used as a verification the processor is in Step mode. However, the STAT(2–0) outputs have a value of 010 if the Freeze (FZ) bit of the Current Processor Status Register is 1 when the Step mode is entered. This indicates the visible registers do not reflect the current program state.

While in the Step mode, the processor does not execute instructions and performs no external accesses. The Timer Facility does not operate (i.e., the Timer Counter Register does not change) while the processor is in the Step mode.

The Step mode is identical to the Halt mode in every respect except one. This difference is apparent on the transition of the CNTL field 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.

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 field has the value 11 (this prevents the need for a time-critical transition on the CNTL field). The processor remains in this condition until the CNTL field transitions to 10 or 01 (or $\overline{\text{RESET}}$ is asserted). The transition to 10 causes the processor to enter the Halt mode and is used to clear the Step mode. The transition to 01 causes the processor to remain in the Step mode so it may perform additional steps.

If the processor is placed in the Halt or Step mode while either a LOADM or STOREM instruction is being executed, the STAT(2–0) outputs indicate the Halt or Step mode for one cycle (STAT(2–0) = 000). They then indicate the Pipeline Hold mode (STAT(2–0) = 001) until the final access of the LOADM or STOREM is complete, at which time they return to indicating the Halt or Step mode. A hardware-development system must therefore ignore any single-cycle Halt/Step mode indication on the STAT(2–0) outputs as an indication the processor is halted.

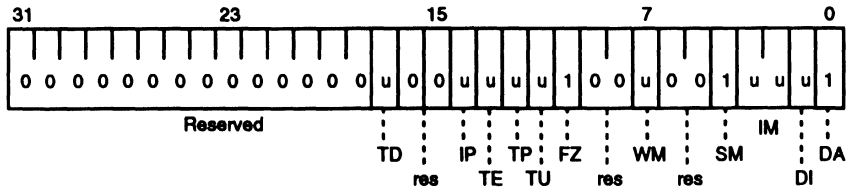
17.6.3 Load Test Instruction Mode

The processor incorporates an Instruction Register (IR) that holds instructions while they are decoded. In the Load Test Instruction mode, the IR is enabled to receive the content of the Instruction Bus regardless of the state of the processor's instruction fetcher. This allows the hardware-development system to provide instructions for execution directly, thereby providing means for the hardware-development system to examine and modify the internal state of the processor without altering the processor's instruction stream.

The hardware-development system can place an instruction in the IR by first placing 00 in the CNTL field. The processor enters the Load Test Instruction mode within two or three cycles after the CNTL field is changed (depending on synchronization time). However, it first completes and terminates any established burst-mode instruction access. The Load Test Instruction mode can be entered only from the Halt or Step modes.

When the processor enters the Load Test Instruction Mode, the processor behaves as though the Current Processor Status Register were forced to the value shown in Figure 17-4, even though the register is not changed (the value "u" means unaffected).

Figure 17-4 Processor Status While in Load Test Instruction Mode



The visible processor state remains unchanged while the processor is in the Load Test Instruction Mode. The processor status shown in Figure 17-4 remains in effect until the next transition to the Normal Mode via the Halt Mode.

While the processor is in the Load Test Instruction mode, it ignores all interrupts and traps, except for the $\overline{\text{WARN}}$ trap.

The STAT(2–0) lines have a value of 010 while the processor is in the Load Test Instruction mode; this may be used as a verification that the processor is loading the IR.

While the processor is in the Load Test Instruction mode, the IR is continually storing the value on the Instruction/Data Bus; any change in the value on this bus is reflected in the IR on the next cycle. The hardware-development system can place a desired instruction into the IR by driving this instruction on the Instruction/Data Bus or via the scan interface.

The processor exits the Load Test Instruction mode in the second cycle following a change to the CNTL field. The only valid change here is either to the Halt mode (CNTL = 10) or the Step mode (CNTL = 01).

When the Load Test Instruction mode is exited, the most recent value stored into the IR is held. If the processor is placed in the Step mode, the IR is marked as having valid content, enabling the processor to decode and execute the instruction. If the processor is placed in the Halt mode, it ignores any instruction placed in the IR by the Load Test Instruction mode and reverts to its normal instruction-fetch mechanism.

Once the IR has been set by the Load Test Instruction mode, the instruction in the IR may be executed via the Step mode as discussed in the previous section. A single step is sufficient to cause the execution of this instruction. However, because of pipelining, multiple steps may be required before the instruction completes execution. If more than one step is performed, the processor executes the instruction in the IR on every step. If it is desired to step an instruction to completion without repeated execution, a NO-OP may be set into the IR (using the Load Test Instruction mode) after the first step.

The Load Test Instruction mode may be used to cause the execution of most processor instructions (restrictions are discussed below). This allows inspection and modification of the processor state.

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.

The contents of the Program Counter 0, Program Counter 1, Program Counter 2, Channel Address, Channel Data, Channel Control, and ALU Status registers are not

updated while instructions are executed via the Load Test Instruction mode, except explicitly by Move To Special Register instructions. Instructions executed using the Load Test Instruction mode may access the protected processor state even though the processor is in the User mode.

Instructions executed via the Load Test Instruction mode may be used to access an external device or memory. Recall that the processor completes any normal data access before completing a step. This allows the processor to access devices and memories on behalf of the hardware-development system and simplifies the timing constraints on the hardware-development system.

During processor execution via the Load Test Instruction mode, the processor retains the information required to resume normal operation. If any processor state is modified by the hardware-development system, this state must be restored properly for normal operation to resume properly.

Once all instructions have been executed via the Load Test Instruction mode, the Halt mode (CNTL=10) prepares the processor to resume normal operation. When the CNTL field transitions to 11, the processor resumes normal operation. The sequence for the CNTL field to clear the Load Test Instruction mode and resume normal operation is thus 00/10/11.

17.6.4 Accessing Internal State Via Boundary Scan

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 and are directed to the ROM address space (for example, address 0): this causes the processor to prevent the resulting access from appearing in the system. The access is visible only via the boundary-scan register. Furthermore, it causes the Am29200 microprocessor to ignore the generation of wait states: the access completes at the end of the next stepped instruction. This provides a means for a hardware-development system to perform accesses.

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.

This section describes the sequence of boundary-scan operations performed to access processor internal state.

17.6.4.1 INSPECTING STATE VIA BOUNDARY SCAN

A hardware-development system uses store instructions to inspect the contents of general-purpose registers. Since the processor internal state can be moved to general-purpose registers, this provides a means to inspect other states as well as the values in general-purpose registers.

With the processor in the Halt mode, the hardware-development system uses the following sequence to retrieve the value in a general-purpose register:

1. Set the CNTL field to 10 (Halt) using the ICTEST1 boundary-scan instruction.
2. Set the CNTL field to 00 (Load Test Instruction) using the ICTEST1 instruction.
3. Using the ICTEST2 instruction, set the IDI(31–0) cells with an instruction to store the desired register into the ROM address space, with OPT=110, and set the

CNTL field to 01 (Step). This places the store instruction into the IR and prepares the processor to step.

4. Using the ICTEST1 instruction, sequence the CNTL field through the values 11, 01, 00 (Normal, Test, Load Test Instruction). This steps the processor and prepares it to receive another instruction.
5. Using the ICTEST2 instruction, set the IDI(31–0) cells to 70400101, hexadecimal (NO-OP), and set the CNTL field to 01. This loads a NO-OP into the IR.
6. Set the CNTL field to 11, then back to 01 using the ICTEST1 instruction. This steps the processor. At the end of the step, the contents of the register are on the ID Bus, and may be obtained in the Capture-DR state of the TAP controller (this state is described in the IEEE 1149.1–1990 Specification). The value will be held on the ID Bus until the next step.
7. Repeat steps 2 through 6 for the remaining registers.

17.6.4.2 Altering State Via Boundary Scan

A hardware-development system uses load instructions to alter the contents of general-purpose registers. Since the contents of general-purpose registers can be moved to special-purpose registers, this provides a means to alter other state as well as the values in general-purpose registers.

With the processor in the Halt mode, the hardware-development system uses the following sequence to modify the value in a general-purpose register:

1. Set the CNTL field to 10 (Halt) using the ICTEST1 boundary-scan instruction.
2. Set the CNTL field to 00 (Load Test Instruction) using the ICTEST1 instruction.
3. Using the ICTEST2 instruction, set the IDI(31–0) cells with an instruction to load the desired register from the ROM address space, with OPT=110, and set the CNTL field to 01 (Step). This places the load instruction into the IR and prepares the processor to step.
4. Using the ICTEST1 instruction, sequence the CNTL field through the values 11, 01, 00 (Normal, Test, Load Test Instruction). This steps the processor and prepares it to receive another instruction.
5. Using the ICTEST2 instruction, set the IDI(31–0) cells to 70400101, hexadecimal (NOOP), and set the CNTL field to 01. This loads a NO-OP into the IR.
6. Using the ICTEST2 instruction, set the IDI(31–0) cells to the value to be loaded and set the CNTL field to 11. This steps the processor and applies the value to be loaded into the register.
7. Set the CNTL field to 01 using the ICTEST1 instruction.
8. Repeat steps 2 through 7 for the remaining registers.

17.6.5 HALT Instructions as Breakpoints

The HALT instruction can be used by a hardware-development system to implement an instruction breakpoint. To use the HALT instruction as an instruction breakpoint, the hardware-development system must disable the protection checking that normally applies to the HALT instruction so the HALT does not cause a Protection Violation trap. To accomplish this, the hardware-development system must perform a special sequence of operations on the boundary-scan interface. This sequence is similar in effect to holding the CNTL(1–0) inputs at 10 during a reset on other 29K Family processors. The special sequence is needed in the Am29200 microprocessor because it

has no CNTL(1–0) inputs, but rather implements a CNTL field in the boundary-scan register. The following sequence disables protection checking on the HALT instruction:

1. Set the CNTL field to 10 using the ICTEST1 JTAG instruction.
2. Reset the boundary-scan cells $\overline{\text{RESET}}$, DBIDRV, BOTDRV, ABIDRV, and TOPDRV to 0 using the INTEST instruction. If the boot ROM in Bank 0 is 8 or 16 bits wide, reset the BOOTW cell to 0. If the boot ROM is 32 bits wide, set the BOOTW cell to 1. This resets the processor.
3. Set the $\overline{\text{RESET}}$cell to 1 using the INTEST instruction. If the boot ROM is 8 bits wide, set the BOOTW cell to 1 (otherwise leave it at 0 or 1). This takes the processor out of reset and configures the boot ROM.
4. Set the CNTL field to 00 using the ICTEST1 instruction.
5. Using the ICTEST2 instruction, set the IDI(31–0) cells to a1000000, hexadecimal, and set the CNTL field to 01. This loads a jump to address 0 into the IR and prepares the processor to step.
6. Using the ICTEST1 instruction, sequence the CNTL field through the values 11, 01, and 00. This steps the processor and prepares it to receive another instruction.
7. Using the ICTEST2 instruction, set the IDI(31–0) cells to 70400101, hexadecimal. This loads a NO-OP into the IR.
8. Using the ICTEST1 instruction, sequence the CNTL field through the values 11, 01, and 11. This starts the processor with protection checking disabled on the HALT instruction. The TMS input must be kept High during this operation.

17.6.6 Forcing Outputs to High Impedance

A hardware-development system can force processor outputs to the high-impedance state by asserting the $\overline{\text{GREQ}}$ input during a processor reset. The outputs remain in the high-impedance state until a processor reset during which $\overline{\text{GREQ}}$ is not asserted. This supports functions such as replacing chip signals by emulator signals.

CHAPTER 18

INSTRUCTION SET

This chapter provides a specification of the Am29200 instruction set. Sections 18.1 and 18.2 describe the terminology and the instruction formats. Section 18.3 describes each instruction in detail; instructions are presented alphabetically by assembler mnemonic. Finally, Section 18.4 gives an index of instructions by operation code.

18.1 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 with obvious meanings.

18.1.1 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 involved in a word operation.
COUNT	The value of the Count Remaining field of the Channel Control Register. Note that COUNT does not refer to this field directly, but rather to the value of the field at the beginning of a LOADM or STOREM instruction.
DEST	The general-purpose register that is the destination of an instruction (i.e., the register used to store the result).
EXTERNAL WORD[*n*]	The word in an external device or memory with address *n*.
FALSE	The Boolean constant FALSE.
FC	The Funnel Shift Count (FC) field of the ALU Status Register.
h'*n*'	The hexadecimal constant *n*.
I16	16-bit immediate data.
IPA	Indirect Pointer A Register.

IPB	Indirect Pointer B Register.
IPC	Indirect Pointer C Register.
PC	The Program Counter Register. This register is not explicitly accessible by instruction, but does appear as an operand for certain instructions. The Program Counter always contains the word address of the instruction being executed, and is 30 bits in length.
Q	The Q Register.
Register RA Register RB Register RC	These designate the general-purpose registers specified by the instruction fields RA, RB, and RC (see Section 18.2).
SPDEST	The special-purpose register that is the destination of an instruction.
SPECIAL	The contents of a special-purpose register, used as an instruction operand.
Special-purpose Register SA	Designates the special-purpose register specified by the instruction field SA (see Section 18.2).
SRCA SRCB	The contents of general-purpose registers, used as instruction operands.
SRCA.BYTE*n* SRCB.BYTE*n*	Designate the byte numbered *n* within the SRCA or SRCB operand.
TARGET	The target-instruction address specified by a jump or call instruction. This address is either absolute or Program-Counter relative.
TRUE	The Boolean constant TRUE.
TWIN	General-purpose registers are paired by absolute-register numbers, 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.

18.1.2 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

18.1.3 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 16.2.2).

18.1.4 Assembler Syntax

This chapter does not contain a full description of the instruction assembler, but provides a rudimentary description of the assembler syntax.

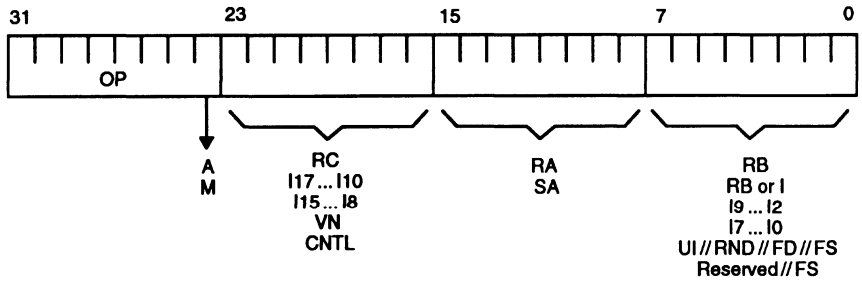
The following notation is used to describe assembler tokens:

cntl	Determines the 7-bit control field in a load or store instruction.
const8	Specifies a constant that can be expressed by 8 bits.
const16	Specifies a constant that can be expressed by 16 bits.
ra rb rc	These tokens name general-purpose registers. In a formal sense these represent the same token since the name of a register does not depend on its instruction use. However, three distinct tokens are used to clarify the relationship between the assembler syntax, instruction operands, and instruction fields.
spid	A symbolic identifier for a special-purpose register.
target	A symbolic label for the target of a jump or call instruction.
vn	Specifies a trap vector number.

18.2 INSTRUCTION FORMATS

All instructions for the Am29200 microprocessor are 32 bits in length and are divided into four fields, as shown in Figure 18-1. 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 18-1 Instruction Format



The instruction fields are defined as follows:

Bits 31–24

OP	This field contains an operation code, defining the operation to be performed. In some instructions, the least significant bit of the operation code selects between two possible operands. For this reason, the least significant bit is sometimes labeled A or M with the following interpretations:
A	(Absolute): The A bit is used to differentiate between Program-Counter relative (A = 0) and absolute (A = 1) instruction addresses when these addresses appear within instructions.
M	(Immediate): The M bit selects between a register operand (M = 0) and an immediate operand (M = 1) when the alternative is allowed by an instruction.

Bits 23–16

RC	The RC field contains a global or local register number.
I17...I10	This field contains the most significant eight bits of a 16-bit instruction address. This is a word address, and may be program-counter relative or absolute depending on the A bit of the operation code.
I15...I8	This field contains the most significant eight bits of a 16-bit instruction constant.
VN	This field contains an 8-bit trap vector number.
CNTL	This field controls a load or store access as described in Section 3.3.1

Bits 15–8

RA	The RA field contains a global or local register number.
SA	The SA field contains a special-purpose register number.

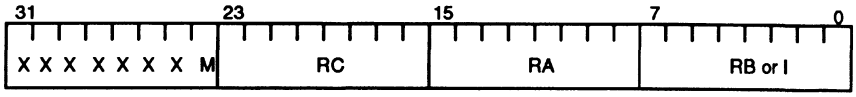
Bits 7–0

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.

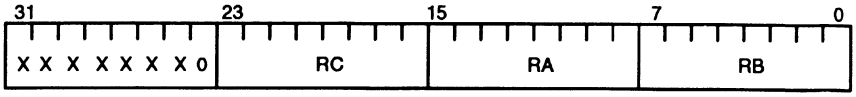
The fields described above may appear in many combinations. However, certain combinations that appear frequently are shown in Figure 18-2.

Figure 18-2 Frequently Occurring Instruction Field Uses

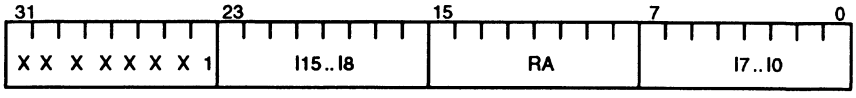
Three operands with possible 8-bit constant:



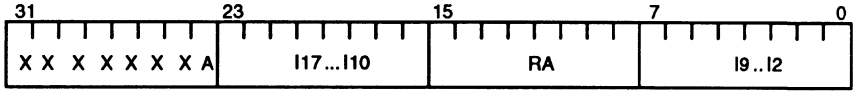
Three operands without constant:



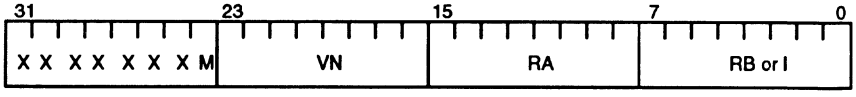
One register operand with 16-bit constant:



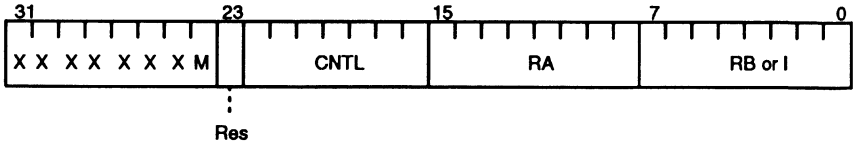
Jumps and calls with 16-bit instruction address:



Two operands with trap vector number:



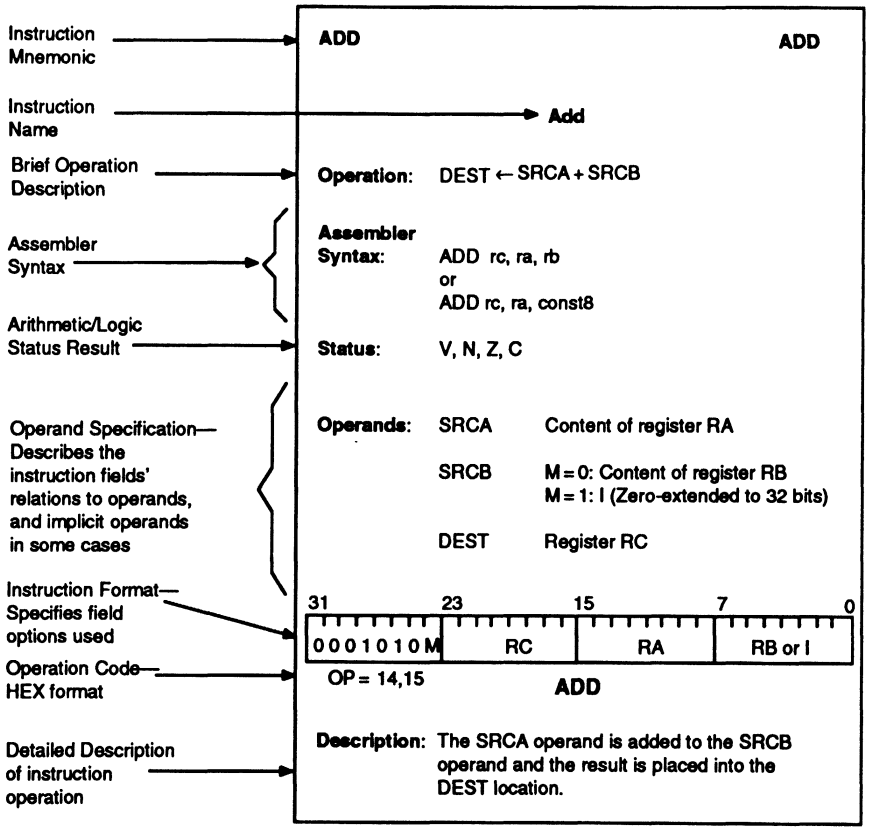
Loads and stores:



18.3 INSTRUCTION DESCRIPTION

This section describes each Am29200 instruction in detail. Figure 18-3 illustrates the layout of the information given for each description.

Figure 18-3 Instruction-Description Format



Add

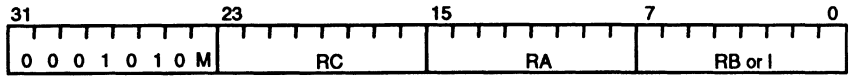
Operation: DEST ← SRCA + SRCB

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



OP = 14, 15 **ADD**

Description: The SRCA operand is added to the SRCB operand and the result is placed into the DEST location.

Add with Carry

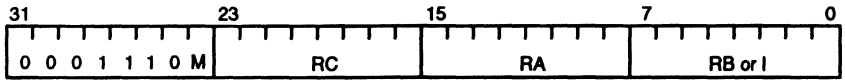
Operation: DEST ← SRCA + SRCB + C

Assembler Syntax:
ADDC rc, ra, rb
or
ADDC rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 1C, 1D

ADDC

Description: The SRCA operand is added to the SRCB operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location.

Add with Carry, Signed

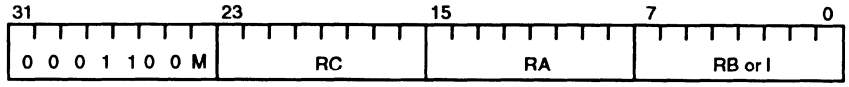
Operation: DEST ← SRCA + SRCB + C
IF signed overflow THEN Trap (Out of Range)

Assembler Syntax:
ADDCS rc, ra, rb
or
ADDCS rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 18, 19

ADDCS

Description: The SRCA operand is added to the SRCB operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location. If the add operation causes a two's-complement signed overflow, an Out-of-Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

Add with Carry, Unsigned

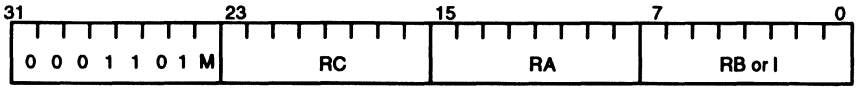
Operation: DEST ← SRCA + SRCB + C
IF unsigned overflow THEN Trap (Out of Range)

Assembler Syntax:
ADDCU rc, ra, rb
or
ADDCU rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 1A, 1B

ADDCU

Description: The SRCA operand is added to the SRCB operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location. If the add operation causes an unsigned overflow, an Out-of-Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

Add, Signed

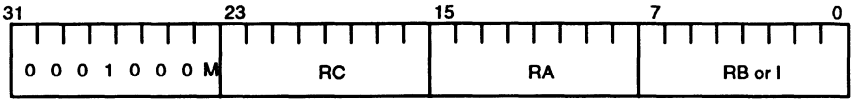
Operation: DEST ← SRCA + SRCB
IF signed overflow THEN Trap (Out of Range)

Assembler Syntax:
ADDS rc, ra, rb
or
ADDS rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 10, 11 **ADDS**

Description: The SRCA operand is added to the SRCB operand and the result is placed into the DEST location. If the add operation causes a two's-complement signed overflow, an Out-of-Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

Add, Unsigned

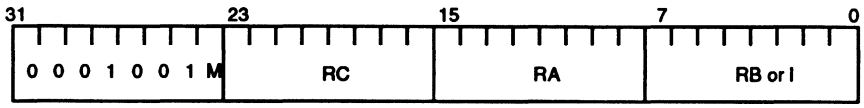
Operation: DEST ← SRCA + SRCB
IF unsigned overflow THEN Trap (Out of Range)

Assembler Syntax:
ADDU rc, ra, rb
or
ADDU rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 12, 13 **ADDU**

Description: The SRCA operand is added to the SRCB operand and the result is placed into the DEST location. If the add operation causes an unsigned overflow, an Out-of-Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

AND Logical

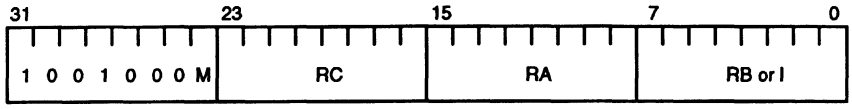
Operation: DEST ← SRCA & SRCB

Assembler Syntax:
AND rc, ra, rb
or
AND rc, ra, const8

Status: N, Z

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 90, 91 **AND**

Description: The SRCA operand is logically ANDed, bit-by-bit, with the SRCB operand and the result is placed into the DEST location.

AND-NOT Logical

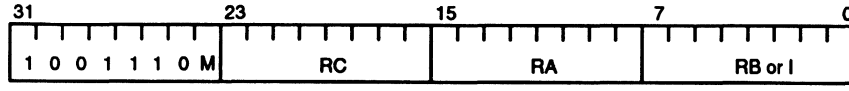
Operation: DEST ← SRCA & ~SRCB

Assembler Syntax:
ANDN rc, ra, rb
or
ANDN rc, ra, const8

Status: N, Z

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	register RC



OP = 9C, 9D

ANDN

Description: The SRCA operand is logically ANDed, bit-by-bit, with the one's-complement of the SRCB operand and the result is placed into the DEST location.

Assert Equal To

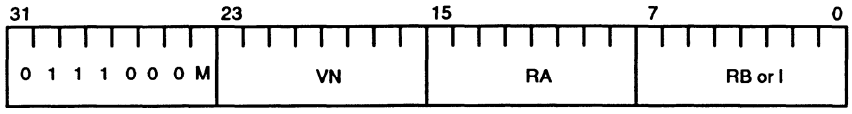
Operation: IF SRCA = SRCB THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASEQ vn, ra, rb
or
ASEQ vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 70, 71 **ASEQ**

Description: If the SRCA operand is equal to the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Greater Than or Equal To

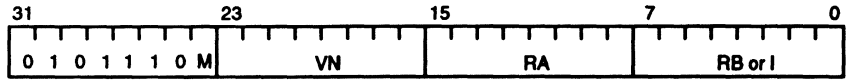
Operation: IF SRCA ≥ SRCB THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASGE vn, ra, rb
or
ASGE vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 5C, 5D

ASGE

Description: If the value of the SRCA operand is greater than or equal to the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Greater Than or Equal To, Unsigned

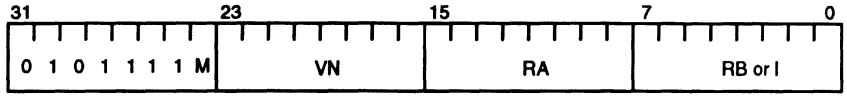
Operation: IF SRCA ≥ SRCB (unsigned) THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASGEU vn, ra, rb
or
ASGEU vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 5E, 5F

ASGEU

Description: If the value of the SRCA operand is greater than or equal to the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs. For the comparison, both operands are treated as unsigned integers.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Greater Than

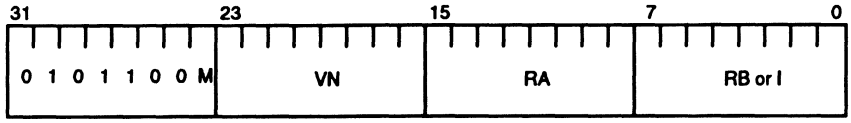
Operation: IF SRCA > SRCB THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASGT vn, ra, rb
or
ASGT vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 58, 59 **ASGT**

Description: If the value of the SRCA operand is greater than the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Greater Than, Unsigned

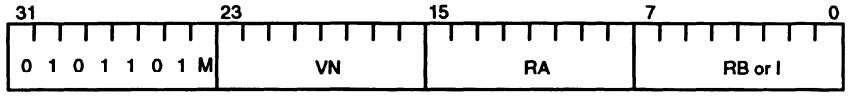
Operation: IF SRCA > SRCB (unsigned) THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASGTU vn, ra, rb
or
ASGTU vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 5A, 5B

ASGTU

Description: If the value of the SRCA operand is greater than the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs. For the comparison, both operands are treated as unsigned integers.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Less Than or Equal To

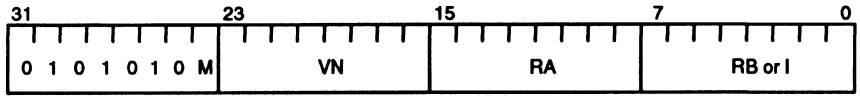
Operation: IF SRCA ≤ SRCB THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASLE vn, ra, rb
or
ASLE vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 54, 55 **ASLE**

Description: If the value of the SRCA operand is less than or equal to the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Less Than or Equal To, Unsigned

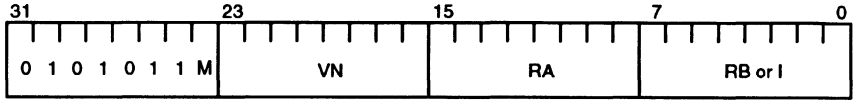
Operation: IF SRCA ≤ SRCB (unsigned) THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASLEU vn, ra, rb
or
ASLEU vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 56, 57

ASLEU

Description: If the value of the SRCA operand is less than or equal to the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs. For the comparison, both operands are treated as unsigned integers.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Less Than

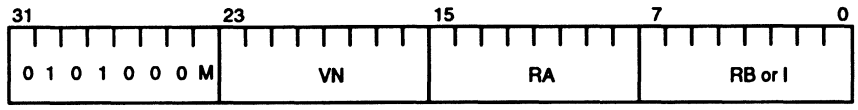
Operation: IF SRCA < SRCB THEN Continue
ELSE Trap(VN)

Assembler Syntax:
ASLT vn, ra, rb
or
ASLT vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 50, 51

ASLT

Description: If the value of the SRCA operand is less than the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Less Than, Unsigned

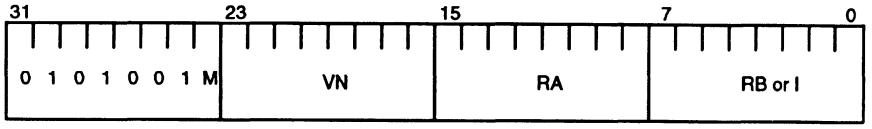
Operation: IF SRCA < SRCB (unsigned) THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASLTU vn, ra, rb
or
ASLTU vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 52, 53

ASLTU

Description: If the value of the SRCA operand is less than the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs. For the comparison, both operands are treated as unsigned integers.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Not Equal To

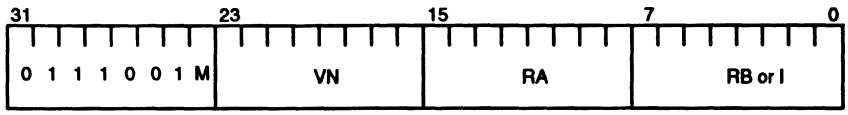
Operation: IF SRCA <> SRCB THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASNEQ vn, ra, rb
or
ASNEQ vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 72, 73

ASNEQ

Description: If the SRCA operand is not equal to the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Call Subroutine

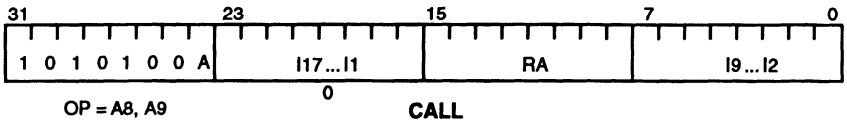
Operation: DEST ← PC // 00 + 8
PC ← TARGET
Execute delay instruction

Assembler Syntax: CALL ra, target

Status: Not affected

Operands:

TARGET	A = 0: I17 ... I10 // I9 ... I2 (sign-extended to 30 bits) + PC
	A = 1: I17 ... I10 // I9 ... I2 (zero-extended to 30 bits)
DEST	Register RA



Description: The address of the second following instruction is placed into the DEST location and a non-sequential instruction fetch occurs to the instruction address given by the TARGET operand. The instruction following the CALL is executed before the non-sequential fetch occurs.

Call Subroutine, Indirect

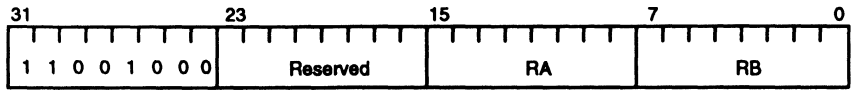
Operation: DEST ← PC // 00 + 8
PC ← SRCB
Execute delay instruction

Assembler Syntax: CALLI ra, rb

Status: Not affected

Operands:

SRCB	Content of register RB
DEST	Register RA



OP = C8

CALLI

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

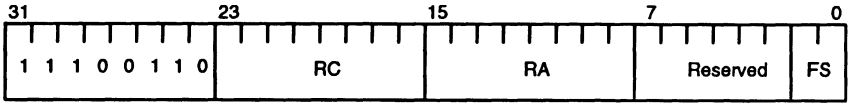
Status: None

Operands:

SRCA	Content of register RA (single-precision floating-point) or Content of register RA and the twin of register RA (double-precision floating-point)
DEST	Register RC

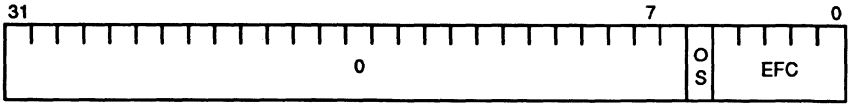
Control:

FS	Format of source operand SRCA
00	Reserved for future use
01	Single-precision floating-point
10	Double-precision floating-point
11	Reserved for future use



OP = E6 **CLASS**

Description: A 32-bit classification code for operand SRCA is placed into the DEST location. Operand SRCA is a single- or double-precision operand, as specified by FS. The classification code has the following format:



Bits 31–6: Reserved (forced to 0).

Bit 5: Operand Sign (OS). The OS bit is 1 for a negative operand (including negative zero) and 0 for a non-negative operand.

Bits 4–0: Exponent-Fraction Class (EFC). This field classifies the biased exponent and fraction fields of the source operand as follows:

EFC	Biased Exp (bexp)	Fraction (frac)	Comments
00000	0	0	zero
00001			unused
00010	0	0 < frac < .111 ... 1	denormalized
00011	0	.111...1	denormalized
00100	1		0
00101			unused
00110	1		0 < frac < .111 ... 1
00111	1	.111 ... 1	
01000	1 < bexp < Max	0	
01001			unused
01010	1 < bexp < Max	0 < frac < .111 ... 1	
01011	1 < bexp < Max	.111... 1	
01100	Max	0	
01101			unused
01110	Max	0 < frac < .111 ... 1	
01111	Max	.111 ... 1	
10000	Max + 1	0	infinity
10001			unused
10010	Max + 1, frac MSB = 0	<> 0	SNaN
10011	Max + 1, frac MSB = 1	<> 0	QNaN

Note: Max is the largest biased exponent used to represent a finite number in a given format. Max is 254 for single-precision and 2,046 for double-precision.

This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes a CLASS trap. When the trap occurs, the IPA and IPC registers are set to reference SRCA and DEST, and the IPB Register is set with the value of the FS field.

Count Leading Zeros

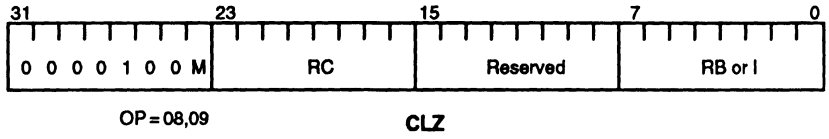
Operation: DEST ← count of number of leading zeros in SRCB or I

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



Description: A count of the number of zero-bits to the first one-bit in the SRCB operand is placed into the DEST location. If the most significant bit of the SRCB operand is 1, the resulting count is zero. If the SRCB operand is zero, the resulting count is 32.

Constant

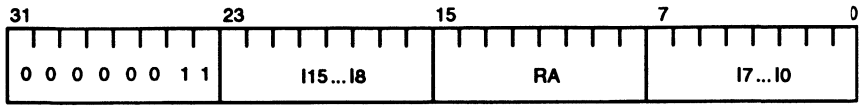
Operation: DEST ← 0I16

Assembler Syntax: CONST ra, const16

Status: Not affected

Operands:

0I16	I15 ... 8 // I7 ... I0 (Zero-extended to 32 bits)
DEST	Register RA



Description: The 0I16 operand is placed into the DEST location.

Note: To improve code readability, some assemblers implement CONST to take a 32-bit argument (rather than const16). The lower half of the argument is constructed by the CONST.

Constant, High

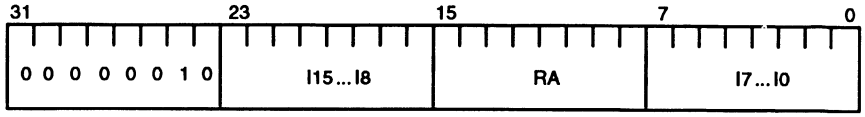
Operation: Replace high-order half-word of SRCA by I16

Assembler Syntax: CONSTH ra, const16

Status: Not affected

Operands:

SRCA	Content of register RA
I16	I15 ... I8 // I7 ... I0
DEST	Register RA



OP = 02

CONSTH

Description: The low-order half-word of the SRCA operand is appended to the I16 operand and the result is placed into the DEST operand. Note that the destination register for this instruction is the same as the source register.

Note: To improve code readability, some assemblers implement CONSTH to take a 32-bit argument (rather than const16). The upper half of the argument is constructed by the CONSTH.

Constant, Negative

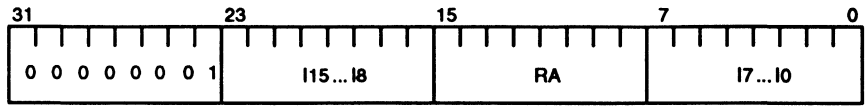
Operation: DEST ← I16

Assembler Syntax: CONSTN ra, const16

Status: Not affected

Operands:

I16	I15 ... I8 // I7 ... I0 (ones-extended to 32 bits)
DEST	Register RA



Description: The I16 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

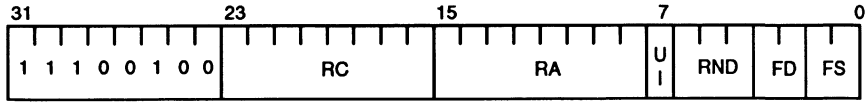
Status: fpX, fpU, fpV, fpR, fpN

Operands:

SRCA	Content of register RA (single-precision floating-point) or Content of register RA and the twin of register RA (double-precision floating-point)
DEST	Content of register RC (single-precision floating-point) or Content of register RC and the twin of register RA (double-precision floating-point)

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 floating-point 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

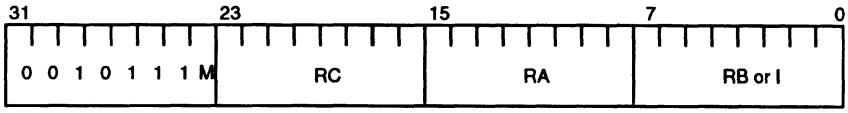
Operation: IF (SRCA.BYTE0 = SRCB.BYTE0) OR
(SRCA.BYTE1 = SRCB.BYTE1) OR
(SRCA.BYTE2 = SRCB.BYTE2) OR
(SRCA.BYTE3 = SRCB.BYTE3) THEN
DEST ← TRUE ELSE DEST ← FALSE

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

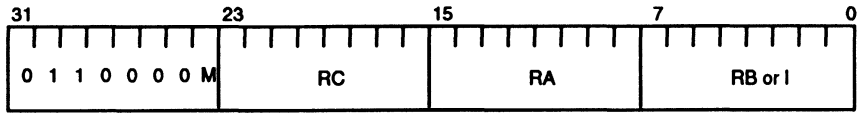
Operation: IF SRCA = SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

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

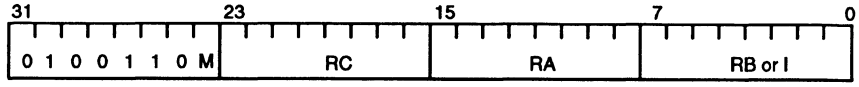
Operation: IF SRCA ≥ SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

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

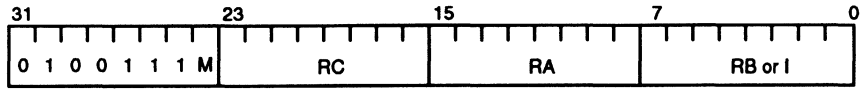
Operation: IF SRCA ≥ SRCB (unsigned) THEN DEST ← TRUE
ELSE DEST ← FALSE

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

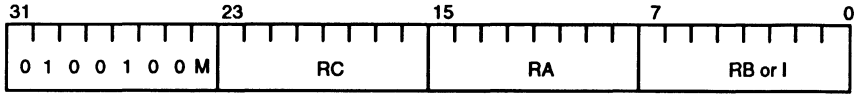
Operation: IF SRCA > SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

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

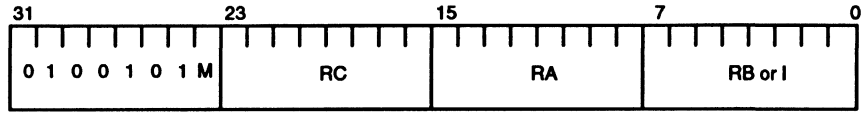
Operation: IF SRCA > SRCB (unsigned) THEN DEST ← TRUE
ELSE DEST ← FALSE

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

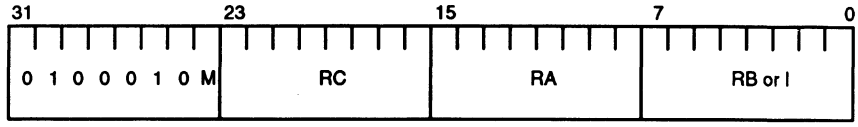
Operation: IF SRCA ≤ SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

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

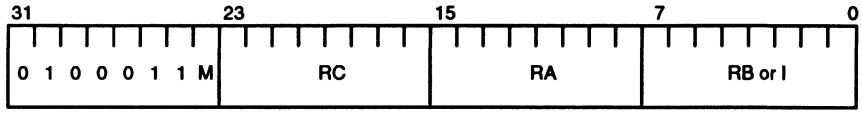
Operation: IF SRCA ≤ SRCB (unsigned) THEN DEST ← TRUE
ELSE DEST ← FALSE

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

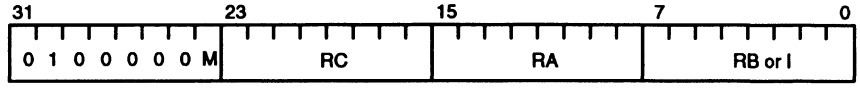
Operation: IF SRCA < SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

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

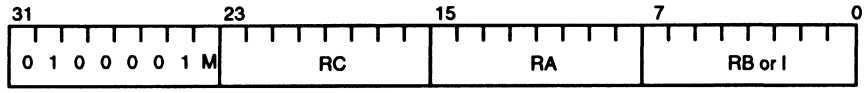
Operation: IF SRCA < SRCB (unsigned) THEN DEST ← TRUE
ELSE DEST ← FALSE

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

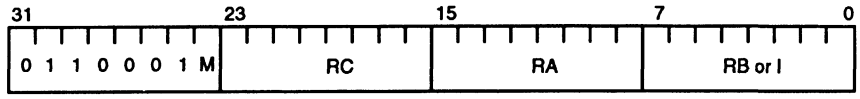
Operation: IF SRCA <> SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

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

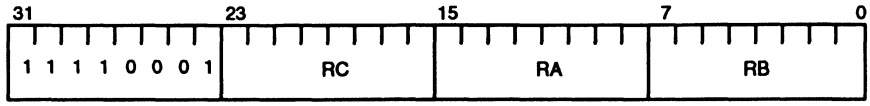
Operation: DEST (double-precision) ← SRCA (double-precision) + SRCB (double-precision)

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 the FRM field of the Floating-Point Environment Register and placed into the DEST location. The operands and the 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

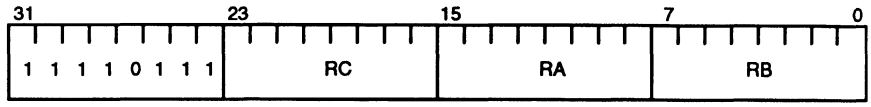
Operation: DEST (double-precision) ← SRCA (double-precision) / SRCB (double-precision)

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 the FRM field of the Floating-Point Environment Register and placed into the DEST location. The operands and the result of the division are double-precision floating-point numbers.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes a DDIV trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Floating-Point Equal To, Double-Precision

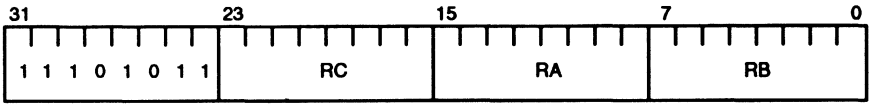
Operation: IF SRCA (double-precision) = SRCB (double-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax: DEQ rc, ra, rb

Status: fpl

Operands:

SRCA	Content of register RA and the twin of register RA
SRCB	Content of register RB and the twin of register RB
DEST	Register RC



OP = EB DEQ

Description: If the SRCA operand is equal to the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location. SRCA and SRCB are double-precision floating-point numbers.

The rounding mode specified by the FRM field of the Floating-Point Environment Register has no effect on this operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes a DEQ trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Floating-Point Greater Than Or Equal To, Double-Precision

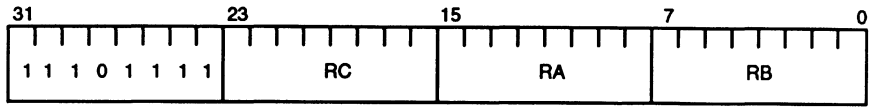
Operation: IF SRCA (double-precision) ≥ SRCB (double-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax: DGE rc, ra, rb

Status: fpl

Operands:

SRCA	Content of register RA and the twin of register RA
SRCB	Content of register RB and the twin of register RB
DEST	Register RC



OP = EF

DGE

Description: If the SRCA operand is greater than or equal to the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location. SRCA and SRCB are double-precision floating-point numbers.

The rounding mode specified by the FRM field of the Floating-Point Environment Register has no effect on this operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes a DGE trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Floating-Point Greater Than, Double-Precision

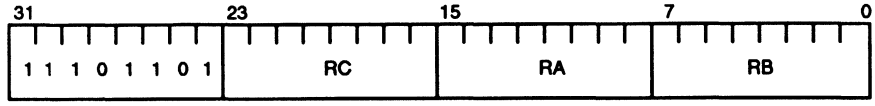
Operation: IF SRCA (double-precision) > SRCB (double-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax: DGT rc, ra, rb

Status: fpl

Operands:

SRCA	Content of register RA and the twin of register RA
SRCB	Content of register RB and the twin of register RB
DEST	Register RC



OP = ED

DGT

Description: If the SRCA operand is greater than the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location. SRCA and SRCB are double-precision floating-point numbers.

The rounding mode specified by the FRM field of the Floating-Point Environment Register has no effect on this operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes a DGT trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Divide Step

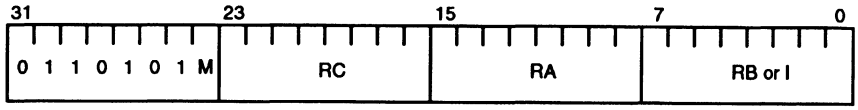
Operation: Perform one-bit step of a divide operation (unsigned)

Assembler Syntax:
DIV rc, ra, rb
or
DIV rc, ra, const 8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 6A, 6B

DIV

Description: If the Divide Flag (DF) bit of the ALU Status Register is 1, the SRCB operand is subtracted from the SRCA operand. If the DF bit is 0, the SRCB operand is added to the SRCA operand.

The carry-out of the add or subtract operation is exclusive-ORed with the value of the DF bit and the value of the Negative (N) bit of the ALU Status Register; the resulting value is complemented and placed into the DF bit. The sign of the result of the add or subtract is placed into the N bit.

The content of the Q Register is appended to the result of the add or subtract, and the resulting 64-bit value is shifted left by one bit position; the value computed for the DF bit above fills the vacated bit position. The high-order 32 bits of the 64-bit shifted value are placed into the DEST location. The low-order 32 bits of the shifted value are placed into the Q Register.

Examples of integer divide operations appear in Section 2.6.3.

Divide Initialize

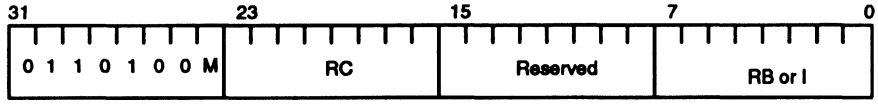
Operation: Initialize for a sequence of divide steps (unsigned)

Assembler Syntax:
DIV0 rc, rb
or
DIV0 rc, const8

Status: V, N, Z, C

Operands:

SRCB	M=0:	Content of register RB
	M=1:	I (Zero-extended to 32 bits)
DEST		Register RC



OP = 68, 69 DIV0

Description: The Divide Flag (DF) bit of the ALU Status Register is set. The sign of the SRCB operand is placed into the Negative bit of the ALU Status Register.

The content of the Q register is appended to the SRCB operand, and the resulting 64-bit value is shifted left by one bit position; a 0 fills the vacated bit position. The high-order 32 bits of the 64-bit shifted value are placed into the DEST location. The low-order 32 bits of the shifted value are placed into the Q Register.

Examples of integer divide operations appear in Section 2.6.3.

Floating-Point Multiply, Double-Precision

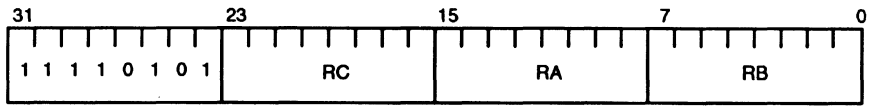
Operation: DEST (double-precision) ← SRCA (double-precision) * SRCB (double-precision)

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 the FRM field of the Floating-Point Environment Register and is placed into the DEST location. The operands and the result of the multiplication are double-precision floating-point numbers.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes a DMUL trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Floating-Point Subtract, Double-Precision

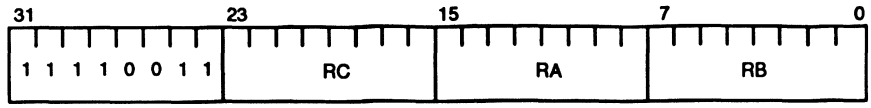
Operation: DEST (double-precision) ← SRCA (double-precision) − SRCB (double-precision)

Assembler Syntax: DSUB rc, ra, rb

Status: fpX, fpU, fpV, fpR, fpN

Operands:

SRCA	Content of register RA and the twin of register RA
SRCB	Content of register RB and the twin of register RB
DEST	Register RC



OP = F3 **DSUB**

Description: The SRCB operand is subtracted from the SRCA operand; the result is rounded according to the FRM field of the Floating-Point Environment Register and is placed into the DEST location. The operands and the result of the subtraction are double-precision floating-point numbers.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes a DSUB trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Trap to Software Emulation Routine

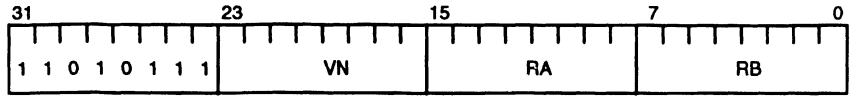
Operation: Load IPA and IPB registers with operand register numbers and Trap (VN)

Assembler Syntax: EMULATE vn, ra, rb

Status: Not affected

Operands: Absolute-register numbers for registers RA and RB

VN Trap vector number



OP = D7 **EMULATE**

Description: The IPA and IPB registers are set to the register numbers of registers RA and RB, respectively. A trap with the specified vector number occurs.

Note that the IPC register is also affected by this instruction, but 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. A Protection Violation trap also occurs if RA or RB specifies a register protected by the Register Bank Protect Register.

Extract Byte

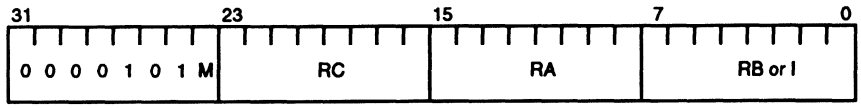
Operation: DEST ← SRCB, with low-order byte replaced by byte in SRCA selected by BP

Assembler Syntax:
EXBYTE rc, ra, rb
or
EXBYTE rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 0A, 0B **EXBYTE**

Description: A byte in the SRCA operand is selected by the Byte Pointer (BP) field of the ALU Status Register and the Byte Order (BO) bit of the Configuration Register. The selected byte replaces the low-order byte of the SRCB operand and the resulting word is placed into the DEST location.

Note: The selection of bytes within words is specified in Section 3.3.6.1.

Extract Half-Word

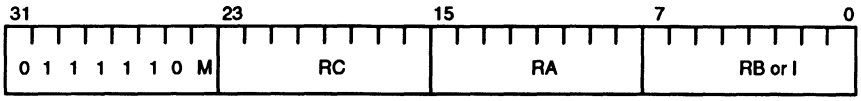
Operation: DEST ← SRCB, with low-order half-word replaced by half-word in SRCA selected by BP

Assembler Syntax:
EXHW rc, ra, rb
or
EXHW rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 7C, 7D **EXHW**

Description: A half-word in the SRCA operand is selected by the Byte Pointer (BP) field of the ALU Status Register and the Byte Order (BO) bit of the Configuration Register. The selected half-word replaces the low-order half-word of the SRCB operand and the resulting word is placed into the DEST location.

Note: The selection of half-words within words is specified in Section 3.3.6.1.

Extract Half-Word, Sign-Extended

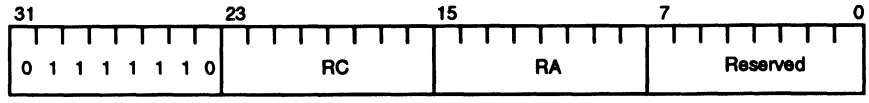
Operation: DEST ← half-word in SRCA selected by BP, sign-extended to 32 bits

Assembler Syntax: EXHWS rc, ra

Status: Not affected

Operands:

SRCA	Content of register RA
DEST	Register RC



OP = 7E

EXHWS

Description: A half-word in the SRCA operand is selected by the Byte Pointer (BP) field of the ALU Status Register and the Byte Order (BO) bit of the Configuration Register. The selected half-word is sign-extended to 32 bits and the resulting word is placed into the DEST location.

Note: The selection of half-words within words is specified in Section 3.3.6.1.

Extract Word, Bit-Aligned

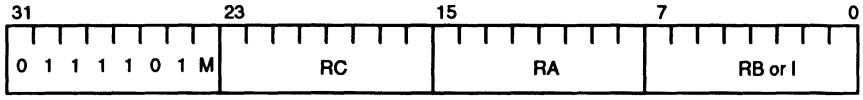
Operation: DEST ← high-order word of (SRCA // SRCB << FC)

Assembler Syntax:
EXTRACT rc, ra ,rb
or
EXTRACT rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 7A, 7B **EXTRACT**

Description: The SRCB operand is appended to the SRCA operand and the resulting 64-bit value is shifted left by the number of bit-positions specified by the Funnel Shift Count (FC) field of the ALU Status register. The high-order 32 bits of the 64-bit shifted value are placed in the DEST location.

If the SRCB operand is the same as the SRCA operand, the EXTRACT instruction performs a rotate operation.

Floating-Point Add, Single-Precision

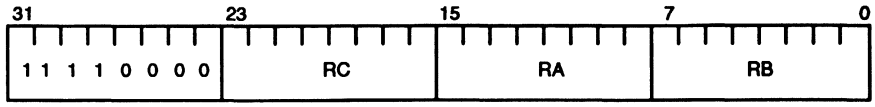
Operation: DEST (single-precision) ← SRCA (single-precision) + SRCB (single-precision)

Assembler Syntax: FADD rc, ra, rb

Status: fpX, fpU, fpV, fpR, fpN

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = F0 **FADD**

Description: The SRCA operand is added to the SRCB operand; the result is rounded according to the FRM field of the Floating-Point Environment Register and placed into the DEST location. The operands and the result of the addition are single-precision floating-point numbers.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes an FADD trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Floating-Point Divide, Single-Precision

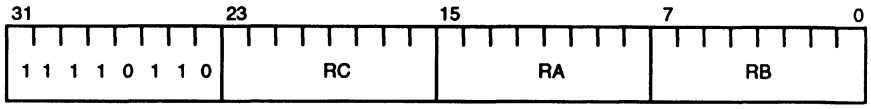
Operation: DEST (single-precision) ← SRCA (single-precision) / SRCB (single-precision)

Assembler Syntax: FDIV rc, ra, rb

Status: fpD, fpX, fpU, fpV, fpR, fpN

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = F6 FDIV

Description: The SRCA operand is divided by the SRCB operand; the result is rounded according to the FRM field of the Floating-Point Environment Register and placed into the DEST location. The operands and the result of the division are single-precision floating-point numbers.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes an FDIV trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Floating-Point Multiply, Single-to-Double Precision

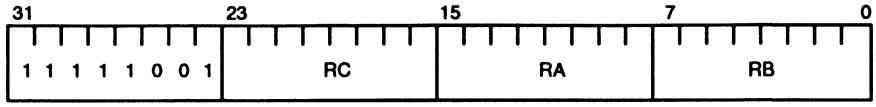
Operation: DEST (double-precision) ← SRCA (single-precision) * SRCB (single-precision)

Assembler Syntax: FDMUL rc, ra, rb

Status: fpR, fpN

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = F9 **FDMUL**

Description: The SRCB operand is multiplied by the SRCA operand; the result is placed into the DEST location. SRCA and SRCB are single-precision floating-point numbers; the result is produced in double-precision format. Because the product of two single-precision operands can always be represented exactly as a double-precision number, the FDMUL result does not depend on the FRM field of the Floating-Point Environment Register.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes an FDMUL trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Floating-Point Equal To, Single-Precision

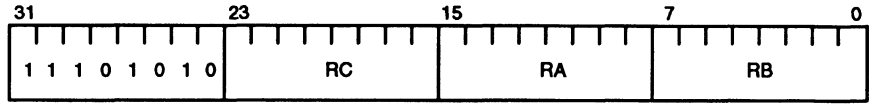
Operation: IF SRCA (single-precision) = SRCB (single-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax: FEQ rc, ra, rb

Status: fpN

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = EA FEQ

Description: If the SRCA operand is equal to the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location. SRCA and SRCB are single-precision floating-point numbers.

The rounding mode specified by the FRM field of the Floating-Point Environment Register has no effect on this operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes an FEQ trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Floating-Point Greater Than Or Equal To, Single-Precision

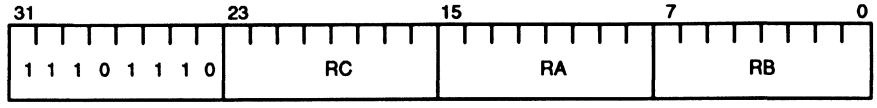
Operation: IF SRCA (single-precision) ≥ SRCB (single-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax: FGE rc, ra, rb

Status: fpN

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = EE

FGE

Description: If the SRCA operand is greater than or equal to the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location. SRCA and SRCB are single-precision floating-point numbers.

The rounding mode specified by the FRM field of the Floating-Point Environment Register has no effect on this operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes an FGE trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Floating-Point Greater Than, Single-Precision

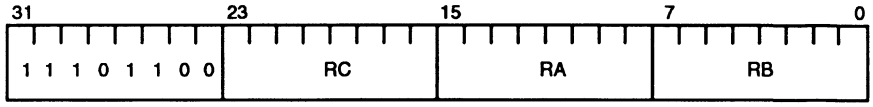
Operation: IF SRCA (single-precision) > SRCB (single-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

Assembler Syntax: FGT rc, ra, rb

Status: fpN

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = EC FGT

Description: If the SRCA operand is greater than the SRCB operand, a Boolean TRUE is placed into the DEST location; otherwise, a Boolean FALSE is placed into the DEST location. SRCA and SRCB are single-precision floating-point numbers.

The rounding mode specified by the FRM field of the Floating-Point Environment Register has no effect on this operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes an FGT trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Floating-Point Multiply, Single-Precision

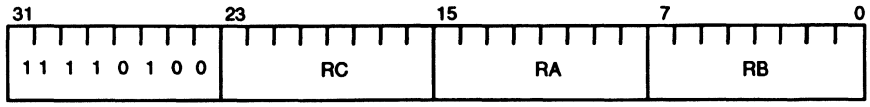
Operation: DEST (single-precision) ← SRCA (single-precision) * SRCB (single-precision)

Assembler Syntax: FMUL rc, ra, rb

Status: fpX, fpU, fpV, fpR, fpN

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = F4 **FMUL**

Description: The SRCA operand is multiplied by the SRCB operand; the result is rounded according to the FRM field of the Floating-Point Environment Register and placed into the DEST location. The operands and the result of the multiplication are single-precision floating-point numbers.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes an FMUL trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Floating-Point Subtract, Single-Precision

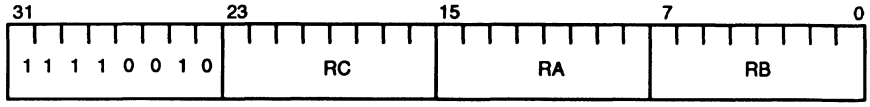
Operation: DEST (single-precision) ← SRCA (single-precision) – SRCB (single-precision)

Assembler Syntax: FSUB rc, ra, rb

Status: fpX, fpU, fpV, fpR, fpN

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = F2 **FSUB**

Description: The SRCB operand is subtracted from the SRCA operand; the result is rounded according to the FRM field of the Floating-Point Environment Register and placed into the DEST location. The operands and the 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.

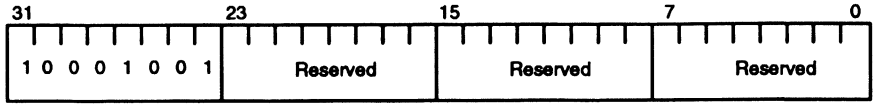
Enter Halt Mode

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 in the next cycle, or in the cycle after an external data access is completed if an access is in progress.

This instruction may be executed only by Supervisor-mode programs. An attempted execution by a User-mode program causes a Protection Violation trap to occur unless the Protection Violation trap was disabled during reset (see Section 17.6.5).

If the instruction following a Halt instruction has an exception (e.g., TLB Miss), the trap associated with this exception is taken before the processor enters the Halt mode.

Insert Byte

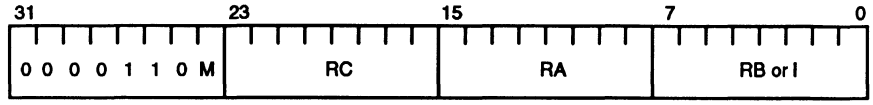
Operation: DEST ← SRCA, with byte selected by BP replaced by low-order byte of SRCB

Assembler Syntax:
INBYTE rc, ra, rb
or
INBYTE rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP =0C, 0D **INBYTE**

Description: A byte in the SRCA operand is selected by the Byte Pointer (BP) field of the ALU Status Register and the Byte Order (BO) bit of the Configuration Register. The selected byte is replaced by the low-order byte of the SRCB operand and the resulting word is placed into the DEST location.

Note: The selection of bytes within words is specified in Section 3.3.6.1.

Insert Half-Word

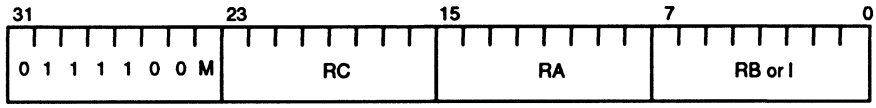
Operation: DEST ← SRCA, with half-word selected by BP replaced by low-order half-word of SRCB

Assembler Syntax:
INHW rc, ra, rb
or
INHW rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 78, 79 **INHW**

Description: A half-word in the SRCA operand is selected by the Byte Pointer (BP) field of the ALU Status Register and the Byte Order (BO) bit of the Configuration Register. The selected half-word is replaced by the low-order half-word of the SRCB operand and the resulting word is placed into the DEST location.

Note: The selection of half-words within words is specified in Section 3.3.6.1.

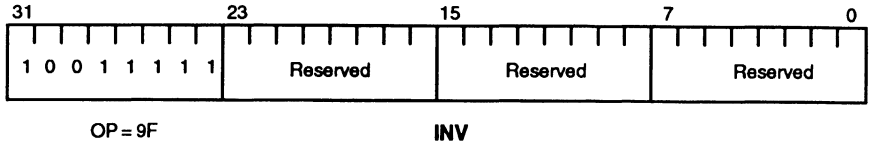
Invalidate

Operation: None

Assembler Syntax: INV

Status: Not affected

Operands: Not applicable



Description: In 29K Family processors with instruction caches, this instruction causes all cache valid bits to be reset. In the Am29200 microprocessor, this instruction performs no operation, except it is a privileged instruction. Attempted execution by a User-mode program causes a Protection Violation trap to occur.

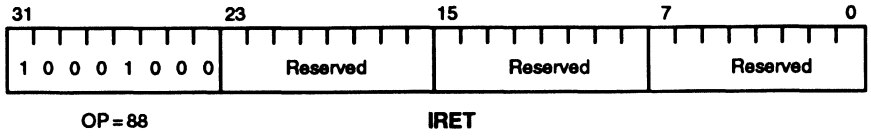
Interrupt Return

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 16.3.4.

This instruction may be executed only by Supervisor-mode programs. An attempted execution by a User-mode program causes a Protection Violation trap to occur.

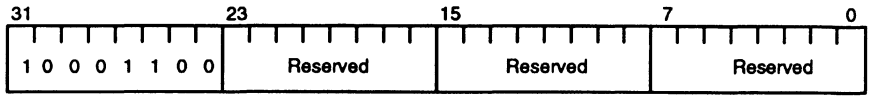
Interrupt Return and Invalidate

Operation: Perform an interrupt return sequence

Assembler Syntax: IRETINV

Status: Not affected

Operands: Not applicable



OP = 8C **IRETINV**

Description: This instruction performs the interrupt return sequence described in Section 16.3.4. In 29K Family processors with an instruction cache, this instruction also resets the valid bits in the cache. In the Am29200 microprocessor, this instruction is identical to the IRET instruction.

This instruction may be executed only by Supervisor-mode programs. An attempted execution by a User-mode program causes a Protection Violation trap to occur.

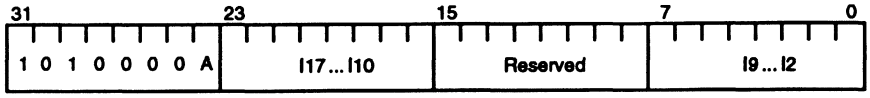
Jump

Operation: PC ← TARGET
Execute delay instruction

Assembler Syntax: JMP target

Status: Not affected

Operands: TARGET A = 0: I17 ... I10 // I9 ... I2 (sign-extended to 30 bits) + PC
A = 1: I17 ... I10//I9 ... I2 (zero-extended to 30 bits)



OP = A0, A1 **JMP**

Description: A non-sequential instruction fetch occurs to the instruction address given by the TARGET operand. The instruction following the JMP is executed before the non-sequential fetch occurs.

Jump False

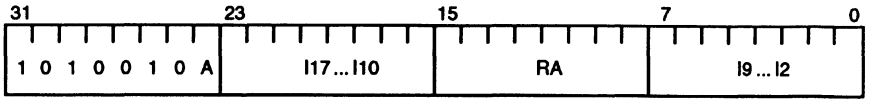
Operation: IF SRCA = FALSE THEN PC ← TARGET
Execute delay instruction

Assembler Syntax: JMPF ra, target

Status: Not affected

Operands: SRCA Content of register RA

TARGET A = 0: I17 ... I10 // I9 ... I2 (sign-extended to 30 bits) + PC
A = 1: I17 ... I10 // I9 ... I2 (zero-extended to 30 bits)



OP = A4, A5 **JMPF**

Description: If SRCA is a Boolean FALSE, a non-sequential instruction fetch occurs to the instruction address given by the TARGET operand.

If SRCA is a Boolean TRUE, this instruction has no effect.

The instruction following the JMPF is executed regardless of the value of SRCA.

Jump False and Decrement

Operation:

```
IF SRCA = FALSE THEN
    SRCA ← SRCA – 1
    PC ← TARGET
ELSE
    SRCA ← SRCA – 1
Execute delay instruction
```

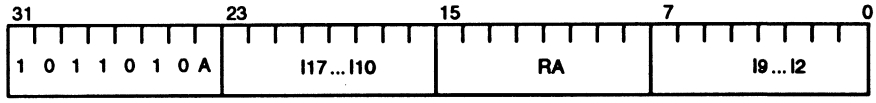
Assembler Syntax: JMPFDEC ra, target

Status: Not affected

Operands:

SRCA	Content of register RA

TARGET	A = 0: I17 ... I10 // I9 ... I2 (sign-extended to 30 bits) + PC
	A = 1: I17 ... I10 // I9 ... I2 (zero-extended to 30 bits)



OP = B4, B5

JMPFDEC

Description: If SRCA is a Boolean FALSE, a non-sequential instruction fetch occurs to the instruction address given by the TARGET operand.

If SRCA is a Boolean TRUE, this instruction has no effect on the instruction-execution sequence.

The SRCA operand is decremented by one, regardless of whether or not the non-sequential instruction fetch occurs. Note that a negative number for the SRCA operand is a Boolean TRUE.

The instruction following the JMPFDEC is executed regardless of the value of SRCA.

Jump False Indirect

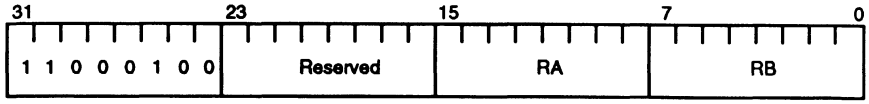
Operation: IF SRCA = FALSE THEN PC ← SRCB
Execute delay instruction

Assembler Syntax: JMPFI ra, rb

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	Content of register RB



OP = C4 **JMPFI**

Description: If the SRCA is a Boolean FALSE, a non-sequential instruction fetch occurs to the instruction address given by the SRCB operand.

If SRCA is a Boolean TRUE, this instruction has no effect.

The instruction following the JMPFI is executed regardless of the value of SRCA.

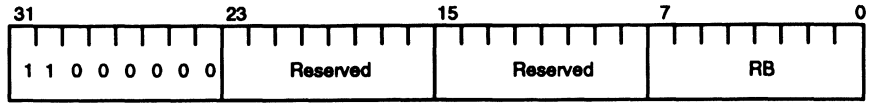
Jump Indirect

Operation: PC ← SRCB
Execute delay instruction

Assembler Syntax: JMPI rb

Status: Not affected

Operands: SRCB Content of register RB



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

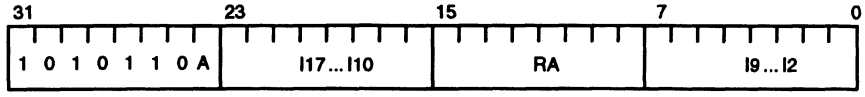
Assembler Syntax: JMPT ra, target

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

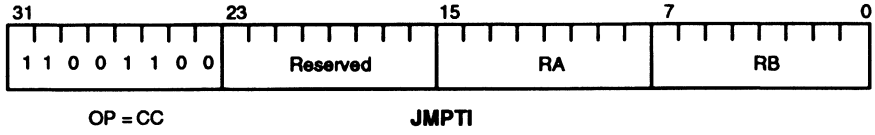
Operation: IF SRCA = TRUE THEN PC ← SRCB
Execute delay instruction

Assembler Syntax: JMPTI ra, rb

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	Content of register RB



Description: If the SRCA is a Boolean TRUE, a non-sequential instruction fetch occurs to the instruction address given by the SRCB operand.

If SRCA is a Boolean FALSE, this instruction has no effect.

The instruction following the JMPTI is executed regardless of the value of SRCA.

Load

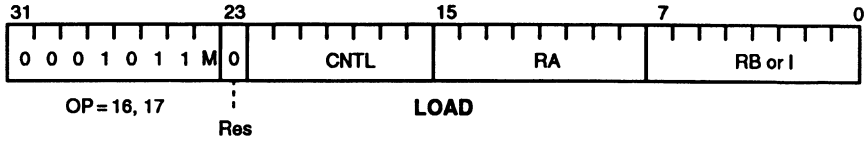
Operation: DEST ← EXTERNAL WORD [SRCB]

Assembler Syntax:
LOAD 0, cntl, ra, rb
or
LOAD 0, cntl, ra, const8

Status: Not affected

Operands:

SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RA



Description: The external word addressed by the SRCB operand is placed into the DEST location.

The CNTL field of the LOAD instruction affects the bus access as described in Section 3.3.1.

Load and Lock

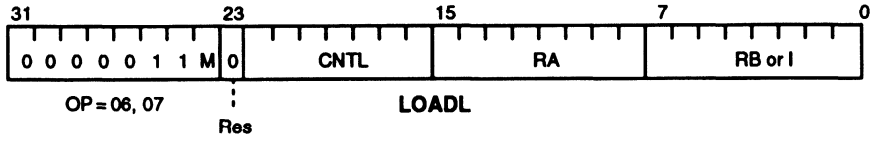
Operation: DEST ← EXTERNAL WORD [SRCB]

Assembler Syntax:
LOADL 0, cntl, ra, rb
or
LOADL 0, cntl, ra, const8

Status: Not affected

Operands:

SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RA



Description: The external word addressed by the SRCB operand is placed into the DEST location.

The CNTL field of the LOADL instruction affects the bus access as described in Section 3.3.1.

In other 29K Family processors, this instruction is provided for the implementation of interlock protocols. In the Am29200 microprocessor, the LOADL instruction is identical to the LOAD instruction.

Load Multiple

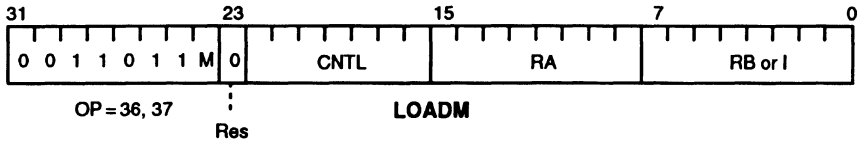
Operation: DEST... DEST+COUNT ← EXTERNAL WORD [SRCB] ...
EXTERNAL WORD [SRCB + (COUNT * 4)]

Assembler Syntax:
LOADM 0, cntl, ra, rb
or
LOADM 0, cntl, ra, const8

Status: Not affected

Operands:

SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	register RA



Description: External words at consecutive word addresses beginning with the word addressed by the SRCB operand, are placed into consecutive registers beginning with the DEST location.

The total number of words accessed in the sequence is specified by the Count Remaining (CR) field of the Channel Control Register (which also appears in the Load/Store Count Remaining Register) at the beginning of the bus access. The total number of words is the value of the CR field plus one. The CNTL field of the LOADM instruction affects the access as described in Section 3.3.1.

Note: The address and register-number sequences for the LOADM instruction are specified in Section 3.3.4. Because this instruction uses the Channel Address and Control Registers, it should not be executed when the FZ bit is 1.

Load and Set

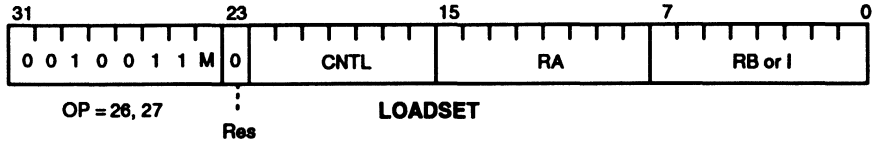
Operation: DEST ← EXTERNAL WORD [SRCB]
EXTERNAL WORD [SRCB] ← h'FFFFFFFF'

Assembler Syntax:
LOADSET 0, cntl, ra, rb
or
LOADSET 0, cntl, ra, const8

Status: Not affected

Operands:

SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RA



Description: The external word addressed by the SRCB operand is placed into the DEST location. After the DEST location is altered, the external word addressed by the SRCB operand is written, atomically, with a word consisting of a 1 in every bit position.

The CNTL field of the LOADSET instruction affects the bus access as described in Section 3.3.1.

Move from Special Register

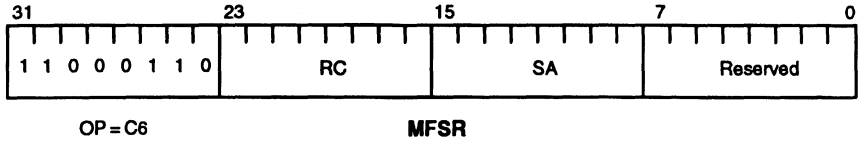
Operation: DEST ← SPECIAL

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

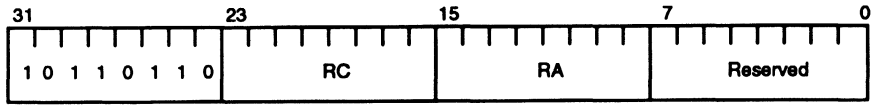
Operation: None

Assembler Syntax: MFTLB rc, ra

Status: Not affected

Operands:

SRCA	Content of register RA, bits 6 ... 0
DEST	Register RC



OP = B6 **MFTLB**

Description: In 29K Family processors with an MMU, this instruction reads TLB entries. In the Am29200 microprocessor, this instruction performs no operation except it is a privileged instruction. Attempted execution by a User-mode program causes a Protection Violation trap to occur.

Move to Special Register

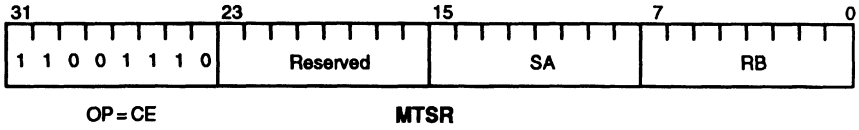
Operation: SPDEST ← SRCB

Assembler Syntax: MTSR spid, rb

Status: Not affected unless the destination is the ALU Status Register

Operands:

SRCB	Content of register RB
SPDEST	Special-purpose register SA



Description: The SRCB operand is placed into the SPECIAL location.

For programs in the User mode, a Protection Violation trap occurs if SA specifies a protected special-purpose register. If a trap occurs, the SPDEST location is not altered.

Move to Special Register Immediate

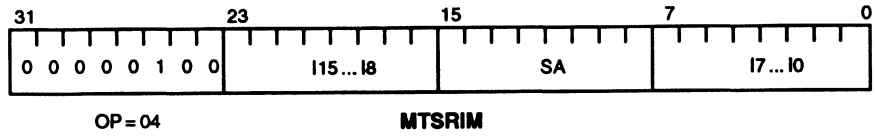
Operation: SPDEST ← 0I16

Assembler Syntax: MTSRIM spid, const16

Status: Not affected unless the destination is the ALU Status Register

Operands:

0I16	I15 ... I8 // I7 ... I0 (zero-extended to 32 bits)
SPDEST	Special-purpose register SA



Description: The 0I16 operand is placed into the SPECIAL location.

For programs in the User mode, a Protection Violation trap occurs if SA specifies a protected special-purpose register. If a trap occurs, the SPDEST location is not altered.

Move to Translation Look-Aside Buffer Register

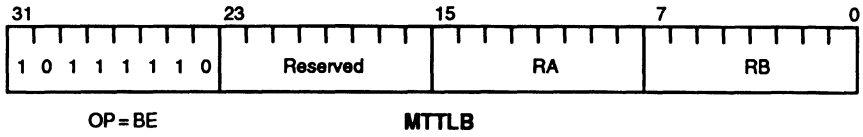
Operation: None

Assembler Syntax: MTTLB ra, rb

Status: Not affected

Operands:

SRCA	Content of register RA, bits 6...0
SRCB	Content of register RB



Description: In 29K Family processors with an MMU, this instruction modifies TLB entries. In the Am29200 microprocessor, this instruction performs no operation except it is a privileged instruction. Attempted execution by a User-mode program causes a Protection Violation trap to occur.

Multiply Step

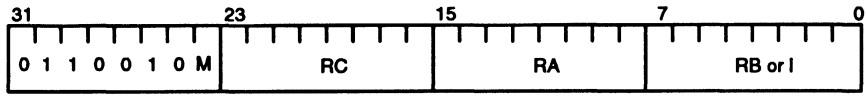
Operation: Perform one-bit step of a multiply operation

Assembler Syntax:
MUL rc, ra, rb
or
MUL rc, ra, const 8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 64, 65 **MUL**

Description: If the least significant bit of the Q Register is 1, the SRCA operand is added to the SRCB operand. If the least significant bit of the Q register is 0, a zero word is added to the SRCB operand.

The content of the Q Register is appended to the result of the add and the resulting 64-bit value is shifted right by one bit position; the true sign of the result of the add fills the vacated bit position (i.e., the sign of the result is complemented if an overflow occurred during the add operation). The high-order 32 bits of the 64-bit shifted value are placed into the DEST location. The low-order 32 bits of the shifted value are placed into the Q Register.

Examples of integer multiply operations appear in Section 2.6.2.

Multiply Last Step

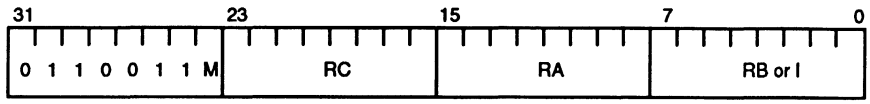
Operation: Complete a sequence of multiply steps (for signed multiply)

Assembler Syntax:
MULL rc, ra, rb
or
MULL rc, ra, const 8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 66, 67 **MULL**

Description: If the least significant bit of the Q Register is 1, the SRCA operand is subtracted from the SRCB operand. If the least significant bit of the Q register is 0, a zero word is subtracted from the SRCB operand.

The content of the Q Register is appended to the result of the subtract and the resulting 64-bit value is shifted right by one bit position; the true sign of the result of the subtract fills the vacated bit position (i.e., the sign of the result is complemented if an overflow occurred during the subtract operation). The high-order 32 bits of the 64-bit shifted value are placed into the DEST location. The low-order 32 bits of the shifted value are placed into the Q Register.

Examples of integer multiply operations appear in Section 2.6.2.

Integer Multiply, Unsigned

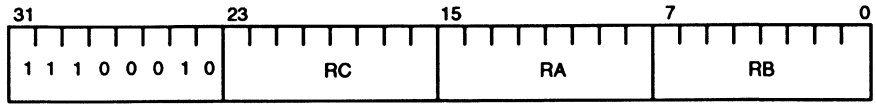
Operation: DEST ← SRCA * SRCB

Assembler Syntax: MULTIPLU rc, ra, rb

Status: None

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = E2 **MULTIPLU**

Description: The SRCA operand is multiplied by the SRCB operand. The low-order 32 bits of the 64-bit result are placed into the DEST location. This operation treats the SRCA and SRCB operands as unsigned integers and produces an unsigned result.

The contents of the Q register are undefined after a MULTIPLU operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes an MULTIPLU trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Integer Multiply, Signed

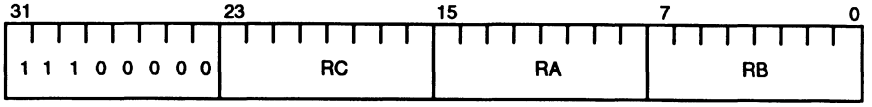
Operation: DEST ← SRCA * SRCB

Assembler Syntax: MULTIPLY rc, ra, rb

Status: None

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = E0

MULTIPLY

Description: The SRCA operand is multiplied by the SRCB operand. The low-order 32 bits of the 64-bit result are placed into the DEST location. This operation treats the SRCA and SRCB operands as two's-complement integers and produces a two's-complement result.

The contents of the Q register are undefined after a MULTIPLY operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes 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

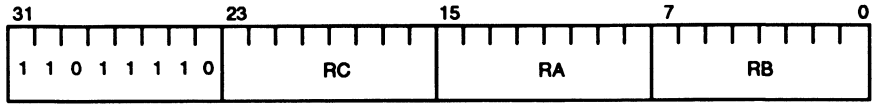
Operation: DEST ← SRCA * SRCB

Assembler Syntax: MULTM rc, ra, rb

Status: None

Operands:

	SRCA	Content of register RA
	SRCB	Content of register RB
	DEST	Register RC



OP = DE MULTM

Description: The SRCA operand is multiplied by the SRCB operand. The high-order 32 bits of the 64-bit result are placed into the DEST location. This operation treats the SRCA and SRCB operands as two's-complement integers and produces a two's-complement result.

The contents of the Q register are undefined after a MULTM operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes 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

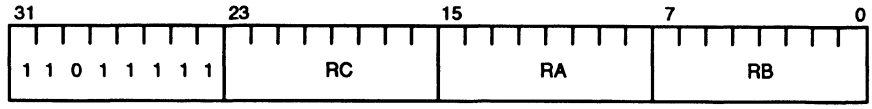
Operation: DEST ← SRCA * SRCB

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.

The contents of the Q register are undefined after a MULTMU operation.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes an MULTMU trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Multiply Step, Unsigned

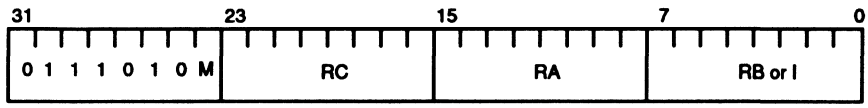
Operation: Perform one-bit step of a multiply operation (unsigned)

Assembler Syntax:
MULU rc, ra, rb
or
MULU rc, ra, const 8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 74, 75 **MULU**

Description: If the least significant bit of the Q Register is 1, the SRCA operand is added to the SRCB operand. If the least significant bit of the Q register is 0, a zero word is added to the SRCB operand.

The content of the Q register is appended to the result of the add and the resulting 64-bit value is shifted right by one bit position; the carry-out of the add fills the vacated bit position. The high-order 32 bits of the 64-bit shifted value are placed into the DEST location. The low-order 32 bits of the shifted value are placed into the Q Register.

Examples of integer multiply operations appear in Section 2.6.2.

NAND Logical

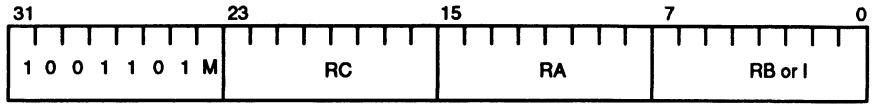
Operation: DEST ← ~(SRCA & SRCB)

Assembler Syntax: NAND rc, ra, rb
or
NAND rc, ra, const8

Status: N, Z

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 9A, 9B

NAND

Description: The SRCA operand is logically ANDed, bit-by-bit, with the SRCB operand. The one's-complement of the result is placed into the DEST location.

NOR Logical

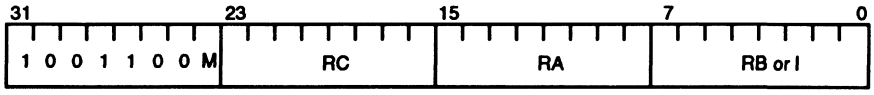
Operation: DEST ← ~(SRCA | SRCB)

Assembler Syntax:
NOR rc, ra, rb
or
NOR rc, ra, const8

Status: N, Z

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 98, 99 **NOR**

Description: The SRCA operand is logically ORed, bit-by-bit, with the SRCB operand. The one's-complement of the result is placed into the DEST location.

OR Logical

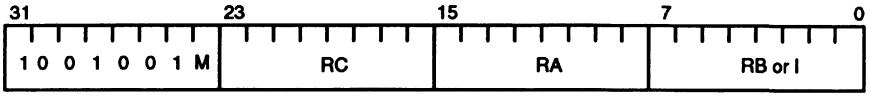
Operation: DEST ← SRCA | SRCB

Assembler Syntax:
OR rc, ra, rb
or
OR rc, ra, const8

Status: N, Z

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 92, 93 **OR**

Description: The SRCA operand is logically ORed, bit-by-bit, with the SRCB operand, and the result is placed into the DEST location.

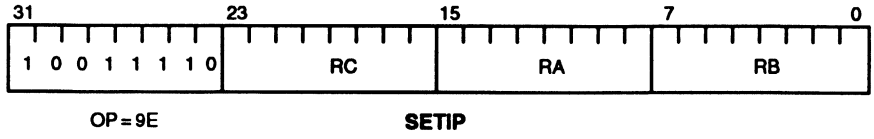
Set Indirect Pointers

Operation: Load IPA, IPB, and IPC registers with operand-register numbers

Assembler Syntax: SETIP rc, ra, rb

Status: Not affected

Operands: Absolute-register numbers for registers RA, RB, and RC



Description: The IPA, IPB, and IPC registers are set to the register numbers of registers RA, RB, and RC, respectively.

For programs in the User mode, a Protection Violation trap occurs if RA, RB, or RC specifies a register protected by the Register Bank Protect Register.

Note: This instruction has a delayed effect on the indirect pointer registers as discussed in Section 5.6.

Shift Left Logical

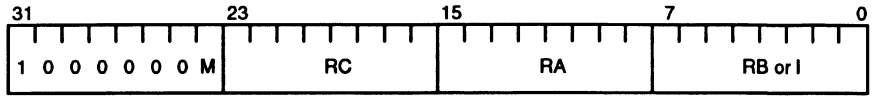
Operation: DEST ← SRCA << SRCB (zero fill)

Assembler Syntax:
SLL rc, ra, rb
or
SLL rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB, bits 4 ... 0 M = 1: I, bits 4...0
DEST	Register RC



OP = 80, 81 **SLL**

Description: The SRCA operand is shifted left by the number of bit positions specified by the SRCB operand; zeros fill vacated bit positions. The result is placed into the DEST location.

Floating-Point Square Root

Operation: DEST ← SQRT(SRCA)

Assembler Syntax: SQRT rc, ra, FS

Status: fpX, fpR, fpN

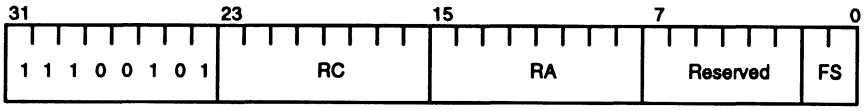
Operands:

SRCA Content of register RA (single-precision floating-point)
or
Content of register RA and the twin of register RA (double-precision floating-point)

DEST Register RC (single-precision floating-point)
or
Register RC and twin of Register RC (double-precision floating-point)

Control:

FS	Format of source operand SRCA
00	Reserved for future use
01	Single-precision floating-point
10	Double-precision floating-point
11	Reserved for future use



OP = E5

SQRT

Description: This operation computes the square root of floating-point operand SRCA; the result is rounded according to FRM field of the Floating-Point Environment Register and placed into the DEST location. The operand and result are single- or double-precision floating-point numbers as specified by FS.

Note: This instruction is not supported directly in processor hardware. In the current implementation, this instruction causes an SQRT trap. When the trap occurs, the IPA and IPC registers are set to reference SRCA and DEST and the IPB Register is set with the value of the FS field.

Shift Right Arithmetic

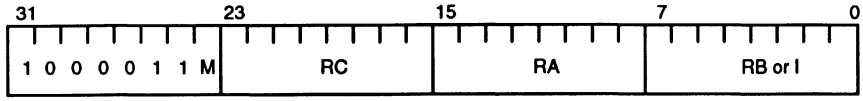
Operation: DEST ← SRCA >> SRCB (sign fill)

Assembler Syntax:
SRA rc, ra, rb
or
SRA rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB, bits 4 ... 0 M = 1: I, bits 4 ... 0
DEST	Register RC



OP = 86, 87 **SRA**

Description: The SRCA operand is shifted right by the number of bit positions specified by the SRCB operand; the sign of the SRCA operand fills vacated bit positions. The result is placed into the DEST location.

Shift Right Logical

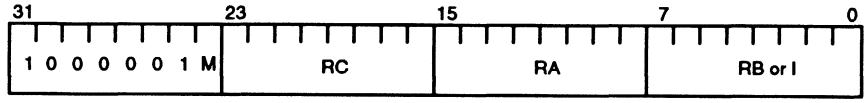
Operation: DEST ← SRCA >> SRCB (zero fill)

Assembler Syntax:
SRL rc, ra, rb
or
SRL rc, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB, bits 4 ... 0 M = 1: I, bits 4 ... 0
DEST	Register RC



OP = 82, 83

SRL

Description: The SRCA operand is shifted right by the number of bit positions specified by the SRCB operand; zeros fill vacated bit positions. The result is placed into the DEST location.

Store

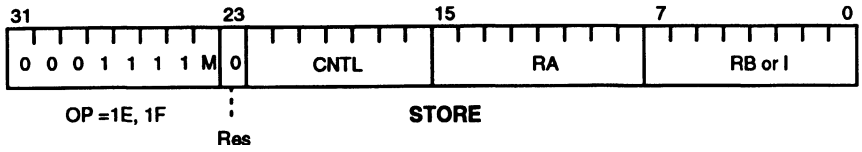
Operation: EXTERNAL WORD [SRCB] ← SRCA

Assembler Syntax:
STORE 0, cntl, ra, rb
or
STORE 0, cntl, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)



Description: The SRCA operand is placed into the external word addressed by the SRCB operand.

The CNTL field of the STORE instruction affects the bus access as described in Section 3.3.1.

Store and Lock

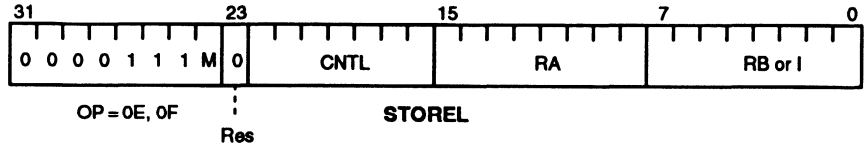
Operation: EXTERNAL WORD [SRCB] ← SRCA

Assembler Syntax:
STOREL 0, cntl, ra, rb
or
STOREL 0, cntl, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)



Description: The SRCA operand is placed into the external word addressed by the SRCB operand.

The CNTL field of the STOREL instruction affects the bus access as described in Section 3.3.1.

In other 29K Family processors, this instruction is provided for the implementation of interlock protocols. In the Am29200 microprocessor, the STOREL instruction is identical to the STORE instruction.

Store Multiple

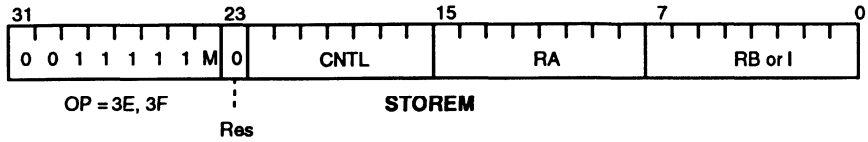
Operation: EXTERNAL WORD [SRCB] ... EXTERNAL WORD [SRCB + (COUNT * 4)]
← SRCA ... SRCA+COUNT

Assembler Syntax:
STOREM 0, cntl, ra, rb
or
STOREM 0, cntl, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)



Description: The contents of consecutive registers, beginning with the SRCA operand, are placed into external words at consecutive word addresses, beginning with the word addressed by the SRCB operand.

The total number of words accessed in the sequence is specified by the Count Remaining (CR) field of the Channel Control Register (which also appears in the Load/Store Count Remaining Register) at the beginning of the bus access. The total number of words is the value of the CR field plus one. The CNTL field of the STOREM instruction affects the access as described in Section 3.3.1.

Note: The address and register-number sequences for the STOREM instruction are specified in Section 3.3.4. Because this instruction uses the Channel Address, Data, and Control Registers, it should not be executed when the FZ bit is 1.

Subtract

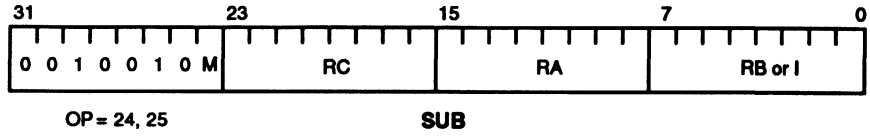
Operation: DEST ← SRCA – SRCB

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

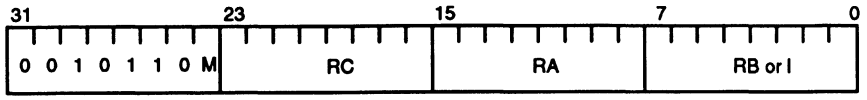
Operation: DEST ← SRCA − SRCB − 1 + C

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

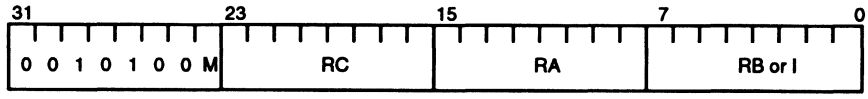
Operation: DEST ← SRCA − SRCB − 1 + C
IF signed overflow THEN Trap (Out of Range)

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

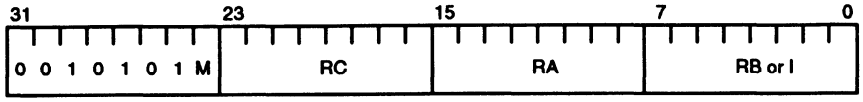
Operation: DEST ← SRCA − SRCB − 1 + C
IF unsigned underflow THEN Trap (Out of Range)

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

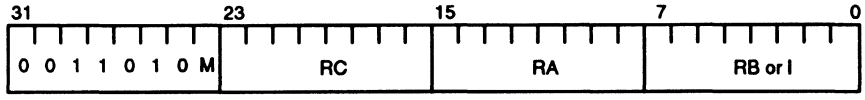
Operation: DEST ← SRCB – SRCA

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

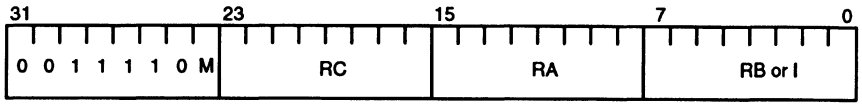
Operation: DEST ← SRCB − SRCA − 1 + C

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

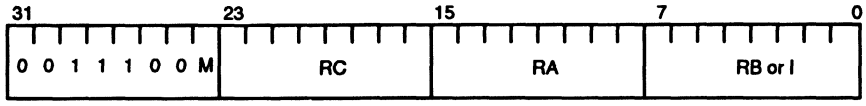
Operation: DEST ← SRCB − SRCA − 1 + C
IF signed overflow THEN Trap (Out of Range)

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



OP = 38, 39 **SUBRCS**

Description: The SRCB operand is added to the one's-complement of the SRCA operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location. If the add operation causes a two's-complement signed overflow, an Out-of-Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

Subtract Reverse with Carry, Unsigned

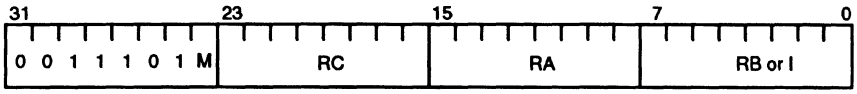
Operation: DEST ← SRCB − SRCA − 1 + C
IF unsigned underflow THEN Trap (Out of Range)

Assembler Syntax:
SUBRCU rc, ra, rb
or
SUBRCU rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 3A, 3B **SUBRCU**

Description: The SRCB operand is added to the one's-complement of the SRCA operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location. If the add operation causes an unsigned underflow, an Out-of-Range trap occurs.

Note that the DEST location is altered whether or not an underflow occurs.

Subtract Reverse, Signed

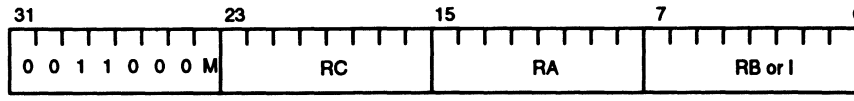
Operation: DEST ← SRCB – SRCA
IF signed overflow THEN Trap (Out of Range)

Assembler Syntax:
SUBRS rc, ra, rb
or
SUBRS rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 30, 31 **SUBRS**

Description: The SRCB operand is added to the two's-complement of the SRCA operand and the result is placed into the DEST location. If the add operation causes a two's-complement signed overflow, an Out-of-Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

Subtract Reverse, Unsigned

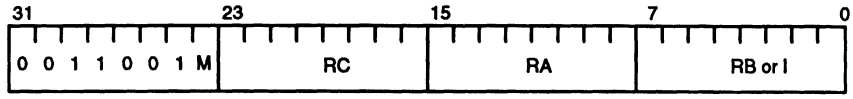
Operation: DEST ← SRCB − SRCA
IF unsigned underflow THEN Trap (Out of Range)

Assembler Syntax:
SUBRU rc, ra, rb
or
SUBRU rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 32, 33 **SUBRU**

Description: The SRCB operand is added to the two's-complement of the SRCA operand and the result is placed into the DEST location. If the add operation causes an unsigned underflow, an Out-of-Range trap occurs.

Note that the DEST location is altered whether or not an underflow occurs.

Subtract, Signed

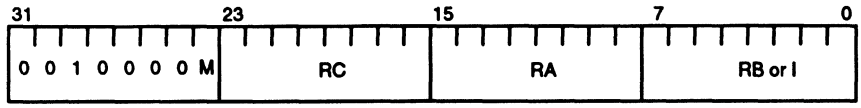
Operation: DEST ← SRCA − SRCB
IF signed overflow THEN Trap (Out of Range)

Assembler Syntax:
SUBS rc, ra, rb
or
SUBS rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 20, 21 **SUBS**

Description: The SRCA operand is added to the two's-complement of the SRCB operand and the result is placed into the DEST location. If the add operation causes a two's-complement signed overflow, an Out-of-Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

Subtract, Unsigned

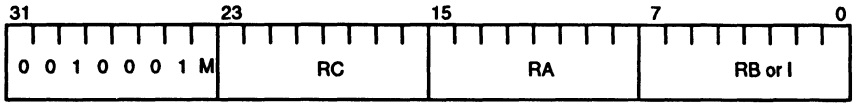
Operation: DEST ← SRCA − SRCB
IF unsigned underflow THEN Trap (Out of Range)

Assembler Syntax:
SUBU rc, ra, rb
or
SUBU rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 22, 23 **SUBU**

Description: The SRCA operand is added to the two's-complement of the SRCB operand and the result is placed into the DEST location. If the add operation causes an unsigned underflow, an Out-of-Range trap occurs.

Note that the DEST location is altered whether or not an underflow occurs.

Exclusive-NOR Logical

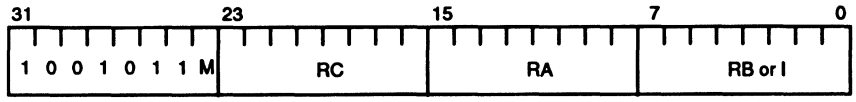
Operation: DEST ← ~(SRCA^SRCB)

Assembler Syntax:
XNOR rc, ra, rb
or
XNOR rc, ra, const8

Status: N, Z

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 96, 97 **XNOR**

Description: The SRCA operand is logically exclusive-ORed, bit-by-bit, with the SRCB operand. The one's-complement of the result is placed into the DEST location.

Exclusive-OR Logical

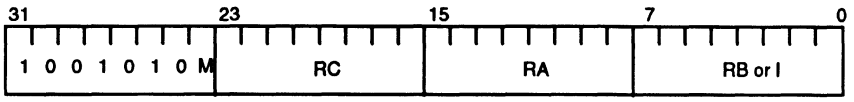
Operation: DEST ← SRCA ^ SRCB

Assembler Syntax:
XOR rc, ra, rb
or
XOR rc, ra, const8

Status: N, Z

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 94, 95 **XOR**

Description: The SRCA operand is logically exclusive-ORed, bit-by-bit, with the SRCB operand, and the result is placed into the DEST location.

18.4 INSTRUCTION INDEX BY OPERATION CODE

01	CONSTN	Constant, Negative
02	CONSTH	Constant, High
03	CONST	Constant
04	MTSRIM	Move to Special Register Immediate
06,07	LOADL	Load and Lock
08,09	CLZ	Count Leading Zeros
0A,0B	EXBYTE	Extract Byte
0C,0D	INBYTE	Insert Byte
0E,0F	STOREL	Store and Lock
10,11	ADDS	Add, Signed
12,13	ADDU	Add, Unsigned
14,15	ADD	Add
16,17	LOAD	Load
18,19	ADDCS	Add with Carry, Signed
1A,1B	ADDCU	Add with Carry, Unsigned
1C,1D	ADDC	Add with Carry
1E,1F	STORE	Store
20,21	SUBS	Subtract, Signed
22,23	SUBU	Subtract, Unsigned
24,25	SUB	Subtract
26,27	LOADSET	Load and Set
28,29	SUBCS	Subtract with Carry, Signed
2A,2B	SUBCU	Subtract with Carry, Unsigned
2C,2D	SUBC	Subtract with Carry
2E,2F	CPBYTE	Compare Bytes
30,31	SUBRS	Subtract Reverse, Signed
32,33	SUBRU	Subtract Reverse, Unsigned
34,35	SUBR	Subtract Reverse
36,37	LOADM	Load Multiple
38,39	SUBRCS	Subtract Reverse with Carry, Signed
3A,3B	SUBRCU	Subtract Reverse with Carry, Unsigned
3C,3D	SUBRC	Subtract Reverse with Carry
3E,3F	STOREM	Store Multiple
40,41	CPLT	Compare Less Than
42,43	CPLTU	Compare Less Than, Unsigned
44,45	CPLE	Compare Less Than or Equal To
46,47	CPLEU	Compare Less Than or Equal To, Unsigned
48,49	CPGT	Compare Greater Than
4A,4B	CPGTU	Compare Greater Than, Unsigned
4C,4D	CPGE	Compare Greater Than or Equal To
4E,4F	CPGEU	Compare Greater Than or Equal To, Unsigned
50,51	ASLT	Assert Less Than
52,53	ASLTU	Assert Less Than, Unsigned
54,55	ASLE	Assert Less Than or Equal To
56,57	ASLEU	Assert Less Than or Equal To, Unsigned
58,59	ASGT	Assert Greater Than

5A,5B	ASGTU	Assert Greater Than, Unsigned
5C,5D	ASGE	Assert Greater Than or Equal To
5E,5F	ASGEU	Assert Greater Than or Equal To, Unsigned
60,61	CPEQ	Compare Equal To
62,63	CPNEQ	Compare Not Equal To
64,65	MUL	Multiply Step
66,67	MULL	Multiply Last Step
68,69	DIV0	Divide Initialize
6A,6B	DIV	Divide Step
6C,6D	DIVL	Divide Last Step
6E,6F	DIVREM	Divide Remainder
70,71	ASEQ	Assert Equal To
72,73	ASNEQ	Assert Not Equal To
74,75	MULU	Multiply Step, Unsigned
78,79	INHW	Insert Half-Word
7A,7B	EXTRACT	Extract Word, Bit-Aligned
7C,7D	EXHW	Extract Half-Word
7E	EXHWS	Extract Half-Word, Sign-Extended
80,81	SLL	Shift Left Logical
82,83	SRL	Shift Right Logical
86,87	SRA	Shift Right Arithmetic
88	IRET	Interrupt Return
89	HALT	Enter HALT Mode
8C	IRETINV	Interrupt Return and Invalidate
90,91	AND	AND Logical
92,93	OR	OR Logical
94,95	XOR	Exclusive-OR Logical
96,97	XNOR	Exclusive-NOR Logical
98,99	NOR	NOR Logical
9A,9B	NAND	NAND Logical
9C,9D	ANDN	AND-NOT Logical
9E	SETIP	Set Indirect Pointers
9F	INV	Invalidate
A0,A1	JMP	Jump
A4,A5	JMPF	Jump False
A8,A9	CALL	Call Subroutine
AC,AD	JMPT	Jump True
B4,B5	JMPFDEC	Jump False and Decrement
B6	MFTLB	Move from Translation Look-Aside Buffer Register
BE	MTTLB	Move to Translation Look-Aside Buffer Register
C0	JMPI	Jump Indirect
C4	JMPFI	Jump False Indirect
C6	MFSR	Move from Special Register
C8	CALLI	Call Subroutine, Indirect
CC	JMPTI	Jump True Indirect
CE	MTSR	Move to Special Register
D7	EMULATE	Trap to Software Emulation Routine

D8–DD	Reserved for emulation (trap vector numbers 24–29)	
DE	MULTM	Integer Multiply Most Significant Bits, Signed
DF	MULTMU	Integer Multiply Most Significant Bits, Unsigned
E0	MULTIPLY	Integer Multiply, Signed
E1	DIVIDE	Integer Divide, Signed
E2	MULTIPLU	Integer Multiply, Unsigned
E3	DIVIDU	Integer Divide, Unsigned
E4	CONVERT	Convert Data Format
E5	SQRT	Square Root
E6	CLASS	Classify Floating-Point Operand
E7–E9	Reserved for emulation (trap vector number 39–41)	
EA	FEQ	Floating-Point Equal To, Single-Precision
EB	DEQ	Floating-Point Equal To, Double-Precision
EC	FGT	Floating-Point Greater Than, Single-Precision
ED	DGT	Floating-Point Greater Than, Double-Precision
EE	FGE	Floating-Point Greater Than or Equal To, Single-Precision
EF	DGE	Floating-Point Greater Than or Equal To, Double-Precision
F0	FADD	Floating-Point Add, Single-Precision
F1	DADD	Floating-Point Add, Double-Precision
F2	FSUB	Floating-Point Subtract, Single-Precision
F3	DSUB	Floating-Point Subtract, Double-Precision
F4	FMUL	Floating-Point Multiply, Single-Precision
F5	DMUL	Floating-Point Multiply, Double-Precision
F6	FDIV	Floating-Point Divide, Single-Precision
F7	DDIV	Floating-Point Divide, Double-Precision
F8	Reserved for emulation (trap vector number 56)	
F9	FDMUL	Floating-Point Multiply, Single-to-Double-Precision
FA–FF	Reserved for emulation (trap vector numbers 58–63)	

APPENDIX A

PROCESSOR REGISTER SUMMARY

Figure A-1 General-Purpose Register Organization

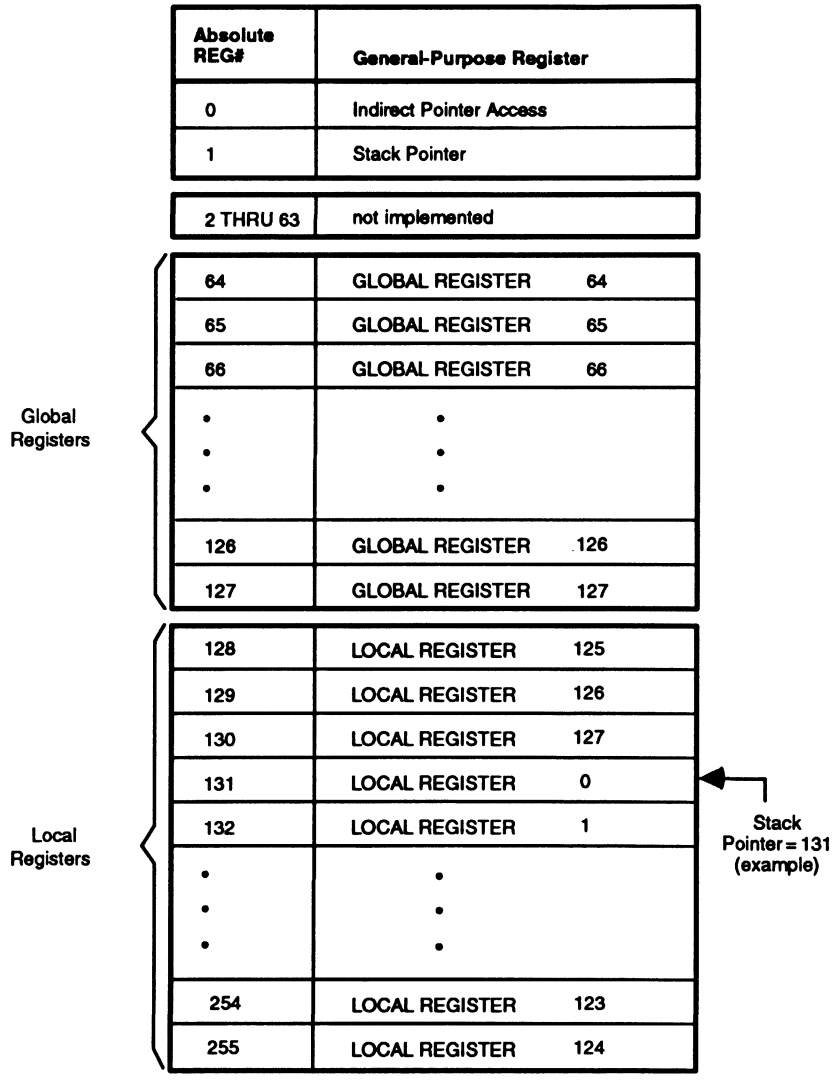


Figure A-2 Register Bank Organization

Register Bank Protect Register Bit	Absolute-Register Numbers	General-Purpose Registers
0	2 through 15	Bank 0 (unimplemented)
1	16 through 31	Bank 1 (unimplemented)
2	32 through 47	Bank 2 (unimplemented)
3	48 through 63	Bank 3 (unimplemented)
4	64 through 79	Bank 4
5	80 through 95	Bank 5
6	96 through 111	Bank 6
7	112 through 127	Bank 7
8	128 through 143	Bank 8
9	144 through 159	Bank 9
10	160 through 175	Bank 10
11	176 through 191	Bank 11
12	192 through 207	Bank 12
13	208 through 223	Bank 13
14	224 through 239	Bank 14
15	240 through 255	Bank 15

Figure A-3 Special Purpose Registers

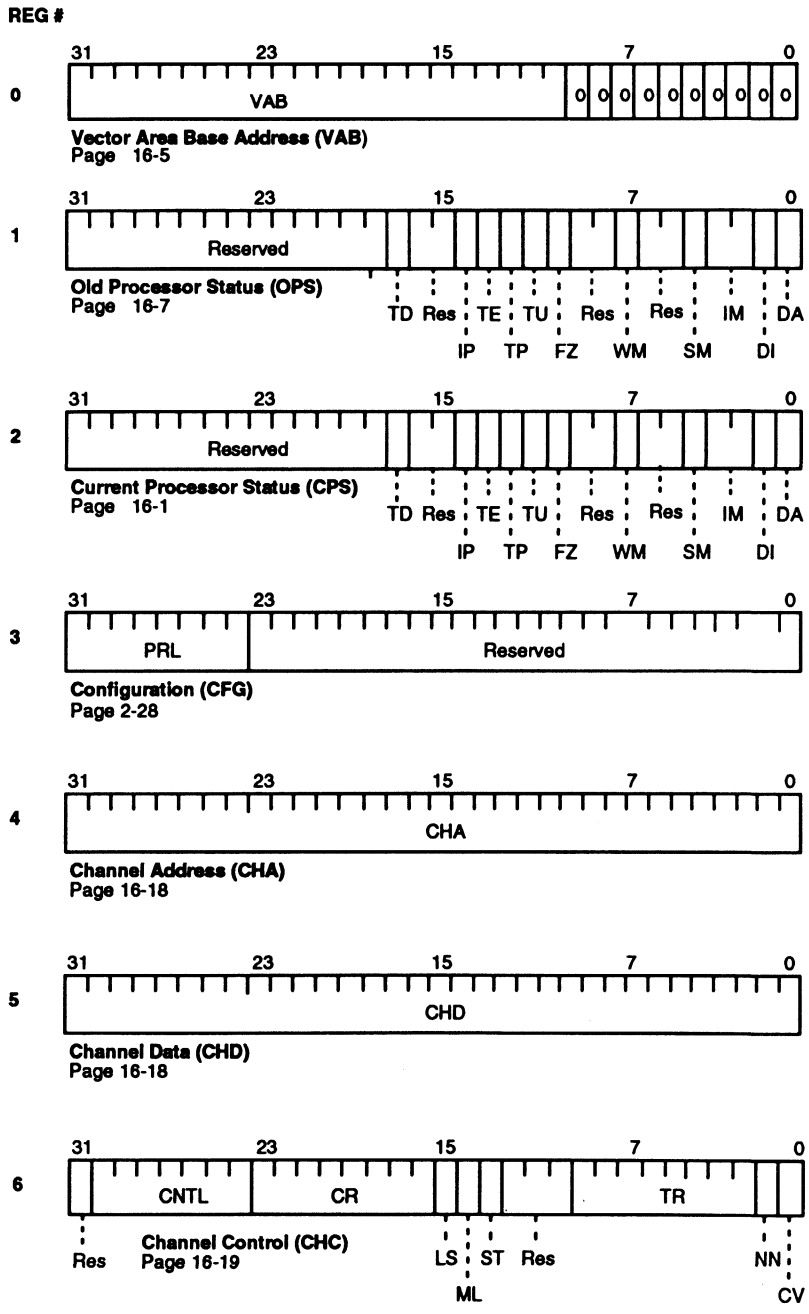
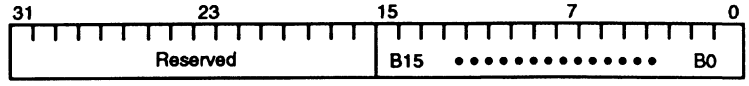


Figure A-3 Special Purpose Registers (continued)

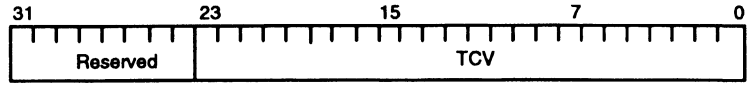
REG #

7



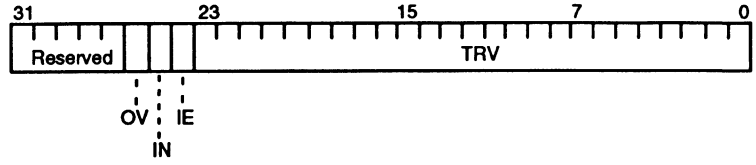
Register Bank Protect (RBP)
Page 6-3

8



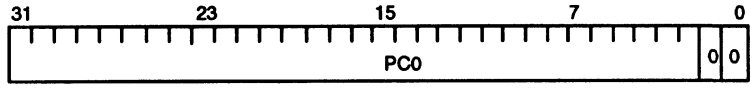
Timer Counter (TMC)
Page 16-22

9



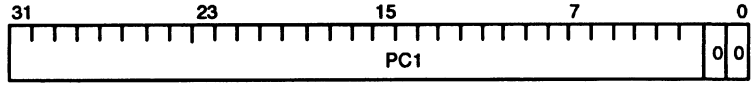
Timer Reload (TMR)
Page 16-23

10



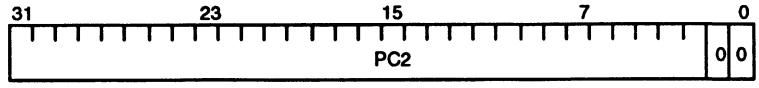
Program Counter 0 (PC0)
Page 16-9

11



Program Counter 1 (PC1)
Page 16-9

12



Program Counter 2 (PC2)
Page 16-10

Figure A-3 Special Purpose Registers (continued)

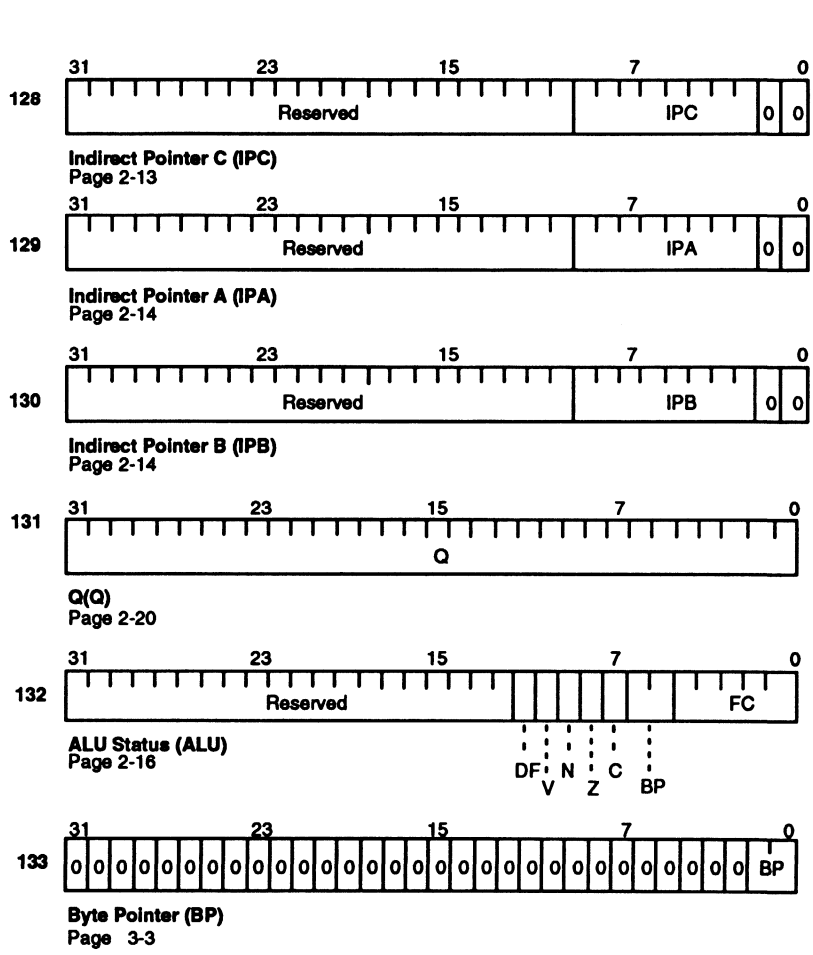


Figure A-3 Special Purpose Registers (continued)

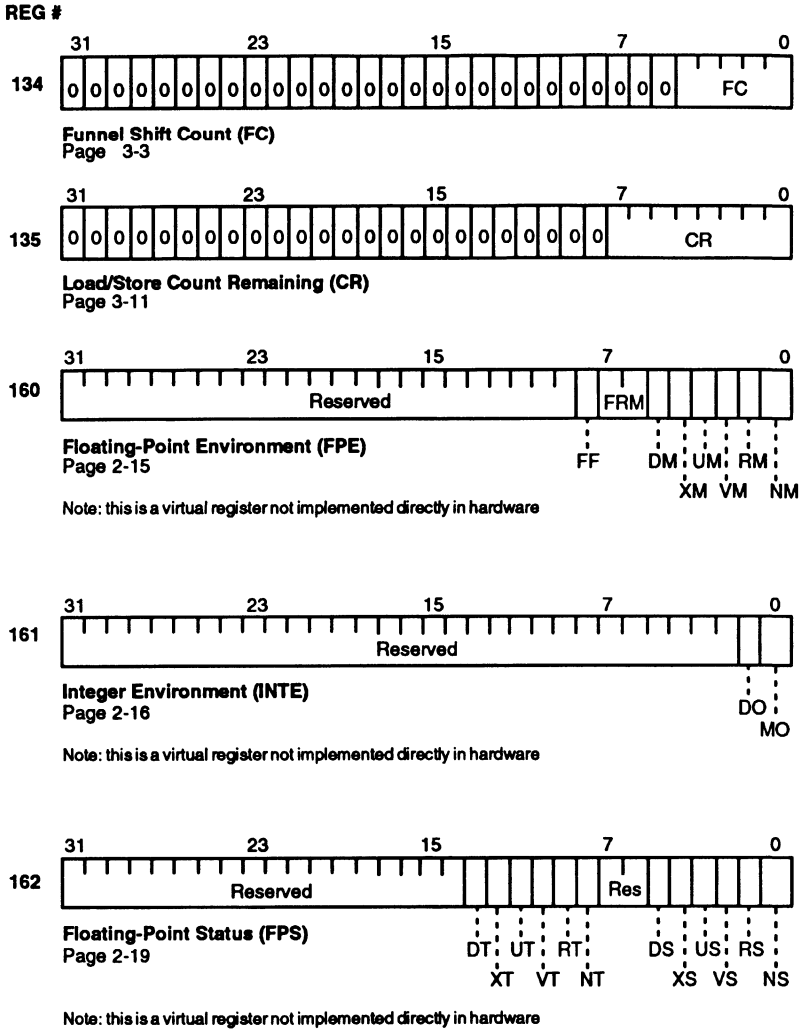


Table A-1 Register Field Summary

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
BP	Byte Pointer	ALU Status Byte Pointer	6–5 1–0
C	Carry	ALU Status	7
CHA	Channel Address	Channel Address	31–0
CHD	Channel Data	Channel Data	31–0
CNTL	Control	Channel Control	30–24
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
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
FF	Fast Floating-Point Select	Floating-Point Environment	8
FC	Funnel Shift Count	ALU Status Funnel Shift Count	4–0 4–0
FRM	Floating-Point Round Mode	Floating-Point Environment	7–6
FZ	Freeze	Current Processor Status Old Processor Status	10 10

Table A-1 Register Field Summary (continued)

Label	Field Name	Register	Bit
IE	Interrupt Enable	Timer Reload	24
IM	Interrupt Mask	Old Processor Status Current Processor Status	3–2 3–2
IN	Interrupt	Timer Reload	25
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
LS	Load/Store	Channel Control	15
ML	Multiple Operation	Channel Control	14
MO	Integer Multiplication Overflow Mask	Integer Environment	0
N	Negative	ALU Status	9
NM	Floating-Point Invalid Operation Mask	Floating-Point Environment	0
NN	Not Needed	Channel Control	1
NS	Floating-Point Invalid Operation Sticky	Floating-Point Status	0
NT	Floating-Point Invalid Operation Trap	Floating-Point Status	8
OV	Overflow	Timer Reload	26
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
PRL	Processor Release Level	Configuration	31–24
Q	Quotient/Multiplier	Q Register	31–0
RM	Floating-Point Reserved Operand Mask	Floating-Point Environment	1
RS	Floating-Point Reserved Operand Sticky	Floating-Point Status	1
RT	Floating-Point Reserved Operand Trap	Floating-Point Status	9
SM	Supervisor Mode	Current Processor Status Old Processor Status	4 4
ST	Set	Channel Control	13
TCV	Timer Count Value	Timer Counter	23–0
TD	Timer Disable	Current Processor Status Old Processor Status	17 17
TE	Trace Enable	Current Processor Status Old Processor Status	13 13

Table A-1 Register Field Summary (continued)

Label	Field Name	Register	Bit
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
UM	Floating-Point Underflow Mask	Floating-Point Environment	3
US	Floating-Point Underflow Sticky	Floating-Point Status	3
UT	Floating-Point Underflow Trap	Floating-Point Status	11
V	Overflow	ALU Status	10
VAB	Vector Area Base	Vector Area Base Address	31–10
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
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

APPENDIX B

PERIPHERAL REGISTER SUMMARY

Figure B-1 On-Chip Peripheral Registers

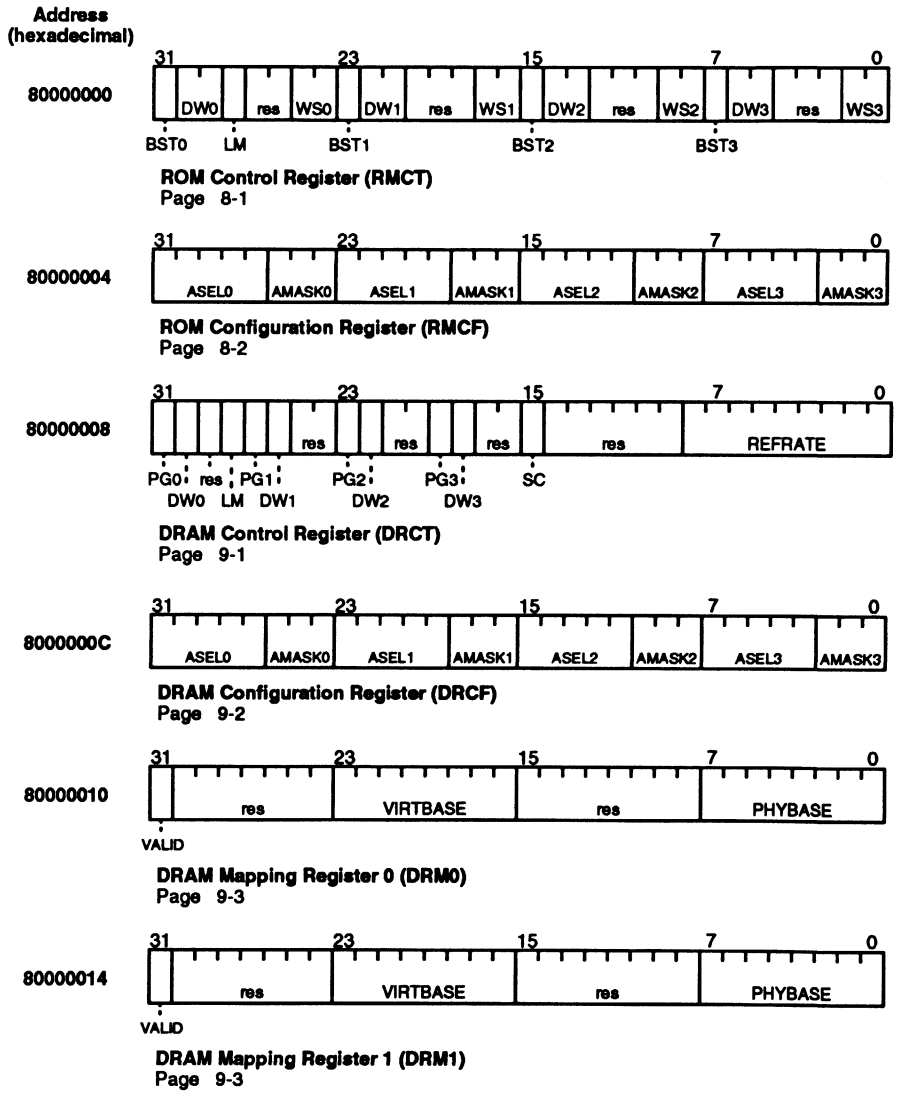
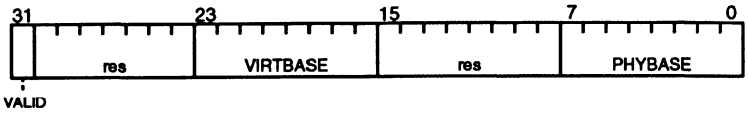


Figure B-1 On-Chip Peripheral Registers (continued)

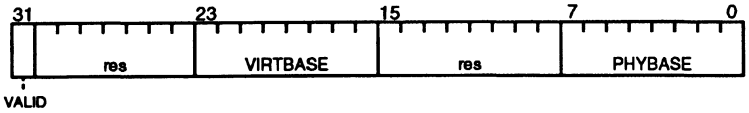
Address (hexadecimal)

80000018



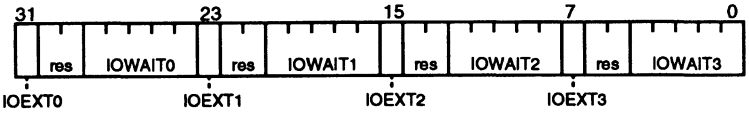
DRAM Mapping Register 2 (DRM2)
Page 9-3

8000001C



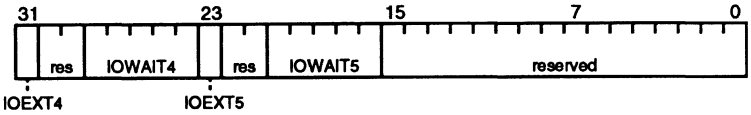
DRAM Mapping Register 3 (DRM3)
Page 9-3

80000020



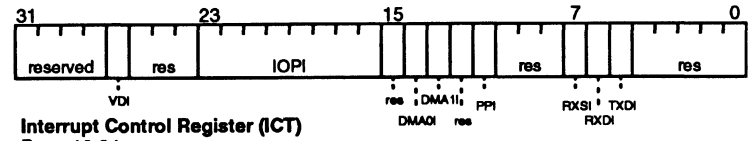
PIA Control Register 0 (PICT0)
Page 10-1

80000024



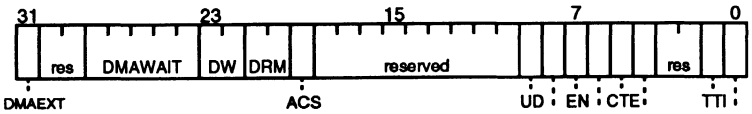
PIA Control Register 1 (PICT1)
Page 10-1

80000028



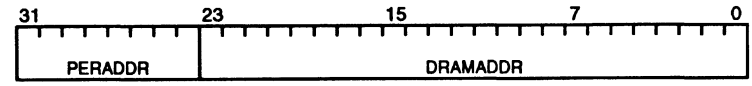
Interrupt Control Register (ICT)
Page 16-24

80000030



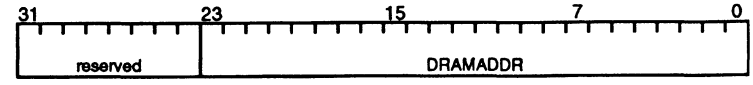
DMA0 Control Register (DMCT0)
Page 11-1

80000034



DMA0 Address Register (DMAD0)
Page 11-3

80000036



DMA0 Address Tail Register (TAD0)
Page 11-4

Figure B-1 On-Chip Peripheral Registers (continued)

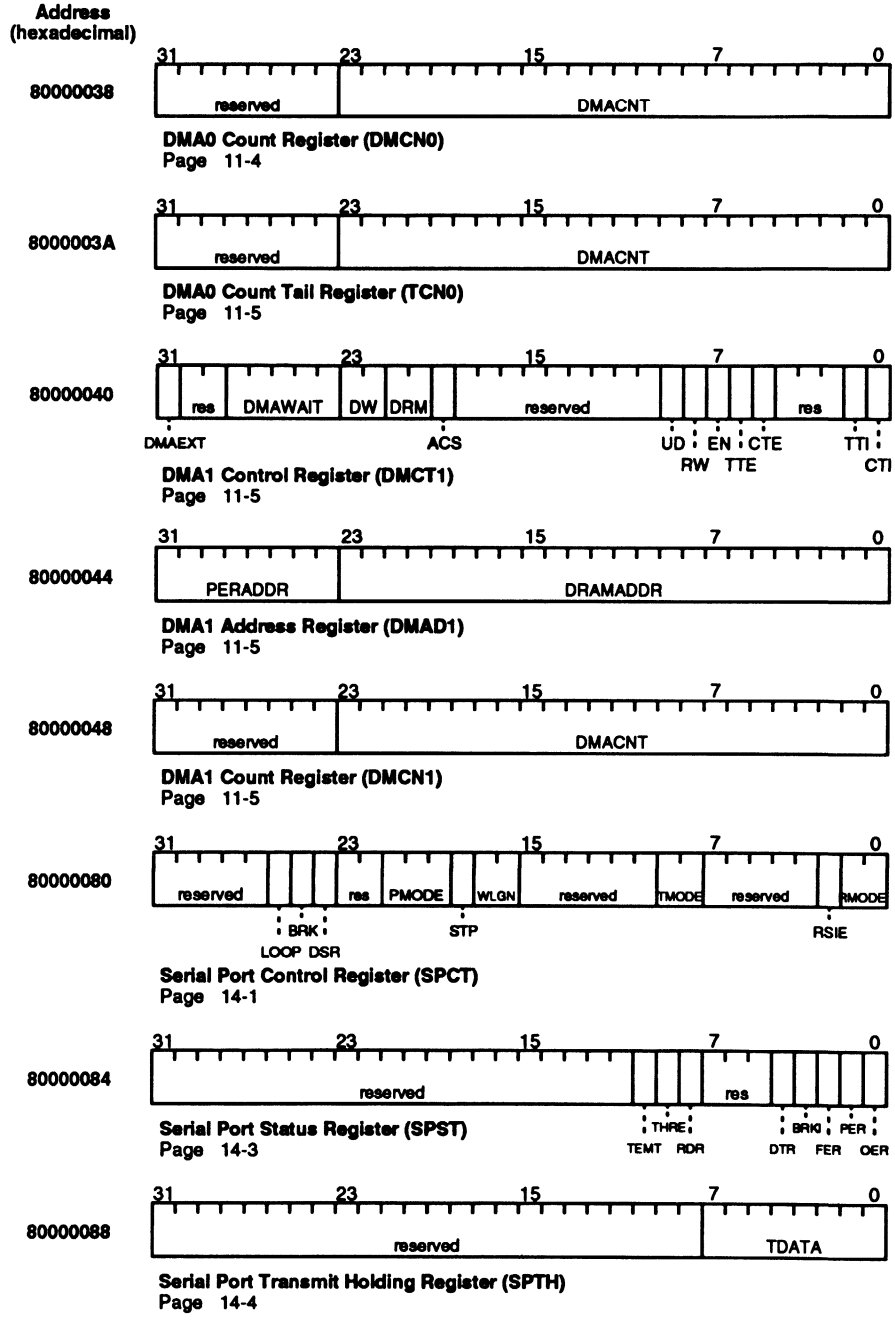


Figure B-1 On-Chip Peripheral Registers (continued)

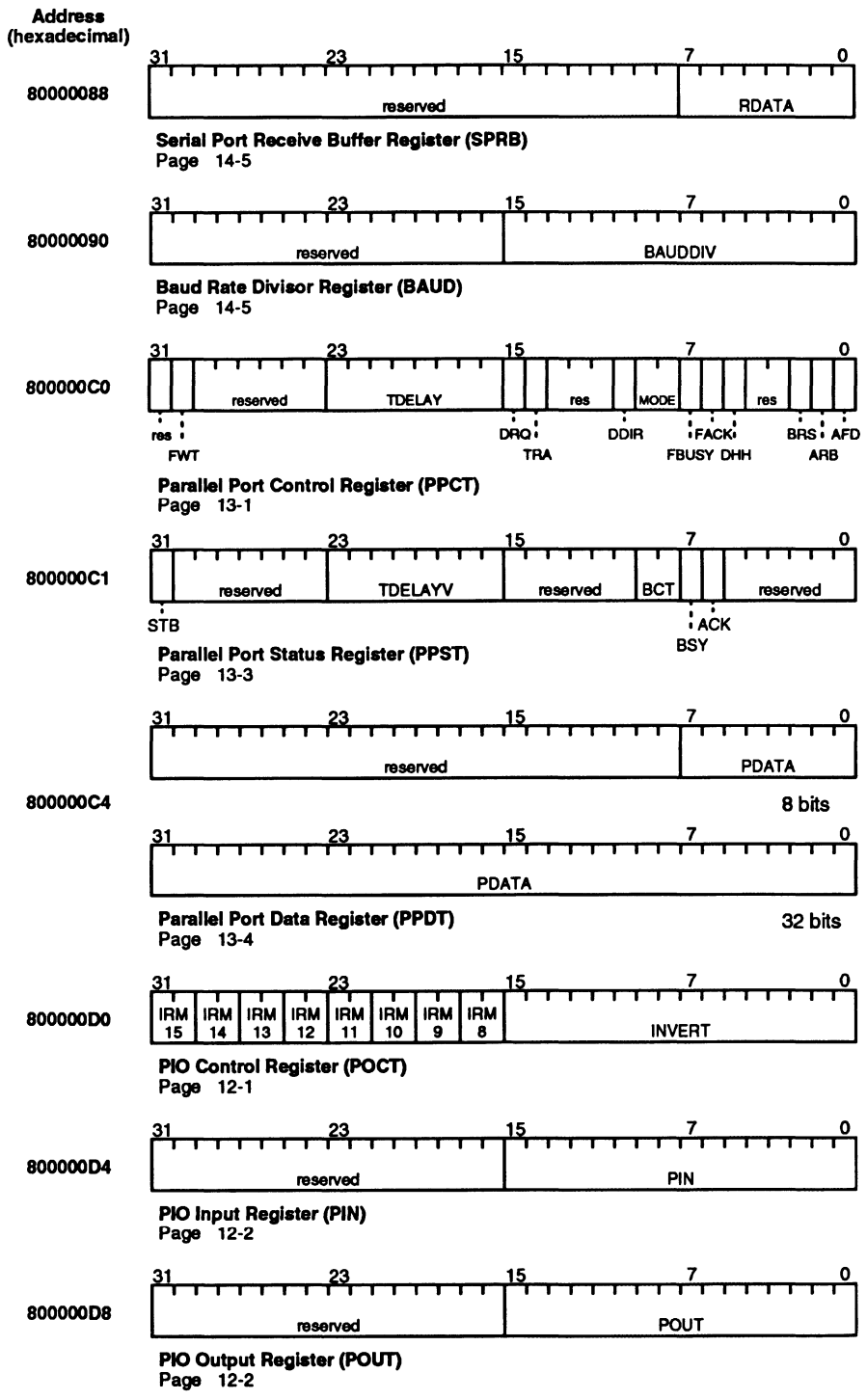


Figure B-1 On-Chip Peripheral Registers (continued)

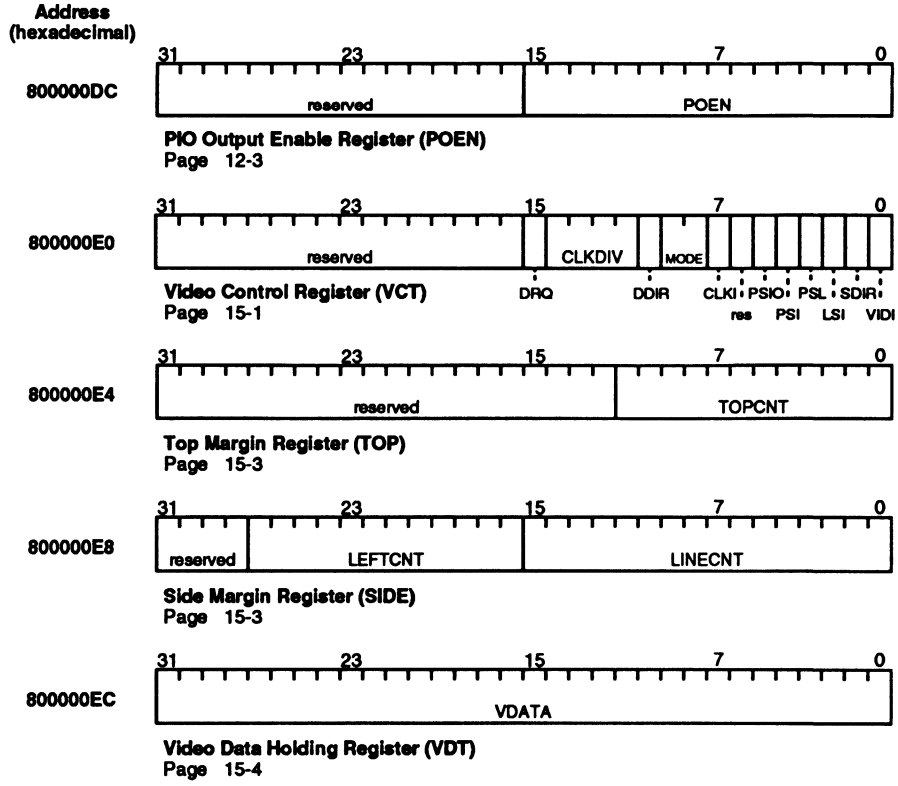


Table B-1 Peripheral Register Field Summary

Label	Field Name	Register	Bit
ACK	PACK Level	Parallel Port Status	6
ACS	Assert Chip Select	DMA0 Control DMA1 Control	19 19
AFD	Autofeed	Parallel Port Control	0
AMASK0	Address Mask, Bank 0	ROM Configuration DRAM Configuration	26–24 26–24
AMASK1	Address Mask, Bank 1	ROM Configuration DRAM Configuration	18–16 18–16
AMASK2	Address Mask, Bank 2	ROM Configuration DRAM Configuration	10–8 10–8
AMASK3	Address Mask, Bank 3	ROM Configuration DRAM Configuration	2–0 2–0
ARB	ACK Relationship to BUSY	Parallel Port Control	1
ASEL0	Address Select, Bank 0	ROM Configuration DRAM Configuration	31–27 31–27
ASEL1	Address Select, Bank 1	ROM Configuration DRAM Configuration	23–19 23–19
ASEL2	Address Select, Bank 2	ROM Configuration DRAM Configuration	15–11 15–11
ASEL3	Address Select, Bank 3	ROM Configuration DRAM Configuration	7–3 7–3
BAUDDIV	Baud Rate Divisor	Baud Rate Divisor	15–0
BCT	Byte Count	Parallel Port Status	9–8
BRK	Send Break	Serial Port Control	25
BRKI	Break Interrupt	Serial Port Status	3
BRS	BUSY Relationship to STROBE	Parallel Port Control	2
BST0	Burst-Mode ROM, Bank 0	ROM Control	31
BST1	Burst-Mode ROM, Bank 1	ROM Control	23
BST2	Burst-Mode ROM, Bank 2	ROM Control	15
BST3	Burst-Mode ROM, Bank 3	ROM Control	7
BSY	$\overline{\text{PBUSY}}$ Level	Parallel Port Status	7
CLKDIV	Clock Divide	Video Control	14–11
CLKI	Clock Invert	Video Control	7
CTE	Count Terminate Enable	DMA0 Control DMA1 Control	5 5
CTI	Count Terminate Interrupt	DMA0 Control DMA1 Control	0 0
DDIR	Data Direction	Parallel Port Control Video Control	10 10

Table B-1 Peripheral Register Field Summary (continued)

Label	Field Name	Register	Bit
DHH	Disable Hardware Handshake	Parallel Port Control	5
DMA0I	DMA Channel 0 Interrupt	Interrupt Control	14
DMA1I	DMA Channel 1 Interrupt	Interrupt Control	13
DMACNT	DMA Count	DMA0 Count DMA0 Count Tail DMA1 Count	23–0 23–0 23–0
DMAEXT	DMA Extend	DMA0 Control DMA1 Control	31 31
DMAWAIT	DMA Wait States	DMA0 Control DMA1 Control	28–24 28–24
DRAMADDR	DRAM Address	DMA0 Address DMA0 Address Tail DMA1 Address	23–0 23–0 23–0
DRM	DMA Request Mode	DMA0 Control DMA1 Control	21–20 21–20
DRQ	Data Request	Parallel Port Control Video Control	15 15
DSR	Data Set Ready	Serial Port Control	24
DTR	Data Terminal Ready	Serial Port Status	4
DW	Data Width	DMA0 Control DMA1 Control	22–23 22–23
DW0	Data Width, Bank 0	ROM Control DRAM Control	30–29 30
DW1	Data Width, Bank 1	ROM Control DRAM Control	22–21 26
DW2	Data Width, Bank 2	ROM Control DRAM Control	14–13 22
DW3	Data Width, Bank 3	ROM Control DRAM Control	6–5 18
EN	Enable	DMA0 Control DMA1 Control	7 7
FACK	Force ACK	Parallel Port Control	6
FBUSY	Force Busy	Parallel Port Control	7
FER	Framing Error	Serial Port Status	2
FWT	Full Word Transfer	Parallel Port Control	30
INVERT	PIO Inversion	PIO Control	15–0

Table B-1 Peripheral Register Field Summary (continued)

Label	Field Name	Register	Bit
IOEXT0	Input/Output Extend, Region 0	PIA Control 0	31
IOEXT1	Input/Output Extend, Region 1	PIA Control 0	23
IOEXT2	Input/Output Extend, Region 2	PIA Control 0	15
IOEXT3	Input/Output Extend, Region 3	PIA Control 0	7
IOEXT4	Input/Output Extend, Region 4	PIA Control 1	31
IOEXT5	Input/Output Extend, Region 5	PIA Control 1	23
IOPI	I/O Port Interrupt	Interrupt Control	23–15
IOWAIT0	Input/Output Wait States, Region 0	PIA Control 0	28–24
IOWAIT1	Input/Output Wait States, Region 1	PIA Control 0	20–16
IOWAIT2	Input/Output Wait States, Region 2	PIA Control 0	12–8
IOWAIT3	Input/Output Wait States, Region 3	PIA Control 0	4–0
IOWAIT4	Input/Output Wait States, Region 4	PIA Control 1	28–24
IOWAIT5	Input/Output Wait States, Region 5	PIA Control 1	20–16
IRM8	Interrupt Request Mode, PIO8	PIO Control	17–16
IRM9	Interrupt Request Mode, PIO9	PIO Control	19–18
IRM10	Interrupt Request Mode, PIO10	PIO Control	21–20
IRM11	Interrupt Request Mode, PIO11	PIO Control	23–22
IRM12	Interrupt Request Mode, PIO12	PIO Control	25–24
IRM13	Interrupt Request Mode, PIO13	PIO Control	27–26
IRM14	Interrupt Request Mode, PIO14	PIO Control	29–18
IRM15	Interrupt Request Mode, PIO15	PIO Control	31–30
LEFTCNT	Left Margin Count	Side Margin	27–16
LINECNT	Line Count	Side Margin	15–0
LM	Large Memory	ROM Control DRAM Control	28 28
LOOP	Loopback	Serial Port Control	26
LSI	Line Sync Invert	Video Control	2
MODE	Parallel Port Mode Video Interface Mode	Parallel Port Control Video Control	9–8 9–8
OER	Overrun Error	Serial Port Status	0
PDATA	Parallel Port Data	Parallel Port Data	7–0 31–0
PER	Parity Error	Serial Port Status	1
PERADDR	Peripheral Address	DMA0 Address DMA1 Address	31–24 31–24

Table B-1 Peripheral Register Field Summary (continued)

Label	Field Name	Register	Bit
PG0	Page-Mode DRAM, Bank 0	DRAM Control	31
PG1	Page-Mode DRAM, Bank 1	DRAM Control	27
PG2	Page-Mode DRAM, Bank 2	DRAM Control	23
PG3	Page-Mode DRAM, Bank 3	DRAM Control	19
PHYBASE	Physical Base Address	DRAM Mapping 0 DRAM Mapping 1 DRAM Mapping 2 DRAM Mapping 3	7–0 7–0 7–0 7–0
PIN	PIO Input	PIO Input	15–0
PMODE	Parity Mode	Serial Port Control	21–19
POEN	PIO Output Enable	PIO Output Enable	15–0
POUT	PIO Output	PIO Output	15–0
PPI	Parallel Port Interrupt	Interrupt Control	11
PSI	Page Sync Invert	Video Control	4
PSIO	Page Sync Input/Output	Video Control	5
PSL	Page Sync Level	Video Control	3
QEN	Queue Enable	DMA0 Control	4
RDATA	Receive Data	Serial Port Receive Buffer	7–0
RDR	Receive Data Ready	Serial Port Status	8
REFRATE	Refresh Rate	DRAM Control	8–0
RMODE	Receive Mode	Serial Port Control	1–0
RSIE	Receive Status Interrupt Enable	Serial Port Control	2
RW	Read/Write	DMA0 Control DMA1 Control	8 8
RXDI	Serial Port Receive Data Interrupt	Interrupt Control	6
RXSI	Serial Port Receive Status Interrupt	Interrupt Control	7
SC	Static-Column DRAM	DRAM Control	15
SDIR	Shift Direction	Video Control	1
STB	PSTROBE Level	Parallel Port Status	31
STP	Stop Bits	Serial Port Control	18
TDATA	Transmit Data	Serial Port Transmit Holding	7–0
TDELAY	Transfer Delay	Parallel Port Control	23–16
TDELAYV	TDELAY Counter Value	Parallel Port Status	23–16
TEMT	Transmitter Empty	Serial Port Status	10
THRE	Transmit Holding Register Empty	Serial Port Status	9

Table B-1 Peripheral Register Field Summary (continued)

Label	Field Name	Register	Bit
TMODE	Transmit Mode	Serial Port Control	9–8
TOPCNT	Top Margin Count	Top Margin	11–0
TRA	Transfer Active	Parallel Port Control	14
TTE	TDMA Terminate Enable	DMA0 Control DMA1 Control	6 6
TTI	TDMA Terminate Interrupt	DMA0 Control DMA1 Control	1 1
TXDI	Serial Port Transmit Data Interrupt	Interrupt Control	5
UD	Transfer Up/Down	DMA0 Control DMA1 Control	9 9
VALID	Valid Mapping	DRAM Mapping 0 DRAM Mapping 1 DRAM Mapping 2 DRAM Mapping 3	31 31 31 31
VIDI	Video Invert	Video Control	0
VDATA	Video Data	Video Data Holding	31–0
VDI	Video Interrupt	Interrupt Control	27
VIRTBASE	Virtual Base Address	DRAM Mapping 0 DRAM Mapping 1 DRAM Mapping 2 DRAM Mapping 3	23–16 23–16 23–16 23–16
WLGN	Word Length	Serial Port Control	17–16
WS0	Wait States, Bank 0	ROM Control	25–24
WS1	Wait States, Bank 1	ROM Control	17–16
WS2	Wait States, Bank 2	ROM Control	9–8
WS3	Wait States, Bank 3	ROM Control	1–0

ADVANCE INFORMATION

Am29200™

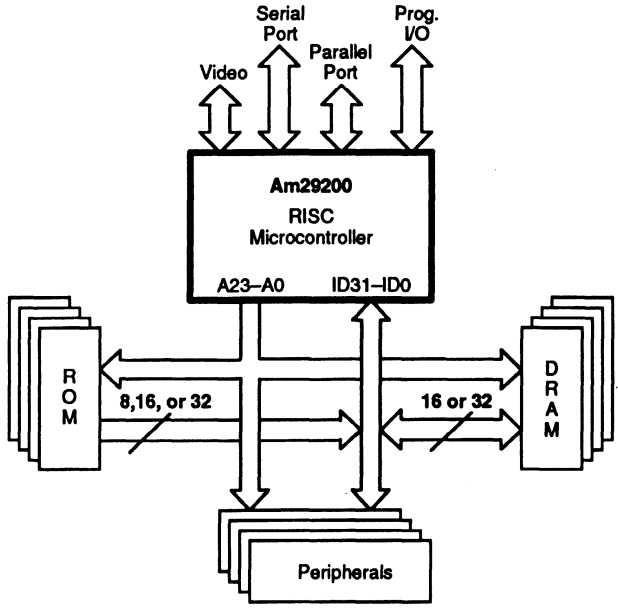
RISC Microcontroller

Advanced Micro Devices

DISTINCTIVE CHARACTERISTICS

- Completely integrated system for embedded applications
- Full 32-bit architecture
- CMOS technology/TTL-compatible
- 16-MHz operating frequency
- 8 million instructions per second (MIPS) sustained at 16 MHz
- 304-Mb address space
- 192 general-purpose registers
- Three-address instruction architecture
- Fully pipelined
- Glueless system interfaces with on-chip wait state control
- 8-, 16-, or 32-bit ROM interface
- 16- or 32-bit DRAM interface
- Burst-Mode and Page-Mode access support
- DRAM mapping on-chip
- Two-channel DMA Controller
- Six-port Peripheral Interface Adapter
- 16-line Programmable I/O Port
- Bi-directional Video Interface
- Serial Port (UART)
- Bi-directional Parallel Port for IBM-compatible personal computers
- Interrupt Controller
- On-chip Timer
- Software compatible with all other 29K™ Family microprocessors
- Advanced debugging support
- IEEE Std. 1149.1–1990 (JTAG) compliant Standard Test Access Port and Boundary Scan Architecture

SIMPLIFIED SYSTEM DIAGRAM



Publication #: 16361 Rev. B Amendment: /0
Issue Date: November 1991

GENERAL DESCRIPTION

The Am29200 RISC microcontroller is a highly integrated, general-purpose, 32-bit microcontroller implemented in CMOS technology. Through submicron technology, the Am29200 microcontroller incorporates a complete set of system facilities commonly found in printing, imaging, graphics, and other embedded applications. The on-chip functions include: a ROM Controller, a DRAM Controller, a Peripheral Interface Adapter, a DMA Controller, a Video Interface, a Programmable I/O Port, a Parallel Port, a Serial Port, and an Interrupt Controller.

The Am29200 microcontroller meets the common requirements of low-cost embedded applications such as laser beam printers, graphics processors, X terminals and servers, application program interface (API) accelerators, and scanners. The Am29200 microcontroller is well suited for these applications since it provides better performance than the CISC processors typically used in these application. Compared to the CISC processors, the Am29200 microcontroller offers lower system cost and complete design flexibility for the designer. Coupled with hardware and software development tools from AMD and the AMD Fusion29K℠ partners, the Am29200 microcontroller provides very quick time-to-market.

The Am29200 microcontroller is available in a 168-lead Plastic Quad Flat Pack (PQFP) package. The PQFP has 140 signal pins and 28 power/ground pins.

29K Family Development Support Products

Contact your local AMD representative for information on the complete set of development support tools.

The following software-development products are available on several hosts:

- Optimizing compilers for common high-level languages
- Assembler and utility packages
- Source- and assembly-level software debuggers
- Target-resident development monitors
- Simulators

RELATED AMD PRODUCTS

29K Family Devices

Part No.	Description
Am29000™	Streamlined Instruction Microprocessor
Am29005™	Low-Cost Streamlined Instruction Microprocessor
Am29030™	RISC Microprocessor with 8-Kb Instruction Cache
Am29035™	RISC Microprocessor with 4-Kb Instruction Cache
Am29050™	Streamlined Instruction Microprocessor with On-chip Floating-Point

Third Party Development Support Products

The Fusion29K Program of Partnerships for Application Solutions provides the user with a vast array of products designed to meet critical time-to-market needs.

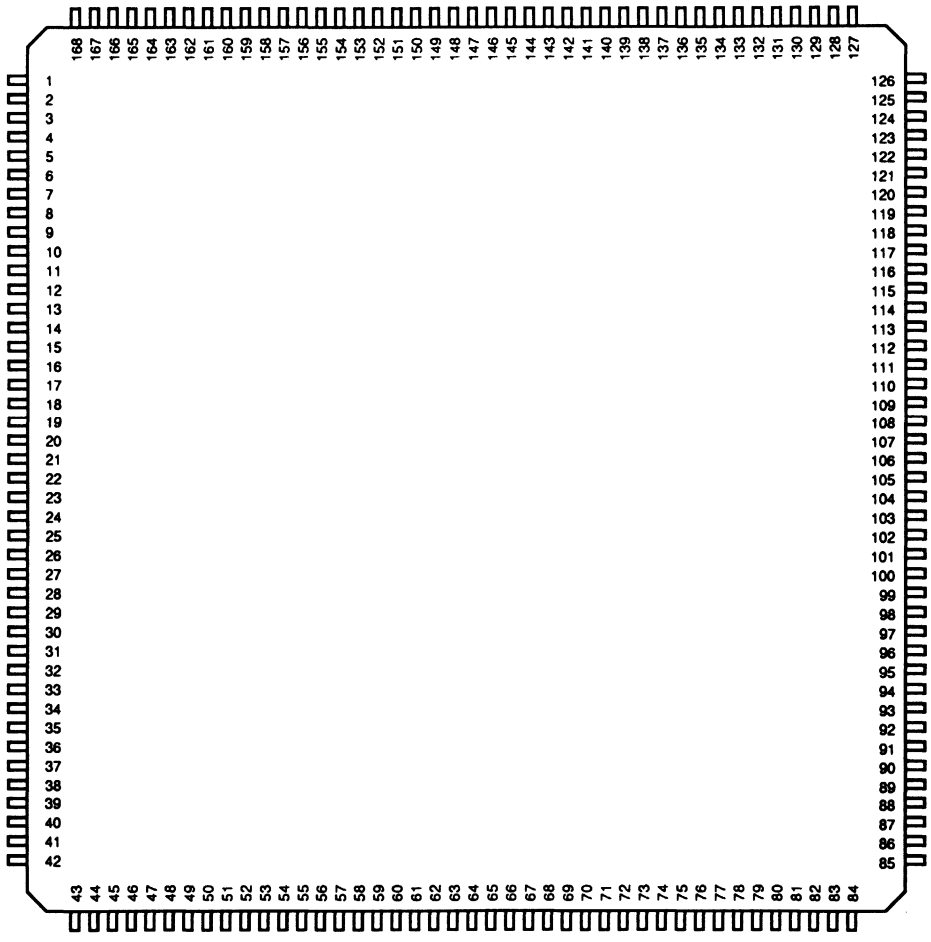
Products/solutions available through the AMD Fusion29K partners include:

- Silicon products
- Software generation and debug tools
- Hardware development tools
- Board level products
- Laser printer solutions
- Multi-user, kernel, and real-time operating systems
- Graphics solutions
- Networking and communications solutions
- Manufacturing support
- Custom support

CONNECTION DIAGRAM

168-Lead PQFP (Top View)

168-Pin PQFP

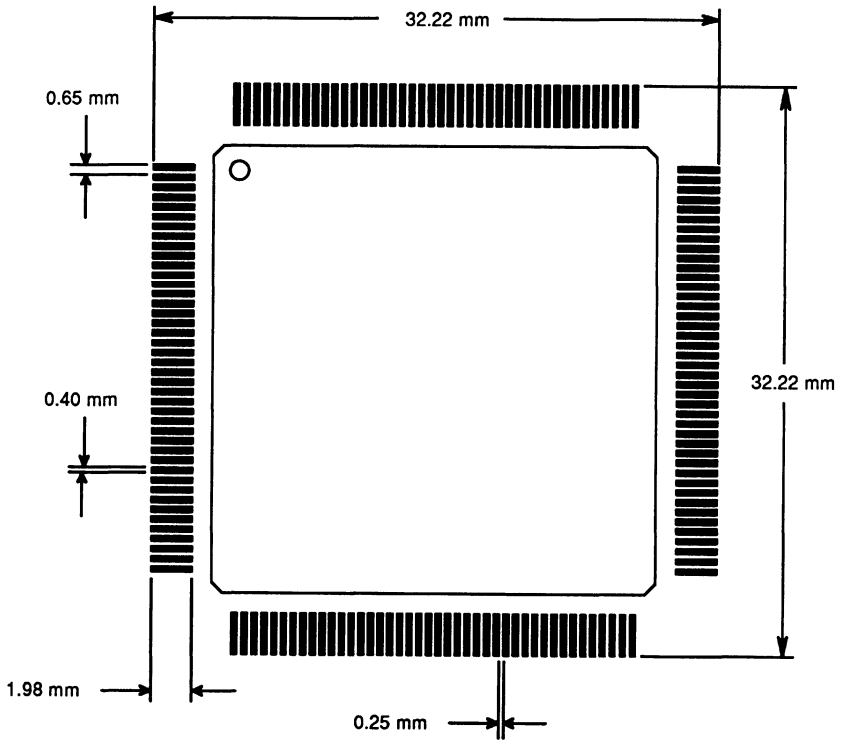


All values are typical and preliminary.

SOLDER LAND RECOMMENDATIONS

168-Lead PQFP

Top View
(not to scale)



Note: All values are typical and preliminary

PQFP PIN DESIGNATION

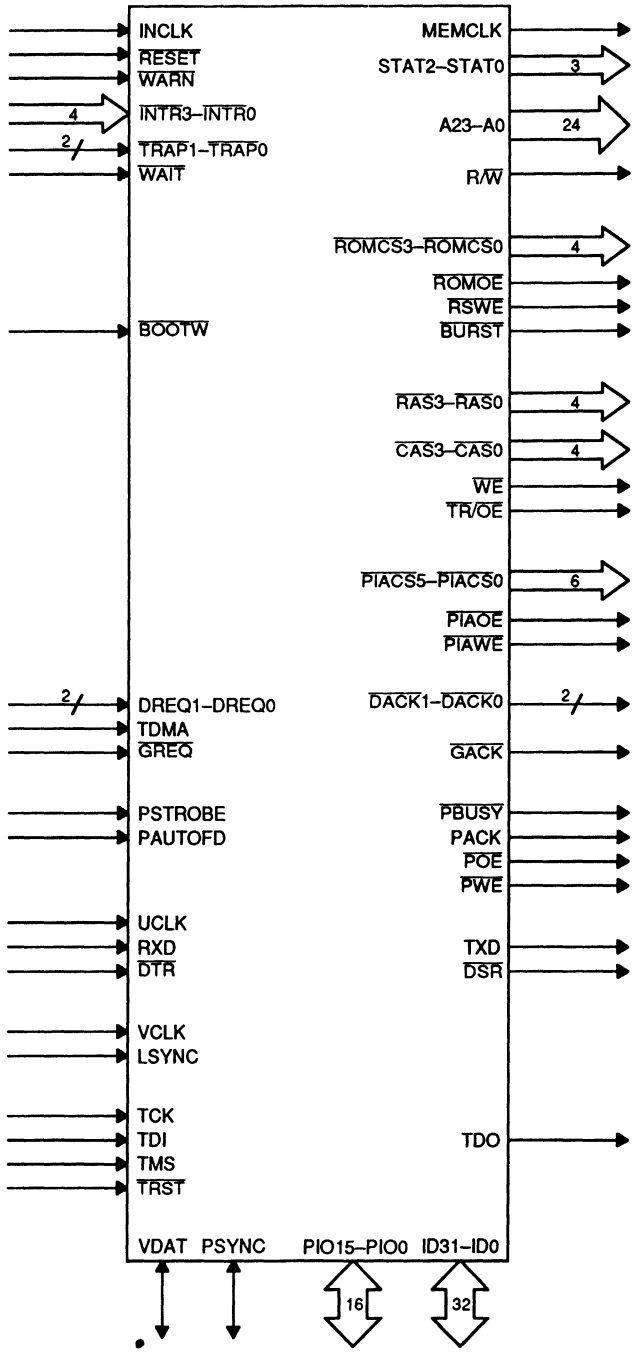
(Sorted by Pin Number)

Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
1	V_{CC}	43	V_{CC}	85	GND	127	PIO12
2	GND	44	GND	86	V_{CC}	128	PIO11
3	MEMCLK	45	$\overline{DTR}$	87	A23	129	PIO10
4	INCLK	46	RXD	88	A22	130	PIO9
5	V_{CC}	47	UCLK	89	A21	131	PIO8
6	GND	48	V_{CC}	90	A20	132	PIO7
7	GND	49	GND	91	A19	133	V_{CC}
8	V_{CC}	50	$\overline{DSR}$	92	A18	134	GND
9	ID31	51	TXD	93	A17	135	PIO6
10	ID30	52	$\overline{ROMCS3}$	94	A16	136	PIO5
11	ID29	53	$\overline{ROMCS2}$	95	A15	137	PIO4
12	ID28	54	$\overline{ROMCS1}$	96	A14	138	PIO3
13	GND	55	$\overline{ROMCS0}$	97	A13	139	PIO2
14	ID27	56	$\overline{BURST}$	98	A12	140	PIO1
15	ID26	57	$\overline{RSWE}$	99	A11	141	PIO0
16	ID25	58	$\overline{ROMOE}$	100	A10	142	TDO
17	ID24	59	$\overline{RAS3}$	101	A9	143	STAT2
18	ID23	60	$\overline{RAS2}$	102	A8	144	STAT1
19	ID22	61	$\overline{RAS1}$	103	A7	145	STAT0
20	ID21	62	$\overline{RAS0}$	104	A6	146	VDAT
21	ID20	63	$\overline{CAS3}$	105	A5	147	PSYNC
22	ID19	64	$\overline{CAS2}$	106	A4	148	GND
23	V_{CC}	65	V_{CC}	107	A3	149	V_{CC}
24	ID18	66	GND	108	A2	150	$\overline{GREQ}$
25	ID17	67	$\overline{CAS1}$	109	A1	151	DREQ1
26	ID16	68	$\overline{CAS0}$	110	A0	152	DREQ0
27	ID15	69	$\overline{TR/OE}$	111	V_{CC}	153	TDMA
28	ID14	70	$\overline{WE}$	112	GND	154	$\overline{TRAP0}$
29	ID13	71	$\overline{GACK}$	113	BOOTW	155	$\overline{TRAP1}$
30	ID12	72	$\overline{PIACS5}$	114	$\overline{WAIT}$	156	$\overline{INTR0}$
31	ID11	73	$\overline{PIACS4}$	115	PAUTOFD	157	$\overline{INTR1}$
32	ID10	74	$\overline{PIACS3}$	116	PSTROBE	158	$\overline{INTR2}$
33	ID9	75	$\overline{PIACS2}$	117	V_{CC}	159	$\overline{INTR3}$
34	ID8	76	$\overline{PIACS1}$	118	GND	160	$\overline{WARN}$
35	ID7	77	$\overline{PIACS0}$	119	$\overline{PWE}$	161	GND
36	ID6	78	$\overline{PIAWE}$	120	$\overline{POE}$	162	VCLK
37	ID5	79	$\overline{PIAOE}$	121	PACK	163	LSYNC
38	ID4	80	R/$\overline{W}$	122	$\overline{PBUSY}$	164	TMS
39	ID3	81	$\overline{DACK1}$	123	PIO15	165	$\overline{TRST}$
40	ID2	82	$\overline{DACK0}$	124	PIO14	166	TCK
41	ID1	83	GND	125	PIO13	167	TDI
42	ID0	84	V_{CC}	126	GND	168	$\overline{RESET}$

PQFP PIN DESIGNATION
(Sorted by Pin Name)

Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
110	A0	49	GND	9	ID31	60	$\overline{RAS2}$
109	A1	66	GND	4	INCLK	59	$\overline{RAS3}$
108	A2	83	GND	156	$\overline{INTR0}$	168	$\overline{RESET}$
107	A3	85	GND	157	$\overline{INTR1}$	55	$\overline{ROMCS0}$
106	A4	112	GND	158	$\overline{INTR2}$	54	$\overline{ROMCS1}$
105	A5	118	GND	159	$\overline{INTR3}$	53	$\overline{ROMCS2}$
104	A6	126	GND	163	LSYNC	52	$\overline{ROMCS3}$
103	A7	134	GND	3	MEMCLK	58	$\overline{ROMOE}$
102	A8	148	GND	121	PACK	57	$\overline{RSWE}$
101	A9	161	GND	115	PAUTOFD	46	RXD
100	A10	150	$\overline{GREQ}$	122	$\overline{PBUSY}$	145	STAT0
99	A11	42	ID0	77	$\overline{PIACS0}$	144	STAT1
98	A12	41	ID1	76	$\overline{PIACS1}$	143	STAT2
97	A13	40	ID2	75	$\overline{PIACS2}$	166	TCK
96	A14	39	ID3	74	$\overline{PIACS3}$	167	TDI
95	A15	38	ID4	73	$\overline{PIACS4}$	153	TDMA
94	A16	37	ID5	72	$\overline{PIACS5}$	142	TDO
93	A17	36	ID6	79	$\overline{PIAOE}$	164	TMS
92	A18	35	ID7	78	$\overline{PIAWE}$	69	$\overline{TR/OE}$
91	A19	34	ID8	141	PIO0	154	$\overline{TRAP0}$
90	A20	33	ID9	140	PIO1	155	$\overline{TRAP1}$
89	A21	32	ID10	139	PIO2	165	$\overline{TRST}$
88	A22	31	ID11	138	PIO3	51	TXD
87	A23	30	ID12	137	PIO4	47	UCLK
113	BOOTW	29	ID13	136	PIO5	1	V_{CC}
56	$\overline{BURST}$	28	ID14	135	PIO6	5	V_{CC}
68	$\overline{CAS0}$	27	ID15	132	PIO7	8	V_{CC}
67	$\overline{CAS1}$	26	ID16	131	PIO8	23	V_{CC}
64	$\overline{CAS2}$	25	ID17	130	PIO9	43	V_{CC}
63	$\overline{CAS3}$	24	ID18	129	PIO10	48	V_{CC}
82	$\overline{DACK0}$	22	ID19	128	PIO11	65	V_{CC}
81	$\overline{DACK1}$	21	ID20	127	PIO12	84	V_{CC}
152	DREQ0	20	ID21	125	PIO13	86	V_{CC}
151	DREQ1	19	ID22	124	PIO14	111	V_{CC}
50	$\overline{DSR}$	18	ID23	123	PIO15	117	V_{CC}
45	$\overline{DTR}$	17	ID24	120	$\overline{POE}$	133	V_{CC}
71	$\overline{GACK}$	16	ID25	116	PSTROBE	149	V_{CC}
2	GND	15	ID26	147	PSYNC	162	VCLK
6	GND	14	ID27	119	$\overline{PWE}$	146	VDAT
7	GND	12	ID28	80	R/$\overline{W}$	114	$\overline{WAIT}$
13	GND	11	ID29	62	$\overline{RAS0}$	160	$\overline{WARN}$
44	GND	10	ID30	61	$\overline{RAS1}$	70	$\overline{WE}$

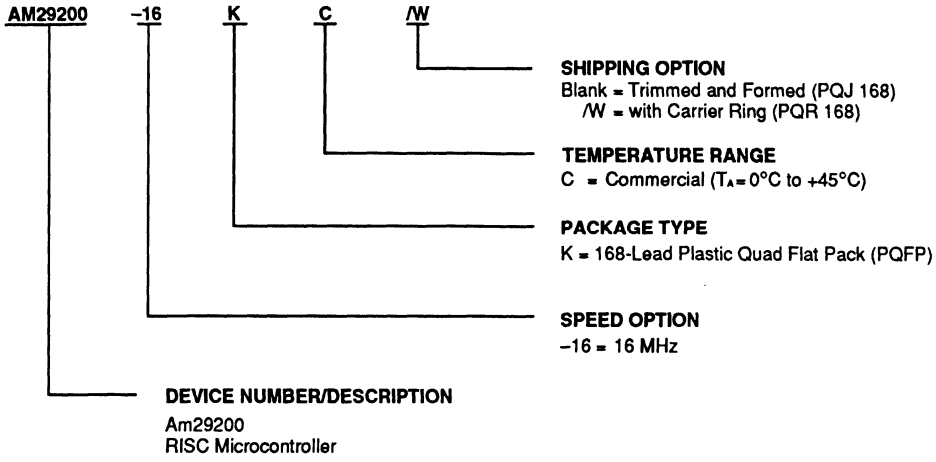
LOGIC SYMBOL



ORDERING INFORMATION

Standard Products

AMD standard products are available in several packages and operating ranges. Valid order numbers are formed by a combination of the elements below.



Valid Combinations	
AM29200	-16KC
	-16KC/W

Valid Combinations

Valid Combinations lists configurations planned to be supported in volume. Consult the local AMD sales office to confirm availability of specific valid combinations, to check on newly released combinations, and to obtain additional data on AMD standard military grade products.

ABSOLUTE MAXIMUM RATINGS

Storage Temperature ..... −65°C to +150°C
Voltage on any Pin
with Respect to GND ..... −0.5 to V_{CC} + 0.5 V

Stresses above those listed under ABSOLUTE MAXIMUM RATINGS may cause permanent device failure. Functionality at or above these limits is not implied. Exposure to absolute maximum ratings for extended periods may affect device reliability.

OPERATING RANGES

Commercial (C) Devices

Ambient Temperature (T_A) ..... 0°C to +55°C
Supply Voltage (V_{CC}) ..... +4.75 to +5.25 V

Operating ranges define those limits between which the functionality of the device is guaranteed.

DC CHARACTERISTICS over COMMERCIAL operating ranges

Parameter Symbol	Parameter Description	Test Conditions	Advance Information Min	Advance Information Max	Unit
V_{IL}	Input Low Voltage		−0.5	0.8	V
V_{IH}	Input High Voltage		2.0	V_{CC} +0.5	V
$V_{ILINCLK}$	INCLK Input Low Voltage		−0.5	0.8	V
$V_{IHINCLK}$	INCLK Input High Voltage		2.0	V_{CC} +0.5	V
V_{OL}	Output Low Voltage for All Outputs except MEMCLK	I_{OL} = 3.2 mA		0.45	V
V_{OH}	Output High Voltage for All Outputs except MEMCLK	I_{OH} = −400 µA	2.4		V
I_{LI}	Input Leakage Current (Note 1)	0.45 V ≤ V_{IN} ≤ V_{CC} − 0.45 V		±10 or +10/−200	µA
I_{LO}	Output Leakage Current	0.45 V ≤ V_{OUT} ≤ V_{CC} − 0.45 V		±10	µA
I_{CCOP}	Operating Power-Supply Current	V_{CC} = 5.25 V, Outputs Floating; Holding $\overline{RESET}$ active		14	mA/MHz
V_{OLC}	MEMCLK Output Low Voltage	I_{OLC} = 20 mA		0.6	V
V_{OHC}	MEMCLK Output High Voltage	I_{OHC} = −20 mA	V_{CC}−0.6		V

Note 1. The Low input leakage current for the inputs TCK, TDI, $\overline{TRST}$, DREQ1–DREQ0, and $\overline{GREQ}$ is −200 µA. These pins have internal pull-up resistors.

CAPACITANCE

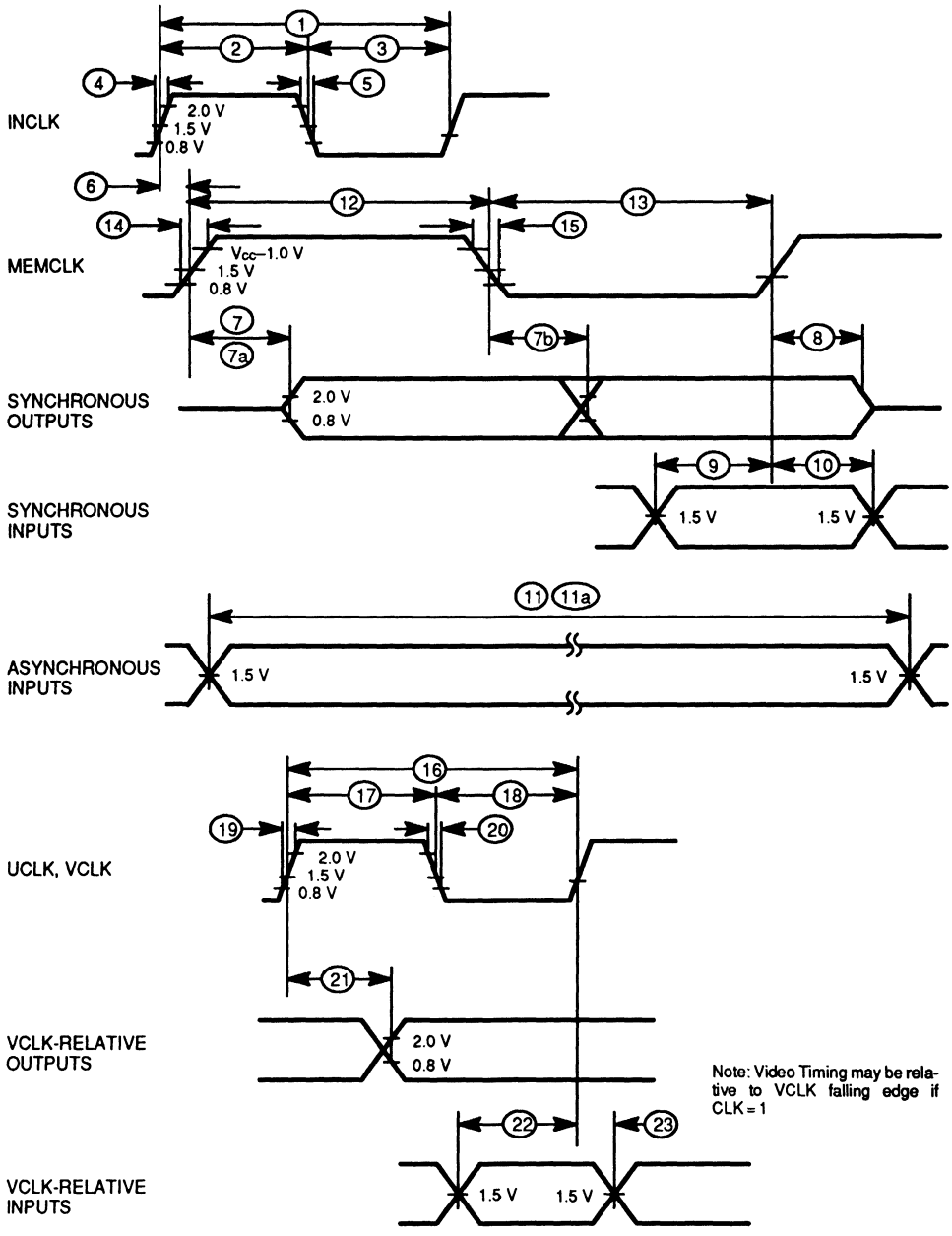
Parameter Symbol	Parameter Description	Test Conditions	Advance Information Min	Advance Information Max	Unit
C_{IN}	Input Capacitance	fC = 10 MHz		15	pF
C_{INCLK}	INCLK Input Capacitance			15	pF
C_{MEMCLK}	MEMCLK Capacitance			20	pF
C_{OUT}	Output Capacitance			20	pF
C_{VO}	I/O Pin Capacitance			20	pF

SWITCHING CHARACTERISTICS over COMMERCIAL operating range

No.	Parameter Description	Test Conditions (Note 1)	Advance Information Min	Advance Information Max	Unit
1	INCLK Period (= .5T)	Note 2	30	62.5	ns
2	INCLK High Time	Note 2	9	53.5	ns
3	INCLK Low Time	Note 2	9	53.5	ns
4	INCLK Rise Time	Note 2	0	4	ns
5	INCLK Fall Time	Note 2	0	4	ns
6	MEMCLK Delay from INCLK		0	10	ns
7	Synchronous Output Valid Delay from MEMCLK ↑	Note 3	1	10	ns
7a	Synchronous Output Valid Delay from MEMCLK ↑	Note 3	1	12	ns
7b	Synchronous Output Valid Delay from MEMCLK ↓		1	10	ns
8	Synchronous Output Disable Delay from MEMCLK ↑		1	10	ns
9	Synchronous Input Setup Time		6		ns
10	Synchronous Input Hold Time		0		ns
11	Asynchronous Pulse Width	Note 4	4T		ns
11a	Asynchronous Pulse Width	Note 4	Note 4		
12	MEMCLK High Time	Note 5	.5T–3	.5T+3	ns
13	MEMCLK Low Time	Note 5	.5T–3	.5T+3	ns
14	MEMCLK Rise Time	Note 5	0	4	ns
15	MEMCLK Fall Time	Note 5	0	4	ns
16	UCLK, VCLK Period	Note 2	30		ns
17	UCLK, VCLK High Time	Note 2	9		ns
18	UCLK, VCLK Low Time	Note 2	9		ns
19	UCLK, VCLK Rise time	Note 2	0	4	ns
20	UCLK, VCLK Fall Time	Note 2	0	4	ns
21	Synchronous Output Valid Delay from VCLK ↑↓	Note 6	1	15	ns
22	Input Setup Time to VCLK ↑↓	Notes 6, 7	6		ns
23	Input Hold Time to VCLK ↑↓	Notes 6, 7	0		ns
24	TCK Frequency			2	MHz

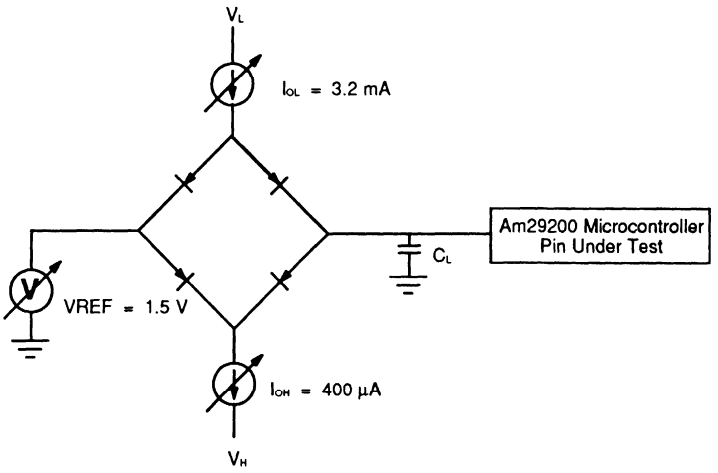
Notes:
1. All outputs driving 80 pF, measured at V_{OL} = 0.8 V and V_{OH} = 2.0 V. For higher capacitance, add 1 ns output delay per 20 pF loading up to 300 pF total.
2. INCLK, VCLK, and UCLK can be driven with TTL inputs. UCLK must be tied High if it is unused.
3. Parameter 7 applies to all outputs except PIO15–PIO0, STAT2–STAT0, $\overline{\text{PIACS5}}$–$\overline{\text{PIACS0}}$, and $\overline{\text{DACK1}}$–$\overline{\text{DACK0}}$. Parameter 7a applies to PIO15–PIO0, STAT2–STAT0, $\overline{\text{PIACS5}}$–$\overline{\text{PIACS0}}$, and $\overline{\text{DACK1}}$–$\overline{\text{DACK0}}$.
4. Parameter 11 applies to all asynchronous inputs except LSYNC and PSYNC. LSYNC and PSYNC minimum width is 2 bit times. A bit time is one period of the internal video clock, which is determined by dividing down VCLK.
5. MEMCLK can drive an external load of 100 pF.
6. Active VCLK edge depends on CLKI bit in Video Control Register.
7. LSYNC and PSYNC may be treated as synchronous signals by meeting the setup and hold times, though the synchronization delay still applies.

SWITCHING WAVEFORMS



Note: During AC testing, all inputs are driven at V_{IL} = 0.45 V, V_{IH} = 2.4 V.

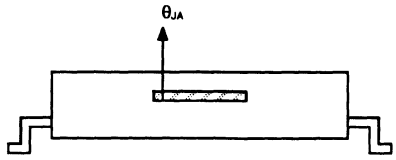
SWITCHING TEST CIRCUIT



C_L is guaranteed to 80 pF. For capacitive loading greater than 80 pF, add 1 ns output delay per 20 pF loading up to 300 pF total.

THERMAL CHARACTERISTICS

Plastic Quad Flat Pack (PQFP) Package



Thermal Resistance – °C/Watt

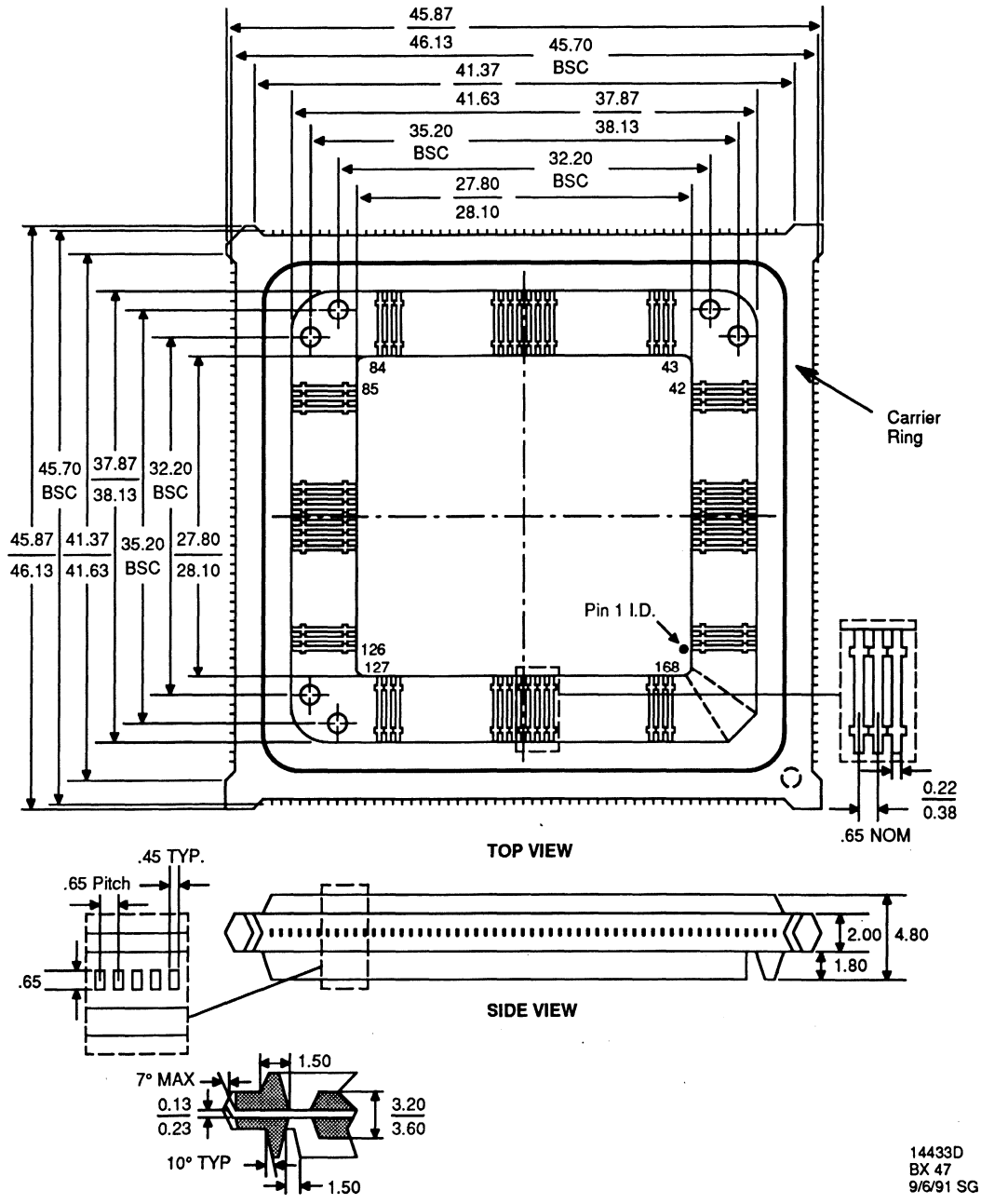
Advance Information

Parameter	°C/Watt
θ_{JA} Junction-to-Ambient	36

PHYSICAL DIMENSIONS

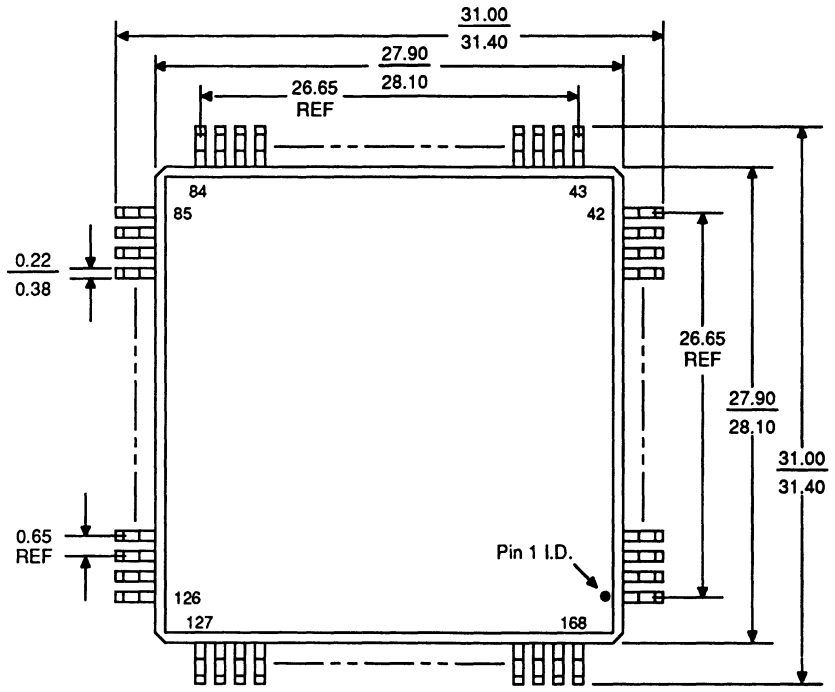
For reference only. All measurements are given in millimeters unless otherwise noted. BSC is an ANSI standard for Basic Space Centering.

PQR–168
(measured in millmeters)

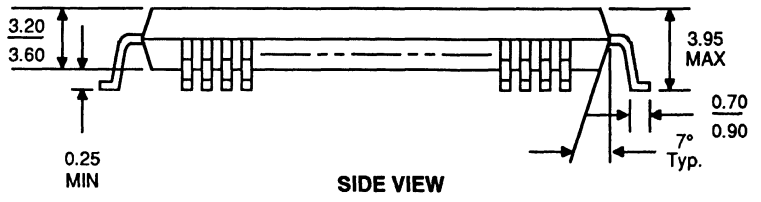


PHYSICAL DIMENSIONS (continued)

PQJ 168
(measured in millimeters)



TOP VIEW



SIDE VIEW

BU 43
9/9/91 SG

INDEX