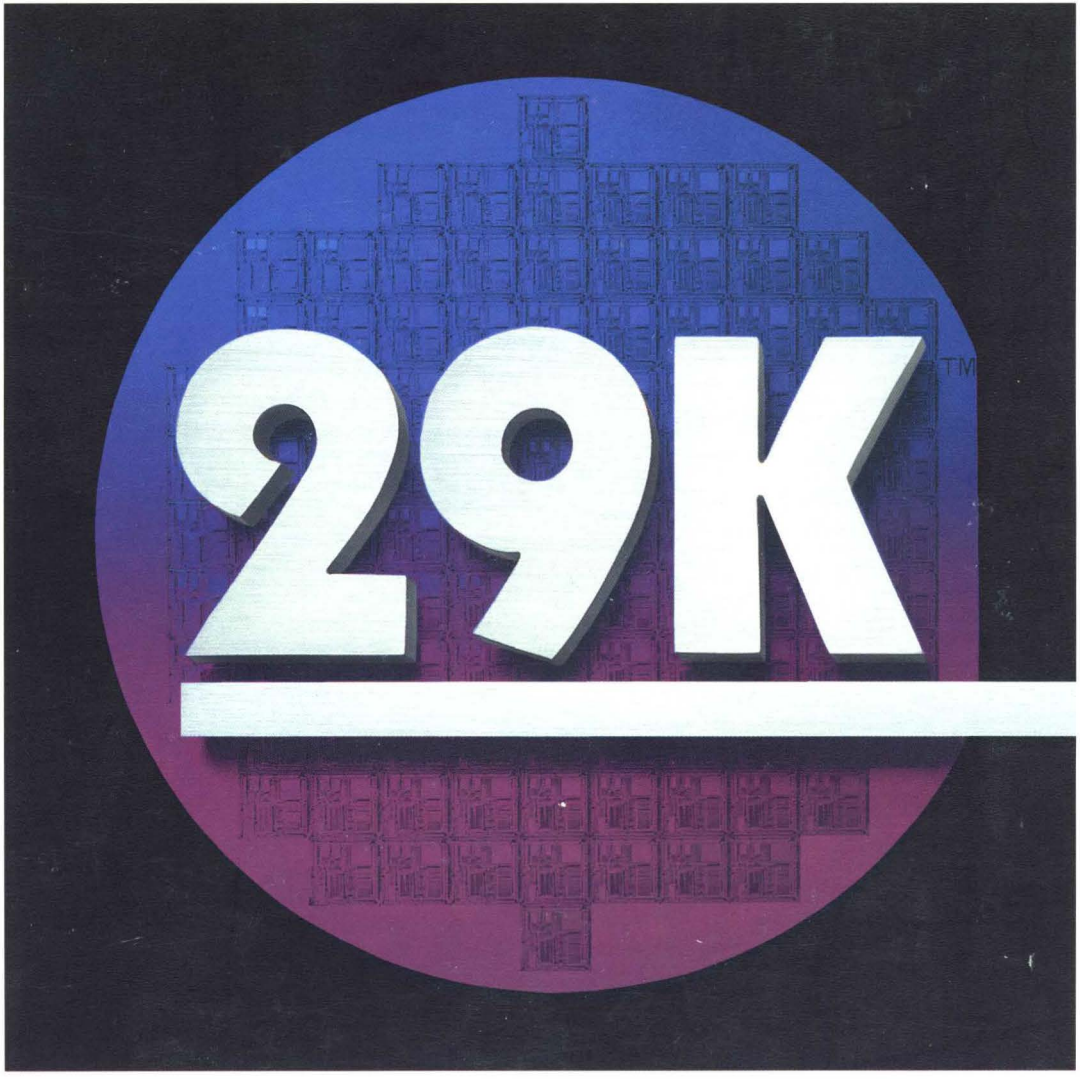
Am29240™, Am29245™, and Am29243™ RISC Microcontrollers

User's Manual and Data Sheet

Advanced Micro Devices



AMD's Marketing Communications Department specifies environmentally sound agricultural inks and recycled papers, making this book highly recyclable.

Am29240™, Am29245™, and Am29243™ RISC Microcontrollers

User's Manual and Data Sheet

Rev. 1, 1993

ADVANCED MICRO DEVICES

Trademarks

IF YOU HAVE QUESTIONS, WE'RE HERE TO HELP YOU.

Customer Service

AMD's customer service network includes U.S. offices, international offices, and a customer training center. Expert technical assistance is available from AMD's worldwide staff of field application engineers and factory support staff to answer 29K™ Family hardware and software development questions.

Hotline, eMail, and Bulletin Board Support

For answers to technical questions, AMD provides a toll-free number for direct access to our engineering support staff. For overseas customers, the easiest way to reach the engineering support staff with your questions is via fax with a short description of your question. AMD 29K Family customers also receive technical support through electronic mail. This worldwide service is available to 29K product users via the International UNIX eMail service. Also available is the AMD bulletin board service, which provides the latest 29K product information, including technical information and data on upcoming product releases.

Engineering Support Staff

(800) 292-9263 ext 2	toll-free for U.S.
0031-11-1163	toll-free for Japan
(512) 462-4118	direct dial worldwide
44-(0)256-811101	U.K. and Europe hotline
(512) 462-5031	fax
29k-support@amd.com	eMail

Bulletin Board

(800) 292-9263 ext 1	toll-free for U.S.
(512) 462-4898	worldwide and local for U.S.

Documentation and Literature

The 29K Family Customer Support Group responds quickly to information and literature requests. A simple phone call gets you free 29K Family information, such as data books, user's manuals, data sheets, application notes, the Fusion29K℠ Partner Solutions Catalog and Newsletter, and other literature. Internationally, contact your local AMD sales office for complete 29K Family literature.

Customer Support Group

(800) 292-9263 ext 3	toll-free for U.S.
(512) 462-5651	local for U.S.
(512) 462-5051	fax for U.S.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

INTRODUCTION AND OVERVIEW

THE Am29240 RISC MICROCONTROLLER SERIES

The Am29240 microcontroller series continues the 32-bit processor series initiated by the Am29200™ and the low-cost Am29205™ microcontrollers. The Am29240 microcontroller series extends the performance range of the RISC microcontroller family, employing submicron circuits to add on-chip caches and 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 Am29240 microcontroller series can achieve higher performance at lower cost than existing systems. The Am29240 microcontroller series is binary compatible with all other members of the 29K Family, further broadening the price/performance range of the 29K Family.

The Am29240 microcontroller series, which includes the Am29240, Am29245, and Am29243 microcontrollers, was designed expressly to meet the requirements of embedded applications such as laser printers, telecommunications, networking, graphics processing, mass storage, 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 set of development, debug, and benchmarking tools is critical for reducing product development time.
- A rational, easy upgrade path: The processor family must provide bus- and software-compatibility so processor upgrades are transparent to both hardware and software.

The Am29200 family of RISC microcontrollers is optimized for any embedded application requiring better-than-CISC performance at minimal system cost. With the introduction of the Am29240 microcontroller series, the RISC microcontroller family spans a performance range of 3–30 MIPS. The electronic components for most embedded systems amount to little more than an Am29240 microcontroller series device, ROM, DRAM, and electrical buffering.

PURPOSE OF THIS MANUAL

This manual describes the technical features, programming interface, on-chip peripherals, and complete instruction set of the Am29240 microcontroller series.

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 Am29240 microcontroller series.

USER'S MANUAL OVERVIEW

This manual contains information on the Am29240 microcontroller series and 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 twenty-one chapters:

- Chapter 1 introduces the **features and performance** aspects of the Am29240 microcontroller series.
- Chapter 2 describes the **programmer's model** of the Am29240 microcontroller series, 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 Am29240 microcontroller series.
- Chapter 7 describes the **memory-management unit**.
- Chapter 8 describes the operation and control of the **instruction cache**.
- Chapter 9 describes the operation and control of the **data cache**.
- Chapter 10 provides an overview of the processor's **system interfaces** and the system components that are integrated on-chip.
- Chapter 11 describes the **ROM interface**.
- Chapter 12 describes the **DRAM interface**.
- Chapter 13 describes the **peripheral interface adapter,** which is used for glueless attachment of a number of peripheral components.
- Chapter 14 describes the **DMA controller**.
- Chapter 15 describes the **programmable I/O port**.
- Chapter 16 describes the **parallel port**.
- Chapter 17 describes the **serial ports**.
- Chapter 18 describes the **video interface**.
- Chapter 19 provides a description of the **interrupt and trap mechanism**, including the operation of the on-chip interrupt controller.
- Chapter 20 describes the software and hardware facilities for **debugging and testing**.
- Chapter 21 provides a detailed description of the **instruction set**.

For those readers desiring only a **brief overview** of the Am29240 microcontroller series, Chapter 1 identifies the outstanding features of the processors. This chapter addresses the basic software and hardware concerns.

Chapters 2, 3, and 5 are recommended reading for both **hardware and software** developers.

For software architects and system programmers interested mainly in **software-related issues**, Chapters 4, 6, 7, and 19–21 provide the necessary information.

For hardware architects and systems hardware designers interested mainly in **hardware-related issues**, Chapters 7–18, Chapter 20, and Appendix D provide most of the required information. Chapters 5 and 21 also provide related information.

For users already familiar with the Am29200 microcontroller, Chapters 7 through 10 highlight the **enhancements** made in the Am29240 microcontroller series. Other enhancements are described in Chapters 11–20.

For users already familiar with other 29K Family processors, Chapters 7–18 describe the on-chip memory management unit, caches, peripherals, and system functions unique to the Am29240 microcontroller series.

AMD DOCUMENTATION

29K Family

ORDER NO.	DOCUMENT
10620	**Am29000™ and Am29005™ Microprocessors User's Manual and Data Sheet** Describes the Am29000™ and Am29005™ microprocessors' technical features, programming interface, and complete instruction set.
11426	**Fusion29K℠ Catalog** Provides information on more than 215 tools that speed a 29K Family embedded product to market. Includes products from over 115 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 and features new Fusion Partner solutions.
14779	**Am29050™ Microprocessor User's Manual** Describes the Am29050 microprocessor's technical features, programming interface, and complete instruction set.
15723	**Am29030™ and Am29035™ Microprocessors User's Manual and Data Sheet** Describes the Am29030 and Am29035 microprocessors' technical features, programming interface, and complete instruction set.
16362	**Am29200™ RISC Microcontroller User's Manual and Data Sheet** Describes the Am29200 microcontroller's technical features, programming interface, and complete instruction set.
17198	**Am29205™ RISC Microcontroller Data Sheet**
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.

10344 **29K RISC Design-Made-Easy℠ Solutions Brochure**
Presents an overview of the entire 29K Family of microprocessors and microcontrollers. Features development support products.

16693 **RISC Design-Made-Easy Applications Guide**
Presents topics on the 29K Family, including interfaces to integer multipliers, context switching, TLB handlers, benchmarking applications, byte-writable memories for three-bus microprocessors, host interface (HIF) version 2.0 specification, using the Am29000 microprocessor as a high-performance DMA controller, and writing interrupt handlers.

Development Tools

17704 **Am29200 and Am29205 RISC Microcontroller Brochure**
Reviews how the SA-29200 and SA-29205 demonstration boards and the SA-29200 expansion board use the Am29200 or Am29205 microcontroller to meet requirements for low-cost embedded applications. Includes additional support product and ordering information.

10287 **MiniMON29K™ Portable Debug Monitor Data Sheet**

10626 **XRAY29K™ Source-Level Debugger Data Sheet**

10957 **High C® 29K Development Toolkit 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 Catalog No. SH13144
1-800-CS-BOOKS
(fax) 714-821-4010

CHAPTER

1 FEATURES AND PERFORMANCE

The Am29240 microcontroller series greatly enhances the price/performance range of the Am29200 RISC microcontroller family. The Am29240 microcontroller series, which includes the Am29240, Am29245, and Am29243 microcontrollers, extends the performance of existing Am29200 and Am29205 microcontroller applications and provides the computational power for new applications to benefit from the low cost, low parts count, and quick time-to-market of a highly integrated processor family.

This chapter provides a general evaluation of the Am29240 microcontroller series to help the reader consider a particular application. The distinctive features of each microcontroller are compared in Table 1-1. A detailed technical description of the Am29240, Am29245, and Am29243 microcontrollers is contained in subsequent chapters. This chapter informally describes the microcontrollers, concentrating on features that distinguish the Am29240 microcontroller series 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

Table 1-1 Am29240 Microcontroller Series Feature Summary

Feature	Am29240 Microcontroller	Am29245 Microcontroller	Am29243 Microcontroller
Instruction Cache	4K	4K	4K
Data Cache	2K	—	2K
Parity Generation/Checking	—	—	Yes
Memory Management Unit (MMU)	1 TLB, 16-entry	1 TLB, 16-entry	2 TLBs, 32-entry
Internal Registers	192	192	192
DMA Channels (fly-by)	4	2	4
ROM Interface	8-, 16-, 32-bit	8-, 16-, 32-bit	8-, 16-, 32-bit
DRAM Interface	16-, 32-bit	16-, 32-bit	16-, 32-bit
Bidirectional Parallel Port	1	1	1
Serial Ports (UARTs)	2	1	2
Serializer/Deserializer (Video Interface)	Yes	Yes	—
Full- and Double-Speed Clock	Yes	—	Yes

1.1 DISTINCTIVE CHARACTERISTICS

The Am29240 microcontroller series significantly reduces system cost by integrating many system functions onto a single chip. A large on-chip instruction cache and, for the Am29240 and Am29243 microcontrollers, an on-chip data cache allow zero wait states for most processor accesses. The Am29240 and Am29243 microcontrollers also include a single-cycle integer multiply unit.

1.1.1 Am29240 Microcontroller

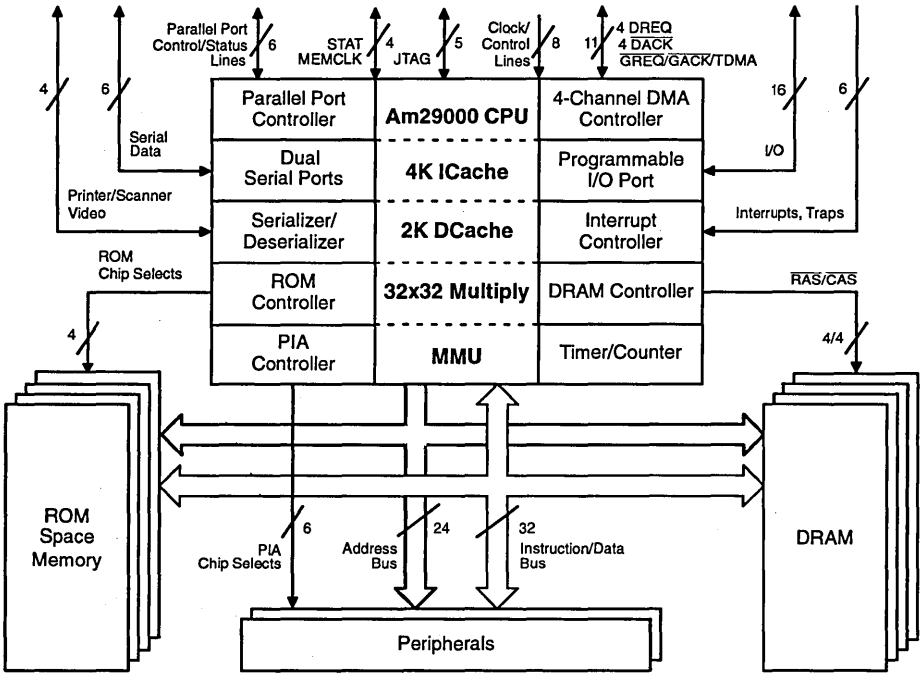
Figure 1-1 shows a block diagram of the Am29240 microcontroller. The following features are included in the Am29240 microcontroller:

- Completely integrated system for embedded applications
- Full 32-bit architecture
- CMOS technology/TTL-compatible
- 20-, 25-, and 33-MHz operational frequency
- 4-Kbyte, two-way set-associative instruction cache. The instruction cache supports fetch-through for best performance. All cache array elements are visible to software for testing and preload.
- 2-Kbyte, two-way set-associative data cache. The data cache performs wrap-around, burst-mode refill with load-through.
- 16-entry Memory Management Unit (MMU). The MMU maps pages that range in size from 1 Kbyte to 16 Mbyte in powers of 4.
- Full-and double-speed internal clock (turbo mode)
- 32-by-32 multiplier. The multiplier performs single-cycle 32-bit integer multiplies. 64-bit results can be produced in two cycles.
- Price/performance flexibility. Support for 8-, 16-, and 32-bit memory systems
- 4-Gbyte virtual address space, 304-Mbyte physical space implemented
- 192 general-purpose registers
- Three-address instruction architecture
- Fully pipelined
- Glueless system interfaces with on-chip wait-state control
- ROM controller, supporting four individual banks of ROM or SRAM, each with its own timing characteristics. 8-, 16-, and 32-bit ROM interfaces are supported.
- DRAM controller, supporting four separate banks of dynamic memory. 16- and 32-bit DRAM interfaces are supported.
- Single-cycle ROM burst-mode and DRAM page-mode access
- Four-channel DMA controller. Each channel is double-buffered to relax constraints on reload time.
- Six-port Peripheral Interface Adapter. The PIA allows for additional system features implemented by external peripheral chips.
- 16-line programmable I/O port
- Bidirectional video interface (serializer/deserializer)
- Two serial ports (UARTs)

- Bidirectional parallel port controller 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
- Optional 3.3-volt or 5-volt operation

Before using the Am29240 microcontroller product, the user should prepare it by setting the PCE field in the DRAM Control Register (Section 12.1.1) to 0.

Figure 1-1 Am29240 Microcontroller Block Diagram



1.1.2 Am29245 Microcontroller

A block diagram of the Am29245 microcontroller is shown in Figure 1-2. Designed as a cost-reduced choice for the Am29240 microcontroller series, the Am29245 microcontroller is similar to the Am29240 microcontroller, with the following differences:

- 16-MHz operational frequency
- Two-channel DMA controller
- One serial port (UART)
- Full-speed internal clock (no double-speed)
- The Am29245 microcontroller does not include a data cache.
- The Am29245 microcontroller does not include the 32-by-32 multiplier.

Before using the Am29245 microcontroller product, the user should prepare it by setting the following fields and signals. In the Configuration Register (Section 2.9.1), the TBO field should be set to 0 and the DD field should be set to 1. In the DRAM Control Register (Section 12.1.1), the PCE field should be set to 0. The RXDB signal (Section 10.1.9) should be set to ground or Vcc.

Figure 1-2 Am29245 Microcontroller Block Diagram

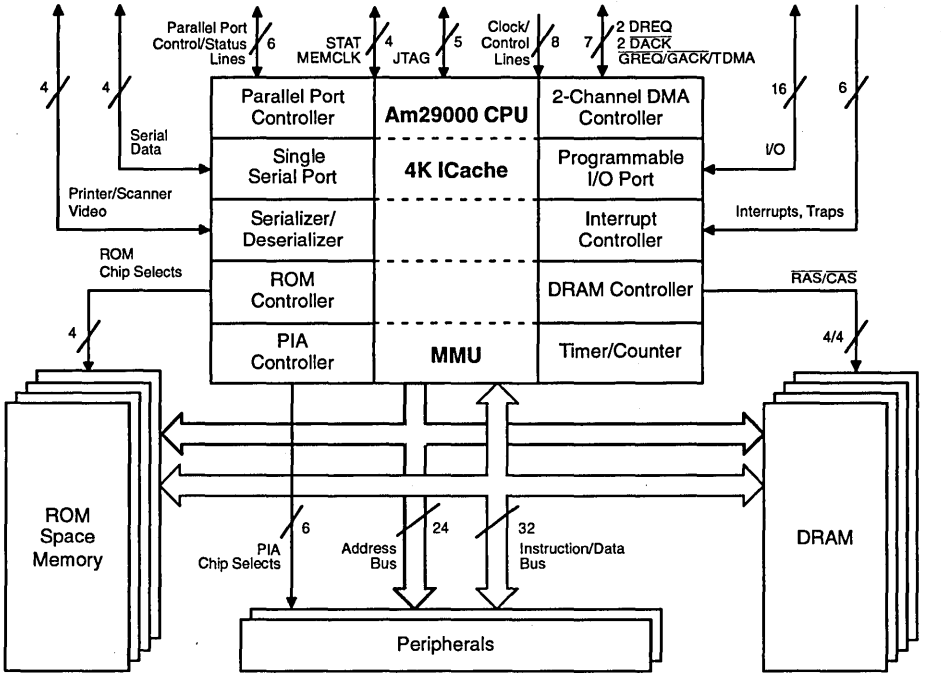
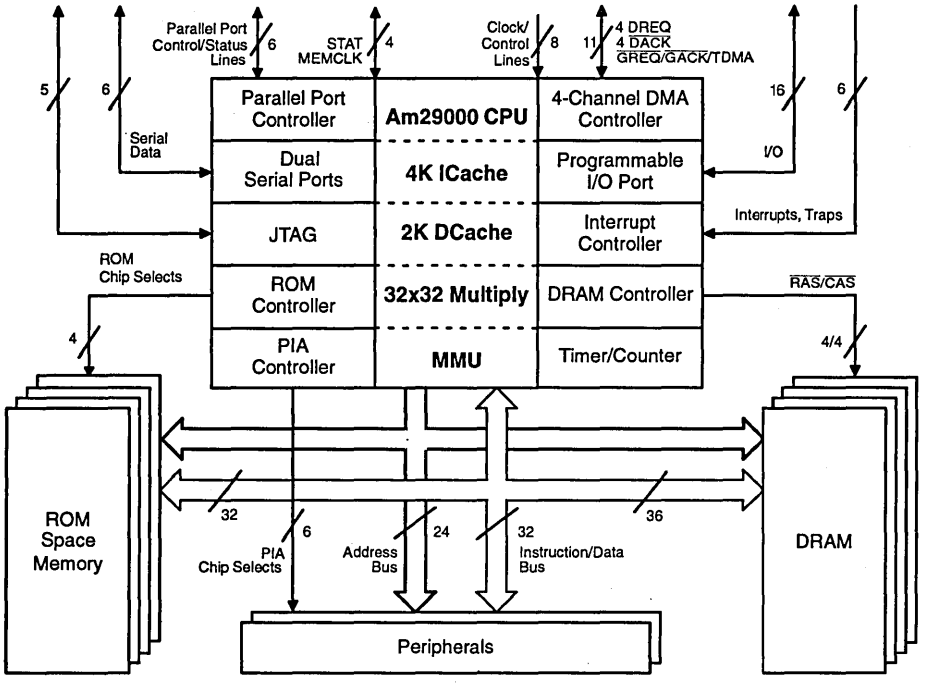


Figure 1-3 Am29243 Microcontroller Block Diagram



1.1.3 Am29243 Microcontroller

Figure 1-3 shows a block diagram of the Am29243 microcontroller. The Am29243 data microcontroller is similar to the Am29240 microcontroller, with the following differences:

- 32-entry MMU with dual Translation Look-Aside Buffers (TLBs). The MMU has two TLBs, each with 16 entries, mapping pages that range in size from 1 Kbyte to 16 Mbyte in powers of 4. The page size of each TLB can be set independently, so it is possible to mix different page sizes in one system. For example, an application may use small pages for code and large pages for frame buffers and/or shared libraries.
- DRAM Parity. The processor can be configured to generate and check even or odd parity on DRAM accesses.
- The video interface (serializer/deserializer) is not supported on the Am29243 microcontroller. The associated I/O pins are reserved for further integration of the data microcontroller line.

Before using the Am29243 microcontroller product, the user should prepare it by setting the LSYNC and VCLK signals (Section 10.1.10) to ground or Vcc.

1.2 KEY FEATURES AND BENEFITS

The Am29240 microcontroller series consists of highly integrated, 32-bit embedded processors implemented in complementary metal-oxide semiconductor (CMOS) technology. They are targeted primarily at office automation, telecommunications, networking, 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 Am29240 microcontroller series extends the line of RISC microcontrollers based on the 29K architecture, providing performance upgrades to the Am29205 and Am29200 microcontrollers. The RISC microcontroller product line allows users to benefit from the very high performance of the 29K architecture, while also capitalizing on the very low system cost made possible by the integration of processor and peripherals.

The Am29240 microcontroller series also expands the price/performance range of systems that can be built with the 29K Family. The Am29240 microcontroller series is fully software compatible with the Am29000, Am29005, Am29030™, Am29035™, and Am29050™ microprocessors, as well as the Am29200 and Am29205 microcontrollers. It can be used in existing 29K Family microcontroller applications without software modifications.

1.2.1 On-Chip Caches (Chapters 8 and 9)

The Am29240 microcontroller series incorporates a 4-Kbyte, two-way instruction cache that supplies most processor instructions without wait states at the processor frequency. For best performance, the instruction cache supports critical-word-first reloading with fetch-through, so that the processor receives the required instruction and the pipeline restarts with minimum delay. The instruction cache has a valid bit per word to minimize the reload overhead. All cache array elements are visible to software for testing and preload.

The Am29240 and Am29243 microcontrollers incorporate a 2-Kbyte, two-way set-associative data cache. The data cache appears in the execute stage of the processor pipeline, so that loaded data is available immediately to the next instruction. This provides the maximum performance for loads without requiring load scheduling. The data cache performs critical-word-first, wrap-around, burst-mode refill with load-through. This minimizes the time the processor waits on external data as well as minimizing the reload time. The data cache uses a write-through policy with a two-entry write buffer. Byte, half-word, and word reads and writes are supported. All cache array elements are visible to software for testing and preload.

1.2.2 Single-Cycle Multiplier

The Am29240 and Am29243 microcontrollers incorporate a full combinatorial multiplier that accepts two 32-bit input operands and produces a 32-bit result in a single cycle. The multiplier can produce a 64-bit result in two cycles. The multiplier permits maximum performance without requiring instruction scheduling, since the latency of the multiply is the same as the latency of other integer operations. High-performance multiplication benefits imaging, signal processing, and state modeling applications.

1.2.3 Complete Set of Common System Peripherals

The Am29240 microcontroller series 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, two serial ports, and an interrupt controller. A video interface is also included in the Am29240 and Am29245 microcontrollers for printer, scanner, and other imaging applications. These facilities allow many simple systems to be built using only the Am29240 microcontroller series, external ROM, and/or DRAM memory.

1.2.3.1 ROM Controller (Chapter 11)

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

1.2.3.2 DRAM Controller (Chapter 12)

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 can appear as a contiguous memory area of up to 64 Mbyte in size. To further enhance the performance, the DRAM controller supports two-cycle accesses with single-cycle page-mode and burst-mode accesses.

1.2.3.3 Peripheral Interface Adapter (Chapter 13)

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.

1.2.3.4 DMA Controller (Chapter 14)

The DMA controller provides up to four channels for transferring data between the DRAM and internal or external peripherals. The DMA channels are double buffered to relax constraints on reload time.

1.2.3.5 I/O Port (Chapter 15)

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.

1.2.3.6 Parallel Port (Chapter 16)

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

1.2.3.7 Serial Port (Chapter 17)

The serial port implements up to two full-duplex UARTs.

1.2.3.8 Serializer/Deserializer (Chapter 18)

The serializer/deserializer (video interface) permits direct connection to a number of laser marking engines, video displays, or raster input devices such as scanners.

1.2.3.9 Interrupt Controller (Section 19.8)

The interrupt controller generates and reports the status of interrupts caused by on-chip peripherals.

1.2.4 Wide Range of Price/Performance Points

To reduce design costs and time-to-market, the product designer can use the Am29200 microcontroller family and one basic system design 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 Am29240 RISC microcontroller series supports this capability through various combinations of on-chip caches, programmable memory widths, programmable wait states, burst-mode and page-mode access support, bus compatibility, and 29K Family software compatibility. A system can be upgraded without hardware and software redesign using various memory architectures.

Within the Am29240 microcontroller series, the external interfaces operate at frequencies in the range of 16 to 25 MHz, and the processor operates at frequencies in the range of 16 to 33 MHz. The internal processor core can operate either at the interface frequency or twice this frequency (turbo mode). For example, the processor can operate at 33 MHz while the interface operates at 16.5 MHz.

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, Am29205, Am29240, Am29245, and Am29243 microcontrollers allow users to address an extremely wide range of price/performance points, with higher performance and lower cost than existing designs based on CISC microprocessors.

1.2.5 Glueless System Interfaces

The Am29240 microcontroller series 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.

1.2.6 Bus- and Software-Compatibility

Compatibility within a processor family is critical for achieving a rational, easy upgrade path. The Am29240 processors are all members of a bus-compatible series of RISC microcontrollers. All members of this family, the Am29205, Am29200, Am29240, Am29245, and Am29243 microcontrollers, allow improvements in price, performance, and system capabilities without requiring that users redesign their system hardware or software. Bus compatibility ensures a convenient upgrade path for future systems.

The Am29240 microcontroller series is available in a 196-pin plastic quad flat-pack (PQFP) package. The Am29240 microcontroller series is signal-compatible with the Am29205 and the Am29200 microcontrollers.

Moreover, the Am29240 microcontroller series is binary compatible with existing RISC microcontrollers and other members of the 29K Family (the Am29000, Am29005, Am29030, Am29035, and Am29050 microprocessors, as well as the Am29205 and Am29200 microcontrollers). The Am29240 microcontroller series

provides a migration path to low-cost, high-performance, highly integrated systems from other 29K Family members, without requiring expensive rewrites of application software.

1.2.7 Complete Development and Support Environment

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

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

- High C® 29K™ optimizing C compiler with assembler, linker, ANSI library functions, and 29K architectural simulator
- XRAY29K™ source-level debugger
- MiniMON29K™ debug monitor
- A complete family of demonstration and development boards

In addition, Advanced Micro Devices has developed a standard host interface (HIF) specification for operating system services, the Universal Debug Interface (UDI) for seamless connection of debuggers to ICEs and target hardware, 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 Am29240 microcontroller series 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 very low cost. This section describes the features of the Am29240 microcontroller series from the point of view of system performance.

1.3.1 Instruction Timing (Section 2.1)

The Am29240 microcontroller series 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 Am29240 microcontroller series by overlapping them with instruction execution, by taking advantage of pipelining, by an on-chip data cache, 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 On-Chip Instruction and Data Caches (Chapters 8 and 9)

On-chip instruction and data caches satisfy most processor fetches without wait states, even when the processor operates at twice the system frequency. The caches are pipelined for best performance. The reload policies minimize the amount of time spent waiting for reload, while optimizing the benefit of locality of reference.

1.3.4 Burst-Mode and Page-Mode Memories (Sections 11.2.4, 12.2.7)

The Am29240 microcontroller series directly supports burst-mode memories. 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 processor can also 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.

1.3.5 Instruction Set Overview (Chapter 2)

All 29K Family members employ a three-address instruction set architecture. The compiler or assembly-language programmer is given complete freedom to allocate register usage. There are 192 general-purpose registers, allowing the retention of intermediate calculations and avoiding needless data destruction. Instruction operands may be contained in any of the general-purpose registers, and the results may be stored into any of the general-purpose registers.

The Am29240 microcontroller series instruction set contains 117 instructions that 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.

1.3.6 Data Formats (Chapter 3)

The Am29240 microcontroller series 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 most significant bit 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 Am29240 microcontroller series.

1.3.7 Protection (Chapter 6)

The Am29240 microcontroller series offers two mutually exclusive modes of execution, the user and supervisor modes, that 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.

1.3.8 Memory Management Unit (Chapter 7)

The Am29240 microcontroller series provides a memory-management unit (MMU) for translating virtual addresses into physical addresses. The page size for translation ranges from 1 Kbyte to 16 Mbyte in powers of four. The Am29245 and Am29240 microcontrollers each have a single, 16-entry TLB. The Am29243 microcontroller has dual 16-entry TLBs, each capable of mapping pages of different size.

1.3.9 Interrupts and Traps (Chapter 19)

When an Am29240, Am29245, or Am29243 microcontroller 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 that 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 that 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 20)

The Am29240 microcontroller series 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 several additional features to assist system debugging and testing:

- 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.
- A Traceable Cache feature permits a hardware-development system to track accesses to the on-chip caches, permitting a high level of visibility into processor operation.

- 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 processor and 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 Am29240 microcontroller series. 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 Am29240 microcontroller series 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. Some arithmetic instructions are not implemented directly in hardware, but are implemented by a virtual arithmetic (software) 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 Am29240 microcontroller series and provides a brief summary of instruction operations. A detailed instruction specification is contained in Chapter 21. Section 21.1 describes the nomenclature used here.

If the processor attempts to execute an unimplemented instruction, 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 Am29245 microcontroller, the integer multiply and divide instructions cause traps to routines that perform the operations. In the Am29240 and Am29243 microcontrollers, the integer multiplication instructions are performed by hardware and only the integer divide instructions cause traps.

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.

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)

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 19.2).

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.

Table 2-3 Logical Instructions

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

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.

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)

2.1.5 Data Movement

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

Table 2-5 Data Movement Instructions

Mnemonic	Operation Description
LOAD	DEST ← EXTERNAL WORD [SRCB]
LOADL	DEST ← EXTERNAL WORD [SRCB] (bypasses/invalidates data cache)
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 (bypasses/invalidates data cache)
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 ← 0I16
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 ← 0I16
CONSTH	Replace high-order half-word of SRCA by I16
CONSTN	DEST ← 1I16

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

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)]

between single-precision, double-precision, and integer number representations. In the Am29240 series 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.

Table 2-8 Branch Instructions

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

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

2.1.9 Miscellaneous

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

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

2.1.10 Reserved Instructions

Sixteen Am29240 microcontroller series operation codes are reserved for instruction emulation. Each instruction causes a trap and sets the indirect pointers IPC, IPA, and IPB. The relevant operation codes and the corresponding trap vectors are as follows:

Table 2-10 Reserved Instructions

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

2.2 REGISTER MODEL

The Am29240 microcontroller series 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 Am29240 microcontroller series 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.

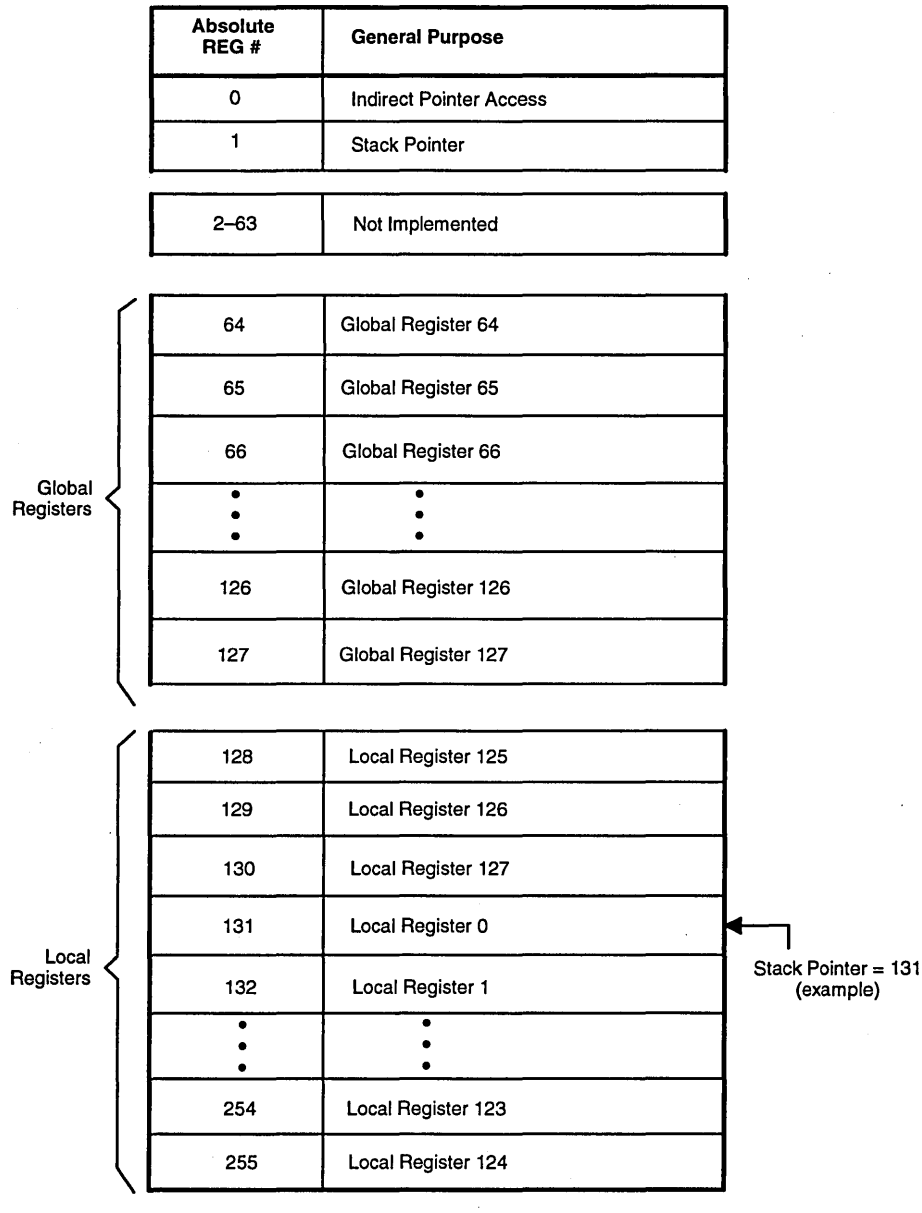
2.2.1 General-Purpose Registers

The Am29240 microcontroller series 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:

- 32-bit addresses
- 32-bit signed or unsigned integers
- 32-bit branch-target addresses
- 32-bit logical bit strings
- 8-bit signed or unsigned characters

Figure 2-1 General-Purpose Register Organization



- 16-bit signed or unsigned integers
- Word-length Booleans
- Single-precision floating-point numbers
- 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.

Instructions for the Am29240 microcontroller series 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:

- Register number, 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.
- Global-register number, a software-level number for a global register. Global-register numbers range from 0 to 127.
- Local-register number, a software-level number for a local register. Local-register numbers range from 0 to 127.
- Absolute-register number, 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 6.2.1).

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

2.2.1.3 Local Registers

When the most significant bit of a register number is 1, a local register is selected. The seven least significant bits of the register number give the local-register number. For local registers, the absolute-register number is obtained by adding the local-register

number to bits 8–2 of the Stack Pointer and truncating the result to seven bits; the most significant bit of the original register number is unchanged (i.e., it remains a 1).

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

2.2.1.4 Local-Register Stack Pointer

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

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

2.2.2 Special-Purpose Registers

The Am29240 microcontroller series contains 28 special-purpose registers. The organization of the special-purpose registers is shown in Figure 2-2.

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

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

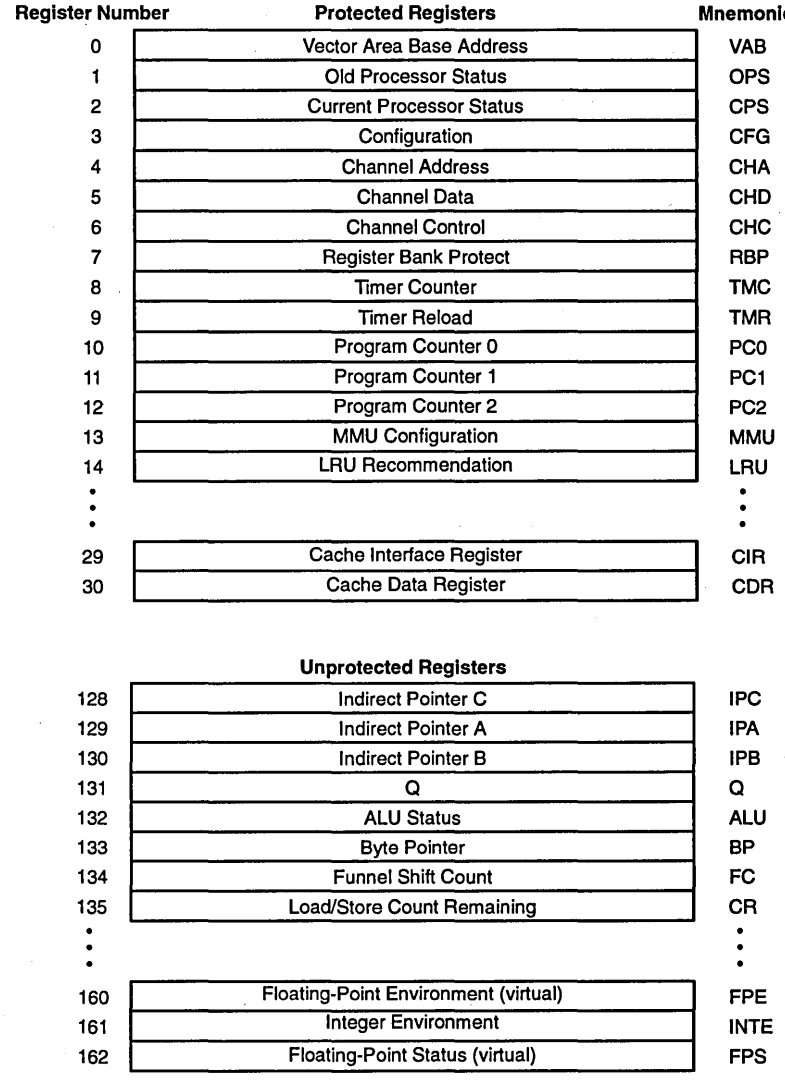
The special-purpose registers are accessed by explicit data movement only. Instructions that move data to or from a special-purpose register specify the special-purpose register by an 8-bit field containing a special-purpose register number. Register numbers are specified directly by instructions.

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

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

The Floating-Point Environment Register and Floating-Point Status Register are not implemented in processor hardware. These registers are implemented via the virtual arithmetic software provided on the Am29240 microcontroller series (see Section 2.8).

Figure 2-2 Special-Purpose Registers



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 trap. 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.

For the Am29240 and Am29243 microcontrollers, the indirect pointers are set by the MTSR, MTSRIM, SETIP, and EMULATE instructions, and by floating-point instructions, DIVIDE and DIVIDU. For the Am29245 microcontroller, the floating-point instructions, MULTIPLY, MULTM, MULTIPLU, MULTMU, also set the indirect pointers.

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 5.6.

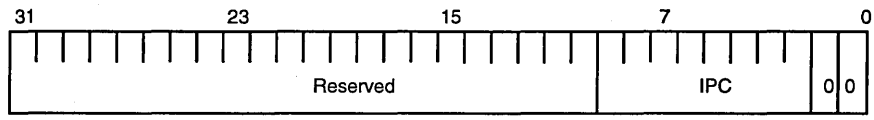
For the remaining instructions, all three indirect pointers are set simultaneously with the absolute-register numbers derived from the register numbers specified by the instruction. For any local registers selected by the instruction, the Stack-Pointer addition is applied to the register numbers before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

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

2.3.1 Indirect Pointer C Register (IPC, Register 128)

This unprotected special-purpose register (Figure 2-3) provides the RC-operand register number (see Section 21.3) 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

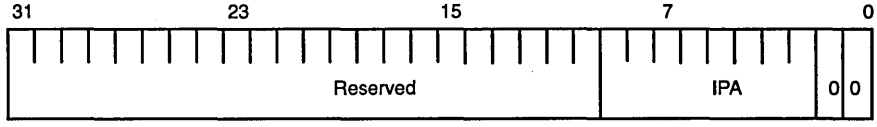
Bits 9–2: Indirect Pointer C (IPC)—The 8-bit IPC field contains an absolute-register number for a general-purpose register. This number directly selects a register. (Stack-Pointer addition is not performed in the case of local registers.)

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

2.3.2 Indirect Pointer A Register (IPA, Register 129)

This unprotected special-purpose register (Figure 2-4) provides the RA-operand register number (see Section 21.3) 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

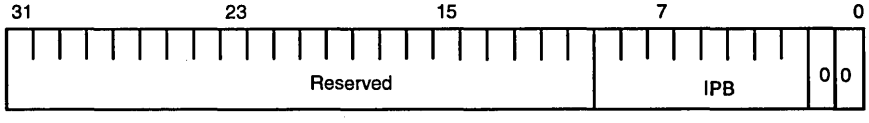
Bits 9–2: Indirect Pointer A (IPA)—The 8-bit IPA field contains an absolute-register number for either a general-purpose register or a local register. This number directly selects a register. (Stack-Pointer addition is not performed in the case of local registers.)

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

2.3.3 Indirect Pointer B Register (IPB, Register 130)

This unprotected special-purpose register (Figure 2-5) provides the RB-operand register number (see Section 21.3) 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.

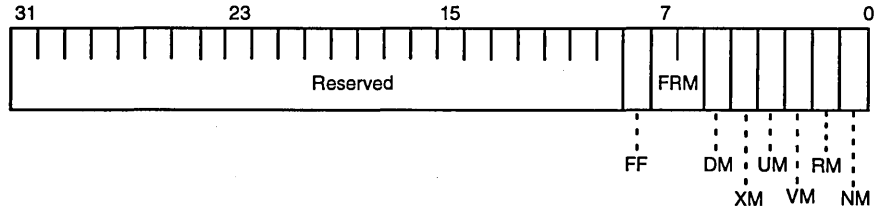
2.4 INSTRUCTION ENVIRONMENT

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

2.4.1 Floating-Point Environment Register (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 software.

Figure 2-6 Floating-Point Environment Register



Bits 31–9: Reserved

Bit 8: Fast Floating-Point 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.

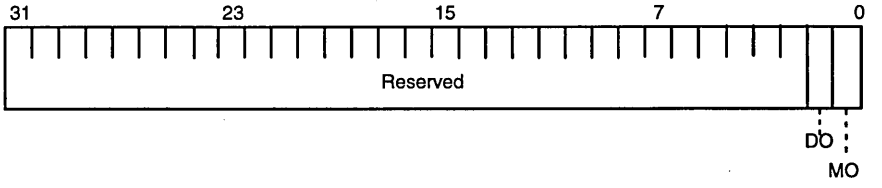
Bit 1: Floating-Point Reserved Operand Mask (RM)—If the RM bit is 0, a Floating-Point Exception trap occurs when one or more input operands to a floating-point operation is a reserved value, or when the result of a floating-point operation is a reserved value. If the RM bit is 1, a Floating-Point Exception 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 Register (INTE, Register 161)

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

Figure 2-7 Integer Environment Register



Bits 31–2: Reserved

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

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

Bit 0: Integer Multiplication Overflow Exception Mask (MO)—If the MO bit is 0, an Out of Range trap occurs when overflow of a signed or unsigned 32-bit result occurs during a MULTIPLY or MULTIPLU instruction, respectively. If the MO bit is 1, an Out of Range trap does not occur for overflow during integer multiply operations. Because 64-bit results cannot overflow, this bit should be set to 1 when obtaining a 64-bit result for multiplication, to avoid spurious out-of-range traps.

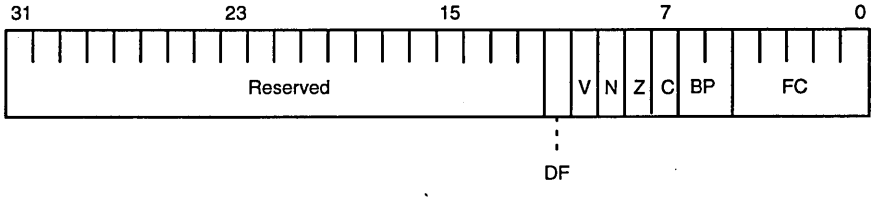
2.5 STATUS RESULTS OF INSTRUCTIONS

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

2.5.1 ALU Status Register (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 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 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 value 11.

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 processor 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.

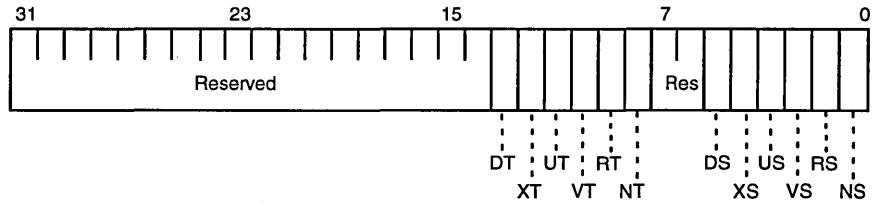
2.5.5 Floating-Point Status Register (FPS, Register 162)

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 software.

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) that report the status of an operation for which a Floating-Point Exception trap is taken. These bits are updated only by an operation that 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 Register



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.

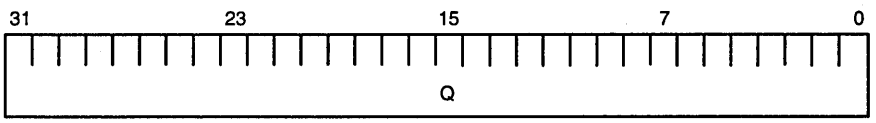
2.6 INTEGER MULTIPLICATION AND DIVISION

The Am29245 microcontroller does not directly support the instructions MULTIPLU, MULTMU, MULTIPLY, MULTM, DIVIDE, and DIVIDU. The Am29240 and Am29243 microcontrollers do not directly support the instructions DIVIDE and DIVIDU. The processor is capable of performing these instructions as a sequence of multiply and/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 Register (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 (Am29240 and Am29243 Microcontrollers)

The Am29240 and Am29243 microcontrollers directly execute the integer multiplication instruction MULTIPLY, MULTIPLU, MULTM, and MULTMU. These processors also implement the MUL, MULU, and MULL instructions for compatibility, but new code generated for the Am29240 and Am29243 microcontrollers can take advantage of the faster integer multiply instructions.

The MULTIPLY and MULTIPLU instructions multiply two 32-bit integers, giving a 32-bit result. MULTIPLY is used for signed integers, and MULTIPLU is used for unsigned integers. Overflow of the 32-bit result is detected when the Integer Multiplication Overflow Exception Mask (MO) bit of the Integer Environment Register is 0. When the MO bit is 0, the MULTIPLY and MULTIPLU operations cause an Out of Range trap upon overflow of a 32-bit signed or unsigned result, respectively.

In general, multiplying 32-bit integers produces a 64-bit result. The most significant 32 bits of a signed or unsigned result are generated by the MULTM and MULTMU instructions, respectively. To obtain a full 64-bit result, a MULTIPLY or MULTIPLU instruction is followed by a MULTM or MULTMU instruction:

```
; 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

        multiply  gr97, lr2, lr3        ; get LSBs
        multm     gr96, lr2, lr3        ; get MSBs

; 32 bit * 32 bit → 64 bit unsigned multiply
; Input: multiplicand in lr2, multiplier in lr3
; Output: result most significant word in gr96, result least significant word in gr97

        multiplu  gr97, lr2, lr3        ; get LSBs
        multmu    gr96, lr2, lr3        ; get MSBs
```

The operation producing the most significant bits of the 64-bit result takes one cycle of latency, so producing a full 64-bit result takes two cycles. Note that the MO bit should be 1 to disable the detection of overflow when obtaining a 64-bit result; 64-bit results cannot overflow.

2.6.3 Multiplication (Am29245 Microcontroller Only)

The Am29245 microcontroller 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. Compiler optimizations use this technique automatically.

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 16-bit 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
```

2.6.4 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 lr4
; 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 INSTRUCTIONS FOR...

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 predefined 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 that 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.

2.7.6 NO-OPs

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

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

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

2.8 VIRTUAL ARITHMETIC PROCESSOR

In order to be object-code compatible with present and future implementations of the 29K Family of microcontrollers, the Am29240 microcontroller series provides a virtual arithmetic software. A virtual implementation is the means by which a processor appears to perform functions that it does not actually perform. In the case of the Am29240 series processor's virtual arithmetic software, 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 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 Am29240 microcontroller series, the Floating-Point, CLASS, CONVERT, DIVIDE, DIVIDU, and SQRT instructions cause traps. In the Am29245 microcontroller, the MULTIPLY, MULTM, MULTIPLU, and MULTMU instructions also cause traps because the Am29245 microcontroller does not incorporate a hardware multiplier. 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) 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.

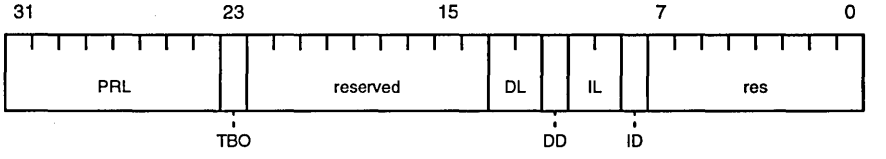
2.9 PROCESSOR INITIALIZATION

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 Register (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 initial value of the PRL field for the Am29240 microcontroller is 60, hexadecimal.

Bit 23: Turbo Mode (TBO)—The TBO bit determines whether or not the processor is clocked at the INCLK frequency (twice the MEMCLK frequency) or at the MEMCLK frequency. After a processor reset, the processor core is clocked at the MEMCLK frequency. If the TBO bit is written with 1, the processor converts its clocks to the INCLK frequency, doubling the maximum achievable performance. Once TBO has been written with 1, it no longer has an effect on processor clocking and may be written with any value; processor clocking does not change until the next reset. All external interface signals operate relative to MEMCLK. MEMCLK must be an output for the TBO bit to double the processor frequency. If MEMCLK is an input, setting the TBO bit has no effect on the processor frequency. The TBO bit must be set to 0 for the Am29245 microcontroller.

Bits 22–14: Reserved

Bit 13–12: Data Cache Lock (DL)—This field controls the locking of all or a portion of the data cache. When a cache block is locked, it is not invalidated and it is not replaced if it is valid. It can be allocated if it is invalid. If a block is not locked, replacement and invalidation occur normally. The DL field controls cache locking as follows:

DL	Effect on Cache
00	No blocks locked
01	Entire cache locked
10	Blocks in column 0 locked
11	Reserved

Bit 11: Data Cache Disable (DD)—If the DD bit is 1, the data cache is disabled and the data cache is not used to satisfy any processor access. Data that is fetched externally is not placed into the cache. However, the cache may be invalidated by an INV or IRETINV instruction. If the DD bit is 0, the data cache is enabled and is involved in the access of cacheable data. The DD bit must be set to 1 for the Am29245 microcontroller.

Bit 10–9: Instruction Cache Lock (IL)—This field controls the locking of all or a portion of the instruction cache. When a cache block is locked, it is not invalidated and it is not replaced if it is valid. It can be allocated if it is invalid. If a block is not locked, replacement and invalidation occur normally. The IL field controls cache locking as follows:

IL	Effect on Cache
00	No blocks locked
01	Entire cache locked
10	Blocks in column 0 locked
11	Reserved

Bit 8: Instruction Cache Disable (ID)—If the ID bit is 1, the instruction cache is disabled, and the instruction cache is not used to satisfy any processor instruction fetch. Also, when the cache is disabled, fetched instructions are not stored into the cache. However, the cache may be invalidated by an INV or IRETINV instruction. If the ID bit is 0, the instruction cache is enabled and the instruction cache satisfies all instruction fetches for which it contains the appropriate instruction.

Bits 5–0: 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.

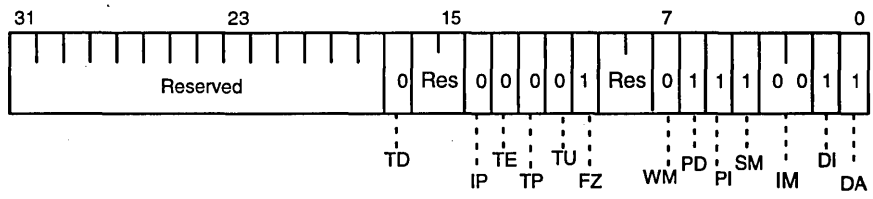
The Reset mode can be entered at any point during operation. If the $\overline{\text{RESET}}$ input is asserted at the time power is first applied to the processor, the processor enters the Reset mode only after four cycles have occurred on the MEMCLK pin.

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 (see Section 19.1.1) is set as shown in Figure 2-12.

Figure 2-12 Current Processor Status Register In Reset Mode



5. The Configuration Register is set as shown in Figure 2-13.
6. The Contents Valid (CV) bit of the Channel Control Register is reset.
7. All valid bits are reset in the instruction and data caches.

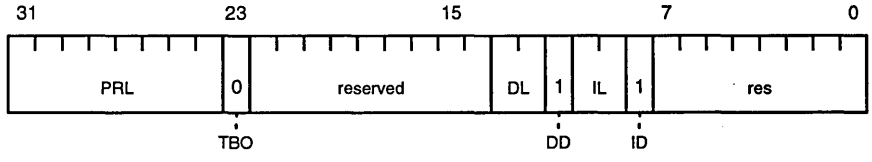
Except as previously noted, the contents of all general-purpose registers and special-purpose registers are undefined.

The Reset mode is exited when the $\overline{\text{RESET}}$ input is deasserted. Either four or five cycles after $\overline{\text{RESET}}$ is deasserted (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 11.1.3).

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, 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 left undefined 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(s) is (are) 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.

Figure 2-13 Configuration Register in Reset Mode



CHAPTER

3 DATA FORMATS AND HANDLING

This section describes the various data types supported by the Am29240 microcontroller series and the mechanisms for accessing data in external devices and memories. The Am29240 microcontroller series includes provisions for the external access of words, bytes, half-words, unaligned words, and unaligned half-words, as described in this section.

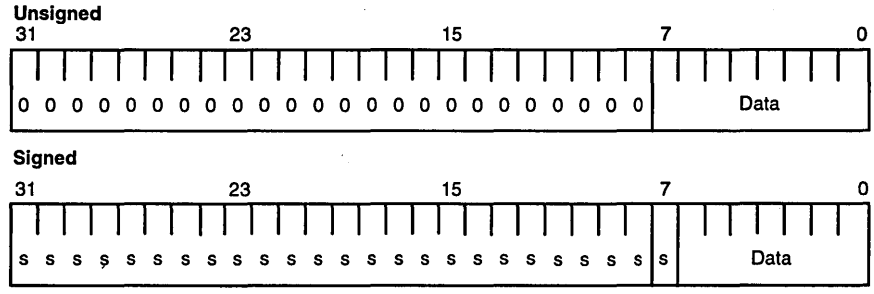
3.1 INTEGER DATA TYPES

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. The data format for the Am29240 microcontroller series is big endian only.

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.

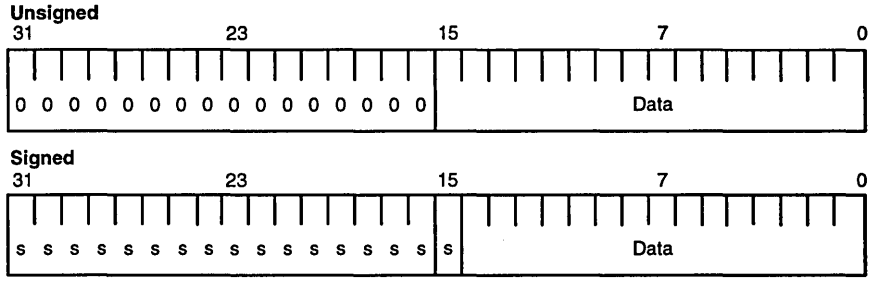
The Insert Byte (INBYTE) instruction replaces an arbitrary byte-aligned character in a destination word with the low-order character of a source word. For the INBYTE instruction, the source operand can be a character constant specified by the instruction.

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.

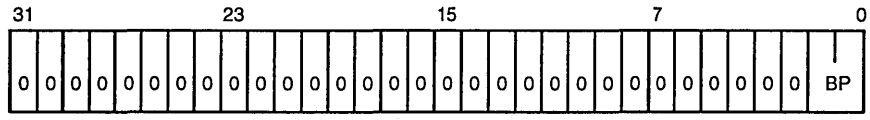
The Extract Half-Word, Sign-Extended (EXHWS) instruction is similar to the EXHW instruction, except that it sign-extends the half-word in the destination word (i.e., it replaces the most significant 16 bits of the destination word with the most significant bit of the source half-word).

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 Register (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.

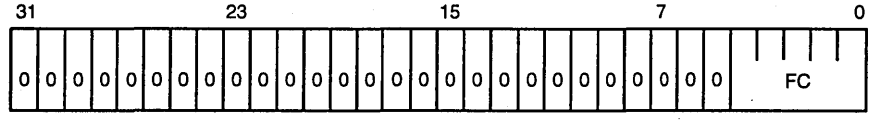
3.1.4 Bit Strings

Graphics and imaging applications often require that a data region be collectively shifted by a specific number of bits. The Am29240 microcontroller series 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 Register (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 points 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 ensure 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 0. 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 1. 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 Am29240 microcontroller series 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 Am29240 microcontroller series.

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

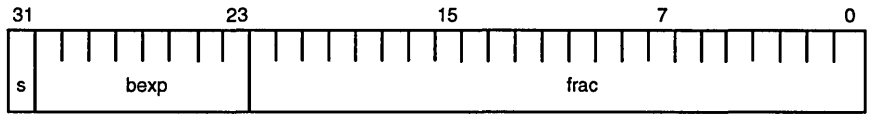
- s: sign bit
- bexp: biased exponent
- frac: fraction
- sig: significand

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.frac * 2**(bexp-127)

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

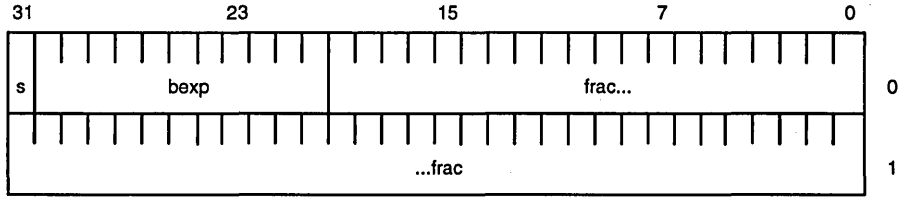


The encoding of special floating-point values is given in Section 3.2.3.

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 depend on the value of the Stack Pointer.

3.2.3 Special Floating-Point Values

The Am29240 microcontroller series 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). An 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 Am29240 microcontroller series distinguishes SNaNs and QNaNs by the most significant bit of the fraction: a 1 indicates a QNaN and a 0 indicates an SNaN.

An operation never generates an 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 Am29240 microcontroller series 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 (+∞) from minus infinity (–∞).

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.frac * 2**(–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

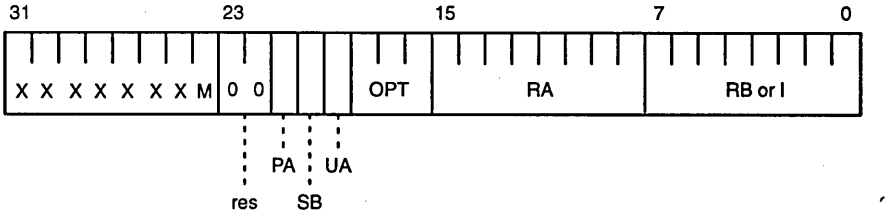
This section discusses external data accesses supported by load and store operations on the Am29240 microcontroller series.

3.3.1 Load/Store Instruction Format

All accesses external to the processor 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.

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.

Figure 3-7 **Load/Store Instruction Format**



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 23–22: Reserved

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

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

Bit 20: Set Byte Pointer/Sign Bit (SB)—If the 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: User Access (UA)—The UA bit allows programs executing in the Supervisor mode to emulate User-mode accesses. This allows checking of the authorization of an access requested by a User-mode program. It also causes address translation (if applicable) to be performed using the PID field of the MMU Configuration Register, rather than the fixed Supervisor-mode process identifier 0.

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

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

Bits 18–16: Option (OPT)—This field 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 20.6.4).

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 5.2).

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 Am29240 and Am29243 microcontrollers, LOADL bypasses the data cache and invalidates any entry found in the cache. This allows the loading of interlocks set by an external master, ensuring that the data cache does not provide a stale value for the interlock. In the Am29245 microcontroller, the 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). In the Am29240 and Am29243 microcontrollers, LOADSET bypasses the data cache and invalidates any entry found in the cache.

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). 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 Am29240 and Am29243 microcontrollers, STOREL is not performed until the write buffer is empty (see Section 9.5) and can be used to ensure that the write buffer is empty; the store is performed in the cache if there is a corresponding entry in the cache. In the Am29245 microcontroller, 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.

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 the valid state of an interrupted program even if the Contents Valid (CV) bit of the Channel Control Register is 0 (see also Section 19.6.2).

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. Later, the multiple access may be resumed at this point by an interrupt return.

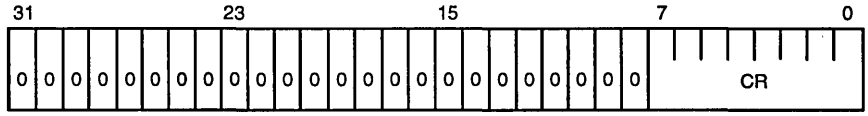
The processor performs multiple accesses using the burst-mode capability of the ROM or the page-mode capability of the DRAM, if possible. Multiple accesses of individual bytes and half-words is not supported. If the memory cannot support burst-mode accesses, a sequence of simple single accesses is performed.

Bursts are stopped and restarted when crossing a 1-Kbyte page boundary.

3.3.4.1 Load/Store Count Remaining Register (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 Addressing and Alignment

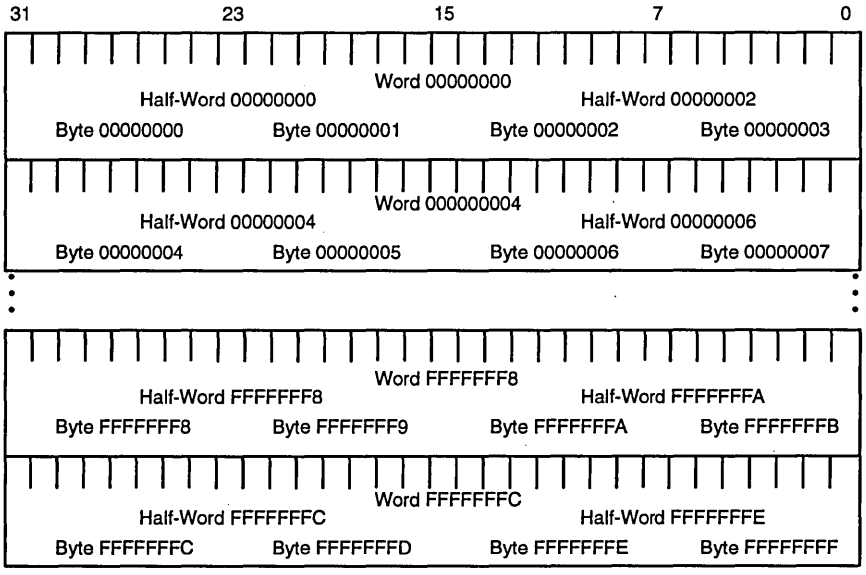
3.3.5.1 Byte and Half-Word Addressing

The Am29240 microcontroller series 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.

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



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 an 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 an 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.

3.3.5.2 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 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 OPT2–OPT0 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.

3.3.5.3 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 Am29240 microcontroller series 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 Am29240 microcontroller series 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.

3.3.5.4 Alignment of Instructions

In the Am29240 microcontroller series, 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 Am29240 microcontroller series 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 Am29240 microcontroller series 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.

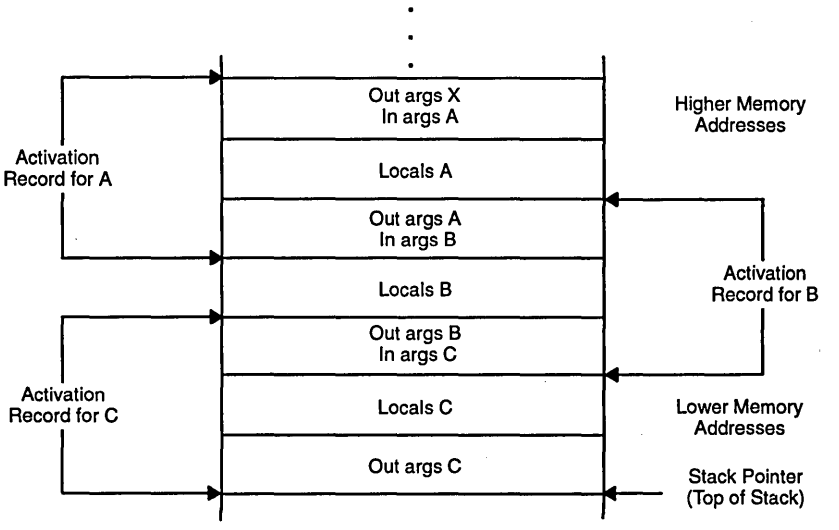
As an example, Figure 4-1 shows three activation records on a run-time stack. This stack configuration was generated by procedure A calling procedure B, which in turn called procedure C. The fact that procedure C is the currently active procedure is reflected by its activation record being on the top 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
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 deallocated, just prior to returning to the caller, by adding to the stack pointer the value subtracted during allocation.

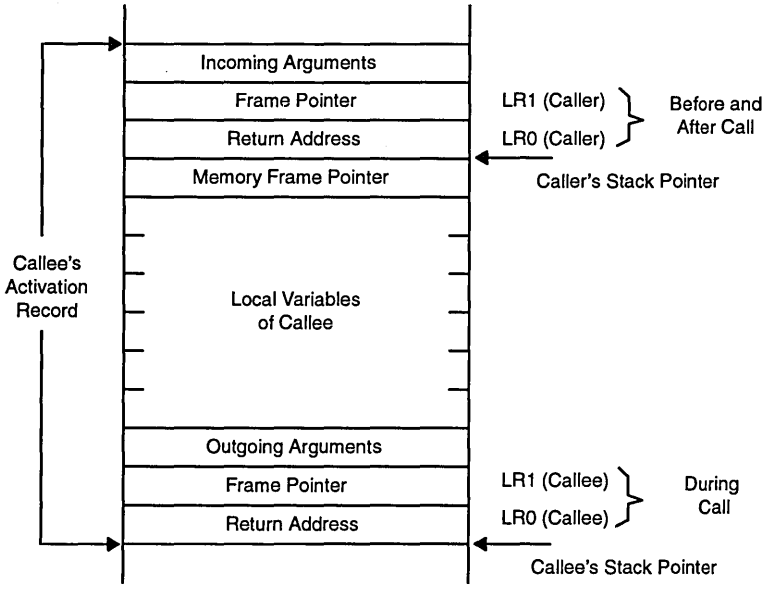
In the Am29240 microcontroller series, run-time storage is actually implemented as two stacks: the Register Stack and the Memory Stack. Storage is allocated and de-allocated on these stacks at the same time. The Register Stack stores activation records associated with all active procedures (except leaf 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 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 Am29240 microcontroller series is designed for efficient implementation of the Register Stack. Specifically, the processor 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 deallocation 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.

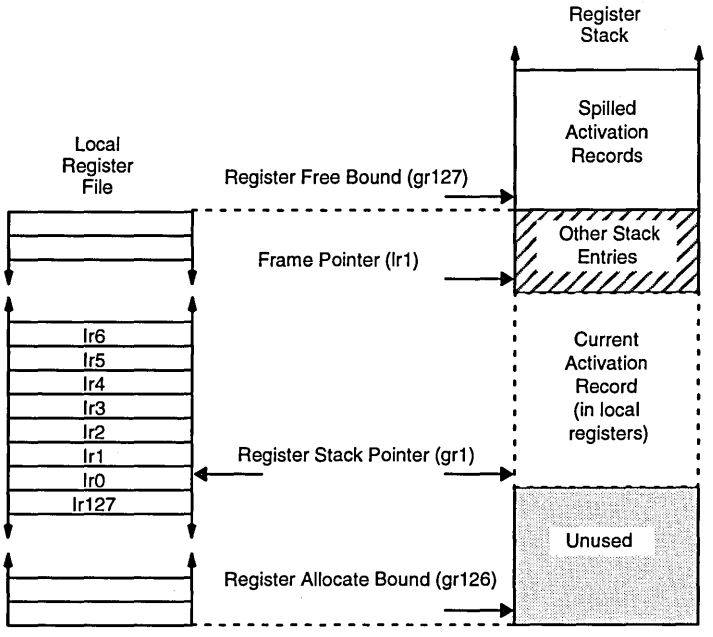
Allocating Register-Stack activation records in the local registers is facilitated by the Stack Pointer in Global Register 1. During the execution of a procedure, the Stack Pointer points simultaneously to the top of the Register Stack in memory and to the local register at the top of the stack cache. 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 deallocation 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 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

Figure 4-3 Relationship of Stack Cache and Register Stack



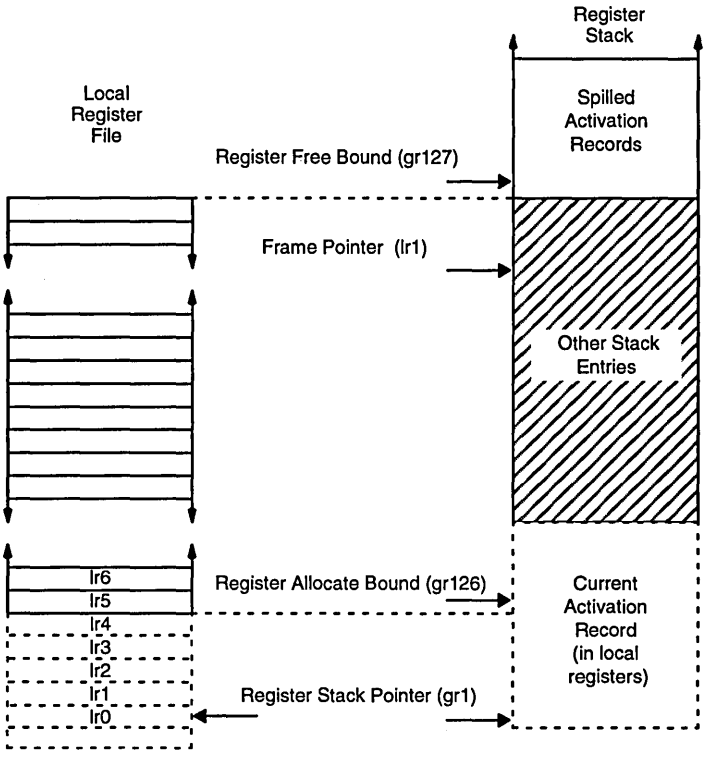
movement of data between the stack cache and the Register Stack in data memory must occur.

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

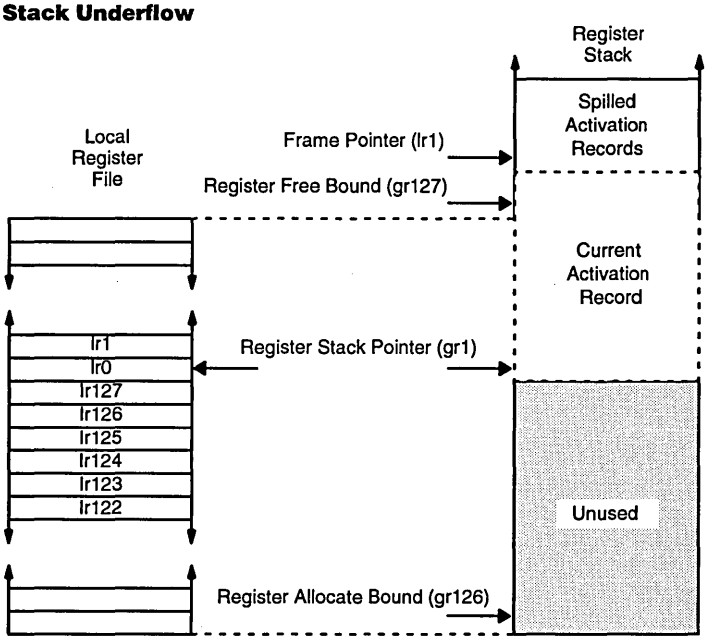


4.1.4 Memory Stack

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.

Figure 4-5 Stack Underflow



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.
- Deallocate 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. Multiword arguments may be split across the Register Stack and the Memory Stack. For example, if a multiword 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 that 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 multiword arguments. Some multiword 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

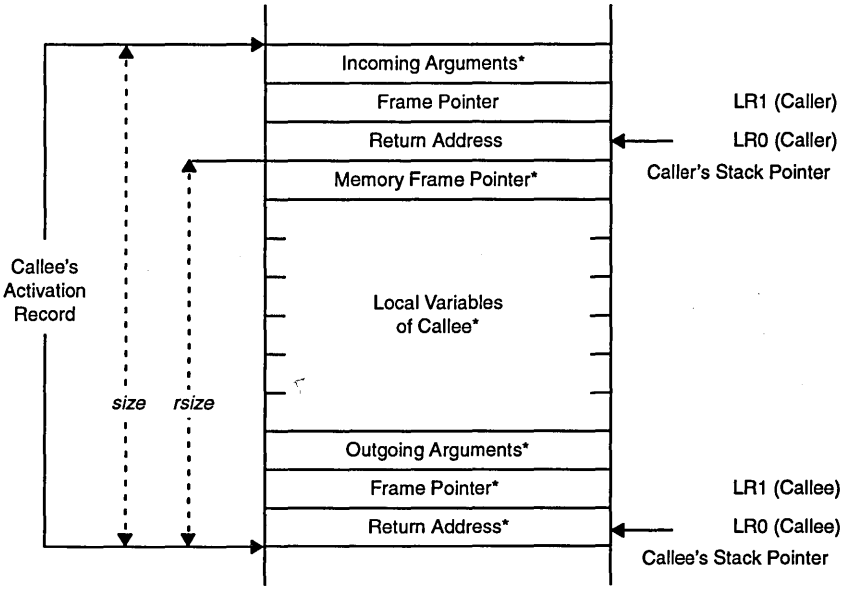
When a procedure is called and the procedure may call another procedure, the callee must allocate a frame for itself on the Register Stack (this is not required for *leaf* procedures that do not call other procedures, as described later). A frame is allocated by decrementing the register stack pointer 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

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



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).

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 deallocates the stack frame allocated by the procedure prologue and returns to the calling procedure. Stack deallocation is accomplished by adding the *rsize* value back to the register stack pointer, after which the deallocated 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
        const   tav, (0x80<<2)          ; local register has high bit set
        or      tav, tav, rfb           ; put starting register number into Indirect
                                        ; Pointer A
        mtsr    IPA, tav
        sub     tav, fp, rfb            ; compute number of bytes to fill
        add     rab, rab, tav           ; 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, rfb          ; fill
        jmpi    tpc                     ; return to trapping procedure
        add     rfb, lr1, 0             ; adjust the free bound
```

4.2.7 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 cannot 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 deallocates 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 deallocated 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     ; deallocate register frame
        add     msp, msp, <msize>       ; deallocate 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
                                          ; deallocate memory frame
        add     rsp, rsp, <rsize>*4       ; deallocate 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 that 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 that 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 COMPLEX PROCEDURE CALL EXAMPLE

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:

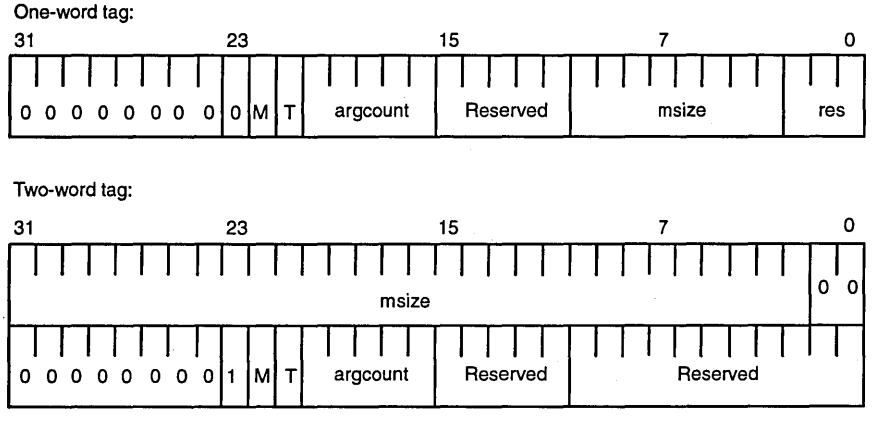
```
        const   tav, (126–2)*4              ; giant register allocation
        sub     rsp, rsp, tav               ; allocate register frame
        asgeu   V_SPILL, rsp, rab
        const   tav, (126–2)*4 + (3*4)      ; incoming arguments and overhead
        add     fp, rsp, tav                ; 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               ; deallocate memory frame
        const   tav, (126–2)*4              ; giant allocation size
        add     rsp, rsp, tav               ; deallocate register frame
        const   gr96, 1                     ; return value
        jmpi    lr0                         ; return to caller
        asleu   V_FILL, fp, rfb             ; insure caller's registers in frame
```

4.5 TRACE-BACK TAGS

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

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

Figure 4-7 Trace-Back Tags



instruction address within a procedure. The call sequence up to the current point in execution can be determined from the *argcount* 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.

1-Word Tag Bits	2-Word Tag Bits	Item	Description
31–24	31–24 (word 2)	opcode	0x00 (an invalid opcode)
23	23 (word 2)	tag type	0=one-word tag; 1=two-word tag
22	22 (word 2)	Mfp	0=no mfp; 1=mfp used
21	21 (word 2)	Transparent	0=normal;1=transparent procedure
20–16	20–16 (word 2)	argcount	Number of arguments in registers (including lr0 and lr1)
15–11	15–0 (word 2)	reserved	Reserved, must be zero
10–3	31–2 (word 1)	msize	Memory frame size in doublewords
2–0	1–0 (word 1)	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 Am29240 microcontroller series pipeline. A description of the 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 microcontroller 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 Am29240 microcontroller series 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 either from the instruction cache or 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. Address translation is performed, and the data cache is accessed in this pipeline stage.

During the write-back stage, the results of the operation performed during the execute stage are stored. In the case of branches or loads that miss in the respective cache, 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 microcontroller (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:

- The processor requires an instruction that is not in the instruction cache or has not been returned by the external instruction memory.
- The processor requires data that is not in the data cache or has not been supplied by an external memory or internal/external peripheral.
- The processor attempts to execute a noncacheable load or store while another noncacheable load or store is in progress.
- The processor attempts to execute a store and the write buffer is full.
- The processor attempts to execute a noncacheable load or store and the write buffer is not empty.
- A data-cache miss occurs and the processor's external interface is busy.
- The processor must perform a serialization operation as described in Section 5.3.
- 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.
- 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.
- 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 Am29240 microcontroller series overlaps data references with other operations in the following situations:

- During a data-cache reload, instruction execution proceeds as long as no instruction depends on the missing data.
- Noncacheable loads and stores are overlapped with the execution of subsequent instructions as long as no instruction depends on the results of the load and no subsequent noncacheable load or store is encountered.
- Cacheable stores are held in the write buffer until they can be performed in the external memory.

These overlapped references must be performed in a way that keeps the processor context constant for the duration of the reference. To ensure that the processor context remains the same, certain operations are serialized.

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

- An access is pending (one of the three cases described previously), and one of the following instructions is encountered:
  Move to Special Register (MTSR)
  Move to Special Register Immediate (MTSRIM)
  Move to TLB (MTTLB)
  Interrupt Return (IRET)
  Interrupt Return and Invalidate (IRETINV)
  Halt (HALT)
- An access is pending, 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 all pending accesses are complete.

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 that 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 Am29240 microcontroller series 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, the Current Processor Status Register, the MMU Configuration Register, the Cache Data Register, and the Cache Interface 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 interrupt or trap execution cycles elapse between the two instructions of the interrupted routine. For this reason, a program should not be written in a manner that relies on the delayed effect; the results of this practice may be unpredictable.

At least one cycle of delay must separate a Move To Special Register instruction that modifies the Page Size (PS0) field (or either PS field in the Am29243 microcontroller) of the MMU Configuration Register and an instruction that performs address translation. The latter instruction includes successful branches, loads, and stores.

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

An access to the Cache Data Register cannot immediately follow a write to the Cache Interface Register. At least one instruction must separate the two accesses.

CHAPTER

6 SYSTEM PROTECTION

The Am29240 microcontroller series 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 SUPERVISOR AND USER MODES

At any given time, the microcontroller operates in one of two mutually exclusive program modes: the Supervisor mode or the User mode. All system-protection features are based on these 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 19.1.1). In the Supervisor mode, executing programs have access to all processor resources. However, virtual pages mapped by the Memory Management Unit are protected from Supervisor write access when the Supervisor Write bit is 0 in the corresponding Translation Look-Aside Buffer entry (see Chapter 7).

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

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 Translation Look-Aside Buffer (TLB) register.
2. An attempted access of any general-purpose register for which a bit in the Register Bank Protect Register is 1 (see Section 6.2).
3. An attempted execution of a load or store instruction for which the PA bit is 1 or for which the UA bit is 1 (see Section 3.3.1).
4. An attempted execution of one of the following instructions: Interrupt Return, Interrupt Return and Invalidate, Invalidate, or Halt. However, a hardware-development system can disable protection checking for the Halt instruction, so this instruction may be used to implement instruction breakpoints in User-mode programs (see Sections 20.2).
5. An attempted access of special-purpose register in the range of 0 to 127 or 160 to 255.
6. An attempted execution of an assert or Emulate instruction that specifies a vector number between 0 and 63, inclusive (see Section 19.2.2).

7. An attempted access (read, write, or execute) in a virtual page mapped by the Memory Management Unit when the appropriate permission bit (UR, UW, or UE, respectively) is 0 in the corresponding TLB entry.

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. Bit 0–Bit 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.

Figure 6-1 Register Bank Organization

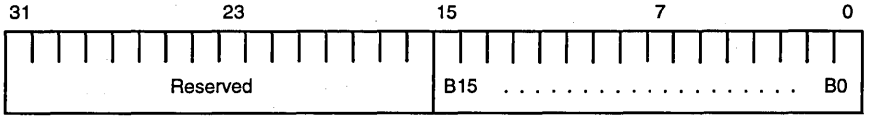
Register Bank Protect Register Bit	Absolute-Register Numbers	General-Purpose Registers
0	2 through 15	Bank 0 (not implemented)
1	16 through 31	Bank 1 (not implemented)
2	32 through 47	Bank 2 (not implemented)
3	48 through 63	Bank 3 (not implemented)
4	64 through 79	Bank 4
5	80 through 95	Bank 5
6	96 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

6.2.1 Register Bank Protect Register (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 6-2 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).

6.3 MEMORY PROTECTION

Memory access protection is provided by the MMU. Each Translation Look-Aside Buffer (TLB) entry in the MMU contains protection bits that determine whether or not an access is permitted to the page associated with the entry.

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

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

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

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

Table 6-1 Access Protection

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

CHAPTER

7 MEMORY MANAGEMENT UNIT

The Am29240 microcontroller series incorporates a Memory Management Unit (MMU) for performing virtual-to-physical address translation and memory access protection. The MMU also supports the DRAM mapping function of the Am29200 and Am29205 microcontrollers in a way that is transparent to applications software (however, the system software is different for the Am29200 or Am29205 and the Am29240 microcontroller series). This chapter describes the logical operation of the MMU.

The fundamental structure of the MMU is the Translation Look-Aside Buffer (TLB). A TLB translates pages ranging in size from 1 Kbyte to 16 Mbyte in powers of four. This chapter also describes the structure of the TLB and the issues related to software management of the TLB.

In the Am29243 microcontroller, address translation is performed by two, two-way set-associative translation look-aside buffers, TLB0 and TLB1. The page size is individually selectable for each TLB. Alternatively, the two TLBs in the Am29243 microcontroller can be configured as a single, four-way set-associative TLB with pages of a single size. In the Am29240 and Am29245 microcontrollers, address translation is performed by a single TLB that corresponds to TLB 0 in the Am29243 microcontroller.

7.1 TRANSLATION LOOK-ASIDE BUFFER

The MMU stores the most recently performed address translations in two Translation Look-Aside Buffers (TLBs). The TLBs reflect information in the system page tables, except that they specify the translation for many fewer pages.

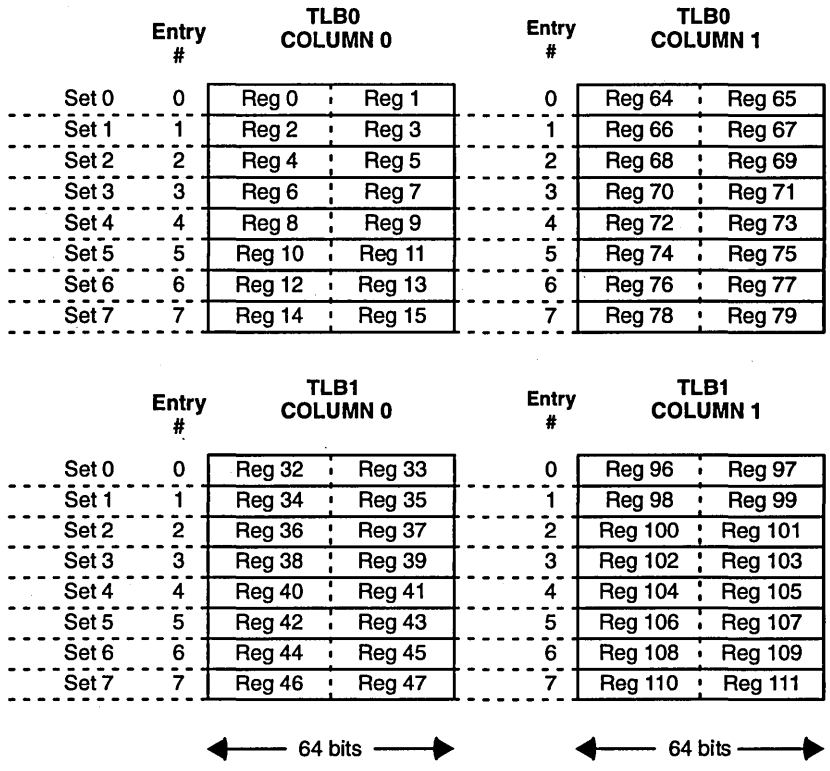
A diagram of the TLBs is shown in Figure 7-1. Each TLB is a table of 16 entries, divided into two equal columns, called column 0 and column 1. Within each column of each TLB, entries are numbered 0 to 7. Entries in different columns that have equivalent entry-numbers are grouped into a unit called a set; for example, there are eight sets in each TLB, numbered 0 to 7, and a total of 16 sets for both TLBs on the Am29243 microcontroller.

Each TLB entry is 64 bits long and contains mapping and protection information for a single virtual page. Pages mapped by the TLBs range in size from 1 Kbyte to 16 Mbyte. TLB entries may be inspected and modified by processor instructions executed in the Supervisor mode. The layout of TLB entries is described in Section 7.2.

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

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

Figure 7-1 Translation Look-Aside Buffer Organization



7.2 TLB REGISTERS

The Am29243 microcontroller contains 64 TLB registers. The organization of the TLB registers is shown in Figure 7-1.

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

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

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

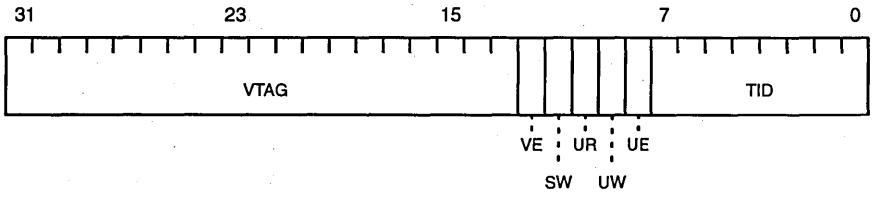
TLB entries are accessed as registers numbered 0–111 (not all registers are implemented within this range). Since two words are required to completely specify a TLB entry, two registers are required for each TLB entry. The words corresponding to an

entry are paired as two sequentially numbered registers starting on an even-numbered register. The word with the even register number is called Word 0, and the word with the odd register number is called Word 1. The entries for TLB Column 0 are in registers numbered 0–47, and the entries for TLB Column 1 are in registers numbered 64–111 (not all registers are implemented within these ranges).

7.2.1 TLB Entry Word 0 Register

The TLB Entry Word 0 register is shown in Figure 7-2.

Figure 7-2 TLB Entry Word 0 Register



Bits 31–13: Virtual Tag (VTAG)—When the TLB is searched for an address translation, the VTAG field of the TLB entry must match high-order bits of the address being translated for the search to be successful. The bits that must match depend on the page size.

When software loads a TLB entry with an address translation, the high-order bits of the Virtual Tag are set with the most significant 5 bits of the virtual address whose translation is being loaded into the TLB. The remaining bits of the Virtual Tag must be set either to the corresponding bits of the address or to zeros depending on the page size, as follows ("A" refers to corresponding address bits):

Page Size	VTAG 13–0 (TLB Word 0 bits 26–13)
1 Kbyte	AAAAAAAAAAAAAA
4 Kbyte	AAAAAAAAAAAA00
16 Kbyte	AAAAAAAAAA0000
64 Kbyte	AAAAAAAA000000
256 Kbyte	AAAAAA00000000
1 Mbyte	AAAA0000000000
4 Mbyte	AA000000000000
16 Mbyte	00000000000000

Bit 12: Valid Entry (VE)—If this bit is 1, the associated TLB entry is valid; if it is 0, the entry is invalid.

Bit 11: Supervisor Write (SW)—If the SW bit is 1, Supervisor-mode store operations to the virtual page are allowed; if it is 0, Supervisor-mode stores are not allowed.

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

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

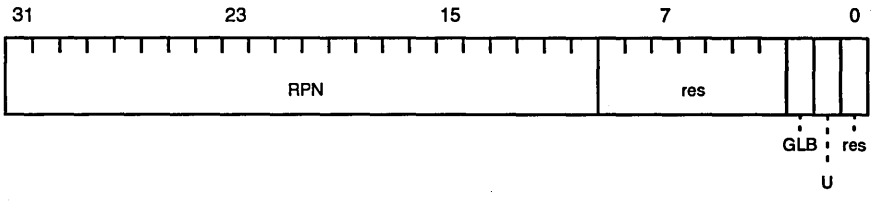
Bit 8: User Execute (UE)—If the UE bit is 1, User-mode instruction accesses to the virtual page are allowed; if it is 0, User-mode instruction accesses are not allowed.

Bits 7–0: Task Identifier (TID)—When the TLB is searched for an address translation, the TID must match the Process Identifier (PID) in the MMU Configuration Register for the translation to be successful. This field allows the TLB entry to be associated with a particular process. The TID comparison is ignored if the Global Page (GLB) bit is set in Word 1.

7.2.2 TLB Entry Word 1 Register

The TLB Entry Word 1 register is shown in Figure 7-3.

Figure 7-3 TLB Entry Word 1 Register



Bits 31–10: Real Page Number (RPN)—The RPN field gives the high-order bits of the physical address of the page. It is concatenated to low-order bits of the address being translated to form the physical address for the access.

When software loads a TLB entry with an address translation, the most significant 8 bits of the Real Page Number are set with the most significant 8 bits of the physical address associated with the translation. The remaining bits of the Real Page Number must be set either to the corresponding bits of the physical address,or to zeros, depending on the page size, as follows ("A" refers to corresponding address bits):

Page Size	RPN 13–0 (TLB Word 1 bits 23–10)
1 Kbyte	AAAAAAAAAAAAAA
4 Kbyte	AAAAAAAAAAAA00
16 Kbyte	AAAAAAAAAA0000
64 Kbyte	AAAAAAAA000000
256 Kbyte	AAAAAA00000000
1 Mbyte	AAAA0000000000
4 Mbyte	AA000000000000
16 Mbyte	00000000000000

Bits 9–3: Reserved

Bit 2: Global Page (GLB)—This bit indicates that the page is global: that is, the page is mapped to all processes. If the GLB bit is set in the TLB, the TID-to-PID comparison is ignored during address translation.

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

the entry containing the translation. This bit is set whenever the virtual-to-physical translation is valid, regardless of the outcome of protection checking.

Bit 0: Reserved

7.3 ADDRESS TRANSLATION CONTROLS

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

7.3.1 Enabling and Disabling Address Translation

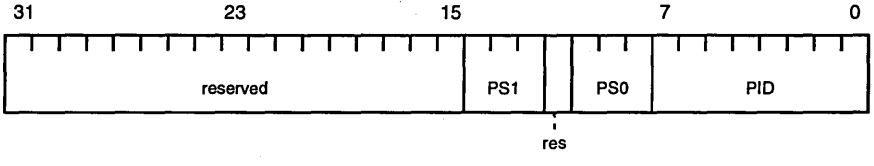
The processor attempts to perform address translation for the following external accesses.

- Instruction accesses, if the Physical Addressing/Instructions (PI) bit of the Current Processor Status (CPS) Register is 0, or if the PI bit is 1 and the address is in the range 50000000–53FFFFFF. The latter case allows the MMU to support mapped DRAM accesses that are compatible with the Am29200 and Am29205 microcontrollers.
- User-mode data accesses, if the Physical Addressing/Data (PD) bit of the CPS is 0, or if the PD bit is 1 and the address is in the range 50000000–53FFFFFF. The latter case allows the MMU to support mapped DRAM accesses that are compatible with the Am29200 and Am29205 microcontrollers.
- Supervisor-mode data accesses, if the Physical Address (PA) bit of the load or store instruction is 0 and the PD bit of the CPS is 0, or if the PA or PD bit is 1 and the address is in the range 50000000–53FFFFFF. The latter case allows the MMU to support mapped DRAM accesses that are compatible with the Am29200 and Am29205 microcontrollers.

7.3.2 MMU Configuration Register (MMU, Register 13)

This protected special-purpose register (Figure 7-4) specifies parameters associated with the MMU. The Am29243 microcontroller has two TLBs so it has fields to specify the page size for each of the TLBs independently. The Am29240 and Am29245 microcontrollers each have a single page-size field.

Figure 7-4 MMU Configuration Register



Bits 31–15: Reserved

Bits 14–12: Page Size, TLB1 (PS1), Am29243 microcontroller only—The PS1 field specifies the page size for address translation by TLB1. The PS1 field has a delayed effect on address translation. At least one cycle of delay must separate an instruction that sets the PS1 field and an instruction that performs address translation. The PS1 field is encoded as follows:

PS1	Page Size
000	1 Kbyte
001	4 Kbyte
010	16 Kbyte
011	64 Kbyte
100	256 Kbyte
101	1 Mbyte
110	4 Mbyte
111	16 Mbyte

Bit 11: Reserved

Bits 10–8: Page Size, TLB0 (PS0)—The PS0 field specifies the page size for address translation by TLB0. The PS0 field has a delayed effect on address translation. At least one cycle of delay must separate an instruction that sets the PS0 field and an instruction that performs address translation. The PS0 field encoding is the same as the PS1 encoding.

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

7.4 ADDRESS TRANSLATION DESCRIPTION

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

7.4.1 Virtual Address Structure

Virtual addresses are partitioned into three fields for TLB address translation, as shown in Figure 7-5. The partitioning of the virtual address is based on the page size. The page size is specified by the MMU Configuration Register.

7.4.2 Address-Translation Process

The TLB address-translation process is diagrammed in Figure 7-6 (Figure 7-6 shows a single TLB—the Am29243 microcontroller has two TLBs, each identical to that shown in Figure 7-6). Address translation is performed by the following fields in the TLB entry: the Virtual Tag (VTAG), the Task Identifier (TID), the Valid Entry (VE) bit, the Real Page Number (RPN) field, and the Global Page (GLB) bit. To perform an address translation, the processor accesses the TLB set (or sets) whose number is

Figure 7-5 Virtual Address Structure

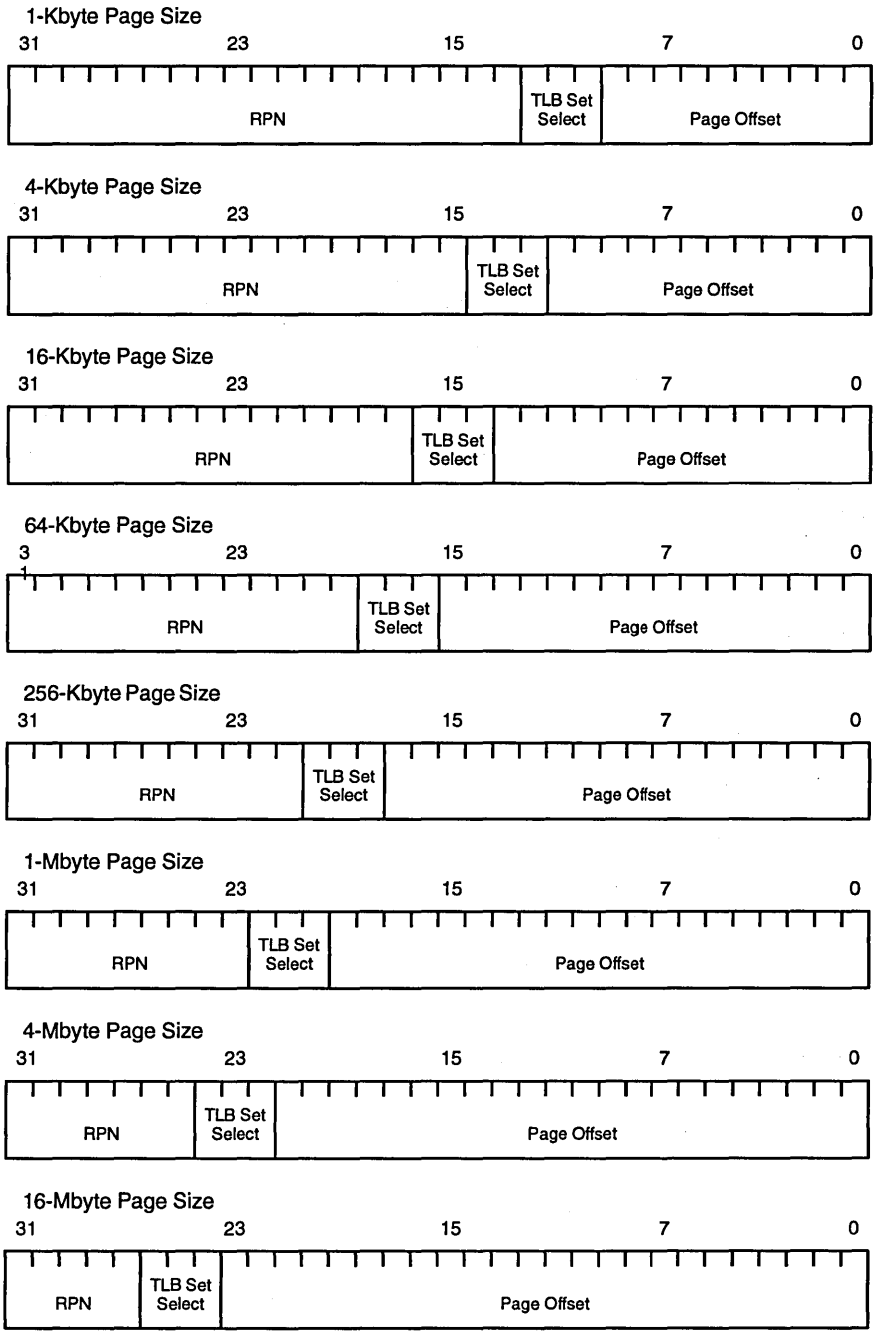
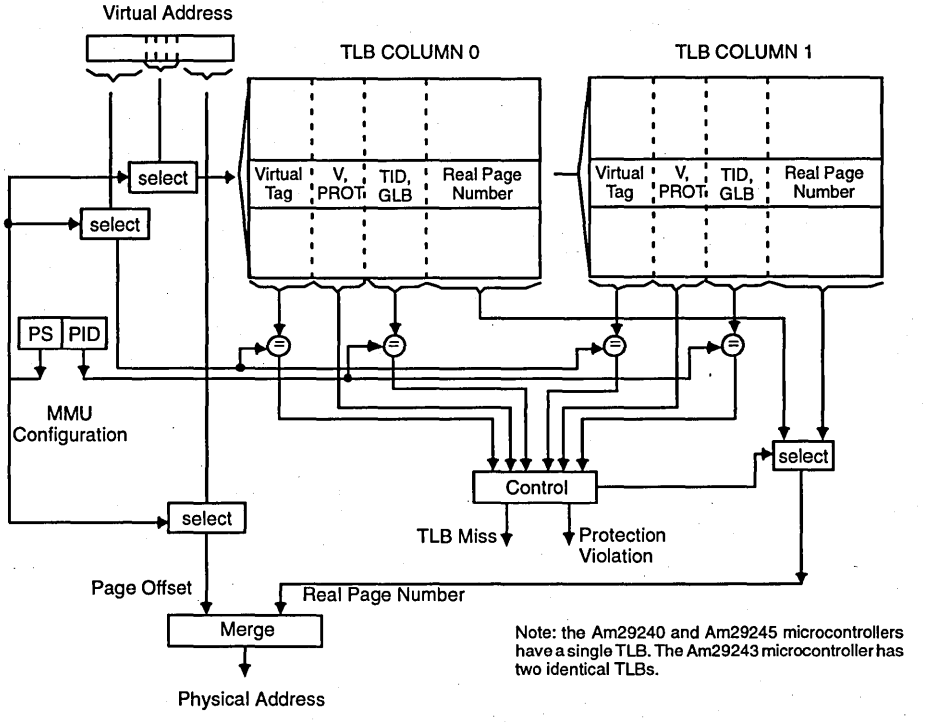


Figure 7-6 TLB Address-Translation Process (Single TLB)



given by certain bits in the virtual address. The bits used depend on the page size (and therefore can be different for each TLB in the Am29243 microcontroller) as follows:

Page Size	Virtual Address Bits (for Set Access)
1 Kbyte	12–10
4 Kbyte	14–12
16 Kbyte	16–14
64 Kbyte	18–16
256 Kbyte	20–18
1 Mbyte	22–20
4 Mbyte	24–22
16 Mbyte	26–24

In the Am29240 and Am29245 microcontrollers, one TLB set is accessed for a total of two entries. In the Am29243 microcontroller, two TLB sets (one from each TLB) are accessed for a total of four entries. The VTAG field of each entry is compared to bits in the virtual address. This comparison depends on the page size (and therefore can be different for each TLB in the Am29243 microcontroller) as shown in the following table. (Note that VTAG bit numbers are relative to the VTAG field, not the TLB entry.)

Page Size	Virtual Address Bits	VTAG Bits
1 Kbyte	31–13	18–0
4 Kbyte	31–15	18–2
16 Kbyte	31–17	18–4
64 Kbyte	31–19	18–6
256 Kbyte	31–21	18–8
1 Mbyte	31–23	18–10
4 Mbyte	31–25	18–12
16 Mbyte	31–27	18–14

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

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

- The virtual address bits must match corresponding bits of a VTAG field for one of the TLB entries, as specified above.
- For a User-mode access, the TID field in the matching TLB entry must match the PID field in the MMU Configuration Register or the GLB bit in the TLB entry must be 1. For a Supervisor-mode access, the TID field must be zero or the GLB bit in the TLB entry must be 1.
- The VE bit in the matching TLB entry must be 1.
- Only one entry in the set can meet conditions 1, 2, and 3 above. If this condition is not met, the results of the translation may be treated as valid by the processor, but the results are unpredictable.

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

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

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

Page Size	RPN Bits	Virtual Address Bits for Page Offset
1 Kbyte	21–0	9–0
4 Kbyte	21–2	11–0
16 Kbyte	21–4	13–0
64 Kbyte	21–6	15–0
256 Kbyte	21–8	17–0
1 Mbyte	21–10	19–0
4 Mbyte	21–12	21–0
16 Mbyte	21–14	23–0

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

Once the physical address is formed, the processor applies the address range decoding described in Sections 10.3 and 10.4. External and internal peripherals can

be mapped and protected by the MMU. Also, the processor determines cacheable data based on the translated physical address (only access to ROM and DRAM regions are cached, regardless of the virtual address). Finally, if the physical address is in the mapped DRAM region (physical address 50000000–53FFFFFF), the processor accesses the DRAM using address bits 25–0 as if the access were a normal DRAM access, and no further mapping is applied to the physical address.

7.4.3 Successful and Unsuccessful Translations

If the TLB cannot translate an address, a TLB miss occurs. If an address translation is successful, the TLB entry is further used to perform protection checking for the access. Bits in the TLB make it possible to restrict User-mode accesses to any combination of load, store, and instruction accesses, or to no access. Supervisor-mode programs can be restricted to read-only access to allow early detection of invalid Supervisor writes. Section 6.3 describes MMU protection in more detail.

The MMU causes a trap if either a TLB miss occurs, or the translation is successful and a protection violation is detected. The processor distinguishes between traps caused by instruction and data accesses, and between traps caused by User- and Supervisor-mode accesses, as follows:

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

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

7.4.4 Cache Considerations

The instruction cache is accessed with virtual addresses if address translation is enabled for instruction accesses. Because of this, the cache may contain entries that the processor might consider valid, even though they are not.

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

To avoid invalid cache accesses, the contents of the cache must be invalidated explicitly whenever address translation is changed. This can be accomplished by executing an Invalidate (INV) instruction that specifies the instruction cache whenever an address translation is changed for instructions. The INV instruction causes all entries of the instruction cache to become invalid. Invalidation occurs after the next successful branch or cache block boundary.

Since a change in address translation rarely affects the logical behavior of the program performing the change, the INV may unnecessarily affect the performance of this program by flushing the cache of instructions used by the program. The IRETINV instruction has the same effect on the instruction cache as the INV instruction, but reduces the performance impact by delaying invalidation until an interrupt return is executed, eliminating the need to disrupt the routine that changes address translation.

At the point of interrupt return, the contents of the cache are most likely not of much use anyway.

Disabling the instruction cache does not cause automatic invalidation. When disabled, the cache retains its previous contents, but the processor considers its contents to be invalid. Furthermore, the cache may have to be invalidated before it is re-enabled, because the cache may retain contents that the processor may incorrectly treat as valid when the cache is enabled.

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

If a TLB miss occurs during address translation for a branch target instruction, the processor considers the contents of the instruction cache to be invalid, regardless of whether the target instruction is in the cache. This is required to properly sequence the LRU Recommendation Register.

7.4.5 Selecting the Virtual Page Size

The selection of page size is based on several considerations:

- For a given page size, any allocation of pages to a process will, on average, waste half of one page. With smaller page sizes, the waste is smaller. In systems with a large number of processes, each with a small amount of memory, small page sizes can reduce waste significantly.
- Smaller page sizes allow finer memory-protection granularity.
- The maximum amount of memory that can be referenced by Translation Look-Aside Buffer (TLB) entries is set by the number of TLB entries and the page size. Larger page sizes allow the fixed number of TLB entries to address more memory, and generally reduce the number of TLB misses. For example, with 1-Kbyte pages, a process requiring 8 Kbytes of contiguous memory would create eight TLB misses. With 8-Kbyte pages, the process would create only one TLB miss.
- The page is usually the unit of memory moved between memory and backing storage. The design of the backing storage sub-system may also influence the choice of page size, because of transfer-efficiency considerations. For example, if the backing storage is a disk, the disk seek time is large compared to transfer time. Thus, it is more efficient to transfer large amounts of data with a single seek. Efficiency may also depend on disk organization (i.e., the number of seeks possibly required to transfer a page).

The Am29243 microcontroller MMU allows pages of two different sizes, providing more flexibility in configuring the page size for the desired characteristics. For example, the memory waste of allocating large pages to handle small variations in memory requirements can be avoided by using one TLB to translate small pages for dynamically-allocated structures, such as the run-time stack. At the same time, the other TLB can translate large pages for very large structures such as frame buffers and shared libraries, permitting the entire structure to be addressed without TLB misses.

7.5 HANDLING TLB MISSES

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

If a required address translation cannot be performed by any entry in the TLB, a TLB miss trap occurs. The trap handling routine—called the TLB reload routine—accesses the system page tables to determine the required translation and sets the appropriate TLB entry. Note that the access requiring this translation can be restarted by the interrupt return at the end of the TLB reload routine (see Section 19.6.2).

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

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

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

7.5.1 TLB Reload

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

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

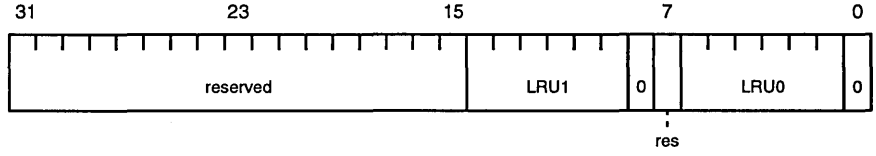
The TLB reload routine determines the translation for the address given by the Program Counter 1 Register or Channel Address Register, as appropriate. The TLB reload routine uses an external page table to determine the required translation, and loads the TLB entry indicated by the LRU Recommendation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

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

7.5.2 LRU Recommendation Register (LRU, Register 14)

This protected special-purpose register (Figure 7-7) assists TLB reloading by indicating the least recently used TLB entry in the required replacement set. The Am29243 microcontroller has two TLBs, so it has fields to specify the entry for each of the TLBs independently (system software must determine which of the TLBs is to be loaded). The Am29240 and Am29245 microcontrollers each have a single field to indicate the least recently used entry.

Figure 7-7 LRU Recommendation Register



Bits 31–15: Reserved

Bits 14–9: Least-Recently Used Entry, TLB1 (LRU1)—The LRU1 field is updated whenever a TLB miss occurs. It gives the TLB register number of the TLB1 entry that would be selected for replacement. This is used by software to reload TLB1 if the translation should have been performed by TLB1. The LRU1 field also is updated whenever a memory-protection violation occurs; however, it has no interpretation in this case.

Bit 8: Zero—The appended 0 serves to identify Word 0 of the TLB1 entry.

Bit 7: Reserved

Bits 6–1: Least-Recently Used Entry, TLB0 (LRU0), Am29243 microcontroller only—The LRU0 field, used only by the Am29243 microcontroller, is updated whenever a TLB miss occurs. It gives the TLB register number of the TLB0 entry that would be selected for replacement. This is used by software to reload TLB0 if the translation should have been performed by TLB0. The LRU0 field is also updated whenever a memory-protection violation occurs; however, it has no interpretation in this case.

Bit 0: Zero—The appended 0 serves to identify Word 0 of the TLB0 entry.

7.5.3 Page Reference and Change Information

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

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

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

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

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

7.5.4 Warm Start

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

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

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

7.5.5 Minimum Number of Resident Pages

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

For the Am29240 microcontroller series, two pages are required for a process to make progress through the system. The reason for this requirement is that the Am29240 microcontroller series, on interrupt return, restarts an interrupted Load Multiple or Store Multiple only after fetching two instructions (see Section 19.3.4). The first of these instructions must be resident in memory—and mapped by the TLB—and

the page required to complete the Load Multiple or Store Multiple must also be resident—and mapped by the TLB—for the interrupt return to complete successfully.

7.6 INVALIDATING TLB ENTRIES

There are two methods for invalidating TLB entries that are no longer required at a given point in program execution. The first involves resetting the Valid Entry bit of a single entry (this is done by a Move To TLB instruction). The second involves changing the value of the Process Identifier (PID) field of the MMU Configuration Register; this invalidates all entries whose Task Identifier (TID) fields do not match the new value.

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

CHAPTER

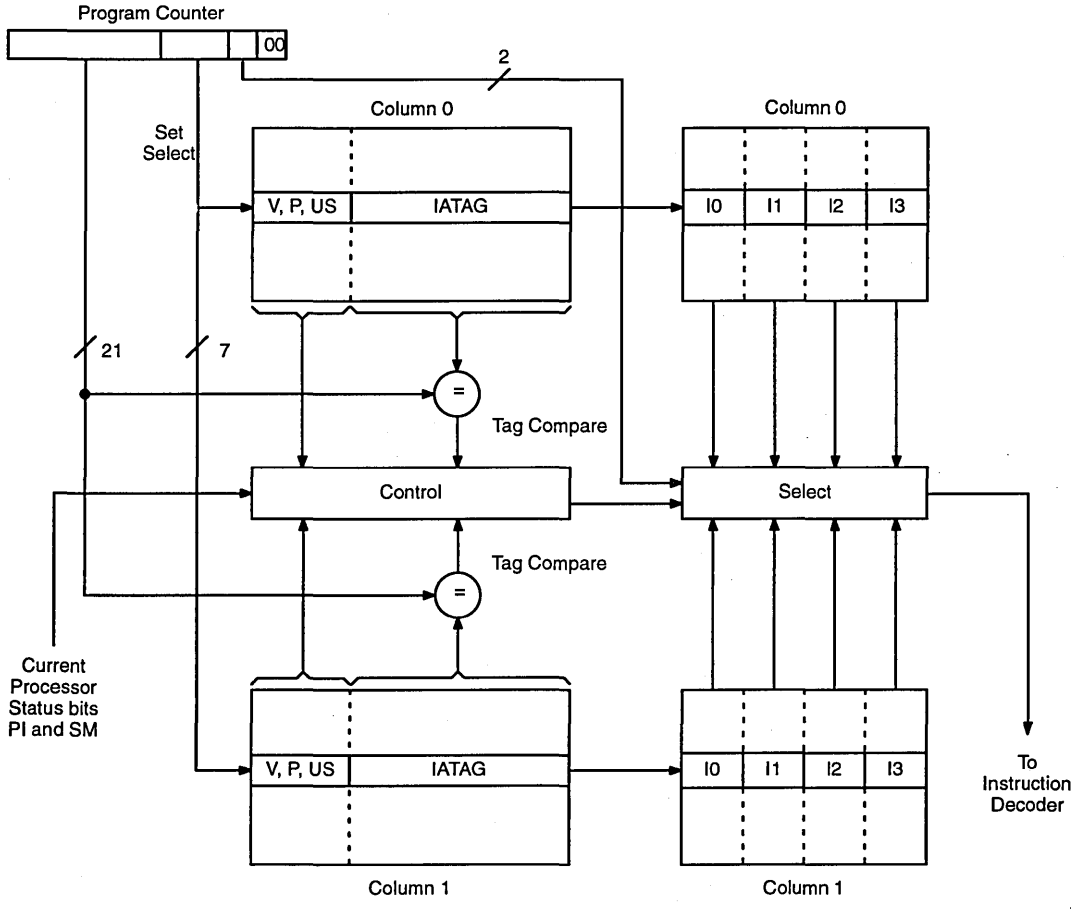
8 INSTRUCTION CACHE

This chapter describes the instruction cache of the Am29240 microcontroller series and the mechanisms used to load this cache.

8.1 INSTRUCTION CACHE OVERVIEW

The Am29240 microcontroller series has a 4-Kbyte, two-way set-associative instruction cache (Figure 8-1). The block size is four words (16 bytes). The cache stores the most recent instructions fetched by the processor. In addition to instructions, the instruction cache maintains status information for each cache block.

Figure 8-1 Instruction Cache Organization



The instruction cache is enabled and disabled by the Instruction Cache Disable (ID) bit of the Configuration Register. If the instruction cache is enabled, instruction fetches may be satisfied by the cache. If the instruction cache is disabled, instruction fetches are satisfied only by the external instruction/data memory and the cache does not store fetched instructions. A disabled cache can be invalidated by an INV or IRETINV instruction that specifies the instruction cache.

To keep critical routines in the cache, blocks in the instruction cache can be locked by the Instruction Cache Lock (IL) field of the Configuration Register. The IL field can lock either all blocks in the cache or blocks in column 0. When a block is locked, it is not available for replacement if it is valid. A locked block may be allocated if it is invalid—this allows a critical routine to be loaded into the cache simply by executing the routine after the cache is invalidated. Also a locked block cannot be invalidated (unless the cache is also disabled—the disable overrides the lock).

The instruction cache has a valid bit per word, so that it can fetch and store partially-valid blocks. During reload, the valid bit of a word is set as the word is written into the cache. All valid bits are cleared in a single cycle by a processor reset or by the execution of an INV or IRETINV instruction.

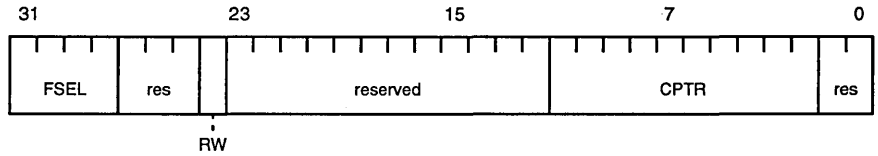
8.2 ACCESSING CACHE FIELDS

Each instruction cache block is accessible via the Cache Interface Register and Cache Data Register. The Cache Interface Register contains a pointer to the accessed block and specifies the accessed field. The Cache Data Register is used to transfer data to and from the cache. The contents of the Cache Data Register may not survive across a cache write or a register read. The cache should be disabled while cache fields are read and written, to prevent interference from cache reloading.

8.2.1 Cache Interface Register (CIR, Register 29)

This protected special-purpose register (Figure 8-2) allows fields of the instruction and data caches to be read or written. Cache fields are read or written when the Cache Interface Register is written. The Cache Data Register receives or supplies the associated data. This allows cache testing as well as the implementation of operations such as cache preload.

Figure 8-2 Cache Interface Register



Bits 31–28: Cache Field Select (FSEL)—The FSEL field selects the cache field that is read or written when the Cache Interface Register is written, as follows:

FSEL Value	Cache Field Selected/Cache Data Register Bits
0000	Instruction word/31–0
0001	Instruction address tag/31–11, status/5–0
0010–0111	Reserved for instruction cache
1000	Data cache word/31–0
1001	Data address tag/31–10, status/0
1010–1111	Reserved for data cache

Bits 27–25: Reserved

Bit 24: Read/Write (RW)—If the RW bit is 0, the cache field selected by the FSEL field is read into the Cache Data Register when the Cache Interface Register is written. If the RW bit is 1, the contents of the Cache Data Register are written into the cache field.

Bits 23–12: Reserved

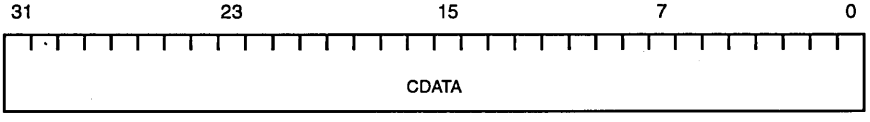
Bit 11–2: Cache Pointer (CPTR)—The Cache Pointer field selects the block or word within the cache to be read or written. If the FSEL field selects a block-level field (for example, an address tag), the two least significant bits of the CPTR field are ignored in the selection of the cache field. If the FSEL field selects a word-level field (for example, an instruction word), the entire CPTR field is used in the selection of the cache field. The most significant bit of the CPTR field distinguishes between column 0 (msb=0) and column 1 (msb=1) of the instruction cache. Fields in the data cache are selected by the nine low-order bits of the CPTR field (bits 10–2 of the Cache Interface Register; bit 10 selects the column).

Bits 1–0: Reserved

8.2.2 Cache Data Register (CDR, Register 30)

This protected special-purpose register (Figure 8-3) receives or provides data for cache read or write operations, respectively. The Cache Data Register is not persistent: its contents may be destroyed by a cache write or by a register read.

Figure 8-3 Cache Data Register

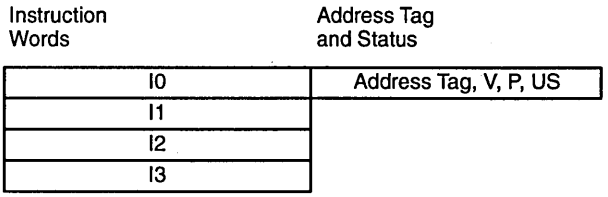


Bits 31–0: Cache Data (CDATA)—When a cache field is written, the data for the write is supplied by the appropriate bits of the CDATA field. When a cache field is read, the data for the read is stored into the CDATA field. The description of the FSEL field in the Cache Interface Register relates the CDATA fields to cache fields.

8.2.3 Instruction Cache Access

Figure 8-4 shows the organization of an individual instruction cache block. There are 256 such blocks in the instruction cache, organized as two columns of 128 blocks each. For access, a particular column and block are selected by 8 bits of the CPTR field (bits 11–4 of the Cache Interface Register; bit 11 selects the column). The

Figure 8-4 Instruction Cache Block Organization

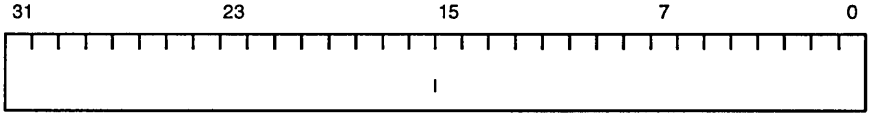


accessed field in the block is specified by the Cache Field Select (FSEL) field. When an instruction word is accessed, the instruction is further selected by the two least significant bits of the CPTR field (bits 3–2 of the Cache Interface Register). This section describes the fields within the instruction cache and how they relate to data in the Cache Data Register upon access.

8.2.3.1 Instruction Words

Figure 8-5 shows the placement of an instruction in the Cache Data Register, for reading or writing the instruction cache. In this case, the instruction takes up the entire register.

Figure 8-5 Instruction Word in Cache Data Register

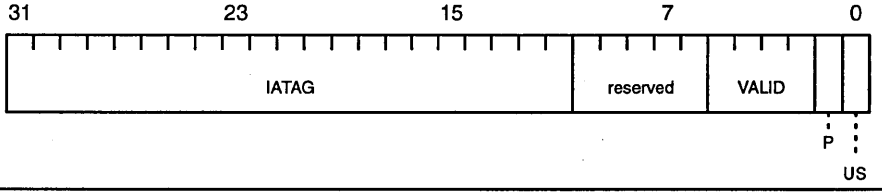


Bits 31–0: Instruction (I)—This is the 32-bit instruction that is read or written.

8.2.3.2 Address Tag and Status Information

Figure 8-6 shows the placement of the tag and status fields in the Cache Data Register, for reading or writing the instruction cache.

Figure 8-6 Instruction Address Tag and Block Status in Cache Data Register



Bits 31–11: Instruction Address Tag (IATAG)—The IATAG field specifies which address in ROM or DRAM is mapped by the cache block.

Bits 10–6: Reserved

Bits 5–2: Valid (VALID)—A bit is set in this field if the corresponding instruction word is valid. The most significant bit is the valid bit for the first word in the block and the least significant bit is the valid bit for the fourth word in the block. All valid bits in the cache are cleared in a single cycle by a processor reset and by the execution of an INV or IRETINV that specifies the instruction cache.

Bit 1: Physical Address (P)—The P bit indicates whether the cache tag reflects a physical address or a virtual or mapped DRAM address. If the P bit is 1, the cache block contains instructions from the physical address space. If the P bit is 0, the cache block contains instructions from a virtual address space or mapped DRAM region.

Bit 0: User or Supervisor Block (US)—The US bit reflects the state of the SM bit in the Current Processor Status Register at the time the instructions in the block were fetched. If the US bit is 1, the block contains instructions from a Supervisor-mode program. If the US bit is 0, the block contains instructions from a User-mode program.

8.3 CACHE HITS AND MISSES

On every cycle, bits of the processor's program counter (PC) are used to access the cache and tag arrays. Bits 10–4 of the PC are used to access columns 0 and 1 of the cache and tag arrays. Towards the end of the cycle, bits 31–11 of the PC are compared to the Instruction Address Tag field in each column's tag entry. A cache hit (meaning that the instruction is in the cache) is detected if the following conditions are true for one of the columns:

- Bits 31–11 of the PC match the Instruction Address Tag field.
- The valid status bit of the accessed word is 1.
- The Instruction Cache Disable (ID) bit of the Configuration Register is 0.
- The P status bit is 1 and the PC address is physical, or the P bit is 0 and the PC address is virtual or a mapped DRAM address.
- The US status bit matches the SM bit in the Current Processor Status Register.
- If the fetch is for a branch target and the target address is a virtual or mapped DRAM address, a TLB hit occurs during address translation or mapping. Address translation or mapping is performed during the execution of the branch, at the same time that the cache is accessed.

If the above conditions do not hold for a block in either column, a cache miss occurs. Sections 8.4 and 8.5 discuss behavior for cache misses.

While fetching instructions from the cache, the processor has sufficient time to check for a hit in the next sequential cache block before it exhausts the supply of instructions in the current block. The processor can transition to the next cache block entry without taking additional cycles to check for a hit in the next block.

8.4 EXTERNAL FETCHING AND CACHE RELOAD

When a cache miss is detected and the cache is enabled (see Section 8.3), the processor attempts to place the missing instructions into the cache by initiating an external instruction fetch. This is called *cache reloading*. If the cache is disabled, the missing instructions are not placed into the cache since the processor does not update a disabled cache. Similarly, the processor does not replace a valid block in a locked column.

In the following discussion, a valid block is a block that contains one or more valid instructions (one or more valid bits are set), and an invalid block is one that contains no valid instructions (no valid bit is set).

8.4.1 Cache Replacement

Whenever a miss is detected, a candidate block is normally selected for replacement and the reloaded instructions placed into the selected block. This occurs unless the miss is caused by a valid bit being 0 in a block that is otherwise valid for the address, in which case the missing instructions are reloaded into the block that is already allocated. When a block does have to be replaced, the replacement algorithm is as follows:

- If one of the blocks accessed during the cache search is invalid, this invalid block is selected for replacement. If both of the columns contain invalid blocks, the block in column 0 is selected.
- If both blocks are valid and neither is locked, the replaced block is randomly chosen.
- If the block in column 0 is locked and valid and the block in column 1 is not locked, the block in column 1 is selected.
- If the entire cache is locked and the blocks in both columns are valid, no block is available for replacement. The instruction fetch is satisfied by external memory and the instruction is not placed into the cache.

8.4.2 Overview of Cache Reload

Once a candidate block is selected, its tag and status bits are set according to the missing address and all valid bits are reset (the valid bits are not reset if the block is already properly allocated and the miss is simply caused by a valid bit of 0). External instruction fetches begin with the instruction that the processor requires and continues until a branch or higher priority external access occurs, or until an instruction is found in the cache. The processor begins executing instructions as soon as the first one is received, and the remainder of the cache reload occurs in parallel with execution. After the first instruction is fetched, subsequent instructions in the block are fetched and written into the cache as they are received from the external memory. The valid bit for a word is set when the word is written (assuming there is no TLB miss on the fetch, in which case the valid bit is not set). If the processor pipeline stalls during prefetching, the instructions received for the rest of the block are placed into the prefetch buffer and remain there until the decode stage can accept them.

If a taken branch occurs during reload or if the memory interface is needed for a higher priority operation (DMA, load miss, store buffer full, etc.), the reload is terminated immediately and the branch is taken or other external access is performed. Reloading may then resume if the next required instruction is not in the cache (reloading may occur for the target instruction in the case of a branch).

8.5 INSTRUCTION PREFETCHING

After the processor starts an external fetch, it may have to continue externally fetching instructions beyond the missing block. The processor prefetches these instructions so that they are requested in advance of execution, giving the external memory ample time to perform the fetches with no wait states if the memory has sufficient bandwidth. This is particularly appropriate for the burst-mode or page-mode memory systems that are anticipated to be used with the Am29240 microcontroller series.

8.5.1 Operation During Prefetching

The processor checks for the presence of the next sequential cache block while servicing a cache miss. Thus, before the fetch of the current block is complete, the processor knows whether or not this next block is present. The processor considers the next block to be present if all instructions in the block are valid. If any instructions are not valid, the processor considers the entire block to be not present and continues the external fetch, allocating the block if necessary by setting the tag field. The processor can initiate a prefetch for the next block as soon as it has initiated all fetches for the current block, unless there is a taken branch in the current block that causes the next block not to be needed.

Normally, prefetching is initiated before all instructions have been received in the current block, to prevent wasted fetch cycles at block boundaries. During prefetching, the processor continues to examine the next block to determine whether or not prefetching should continue. However, if the next block is not present, the cache block is not allocated until it is certain that the current block does not contain a branch that would cause this block to be unneeded. If there is such a branch, the fetch for the next block may be started, but the results of the fetch are discarded.

The processor does not prefetch across 1-Kbyte address boundaries. When the processor encounters a 1-Kbyte address boundary during instruction prefetch, it cancels the prefetch. If the processor needs instructions from the canceled prefetch, this is detected by a subsequent cache miss. The processor performs all steps required to handle the cache miss, including address translation, if applicable, before it establishes another prefetch.

8.5.2 Role of the Prefetch Buffer

During prefetching, the processor requests instructions one at a time and is always able to accept the requested instructions. Instructions fetched externally are placed into the prefetch buffer in the cycle after they are received. From the prefetch buffer, instructions are written into the cache and sent to the decoder. If the decoder cannot accept an instruction because of a pipeline stall, the instruction remains in the prefetch buffer until the stall condition no longer exists. The instruction is retired from the prefetch buffer only after it is sent to the decoder and written to the cache (writing into the cache occurs only if a block has been allocated; that is, if the cache is not locked or contained an invalid block when the miss was detected).

The primary purpose of the prefetch buffer is to allow the processor to get to a convenient and/or efficient point at which to suspend external instruction fetching without the complications of being coupled directly to the processor's decode stage. For example, a load miss waits on the cancellation of an instruction cache reload, causing a pipeline hold until the reload is canceled. During the pipeline hold, the decoder is not available to receive reloaded instructions. When the pipeline hold condition is detected, the processor has three instructions in various stages of fetch. The prefetch buffer is used to store these instructions until they can be written into the cache and/or sent to the decoder. The instructions received during the pipeline hold are written into the cache if the instructions are in the same block as the instructions being sent to the decoder. During the pipeline hold, the next instruction required by the processor is held in the prefetch buffer. This simplifies the operation of the fetcher: it is easier to assume that instructions are always supplied by the prefetch buffer during reload, rather than switching between the prefetch buffer and the cache depending on pipeline holds.

8.5.3 Terminating Instruction Prefetching Because of a Cache Hit

Prefetching causes cache allocation, external fetching, and reloading to continue until the processor determines that the next required block is in the cache. The next required block may be addressed either sequentially or non-sequentially, but this section considers only sequentially-addressed blocks. In this case, the processor knows about the hit at a fixed time with respect to the reload of the current block. In contrast, a non-sequential fetch (branch) can occur at any point during reload.

If the processor detects a cache hit in a sequentially-addressed block, the hit is detected early enough to stop all external fetches for the next block. The processor completes any reload in progress before resuming instruction fetching from the cache.

8.5.4 Terminating Instruction Prefetching Because of a Branch

Terminating an instruction prefetch because of a branch is complicated by several factors. First, the branch can occur at any point during the reload of the current block, because instructions are executed while the block is being reloaded. Second, the target instruction can either hit or miss in the cache. If the target hits, the processor terminates external fetches. If the target misses, the processor must terminate the current fetch and resume a new fetch. Finally, the reload of the current block must be canceled before the target instruction can be fetched.

When a branch is taken during prefetching, in no case is there enough time to stop the prefetch of the next sequentially-addressed block, even though this block is needed only when the branch is the last instruction of the block (because the branch delay instruction is in the next block). Thus, some external memory capacity is taken for unnecessary fetches beyond the branch, and these instructions are discarded even if they are not present in the cache.

8.5.5 Instruction Access and Data Access Collisions

Because the processor includes both an instruction cache and a buffered data cache, it is rare that the external memory interface is needed for an instruction and a data access at the same time. However, since the processor decodes instructions during cache reload, there can be collisions between instruction and data accesses if, during instruction reload, a load misses in the data cache or a store is performed with a full write buffer. This section describes the behavior of the processor in these cases.

If a data access collides with an instruction access, the processor cancels the instruction fetch before servicing the data access. The load or store instruction creating the data access is allowed to complete execution while it is waiting on the reload to be canceled. However, the load or store is held in the write-back stage and subsequent instructions are held in earlier pipeline stages. This permits the external load/store access to begin immediately after the instruction fetch has been canceled.

Once the servicing of the data access is complete, the processor can restart external fetching. This is triggered by the normal mechanisms used to detect cache misses and to start external fetches. If another data access is required before the reload starts (that is, if another load or store immediately follows the first load or store in the instruction stream), the second load or store is performed before the reload.

If a load or store is the delay instruction of a branch whose target misses in the cache, the fetch for the target instruction of the branch is completed before an external access for the load or store is performed.

8.6 CACHE INVALIDATION

If the instruction cache is accessed with translated or mapped DRAM addresses, the cache must be flushed of all contents whenever the translation or mapping is changed in a way that affects the mapping of instructions in the cache. Flushing is accomplished by resetting all valid bits of all cache blocks. The valid bits are reset in a single cycle by a processor reset and by the execution of an Invalidate (INV) or Interrupt Return and Invalidate (IRETINV) instruction specifying that the instruction cache should be invalidated. (These instructions can invalidate either the instruction cache, the data cache, or both.) The INV and IRETINV instructions must be executed in the Supervisor mode to have the effect of flushing the cache.

When an INV instruction is executed, the processor does not reset the valid bits until the next branch *or* the next cache-block boundary, whichever comes first. If the INV is the last instruction in a block, the block boundary at which invalidation occurs is the end of the next block. This approach allows the processor pipeline to complete the execution of the instruction in decode when the INV instruction is executed, without forcing the instruction to be invalidated in the pipeline and refetched externally.

The processor does not invalidate locked cache blocks unless the cache is also disabled.

CHAPTER

9 DATA CACHE

The Am29240 and Am29243 microcontrollers incorporate a data cache that satisfies most processor data references by the end of the execute stage of the processor pipeline. This chapter describes the data cache and the mechanisms used to load and access this cache. This chapter also describes the behavior of the write buffer.

9.1 DATA CACHE OVERVIEW

The Am29240 and Am29243 microcontrollers have a 2-Kbyte, two-way set-associative data cache (Figure 9-1). The block size is four words (16 bytes). Individual bytes and half-words may be written within a word. The data cache stores the most recent data fetched by the processor from the DRAM or ROM address regions. Accesses to internal peripherals and the PIA regions are not cached.

The data cache is accessed by physical address. The cache is accessed in the execute stage of the pipeline, and the latency of a load that hits in the cache is one cycle. Consequently, data from a load that hits in the cache is available to the instruction that immediately follows the load without causing a pipeline hold. Address translation or mapping, if applicable, is performed at the same time as the cache access so that the physical address is available at the end of the cycle for the cache tag compare.

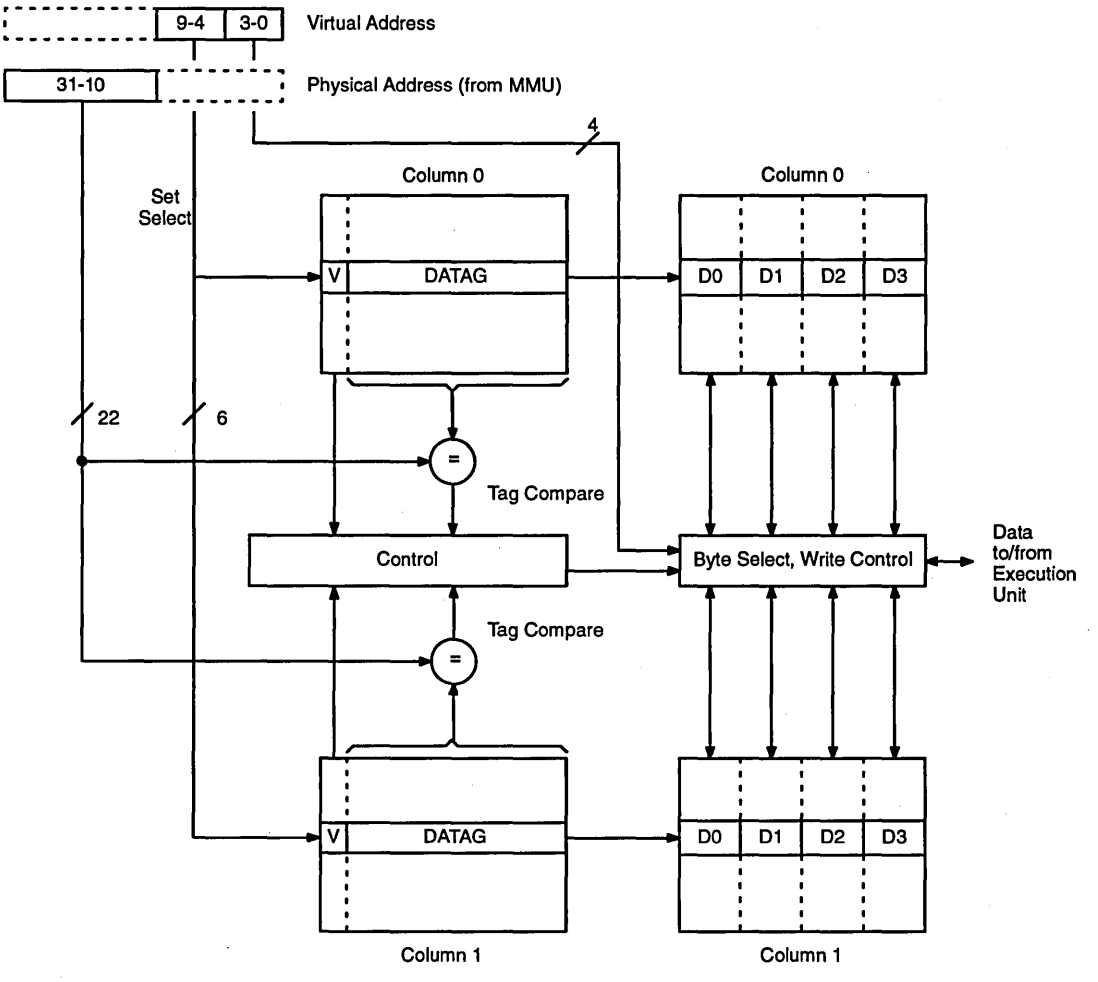
The data cache implements a buffered write-through policy for stores, with no write allocation. Every store that is performed in the cache is also performed in the external memory. However, the processor does not wait on the store to complete in the external memory before it proceeds. The store request is simply placed into a two-word write buffer and is performed at a later time when the memory interface is available. A store that misses in the cache does not cause a cache block to be allocated; the store is performed in the main memory via the write buffer. Dependency logic ensures that processor load misses do not depend on incomplete stores that are in the write buffer.

The data cache is enabled and disabled by the Data Cache Disable (DD) bit of the Configuration Register. If the data cache is enabled, data loads and stores may be performed by the cache. If the data cache is disabled, loads and stores are performed in the external instruction/data memory and the cache does not reload data. However, if the cache is disabled when it contains valid data, it retains this data. The write buffer is not used when the cache is disabled. A disabled cache can be invalidated by an INV or IRETINV instruction that specifies the data cache.

To keep critical data in the cache, or to allow the cache to appear as a small data memory, the data cache can be locked by the Data Cache Lock (DL) bit of the Configuration Register. When the data cache is locked, a block is not available for replacement if it is valid. However, a locked block may be allocated if it is invalid—this allows critical data to be placed into the cache simply by loading the data. Also, a locked block cannot be invalidated (unless the cache is also disabled—the disable overrides the lock).

The data cache never contains partially-valid blocks, so it either contains the entire four words of a block or stores nothing. Thus, if the valid bit of a cache block is 1, all

Figure 9-1 Data Cache Organization



data words in the block are valid. This simplifies the cache and improves its ability to take advantage of spatial locality, especially for sequential data access patterns. During reload, the valid bit of a block is set when the final word in the block is written into the cache.

The data cache can be invalidated in a single cycle by an INV or IRETINV instruction that specifies the data cache (these instructions can invalidate either the instruction cache, the data cache, or both).

9.2 ACCESSING CACHE FIELDS

Each data-cache block is accessible via the Cache Interface Register (see Section 8.2.1) and Cache Data Register (see Section 8.2.2). The Cache Interface Register contains a pointer to the accessed block and specifies the accessed field. The Cache Data Register is used to transfer data to and from the cache.

Figure 9-2 Data Cache Block Organization

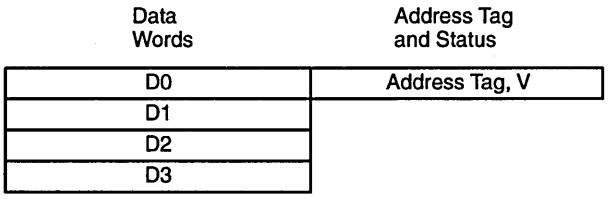
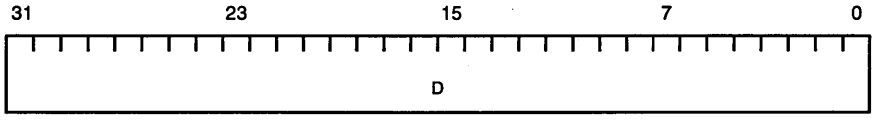


Figure 9-2 shows the organization of an individual data cache block. There are 128 such blocks in the cache, organized as two columns of 64 blocks each. For access, a particular column and block are selected by 7 bits of the CPTR field (bits 10–4 of the Cache Interface Register; bit 10 selects the column). The accessed field in the block is specified by the Cache Field Select (FSEL) field. When a data word is accessed, the word is further selected by the two least significant bits of the CPTR field (bits 3–2 of the Cache Interface Register). This section describes the fields within the data cache and how they relate to data in the Cache Data Register upon access.

9.2.1 Data Words

Figure 9-3 shows the placement of data in the Cache Data Register, for reading or writing the data cache. In this case, the data takes up the entire register. It is not possible to write individual bytes or half-words via the Cache Data Register.

Figure 9-3 Data Word in Cache Data Register

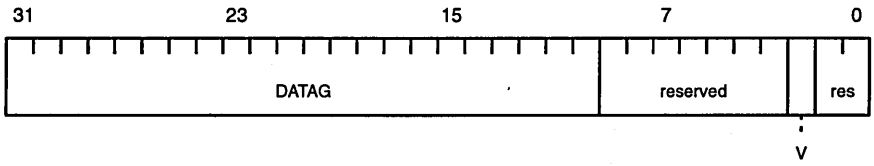


Bits 31–0: Data (D)—This is the 32-bit data word that is read from or written into the data cache.

9.2.2 Address Tag and Status Information

Figure 9-4 shows the placement of the tag and status fields in the Cache Data Register, for reading or writing the data cache.

Figure 9-4 Data Address Tag and Block Status in Cache Data Register



Bits 31–10: Data Address Tag (DATAG)—The DATAG field specifies which address in ROM or DRAM is mapped by the cache block.

Bits 9–3: Reserved

Bit 2: Valid (V)—The V bit is 1 if the cache block contains valid data, otherwise it is 0. All valid bits in the cache are cleared in a single cycle by a processor reset and by the execution of an INV or IRETINV that specifies the data cache.

Bits 1–0: Reserved

9.3 CACHE ACCESSES

This section describes the normal operation of the data cache. The following section describes the actions taken by the processor when data is not found in the cache.

When a load or store is executed, the processor accesses the cache data and tag/status arrays using bits 9–4 of the load or store address. Note that since the minimum virtual page size is 1 Kbyte, bits 9–4 of the address are not affected by address translation. A cache hit (meaning the data is in the cache) is detected if the following conditions are true for one of the columns:

- Bits 31–10 of the physical address match the Data Address Tag field.
- The valid status bit is set.
- The Data Cache Disable (DD) bit of the Configuration Register is 0.
- For a virtual or mapped DRAM address, a TLB hit occurs during address translation or mapping.

If the above conditions do not hold for a block in either column, a cache miss occurs. Section 9.4 discusses behavior for cache misses. Address translation (including DRAM mapping), if applicable, is overlapped with cache access. If address translation is performed in parallel with cache access, translation must be successful for the cache hit to be successful. If address translation is unsuccessful (because a TLB miss occurs, for example), a trap occurs to handle the unsuccessful translation and the cache access is irrelevant.

If a cache hit is detected on a load, the cache supplies the data by the end of the execute stage of the load. The data can be forwarded directly to the instruction following the load, avoiding a pipeline stall.

If a cache hit is detected on a store, the data is written into the cache in the same cycle. The cache is available for a subsequent load or store in the next cycle.

9.4 EXTERNAL ACCESSES AND CACHE RELOAD

When a cache miss is detected during a load, when the load is from the ROM or DRAM region as determined by the physical address, and when the data cache is enabled, the processor allocates a cache block to receive the missing data and reloads the data (unless the cache is locked and the blocks in both columns are valid). The cache reloads a locked block only if the block is not valid. The cache does not allocate a block on a store miss: the store request is simply placed into the write buffer to be written to memory. The cache does not store internal peripheral or PIA data.

Whenever a miss is detected on a ROM or DRAM load, a candidate block is normally selected for replacement and the reloaded data is placed into the selected block. The replacement algorithm is as follows:

- If one of the blocks accessed during the cache search is invalid, this invalid block is selected for replacement. If both of the columns contain invalid blocks, the block in column 0 is selected.
- If both blocks are valid and neither is locked, the replaced block is chosen at random.
- If the block in column 0 is locked and valid and the block in column 1 is not locked, the block in column 1 is selected.
- If the entire cache is locked and the blocks in both columns are valid, no block is available for replacement. The data access is satisfied by external memory and the data is not placed into the cache.

Once a candidate block is selected, its tag is set according to the missing address and the valid bit is reset. Data-cache reloading always fills an entire cache block. The reload begins with the data word that is required by the processor and wraps at the end of the block to fill the remainder of the block. For example, if a miss occurs when the processor attempts to access the third word in the block, the third word is loaded first, followed by the fourth, first, then second word. Reloading from DRAM can occur at a rate of up to one word per cycle using page-mode accesses, even though reload addressing is non-sequential.

Reloading in this manner minimizes the latency of load misses. The word required by the processor is received as soon as possible, before any other words in the block. When received, the required data is forwarded to the execution unit. Instruction execution proceeds in parallel with reloading the remainder of the block.

Cache reloading is buffered: reload data is placed into a four-word buffer before being written into the cache. This frees the cache to service subsequent data accesses during reload, as long as these accesses hit in the cache. The entire cache block is written from the reload buffer in a single cycle, at a time when the processor does not need to access the cache.

Because the cache continues to service processor requests during a reload, it is possible that a second cache miss is detected during reload. If this occurs, the processor enters the Pipeline Hold mode until the reload is complete. The cache is then accessed a second time to ensure that the miss has not been eliminated by the just completed reload. If a cache miss still occurs, the second reload begins.

Any higher priority access such as a DMA transfer can pre-empt the cache reload. Once the higher priority access is complete, the reload continues at the point of pre-emption.

9.5 WRITE BUFFER

The data cache implements a write-through policy for stores, meaning that the data of each store is written to the external memory regardless of whether the store hits in the cache (if a store hits in the cache, it is written into both the cache and into external memory). The write-though policy ensures that data in the external memory is consistent with data in the cache. If there were no write buffer, the write-through policy would reduce processor performance because stores cannot take advantage of a cache hit. The performance of stores is determined by the speed of the external memory.

To reduce the performance impact of the write-through policy, the memory interface of the data cache includes a two-entry write buffer. This decouples the processor from the performance of memory writes by latching a store request in a single cycle and

allowing the processor to proceed beyond the store. The write buffer then later performs the store in the external memory, usually when the memory interface is otherwise idle. The processor is rarely held up by stores because stores typically represent 5–10% of dynamically executed instructions and each DRAM store takes a maximum of six processor cycles (three MEMCLK cycles when the processor operates at the INCLK frequency). Thus, the maximum average store execution rate to DRAM is in the range of 0.3–0.6 cycles for each store instruction, well below the cycle per store taken to place the store request in the write buffer.

The write buffer contains store requests only for cacheable stores; that is, for DRAM and ROM space stores. Other stores are performed directly to the internal peripheral or PIA region. However, for proper ordering of stores, a noncacheable load or store is not performed until the write buffer is empty. Also, the write buffer is not used if the data cache is disabled.

The buffer is organized as a two-entry FIFO. Each entry of the FIFO contains a 32-bit physical address (after address translation or DRAM mapping, if applicable), 32-bit data, bits to indicate the store data width, and a valid bit. When the processor performs a cacheable store and there is an available write buffer entry, the store address, data, and data width are placed into the available entry, and the valid bit is set. The head entry is used if it is available, otherwise the tail entry is used. The valid bit being 1 indicates that an entry has an active store request.

Writes to memory occur only from the head entry. The write buffer services an active request whenever the memory interface is free—the write buffer has the lowest priority of any other request to use the memory interface. After the external write begins, the write buffer is freed as soon as the address and data are no longer needed: the buffer is free on the last cycle of a DRAM store (during $\overline{\text{RAS}}$ precharge) and on the cycle after the final cycle on a store to the ROM space. The head entry can be used immediately, in the cycle that it becomes free, to receive a request from the tail entry or the processor. The valid bit can remain set in this case though the entry latches a new address, data, and data width. If an active tail entry is moved to the head entry, the tail entry likewise can receive a new processor request immediately and its valid bit can remain set.

If the processor attempts a cacheable store and neither write buffer entry is available, a pipeline stall occurs until the tail entry is available. In this situation, the priority of the request at the head of the write buffer is elevated to the priority of a processor data request, taking priority over processor instruction fetches. When the store at the head of the buffer has been completed, the tail entry moves to the head entry, the processor store request is written into the tail entry, and the processor pipeline advances. Note that for a DRAM store, the processor proceeds before the external DRAM cycle is complete (during the $\overline{\text{RAS}}$ precharge cycle).

To reduce load latency, data cache reload requests normally bypass store requests in the write buffer and do not wait for the write buffer to be emptied. Consequently, when a load miss occurs, the block to be loaded may not be current because of an uncompleted store in the write buffer. The processor detects this dependency by comparing bits 9–4 of the load address to bits 9–4 of the address of each active request in the write buffer. If there is no match, the reload is allowed to proceed and the write buffer continues to hold the store requests. If there is a match, the reload is *possibly* dependent on the store request, and the write buffer completes the stores necessary to remove the dependency before the reload proceeds (if the tail entry creates the dependency, the head entry is serviced before the tail entry).

Dependency checking also occurs for single loads that are performed because the data cache is locked and no entry is available for reload. In this case, the single load is held until the potential dependency is removed from the write buffer.

When the processor performs serialization, all active requests in the write buffer are performed before the processor proceeds. For example, the processor empties the write buffer before it takes an interrupt.

9.6 CACHE INVALIDATION

External agents may write the processor's memory using DMA or the $\overline{\text{GREQ}}/\overline{\text{GACK}}$ protocol (see Section 14.6). The data cache does not maintain coherency with writes that are not performed by the processor. Because of this, the cache must be flushed of all contents before the processor attempts to read locations for which stale data may exist in the data cache. Flushing is accomplished by resetting the valid bits of all cache blocks. The valid bits are reset in a single cycle by a processor reset and by the execution of an Invalidate (INV) or Interrupt Return and Invalidate (IRETINV) instruction that specifies the data cache. The INV and IRETINV instructions must be executed in the Supervisor mode to have the effect of flushing the cache.

When an INV or IRETINV instruction is executed, the processor resets the valid bits in the data cache so that the next cache access does not see any valid bit set. Because the cache is write-through, invalidating the cache does not cause any modifications to be lost. All modifications are either in memory or in the write buffer when invalidation occurs, and invalidation does not affect the write buffer.

The processor does not invalidate locked cache blocks unless the cache is also disabled.

9.7 LOCK ACCESSES

External agents may write an interlock into memory using DMA or the $\overline{\text{GREQ}}/\overline{\text{GACK}}$ protocol (see Section 14.6). Because the data cache does not maintain coherency with writes performed by external agents, the processor might not access an updated value written by an external agent if the processor uses a normal load to access the interlock. However, the LOADL and LOADSET instructions operate in a way that guarantees that the most recent value is obtained. The LOADL and LOADSET instructions bypass the data cache and load directly from the external memory. If a cache hit is detected during the execution of these instructions, the associated cache block is invalidated.

CHAPTER

10 SYSTEM OVERVIEW

The Am29240 microcontroller series 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 Am29240 microcontroller series.

10.1 SIGNAL DESCRIPTION

The Am29240 microcontroller series uses 154 pins for signal inputs and outputs; however, each of the Am29240, Am29245, and Am29243 microcontrollers defines pins associated with non-supported features as no-connects (see Section 10.1.12).

10.1.1 Clocks

INCLK — **Input Clock (input)**
This is an oscillator input at twice the system operating frequency. The processor operates either at the system operating frequency or at the INCLK frequency, as controlled by the TBO bit in the Configuration Register. The processor can operate at the INCLK frequency only if MEMCLK (see below) is an output. INCLK can be driven at TTL levels.

MEMCLK — **Memory Clock (input/output)**
This is either a clock output or an input from an external clock generator, as determined by the MEMDRV input. It operates at the system operating frequency, which is half of the INCLK frequency. Most processor inputs and outputs are synchronous to MEMCLK. MEMCLK must be driven with CMOS levels. MEMCLK must be an output if the processor operates at the INCLK frequency.

MEMDRV — **MEMCLK Drive Enable (input, internal pull-up resistor)**
This input determines whether MEMCLK is an output or an input. If this pin is High, the processor generates a clock on the MEMCLK output. If this pin is Low, the processor accepts a clock generated by the system on the MEMCLK input. This signal is tied High through an internal pull-up resistor so the signal can be left unconnected to configure MEMCLK as an output.

10.1.2 Processor Signals

A23–A0 — **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 A14–A1. A2–A0 are also used to provide a clock to an optional burst-mode EPROM.

ID31–ID0 — **Instruction/Data Bus (bidirectional, synchronous)**
The Instruction/Data Bus (ID Bus) transfers instructions to, and data to and from the processor.

IDP3–IDP0 — **Instruction/Data Parity (bidirectional, synchronous)**
If parity checking is enabled by the PCE bit of the DRAM Control Register, IDP3–IDP0 are parity bits for the ID Bus during DRAM

accesses. IDP3 is the parity bit for ID31–ID24, IDP2 is the parity bit for ID23–ID16, and so on. If parity is enabled, the processor drives IDP3–IDP0 with valid parity during DRAM writes, and expects IDP3–IDP0 to be driven with valid parity during DRAM reads. These signals are not supported on the Am29240 and Am29243 microcontrollers.

$\overline{\text{WAIT}}$ **Add Wait States (input, synchronous, internal pull-up)**
External accesses are normally timed by the processor. 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, internal pull-up)**
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, internal pull-up)**
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}}$3–$\overline{\text{INTR}}$0 **Interrupt Requests 3-0 (input, asynchronous, internal pull-ups)**
These inputs generate prioritized interrupt requests. The interrupt caused by $\overline{\text{INTR}}$0 has the highest priority, and the interrupt caused by $\overline{\text{INTR}}$3 has the lowest priority. The interrupt requests are masked in prioritized order by the Interrupt Mask field in the Current Processor Status Register 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}}$1–$\overline{\text{TRAP}}$0 **Trap Requests 1–0 (input, asynchronous, internal pull-ups)**
These inputs generate prioritized trap requests. The trap caused by $\overline{\text{TRAP}}$0 has the highest priority. These trap requests are disabled by the DA bit of the Current Processor Status Register. These signals have special hardening against metastable states, allowing them to be driven with slow-transition-time signals.

CNTL1–CNTL0 **CPU Control (input, asynchronous, internal pull-ups)**
These inputs control the processor mode, as follows:

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

STAT2–STAT0 **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	Interrupt/Trap Vector Fetch (vector valid)
0	1	0	Load Test Instruction Mode, Halt/Freeze
0	1	1	Non-sequential instruction fetch (internal cache hit, or external access and instruction valid)
1	0	0	External data access (data valid)
1	0	1	External sequential instruction access (instruction valid)
1	1	0	Internal peripheral access (data valid)
1	1	1	Idle or data/instruction not valid

The status conditions are prioritized in the order listed, with STAT2–STAT0=000 having highest priority. The STAT2–STAT0 outputs are changed at the end of every processor cycle to indicate the processor status in the previous cycle. Thus, if the processor operates at twice the system frequency, the STAT2–STAT0 outputs change on both the rising and falling edge of MEMCLK. If the processor operates at twice the system frequency, the status indication related to an external access (such as an external instruction access) appears in the first half-cycle of MEMCLK (MEMCLK High) just after the completion of the external access; in the second half-cycle of this MEMCLK cycle (MEMCLK Low), the processor's internal condition is indicated. If the processor operates at the system frequency, the status indication related to an external access appears for the entire MEMCLK cycle following the completion of the access.

The processor can be placed into a slave configuration that allows tracing of a master processor. In this tracing configuration, certain status encodings are changed as follows:

Tracing Configuration

STAT2	STAT1	STAT0	Condition
1	0	0	Load access (internal access and cache hit, or external access and data valid)
1	0	1	Store access (internal access and cache hit, or external access and data valid)
1	1	0	Return from interrupt (first target instruction cache hit or valid on ID Bus)
— all others —			Same as master processor

$\overline{\text{TRIST}}$ **Three-State Control (input, asynchronous, pull-up resistor)** This input is asserted to force all processor outputs into the high-impedance state. This signal is tied High through an internal pull-up resistor.

10.1.3 ROM Interface

$\overline{\text{ROMCS3}}$–$\overline{\text{ROMCS0}}$
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. $\overline{\text{ROMCS3}}$ 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.

$\overline{\text{RSWE}}$
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 $\overline{\text{RESET}}$.

10.1.4 DRAM Interface

$\overline{\text{RAS3}}$–$\overline{\text{RAS0}}$
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. $\overline{\text{RAS3}}$ starts an access in DRAM Bank 3, and so on. These signals also are used in other special DRAM cycles.

$\overline{\text{CAS3}}$–$\overline{\text{CAS0}}$
Column Address Strobes, Byte 3–0 (output, synchronous)
A High-to-Low transition on these signals causes the DRAM selected by $\overline{\text{RAS3}}$–$\overline{\text{RAS0}}$ 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. $\overline{\text{CAS3}}$ is the column address strobe for the DRAMs, in all banks, attached to ID31–ID24. $\overline{\text{CAS2}}$ is for the DRAMs attached to ID23–ID16, 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.

$\overline{TR/OE}$ **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.

10.1.5 Peripheral Interface Adapter (PIA)

$\overline{PIACS5}$–$\overline{PIACS0}$
Peripheral Chip Selects, Regions 5-0 (output, synchronous)
These signals are used to select individual peripheral devices. DMA channels may be programmed to use dedicated chip selects during an external peripheral access.

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

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

10.1.6 DMA Controller

DREQD–DREQA
DMA Request D through A (input, asynchronous, pull-up resistors)
These signals request an external transfer on a DMA channel. DMA requests are not dedicated to a particular DMA channel—each channel specifies which request line, if any, it is using. Only one channel at a time can use either DREQD, DREQC, DREQB, or DREQA, and this channel acknowledges a transfer using the respective $\overline{DACK}$D through $\overline{DACK}$A signal. 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.

The DMA request acknowledge pairs DREQA/$\overline{DACK}$A and DREQB/$\overline{DACK}$B correspond to the Am29200 microcontroller signals DREQ0/$\overline{DACK}$0 and DREQ1/$\overline{DACK}$1, respectively. The pin placement reflects this correspondence, and a processor reset dedicates these request/acknowledge pairs to DMA channels 0 and 1, respectively. This permits backward-compatible upgrade to an Am29200 microcontroller. The $\overline{DREQ}$D and $\overline{DREQ}$C signals are supported on the Am29240 and Am29243 microcontrollers only.

$\overline{DACK}$D–$\overline{DACK}$A
DMA Acknowledge D through A (output, synchronous)
These signals acknowledge an external transfer on a DMA channel. DMA acknowledgements are not dedicated to a particular DMA channel—each channel specifies which acknowledge line, if any, it is using. Only one channel at a time can use either $\overline{DACK}$D, $\overline{DACK}$C, $\overline{DACK}$B, or $\overline{DACK}$A, and the same channel uses the respective DREQD through DREQA signal for transfer requests. DMA transfers can occur to and from internal peripherals independent of these acknowledgements. The $\overline{DACK}$D and $\overline{DACK}$C signals are supported on the Am29240 and Am29243 microcontrollers only.

TDMA — **Terminate DMA (input/output, synchronous)**
This signal is either an input or an output as controlled by the corresponding DMA Control Register. As an input, this signal can be asserted during an external DMA transfer (non-fly-by) to terminate the transfer after the current access. The TDMA input is ignored during fly-by transfers. As an output, this signal is asserted to indicate the final transfer of a sequence.

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

To support a hardware-development system, $\overline{\text{GREQ}}$ 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 processor's ROM or DRAM, and that the device should provide an address.

The processor can be placed into a slave configuration that allows tracing of a master processor. In this configuration, $\overline{\text{GACK}}$ is used to indicate that the processor pipeline was held during the previous processor cycle.

10.1.7 I/O Port

PIO15–PIO0 — **Programmable Input/Output (input/output, asynchronous)**
These signals are available for direct software control and inspection. PIO15–PIO8 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.

The PIO signals are sampled during a processor reset. After reset, the sampled value is held in the PIO Input Register. This sampled value is supplied the first time this register is read, unless the read is preceded by write to the PIO Input Register or by a read or write of any other PIO register. This may be used to indicate system configuration information to the processor during a reset.

10.1.8 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 processor.

$\overline{\text{PBUSY}}$ — **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 processor 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 an external 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. Then, the buffer drives data to the host.

10.1.9 Serial Ports

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.

TXDA **Transmit Data, Port A (output, asynchronous)**
This output is used to transmit serial data from Serial Port A.

RXDA **Receive Data, Port A (input, asynchronous)**
This input is used to receive serial data to Serial Port A.

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

$\overline{\text{DTRA}}$ **Data Terminal Ready, Port A (input, asynchronous)**
This indicates to the processor that the host is ready to transmit or receive data on Serial Port A.

TXDB **Transmit Data, Port B (output, asynchronous)**
This output is used to transmit data from Serial Port B. This signal is supported on the Am29240 and Am29243 microcontrollers only.

RXDB **Receive Data, Port B (input, asynchronous)**
This input is used to receive data to Serial Port B. This signal is supported on the Am29240 and Am29243 microcontrollers only.

10.1.10 Video Interface (Am29240 and Am29245 Microcontrollers Only)

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.

10.1.11 JTAG 1149.1 Boundary Scan Interface

TCK — **Test Clock Input (asynchronous input, pull-up resistor)**
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, pull-up resistor)**
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, pull-up resistor)**
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 (three-state output, synchronous to TCK)**
This output supplies data from the test logic to an external destination. It changes on the falling edge of TCK. It is in the high-impedance state except when scanning is in progress.

$\overline{\text{TRST}}$ — **Test Reset Input (asynchronous input, pull-up resistor)**
This input asynchronously resets the Test Access Port. If $\overline{\text{TRST}}$ is not driven, it appears High internally. $\overline{\text{TRST}}$ must be tied to $\overline{\text{RESET}}$, even if the Test Access Port is not being used.

10.1.12 Pin Changes for Am29240, Am29245, and Am29243 Microcontrollers

The Am29240, Am29245, and Am29243 microcontrollers define certain pins as no-connects.

- The Am29240 microcontroller defines the IDP3–IDP0 signals as no-connects.
- The Am29245 microcontroller defines the following signals as no-connects: IDP3–IDP0, DREQD–DREQC, $\overline{\text{DACKD}}$–$\overline{\text{DACKC}}$, TXDB, and RXDB.
- The Am29243 microcontroller defines the following signals as no-connects: LSYNC, VCLK, VDAT, and PSYNC.

10.2 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 12.2.8)
2. DMA Channel 0 transfer
3. DMA Channel 1 transfer
4. DMA Channel 2 transfer
5. DMA Channel 3 transfer
6. Memory access request by an external device (see Section 14.6)
7. Processor DRAM, PIA, or ROM access for data (including load misses and stores with a full write buffer)
8. Processor DRAM or ROM access for an instruction
9. Data cache write-through from the write buffer

DMA transfers that do not use fly-by 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 require two or four accesses (a narrow memory is 8 or 16 bits wide); 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 and may be concurrent with other accesses, refresh cycles are not prioritized in the above list.

10.3 SYSTEM ADDRESS PARTITION

All addresses are in the processor's instruction/data memory address space. The I/O address space is unused. The processor's address space is partitioned as shown in Table 10-1. The MMU can translate virtual addresses into addresses in any one of these regions. An access to any unimplemented address or address range has an unpredictable effect on processor operation.

Table 10-1 Internal Peripheral Address Assignments

Address Range (hexadecimal)	Selection
00000000–03FFFFFF	ROM Banks (all)
40000000–43FFFFFF	DRAM Banks (all)
50000000–53FFFFFF	Mapped DRAM Banks (MMU translation)
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

10.4 INTERNAL PERIPHERALS AND CONTROLLERS

Internal peripheral registers are selected by offsets from address 80000000 (hexadecimal). The address assignment of the various internal peripherals and controllers is shown in Table 10-2.

Nearly all registers are read/write and are 32 bits in length. However, a few register bits are read only, and bits in the Interrupt Control Register are reset-only. It is not possible to perform writes on individual bytes or halfwords. Unimplemented bits are read as zeros and should be written with zeros to ensure future compatibility.

Three registers in the Am29240 microcontroller series have alternates, provided for compatibility. The following summary shows the preferred and alternate addresses for each of these registers.

Register	Preferred Address	Alternate Address
DMA0 Address Tail Register	80000070	80000036
DMA0 Count Tail Register	8000003C	8000003A
Parallel Port Status Register	800000C8	800000C1

The alternate DMA0 Address Tail Register and the alternate DMA0 Count Tail Register allow write-only access for compatibility with the Am29200 and Am29205 microcontrollers. These two registers are supported for backwards compatibility and should not be used for new designs. The DMA0 Address Tail Register (address 80000070) and DMA0 Count Tail Register (address 8000003C) should be used instead.

The alternate Parallel Port Status Register is also provided for compatibility with the Am29200 and Am29205 microcontrollers. This register should not be used for new designs. The Parallel Port Status Register (address 800000C8) should be used instead.

Table 10-2 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
Peripheral Interface Adapter	80000020	PIA Control Register 0
	80000024	PIA Control Register 1
Interrupt Controller	80000028	Interrupt Control Register
	8000002C	Interrupt Mask Register
DMA Channel 0	80000030	DMA0 Control Register
	80000034	DMA0 Address Register
	80000036♦	DMA0 Address Tail Register (write only)
	80000038	DMA0 Count Register
	8000003A♦	DMA0 Count Tail Register (write only)
	8000003C	DMA0 Count Tail Register
DMA Channel 1	80000040	DMA1 Control Register
	80000044	DMA1 Address Register
	80000048	DMA1 Count Register
	8000004C	DMA1 Count Tail Register
DMA Channel 2	80000050	DMA2 Control Register
	80000054	DMA2 Address Register
	80000058	DMA2 Count Register
	8000005C	DMA2 Count Tail Register
DMA Channel 3	80000060	DMA3 Control Register
	80000064	DMA3 Address Register
	80000068	DMA3 Count Register
	8000006C	DMA3 Count Tail Register
DMA Address Queue	80000070	DMA0 Address Tail Register
	80000074	DMA1 Address Tail Register
	80000078	DMA2 Address Tail Register
	8000007C	DMA3 Address Tail Register
Serial Port A	80000080	Serial Port A Control Register
	80000084	Serial Port A Status Register
	80000088	Serial Port A Transmit Holding Register
	8000008C	Serial Port A Receive Buffer Register
	80000090	Baud Rate A Divisor Register
Serial Port B	800000A0	Serial Port B Control Register
	800000A4	Serial Port B Status Register
	800000A8	Serial Port B Transmit Holding Register
	800000AC	Serial Port B Receive Buffer Register
	800000B0	Baud Rate B Divisor Register
Parallel Port	800000C0	Parallel Port Control Register
	800000C1♦	Parallel Port Status Register (alternate)
	800000C4	Parallel Port Data Register
	800000C8	Parallel Port Status 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

Note: *♦ These registers are supported for backwards compatibility with the Am29200 and Am29205 microcontrollers and should not be used for new designs. See Section 10.4.*

CHAPTER

11 ROM CONTROLLER

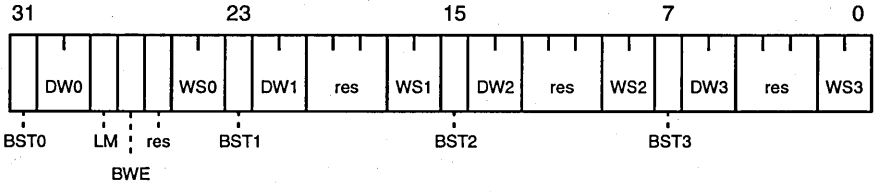
The ROM Interface accommodates up to four banks of ROM that appear as a contiguous memory. Each bank is individually configurable in width and timing. This chapter describes the operation of the ROM controller.

11.1 PROGRAMMABLE REGISTERS

11.1.1 ROM Control Register (RMCT, Address 80000000)

The ROM Control Register (Figure 11-1) controls the access of ROM Banks 0 through 3.

Figure 11-1 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.

Bit 27: Byte Write Enable (BWE)—This bit controls whether or not the $\overline{CAS}3$–$\overline{CAS}0$ signals are used as byte strobes during writes to the ROM address space. If BWE=0, the $\overline{CAS}3$–$\overline{CAS}0$ signals are not used during ROM writes (unless there is a hidden refresh at the same time). If BWE=1, the $\overline{CAS}3$– $\overline{CAS}0$ signals are used as byte strobes during a ROM write (and hidden refresh cannot occur during a ROM read or write).

Bit 26: Reserved

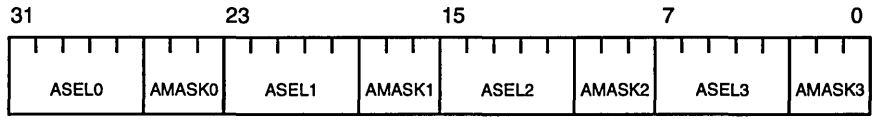
Bits 25–24: Wait States, Bank 0 (WS0)—This field specifies the number of wait states in a ROM access: that is, the number of cycles in addition to one cycle required to access the ROM. Zero-wait-state cycles are supported only for non-burst-mode ROM reads. Writes to the ROM address space have a minimum of one wait state, even when wait states are programmed as zero.

Other bits of this register have a definition similar to BST0, DW0, and WS0 for ROM Banks 1 through 3.

11.1.2 ROM Configuration Register (RMCF, Address 80000004)

The ROM Configuration Register (Figure 11-2) 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 11-2 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.

11.1.3 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.

11.2 ROM ACCESSES

11.2.1 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 if the banks are to be contiguous. 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.

11.2.2 Simple ROM Accesses

Figure 11-3 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 11-4 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).

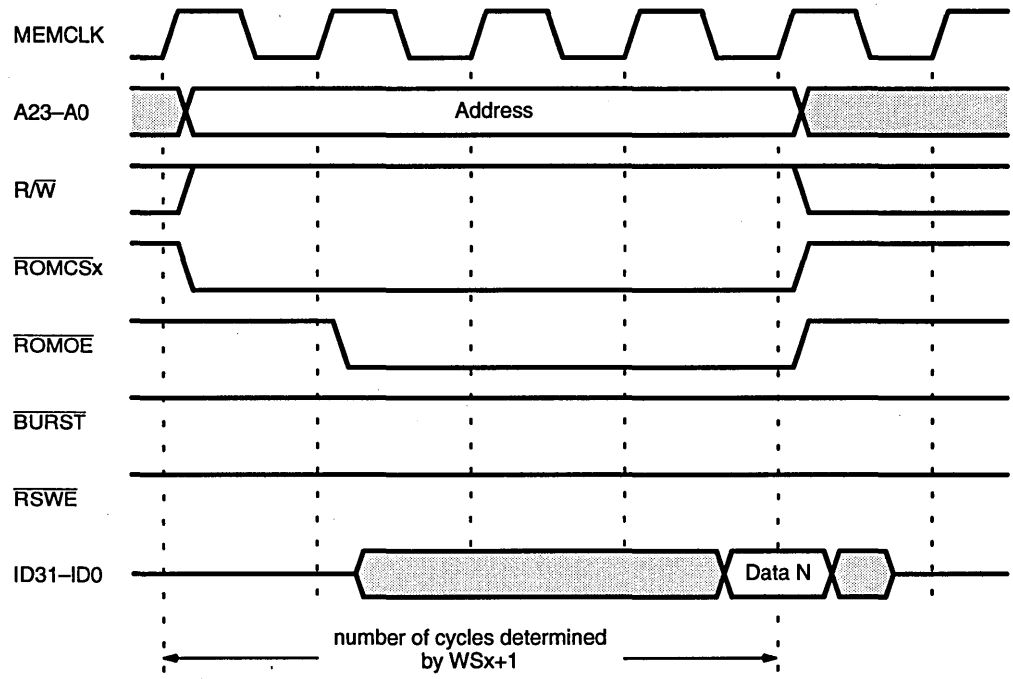
11.2.3 Narrow ROM Accesses

A narrow ROM is one that is less than 32 bits wide. The Am29240 microcontroller series 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 ID31–ID24. A 16-bit-wide ROM is attached to ID31–ID16 and ignores A0. A 32-bit ROM is attached to ID31–ID0 and ignores A1–A0. A narrow ROM can respond to any read access, but the ROM must be at least 16 bits wide to respond to writes. Writes to 8-bit memories are not supported and may provide unreliable results.

11.2.3.1 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 processor 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 A1–A0 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.

Figure 11-3 Simple ROM Read Cycle



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

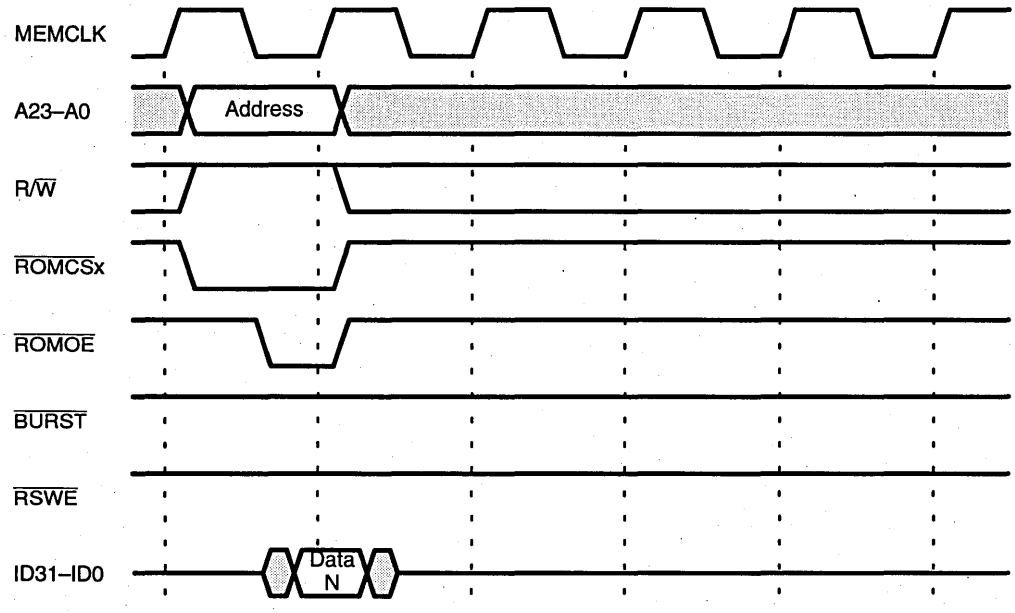
Note that a trap on unaligned access is available and may be used to correct such accesses.

11.2.3.2 16-Bit Narrow Accesses

If the processor expects a word on a read, and a narrow ROM is 16 bits wide, the processor 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 A1–A0 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 processor assembles the final word by placing the first received half-word in the high-order half-word position of the word, and the second received half-word in the low-order half-word position.

Figure 11-4 Simple ROM Read Cycle—Zero Wait States



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.

11.2.4 Writes to the ROM Space

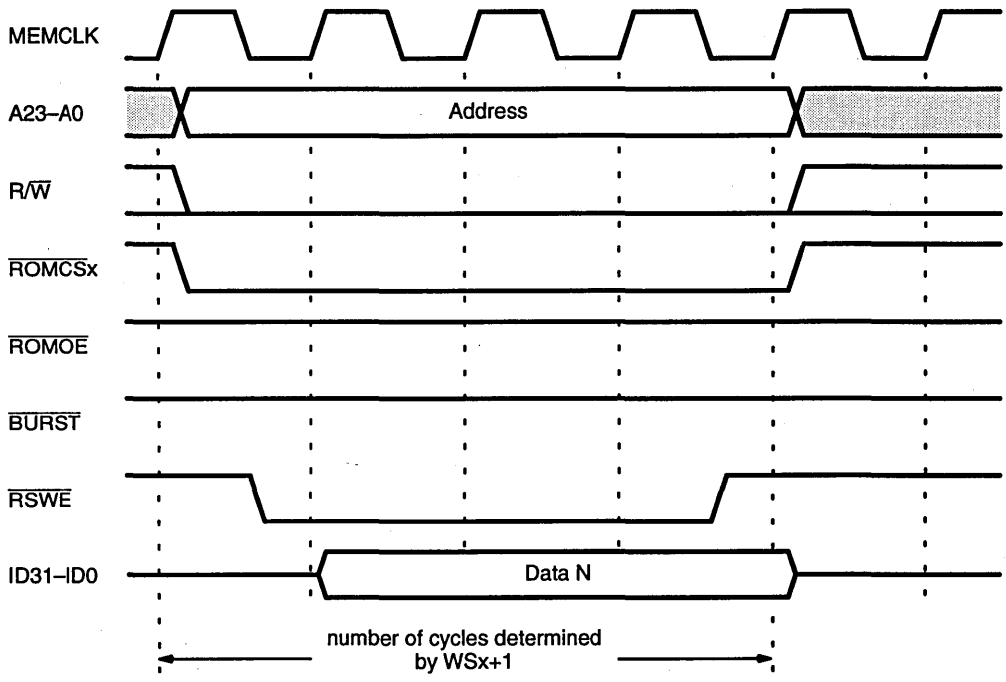
11.2.4.1 Simple Writes

Figure 11-5 shows the timing of a simple 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 of processor limitations, the ROM must be at least 16 bits wide to support writes (see Section 11.2.3). If 32-bit data is written into a 16-bit-wide ROM, the processor performs two simple writes to write the entire 32 bits.

11.2.4.2 Byte Writes

If the BWE bit is set in the ROM Control Register, the processor uses the $\overline{CAS3}$–$\overline{CAS0}$ signals as individual byte strobes, to allow byte and half-word writes to the ROM address space. Note that reusing the $\overline{CAS3}$–$\overline{CAS0}$ signals causes $\overline{CAS}$-only cycles to the memories in the DRAM banks (if present) during ROM writes and causes spurious write enables to non-selected memories in the ROM banks during DRAM accesses. These normally do not cause invalid operation. Furthermore, hidden refresh is disabled during ROM reads or writes if the BWE bit is set, to prevent invalid interference between simultaneous ROM and DRAM cycles. Thus, one slight disad-

Figure 11-5 **Simple Write to ROM Bank (for alterable memories in the ROM address space)**



vantage of using ROM byte writes is that there are fewer hidden refresh cycles and hence slightly degraded system performance.

The CAS3–CAS0 signals are used to write individual bytes for a 32-bit-wide ROM bank as follows:

Data width	A1–A0	CAS3–CAS0 (on write)
8 bits	00	0111
8 bits	01	1011
8 bits	10	1101
8 bits	11	1110
16 bits	0x	0011
16 bits	1x	1100
32 bits	xx	0000

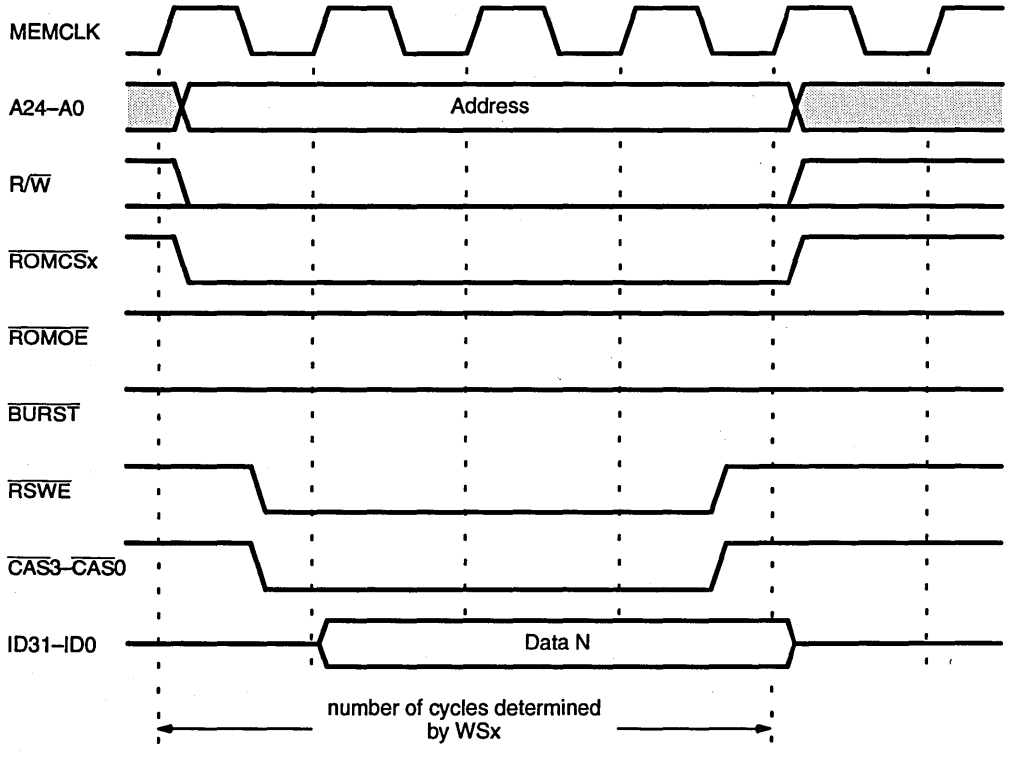
The $\overline{CAS3}$–$\overline{CAS0}$ signals are used to write individual bytes for a 16-bit-wide bank (that is, a narrow bank) as follows:

Data width	A1–A0	$\overline{CAS3}$–$\overline{CAS0}$ (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

Byte writes are not supported for 8-bit-wide narrow banks.

Figure 11-6 shows the timing of a write to the ROM address space. The $\overline{CAS3}$–$\overline{CAS0}$ signals have exactly the same timing as $\overline{RSWE}$.

Figure 11-6 Byte Write to ROM Bank (using $\overline{CAS3}$–$\overline{CAS0}$ as byte strobes)



11.2.5 Burst-Mode ROM Accesses

Figure 11-7 shows the timing of a burst-mode ROM access, for direct connection to burst-mode devices. Burst-mode accesses have a minimum of one wait state in the initial access, even when wait states are programmed as zero. Burst-mode writes are not supported.

11.2.6 Use of $\overline{\text{WAIT}}$ to Extend ROM Cycles

If the $\overline{\text{WAIT}}$ signal is asserted two cycles before the end of a ROM access (that is, two cycles before the cycle in which $\overline{\text{ROMCS}}$x would normally be deasserted), the processor extends the ROM access until $\overline{\text{WAIT}}$ is deasserted. This permits the system to extend the ROM access indefinitely. The access ends on the cycle after $\overline{\text{WAIT}}$ is deasserted, both for reads (Figure 11-8) and for writes (Figure 11-9).

The $\overline{\text{WAIT}}$ 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 $\overline{\text{WAIT}}$ is High.

Figure 11-7 Burst-Mode ROM Read

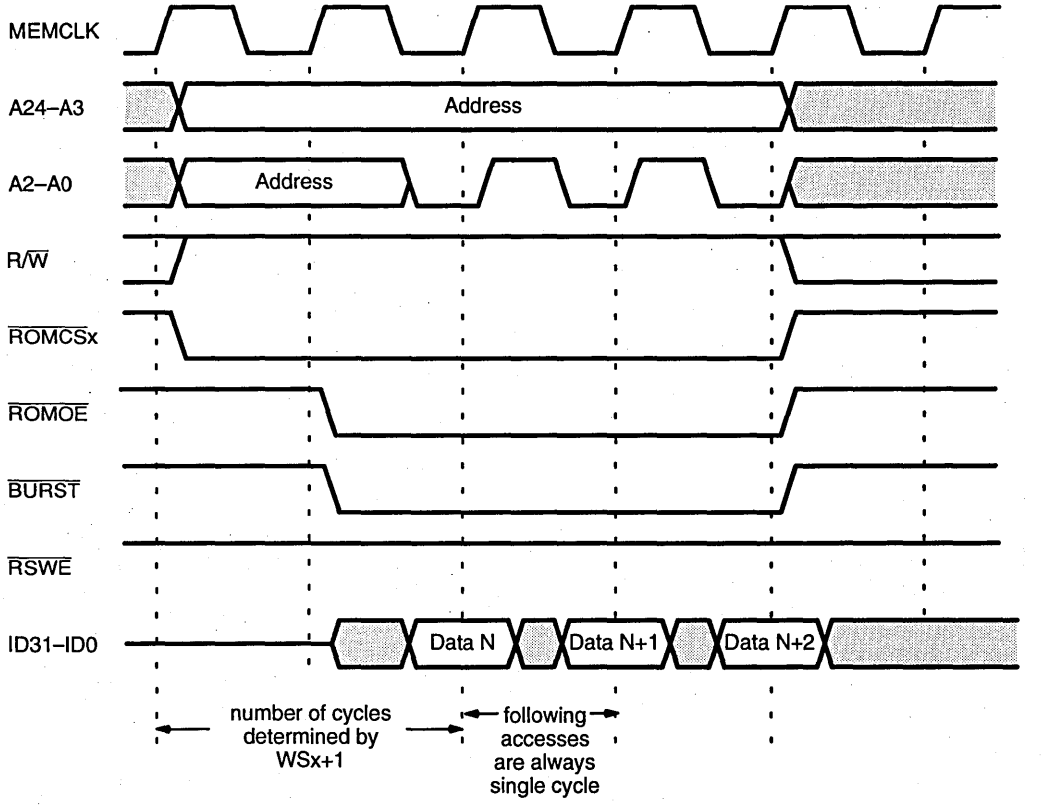


Figure 11-8 Extending a ROM Read Cycle with $\overline{\text{WAIT}}$

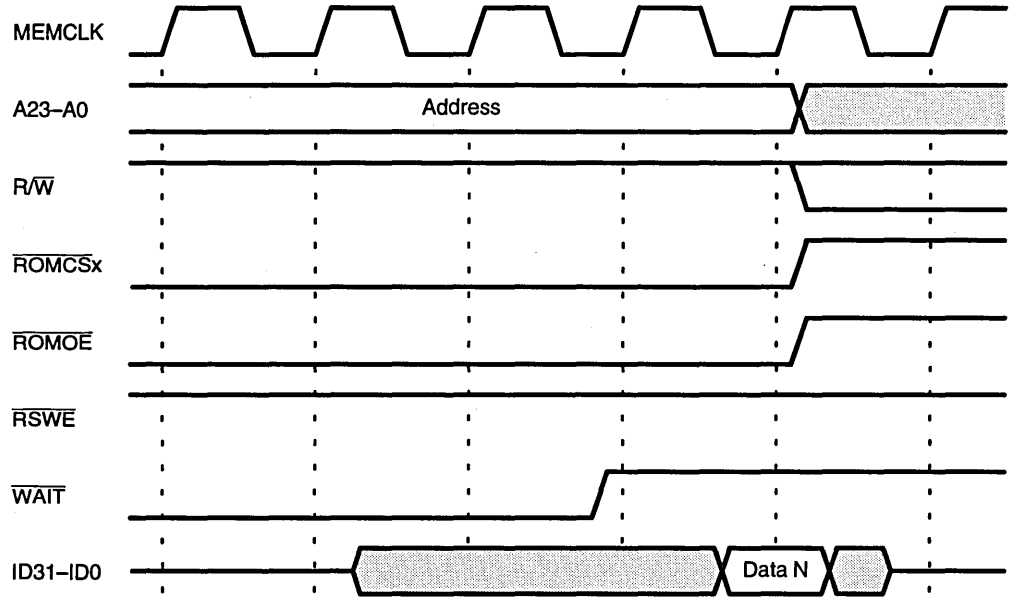
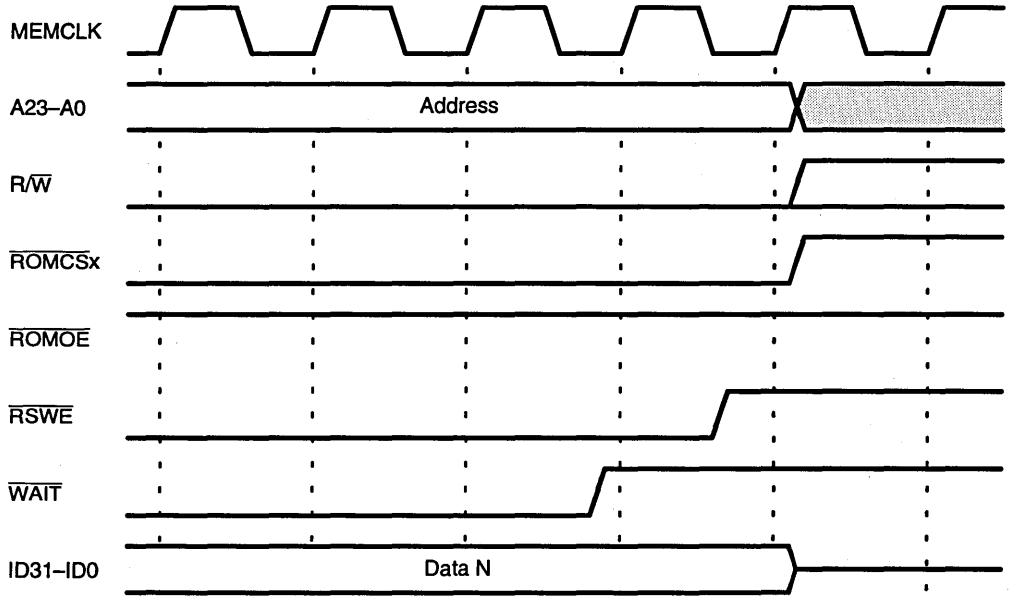


Figure 11-9 Extending a ROM Write Cycle with $\overline{\text{WAIT}}$



CHAPTER

12 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.

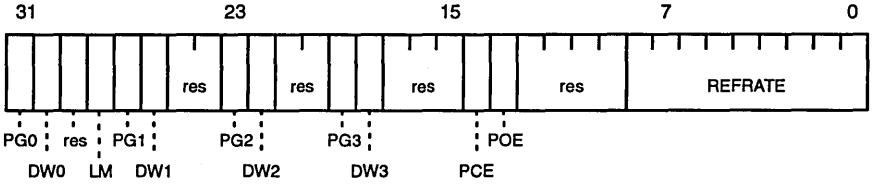
The DRAM controller supports two-cycle accesses, with single-cycle page-mode and burst-mode accesses. The DRAM mapping performed by a special functional unit in the Am29200 and Am29205 microcontrollers is performed by the MMU in the Am29240 microcontroller series; thus there are no DRAM Mapping Registers in the Am29240 microcontroller series.

12.1 PROGRAMMABLE REGISTERS

12.1.1 DRAM Control Register (DRCT, Address 80000008)

The DRAM Control Register (Figure 12-1) controls the access to and refresh of DRAM Banks 0 through 3.

Figure 12-1 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.

Bits 17–15: Reserved

Bit 14 : Parity Check Enable (PCE), Am29243 microcontroller only—A 1 in this bit enables parity generation and checking on DRAM accesses. This bit must be set to 0 for the Am29240 and Am29245 microcontrollers.

Bit 13 : Parity Odd or Even (POE), Am29243 microcontroller only—If parity is enabled by the PCE bit, this bit specifies whether parity is odd (POE=1) or parity is even (POE=0).

Bits 12–9: Reserved

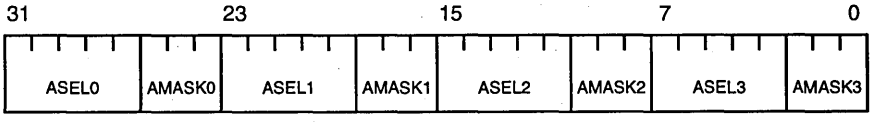
Bits 8–0: Refresh Rate (REFRATE)—This field indicates the number of MEMCLK cycles between DRAM refresh cycles. "CAS before RAS" cycles are performed, overlapped in the background with other non-DRAM accesses when possible. If one or more banks have not been refreshed in the background when the REFRATE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

12.1.2 DRAM Configuration Register (DRCF, Address 8000000C)

The DRAM Configuration Register (Figure 12-2) 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 12-2 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.

12.1.3 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 REFRATE field is initialized on reset to the value 1ff, hexadecimal. DRAM power-up requirements must be guarunteed by software.

12.2 DRAM ACCESSES

12.2.1 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.

12.2.2 Address Multiplexing

The address multiplexing for the DRAMs is performed directly by the processor on the A14–A1 pins, and no external multiplexing is required. As shown in Table 12-1 and Table 12-2, only the odd physical address pins from A9 and above (A9, A11, and A13) are used for 16-bit interfaces, while only even physical address pins above A9 (A10, A11, and A14) are used for 32-bit memories. Address bit A0 is not represented, since the Am29240 microcontroller series supports only 16- and 32-bit DRAM widths. Address multiplexing for 16- and 32-bit DRAM memories is performed as shown in Table 12-1 and Table 12-2 ("*ax*" represents address bit x).

Table 12-1 Address Multiplexing for 16-bit DRAM Memory

Address Pin	RAS Asserted	CAS Asserted	Bank Depth (LM=0) (ea)	Bank Depth (LM=1) (ea)
♦				
A13	a21	a22	4 Mbyte	8 Mbyte
♦				
A11	a19	a20	1 Mbyte	2 Mbyte
♦				
A9	a18	a9	Up to 256 Kbyte	Up to 512 Kbyte
A8	a17	a8		
A7	a16	a7		
A6	a15	a6		
A5	a14	a5		
A4	a13	a4		
A3	a12	a3		
A2	a11	a2		
A1	a10	a1		

Note: *♦ indicates signals not applicable to the bus width.*

Table 12-2 Address Multiplexing for 32-bit DRAM Memory

Address Pin	RAS Asserted	CAS Asserted	Bank Depth (LM=0) (ea)	Bank Depth (LM=1) (ea)
A14	a22	a23	4 Mbyte	16 Mbyte
♦				
A12	a20	a21	2 Mbyte	4 Mbyte
♦				
A10	a19	a10	Up to 512 Kbyte	Up to 1 Mbyte
A9	a18	a9		
A8	a17	a8		
A7	a16	a7		
A6	a15	a6		
A5	a14	a5		
A4	a13	a4		
A3	a12	a3		
A2	a11	a2		
♦				

Note: *♦ indicates signals not applicable to the bus width.*

Table 12-3 shows how this multiplexing of addresses supports various configurations of memory densities and memory widths, assuming the individual DRAMs are 4 bits wide. The addresses shown in Table 12-3 are the address bits for an access. Table 12-4 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.

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

Table 12-3 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 12-4 DRAM Address Connections to the Processor (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

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.

12.2.3 32-Bit DRAM Width

For a data access, the width of each DRAM bank can be programmed to be either 32 or 16 bits by the DRAM Control Register. If the DRAM is 32 bits wide, ID31–ID0 are used to transfer data to and from the processor, and the processor performs one access to read or write a byte, half-word, or word. The $\overline{\text{CAS3}}$–$\overline{\text{CAS0}}$ signals are asserted as follows (the value "0" is Low, "1" is High, and "x" is a don't care):

Data Width	A1–A0	CAS3–CAS0 (on write)
8 bits	00	0111
8 bits	01	1011
8 bits	10	1101
8 bits	11	1110
16 bits	0x	0011
16 bits	1x	1100
32 bits	00 (one cycle)	0000

12.2.4 16-Bit DRAM Width

If the DRAM is 16 bits wide, only ID31–ID16 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 ID31–ID16. Figure 12-3 shows the location of bytes and half-words from a 16-bit DRAM bank. In Figure 12-3, bytes and half-words are numbered as they are numbered in a word.

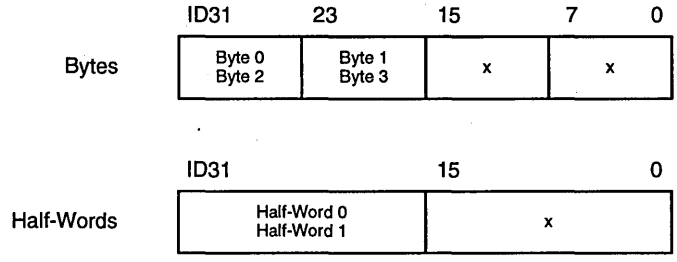
To write a 32-bit word into a 16-bit DRAM bank, the processor first writes the high-order 16 bits of the word, then generates a second access to write the low-order 16 bits of the word. The address is incremented by two for the second access, and the low-order bits of the word appear on ID31–ID16. To 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 ID31–ID24 and ID23–ID16. For a half-word write, the appropriate half-word appears on ID31–ID16. The $\overline{CAS3}$–$\overline{CAS0}$ signals are asserted as follows (the value "0" is Low, "1" is High, and "x" is a don't care):

Data Width	A1–A0	$\overline{CAS3}$–$\overline{CAS0}$ (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

12.2.5 Mapped DRAM Accesses

Untranslated accesses with addresses in the 64-Mbyte address range 50000000–53FFFFFF are mapped by the MMU. This allows the MMU to support DRAM mapping that is compatible with the Am29200 and Am29205 microcontrollers from the perspective of an applications program. However, the Am29200 or Am29205 microcontrollers and Am29240 microcontroller series require different operating-system support for DRAM mapping.

Figure 12-3 Location of Bytes and Half-Words on a 16-Bit Bus



12.2.6 Normal Access Timing

Figure 12-4 shows the timing for a normal DRAM read cycle. Figure 12-5 shows the timing for a normal DRAM write cycle. DRAM cycles are fixed at three 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 second falling edge of MEMCLK after the beginning of the cycle.

On a write, data is driven in the first cycle, as $\overline{RAS}$ falls. To avoid the possibility of a bus collision, the processor inserts one cycle of delay between the end of an external read and the beginning of a DRAM write. This delay is not inserted if there is no read. (For non-DRAM writes, data is not driven in the first cycle, so no delay is needed to avoid bus collision.)

The processor meets the $\overline{RAS}$ address setup time (t_{ASR}) by internal delay between the address and the falling edge of $\overline{RAS}$. The $\overline{CAS}$ address is driven halfway into the first cycle to meet the $\overline{CAS}$ address setup time (t_{ASC}). Also, the processor is guaranteed to meet a $\overline{RAS}$ Low time (t_{RAS}) that is 1.5 times the MEMCLK cycle time and a $\overline{RAS}$ precharge time (t_{RP}) that is 1.2 times the MEMCLK cycle time. For DRAM reads, each byte of data is latched into the processor with the rising edge of the respective $\overline{CAS}$. This increases the available access time by removing the skew between the falling edge of $\overline{CAS}$ and the rising edge of MEMCLK as a factor in the available access time.

The DRAM timing is designed so that 80-ns DRAMs can be used at 16 MHz (MEMCLK frequency), 70-ns DRAMs at 20 MHz, and 60-ns DRAMs at 25 MHz.

Figure 12-4 DRAM Read Cycle

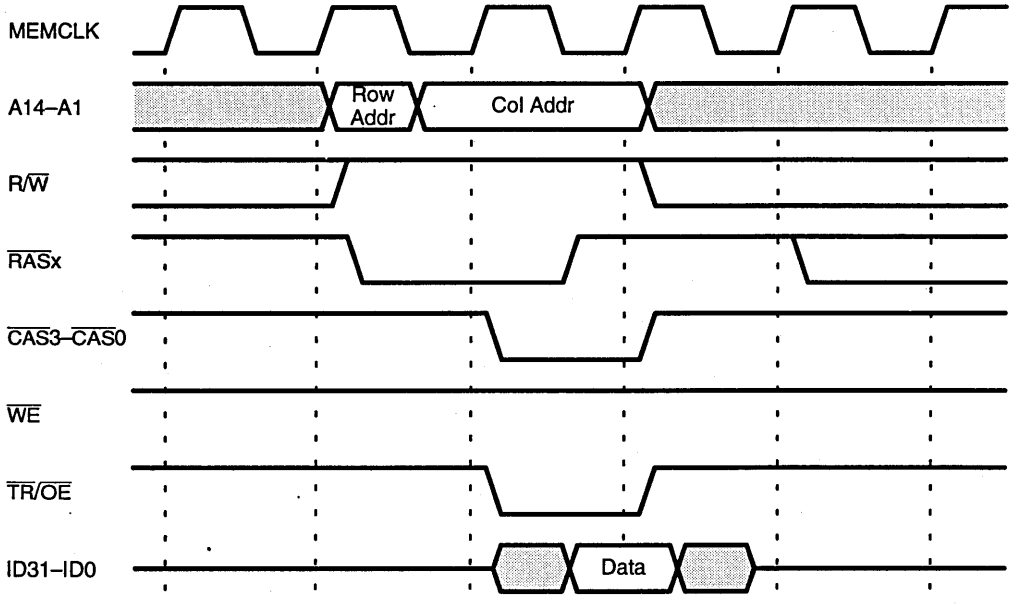
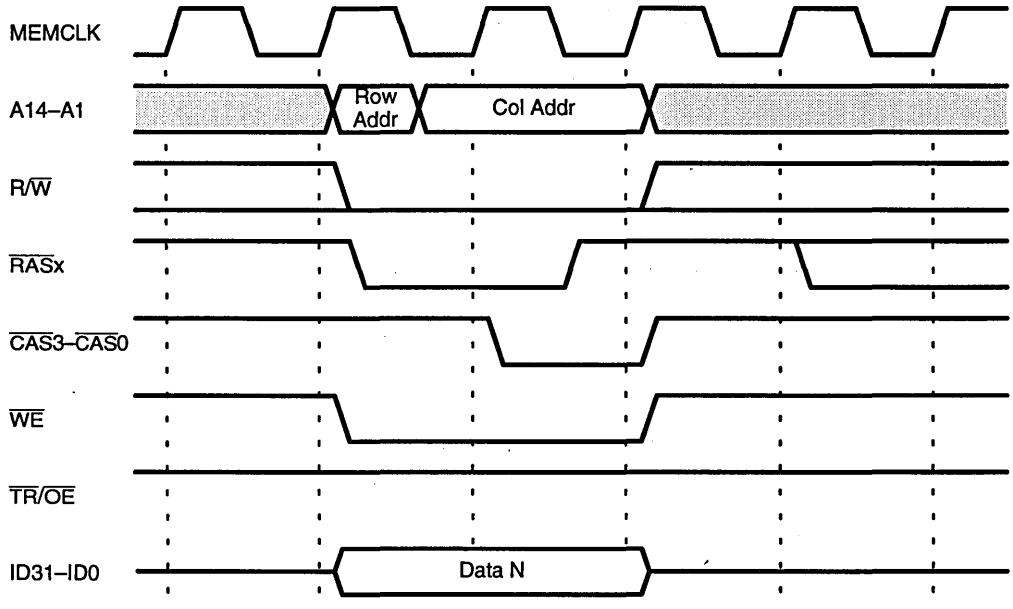


Figure 12-5 DRAM Write Cycle



12.2.7 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, data-cache reload, and for the LOADM and STOREM instructions. Page-mode accesses permit an access time of one cycle 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.

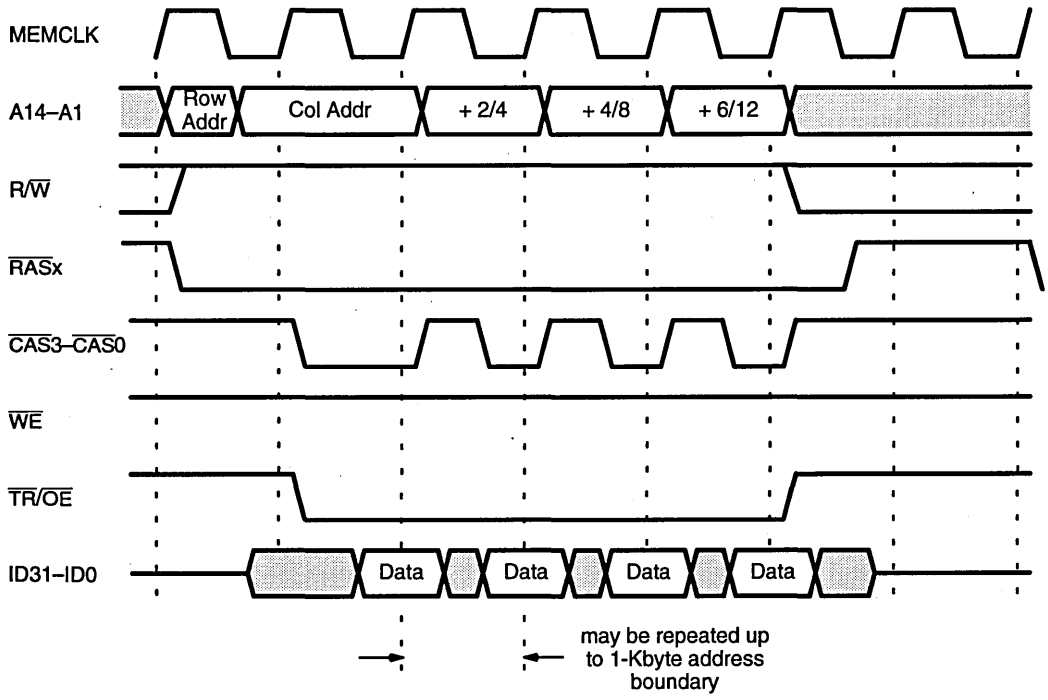
Figure 12-6 shows the timing for a page-mode DRAM read cycle. Figure 12-7 shows the timing for a page-mode DRAM write cycle. The $\overline{CAS}$ Low time (t_{CAS}) and available $\overline{CAS}$ access time (t_{CAC}) are both guaranteed to be 0.4 times the MEMCLK cycle time. The available $\overline{CAS}$ access time is guaranteed by latching each data byte with the rising edge of the respective $\overline{CAS}$. This removes the skew between the falling edge of $\overline{CAS}$ and the rising edge of MEMCLK as a factor in the available access time. Because $\overline{CAS}$ is used to clock data, static-column accesses (for which $\overline{CAS}$ is not toggled) cannot be supported.

Figure 12-6 shows how page-mode accesses might be used to reload a data cache block. However, in the case of a cache block, the addressing pattern is not necessarily sequential because addressing starts with the word that the processor requires and wraps within the block.

12.2.8 DRAM Refresh

"CAS before RAS" refresh cycles are performed periodically by the processor, as determined by the REFRATE field of the DRAM Control Register. The REFRATE field

Figure 12-6 DRAM Page-Mode Read



specifies the number of MEMCLK cycles in a refresh interval; a zero in this field disables refresh. The processor ensures 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 12-8 shows the timing of a refresh cycle. Refresh cycles take a total of four cycles because of the need to assert $\overline{CAS}$ before $\overline{RAS}$ and still meet the normal $\overline{RAS}$ timing requirements. Because refresh cycles use only the $\overline{RAS3}$–$\overline{RAS0}$ and $\overline{CAS3}$–$\overline{CAS0}$ signals, the processor attempts to perform a 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.

12.2.9 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 12-7 DRAM Page-Mode Write

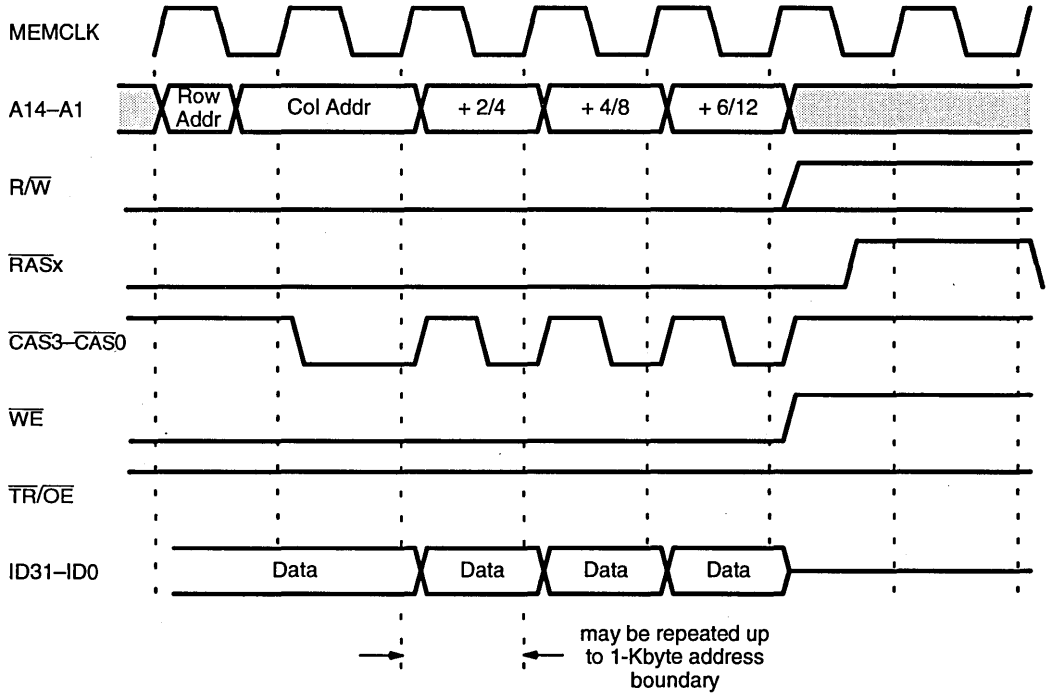


Figure 12-8 DRAM Refresh Cycle

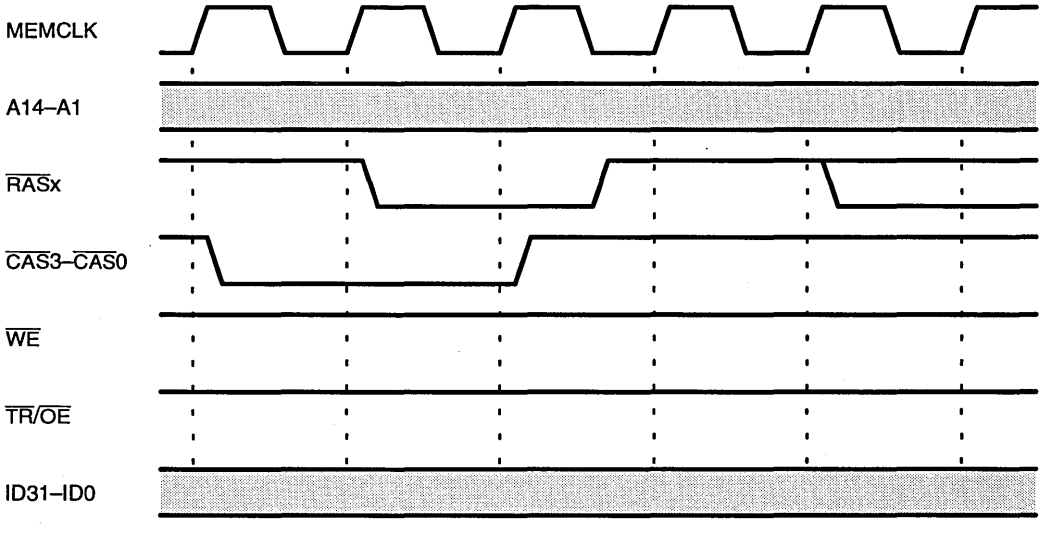


Figure 12-9 shows the timing of a VDRAM transfer cycle. This cycle differs from a normal DRAM cycle in that the signal $\overline{TR/OE}$ is asserted with different timing. VDRAM transfer cycles take a total of four cycles because of the need to assert $\overline{TR/OE}$ before $\overline{RAS}$ and still meet the normal $\overline{RAS}$ timing requirements. Note that the ID bus is not forced to high impedance.

12.3 PARITY (Am29243 MICROCONTROLLER ONLY)

DRAM parity generation and checking is enabled and disabled by the PCE bit of the DRAM Control Register. Parity checking is enabled when the PCE bit is 1, and is disabled when the PCE bit is 0. If parity checking is enabled, the processor generates and checks byte parity for DRAM accesses using the IDP3–IDP0 pins. This section describes the processor operation when parity checking is enabled.

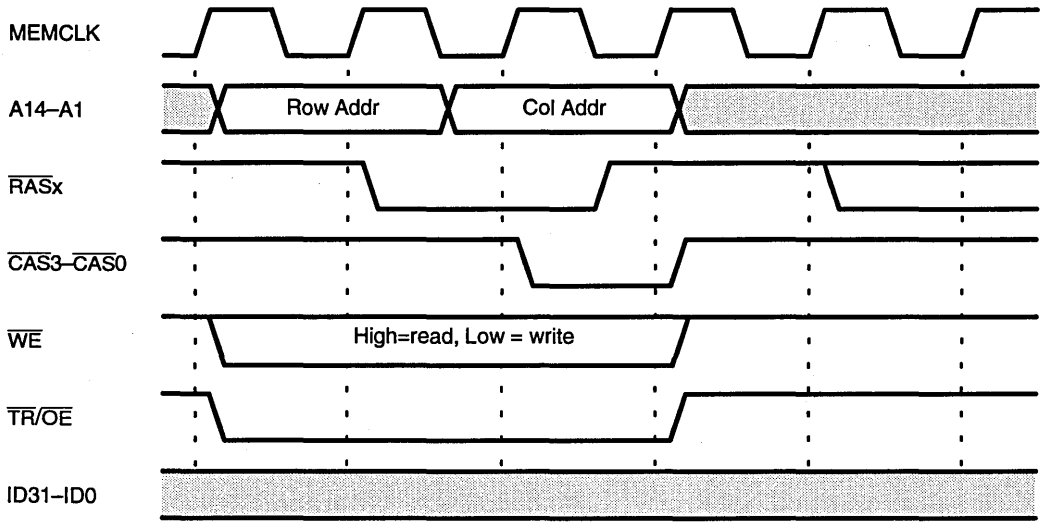
12.3.1 Parity Generation and Checking

When the processor supplies data for a DRAM write, it also supplies the data parity for all bytes on the IDP3–IDP0 pins. The IDP3 pin is the parity bit for ID31–ID24, the IDP2 pin is the byte parity for ID23–ID16, and so on. Parity is either odd or even as controlled by the POE bit of the DRAM Control Register. The parity bits appear on the IDP pins with the same timing as data on the ID Bus.

When the processor reads DRAM for an instruction or data access, it expects valid parity to be supplied for each byte that is transferred. For example, if byte 3 is read on ID7–ID0, only IDP0 need be valid.

When parity is supplied by the system, the processor checks for valid parity during the cycle after the data is received. Parity is checked only for the bytes actually involved in the transfer. If any byte has invalid parity, a Parity Error trap occurs.

Figure 12-9 VDRAM Transfer Cycle



12.3.2 Reporting Parity Errors

The Parity Error trap has a vector number of 4, decimal (this number was previously used in the Am29000, Am29005, and Am29050 microprocessors to indicate a coprocessor exception). The Parity Error trap cannot be masked by the DA bit of the Current Processor Status Register. The parity error may be detected either during a data access or an instruction access, and is handled slightly differently depending on the type of access.

If a parity error is detected during a data access, the PER bit of the Channel Control Register is set and a Parity Error trap occurs. The PER bit causes a trap whenever it is 1, to allow the proper sequencing of the Parity Error trap. Other information related to the access is retained in the Channel Address, Data, and Control Registers, so that the access may be restarted, if possible. Data with invalid parity is not written into the register file. Furthermore, if the invalid data is forwarded to an instruction that is waiting on the data (in the execute stage), the waiting instruction does not complete execution upon receiving the data and is not allowed to write its result into the register file, because this result is invalid. When the Parity Error trap is taken, the PC1 Register contains the address of the instruction that was not completed, so this instruction can be restarted, if possible.

If a parity error is detected during an instruction access, a Parity Error trap occurs if the processor attempts to execute the corresponding instruction. When the trap occurs, the address of the invalid instruction is contained in the PC1 Register. This trap uses the same vector number as a trap caused by a data access parity error. Parity errors on instruction and data accesses can be differentiated from one another by the PER bit of the Channel Control Register, which is set only for a parity error on a data access.

If a parity error is detected while a cache block is being loaded, the cache valid bit is not set.

CHAPTER

13 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.

13.1 PROGRAMMABLE REGISTERS

13.1.1 PIA Control Registers (PICT0/1, Address 80000020/24)

The PIA Control Registers (Figure 13-1 and Figure 13-2) control the access to PIA Regions 0 through 5.

Figure 13-1 PIA Control Register 0 (PICT0, Address 80000020)

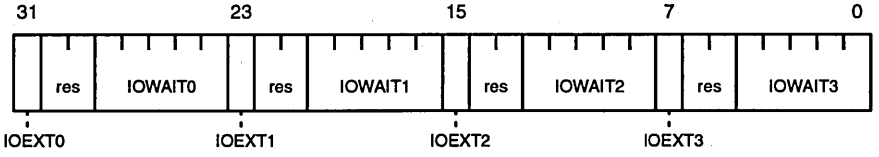
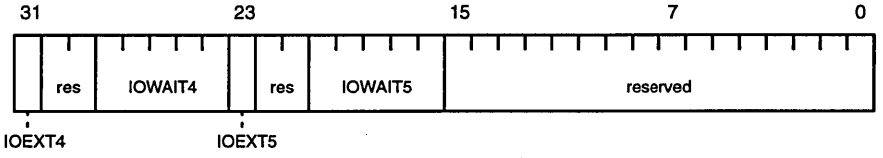


Figure 13-2 PIA Control Register 1 (PICT1, Address 80000024)



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. To achieve all address and data setup and hold times, 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). Read accesses of less than three cycles (two wait states) and write accesses of less than four cycles (three wait states) sacrifice some setup and hold times to achieve the desired number cycles.

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

13.1.2 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.

13.2 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 $\overline{\text{PIACS5}}$–$\overline{\text{PIACS0}}$ 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 ID7–ID0, and a 16-bit device must be attached to ID15–ID0. LOADM and STOREM instructions (possible only for 32-bit accesses) are performed as a series of simple loads or stores.

When a byte access is made to the PIA region, the two least significant bits must be 11. When a half-word access is made, the two least significant bits must be 10.

Instruction fetching from a PIA region is not supported.

13.2.1 Normal Access Timing

Normal read accesses take three cycles and normal write accesses take four cycles. These cycles are required to provide full address and data setup and hold times. Fast accesses, described in Section 13.2.2, take fewer cycles but sacrifice some address and data setup and hold times.

Figure 13-3 shows the timing of a normal 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 processor deasserts $\overline{\text{PIACS}}$x and $\overline{\text{PIAOE}}$. The interface operates such that the processor allows at least one cycle before it drives ID31–ID0 for a new access (though a new address may be driven on A23–A0 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 13-4 shows timing of a normal 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.

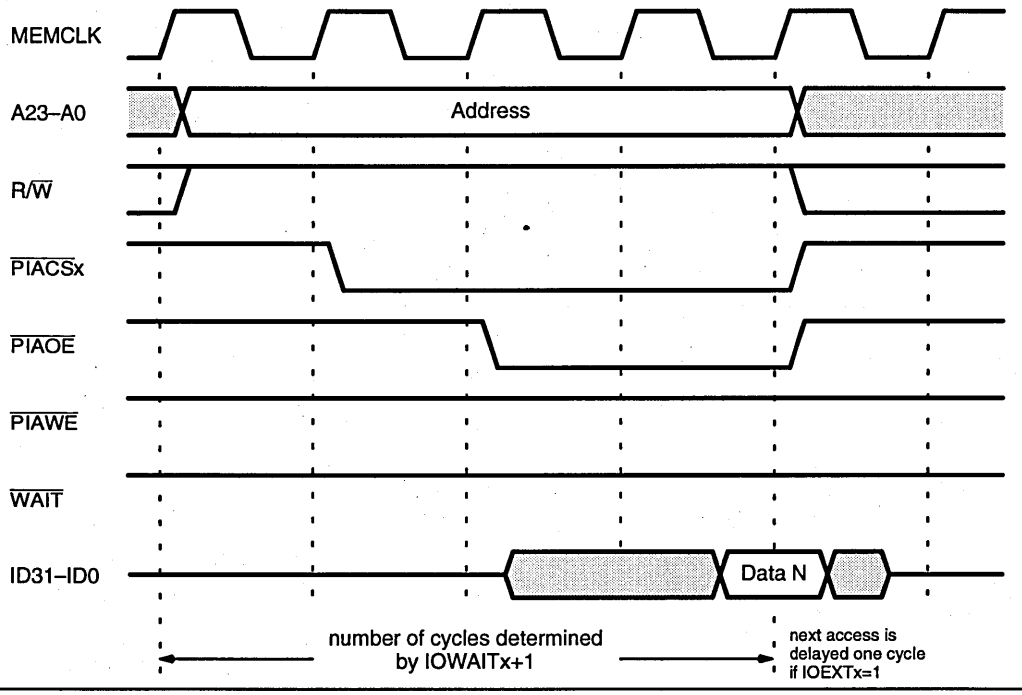
13.2.2 Fast Access Timing

The PIA interface permits peripheral reads with less than two wait states and writes with less than three wait states, using timing that is somewhat more difficult to accommodate than the normal access timing.

PIA reads with one and zero wait states are shown in Figure 13-5 and Figure 13-6, respectively. Both of these options sacrifice the setup time of the address to the falling edge of $\overline{\text{PIACS}}$x. In the case of zero wait states, the $\overline{\text{PIAOE}}$ signal is active for only a half-cycle.

PIA writes with two, one, and zero wait states are shown in Figure 13-7, Figure 13-8, and Figure 13-9, respectively. To obtain two wait states, some of the data setup to the

Figure 13-3 PIA Read Cycle



falling edge and hold time from the rising edge of PIAWE is sacrificed. To obtain one wait state, the address setup time to the falling edge of PIACSx and the data setup time to the falling edge of PIAWE are sacrificed. With zero wait states, the timing is similar to the timing with one wait state (and takes two cycles instead of one), except that the timing of PIACSx is changed to provide some amount of data setup and hold time to the rising edge of PIACSx. This permits PIACSx to be used as a write strobe in certain situations, such as when a simple latch is attached to the PIA interface.

The IOEXT bits have no effect for peripheral reads with less than two wait states and writes with less than three wait states. The WAIT (see Section 13.2.3) signal cannot be used to extend zero-wait-state PIA accesses. WAIT must be asserted two cycles before the end of an access, and zero-wait-state accesses do not provide sufficient time to assert WAIT.

13.2.3 Use of WAIT to Extend I/O Cycles

The WAIT signal is used to extend the number of wait states beyond the number determined by the IOWAITx field. WAIT can be asserted during a read at any time up until two cycles before PIAOE is deasserted, and can be asserted during a write at any time up until two cycles before PIAWE is deasserted. In response to WAIT, the processor extends the access until WAIT is deasserted. If WAIT is asserted within the appropriate amount of time, a normal read access ends on the cycle after WAIT is deasserted (Figure 13-10), and a normal write access ends on the second cycle after WAIT is deasserted, to provide data hold time (Figure 13-11). 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 13-4 PIA Write Cycle

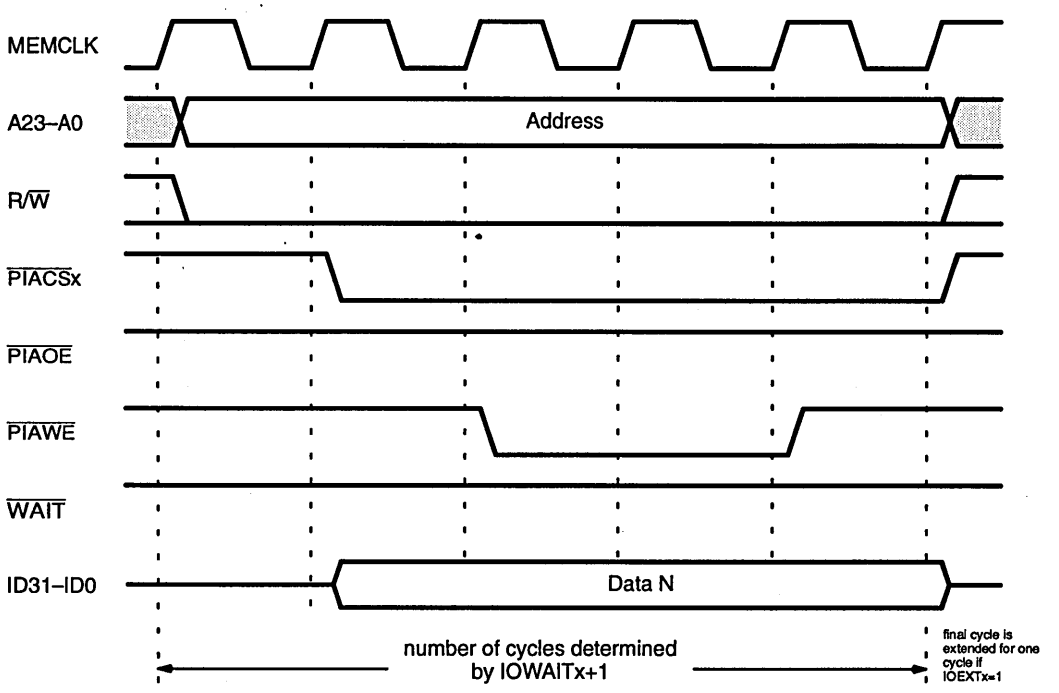


Figure 13-5 PIA Read Cycle—One Wait State

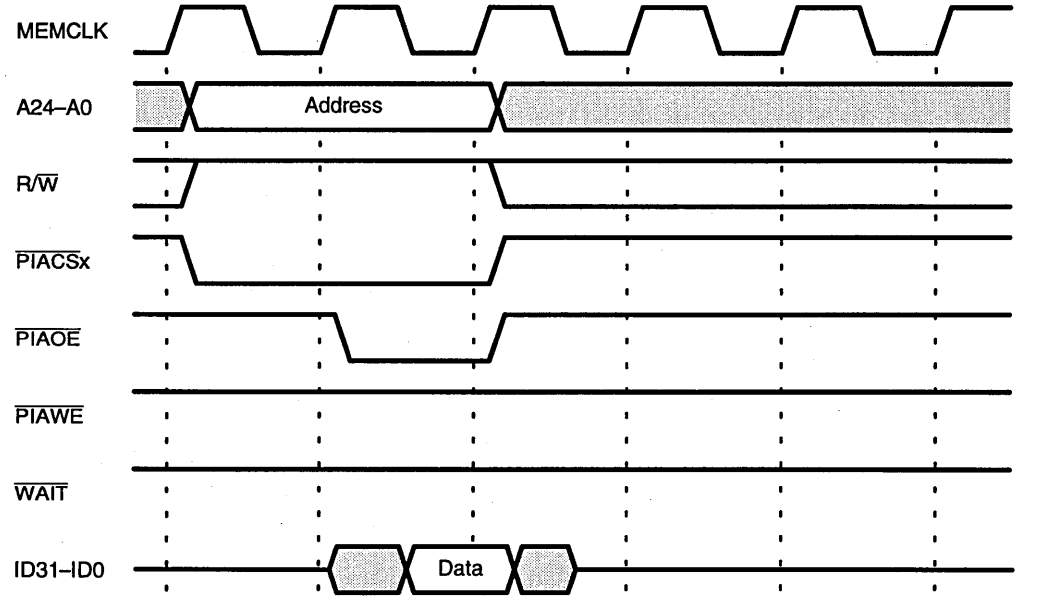


Figure 13-6 PIA Read Cycle—Zero Wait States

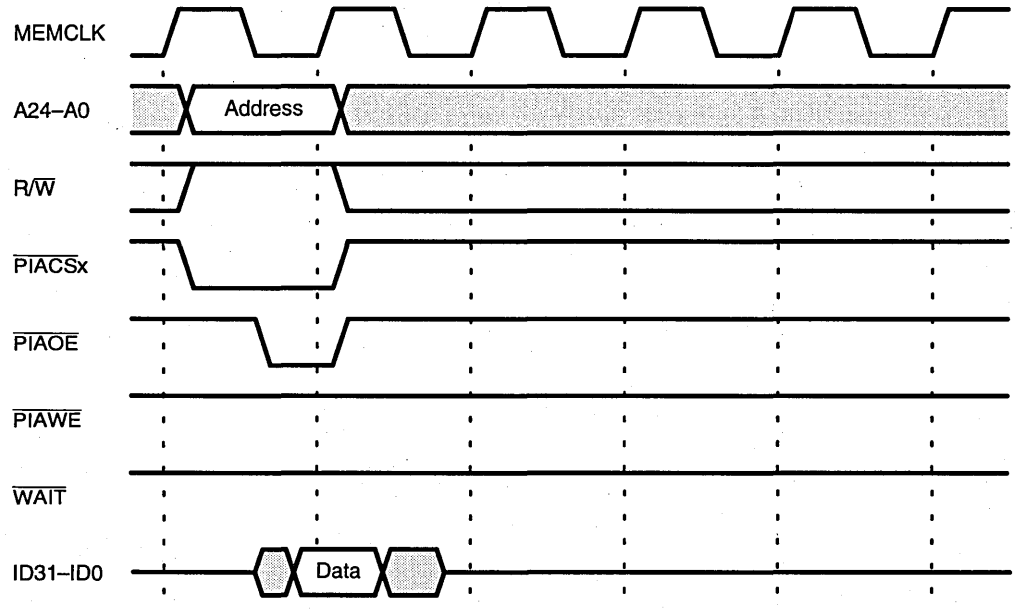


Figure 13-7 PIA Write Cycle—Two Wait States

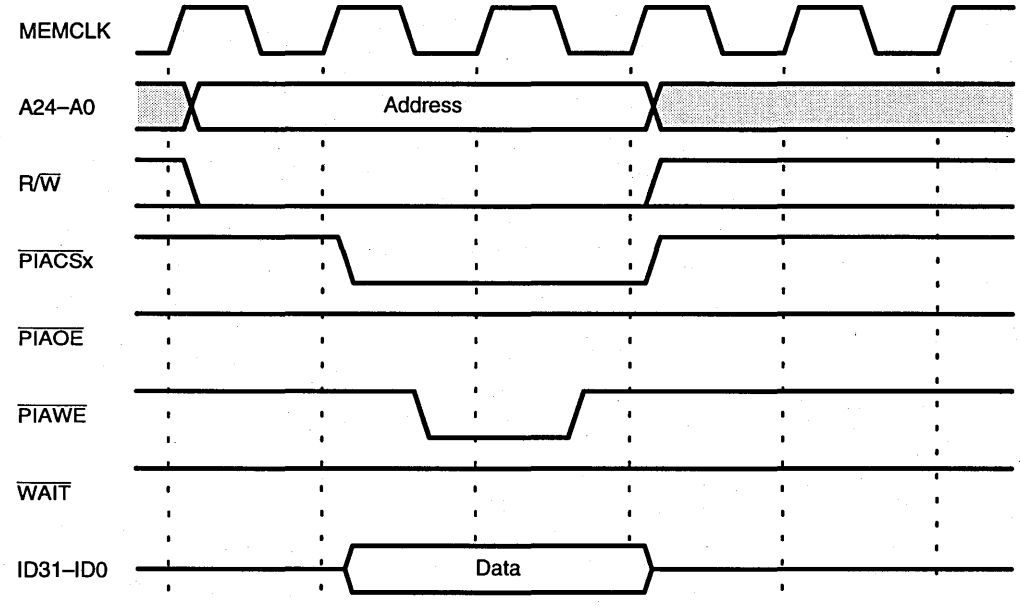


Figure 13-8 PIA Write Cycle—One Wait State

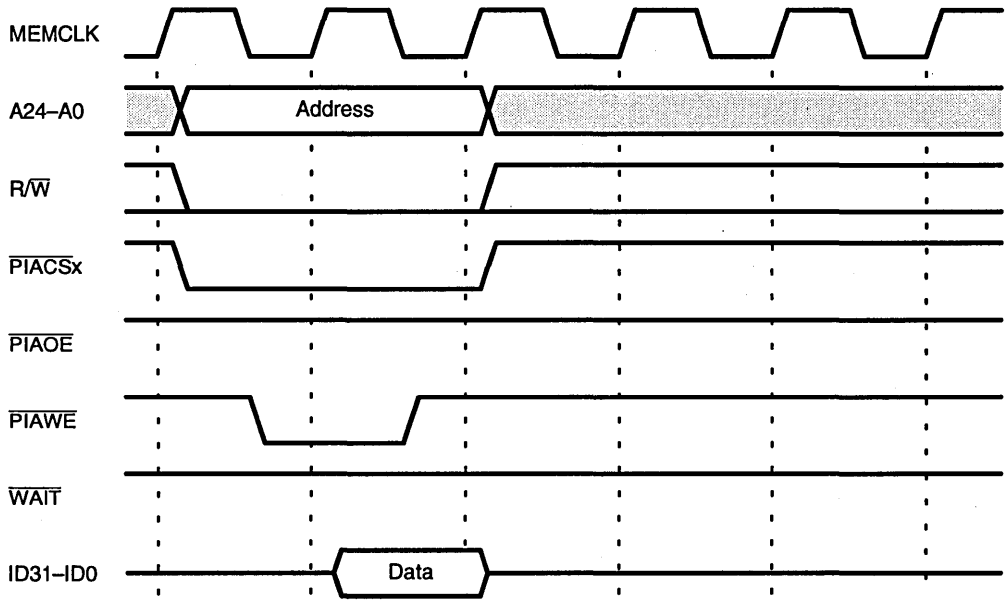


Figure 13-9 PIA Write Cycle—Zero Wait States

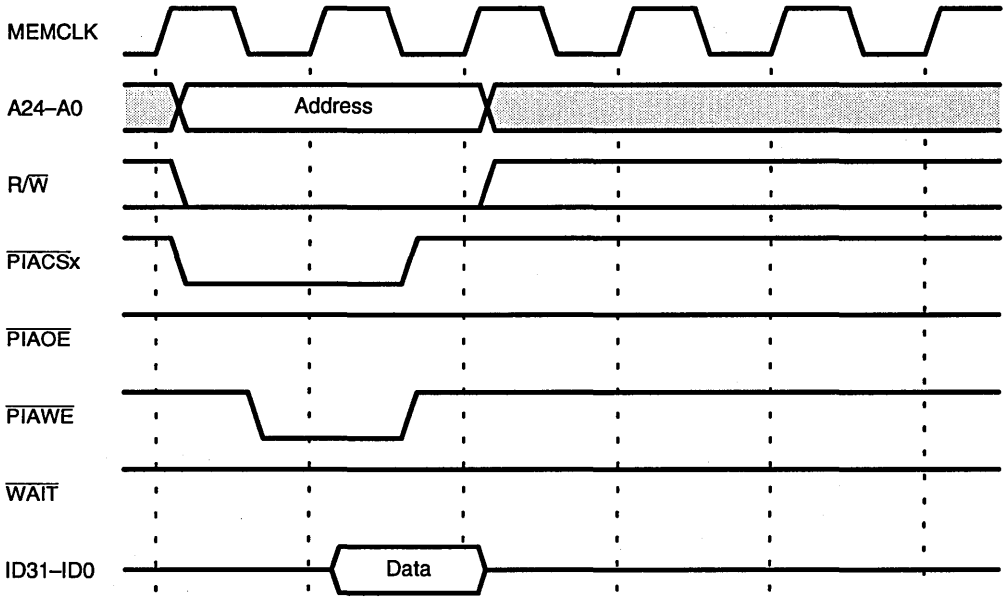


Figure 13-10 Extending a PIA Read Cycle with $\overline{\text{WAIT}}$

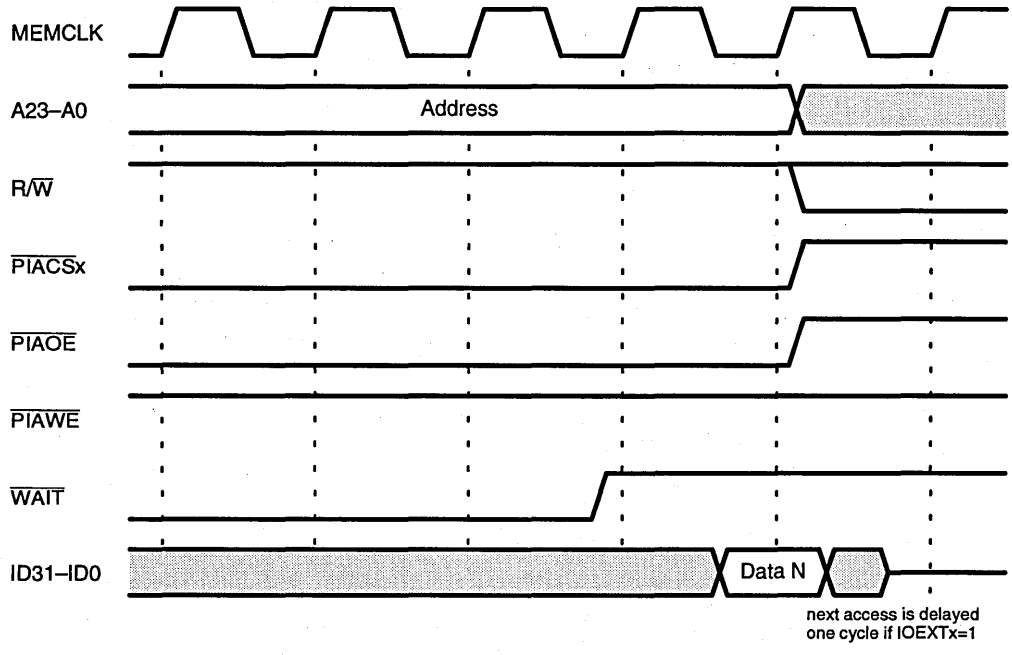
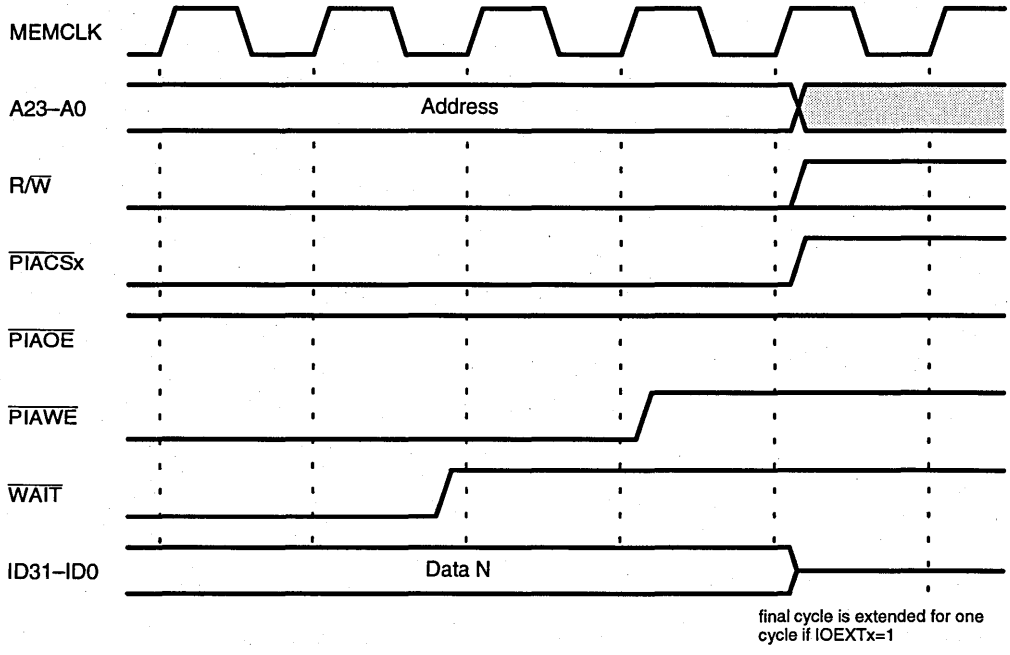


Figure 13-11 Extending a PIA Write Cycle with $\overline{\text{WAIT}}$



CHAPTER

14 DMA CONTROLLER

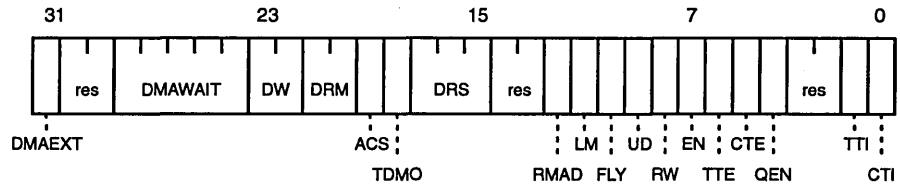
The Am29240 microcontroller series has either two or four DMA channels: the Am29245 microcontroller has two DMA channels, and both the Am29240 and Am29243 microcontrollers have four channels. Each DMA channel is capable of performing either external or internal DMA. An external DMA transfers data between an external peripheral and the DRAM or ROM address spaces. An internal DMA transfers data between an on-chip peripheral and the DRAM or ROM address spaces. All DMA channels support queued transfers. The DMA controller also supports fast fly-by transfers and direct random DRAM or ROM access by an external device such as an external DMA controller.

14.1 PROGRAMMABLE REGISTERS

14.1.1 DMA0 Control Register (DMCT0, Address 80000030)

The DMA0 Control Register (Figure 14-1) controls DMA Channel 0.

Figure 14-1 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 cycle 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 DREQA, as follows:

DRM Value	DREQA 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 DMA Channel 0 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 $\overline{\text{PIACS0}}$.

Bit 18: TDMA Output (TDMO)—This bit determines whether or not TDMA is sampled as an input or driven as an output. If TDMO=0, TMDA is sampled during the last cycle of an external DMA transfer and an active level can terminate the transfer, unless the transfer is a fly-by transfer in which case TDMA is ignored. If TDMO=1, the processor drives TDMA during an external DMA transfer and asserts TDMA on the last transfer as determined by the DMA count.

Bits 17–15: DMA Request Select (DRS)—This field selects which external DMA request/acknowledge pair is used by the DMA channel, as follows:

DRS Value	DMA Request/Acknowledge Pair
000	DREQA/$\overline{\text{DACK}}$A (for compatibility with Am29200/205 microcontrollers
001	Reserved
010	DREQA/$\overline{\text{DACK}}$A
011	DREQB/$\overline{\text{DACK}}$B
100	DREQC/$\overline{\text{DACK}}$C
101	DREQD/$\overline{\text{DACK}}$D
110	$\overline{\text{GREQ}}$ (&DREQA for burst)/$\overline{\text{GACK}}$
111	Reserved

Internal peripherals can generate DMA requests for any channel regardless of the DRS field. The value DRS=000 is used as a default for compatibility with the Am29200 and Am29205 microcontrollers. This value has a different interpretation for DMA Channels 1, 2, and 3.

When $\overline{\text{GREQ}}$ and $\overline{\text{GACK}}$ are used as the DMA request and acknowledge pairs (DRS=110), it is possible to perform a burst-mode transfer using the DMA Count Register to specify the number of transfers that occur in the burst. This is accomplished by setting the FLY bit in the DMA control register and asserting DREQA at the same time as $\overline{\text{GREQ}}$. In this case, the DREQA pin must be programmed as level-sensitive by the DRM field. If no channel specifies $\overline{\text{GREQ}}/\overline{\text{GACK}}$, or if DREQA is not asserted, then the $\overline{\text{GREQ}}/\overline{\text{GACK}}$ protocol transfers a single word.

The values 100–111 for DRS are valid only for the Am29240 and Am29243 microcontrollers. These values should not be used on the Am29245 microcontroller.

Bits 14–13: Reserved

Bit 12: ROM Address (RMAD)—If this bit is 0, the memory addresses for DMA Channel 0 are in the DRAM address space. If this bit is 1, the addresses are in the ROM address space.

Bit 11: Large Memory (LM)—This bit determines the addressing range used by the DMA Channel. If LM=0, a DMA transfer can address anywhere within a 16-Mbyte DRAM region. If LM=1, a DMA transfer can address anywhere within a 64-Mbyte DRAM. The larger DRAM address range is achieved at the expense of a smaller peripheral address range

Bit 10: Fly-By Transfers (FLY)—This bit enables DMA fly-by transfers, whereby data is transferred directly between the DRAM and peripheral at a rate of up to one word per cycle. Fly-by transfers do not support programmable wait states or peripheral addressing. The peripheral must support the maximum DRAM transfer rate and must be addressed with a single chip select or DMA acknowledge.

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. 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

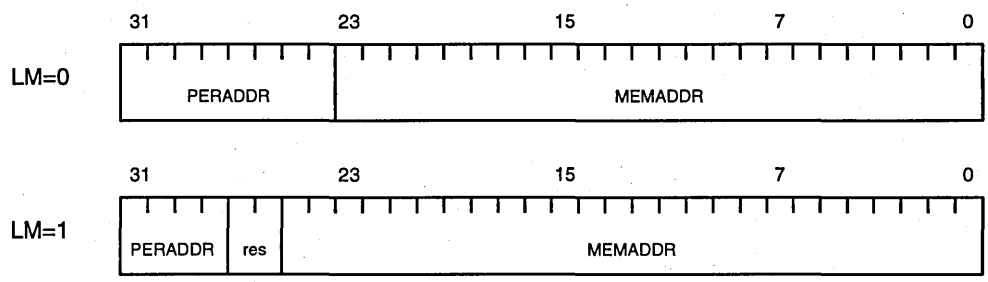
Bit 1: TDMA Terminate Interrupt (TTI)—The TTI bit is used to report that the DMA channel has generated an interrupt because of TDMA termination. If the TTE bit is one and the TDMA signal is asserted during an external DMA transfer, the TTI bit is set and a processor interrupt occurs.

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.

14.1.2 DMA0 Address Register (DMAD0, Address 80000034)

The DMA0 Address Register (Figure 14-2) contains the physical addresses for a transfer by DMA Channel 0. DMA accesses use physical addresses and the addresses cannot be translated by the MMU.

Figure 14-2 DMA0 Address Register



Bits 31–24 (LM=0) or
Bits 31–28 (LM=1): Peripheral Address (PERADDR)—This field specifies peripheral address bits that are driven on A7–A0 (LM=0) or A3–A0 (LM=1) during an external peripheral access by the DMA channel (non-fly-by). A23–A8 or A23–A4 are driven Low during the transfer.

Bits 27–26 (LM=1): Reserved

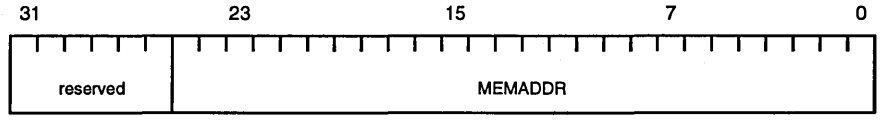
Bits 23–0 (LM=0) or
Bits 25–0 (LM=1): Memory Address (MEMADDR)—This field contains the DRAM or ROM address for the next DMA transfer. The MEMADDR field is incremented or decremented (based on the UD bit) by an amount determined by the width of the DMA

transfer. The increment or decrement amount is 1 for a byte transfer, 2 for a half-word transfer, and 4 for a word transfer. To support repeated transfers from the same word, the address can be left unchanged. The MEMADDR field wraps from the value 000000 to FFFFFF (hexadecimal) when decremented and from FFFFFF to 000000 when incremented.

14.1.3 DMA0 Address Tail Register (TAD0, Address 80000070)

This register (Figure 14-3) 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. For compatibility with the Am29200 microcontroller, this register also allows write-only access at address 80000036.

Figure 14-3 DMA0 Address Tail Register



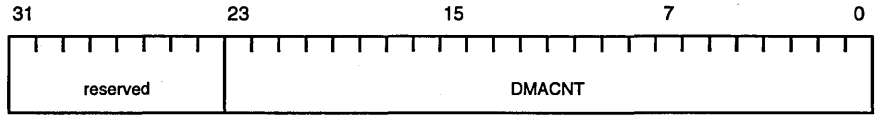
Bits 31–26: Reserved

Bits 25–0: Memory Address (MEMADDR)—This field is written with the beginning DRAM or ROM address for a queued DMA transfer, if queuing is enabled. Bits 25–24 are not used if LM=0.

14.1.4 DMA0 Count Register (DMCN0, Address 80000038)

The DMA0 Count Register (Figure 14-4) specifies the number of transfers remaining to be performed by DMA Channel 0.

Figure 14-4 DMA0 Count Register



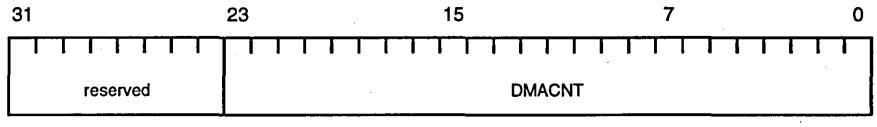
Bits 31–24: Reserved

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.

14.1.5 DMA0 Count Tail Register (TCN0, Address 8000003C)

This write-only register (Figure 14-5) 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. For compatibility with the Am29200 microcontroller, this register also allows write-only access at address 8000003A.

Figure 14-5 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.

14.1.6 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 $\overline{PCS1}$ is asserted, and DRS=00 selects DREQB/$\overline{DACK}$B.

14.1.7 DMA1 Address Register (DMAD1, Address 80000044)

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

14.1.8 DMA1 Address Tail Register (TAD1, Address 80000074)

This register is the tail of the DMA Channel 1 address queue. It is identical in layout and definition to the DMA0 Address Tail Register.

14.1.9 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.

14.1.10 DMA1 Count Tail Register (TCN1, Address 8000004C)

This register is the tail of the DMA Channel 1 count queue. It is identical in layout and definition to the DMA0 Count Tail Register.

14.1.11 Initialization

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

14.2 ADDITIONAL DMA CHANNEL REGISTERS (Am29240 AND Am29243 MICROCONTROLLERS ONLY)

This section describes the registers for the additional DMA channels available on the Am29240 and Am29243 microcontrollers. In general, these channels are very similar to DMA Channels 0 and 1, and are initialized and used in the same way.

14.2.1 DMA2 Control Register (DMCT2, Address 80000050)

The DMA2 Control Register controls DMA Channel 2. It is identical in layout and definition to the DMA0 Control Register, except that the ACS bit controls whether or not $\overline{PCS2}$ is asserted, and DRS=00 disables external DMA transfers. The EN bit in this register must be set to 0 for the Am29245 microcontroller.

14.2.2 DMA2 Address Register (DMAD2, Address 80000054)

The DMA2 Address Register contains the physical addresses for a transfer by DMA Channel 2. It is identical in layout and definition to the DMA0 Address Register.

14.2.3 DMA2 Address Tail Register (TAD2, Address 80000078)

This register is the tail of the DMA Channel 2 address queue. It is identical in layout and definition to the DMA0 Address Tail Register.

14.2.4 DMA2 Count Register (DMCN2, Address 80000058)

The DMA2 Count Register specifies the number of transfers remaining to be performed by DMA Channel 2. It is identical in layout and definition to the DMA0 Count Register.

14.2.5 DMA2 Count Tail Register (TCN2, Address 8000005C)

This register is the tail of the DMA Channel 2 count queue. It is identical in layout and definition to the DMA0 Count Tail Register.

14.2.6 DMA3 Control Register (DMCT3, Address 80000060)

The DMA3 Control Register controls DMA Channel 3. It is identical in layout and definition to the DMA0 Control Register, except that the ACS bit controls whether or not $\overline{PCS3}$ is asserted, and DRS=00 disables external DMA transfers. The EN bit in this register must be set to 0 for the Am29245 microcontroller.

14.2.7 DMA3 Address Register (DMAD3, Address 80000064)

The DMA3 Address Register contains the physical addresses for a transfer by DMA Channel 3. It is identical in layout and definition to the DMA0 Address Register.

14.2.8 DMA3 Address Tail Register (TAD3, Address 8000007C)

This register is the tail of the DMA Channel 3 address queue. It is identical in layout and definition to the DMA0 Address Tail Register.

14.2.9 DMA3 Count Register (DMCN3, Address 80000068)

The DMA3 Count Register specifies the number of transfers remaining to be performed by DMA Channel 3. It is identical in layout and definition to the DMA0 Count Register.

14.2.10 DMA3 Count Tail Register (TCN3, Address 8000006C)

This register is the tail of the DMA Channel 3 count queue. It is identical in layout and definition to the DMA0 Count Tail Register.

14.3 DMA TRANSFERS

A DMA transfer is performed as a result of a DMA request. The DMA request can be generated either by an internal peripheral, or by an external device using DREQD–DREQA. (The Am29245 microcontroller has two external DMA request/acknowledge pairs—the Am29240 and Am29243 microcontrollers have four external DMA request/acknowledge pairs.)

DMA accesses use physical addresses, and the addresses cannot be translated by the MMU.

14.3.1 Assigning DMA Channels

The assignment of DMA request/acknowledge pairs (DREQD–DREQA and $\overline{\text{DACK}}$D–$\overline{\text{DACK}}$A) is controlled by the DRS field of the DMA Control Register. Assigning two or more channels to the same signals at the same time has a unpredictable result on DMA channel operation. The generation of DMA requests by the DREQD–DREQA signals is controlled by the DRM field.

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 $\overline{\text{DACK}}$x 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 $\overline{\text{DACK}}$x is asserted, and must be deasserted during the cycle in which $\overline{\text{DACK}}$x is deasserted unless it is desired to generate a subsequent DMA request.

14.3.2 Specifying the Direction of a DMA Transfer

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 or ROM address space. The DMA channel first performs an access to read the data from the peripheral and then performs a DRAM or ROM 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 or ROM address space to the peripheral. The DMA channel first performs a DRAM or ROM 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.

14.3.3 External DMA Transfers

External DMA transfers appear very much like PIA accesses, except the DMA acknowledge signals $\overline{\text{DACK}}$D– $\overline{\text{DACK}}$A are asserted during the transfer as well as, optionally, $\overline{\text{PIACS}}$3–$\overline{\text{PIACS}}$0. The address bus is driven with an address derived from the DMA Address Register. If the LM bit is 0, bits 23–8 of the address are driven with all 0s, and bits 7–0 are driven with the PERADDR field. If the LM bit is 1, bits 23–4 of

the address are driven with all 0s, and bits 3–0 are driven with the PERADDR field. It is possible to use the $\overline{\text{DACKD}}$–$\overline{\text{DACKA}}$ 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. As with PIA accesses, the peripheral can use $\overline{\text{WAIT}}$ to extend the access.

If the DRAM or ROM is 16 bits wide, a 32-bit DMA DRAM access appears as two 16-bit accesses on ID31–ID16. If the peripheral is 8 or 16 bits wide, a DMA peripheral access appears as a single access on ID7–ID0 or ID15–ID0, respectively. The peripheral must have the same width as the transfer. DMA accesses to 8-bit-wide ROMs are not supported.

Figure 14-6 shows the timing of a DMA read cycle (performed when the RW bit is 0). The $\overline{\text{DACKx}}$ signal (and, optionally, the $\overline{\text{PIACSx}}$ 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. The peripheral can use $\overline{\text{WAIT}}$ to extend the access.

Figure 14-7 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

Figure 14-6 DMA Read Cycle

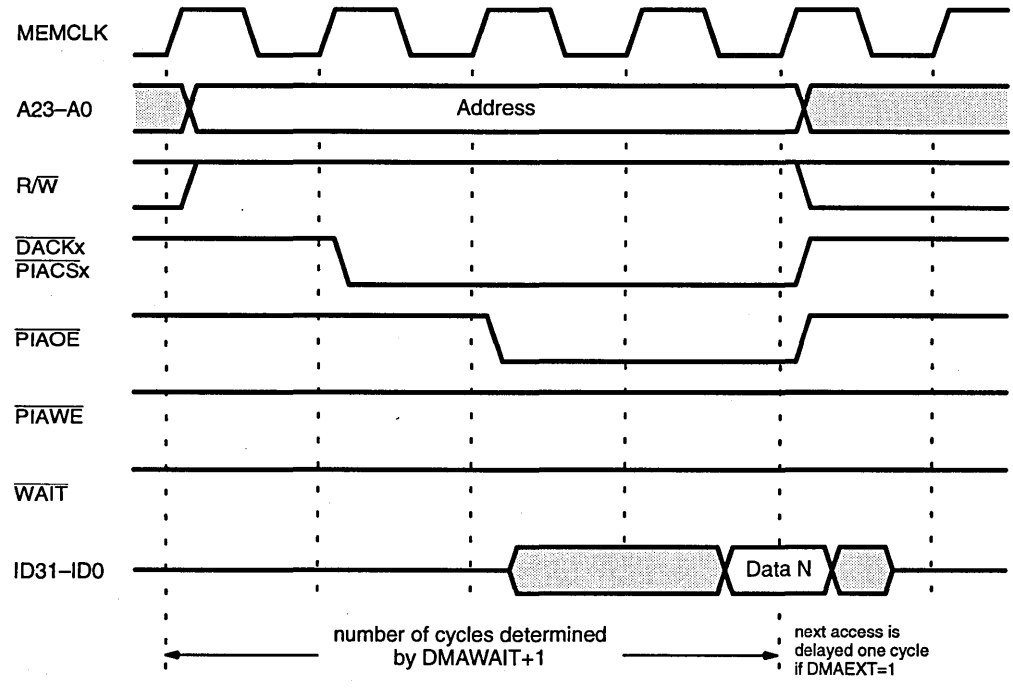
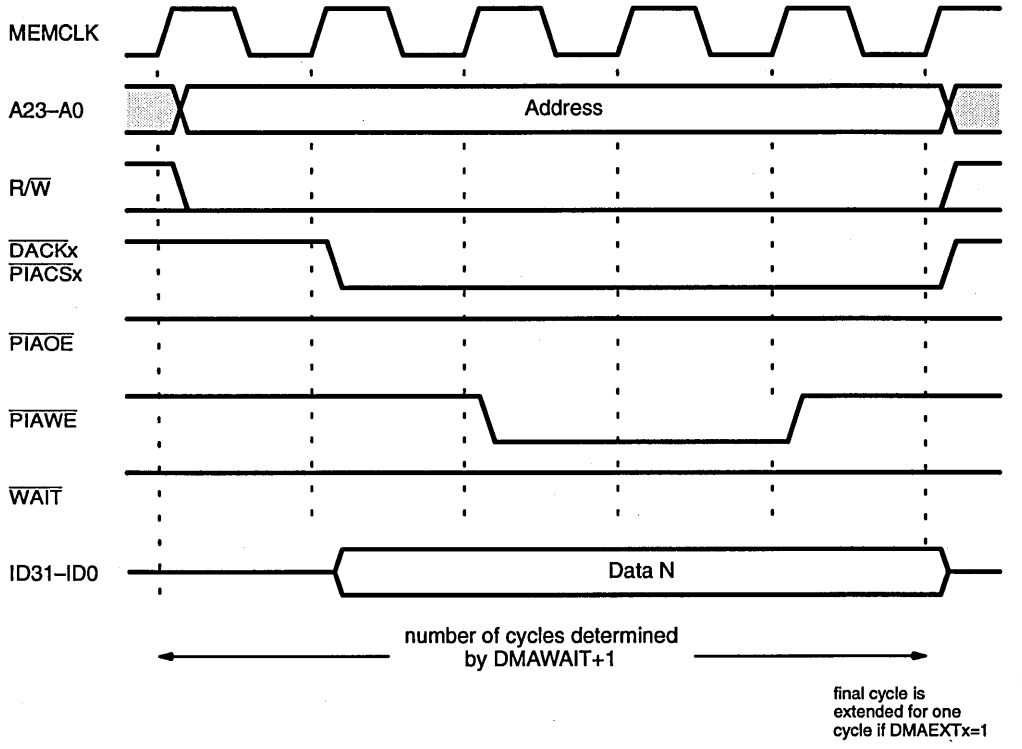


Figure 14-7 DMA Write Cycle



provide data hold time. If DMAEXT=1, the processor inserts one more cycle of data hold time after a write access. The peripheral can use WAIT to extend the access.

If the TDMO bit of the DMA Control Register is 0, an external peripheral can assert the TDMA signal at any time while DACKx is asserted to terminate the transfer, if the DMA channel's TTE bit is 1 and the transfer is not a fly-by transfer. TDMA is ignored during fly-by transfers. If the TDMO bit is 1, the processor asserts TDMA during the final transfer as determined by the DMA count being 0.

The DMA channel continues to perform transfers until the count expires or the TDMA input is asserted (depending on the CTE, TTE, and TDMO bits). When the transfer terminates, the EN bit is reset unless there is an active queued transfer, as explained in Section 14.4.

14.3.4 Latching External DMA Requests

The DMA controller is designed to latch an active transition of the external DREQ line, even if such a transition occurs when the DMA is disabled. This latching occurs for both edge- and level-triggered modes. The latched transition will then be recognized when the DMA channel is enabled, assuming the DRM field has not changed. This latching avoids a problem when using edge-sensitive DMA requests. There is the potential to lose a request between the time a transfer terminates on the count going to zero (which automatically disables the channel, blocking further requests) and the time the DMA interrupt handler restarts the channel.

Any programming of the DMA Control Register that changes the value of the DRM field from its previously programmed value will clear any latched request. Thus, to re-enable a DMA channel and also clear any latched request, the respective DMA Control Register must be written twice. With the first write, the DMA should remain disabled, and a value different from the desired DRM value should be set in the DRM field. On the second write, the DMA should be enabled, and the desired value should be set in the DRM field.

Upon reset, the DRM field is set to 00 (Active Low). Therefore, if the DMA is later enabled with DRM still at 00, any Active Low transition of DREQ since reset will have been latched and will be considered an active request when the DMA is enabled. To clear any such latched request, as noted above, the DMA Control Register should be written twice, once with DMA disabled and DRM=11 (or 10 or 01), and finally with DMA enabled and DRM=00.

14.4 DMA QUEUING

The address and count registers for all DMA channels consist of a two-entry queue, with each entry of the queue separately addressable for loading a new transfer. The DMA Address Register and DMA Count Register are at the head of the queue. The DMA Address Tail Register and DMA Count Tail Register are at the tail of the queue. 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 DMA 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 DMA Count Tail Register, then loading the DMA 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 DMA 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 isG 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.

14.5 FLY-BY DMA

Fly-by DMA transfers data directly between an external peripheral and DRAM or ROM, permitting very high data bandwidth. The transfer occurs at the rate of one 32-bit word per cycle, if DRAM page-mode accesses or ROM burst-mode or single-cycle accesses are enabled. Transfers of length less than one word are not supported. Because the Address Bus is used during a fly-by transfer, it cannot be used to address the peripheral. Also, the peripheral interface must support the special request/acknowledge protocol used for a fly-by transfer. The TDMA input is ignored during fly-by transfers, although TDMA can be used as an output.

If a DMA channel is programmed for fly-by transfers, the corresponding DMA request must be level sensitive. The timing diagrams in this section show active-Low requests for illustration, but active-High requests may also be used. In general, fly-by DMA causes a burst transfer between the peripheral and memory, using page-mode accesses if they are enabled. The DMA request initiates the burst and causes burst continuation, and the DMA acknowledge indicates individual transfers.

14.5.1 Fly-By DRAM Accesses

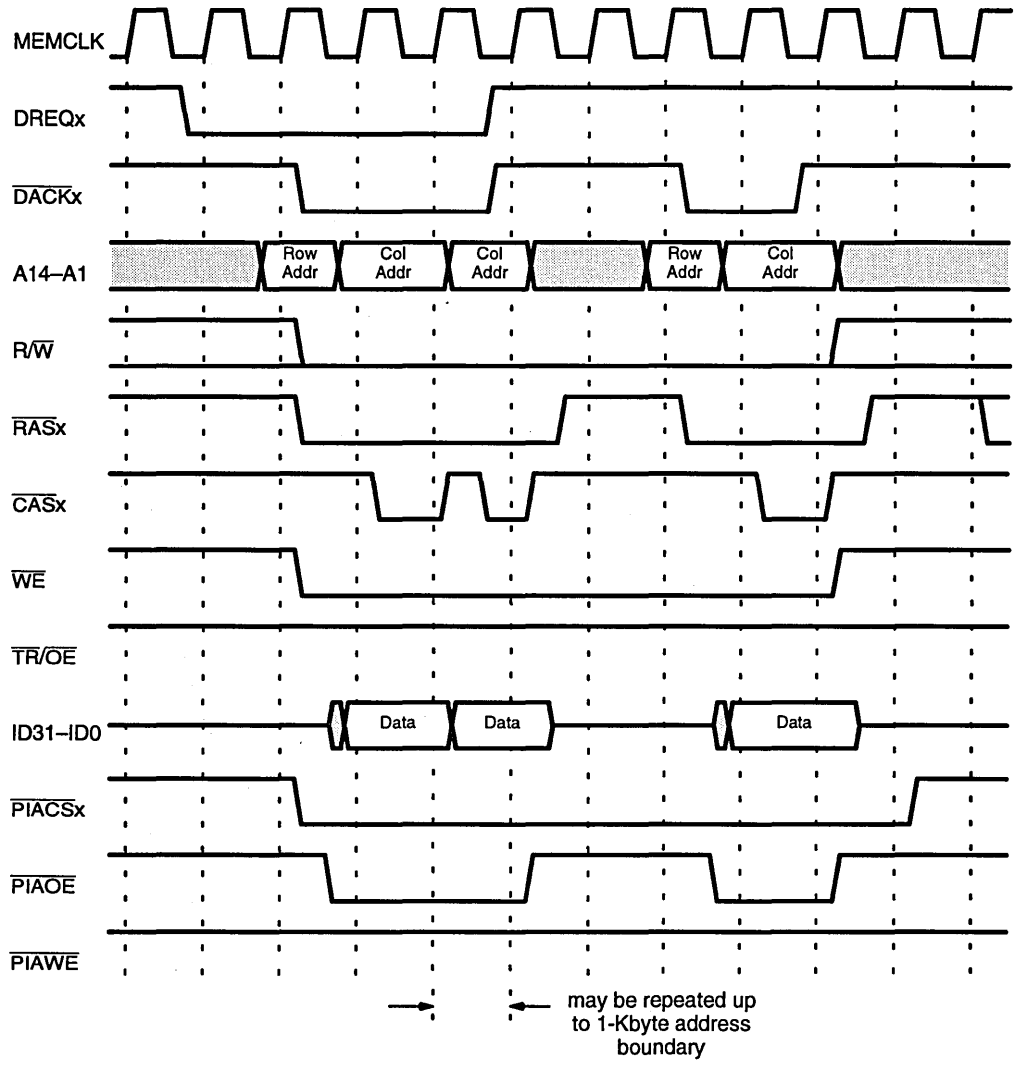
Figure 14-8 shows the timing of a fly-by DMA read (performed when the RW bit is 0). The DREQx signal is asserted to request the transfer. The processor then begins a DRAM write and asserts $\overline{\text{PIACS}}$x if required. The processor asserts the $\overline{\text{DACK}}$x signal the cycle before the peripheral must drive data on the ID Bus and asserts the $\overline{\text{PIAOE}}$ signal in the second half of the same cycle to enable peripheral data onto the bus.

The $\overline{\text{DACK}}$x signal has two purposes during the fly-by transfer. $\overline{\text{DACK}}$x is Low at the falling edge of MEMCLK to acknowledge the current pending transfer request, if there is an active pending request. $\overline{\text{DACK}}$x is Low at the rising edge of MEMCLK to indicate that data will be required in the next cycle. When the processor acknowledges a transfer ($\overline{\text{DACK}}$x Low at the falling edge of MEMCLK) the peripheral can request one more transfer by keeping DREQx active at the next rising edge of MEMCLK (one half cycle later). This additional transfer is acknowledged also by $\overline{\text{DACK}}$x being Low at a falling edge of MEMCLK (possibly as soon as one half-cycle later), and the data is required the cycle after $\overline{\text{DACK}}$x is Low at the rising edge of MEMCLK (not necessarily immediately following the acknowledgement). The processor always asserts $\overline{\text{PIAOE}}$ in time to enable peripheral data onto the ID Bus.

After the first transfer, additional transfers are performed as page-mode accesses, if possible. The $\overline{\text{DACK}}$x and $\overline{\text{PIAOE}}$ signals remain asserted for more than one cycle if a page-mode access is used, because the page-mode access takes a single cycle. If the processor cannot perform a page-mode access, either because page-mode accesses are disabled or because some other event keeps the processor from continuing the DMA (such as a 1-Kbyte address boundary crossing), $\overline{\text{DACK}}$x and $\overline{\text{PIAOE}}$ are deasserted for one or more cycles. One more transfer is performed whenever DREQx is active at the end of a cycle in which $\overline{\text{DACK}}$x is asserted to acknowledge a pending request.

Figure 14-9 shows the timing of a DMA write (performed when the RW bit is 1). The transfer protocol using DREQx and $\overline{\text{DACK}}$x is identical to the protocol for DMA reads. However, the processor asserts $\overline{\text{DACK}}$x at the rising edge of MEMCLK to indicate that data will be valid at the end of the next cycle (rather than needed) and pulses $\overline{\text{PIAWE}}$ during the second half of the cycle in which data is valid. It is important to note that the processor samples data with the rising edge of $\overline{\text{CAS}}$, so a peripheral that uses MEMCLK to latch data sees a different data setup time than the processor does. Also,

Figure 14-8 Fly-By DMA Reads (read peripheral, write DRAM)



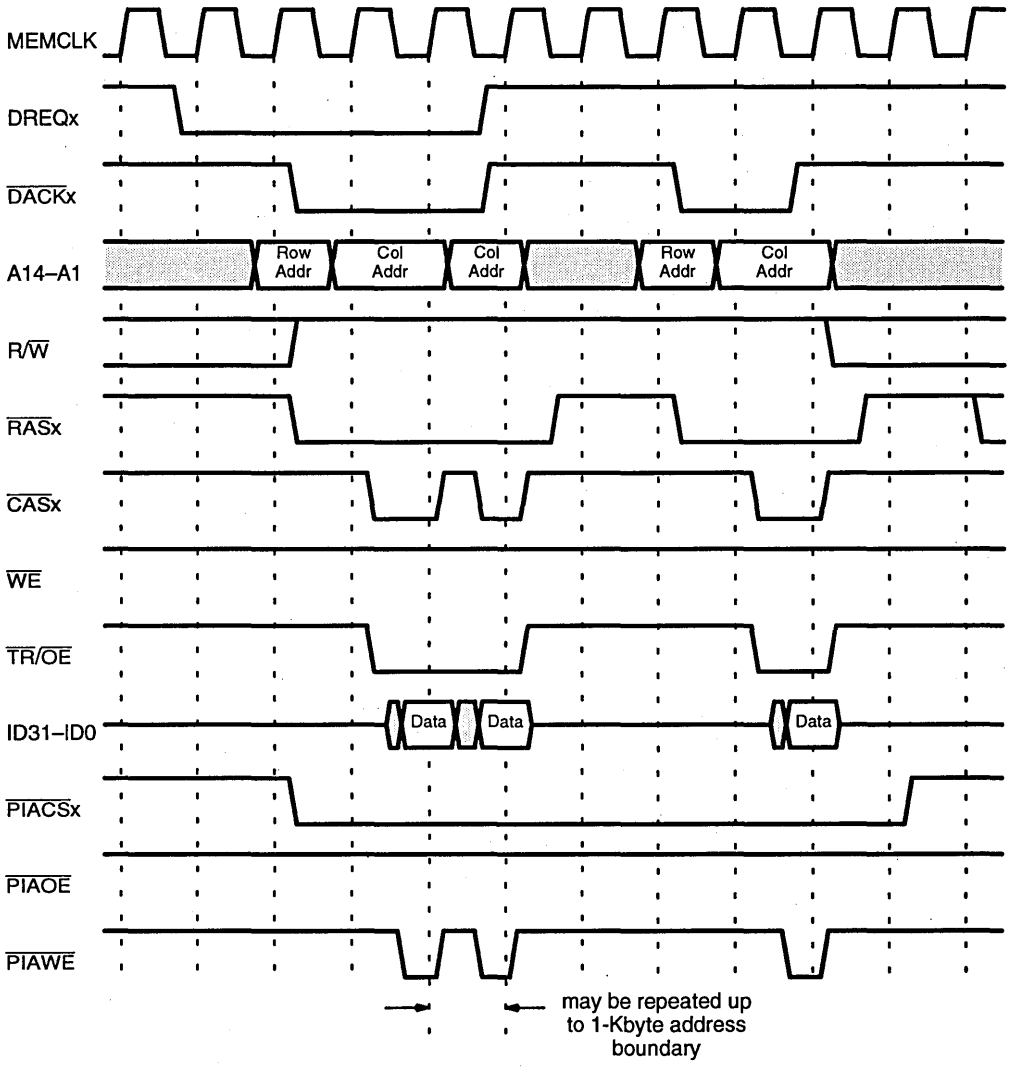
to permit single-cycle transfer rates, the fly-by DMA transfers do not provide as much data hold time to the peripheral as do normal DMA transfers.

Parity generation and checking cannot be performed during fly-by DMA transfers.

14.5.2 Fly-By ROM Accesses

Fly-by DMA can be used to access the ROM address space if the DMA channel is configured to access ROM. Figure 14-10 shows the timing of a fly-by DMA read to ROM (performed when the RW bit is 0), assuming that a ROM access takes two cycles (single-cycle ROM writes are not supported). As with DRAM transfers, the

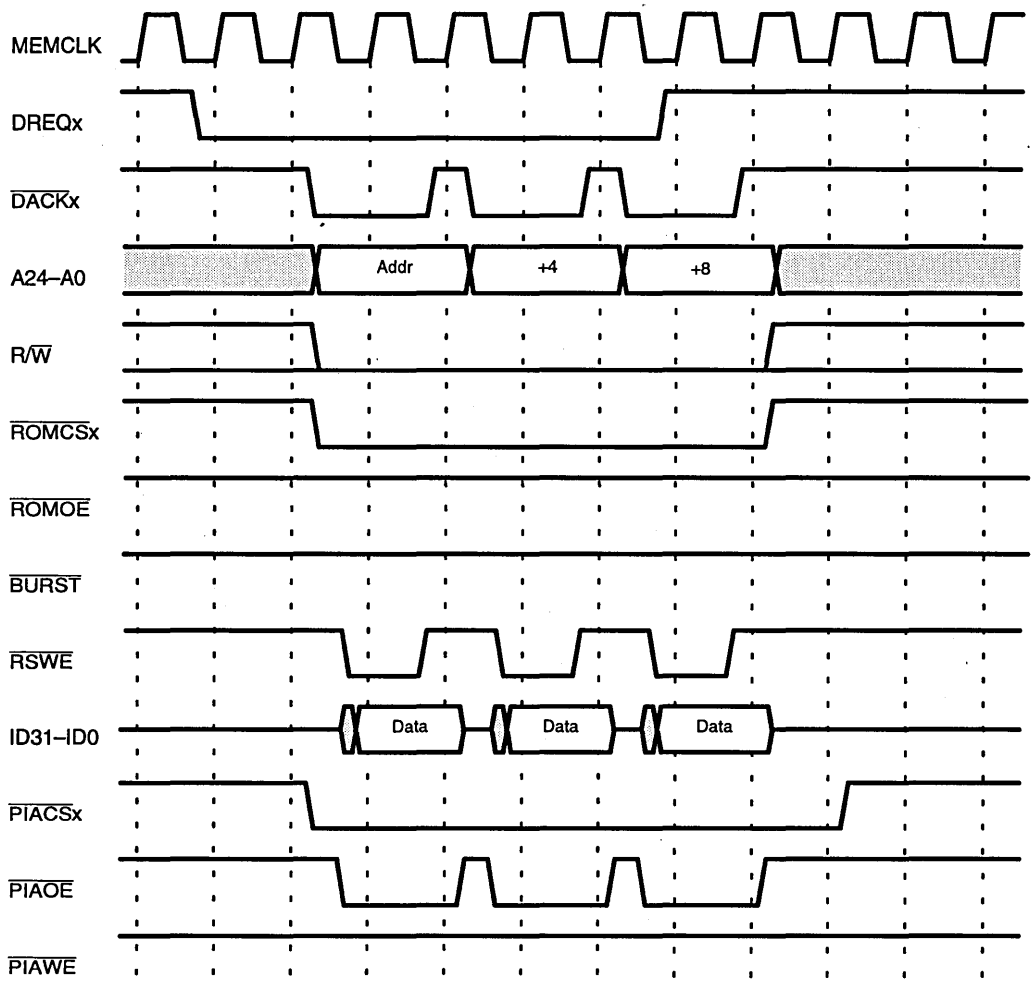
Figure 14-9 Fly-By DMA Writes (read DRAM, write peripheral)



DREQx signal is asserted to request the transfer. The processor then begins the ROM write and asserts PIACSx if required. Also, the processor uses the DACKx signal in the same manner to acknowledge pending requests and to indicate the actual data transfer.

Figure 14-11 shows the timing of a DMA write to ROM (performed when the RW bit is 1), assuming that a ROM access takes two cycles. Because of protocol limitations, fly-by DMA cannot support single-cycle (zero-wait-state) ROMs. The transfer protocol using DREQx and DACKx is identical to the protocol for DMA reads. However, the processor asserts DACKx to indicate that data will be valid at the end of the next cycle (rather than needed) and pulses PIAWE during the second half of the cycle in which

Figure 14-10 Fly-By DMA Reads (read peripheral, write ROM)—Two-Cycle ROM



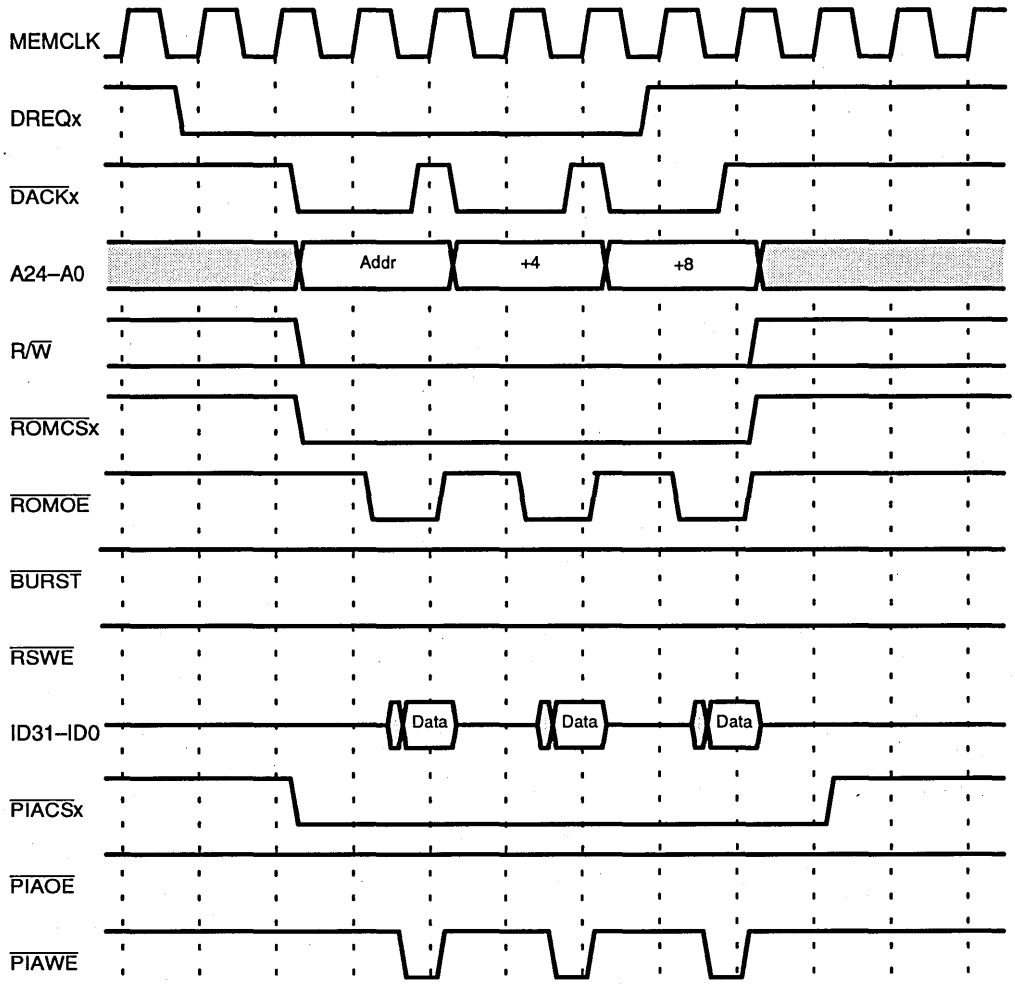
data is valid. The fly-by DMA transfers do not provide as much data hold time to the peripheral as do normal DMA transfers.

Figure 14-12 shows the timing of a DMA write to ROM assuming a burst-mode ROM. The timing is similar to that of DMA write using page-mode DRAM accesses.

14.6 RANDOM DIRECT MEMORY ACCESS BY EXTERNAL DEVICES

The Am29240 microcontroller series 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 GREQ and GACK signals to perform a random memory access via the DRAM or

Figure 14-11 Fly-By DMA Writes (read ROM, write peripheral)—Two-Cycle ROM

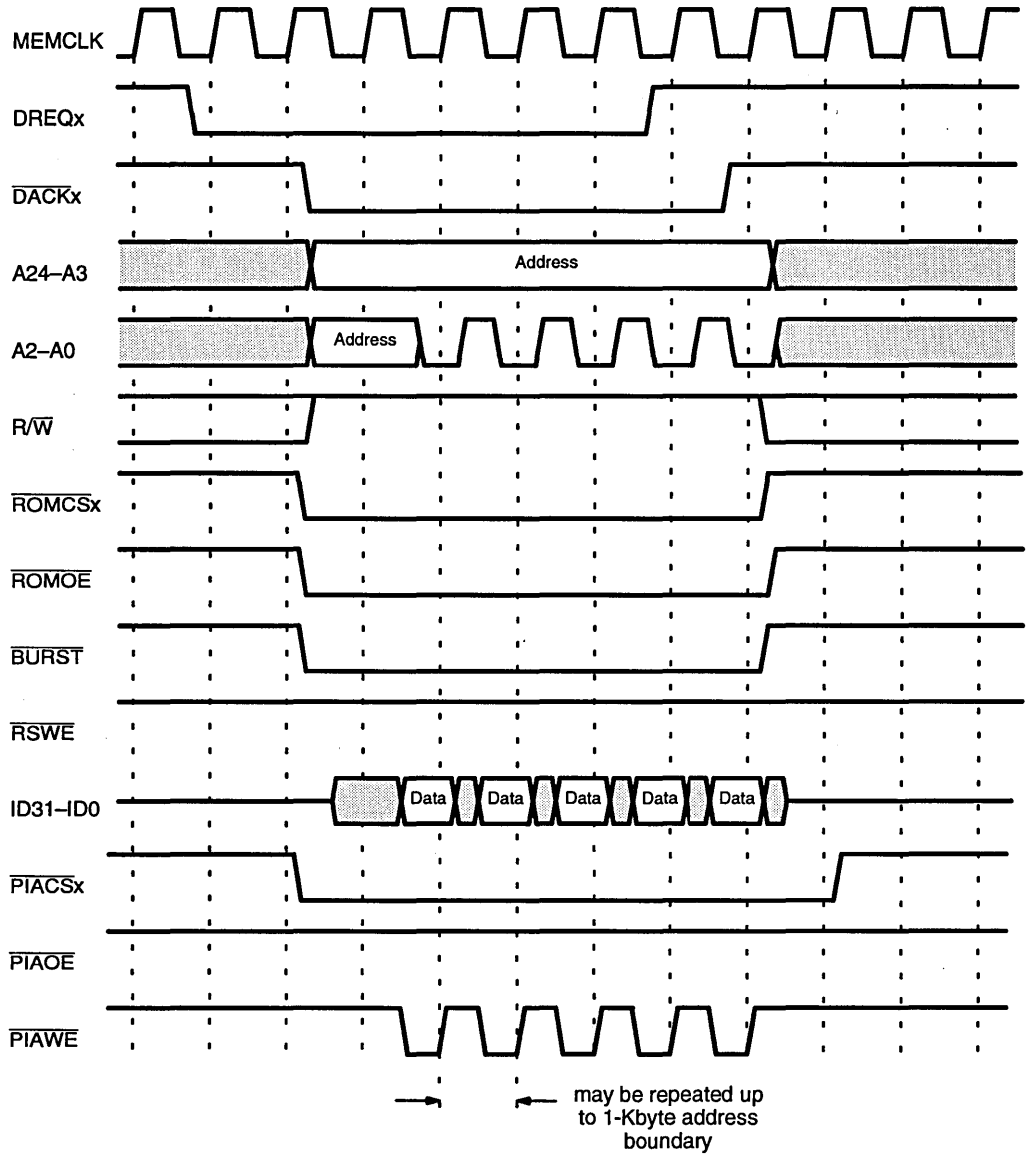


ROM controller. These external random accesses may be either single or burst-mode accesses.

14.6.1 Single External Access

Figure 14-13 shows the timing for a memory read using GREQ and GACK. The external device indicates that it wants to perform a memory access by asserting GREQ. As soon as the processor can perform the access, it asserts GACK. The external device can place the memory address on ID31–ID0 during any cycle following the assertion of GACK: the device indicates that the address is valid by deasserting 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 14-13 shows an access to DRAM, as an example. The

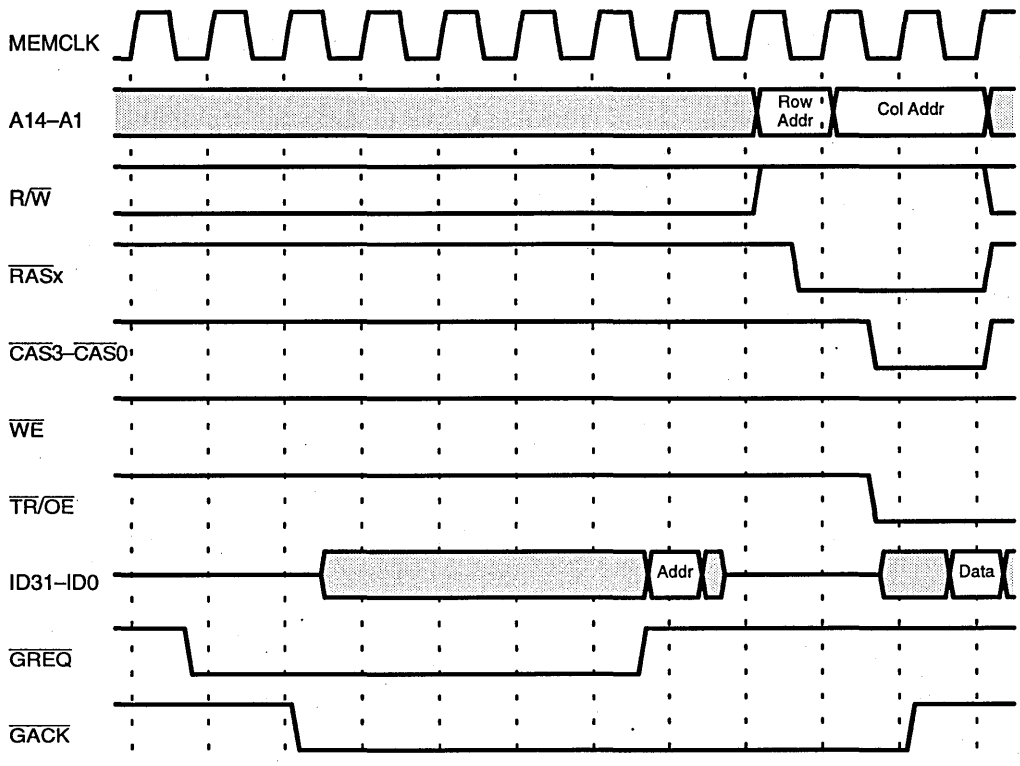
Figure 14-12 Fly-By DMA Writes (read ROM, write peripheral)—Burst-Mode ROM



processor 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 14-14 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 ID31–ID0. 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

Figure 14-13 External Random DRAM Read Cycle



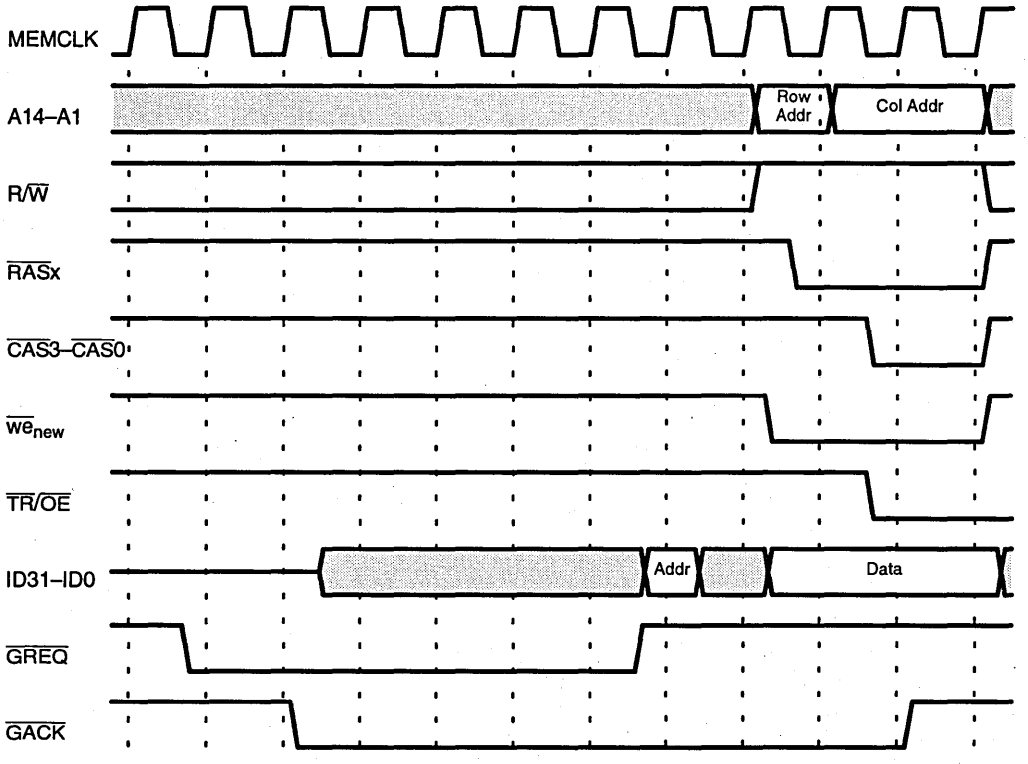
and writes must be made by external logic (which, for example, forms the signal $\overline{we}_{new}$ in Figure 14-14) in a way that meets the memory timing requirements. For example, an AND gate can be used to form the negative OR of the processor's WE signal and the write enable from the external device.

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 ID31–ID0 and deasserts GREQ to indicate that the address is valid.
3. For a read, the device must be able to latch data from ID31–ID0 at the end of the cycle in which GACK is deasserted. For a write, the device must be prepared to drive data on ID31–ID0 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.

To further clarify the use of GREQ and GACK, Figure 14-15 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.

Figure 14-14 External Random DRAM Write Cycle



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.

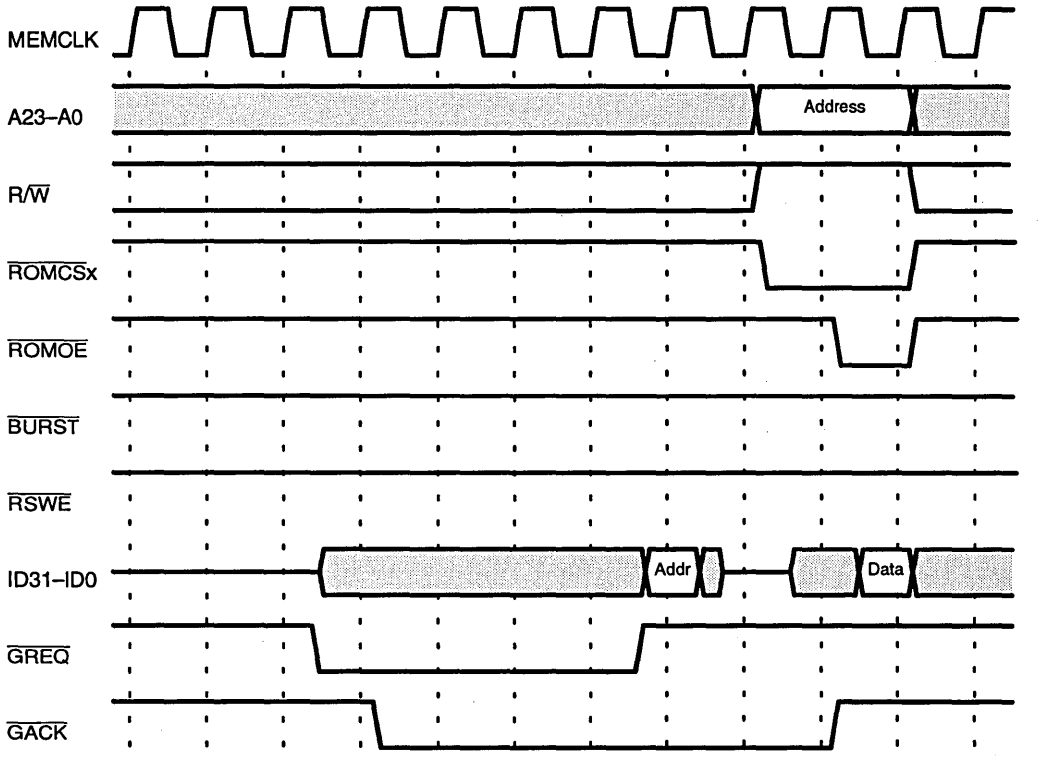
During a processor reset, the GREQ input may be used by a hardware-development system to force processor outputs to the high-impedance state. To prevent driver conflicts, the system should keep GREQ in a high-impedance state during a processor reset.

14.6.2 Burst-Mode External Access

The GREQ/GACK protocol permits an external device to perform random accesses to DRAM or ROM. This protocol is expanded to permit burst-mode transfers.

Burst GREQ/GACK transfers require the use of a set of DMA channel registers and require the use of DREQA to differentiate a burst access from a single access. To permit burst transfers, the selected DMA Control register is programmed with a DRS value of 110, a FLY bit of 1, and a DRM field of 00 or 01 (active Low or High). The corresponding DMA Count register is programmed with the count of the number of transfers to be performed by each burst transfer. As with DMA transfer counts, this count is zero-based. The count is fixed, and small count values should be used because the burst transfer cannot be preempted or interrupted. To get the full benefit

Figure 14-15 External Random ROM Read Cycle

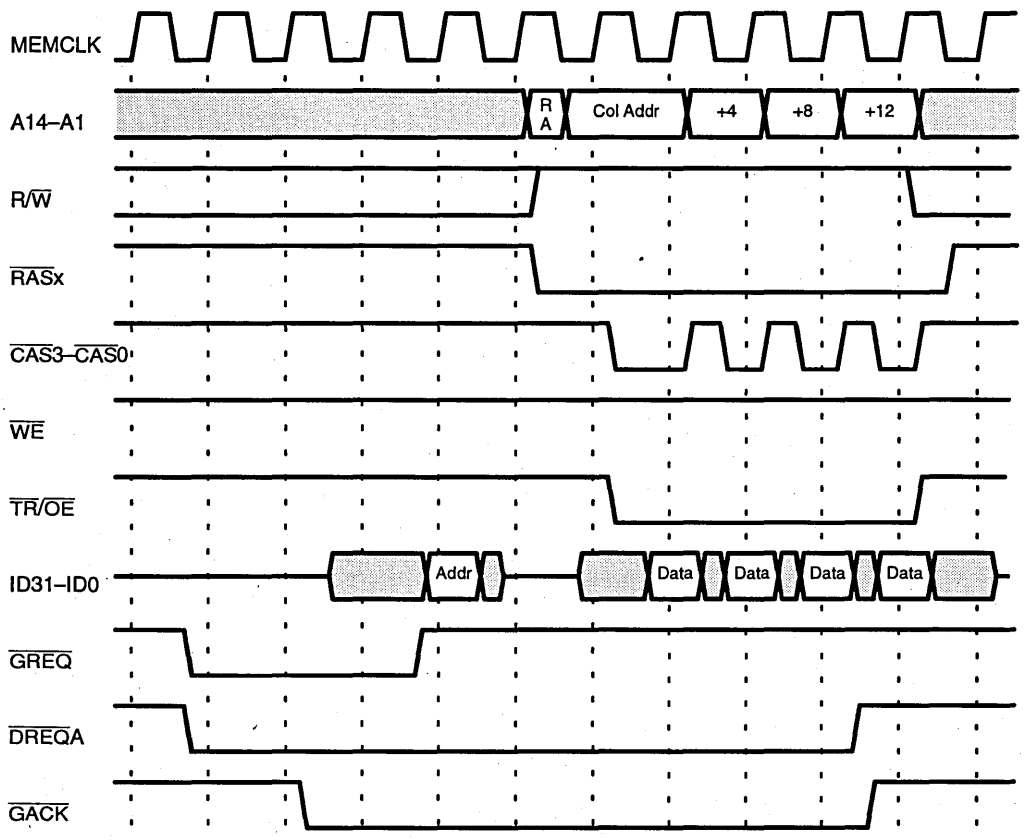


of the burst transfer, the DRAM should be programmed for page-mode accesses (if the transfer is to or from DRAM) or the ROM should be programmed for burst or single-cycle accesses (if the transfer is from ROM).

Figure 14-16 shows the timing of an external burst DRAM read, assuming that the DMA Count register is programmed for four transfers per request (the actual count value is 3, since it is zero-based). This timing diagram also assumes that the DRAM is programmed for page-mode accesses. The external master requests a burst transfer by asserting GREQ and DREQA at the same time (DREQA is shown active Low). If DREQA is not asserted, or if no DMA channel is programmed to support the request, then a single access is performed. In response to the request, the processor asserts GACK, and the external master then deasserts GREQ in the cycle that it provides the DRAM address. The processor begins the burst transfer two cycles later, and provides the first data word three cycles later. Additional words are provided at the rate of one per cycle, and the processor deasserts GACK during the cycle that the final word is transferred. Unlike single GREQ/GACK transfers, GACK deasserts in the second half of the MEMCLK cycle, rather than the first half. Note that the processor does not explicitly indicate the transfer of individual words.

DREQA operates in a manner very similar to that of a DMA request during a fly-by transfer. DREQA must be active to request each access in the burst sequence. In each cycle that the processor completes an access, DREQA must be active to request

Figure 14-16 Burst $\overline{\text{GREQ}}$/$\overline{\text{GACK}}$ DRAM Read (DMA Count=3)



one more access (up to the total count). If DREQA is inactive in any cycle that an access completes, the access is cancelled.

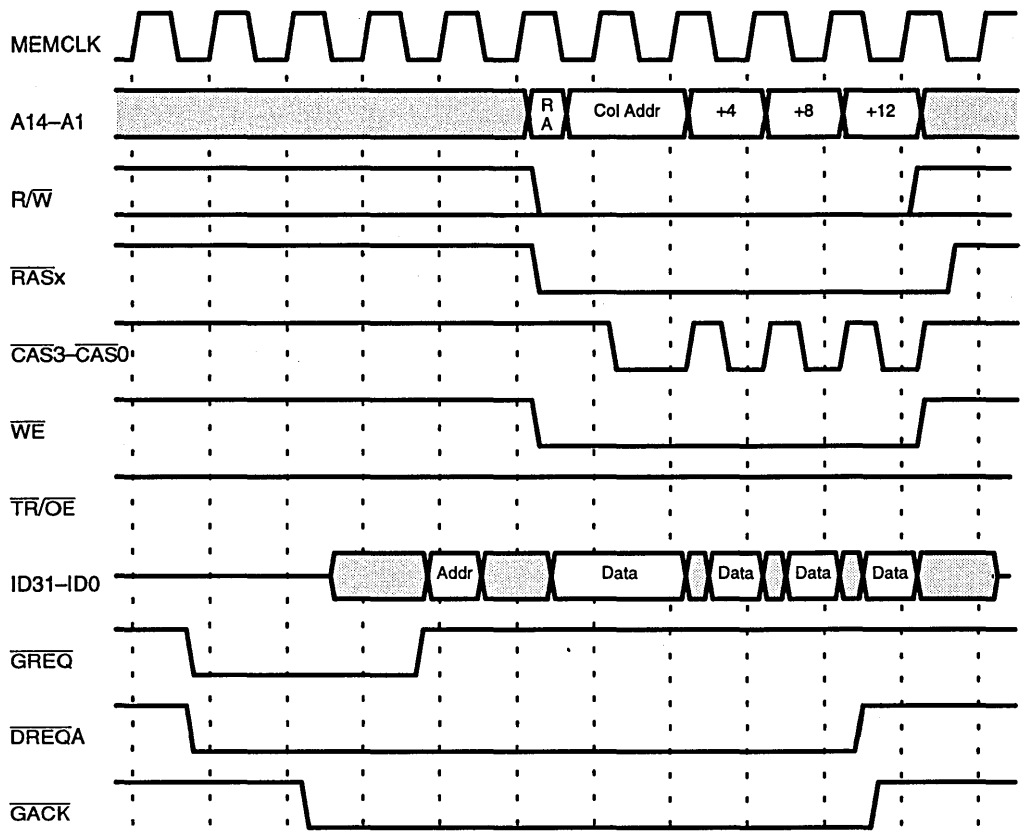
Because the burst transfer is controlled by a DMA channel, burst transfers provide certain features that are not available for single transfers. The R/$\overline{\text{W}}$ output indicates whether the access is a read or write, as controlled by the RW bit in the DMA Control Register. The processor also generates the appropriate write signals ($\overline{\text{WE}}$ for DRAM or $\overline{\text{RSWE}}$ for ROM), again as controlled by the RW bit.

Figure 14-17 shows the timing of an external burst DRAM write. This is identical to a DRAM read, except that the system provides the data and the processor asserts $\overline{\text{WE}}$.

Figure 14-18 shows the timing of an external burst ROM read, assuming that the ROM is a burst-mode ROM. Figure 14-19 shows the timing of an external burst ROM read, assuming that the ROM is enabled to perform single-cycle accesses.

Figure 14-20 shows the timing of an external burst ROM write, assuming that the ROM is enabled to perform two-cycle accesses (single-cycle writes are not supported, nor are writes to burst-mode ROMs). The system provides the data for the write.

Figure 14-17 Burst $\overline{\text{GREQ}}$/$\overline{\text{GACK}}$ DRAM Write (DMA Count=3)



The processor does not perform a burst access across a 1-Kbyte address boundary. If the external master provides an address that causes the processor to cross a 1-KByte address boundary during the burst, the processor performs all accesses up to the point of crossing the boundary, then cancels the burst access. The access does not resume beyond that point. Thus, crossing a 1-Kbyte address boundary can result in the external master receiving fewer words than expected.

14.6.3 Single External Access Controlled by DMA Channel

Because burst-mode transfers can have a burst length of one, a DMA channel can be configured to perform single transfers—this is accomplished by setting the appropriate DMA Count field to zero. This permits the DMA channel to provide the necessary bus steering and write strobes, in contrast to the single accesses described in Section 14.6.1. The R/$\overline{\text{W}}$ output indicates whether the access is a read or write, as controlled by the RW bit in the DMA Control Register. The processor also generates the appropriate write signals ($\overline{\text{WE}}$ for DRAM or $\overline{\text{RSWE}}$ for ROM), again as controlled by the RW bit. However, this benefit is at the cost of a DMA channel.

Figure 14-18 Burst $\overline{\text{GREQ}}$/$\overline{\text{GACK}}$ Burst-Mode ROM Read (DMA Count=3)

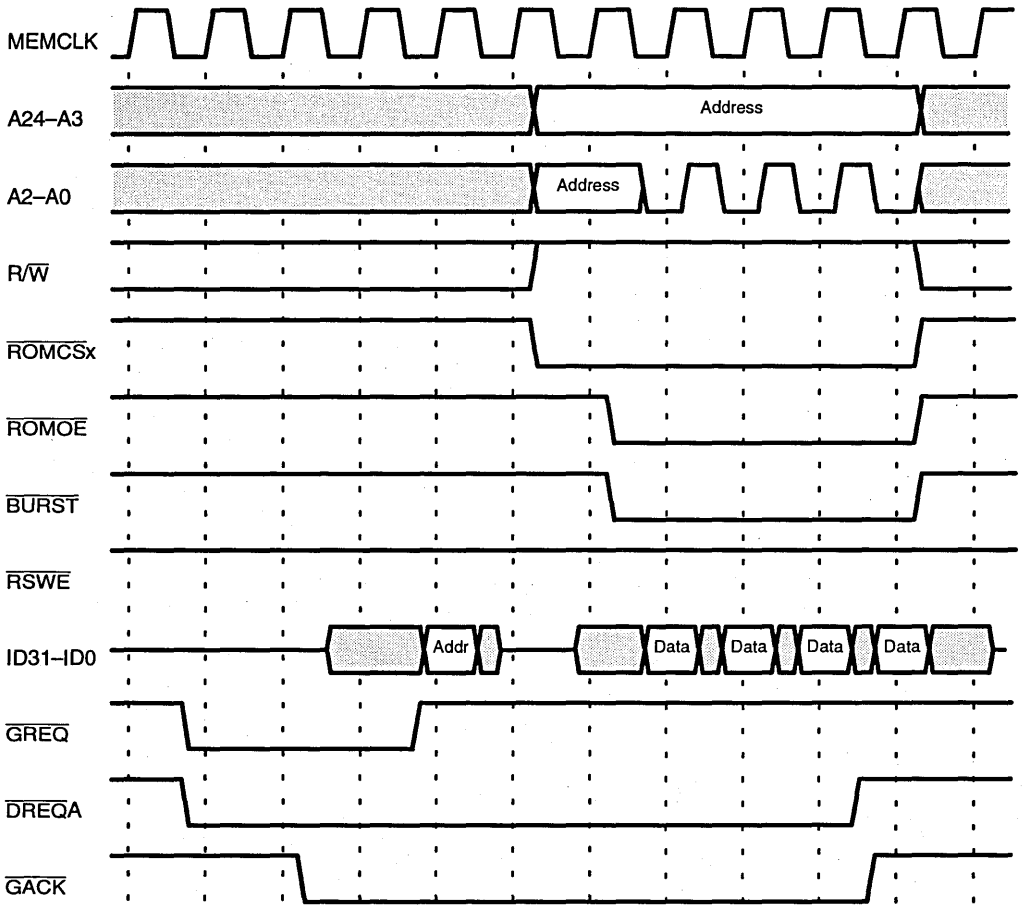


Figure 14-19 Burst $\overline{\text{GREQ}}$/$\overline{\text{GACK}}$ Single-Cycle ROM Read (DMA Count=3)

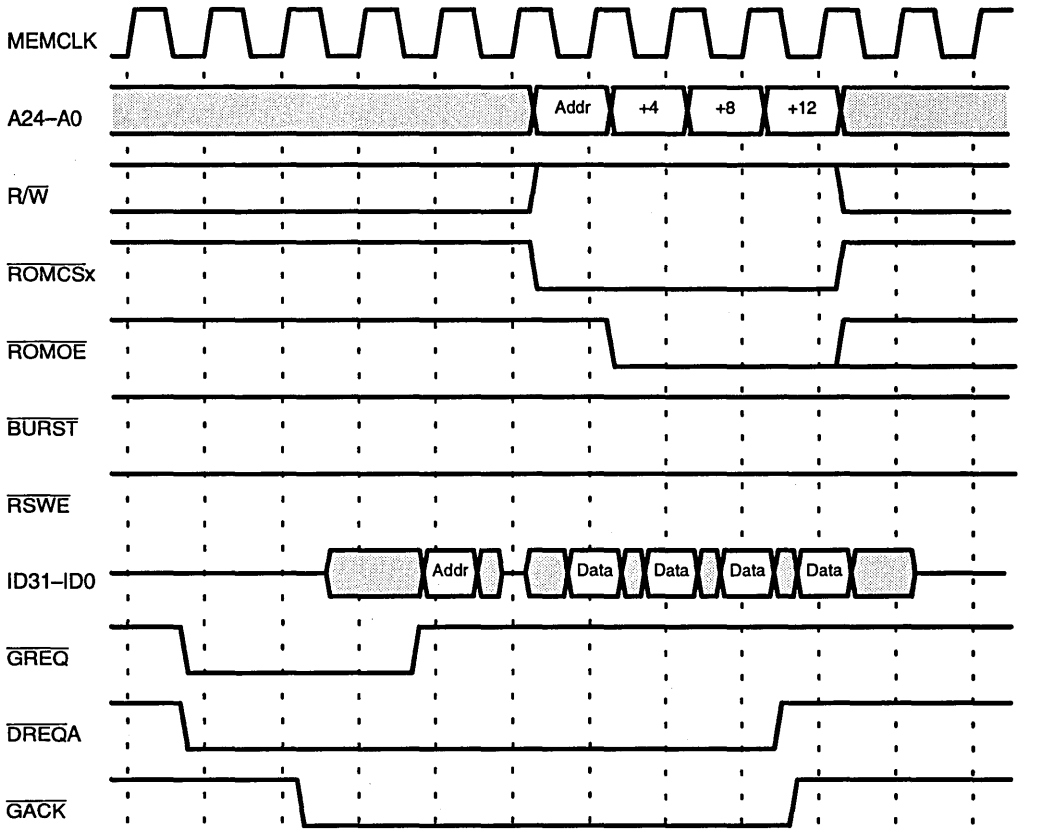
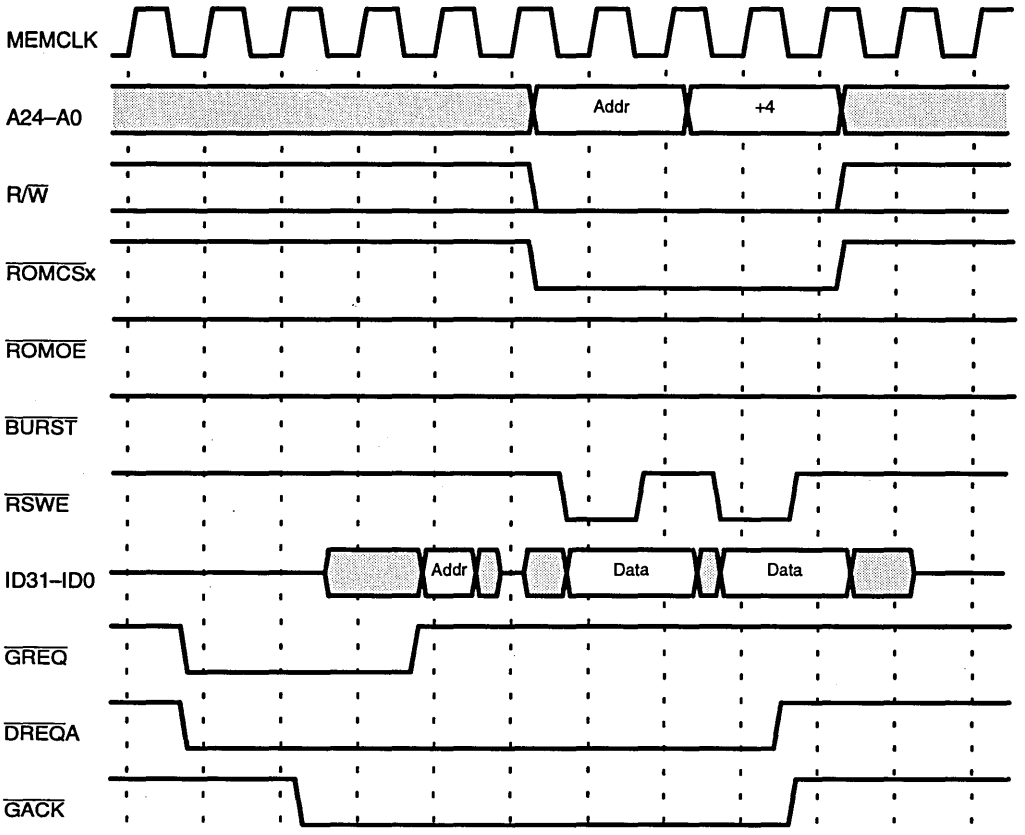


Figure 14-20 Burst $\overline{\text{GREQ}}$/$\overline{\text{GACK}}$ Two-Cycle ROM Write (DMA Count=1)



CHAPTER

15 PROGRAMMABLE I/O PORT

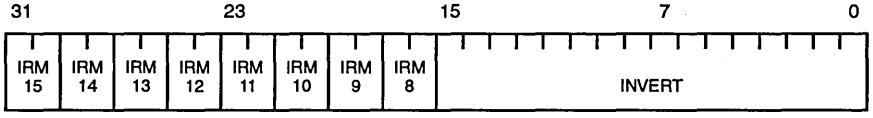
The I/O port permits direct programmable access to the sixteen external signals, PIO15–PIO0, as either inputs, outputs, or open-drain signals. Eight of these signals, PIO15–PIO8, can be programmed to cause interrupts. The I/O port also permits system configuration information to be loaded during a processor reset.

15.1 PROGRAMMABLE REGISTERS

15.1.1 PIO Control Register (POCT, Address 800000D0)

The PIO Control Register (Figure 15-1) controls interrupt generation and determines the polarity of PIO15–PIO0.

Figure 15-1 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 16.1.1), 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.

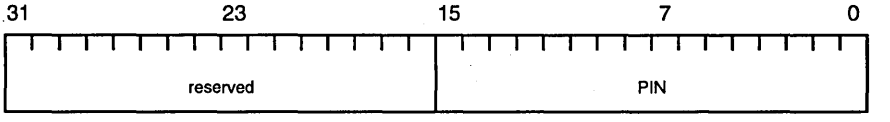
Bits 29–16: IRM14 through IRM8—The IRM14–IRM8 fields enable interrupts and specify level- or edge-sensitivity for PIO14–PIO8, respectively. These fields are identical in definition to IRM15, except that the value 11 is reserved.

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.

15.1.2 PIO Input Register (PIN, Address 800000D4)

The PIO Input Register (Figure 15-2) reflects the external levels on the PIO15– PIO0 signals.

Figure 15-2 PIO Input Register



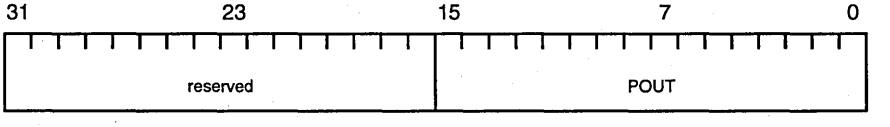
Bits 31–16: Reserved

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.

15.1.3 PIO Output Register (POUT, Address 800000D8)

The PIO Output Register (Figure 15-3) determines the levels driven on the PIO15–PIO0 signals, for those signals enabled to be driven by the PIO Output Enable Register.

Figure 15-3 PIO Output Register



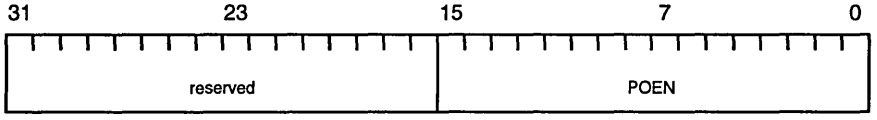
Bits 31–16: Reserved

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.

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

The PIO Output Enable Register (Figure 15-4) determines whether or not the PIO15–PIO0 signals are driven as outputs.

Figure 15-4 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.

15.1.5 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.

The I/O port permits system configuration information to be loaded during a processor reset. During reset, all PIO signals are configured as inputs. The PIO Input Register latches the state of these inputs during a reset and holds this value when $\overline{\text{RESET}}$ is deasserted. The version of $\overline{\text{RESET}}$ used to latch the PIO Input Register comes from an early stage of the $\overline{\text{RESET}}$ synchronization logic, so that the data driven on the I/O Port during a reset can change with the deassertion of $\overline{\text{RESET}}$ (this would occur, for example, if the driver of the configuration information is enabled by $\overline{\text{RESET}}$).

The value latched in the PIO Input Register during a reset is held until the first read or write of any I/O Port Register. This provides time for software to read the configuration information, while remaining compatible with older Am29200 and Am29205 software that would expect the I/O port to operate normally after it had been configured.

15.2 OPERATING THE I/O PORT

The PIO15–PIO0 signals are asynchronous to the processor. A change on PIO15–PIO0 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 PIO15–PIO0 a maximum of one cycle after the change occurs. The PIO15– PIO0 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

16 PARALLEL PORT

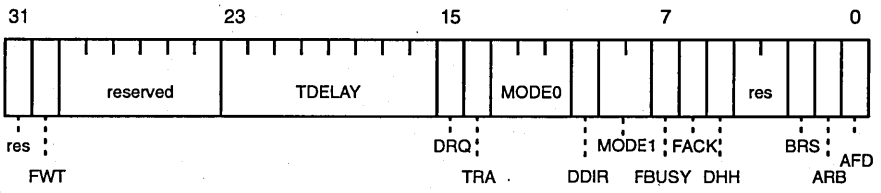
The parallel port connects a host processor to one of the Am29240 microcontrollers. It supports data transfers from host to processor or from processor to host.

16.1 PROGRAMMABLE REGISTERS

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

The Parallel Port Control Register (Figure 16-1) controls the parallel port.

Figure 16-1 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. On transfers from the host, the TDELAY field specifies one less than the number of MEMCLK cycles in the duration interval; in this case, setting TDELAY to 0 will cause PACK to assert for one cycle. On transfers to the host, the TDELAY field specifies the number of MEMCLK cycles in the duration interval. On transfers to the host, if TDELAY is 0, PACK will not assert at all.

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 either the MODE0 or MODE1 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 a hung transfer.

Bits 13–11: Parallel Port Mode 0 (MODE0)—This field enables the parallel port and controls the operational mode of the parallel port, as follows:

MODE0 Value	Effect on Parallel Port
000	Disabled
001	Generate interrupt requests for service
010	Generate DMA Channel 0 requests
011	Generate DMA Channel 1 requests
100	Generate DMA Channel 2 requests
101	Generate DMA Channel 3 requests
110–111	Reserved

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 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. Either the MODE1 or the MODE0 field must be 00 when the DDIR bit is changed.

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.

Bits 9–8: Parallel Port Mode 1 (MODE1)—This field enables the parallel port and controls the operational mode of the parallel port, as follows:

MODE1 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

The MODE1 field is provided for compatibility with the Am29200 and Am29205 microcontrollers' MODE field.

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.

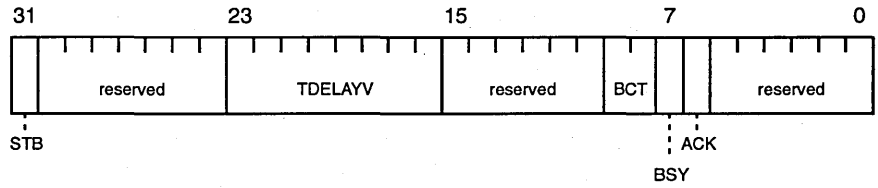
If ARB=1, the PACK pulse follows the $\overline{\text{PBUSY}}$ pulse in time. $\overline{\text{PBUSY}}$ is asserted in response to an assertion of PSTROBE and is deasserted when the Parallel Port Data Register has been read and PSTROBE is Low. PACK is asserted at the same time $\overline{\text{PBUSY}}$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

16.1.2 Parallel Port Status Register (PPST, Address 800000C8)

The Parallel Port Control Register (Figure 16-2) controls the parallel port. For compatibility with the Am29200 microcontroller, this register can also be accessed at address 800000C1.

Figure 16-2 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.

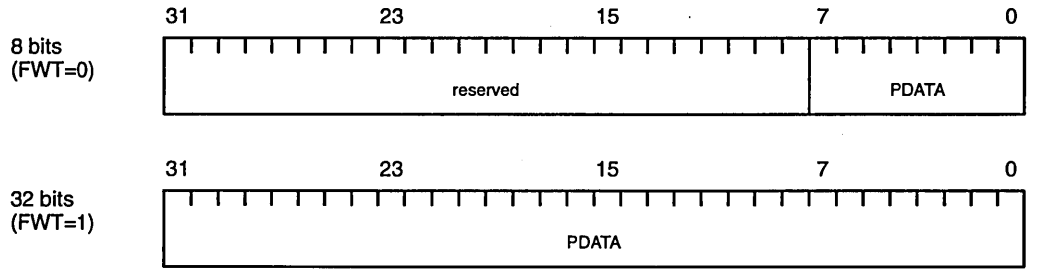
Bit 6: PACK Level (ACK)—This bit indicates the level on the PACK signal. If PACK is Low, this bit is 0; if PACK is High, this bit is 1.

Bits 5–0: Reserved

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

The Parallel Port Data Register (Figure 16-3) 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 16.2) and the access uses the signals $\overline{\text{POE}}$ and $\overline{\text{PWE}}$ to read and write the buffer.

Figure 16-3 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 processor 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.

16.1.4 Initialization

During a processor reset, both the MODE1 and MODE0 fields of the Parallel Port Control Register are reset to 00. The parallel port must be configured by software before the parallel port is enabled. The parallel port can be controlled either by the MODE0 or by the MODE1 fields, but the unused field must be zero.

Writing the value 00 into either the MODE1 or MODE0 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 15.1.1), 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.

16.2 PARALLEL PORT TRANSFERS

The parallel port does not attach directly to the processor, 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 processor 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.

16.2.1 Transfers from the Host

Figure 16-4 shows the state-transition diagram for transferring data from the host to the processor over the parallel port. Figure 16-5 through Figure 16-8 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.

The processor asserts $\overline{\text{PBUSY}}$ 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 processor asserts PACK at the same time as $\overline{\text{PBUSY}}$ if ARB=0. The parallel port then generates either an interrupt request or a DMA request, as controlled by either the MODE1 or MODE0 field, so the data can be read. If ARB=0, both $\overline{\text{PBUSY}}$ 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, $\overline{\text{PBUSY}}$ 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.

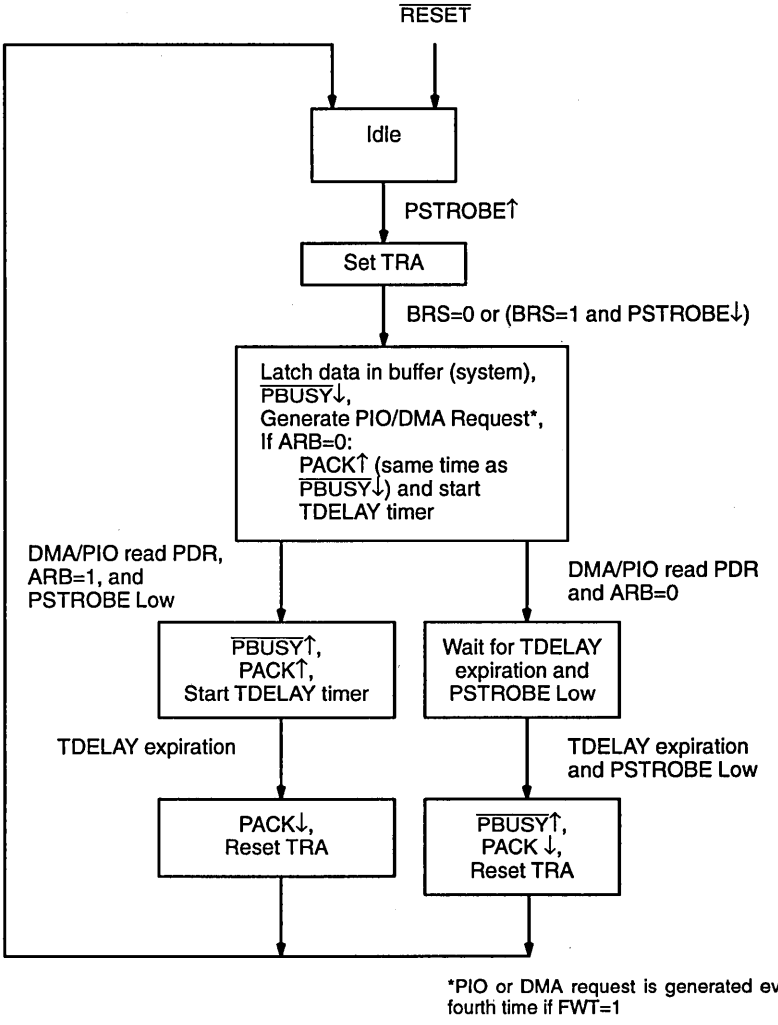
The PDR is mapped to the external buffer register. Figure 16-9 shows the timing of the external access. This external access is treated as either a DMA access or a processor PIA access for the purpose of prioritization with other accesses.

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.

16.2.2 Transfers to the Host

Figure 16-10 shows the state transition diagram for transferring data from the processor to the host over the parallel port. Figure 16-11 shows the 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 $\overline{\text{PBUSY}}$ output active, preventing the host from transferring data to the processor. Either the MODE1 or MODE0 bit must be 00 when the DDIR bit is set or reset.

Figure 16-4 State Transitions for Transfers from the Host



The processor begins the transfer by writing data to the external buffer. Figure 16-12 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. The parallel port automatically generates 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 $\overline{\text{PBUSY}}$ 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.

Figure 16-5 Transfer from the Host on the Parallel Port (BRS=0, ARB=0)

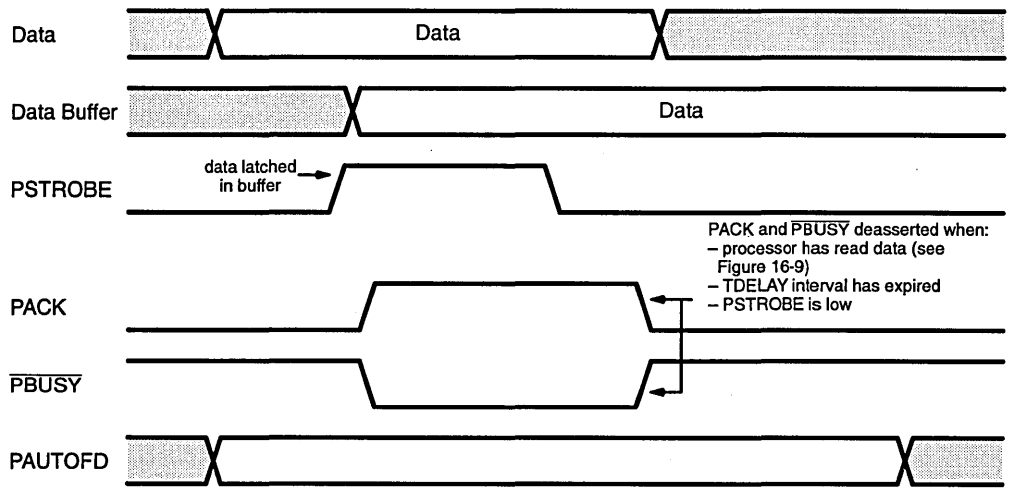
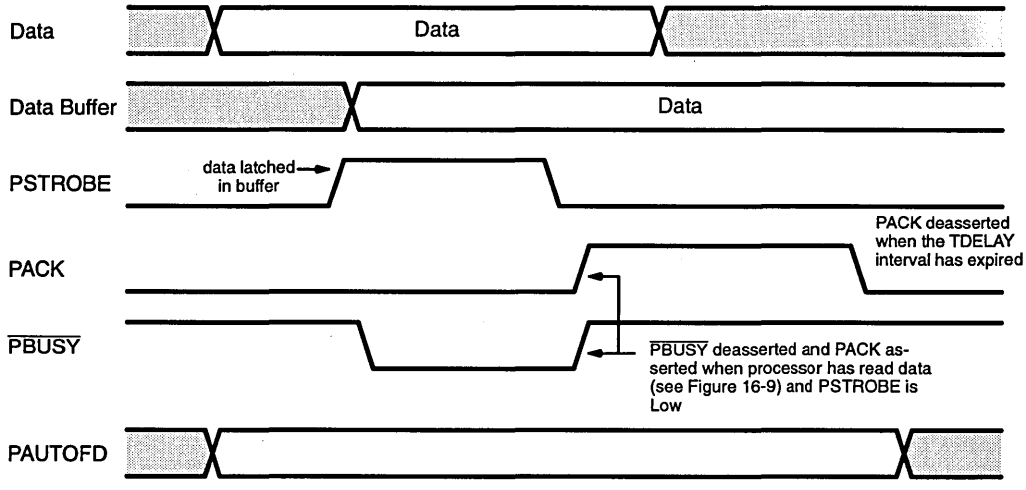


Figure 16-6 Transfer from the Host on the Parallel Port (BRS=0, ARB=1)



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 16-7 Transfer from the Host on the Parallel Port (BRS=1, ARB=0)

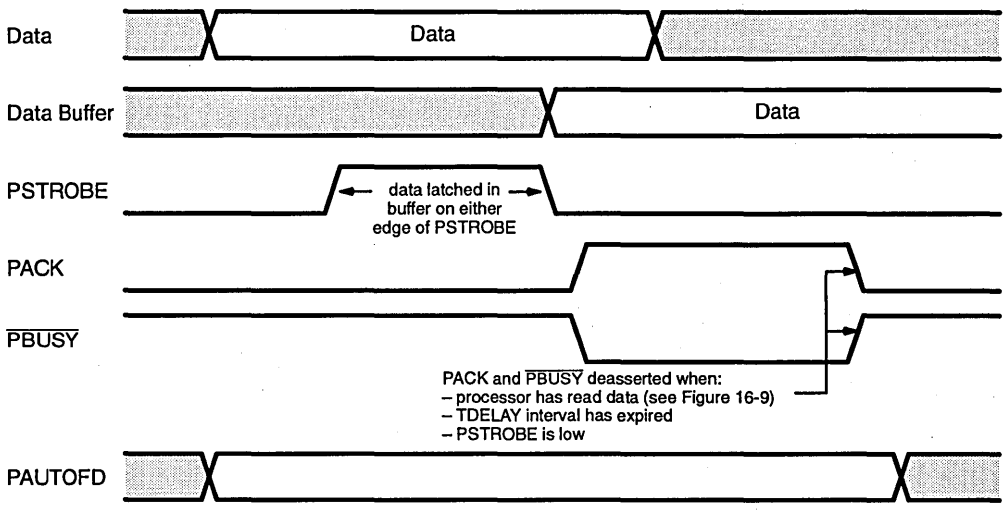


Figure 16-8 Transfer from the Host on the Parallel Port (BRS=1, ARB=1)

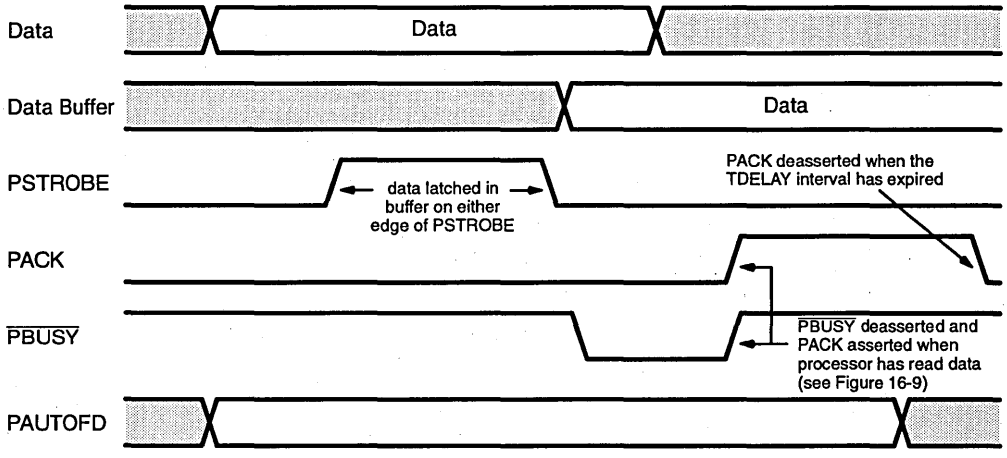


Figure 16-9 Parallel Port Buffer Read Cycle

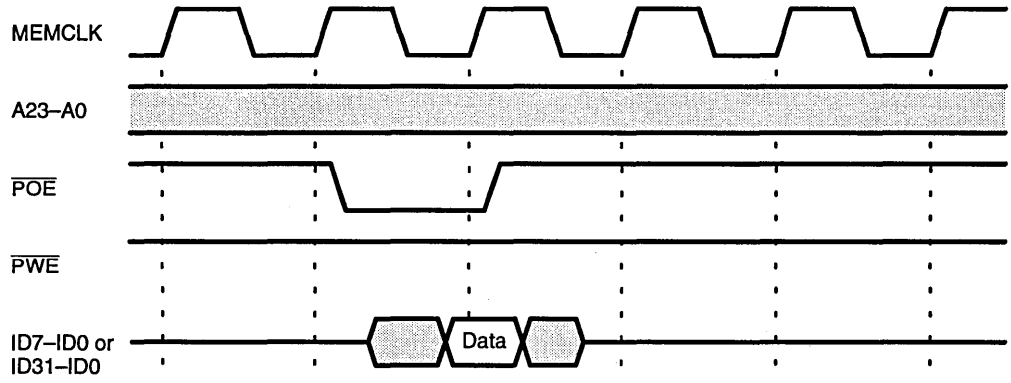


Figure 16-10 State Transitions for Transfers to the Host

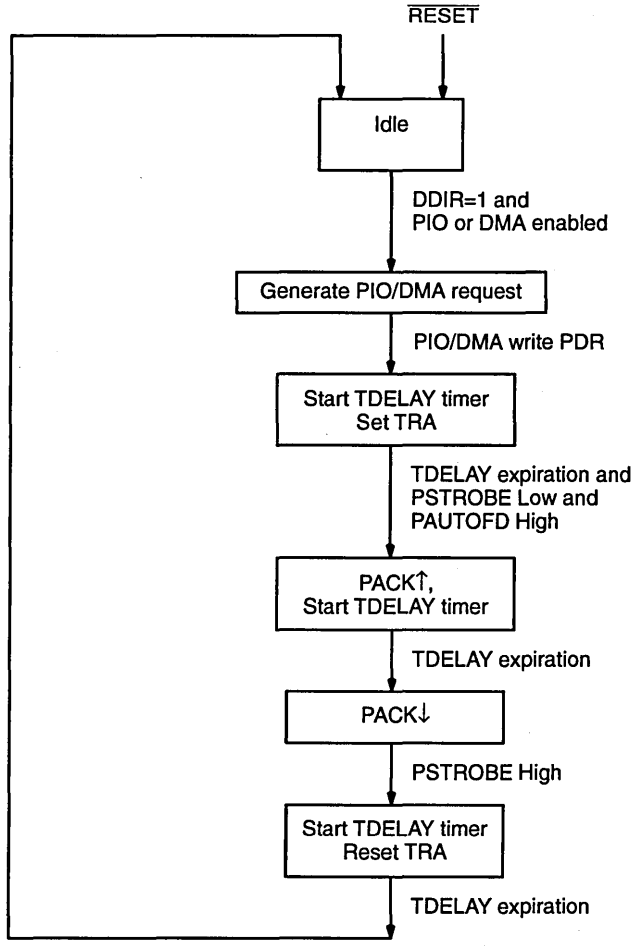


Figure 16-11 Transfer to the Host on the Parallel Port

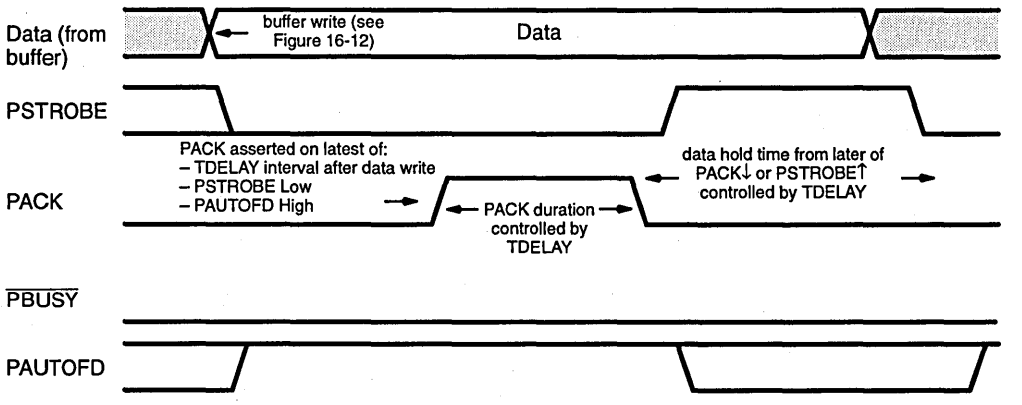
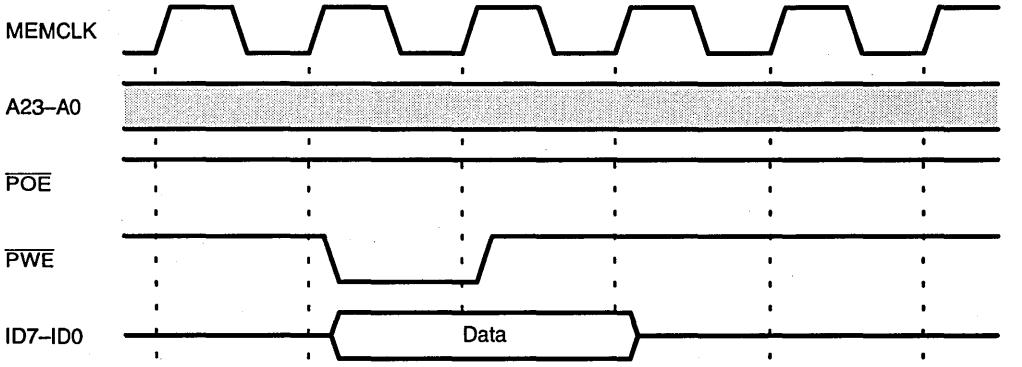


Figure 16-12 Parallel Port Buffer Write Cycle



CHAPTER

17 SERIAL PORTS

The Am29245 microcontroller has a single serial port, named Serial Port A, that permits full-duplex, bidirectional data transfer using the RS-232 protocol.

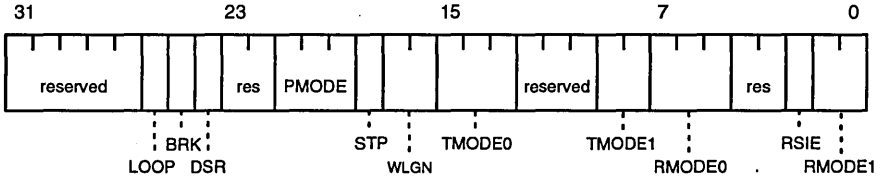
The Am29240 and Am29243 microcontrollers have a second serial port, named Serial Port B, in addition to Serial Port A. These ports are identical except Serial Port A has flow-control signals $\overline{\text{DSR}}$A and $\overline{\text{DTR}}$A. There are no signals dedicated to $\overline{\text{DTR}}$ and $\overline{\text{DSR}}$ functions for Serial Port B. These functions can be implemented for Serial Port B, if desired, by PIO signals.

17.1 PROGRAMMABLE REGISTERS, SERIAL PORT A

17.1.1 Serial Port A Control Register (SPCTA, Address 80000080)

The Serial Port A Control Register (Figure 17-1) controls both the transmit and receive sections of Serial Port A.

Figure 17-1 Serial Port A Control Register



Bits 31–27: Reserved

Bit 26: Loopback (LOOP)—Setting this bit places Serial Port A 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 Serial Port A.

Bit 25: Send Break (BRK)—Setting this bit causes Serial Port A 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 A Status Register), writing the Serial Port A 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 15–13 : Transmit Mode 0 (TMODE0)—This field enables data transmission and controls the operational mode of Serial Port A for the transmission of data, as follows:

TMODE0 Value	Effect on Serial Port
000	Disabled
001	Generate interrupt requests for service
010	Generate DMA Channel 0 requests
011	Generate DMA Channel 1 requests
100	Generate DMA Channel 2 requests
101	Generate DMA Channel 3 requests
110–111	Reserved

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 12–10: Reserved

Bits 9–8: Transmit Mode 1 (TMODE1)—This field enables data transmission and controls the operational mode of Serial Port A for the transmission of data, as follows:

TMODE1 Value	Effect on Serial Port
00	Disabled
01	Generate interrupt requests for service
10	Generate DMA Channel 0 requests
11	Generate DMA Channel 1 requests

The TMODE1 field is provided for compatibility with the Am29200 and Am29205 microcontrollers TMODE field.

Bits 7–5: Receive Mode 0 (RMODE0)—This field enables data reception and controls the operational mode of Serial Port A for the reception of data, as follows:

RMODE0 Value	Effect on Parallel Port
000	Disabled
001	Generate interrupt requests for service
010	Generate DMA Channel 0 requests
011	Generate DMA Channel 1 requests
100	Generate DMA Channel 2 requests
101	Generate DMA Channel 3 requests
110–111	Reserved

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.

Bits 4–3: Reserved

Bit 2: Receive Status Interrupt Enable (RSIE)—This bit enables Serial Port A to generate an interrupt because of an exception during reception. If this bit is 1 and Serial Port A receives a break or experiences a framing error, parity error, or overrun error, Serial Port A generates a Receive Status interrupt.

Bits 1–0: Receive Mode 1 (RMODE1)—This field enables data reception and controls the operational mode of the Serial Port for the reception of data, as follows:

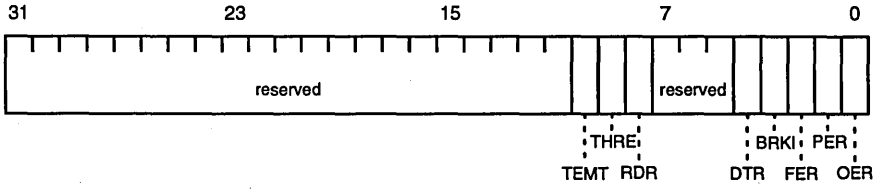
RMODE1 Value	Effect on Serial Port
00	Disabled
01	Generate interrupt requests for service
10	Generate DMA Channel 0 requests
11	Generate DMA Channel 1 requests

The RMODE1 field is provided for compatibility with the Am29200 and Am29205 microcontrollers' TMODE field.

17.1.2 Serial Port A Status Register (SPSTA, Address 80000084)

The Serial Port A Status Register (Figure 17-2) indicates the status of the transmit and receive sections of Serial Port A.

Figure 17-2 Serial Port A 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 either the TMODE1 or TMODE0 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 A 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 $\overline{\text{DTR}}$ pin. The DTR bit is 1 when the $\overline{\text{DTR}}$ pin is active; the DTR bit is 0 when the $\overline{\text{DTR}}$ 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.

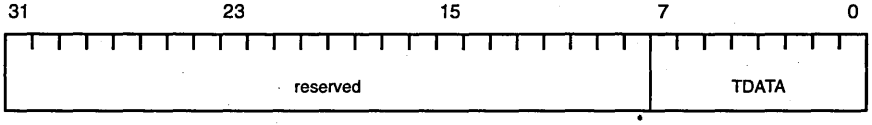
Bit 1: Parity Error (PER)—This bit is set to indicate that a parity error occurred during reception of data. If the RSIE bit is 1, the PER bit being set causes a Receive Status interrupt. The PER bit should be reset by the Receive Status interrupt handler.

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.

17.1.3 Serial Port A Transmit Holding Register (SPTHA, Address 80000088)

The processor writes this register (Figure 17-3) with data to be transmitted on Serial Port A. 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 17-3 Serial Port A Transmit Holding Register



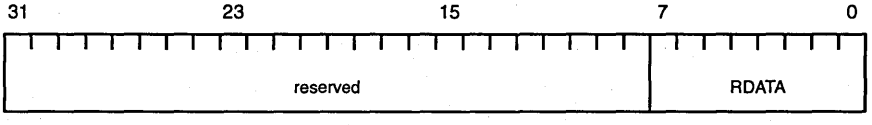
Bits 31–8: Reserved

Bits 7–0: Transmit Data (TDATA)—This field is written with data to be transmitted on Serial Port A. 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.

17.1.4 Serial Port A Receive Buffer Register (SPRBA, Address 8000008C)

This register (Figure 17-4) contains data received over Serial Port A. 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 17-4 Serial Port A Receive Buffer Register



Bits 31–8: Reserved

Bits 7–0: Receive Data (RDATA)—This field contains data received on Serial Port A. The RDR bit of the Serial Port A 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.

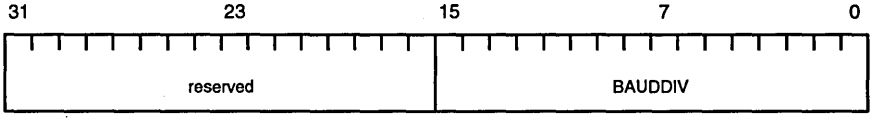
17.1.5 Baud Rate A Divisor Register (BAUDA, Address 80000090)

This register (Figure 17-5) specifies a clock divisor for the generation of a serial clock that controls Serial Port A. The serial clock rate is 16 times the rate of transmission or reception of data. The Baud Rate A 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:

Baud Rate = (Frequency of UCLK) ÷ (BAUDDIV+1) ÷ 32

The maximum baud rate is 1/32 of INCLK, and is achieved by tying UCLK to INCLK with BAUDDIV=0000, hexadecimal.

Figure 17-5 Baud Rate A Divisor Register



Bits 31–16: Reserved

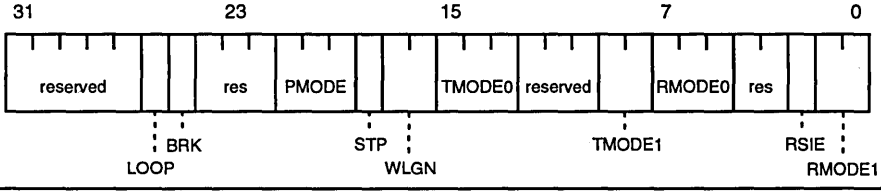
Bit 15–1: Baud Rate Divisor (BAUDDIV)—This field specifies the amount by which the UCLK input is divided to generate one phase of the serial clock. The serial clock operates at 16 times the rate of transmission or reception of data. The BAUDDIV value is zero-based. For example, a value of two specifies a divisor of three.

17.2 PROGRAMMABLE REGISTERS, SERIAL PORT B (Am29240 AND Am29243 MICROCONTROLLERS ONLY)

17.2.1 Serial Port B Control Register (SPCTB, Address 800000A0)

This register (Figure 17-6) is identical in function to the Serial Port A Control Register (Figure 17-1), except that there is no DSR bit (bit 24 of this register is reserved).

Figure 17-6 Serial Port B Control Register

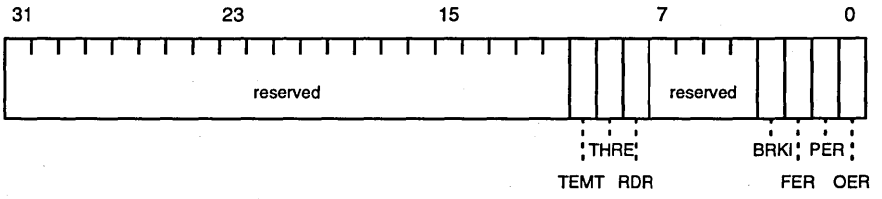


Bit 24 : Reserved (DSR bit in Serial Port A)

17.2.2 Serial Port B Status Register (SPSTB, Address 800000A4)

This register (Figure 17-7) is identical in function to the Serial Port A Status Register (Figure 17-2), except that there is no DTR bit (bit 4 of this register is reserved). For the Am29245 microcontroller, either the TMODE1 or TMODE0 field must be set to 00; and either the RMODE1 or RMODE0 field must be set to 00.

Figure 17-7 Serial Port B Status Register



Bit 4: Reserved (DTR bit in Serial Port A)

17.2.3 Serial Port B Transmit Holding Register (SPTHB, Address 800000A8)

This register is identical in definition and function to the Serial Port A Transmit Holding Register (Figure 17-3).

17.2.4 Serial Port B Receive Buffer Register (SPRBB, Address 800000AC)

This register is identical in definition and function to the Serial Port A Receive Buffer Register (Figure 17-4).

17.2.5 Baud Rate B Divisor Register (BAUDB, Address 800000B0)

This register is identical in definition and function to the Baud Rate A Divisor Register (Figure 17-5).

17.3 SERIAL PORT INITIALIZATION

During a processor reset, the TMODE1,TMODE0, RMODE1, and RMODE0 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. The serial port transmitter or receiver can be controlled by either TMODE field or either RMODE field, respectively, but the unused field must be zero.

CHAPTER

18 VIDEO INTERFACE

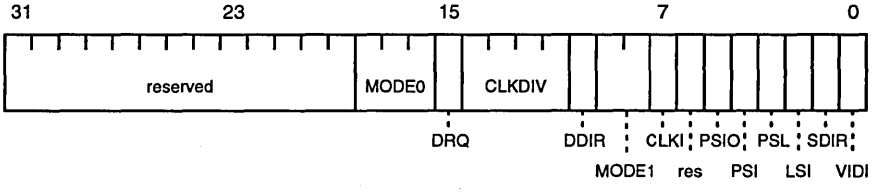
The video interface (serializer/deserializer) provides direct connection to a number of laser-beam marking engines. It may also be used to receive data from a raster input device such as a scanner or to serialize/deserialize a data stream. The video interface is supported in the Am29240 and Am29245 microcontrollers only.

18.1 PROGRAMMABLE REGISTERS

18.1.1 Video Control Register (VCT, Address 800000E0)

This register (see Figure 18-1) controls the operation of the video interface.

Figure 18-1 Video Control Register



Bits 31–19: Reserved

Bits 18–16 : Video Interface Mode 0 (MODE0)—This field enables the video interface and controls the operational mode of the video interface, as follows:

MODE0 Value	Effect on Video Interface
000	Disabled
001	Generate interrupt requests for service
010	Generate DMA Channel 0 requests
011	Generate DMA Channel 1 requests
100	Generate DMA Channel 2 requests
101	Generate DMA Channel 3 requests
110–111	Reserved

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 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 either the MODE1 or MODE0 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 field 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 correct 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 1 (MODE1)—This field enables the video interface and controls the operational mode of the video interface, as follows:

MODE1 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

The MODE1 field is provided for compatibility with the Am29200 and Am29205 microcontrollers' MODE field.

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.

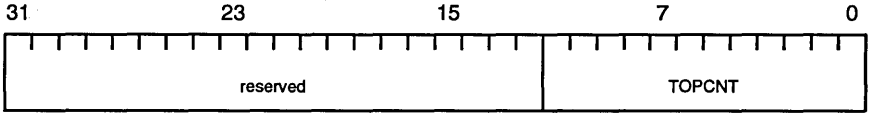
Bit 1: Shift Direction (SDIR)—When this bit is 0, the Video Data Shift Register is shifted right to transfer data, with video data being shifted out of the least significant bit of the register (corresponding to bit 0 of the Video Data Holding Register) or into the most significant bit (corresponding 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.

18.1.2 Top Margin Register (TOP, Address 800000E4)

This register (Figure 18-2) specifies the number of lines in the top margin of a page.

Figure 18-2 Top Margin Register



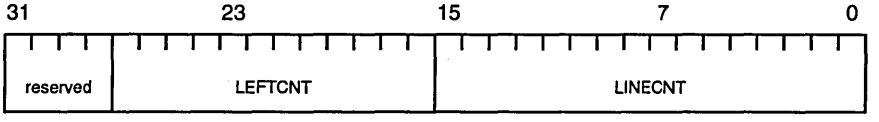
Bits 31–12: Reserved

Bits 11–0: Top Margin Count (TOPCNT)—This field specifies the number of lines in the top margin.

18.1.3 Side Margin Register (SIDE, Address 800000E8)

This register (Figure 18-3) 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 18-3 Side Margin Register



Bits 31–28: Reserved

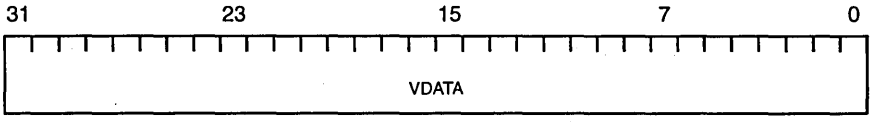
Bits 27–16: Left Margin Count (LEFTCNT)—This field specifies the number of data bit equivalents in the left margin of a page.

Bits 15-0: Line Count (LINECNT)—This field specifies the number of data bits in a raster line of video data.

18.1.4 Video Data Holding Register (VDT, Address 800000EC)

This register (Figure 18-4) 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 18-4 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.

18.1.5 Initialization

During a processor reset, both the MODE0 and MODE1 fields of the Video Control Register are 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. The video interface can be controlled by either the MODE0 or the MODE1 fields, but the unused field must be zero.

18.2 VIDEO INTERFACE OPERATION

The operation of the video interface is synchronous to the VCLK input, which clocks the video interface either 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 either the MODE1 or MODE0 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 up to double that 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.

18.2.1 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

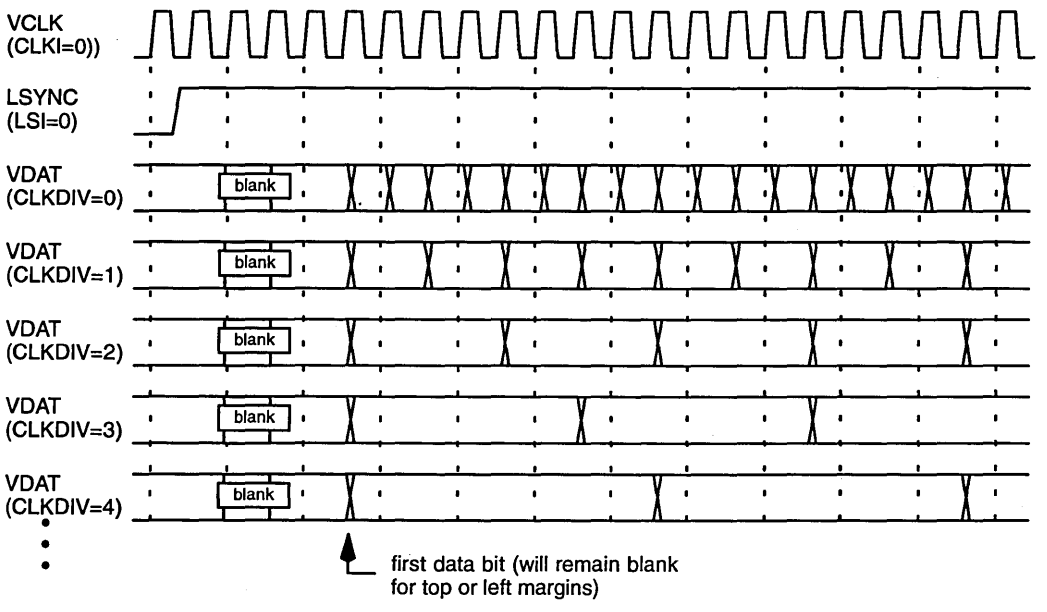
video interface is disabled or is not transferring data, the VDAT output is held at a level corresponding to a 0 data bit (Low if VIDI=0 or High if VIDI=1). Once the video interface has been configured, it is enabled via either the MODE1 or MODE0 field.

Enabling the video interface with DDIR=0 causes the interface to set the DRQ bit, thereby 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 18-5). 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.

Figure 18-5 VCLK, LSYNC, and VDAT Relationships (CLKI=0, LSI=0 for example only)



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, 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.

18.2.2 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

19 INTERRUPTS AND TRAPS

19.1 OVERVIEW

Interrupts and traps cause the Am29240 microcontroller series 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.

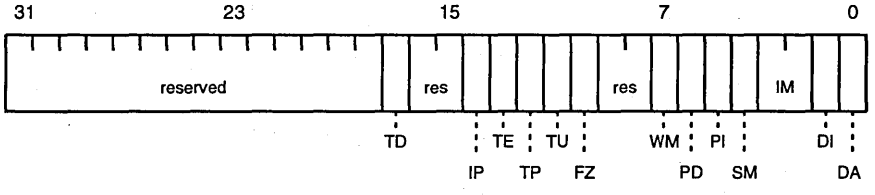
The distinction between interrupts and traps is largely one of causation and enabling. Interrupts allow external devices and the Timer Facility to control processor execution and are always asynchronous to program execution. Traps are intended to be used for certain exceptional events that occur during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

19.1.1 Current Processor Status Register (CPS, Register 2)

This protected special-purpose register (see Figure 19-1) controls the behavior of the processor and its ability to recognize exceptional events.

Figure 19-1 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 20.1 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: Physical Addressing/Data (PD)—The PD bit determines whether address translation is performed for load and store operations. Address translation is performed for a data access only when the PD bit is 0 and the Physical Address (PA) bit in the load or store instruction causing the access is also 0 (the PA bit can be 1 only for Supervisor-mode programs). Physical data addresses in the range 50000000–53FFFFFF are also translated to support mapped DRAM accesses that are compatible with the Am29200 microcontroller.

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

addresses in the range 50000000–53FFFFFF are also translated to support mapped DRAM accesses that are compatible with the Am29200 microcontroller.

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 19.1.2.

Bit 1: Disable Interrupts (DI)—The DI bit prevents the processor from being interrupted by external interrupt requests $\overline{\text{INTR3}}$–$\overline{\text{NTR0}}$ 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; interrupts are taken if otherwise enabled.

19.1.2 Interrupts

Interrupts are caused by signals applied to any of the external inputs $\overline{\text{INTR3}}$–$\overline{\text{INTR0}}$, by the Timer Facility (see Section 19.7), or by internal peripherals (see Section 19.8). 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	$\overline{\text{INTR0}}$ enabled
01	$\overline{\text{INTR1}}$–$\overline{\text{INTR0}}$ enabled
10	$\overline{\text{INTR2}}$–$\overline{\text{INTR0}}$ enabled
11	$\overline{\text{INTR3}}$–$\overline{\text{INTR0}}$ and internal Peripheral interrupts enabled

Note that the $\overline{\text{INTR0}}$ 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.

The $\overline{\text{INTR3}}$ interrupt is indicated in the Interrupt Control Register (Section 19.8.1).

19.1.3 Traps

Traps are caused by signals applied to one of the inputs $\overline{\text{TRAP1}}$–$\overline{\text{TRAP0}}$, 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.

19.1.4 External Interrupts and Traps

An external device causes an interrupt by asserting one of the $\overline{\text{INTR3}}$–$\overline{\text{INTR0}}$ inputs, and causes a trap by asserting one of the $\overline{\text{TRAP1}}$–$\overline{\text{TRAP0}}$ 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 $\overline{\text{INTR3}}$–$\overline{\text{INTR0}}$ 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 $\overline{\text{TRAP1}}$–$\overline{\text{TRAP0}}$ 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 $\overline{\text{INTR3}}$–$\overline{\text{INTR0}}$ inputs. The $\overline{\text{TRAP1}}$–$\overline{\text{TRAP0}}$ inputs cannot be disabled selectively. Both traps, however, can be disabled by the DA bit in the Current Processor Status Register.

The $\overline{\text{INTR3}}$–$\overline{\text{INTR0}}$ and $\overline{\text{TRAP1}}$–$\overline{\text{TRAP0}}$ 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 deasserted 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 $\overline{\text{INTR3}}$–$\overline{\text{INTR0}}$ and $\overline{\text{TRAP1}}$–$\overline{\text{TRAP0}}$ inputs is active. If the input causing an interrupt or trap is deasserted 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 deassertion of an $\overline{\text{INTR3}}$–$\overline{\text{INTR0}}$ or $\overline{\text{TRAP1}}$–$\overline{\text{TRAP0}}$ input to the time the corresponding interrupt or trap is no longer 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 deassertion 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.

19.1.5 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

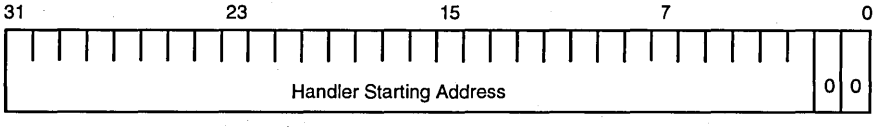
can leave the Wait mode only via a processor reset (see Section 2.9.2) or a $\overline{\text{WARN}}$ trap (see Section 19.4).

19.2 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 19-2. Each vector gives the beginning word-address of the associated interrupt or trap handling routine.

Figure 19-2 Vector Table Entry

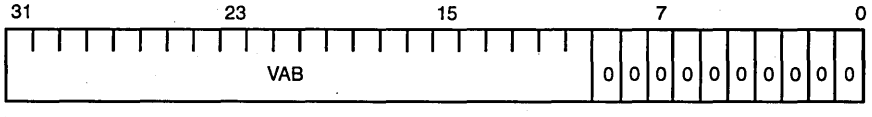


19.2.1 Vector Area Base Address Register (VAB, Register 0)

This protected special-purpose register (Figure 19-3) specifies the beginning address of the interrupt/trap Vector Area. The Vector Area is a table of 256 vectors that point to interrupt and trap handling routines.

When an interrupt or trap is taken, the vector number for the interrupt or trap (see Section 19.2.2) 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 19-3 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 1 K-Byte address-boundary in instruction/data memory.

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

19.2.2 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 19-1 (vector numbers are in decimal notation). Vector numbers 64 to 255 are used by trapping instructions; the definition of the routines associated with these numbers is system dependent.

19.3 INTERRUPT AND TRAP HANDLING

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

19.3.1 Old Processor Status Register (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.

19.3.2 Program Counter Stack

The Program Counter Unit, shown in Figure 19-4, forms and sequences instruction addresses for the Instruction Fetch Unit. It contains the Program Counter (PC), the Program-Counter Multiplexer (PC MUX), the Return Address Latch, and the Program-Counter Buffer (PC Buffer).

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

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

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

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

Table 19-1 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	Reserved	
4	Parity Error	Invalid DRAM parity on read[5]
5	Protection Violation	Invalid User-mode operation[2]
6–7	Reserved	
8	User Instruction TLB Miss	No TLB entry for translation or mapping
9	User Data TLB Miss	No TLB entry for translation or mapping
10	Supervisor Instruction TLB Miss	No TLB entry for translation or mapping
11	Supervisor Data TLB Miss	No TLB entry for translation or mapping
12	Instruction MMU Protection Violation	TLB UE=0
13	Data MMU Protection Violation	TLB UR=0, UW/SW=0 on write
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[4]
31	MULTMU	MULTMU instruction[4]
32	MULTIPLY	MULTIPLY instruction[4]
33	DIVIDE	DIVIDE instruction
34	MULTIPLU	MULTIPLU instruction[4]
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

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).*
4. *Applies to the Am29245 microcontroller only.*
5. *Applies to the Am29243 microcontroller only.*

Table 19-1 Vector Number Assignments (continued)

Number	Type of Trap or Interrupt	Cause
53	DMUL	DMUL instruction
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 (order # 16693).

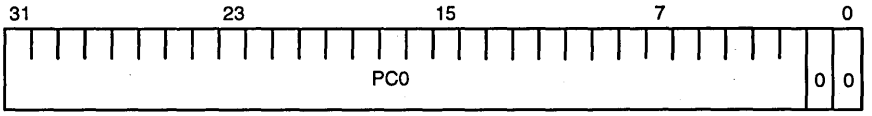
Upon the execution of an interrupt return, the target instruction stream is restarted using the instruction addresses in PC0 and PC1. Two registers are required here because the processor implements delayed branches. An interrupt or trap may be taken when the processor is executing the delay instruction 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.

19.3.2.1 Program Counter 0 Register (PC0, Register 10)

This protected special-purpose register (Figure 19-5) is used on an interrupt return to restart the instruction in the decode stage when the original interrupt or trap was taken.

Figure 19-5 Program Counter 0 Register

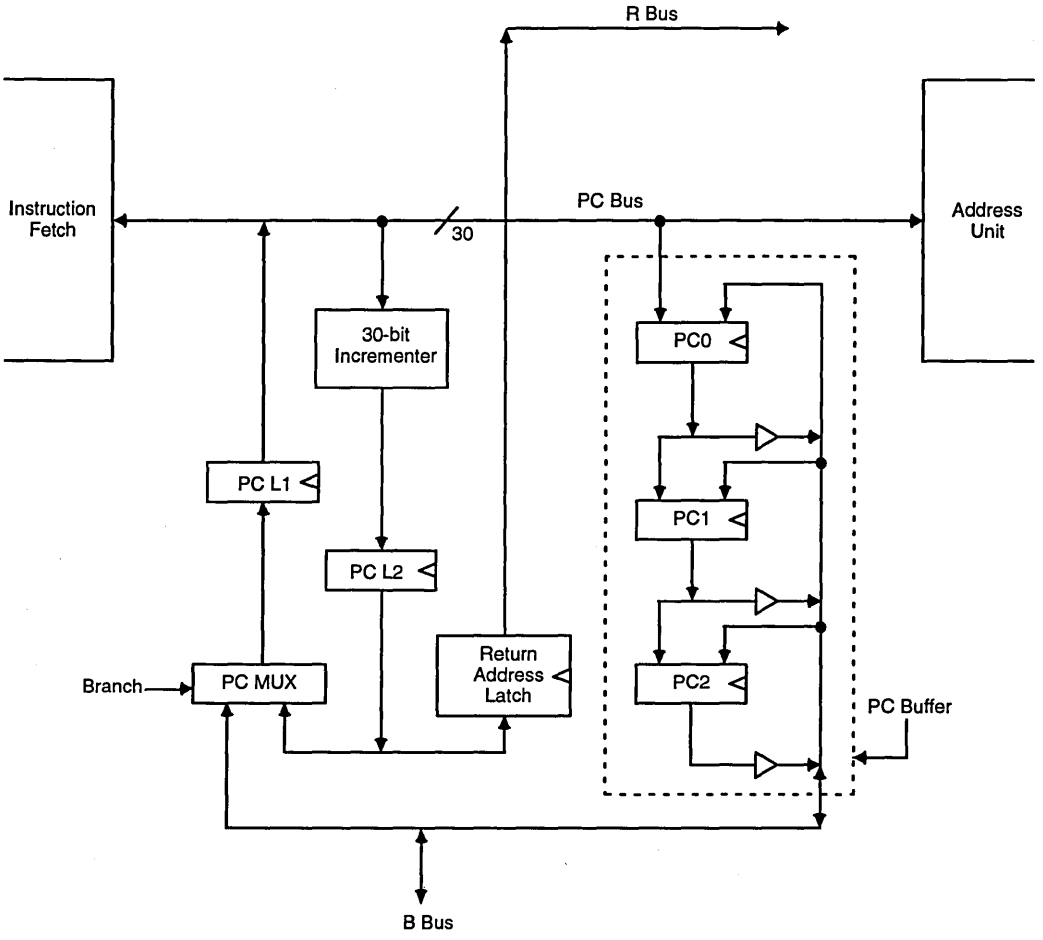


Bits 31–2: Program Counter 0 (PC0)—This field captures the word-address of an instruction as it enters the decode stage of the processor pipeline, unless the Freeze (FZ) bit of the Current Processor Status Register is 1. If the FZ bit is 1, PC0 holds its value.

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.

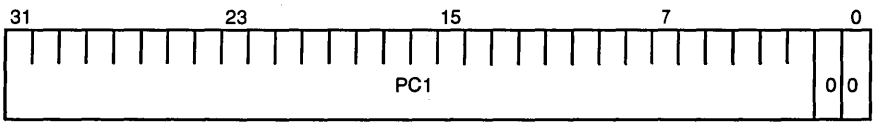
Figure 19-4 Program Counter Unit



19.3.2.2 Program Counter 1 Register (PC1, Register 11)

This protected special-purpose register (Figure 19-6) is used on an interrupt return to restart the instruction in the execute stage when the original interrupt or trap was taken.

Figure 19-6 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.

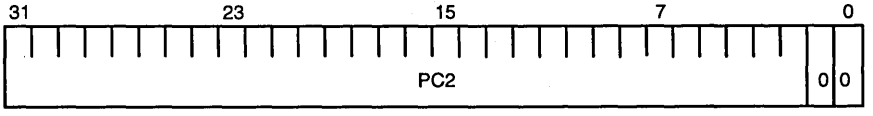
When an interrupt or trap is taken, the PC1 field contains the word-address of the instruction in the execute stage; the interrupt or trap has prevented this instruction from completing execution. The processor uses the PC1 field to restart this instruction on an interrupt return.

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

19.3.2.3 Program Counter 2 Register (PC2, Register 12)

This protected special-purpose register (Figure 19-7) reports the address of certain instructions causing traps.

Figure 19-7 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.

19.3.3 Taking an Interrupt or Trap

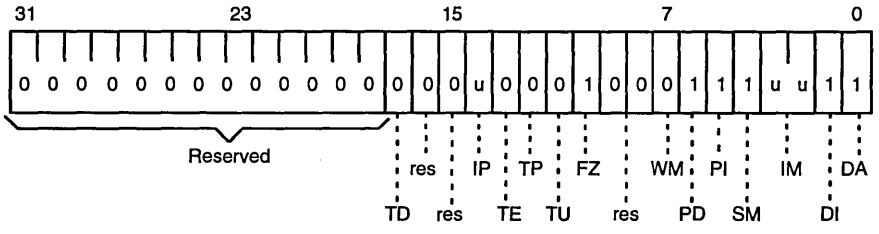
The following operations are performed in sequence by the processor when an interrupt or trap is taken:

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 19-8 (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 responsibility of the interrupt- or trap-handling routine. For proper operation, registers must be saved before any further interrupts or traps may be taken. The FZ bit must be reset at least two instructions before interrupts or traps are re-enabled, to allow program state to be reflected properly in processor registers if an interrupt or trap is taken.

Figure 19-8 Current Processor Status After an Interrupt or Trap



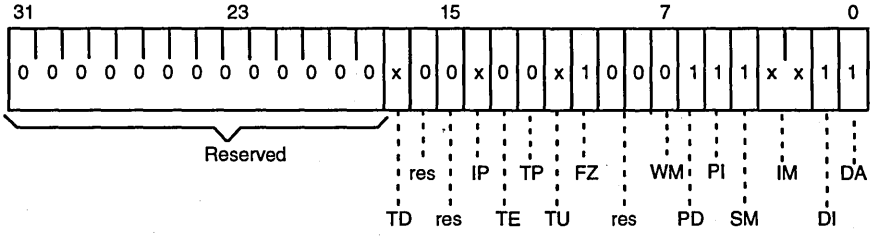
19.3.4 Returning from an Interrupt or Trap

Two instructions are used to resume the execution of an interrupted program: Interrupt Return (IRET), and Interrupt Return and Invalidate (IRETINV). These instructions are identical except in one respect: the IRETINV instruction resets all valid bits in the instruction cache, the data cache, or both caches, 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 19-9 (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 re-start 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.

Figure 19-9 Current Processor Status Before Interrupt Return



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

1. Any in-progress load or store operation is completed. If a load-multiple or store-multiple sequence is in progress, the interrupt return is not executed until the sequence completes.
2. Interrupts and traps are disabled, regardless of the settings of the DA, DI, and IM fields of the Current Processor Status, for steps 3 through 10.
3. The contents of the Old Processor Status Register are copied into the Current Processor Status Register. This normally resets the FZ bit, allowing the Program Counter 0, 1, 2, Channel Address, Data, Control, and ALU Status registers to update normally. Since certain bits of the Current Processor Status Register always are updated by the processor, this copy operation may be irrelevant for certain bits (e.g., the Interrupt Pending bit).
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.

19.3.5 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 lightweight interrupt handlers that do not have all of the overhead normally associated with interrupt handlers.

The processor provides an additional benefit to lightweight interrupts if the Program Counter 0 and Program Counter 1 Registers are not modified by the interrupt or trap handler. If Program Counters 0 and 1 contain the addresses of sequential instructions when an interrupt or trap is taken, and if they are not modified before an interrupt return is executed, step 7 of the interrupt return sequence in Section 19.3.4 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:

- 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.
- Loads and stores that 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.
- Extended arithmetic. The Carry bit of the ALU Status Register is not updated while the FZ bit is 1.
- 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 19.6.6.

19.3.6 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.

19.4 $\overline{\text{WARN}}$ TRAP

The processor recognizes a special trap, caused by the activation of the $\overline{\text{WARN}}$ input, that 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:

- 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.
- 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.

19.4.1 $\overline{\text{WARN}}$ 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 deasserted 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.

19.5 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 19-2. In this table, interrupts and traps are listed in order of decreasing priority.

Table 19-2 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
2	User-Mode Data TLB Miss	Inst	Next	All
	Supervisor-Mode Data TLB 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	Parity Error	Async	Next	All
5	$\overline{\text{TRAP0}}$	Async	Next	Multiple
6	$\overline{\text{TRAP1}}$	Async	Next	Multiple
7	$\overline{\text{INTR0}}$	Async	Next	Multiple
8	$\overline{\text{INTR1}}$	Async	Next	Multiple
9	$\overline{\text{INTR2}}$	Async	Next	Multiple
10	$\overline{\text{INTR3}}$	Async	Next	Multiple
	Internal peripheral interrupts	Async	Next	Multiple
11	Timer	Async	Next	Multiple
12	Trace	Async	Next	Multiple
13	User-mode Inst TLB Miss	Inst	Curr	N/A
	Supervisor-mode Inst TLB Miss	Inst	Curr	N/A
14 (Lowest)	Illegal Opcode	Inst	Curr	N/A
	Protection Violation	Inst	Curr	N/A

Note: *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.*

This section discusses the first three columns of Table 19-2. The last two columns are discussed in Section 19.6.

In Table 19-2, 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 19-2 by the labels *Inst* and *Async.* These labels have the following meaning:

- Inst—Generated by the execution or attempted execution of an instruction.
- 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 19-2 with priority 13 or 14 are regenerated by the re-execution of the causing instruction.
2. Most of the interrupts and traps of priority 4 through 12 must be held by external hardware until they are taken. The exceptions to this are listed in item 3.
3. The exceptions to item 2 are the Parity Error trap, the Timer interrupt, and the Trace trap. These are caused by bits in various registers in the processor and are held by these registers until taken or cleared. The relevant bits are the Parity Error (PER) bit of the Channel Control Register for Parity Error traps, the Interrupt (IN) bit of the Timer Reload Register for Timer interrupts, and the Trace Pending (TP) bit of the Current Processor Status Register for Trace traps.
4. All traps of priority 2 and 3 in Table 19-2, 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.

19.6 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 19-2 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.

19.6.1 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.

19.6.2 Restarting Faulting Accesses

DRAM mapping is performed by the TLB 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.

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

The processor 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. These registers also provide information about accesses that encounter protection violations and parity errors. 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.

Upon an interrupt return (IRET or IRETINV), the processor restarts an access using the Channel Address, Channel Data, and Channel Control registers. 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.

Note that the exception handler for the Parity Error trap must clear the Parity Error (PER) bit in the Channel Control Register. For proper sequencing of traps, the PER bit being 1 causes a Parity Error trap. Failure to clear the PER bit results in the processor taking the Parity Error trap again, once the exception handler returns, causing an infinite series of traps.

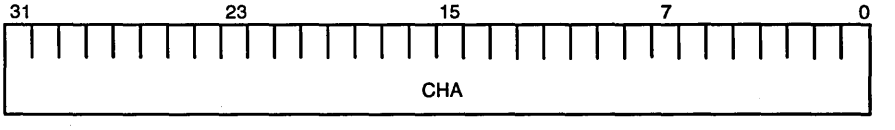
The mechanism used to restart trapping accesses has the additional benefit of allowing a fast interrupt-response time when the processor is performing a load-multiple or store-multiple operation. An interrupted load-multiple or store-multiple is restarted as if it had faulted. In this case, the operation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

19.6.2.1 Channel Address Register (CHA, Register 4)

This protected special-purpose register (Figure 19-10) 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.

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 19-10 Channel Address Register



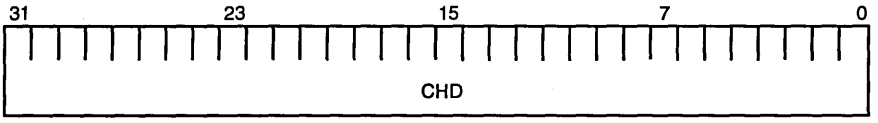
Bits 31–0: Channel Address (CHA)—This field contains the address of the current access (if the FZ bit of the Current Processor Status Register is 0).

19.6.2.2 Channel Data Register (CHD, Register 5)

This protected special-purpose register (Figure 19-11) 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.

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 19-11 Channel Data Register



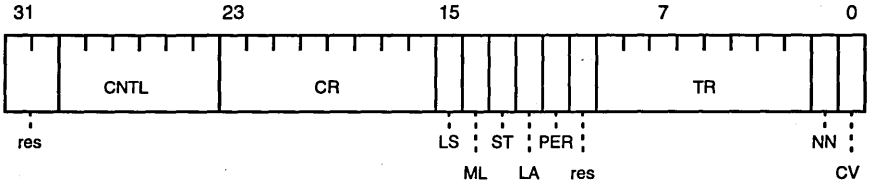
Bits 31–0: Channel Data (CHD)—This field contains the data (if any) associated with the current access (if the FZ bit of the Current Processor Status Register is 0). If the current access is not a store, the value of this field is irrelevant.

19.6.2.3 Channel Control Register (CHC, Register 6)

This protected special-purpose register (Figure 19-12) 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.

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 19-12 Channel Control Register



Bits 31–30: Reserved

Bits 29–24:—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: Lock Active (LA)—The LA bit is 1 if the current access is for a load and lock or store and lock instruction; otherwise it is 0.

Bit 11: Parity Error (PER)—The PER bit indicates that data received during a DRAM access did not have valid parity. This bit is set when parity checking is enabled for DRAM accesses and the processor detects invalid parity on one of the bytes received during a load. The processor checks only those bytes actually loaded. This bit causes a Parity Error trap when it is set.

Bit 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.

19.6.3 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 Am29245 microcontroller 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) is unchanged if an Out-of-Range trap is taken.

19.6.4 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 19.3.3, 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.

19.6.5 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.

19.6.6 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.

19.7 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.

19.7.1 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). 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.

19.7.2 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.

19.7.3 Handling Timer Interrupts

The following is a suggested list of actions 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.

19.7.4 Timer Facility Uses

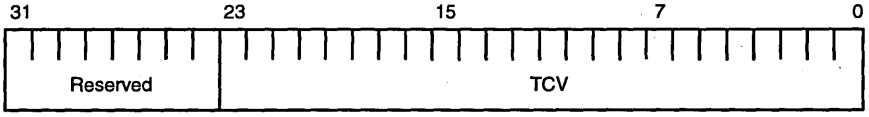
Since the Timer Facility has a resolution of a single processor cycle, it may be used to perform precise timing of system events. For example, it may be used to determine an exact measurement of the number of cycles between two events in the system or to perform precise time-critical control functions. 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.

19.7.5 Timer Counter Register (TMC, Register 8)

This protected special-purpose register (Figure 19-13) contains the counter for the Timer Facility.

Figure 19-13 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.

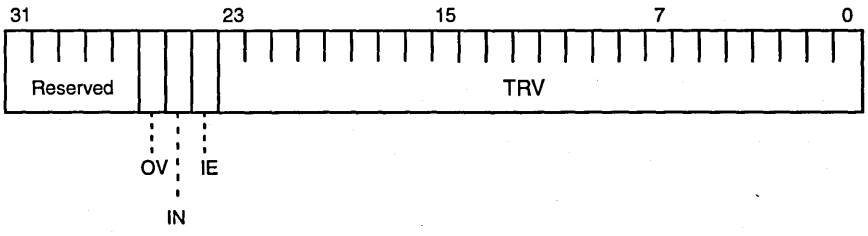
19.7.6 Timer Reload Register (TMR, Register 9)

This protected special-purpose register (Figure 19-14) maintains synchronization of the Timer Counter Register, enables Timer interrupts, and maintains Timer Facility status information.

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

Figure 19-14 Timer Reload Register



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. The Timer interrupt can also be disabled by the TD bit of the Current Processor Status Register, regardless of the value of IE and/or DA.

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.

19.8 INTERNAL INTERRUPT CONTROLLER

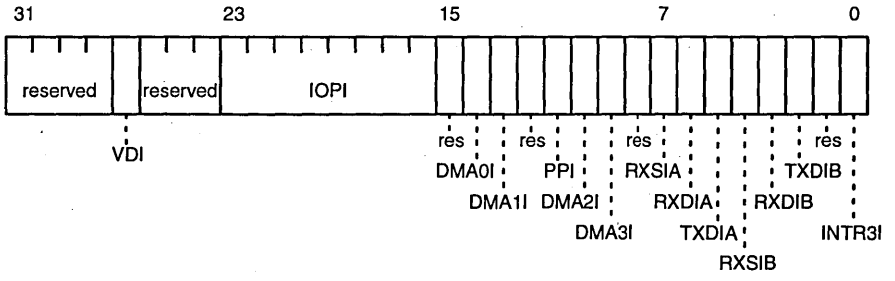
The various peripherals and controllers on the Am29240 microcontroller series can cause interrupts having the same effect on the processor as asserting the processor's $\overline{\text{INTR3}}$ input. The interrupt controller provides a central location for generating and masking interrupts, indicating which interrupts are active, and permitting software to reset the interrupts independent of servicing the interrupting peripheral.

19.8.1 Interrupt Control Register (ICT, Address 80000028)

Bits of the Interrupt Control Register (Figure 19-15) 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 $\overline{\text{INTR3}}$ input (there is no external indication of this on $\overline{\text{INTR3}}$). 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 19-15 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.

Bit 10 : DMA Channel 2 Interrupt (DMA2I)—A 1 in this bit indicates that DMA Channel 2 has generated an interrupt request.

Bit 9 : DMA Channel 3 Interrupt (DMA3I)—A 1 in this bit indicates that DMA Channel 3 has generated an interrupt request.

Bit 8: Reserved

Bit 7: Serial Port A Receive Status Interrupt (RXSIA)—A 1 in this bit indicates that Serial Port A has generated an interrupt request because of the status of its receive logic.

Bit 6: Serial Port A Receive Data Interrupt (RXDIA)—A 1 in this bit indicates that Serial Port A has generated an interrupt request because its receive data is ready.

Bit 5: Serial Port A Transmit Data Interrupt (TXDIA)—A 1 in this bit indicates that Serial Port A has generated an interrupt request because its Transmit Holding Register is empty.

Bit 4: Serial Port B Receive Status Interrupt (RXSIB)—A 1 in this bit indicates that Serial Port B has generated an interrupt request because of the status of its receive logic.

Bit 3: Serial Port B Receive Data Interrupt (RXDIB)—A 1 in this bit indicates that Serial Port B has generated an interrupt request because its receive data is ready.

Bit 2: Serial Port B Transmit Data Interrupt (TXDIB)—A 1 in this bit indicates that Serial Port B has generated an interrupt request because its Transmit Holding Register is empty.

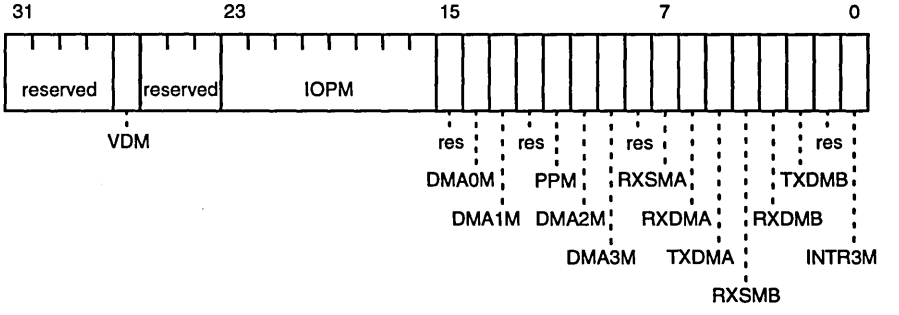
Bit 1: Reserved

Bit 0: $\overline{\text{INTR3}}$ Interrupt (INTR3I)—A 1 in this bit indicates that the $\overline{\text{INTR3}}$ input is active.

19.8.2 Interrupt Mask Register (IMASK, Address 8000002C)

Bits in this register (Figure 19-16) disable the corresponding interrupt sources from interrupting the processor.

Figure 19-16 Interrupt Mask Register



Bits 31–28: Reserved

Bit 27: Video Mask (VDM)—A 1 in this bit masks video interface interrupts. This bit is reserved on the Am29243 microcontroller.

Bits 26–24: Reserved

Bits 23–16: I/O Port Mask (IOPM)—A 1 in this field masks the respective PIO interrupt. A 1 in the most significant bit of the IOPM field masks the PIO15 interrupt, the next bit masks the PIO14 interrupt, and so on.

Bit 15: Reserved

Bit 14: DMA Channel 0 Mask (DMA0M)—A 1 in this bit masks DMA Channel 0 interrupts.

Bit 13: DMA Channel 1 Mask (DMA1M)—A 1 in this bit masks DMA Channel 1 interrupts

Bit 12: Reserved

Bit 11: Parallel Port Mask (PPM)—A 1 in this bit masks parallel port interrupts.

Bit 10: DMA Channel 2 Mask (DMA2M)—A 1 in this bit masks DMA Channel 2 interrupts. This bit is reserved on the Am29245 microcontroller.

Bit 9: DMA Channel 3 Mask (DMA3M)—A 1 in this bit masks DMA Channel 3 interrupts. This bit is reserved on the Am29245 microcontroller.

Bit 8: Reserved

Bit 7: Serial Port A Receive Status Mask (RXSMA)—A 1 in this bit masks Serial Port A receive status interrupts.

Bit 6: Serial Port A Receive Data Mask (RXDMA)—A 1 in this bit masks Serial Port A receive data interrupts.

Bit 5: Serial Port A Transmit Data Mask (TXDMA)—A 1 in this bit masks Serial Port A transmit data interrupts.

Bit 4: Serial Port B Receive Status Mask (RXSMB)—A 1 in this bit masks Serial Port B receive status interrupts. This bit is reserved on the Am29245 microcontroller.

Bit 3: Serial Port B Receive Data Mask (RXDMB)—A 1 in this bit masks Serial Port B receive data interrupts. This bit is reserved on the Am29245 microcontroller.

Bit 2: Serial Port B Transmit Data Mask (TXDMB)—A 1 in this bit masks Serial Port B transmit data interrupts. This bit is reserved on the Am29245 microcontroller.

Bit 1: Reserved

Bit 0: $\overline{\text{INTR3}}$ Mask (INTR3M)—A 1 in this bit masks $\overline{\text{INTR3}}$ interrupts.

19.8.3 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.

19.8.4 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

20 DEBUGGING AND TESTING

This chapter details the features of the Am29240 microcontroller series that support debugging and testing. The chapter first describes the Trace Facility and instruction breakpoints that aid in software debugging. Next, the support for hardware-development systems, including the Test/Development Interface and the Traceable Cache™ technology feature, is described. Finally, the Test Access Port and the Boundary-Scan Architecture are discussed.

20.1 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.

20.2 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 20.6.1.

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 instruction is used.

20.3 PROCESSOR STATUS OUTPUTS

The STAT2–STAT0 outputs indicate certain information about processor modes along with information about processor operation. STAT2–STAT0 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 STAT2–STAT0 is as follows:

STAT2	STAT1	STAT0	Condition
0	0	0	Halt or Step Modes
0	0	1	Interrupt/Trap Vector Fetch (vector valid)
0	1	0	Load Test Instruction Mode, Halt/Freeze
0	1	1	Non-sequential instruction fetch (internal cache hit, or external access and instruction valid)
1	0	0	External data access (data valid)
1	0	1	External sequential instruction access (instruction valid)
1	1	0	Internal peripheral access (data valid)
1	1	1	Idle or data/instruction not valid

The status conditions are prioritized in the order listed, with STAT2–STAT0=000 having highest priority. The STAT2–STAT0 outputs are changed at the end of every processor cycle to indicate the processor status in the previous cycle. Thus, if the processor operates at twice the system frequency, the STAT2–STAT0 outputs change on both the rising and falling edge of MEMCLK. If the processor operates at twice the system frequency, the status indication related to an external access (such as an external instruction access) appears in the first half-cycle of MEMCLK (MEMCLK High) just after the completion of the external access; in the second half-cycle of this MEMCLK cycle (MEMCLK Low), the processor's internal condition is indicated. If the processor operates at the system frequency, the status indication related to an external access appears for the entire MEMCLK cycle following the completion of the access.

For the status conditions related to external accesses (STAT2–STAT0 = 100, 101, or 110), the R/$\overline{W}$ output indicates the direction of the access. If an access is extended by $\overline{WAIT}$, the appropriate status is shown for every additional cycle until the access completes. The address for an access that does not hit in the cache always appears

on A23–A0, 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 ID31–ID0, except on a read of an internal peripheral.

Other status information is available at the outputs of a processor that is configured as a tracing processor (see Section 20.7).

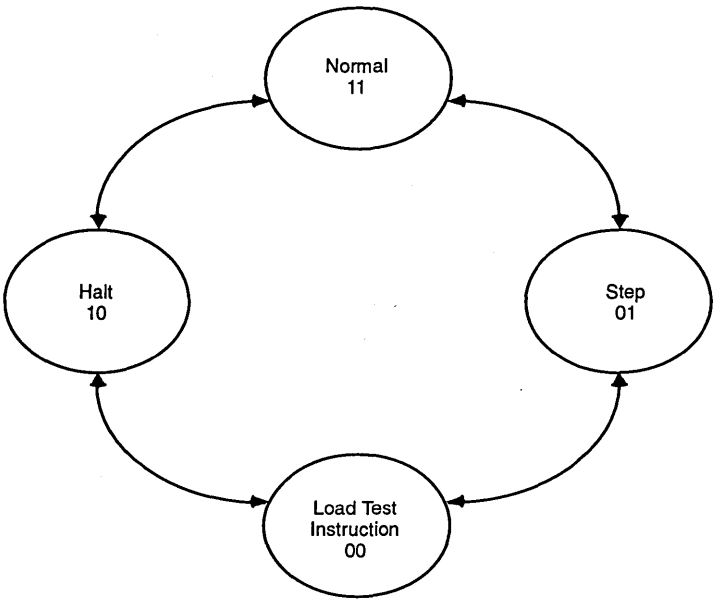
20.4 CPU CONTROL INPUTS

Certain processor operation modes are under control of the CNTL1–CNTL0 inputs. These inputs affect the processor mode as follows:

CNTL1–CNTL0	Mode
00	Load Test Instruction
01	Step
10	Halt
11	Normal

These inputs are asynchronous to the processor clock. In addition, changes on the CNTL1–CNTL0 inputs are restricted so that only CNTL1 or CNTL0, but not both, may change in any given processor cycle. The allowed transitions are shown in Figure 20-1. The restriction on transitions of CNTL1–CNTL0 allows these inputs to be driven directly by an external hardware-development system or tester without any intervening logic. Proper operation is insured by making only single-input changes on CNTL1–CNTL0 and by restricting the interval between all changes to be greater than a processor cycle. If these restriction are violated, processor operation is unpredictable and a processor reset is required to resume predictable operation.

Figure 20-1 Valid Transitions for the CNTL1–CNTL0 Pins



Because of the restrictions just described, it is not possible to transition directly between all possible modes controlled by the CNTL1–CNTL0 pins. For example, the processor cannot go from the Load Test Instruction mode to Normal operation without first entering the Halt or Step modes.

20.5 TEST ACCESS PORT

The Am29240 microcontroller series 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 and MEMCLK pins have capture-only cells. 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 considerations specific to the Am29240 microcontroller series.

20.5.1 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 boundary-scan cells.

Figure 20-2 shows the design of an input boundary-scan cell, and Figure 20-3 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.

Figure 20-2 Input Boundary-Scan Cell

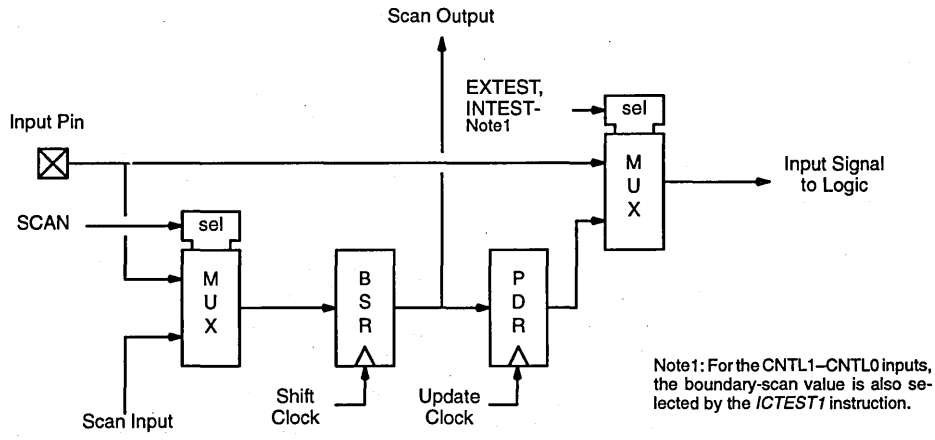
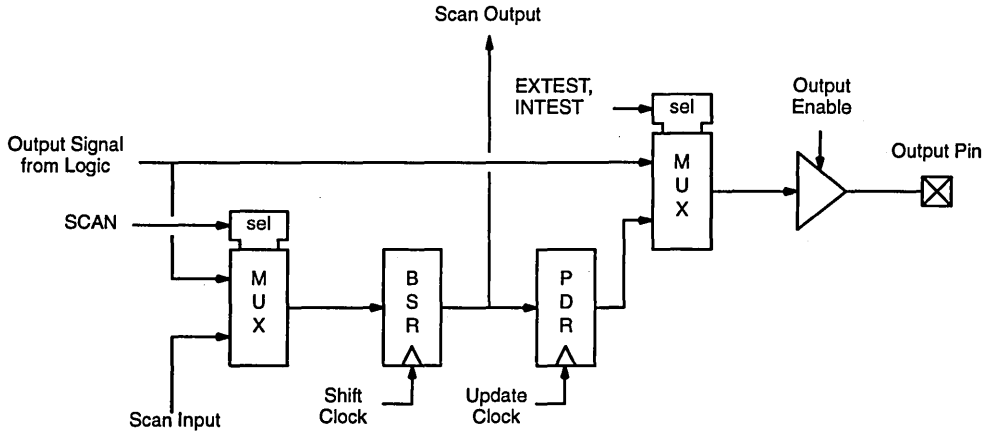


Figure 20-3 Output Boundary-Scan Cell



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 A23–A0 outputs can be disabled).

The boundary-scan cells for the CNTL1–CNTL0 field and STAT2–STAT0 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 20.5.2). If the ICTEST1 instruction is active, no other boundary-scan cell is scanned. However, the contents of the other scan cells are undefined after this operation.

The INCLK input does not have a standard boundary-scan cell because this cell can only capture the value on the INCLK pin. 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 also does not have a standard boundary-scan cell, because this cell can only capture the value on the MEMCLK pin.

20.5.2 Instruction Register and Implemented Instructions

The Instruction Register (IREG) of the Test Access Port is a 5-bit register. The least significant bit (IREG0) is the bit nearest the TDO output. Instructions are encoded as follows:

IREG4–IREG0	Instruction
00000	EXTEST
00001	HIZ
00010	ICTEST2
00011	IDCODE
00100	INTEST
00101	SAMPLE
00110	ICTEST1
00111	Reserved (acts like BYPASS)
01000	TRACECACHE
01001	TRACEOFF
01010–01110	Reserved (acts like BYPASS)
01111	RUNBIST
10000–11110	AMD private instructions (factory test)
11111	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 Am29240 microcontroller series.

20.5.2.1 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 Boundary-Scan Register (BSR) for scanning. During execution:

1. Processor outputs are driven from the Parallel Data Register (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.

20.5.2.2 HIZ

The HIZ instruction acts exactly like the EXTEST instruction, except that all outputs are placed into the high-impedance state.

20.5.2.3 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 ID31–ID0 and STAT2–STAT0 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 ID31–ID0 and STAT2–STAT0 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 ID31–ID0 are driven from the PDR. This allows a hardware-development system to provide data to the processor, independent of system controls.

20.5.2.4 IDCODE

The IDCODE instruction connects the processor identification register from TDI to TDO. This value can be shifted out in the SHIFTDR controller state. This instruction does not affect the operation of the processor or the boundary-scan logic.

20.5.2.5 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.

20.5.2.6 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.

20.5.2.7 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 CNTL1–CNTL0 pins and sample the STAT2–STAT0 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 STAT2–STAT0.
3. Processor input signals are sampled into the BSR. This is default behavior.
4. The processor CNTL1–CNTL0 pins are driven by the PDR. Processor internal inputs are driven from the processor inputs.

20.5.2.8 TRACECACHE

This instruction enables the Traceable Cache technology feature described in Section 20.7.

20.5.2.9 TRACEOFF

This instruction disables the Traceable Cache technology feature.

20.5.2.10 RUNBIST

The RUNBIST instruction is used to initiate the internal self test, which includes testing the on-chip caches. This instruction is started when the TAP controller is placed in the Run-Test/Idle state. After 38,640 processor cycles, the processor's internal self test is complete and the PASS/FAIL status is loaded into the CBIST test data register.

The CBIST register is a 4-bit register that is connected between TDI and TDO during the RUNBIST instruction. After the self test is complete, the test result status can be shifted out in the SHIFTDR state.

20.5.2.11 Private Instructions

There are several instructions used to apply test patterns and observe results. These are intended for manufacturing tests and should not be invoked by users.

20.5.2.12 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.

20.5.3 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 Am29240 microcontroller series, 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. For compatibility, pins on the Am29245 microcontroller that are reserved for features on the Am29245 and Am29243 microcontrollers are assigned boundary-scan cells, even though the associated pins are reserved.

20.5.3.1 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 IREG2–IREG0 = 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 20.5.2.

Table 20-1 Instruction Scan Path

Bit	Cell Name
1	IREG4
2	IREG3
3	IREG2
4	IREG1
5	IREG0

20.5.3.2 Bypass Path

This is a 1-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.

20.5.3.3 Main Data Path

This is a 208-cell path used to access the processor pins. This path is divided into five sets of cells. Where applicable, each set has a cell that enables the outputs of the set to be driven on the processor's pins. These drive-enable cells are not connected to a processor pin. For convenience, the drive-enable cells are shown in Table 20-2 in boldface. Some of these enable 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.

Table 20-2 Main Data Scan Path

Bit	Cell Name	Comments
1	INCLK	The INCLK scan cell is a capture-only cell: it captures the value on the INCLK pin
2	PCLK	
3	MEMCLK	The MEMCLK scan cell is a capture-only cell: it captures the value on MEMCLK
4	$\overline{\text{TRIST}}$	
5	CNTL0	
6	CNTL1	
7	$\overline{\text{RESET}}$	
8	LSYNC	
9	VCLK	
10	$\overline{\text{WARN}}$	
11	$\overline{\text{INTR3}}$	
12	$\overline{\text{INTR2}}$	
13	$\overline{\text{INTR1}}$	
14	$\overline{\text{INTR0}}$	
15	$\overline{\text{TRAP1}}$	
16	$\overline{\text{TRAP0}}$	
17	TDMAI	TDMA input
18	TDMAO	TDMA output
19	DREQA	
20	DREQB	
21	$\overline{\text{GREQ}}$	

Table 20-2 Main Data Scan Path (continued)

Bit	Cell Name	Comments
22	**TOPDRV**	Enables the drivers for PSYNC through PWE
23	PSYNCI	PSYNC input
24	PSYNCO	PSYNC output
25	VDATI	VDAT input
26	VDATO	VDAT output
27	STAT0	
28	STAT1	
29	STAT2	
30	PIOI0	PIO0 input
31	PIOO0	PIO0 output
32	PIOI1	PIO1 input
33	PIOO1	PIO1 output
34	PIOI2	PIO2 input
35	PIOO2	PIO2 output
36	PIOI3	PIO3 input
37	PIOO3	PIO3 output
38	PIOI4	PIO4 input
39	PIOO4	PIO4 output
40	PIOI5	PIO5 input
41	PIOO5	PIO5 output
42	PIOI6	PIO6 input
43	PIOO6	PIO6 output
44	PIOI7	PIO7 input
45	PIOO7	PIO7 output
46	PIOI8	PIO8 input
47	PIOO8	PIO8 output
48	PIOI9	PIO9 input
49	PIOO9	PIO9 output
50	PIOI10	PIO10 input
51	PIOO10	PIO10 output
52	PIOI11	PIO11 input
53	PIOO11	PIO11 output
54	PIOI12	PIO12 input
55	PIOO12	PIO12 output
56	DREQC	
57	DREQD	
58	PIOI13	PIO13 input
59	PIOO13	PIO13 output
60	PIOI14	PIO14 input
61	PIOO14	PIO14 output
62	PIOI15	PIO15 input
63	PIOO15	PIO15 output
64	$\overline{\text{PBUSY}}$	
65	PACK	
66	$\overline{\text{POE}}$	
67	$\overline{\text{PWE}}$	
68	PSTROBE	
69	PAUTOFD	
70	$\overline{\text{WAIT}}$	
71	BOOTW	
72	**ABIDRV**	Enables the driving of the A23–A0 outputs
73	A0	
74	A1	
.		
.		
96	A23	

Table 20-2 Main Data Scan Path (continued)

Bit	Cell Name	Comments
97	**BOTDRV**	Enables the drivers for DACK0 through IDP3
98	DACKC	
99	DACKD	
100	DACKA	
101	DACKB	
102	R/W	
103	PIAOE	
104	PIAWE	
105	PIACS0	
106	PIACS1	
107	PIACS2	
108	PIACS3	
109	PIACS4	
110	PIACS5	
111	GACK	
112	WE	
113	TR	
114	CAS0	
115	CAS1	
116	CAS2	
117	CAS3	
118	RAS0	
119	RAS1	
120	RAS2	
121	RAS3	
122	ROMOE	
123	RSWE	
124	BURST	
125	ROMCS0	
126	ROMCS1	
127	ROMCS2	
128	ROMCS3	
129	TXDA	
130	DSRA	
131	UCLK	
132	RXDA	
133	DTRA	
134	RXDB	
135	TXDB	
136	IDPI0	IDP0 input
137	IDPO0	IDP0 output
138	IDPI1	IDP1 input
139	IDPO1	IDP1 output
140	IDPI2	IDP2 input
141	IDPO2	IDP2 output
142	IDPI3	IDP3 input
143	IDPO3	IDP3 output
144	**DBIDRV**	Enables the ID bus drivers
145	IDI0	ID0 input
146	IDO0	ID0 output
147	IDI1	ID1 input
148	IDO1	ID1 output
.		
.		
207	IDI31	ID31 input
208	IDO31	ID31 output

Note: *Drive-enable cells are shown in boldface.*

20.5.3.4 ICTEST1 Path

This is a 5-bit path used to provide quick access to the CNTL1–CNTL0 and the STAT2–STAT0 output signals while keeping other inputs and outputs in their normal system connection.

Table 20-3 ICTEST1 Scan Path

Bit	Cell Name	Comments
1	CNTL0	—
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 STAT2–STAT0 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.

20.5.3.5 ICTEST2 Path

The ICTEST2 path includes only the ID Bus, the CNTL1–CNTL0 and the STAT2–STAT0 signals. It is provided so a hardware-development system can access the processor without disrupting the system.

Table 20-4 ICTEST2 Scan Path

Bit	Cell Name	Comments
1	CNTL0	—
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 STAT2– STAT0 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

20.6 IMPLEMENTING A HARDWARE-DEVELOPMENT SYSTEM

The Halt, Step, and Load Test Instruction modes of operation, invoked using the CNTL1–CNTL0 pins, 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 CNTL1–CNTL0 and STAT2–STAT0 pins.

20.6.1 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 CNTL1–CNTL0 pins. The processor enters the Halt mode within two or three cycles after the CNTL1–CNTL0 pins are 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 CNTL1–CNTL0 pins are 11. However, the processor cannot exit the Halt mode except as the result of the CNTL1–CNTL0 pins 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 by holding the CNTL1–CNTL0 inputs at 10 during a reset: this signals the presence of an external debugger and disables protection checking for Halt instructions until the next processor reset.

In most cases, the STAT2–STAT0 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 STAT2–STAT0 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 CNTL1–CNTL0 pins place the processor into another mode. The only valid transitions on the CNTL1–CNTL0 pins 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.

20.6.2 Step Mode

The Step mode causes the processor 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 on the CNTL1–CNTL0 pins. The processor enters the Step mode within two or three cycles after the CNTL1–CNTL0

pins are changed (depending on synchronization time), except it first completes any external data access in progress.

In most cases, the STAT2–STAT0 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 STAT2–STAT0 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 CNTL1–CNTL0 pins 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 that 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 STAT2–STAT0 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 CNTL1–CNTL0 pins have the value 11 (this prevents the need for a time-critical transition on the CNTL1–CNTL0 pins). The processor remains in this condition until the CNTL1–CNTL0 pins transition to 10 or 01 (or $\overline{\text{RESET}}$ is asserted). The transition to 10 causes the processor to enter the Halt mode and is used to clear the Step mode. The transition to 01 causes the processor to remain in the Step mode so 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 STAT2–STAT0 outputs indicate the Halt or Step mode for one cycle (STAT2–STAT0 = 000). They then indicate the Pipeline Hold mode (STAT2–STAT0 = 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 STAT2–STAT0 outputs as an indication the processor is halted.

20.6.3 Load Test Instruction Mode

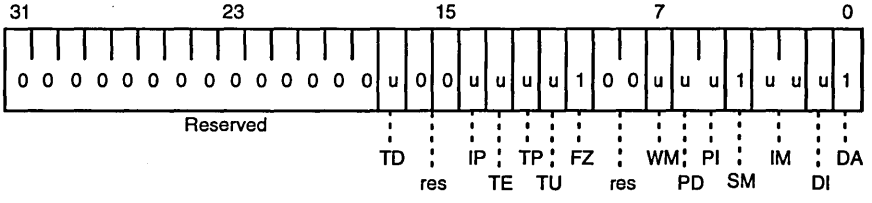
The processor incorporates an Instruction Register (IR) that holds instructions while they are decoded. In the Load Test Instruction mode, the IR is enabled to receive the content of the Instruction Bus regardless of the state of the processor's instruction fetcher. This allows the hardware-development 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 on the CNTL1–CNTL0 pins. The processor enters the Load Test Instruction mode within two or three cycles after the CNTL1–CNTL0 pins are 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 20-4, even though the register is not changed (the value "u" means unaffected).

Figure 20-4 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 20-4 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 STAT2–STAT0 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 CNTL1–CNTL0 pins. 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 CNTL1–CNTL0 pins transition to 11, the processor resumes normal operation. The sequence for the CNTL1–CNTL0 pins to clear the Load Test Instruction mode and resume normal operation is thus 00/10/11.

20.6.4 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 processor 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.

20.6.4.1 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 cells to 10 (Halt) using the ICTEST1 boundary-scan instruction.
2. Set the CNTL cells to 00 (Load Test Instruction) using the ICTEST1 instruction.
3. Using the ICTEST2 instruction, set the IDI31–IDI0 cells with an instruction to load the desired register from the ROM address space with OPT=110, and set the CNTL cells 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 cells through the values 11, 01, 00 (Normal, Step, and Load Test Instruction). This steps the processor and prepares it to receive another instruction.
5. Using the ICTEST2 instruction, set the IDI31–IDI0 cells to 70400101, hexadecimal (NOOP), and set the CNTL cells to 01. This loads a NO-OP into the IR.
6. Using the ICTEST2 instruction, set the IDI31–IDI0 cells to the value to be loaded, and set the CNTL cells to 11. This steps the processor and applies the value to be loaded into the register.
7. Set the CNTL cells to 01 using the ICTEST1 instruction.
8. Repeat steps 2 through 7 for the remaining registers.

20.6.4.2 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 cells to 10 (Halt) using the ICTEST1 boundary-scan instruction.
2. Set the CNTL cells to 00 (Load Test Instruction) using the ICTEST1 instruction.
3. Using the ICTEST2 instruction, set the IDI31–IDI0 cells with an instruction to store the desired register into the ROM address space, with OPT=110, and set the CNTL cells to 01 (Step). This places the store instruction into the IR and prepares the processor to step.
4. Sequence the CNTL cells through the values 11, 01, 00 (Normal, Step, Load Test Instruction). This steps the processor and prepares it to receive another instruction.
5. Using the ICTEST2 instruction, set the IDI31–IDI0 cells to 70400101, hexadecimal (NO-OP), and set the CNTL cells to 01. This loads a NO-OP into the IR.
6. Set the CNTL cells 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.

20.6.5 Forcing Outputs to High Impedance

A hardware-development system can force processor outputs to the high-impedance state using the HIZ instruction of the Test Access Port or by asserting the $\overline{\text{TRIST}}$ pin.

20.7 TRACEABLE CACHE™ TECHNOLOGY FEATURE

The Am29240 microcontroller series incorporates a Traceable Cache technology feature similar to the Am29030 and Am29035 microprocessors. Traceable Cache technology permits a hardware-development system to trace the instruction execution of the processor while the processor is executing out of the instruction cache.

Instruction tracing is accomplished using two processors in tandem: a main processor and a tracing processor. The main processor performs all the required operations and the tracing processor duplicates the operation of the main processor, except that it uses the outputs $\overline{\text{PIACS5}}$–$\overline{\text{PIACS0}}$, $\overline{\text{PIAWE}}$, $\overline{\text{PIAOE}}$, A24–A0, STAT2–STAT0, and $\overline{\text{GACK}}$ to indicate the instruction trace. The tracing processor is connected in parallel to the main processor with all of its outputs disabled, similar to a master/slave connection in other 29K Family processors, except that the outputs $\overline{\text{PIACS5}}$–$\overline{\text{PIACS0}}$, $\overline{\text{PIAWE}}$, $\overline{\text{PIAOE}}$, A24–A0, STAT2–STAT0, and $\overline{\text{GACK}}$ of the tracing processor are left unconnected to the main processor. Because the tracing processor uses some outputs to indicate the instruction trace, the tracing processor relies on the main processor to perform all accesses on its behalf. The tracing processor simply latches the results of accesses by the main processor. Also, all processor outputs are driven by the main processor. Both the enabling of the tracing feature and the disabling of the tracing processor's outputs are accomplished via the boundary-scan interface.

Address tracing reflects the full, internal, 32-bit address on the tracing processor's pins, as follows:

Internal Address Bits	Tracing Processor Pins
31–26	$\overline{\text{PIACS5}}$–$\overline{\text{PIACS0}}$
25	$\overline{\text{PIAWE}}$
24	$\overline{\text{PIAOE}}$
23–0	A23–A0

20.7.1 Status Outputs of Tracing Processor

The STAT2–STAT0 outputs on the tracing processor contain information that is not provided by the main processor. Primarily, the tracing processor indicates internal accesses to the data cache and differentiates load and store accesses, whereas the main processor indicates only external accesses and that the external access is a data access. The tracing processor also indicates a return from interrupt, in the cycle that the first instruction of the target routine is executed.

The status encodings of the tracing processor are as follows:

STAT2	STAT1	STAT0	Condition
1	0	0	Load access (internal access and cache hit, or external access and data valid)
1	0	1	Store access (internal access and cache hit, or external access and data valid)
1	1	0	Return from interrupt (first target instruction cache hit or valid on ID Bus)
— all others —			Same as master processor

As with the main processor, the STAT2–STAT0 outputs are driven at the processor frequency. If the processor operates at the system frequency, the STAT2–STAT0 outputs are driven on every rising edge of MEMCLK to reflect the state of the processor during the previous MEMCLK period. If the processor operates at twice the system frequency (that is, if the processor is in turbo mode), the STAT2–STAT0 outputs are driven on every rising and falling edge of MEMCLK to reflect the state of the processor during the previous processor cycle. If the STAT2–STAT0 outputs reflect external access activity, such as a load access, and the processor is in turbo mode, the status indication is driven during the first half-cycle of MEMCLK (MEMCLK High) to reflect the state of the access at the end of the previous MEMCLK period.

20.7.2 Instruction Address Tracing

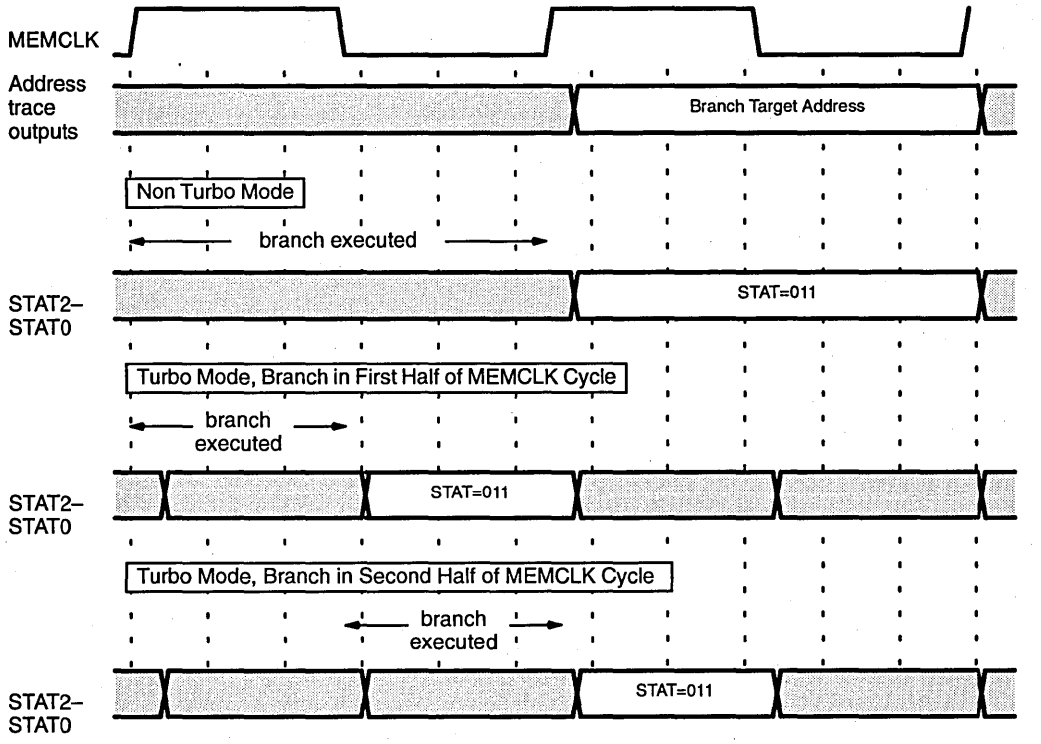
Since the processor can operate at twice the system frequency, and because of the number of address bus signals, it is not possible to reflect all processor addresses on the outputs of the tracing processor. Thus, the tracing processor reflects only the target addresses of branches and relies on the hardware-development system to reconstruct the trace between branches. When the tracing processor executes a branch, the tracing processor drives the address of the target of the branch on the address bus in the MEMCLK cycle following the execution of the branch. The branch target address appears for a full MEMCLK cycle. However, the STAT2–STAT0 signals still reflect the state of the processor on every cycle. When the tracing processor executes a branch, it drives STAT2–STAT0 to 011 in the next processor cycle for the duration of a single processor cycle. Because the branch target addresses are driven at the MEMCLK rate while the STAT2–STAT0 indication is driven at the processor clock rate, there are several relative timings of the status relative to the branch target address, as shown in Figure 20-5.

The tracing processor does not allow tracing of an instruction sequence that uses visiting (that is, when a branch is in the delay slot of another branch), since the tracing processor drives branch target address for a full MEMCLK cycle. In this case, the final branch target address is reflected and the visited instruction address is lost.

20.7.3 Data Access Tracing

When a load or store passes through the execute stage of the processor pipeline, the tracing processor drives the corresponding address on the trace outputs in the next MEMCLK cycle. The relative timing of the load or store indication on STAT2–STAT0 and the load or store address is similar to the relative timing shown in Figure 20-5 between a branch indication and the branch target address. However, if the processor is in the turbo mode and executes a branch in the same MEMCLK cycle as the load or store, the branch target address is driven instead of the load or store address.

Figure 20-5 Possible Timing of STAT2–STAT0 Signals Relative to Branch Target Address in Tracing Processor



The contents of the data cache can be reconstructed by the hardware-development system, if necessary, because the data cache implements a write-through policy and thus reflects all writes on the external interface. These writes appear on the interface of the main processor.

20.7.4 Pipeline Hold Indication

The tracing processor uses its $\overline{\text{GACK}}$ output to indicate that it and the main processor are in the Pipeline Hold mode. The timing of this output is similar to the STAT2–STAT0 outputs, because it is driven at the processor frequency and thus changes on both the rising and falling edges of MEMCLK when the processor is in the Turbo mode. If the tracing processor is in the Pipeline Hold mode on any given cycle, it drives the $\overline{\text{GACK}}$ output Low in the next processor cycle; otherwise it drives $\overline{\text{GACK}}$ High.

CHAPTER

21 INSTRUCTION SET

This chapter provides a specification of the Am29240 microcontroller series instruction set. Sections 21.1 and 21.2 describe the terminology and the instruction formats. Section 21.3 describes each instruction in detail; instructions are presented alphabetically by assembler mnemonic. Finally, Section 21.4 gives an index of instructions by operation code.

21.1 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.

21.1.1 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	Boolean constant FALSE
FC	Funnel Shift Count (FC) field of the ALU Status Register
h'*n*'	hexadecimal constant *n*
I16	16-bit immediate data
IPA	Indirect Pointer A Register

IPB	Indirect Pointer B Register
IPC	Indirect Pointer C Register
PC	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	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 21.2).
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 21.2).
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	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.

21.1.2 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 that 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

21.1.3 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 19.2.2).

21.1.4 Assembler Syntax

This chapter does not contain a full description of the instruction assembler, but provides a rudimentary description of the assembler syntax.

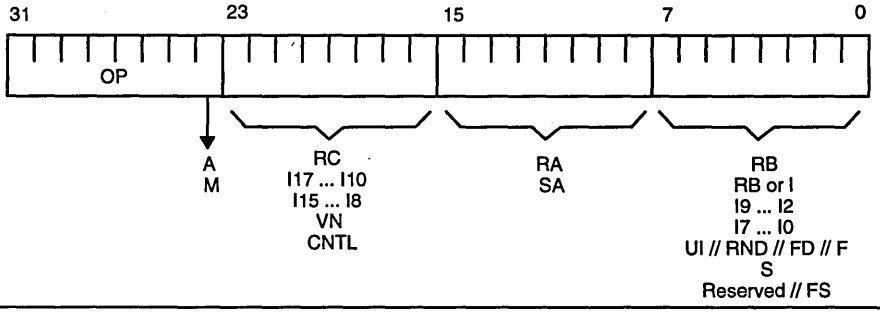
The following notation is used to describe assembler tokens:

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.

21.2 INSTRUCTION FORMATS

All instructions for the Am29240 microcontroller series are 32 bits in length and are divided into four fields, as shown in Figure 21-1. 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 21-1 Instruction Format



The instruction fields are defined as follows:

Bits 31–24

OP — This field contains an operation code, that defines 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.

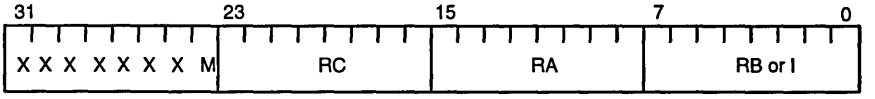
The fields described above may appear in many combinations. However, certain combinations that appear frequently are shown in Figure 21-2.

21.3 INSTRUCTION DESCRIPTION

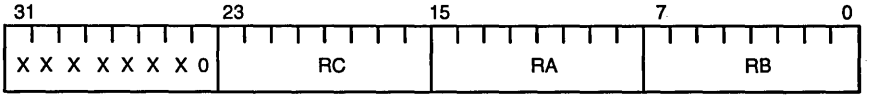
This section describes each instruction in detail. Figure 21-3 illustrates the layout of the information given for each description.

Figure 21-2 Frequently Occurring Instruction Field Uses

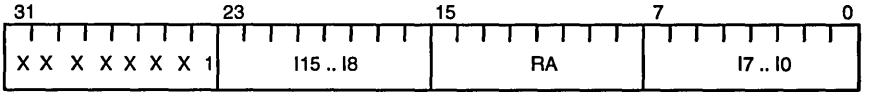
Three operands with possible 8-bit constant:



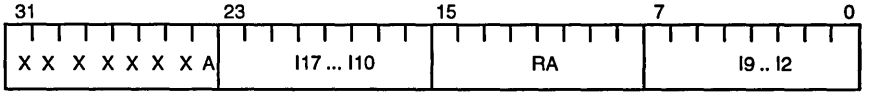
Three operands without constant:



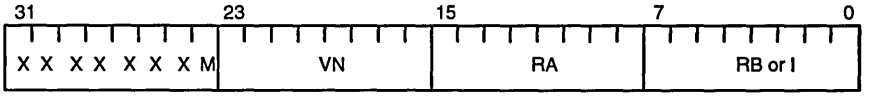
One register operand with 16-bit constant:



Jumps and calls with 16-bit instruction address:



Two operands with trap vector number:



Loads and stores:

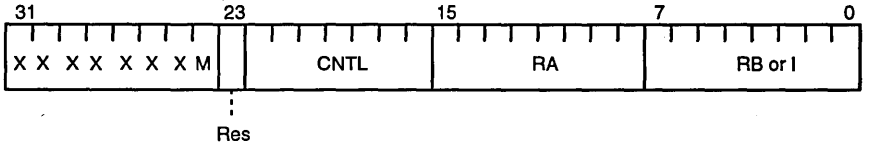
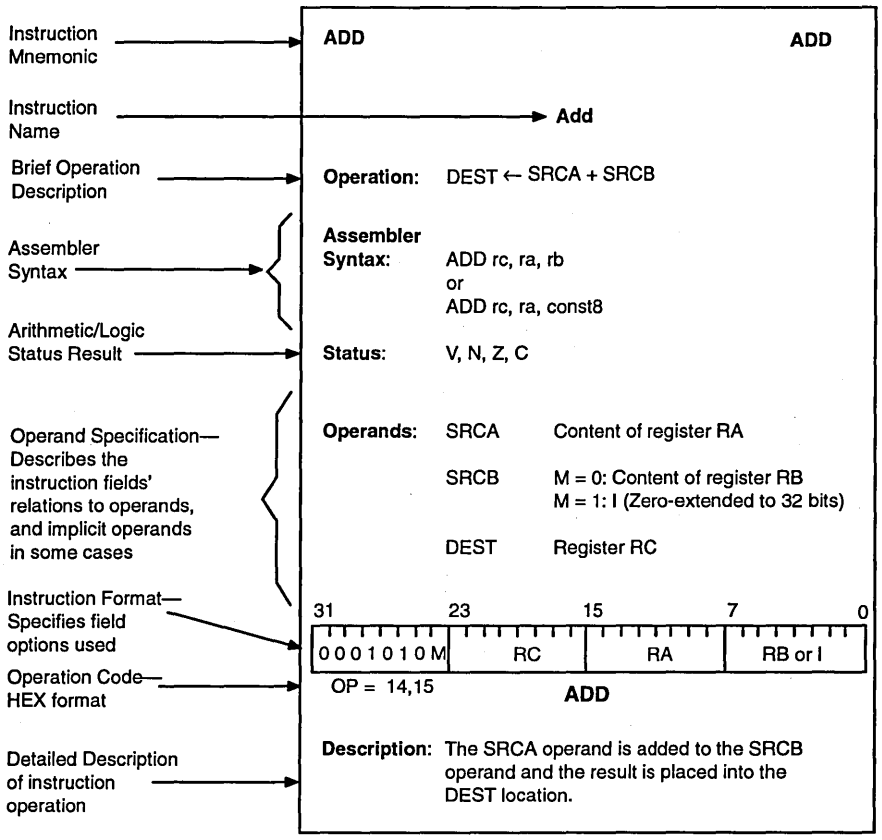


Figure 21-3 Instruction-Description Format



Add

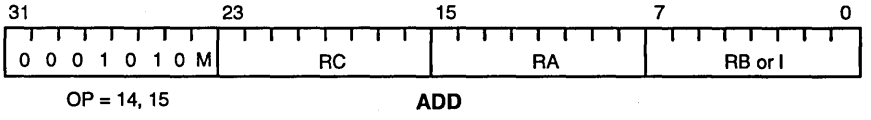
Operation: DEST ← SRCA + SRCB

Assembler Syntax:
ADD rc, ra, rb
or
ADD rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



Description: The SRCA operand is added to the SRCB operand and the result is placed into the DEST location.

Add with Carry

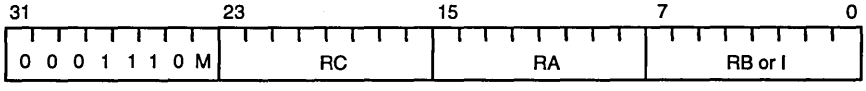
Operation: DEST ← SRCA + SRCB + C

Assembler Syntax:
ADDC rc, ra, rb
or
ADDC rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 1C, 1D

ADDC

Description: The SRCA operand is added to the SRCB operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location.

Add with Carry, Signed

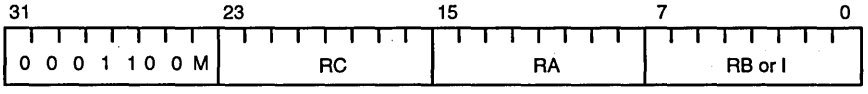
Operation: DEST ← SRCA + SRCB + C
IF signed overflow THEN Trap (Out of Range)

Assembler Syntax: ADDCS rc, ra, rb
or
ADDCS rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 18, 19 **ADDCS**

Description: The SRCA operand is added to the SRCB operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location. If the add operation causes a two's-complement signed overflow, an Out-of-Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

Add with Carry, Unsigned

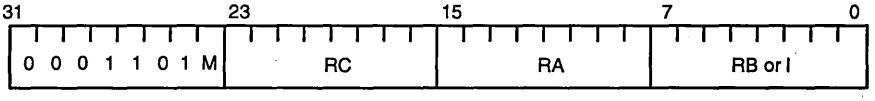
Operation: DEST ← SRCA + SRCB + C
IF unsigned overflow THEN Trap (Out of Range)

Assembler Syntax:
ADDCU rc, ra, rb
or
ADDCU rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 1A, 1B **ADDCU**

Description: The SRCA operand is added to the SRCB operand and the value of the ALU Status Carry bit, and the result is placed into the DEST location. If the add operation causes an unsigned overflow, an Out-of-Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

Add, Signed

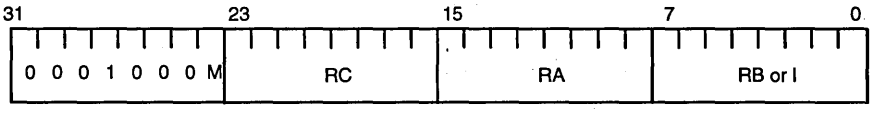
Operation: DEST ← SRCA + SRCB
IF signed overflow THEN Trap (Out of Range)

Assembler Syntax:
ADDS rc, ra, rb
or
ADDS rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 10, 11 **ADDS**

Description: The SRCA operand is added to the SRCB operand and the result is placed into the DEST location. If the add operation causes a two's-complement signed overflow, an Out-of-Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

Add, Unsigned

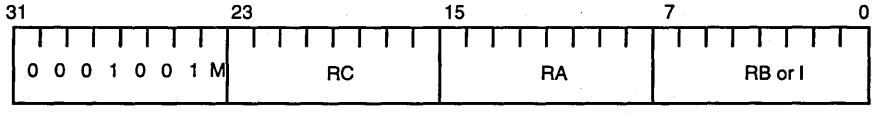
Operation: DEST ← SRCA + SRCB
IF unsigned overflow THEN Trap (Out of Range)

Assembler Syntax:
ADDU rc, ra, rb
or
ADDU rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 12, 13 **ADDU**

Description: The SRCA operand is added to the SRCB operand and the result is placed into the DEST location. If the add operation causes an unsigned overflow, an Out-of-Range trap occurs.

Note that the DEST location is altered whether or not an overflow occurs.

AND Logical

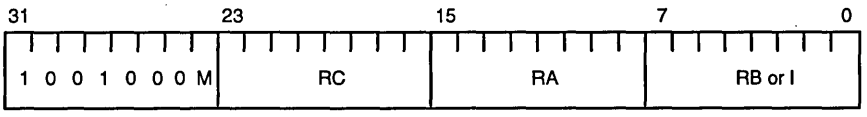
Operation: DEST ← SRCA & SRCB

Assembler Syntax: AND rc, ra, rb
or
AND rc, ra, const8

Status: N, Z

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 90, 91 **AND**

Description: The SRCA operand is logically ANDed, bit-by-bit, with the SRCB operand and the result is placed into the DEST location.

AND-NOT Logical

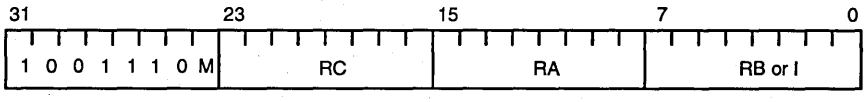
Operation: DEST ← SRCA & ~SRCB

Assembler Syntax:
ANDN rc, ra, rb
or
ANDN rc, ra, const8

Status: N, Z

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	register RC



OP = 9C, 9D

ANDN

Description: The SRCA operand is logically ANDed, bit-by-bit, with the one's-complement of the SRCB operand and the result is placed into the DEST location.

Assert Equal To

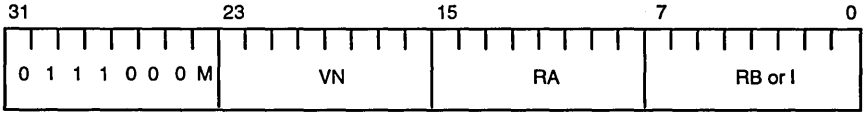
Operation: IF SRCA = SRCB THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASEQ vn, ra, rb
or
ASEQ vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 70, 71 **ASEQ**

Description: If the SRCA operand is equal to the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Greater Than or Equal To

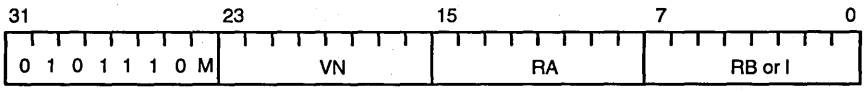
Operation: IF SRCA ≥ SRCB THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASGE vn, ra, rb
or
ASGE vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 5C, 5D

ASGE

Description: If the value of the SRCA operand is greater than or equal to the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Greater Than or Equal To, Unsigned

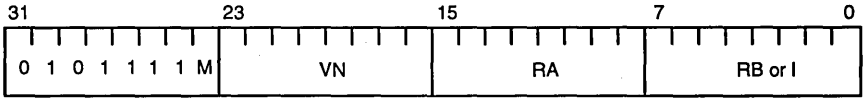
Operation: IF SRCA ≥ SRCB (unsigned) THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASGEU vn, ra, rb
or
ASGEU vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 5E, 5F

ASGEU

Description: If the value of the SRCA operand is greater than or equal to the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs. For the comparison, both operands are treated as unsigned integers.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Greater Than

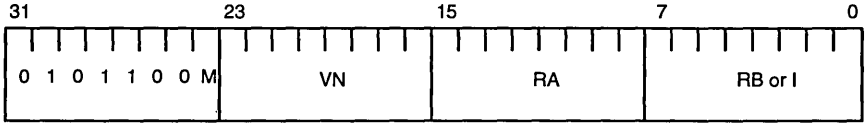
Operation: IF SRCA > SRCB THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASGT vn, ra, rb
or
ASGT vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M =1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 58, 59

ASGT

Description: If the value of the SRCA operand is greater than the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Greater Than, Unsigned

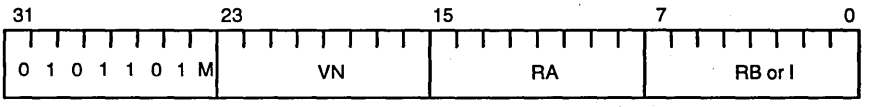
Operation: IF SRCA > SRCB (unsigned) THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASGTU vn, ra, rb
or
ASGTU vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 5A, 5B

ASGTU

Description: If the value of the SRCA operand is greater than the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs. For the comparison, both operands are treated as unsigned integers.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Less Than or Equal To

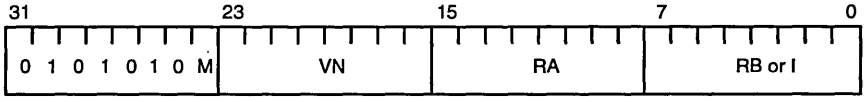
Operation: IF SRCA ≤ SRCB THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASLE vn, ra, rb
or
ASLE vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 54, 55

ASLE

Description: If the value of the SRCA operand is less than or equal to the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Less Than or Equal To, Unsigned

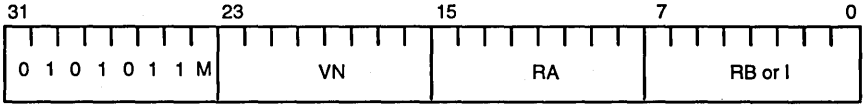
Operation: IF SRCA ≤ SRCB (unsigned) THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASLEU vn, ra, rb
or
ASLEU vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 56, 57 **ASLEU**

Description: If the value of the SRCA operand is less than or equal to the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs. For the comparison, both operands are treated as unsigned integers.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Less Than

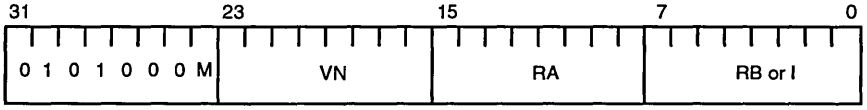
Operation: IF SRCA < SRCB THEN Continue
ELSE Trap(VN)

Assembler Syntax:
ASLT vn, ra, rb
or
ASLT vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 50, 51

ASLT

Description: If the value of the SRCA operand is less than the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Less Than, Unsigned

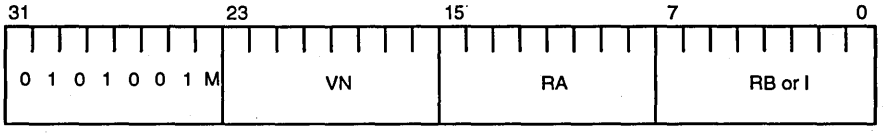
Operation: IF SRCA < SRCB (unsigned) THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASLTU vn, ra, rb
or
ASLTU vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 52, 53 **ASLTU**

Description: If the value of the SRCA operand is less than the value of the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs. For the comparison, both operands are treated as unsigned integers.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Assert Not Equal To

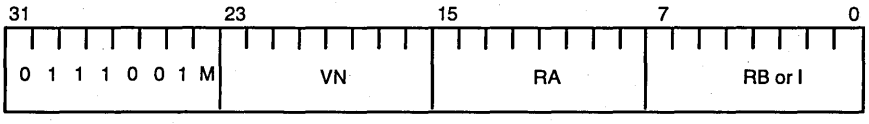
Operation: IF SRCA <> SRCB THEN Continue
ELSE Trap (VN)

Assembler Syntax:
ASNEQ vn, ra, rb
or
ASNEQ vn, ra, const8

Status: Not affected

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
VN	Trap vector number



OP = 72, 73

ASNEQ

Description: If the SRCA operand is not equal to the SRCB operand, instruction execution continues; otherwise, a trap with the specified vector number occurs.

For programs in the User mode, a Protection Violation trap occurs—instead of the assert trap—if a vector number between 0 and 63 is specified.

Call Subroutine

Operation: DEST ← PC // 00 + 8
PC ← TARGET
Execute delay instruction

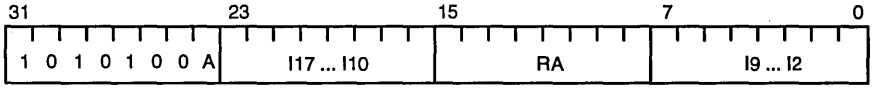
Assembler Syntax: CALL ra, target

Status: Not affected

Operands:

TARGET	A = 0: I17 ... I10 // I9 ... I2 (sign-extended to 30 bits) + PC
	A = 1: I17 ... I10 // I9 ... I2 (zero-extended to 30 bits)

DEST	Register RA



OP = A8, A9 **CALL**

Description: The address of the second following instruction is placed into the DEST location and a non-sequential instruction fetch occurs to the instruction address given by the TARGET operand. The instruction following the CALL is executed before the non-sequential fetch occurs.

Call Subroutine, Indirect

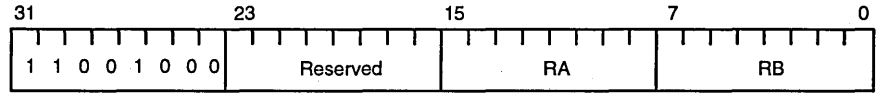
Operation: DEST ← PC // 00 + 8
PC ← SRCB
Execute delay instruction

Assembler Syntax: CALLI ra, rb

Status: Not affected

Operands:

SRCB	Content of register RB
DEST	Register RA



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

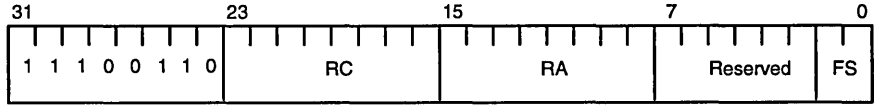
Status: None

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

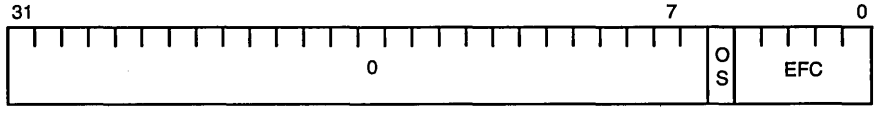
Control:

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

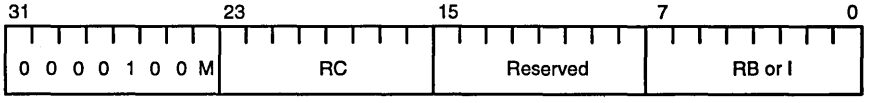
Operation: DEST ← count of number of leading zeros in SRCB or I

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



OP = 08,09

CLZ

Description: A count of the number of zero-bits to the first one-bit in the SRCB operand is placed into the DEST location. If the most significant bit of the SRCB operand is 1, the resulting count is zero. If the SRCB operand is zero, the resulting count is 32.

Constant

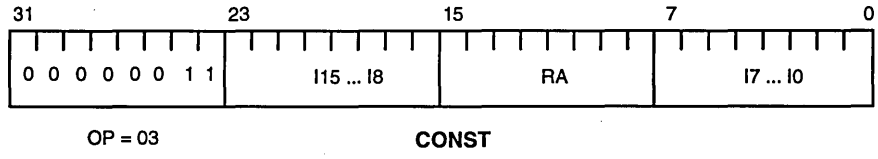
Operation: DEST ← 0I16

Assembler Syntax: CONST ra, const16

Status: Not affected

Operands:

0I16	I15 ... 8 // I7 ... I0 (Zero-extended to 32 bits)
DEST	Register RA



Description: The 0I16 operand is placed into the DEST location.

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

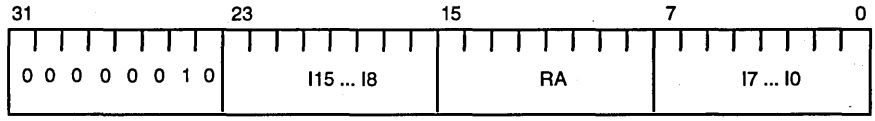
Operation: Replace high-order half-word of SRCA by I16

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

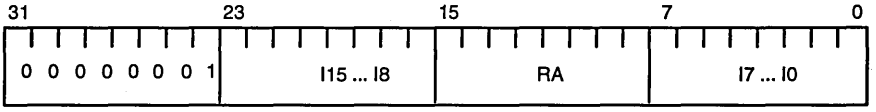
Operation: DEST ← 1I16

Assembler Syntax: CONSTN ra, const16

Status: Not affected

Operands:

	1I16	I15 ... I8 // I7 ... I0 (ones-extended to 32 bits)
	DEST	Register RA



OP = 01 **CONSTN**

Description: The 1I16 operand is placed into the DEST location.

Convert Data Format

Operation: DEST ← SRCA, with format modified per UI, RND, FD, FS

Assembler Syntax: CONVERT rc, ra, UI, RND, FD, FS

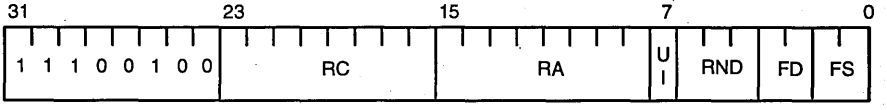
Status: fpX, fpU, fpV, fpR, fpN

Operands:

SRCA	Content of register RA (single-precision 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

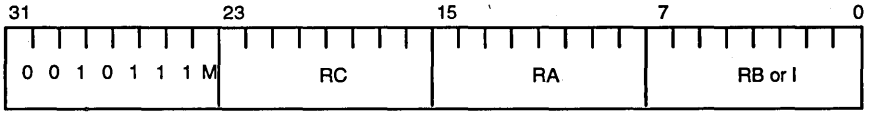
Operation:
IF (SRCA.BYTE0 = SRCB.BYTE0) OR
(SRCA.BYTE1 = SRCB.BYTE1) OR
(SRCA.BYTE2 = SRCB.BYTE2) OR
(SRCA.BYTE3 = SRCB.BYTE3) THEN
DEST ← TRUE ELSE DEST ← FALSE

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

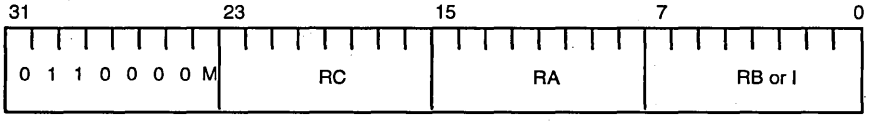
Operation: IF SRCA = SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

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

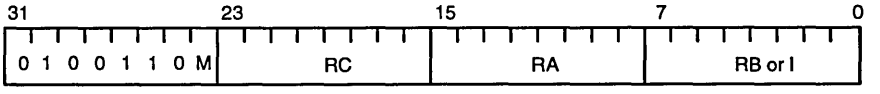
Operation: IF SRCA ≥ SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

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

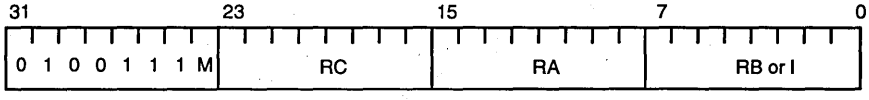
Operation: IF SRCA ≥ SRCB (unsigned) THEN DEST ← TRUE
ELSE DEST ← FALSE

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

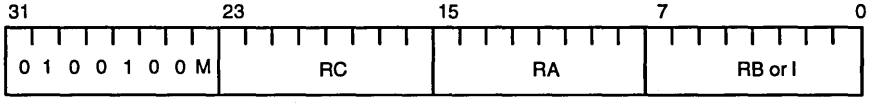
Operation: IF SRCA > SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

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

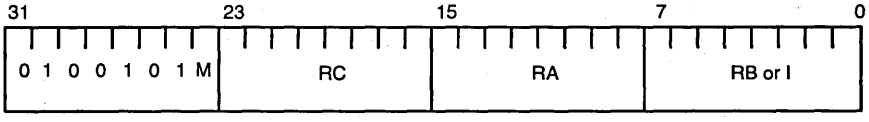
Operation: IF SRCA > SRCB (unsigned) THEN DEST ← TRUE
ELSE DEST ← FALSE

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

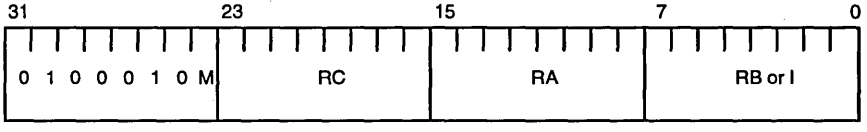
Operation: IF SRCA ≤ SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

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

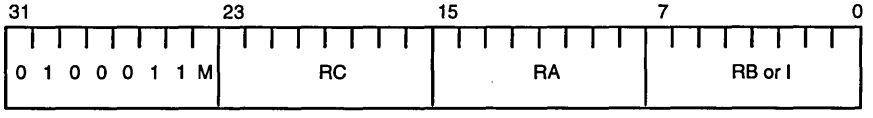
Operation: IF SRCA ≤ SRCB (unsigned) THEN DEST ← TRUE
ELSE DEST ← FALSE

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

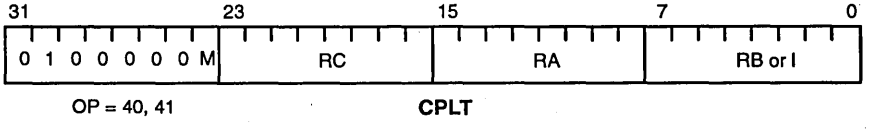
Operation: IF SRCA < SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

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



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

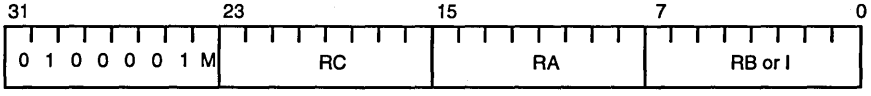
Operation: IF SRCA < SRCB (unsigned) THEN DEST ← TRUE
ELSE DEST ← FALSE

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

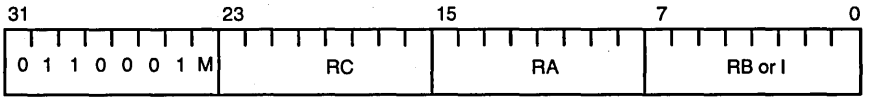
Operation: IF SRCA <> SRCB THEN DEST ← TRUE
ELSE DEST ← FALSE

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

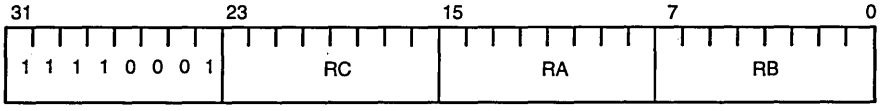
Operation: DEST (double-precision) ← SRCA (double-precision) + SRCB (double-precision)

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

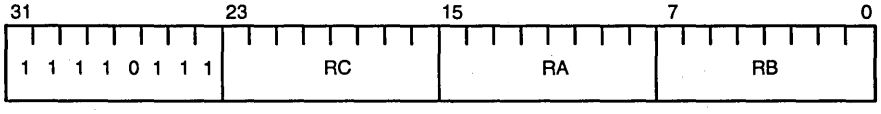
Operation: DEST (double-precision) ← SRCA (double-precision) / SRCB (double-precision)

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

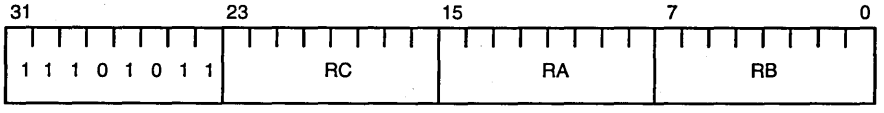
Operation: IF SRCA (double-precision) = SRCB (double-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

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

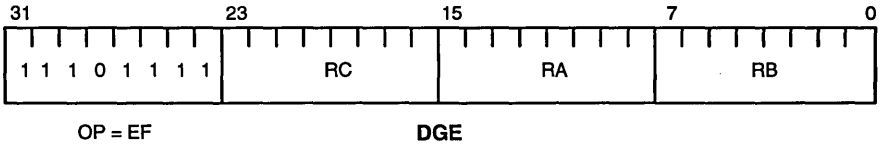
Operation: IF SRCA (double-precision) ≥ SRCB (double-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

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



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

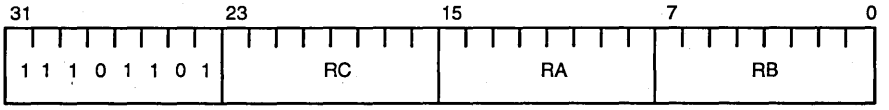
Operation: IF SRCA (double-precision) > SRCB (double-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

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

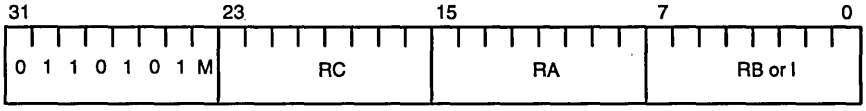
Operation: Perform one-bit step of a divide operation (unsigned)

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 2.6.4.

Divide Initialize

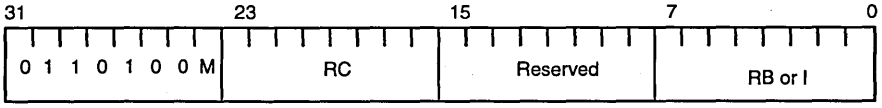
Operation: Initialize for a sequence of divide steps (unsigned)

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 2.6.4.

Integer Divide, Signed

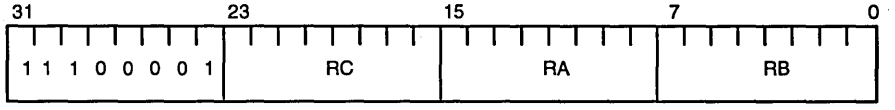
Operation: DEST ← (Q // SRCA) / SRCB (signed)
Q ← Remainder

Assembler Syntax: DIVIDE rc, ra, rb

Status: Not affected

Operands:

Q	Content of the Q Register
SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = E1 **DIVIDE**

Description: The SRCA operand is appended to the content of the Q register. The resulting 64-bit value is divided by the SRCB operand and the result is placed into the DEST location. This operation treats the operands as signed two's-complement integers and produces a signed two's-complement result.

The remainder is placed into the Q register. A non-zero remainder always has the same sign as the dividend.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes a DIVIDE trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Integer Divide, Unsigned

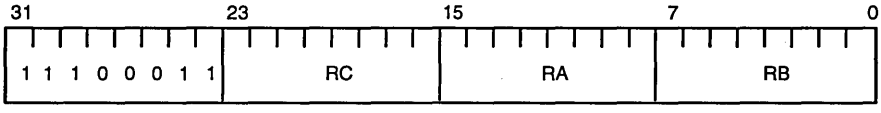
Operation: DEST ← (Q // SRCA) / SRCB (unsigned)
Q ← Remainder

Assembler Syntax: DIVIDU rc, ra, rb

Status: Not affected

Operands:

Q	Content of the Q Register
SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = E3 **DIVIDU**

Description: The SRCA operand is appended to the content of the Q Register. The resulting 64-bit value is divided by the SRCB operand and the result is placed into the DEST location. This operation treats the operands as unsigned integers and produces an unsigned result.

The remainder is placed into the Q Register. The remainder is also unsigned.

Note: This instruction is not supported directly in processor hardware. In the current implementation this instruction causes a DIVIDU trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Divide Last Step

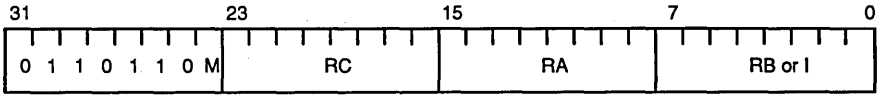
Operation: Complete a sequence of divide steps (unsigned)

Assembler Syntax: DIVL rc, ra, rb

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 6C, 6D DIVL

Description: If the Divide Flag (DF) bit of the ALU Status Register is 1, the SRCB operand is subtracted from the SRCA operand. If the DF bit is 0, the SRCB operand is added to the SRCA operand. The result is placed into the DEST location.

The carry-out of the add or subtract operation is exclusive-ORed with the value of the DF bit and the value of the Negative (N) bit of the ALU Status Register; the resulting value is complemented and placed into the DF bit. The sign of the result of the add or subtract is placed into the N bit.

The content of the Q register is shifted left by one bit position; the value computed for the DF bit above fills the vacated bit position. The shifted value is placed into the Q Register.

Examples of integer divide operations appear in Section 2.6.4.

Divide Remainder

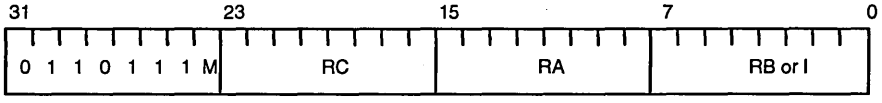
Operation: Generate remainder for divide operation (unsigned)

Assembler Syntax:
DIVREM rc, ra, rb
or
DIVREM rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 6E, 6F **DIVREM**

Description: If the Divide Flag (DF) bit of the ALU Status Register is 1, the SRCA operand is placed into the DEST location.

If the DF bit is 0, the SRCB operand is added to the SRCA operand and the result is placed into the DEST location.

Examples of integer divide operations appear in Section 2.6.4.

Floating-Point Multiply, Double-Precision

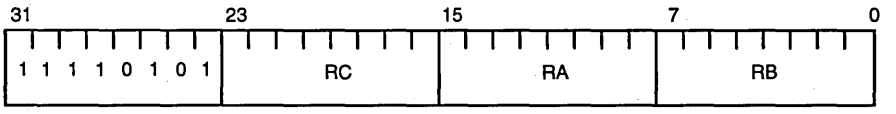
Operation: DEST (double-precision) ← SRCA (double-precision) * SRCB (double-precision)

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

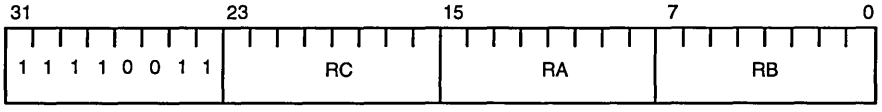
Operation: DEST (double-precision) ← SRCA (double-precision) – SRCB (double-precision)

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

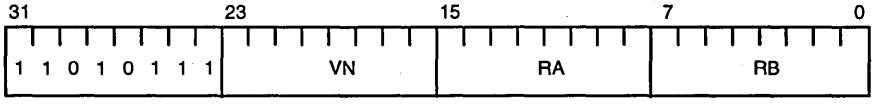
Operation: Load IPA and IPB registers with operand register numbers and Trap (VN)

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

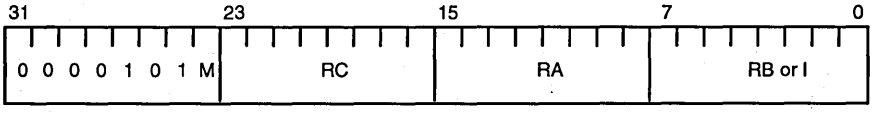
Operation: DEST ← SRCB, with low-order byte replaced by byte in SRCA selected by BP

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. 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 3.3.5.1.

Extract Half-Word

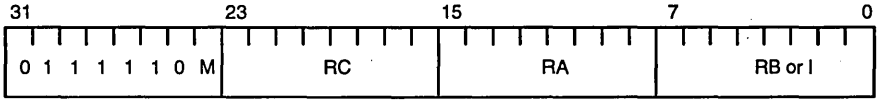
Operation: DEST ← SRCB, with low-order half-word replaced by half-word in SRCA selected by BP

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. 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 3.3.5.1.

Extract Half-Word, Sign-Extended

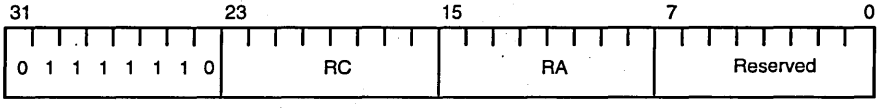
Operation: DEST ← half-word in SRCA selected by BP, sign-extended to 32 bits

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. 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 3.3.5.1.

Extract Word, Bit-Aligned

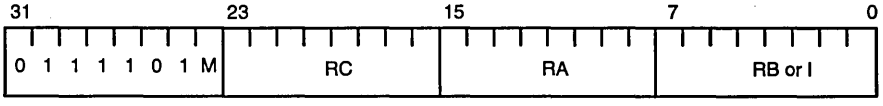
Operation: DEST ← high-order word of (SRCA // SRCB << FC)

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

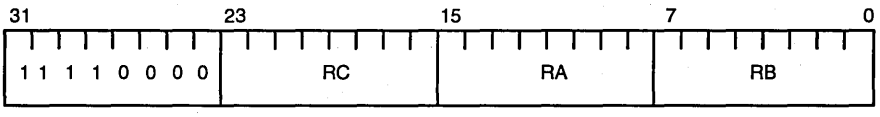
Operation: DEST (single-precision) ← SRCA (single-precision) + SRCB (single-precision)

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

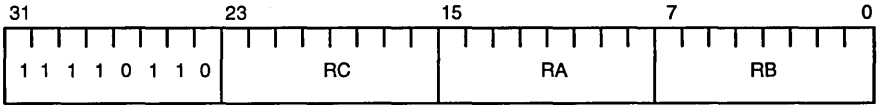
Operation: DEST (single-precision) ← SRCA (single-precision) / SRCB (single-precision)

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

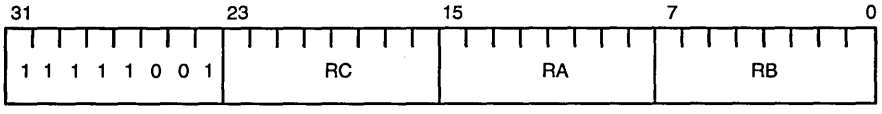
Operation: DEST (double-precision) ← SRCA (single-precision) * SRCB (single-precision)

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

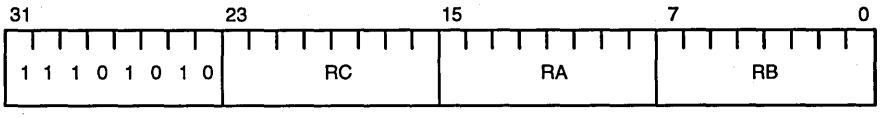
Operation: IF SRCA (single-precision) = SRCB (single-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

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

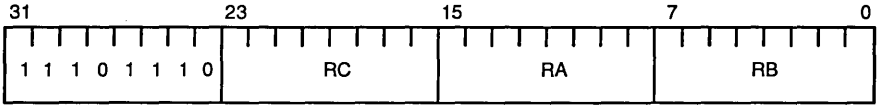
Operation: IF SRCA (single-precision) ≥ SRCB (single-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

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

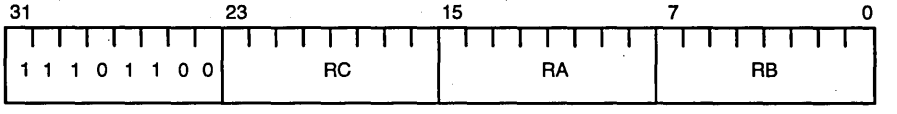
Operation: IF SRCA (single-precision) > SRCB (single-precision)
THEN DEST ← TRUE
ELSE DEST ← FALSE

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

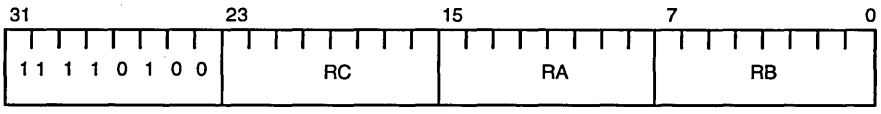
Operation: DEST (single-precision) ← SRCA (single-precision) * SRCB (single-precision)

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

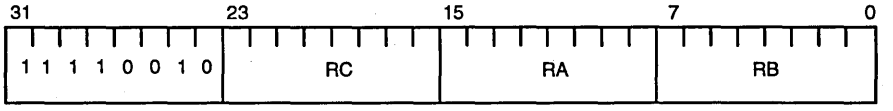
Operation: DEST (single-precision) ← SRCA (single-precision) – SRCB (single-precision)

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.

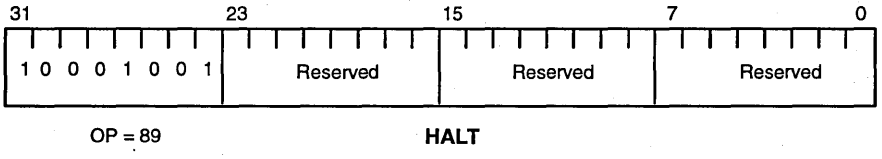
Enter Halt Mode

Operation: Enter Halt mode on next cycle

Assembler Syntax: HALT

Status: Not affected

Operands: Not applicable



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.

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

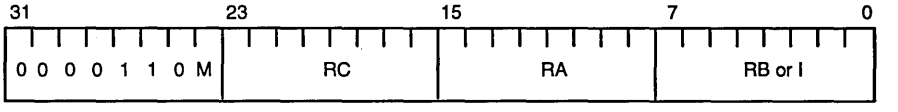
Operation: DEST ←SRCA, with byte selected by BP replaced by low-order byte of SRCB

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. 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 3.3.5.1.

Insert Half-Word

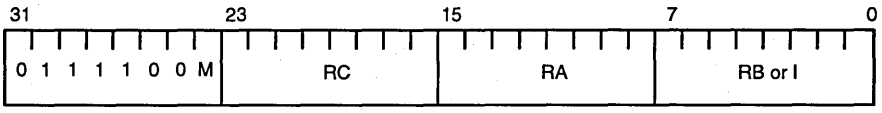
Operation: DEST ←SRCA, with half-word selected by BP replaced by low-order half-word of SRCB

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. 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 3.3.5.1.

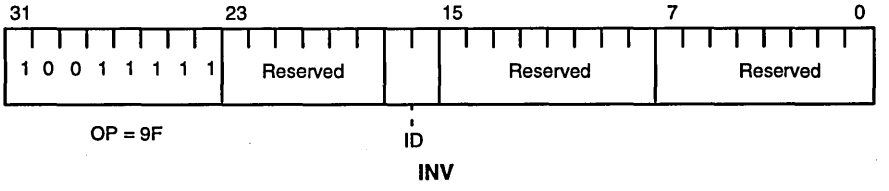
Invalidate

Operation: None

Assembler Syntax: INV [ID]

Status: Not affected

Operands: The optional parameter ID



Description: This instruction resets all cache valid bits in the instruction cache, the data cache, or both caches. Bits 17–16 of the INV instruction select the cache to be invalidated, as follows:

Bits 17–16	Effect on Cache
00	Both caches invalidated
01	Instruction cache invalidated
10	Data cache invalidated
11	Reserved

This instruction may be executed only by Supervisor-mode programs. An attempted execution by a User-mode program causes a Protection Violation trap to occur.

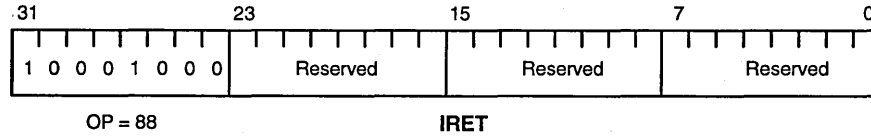
Interrupt Return

Operation: Perform an interrupt return sequence

Assembler Syntax: IRET

Status: Not affected

Operands: Not applicable



Description: This instruction performs the interrupt return sequence described in Section 19.3.4.

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.

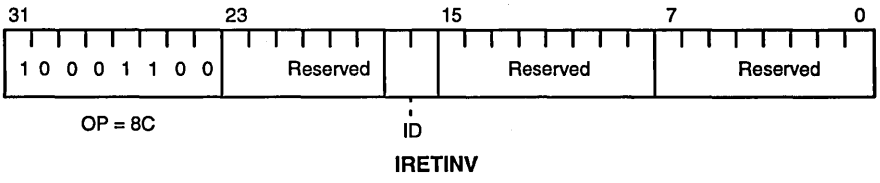
Interrupt Return and Invalidate

Operation: Perform an interrupt return sequence

Assembler Syntax: IRETINV [ID]

Status: Not affected

Operands: The optional parameter ID



IRETINV

Description: This instruction performs the interrupt return sequence described in Section 19.3.4. This instruction also resets all cache valid bits in the instruction cache, the data cache, or both caches. Bits 17–16 of the INV instruction select the cache to be invalidated, as follows:

Bits 17–16	Effect on Cache
00	Both caches invalidated
01	Instruction cache invalidated
10	Data cache invalidated
11	Reserved

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.

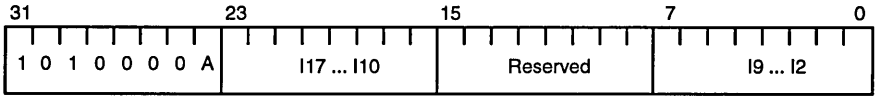
Jump

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

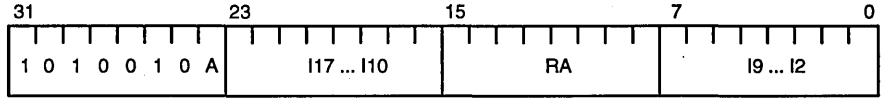
Operation: IF SRCA = FALSE THEN PC ← TARGET
Execute delay instruction

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
```

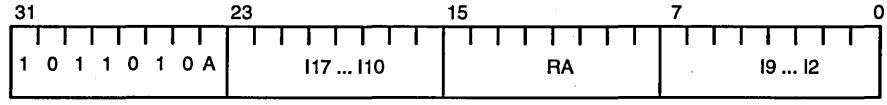
Assembler Syntax: JMPFDEC ra, target

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

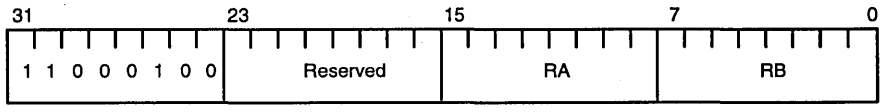
Operation: IF SRCA = FALSE THEN PC ← SRCB
Execute delay instruction

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.

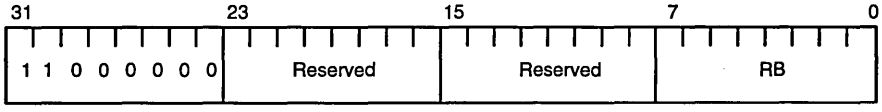
Jump Indirect

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

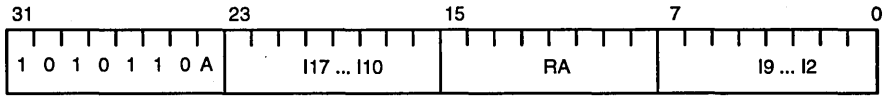
Assembler Syntax: JMPT ra, target

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

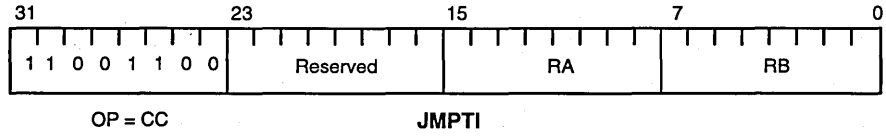
Operation: IF SRCA = TRUE THEN PC ← SRCB
Execute delay instruction

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

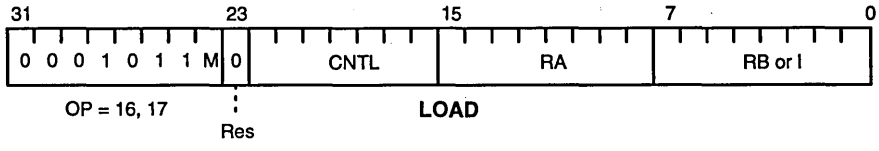
Operation: DEST ← EXTERNAL WORD [SRCB]

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 access as described in Section 3.3.1.

Load and Lock

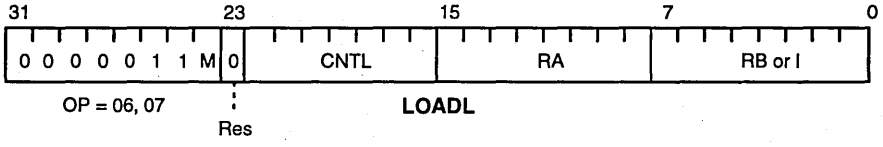
Operation: DEST ← EXTERNAL WORD [SRCB]

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 access as described in Section 3.3.1.

The load is treated as a noncacheable access that loads only from the external memory. The write buffer is emptied before the load is performed, and, if the associated block is found in the data cache, the block is invalidated.

Load Multiple

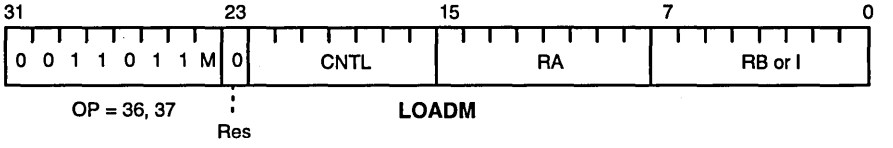
Operation: DEST... DEST+COUNT ← EXTERNAL WORD [SRCB] ...
EXTERNAL WORD [SRCB + (COUNT * 4)]

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 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

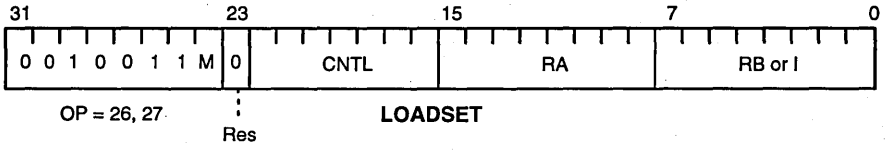
Operation: DEST ← EXTERNAL WORD [SRCB]
EXTERNAL WORD [SRCB] ← h'FFFFFFFF'

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 access as described in Section 3.3.1.

The load and store are treated as noncacheable accesses that are performed only in the external memory. The write buffer is emptied before the load/store is performed and, if the associated block is found in the data cache, the block is invalidated.

Move from Special Register

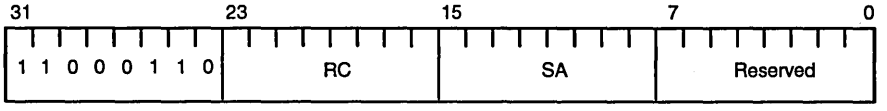
Operation: DEST ← SPECIAL

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

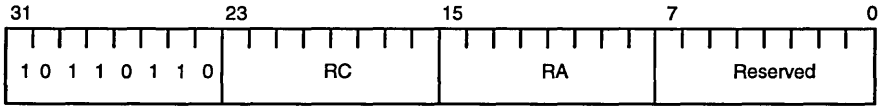
Operation: None

Assembler Syntax: MFTLB rc, ra

Status: Not affected

Operands:

SRCA	Content of register RA, bits 6 ... 0
DEST	Register RC



OP = B6 **MFTLB**

Description: The Translation Look-Aside Buffer (TLB) register whose register number is specified by the SRCA operand is placed into the DEST location.

This instruction may be executed only by Supervisor-mode programs. An attempted execution by a User-mode program causes a Protection Violation trap to occur. If a trap occurs, the DEST location is not altered.

Move to Special Register

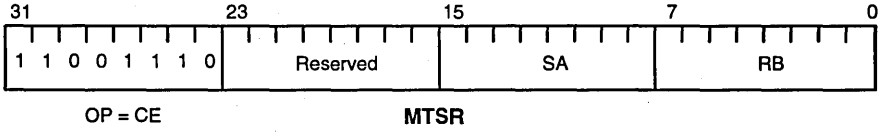
Operation: SPDEST ← SRCB

Assembler Syntax: MTSR spid, rb

Status: Not affected unless the destination is the ALU Status Register

Operands:

SRCB	Content of register RB
SPDEST	Special-purpose register SA



Description: The SRCB operand is placed into the SPECIAL location.

For programs in the User mode, a Protection Violation trap occurs if SA specifies a protected special-purpose register. If a trap occurs, the SPDEST location is not altered.

Move to Special Register Immediate

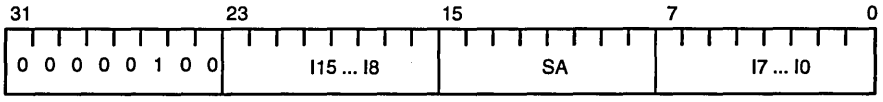
Operation: SPDEST ← 0I16

Assembler Syntax: MTSRIM spid, const16

Status: Not affected unless the destination is the ALU Status Register

Operands:

0I16	I15 ... I8 // I7 ... I0 (zero-extended to 32 bits)
SPDEST	Special-purpose register SA



OP = 04 **MTSRIM**

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

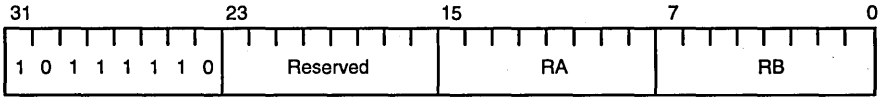
Operation: None

Assembler Syntax: MTTLB ra, rb

Status: Not affected

Operands:

SRCA	Content of register RA, bits 6...0
SRCB	Content of register RB



OP = BE MTTLB

Description: The SRCB operand is placed into the Translation Look-Aside Buffer (TLB) register whose register number is specified by the SRCA operand.

This instruction may be executed only by Supervisor-mode programs. An attempted execution by a User-mode program causes a Protection Violation trap to occur. If a trap occurs, the TLB register is not altered.

Multiply Step

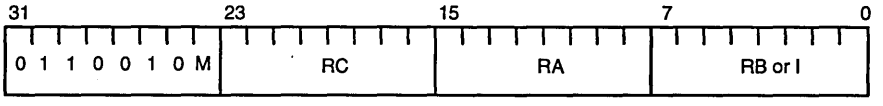
Operation: Perform one-bit step of a multiply operation

Assembler Syntax:
MUL rc, ra, rb
or
MUL rc, ra, const 8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 64, 65 **MUL**

Description: If the least significant bit of the Q Register is 1, the SRCA operand is added to the SRCB operand. If the least significant bit of the Q register is 0, a zero word is added to the SRCB operand.

The content of the Q Register is appended to the result of the add and the resulting 64-bit value is shifted right by one bit position; the true sign of the result of the add fills the vacated bit position (i.e., the sign of the result is complemented if an overflow occurred during the add operation). The high-order 32 bits of the 64-bit shifted value are placed into the DEST location. The low-order 32 bits of the shifted value are placed into the Q Register.

Examples of integer multiply operations appear in Section 2.6.3.

Multiply Last Step

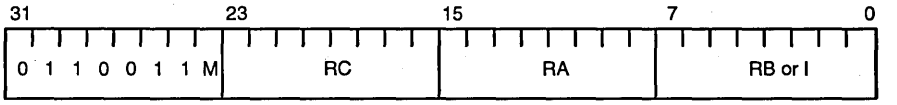
Operation: Complete a sequence of multiply steps (for signed multiply)

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 2.6.3.

Integer Multiply, Unsigned

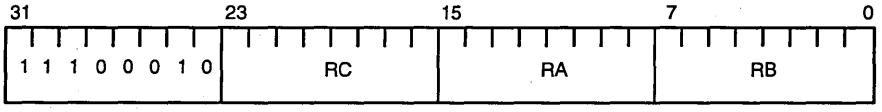
Operation: DEST ← SRCA * SRCB

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: In the Am29245 microcontroller, this instruction is not supported directly in processor hardware. Instead, the instruction causes a MULTIPLU trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Integer Multiply, Signed

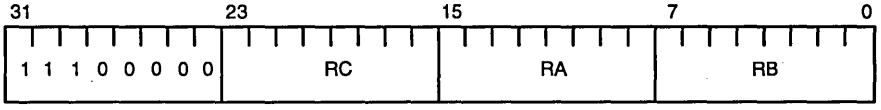
Operation: DEST ← SRCA * SRCB

Assembler Syntax: MULTIPLY rc, ra, rb

Status: None

Operands:

SRCA	Content of register RA
SRCB	Content of register RB
DEST	Register RC



OP = E0 **MULTIPLY**

Description: The SRCA operand is multiplied by the SRCB operand. The low-order 32 bits of the 64-bit result are placed into the DEST location. This operation treats the SRCA and SRCB operands as two's-complement integers and produces a two's-complement result.

The contents of the Q register are undefined after a MULTIPLY operation.

Note: In the Am29245 microcontroller, this instruction is not supported directly in processor hardware. Instead, the 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

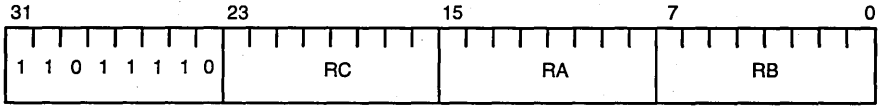
Operation: DEST ← SRCA * SRCB

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: In the Am29245 microcontroller, this instruction is not supported directly in processor hardware. Instead, the 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

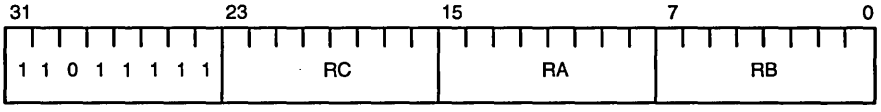
Operation: DEST ← SRCA * SRCB

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: In the Am29245 microcontroller, this instruction is not supported directly in processor hardware. Instead, the instruction causes a MULTMU trap. When the trap occurs, the IPA, IPB, and IPC registers are set to reference SRCA, SRCB, and DEST.

Multiply Step, Unsigned

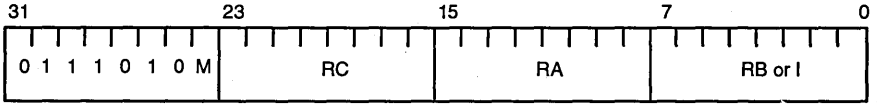
Operation: Perform one-bit step of a multiply operation (unsigned)

Assembler Syntax:
MULU rc, ra, rb
or
MULU rc, ra, const 8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 74, 75 **MULU**

Description: If the least significant bit of the Q Register is 1, the SRCA operand is added to the SRCB operand. If the least significant bit of the Q register is 0, a zero word is added to the SRCB operand.

The content of the Q register is appended to the result of the add and the resulting 64-bit value is shifted right by one bit position; the carry-out of the add fills the vacated bit position. The high-order 32 bits of the 64-bit shifted value are placed into the DEST location. The low-order 32 bits of the shifted value are placed into the Q Register.

Examples of integer multiply operations appear in Section 2.6.3.

NAND Logical

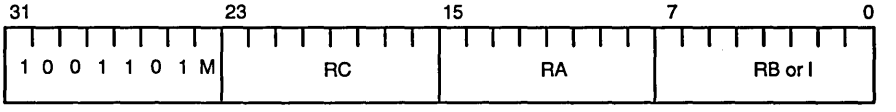
Operation: DEST ← ~(SRCA & SRCB)

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

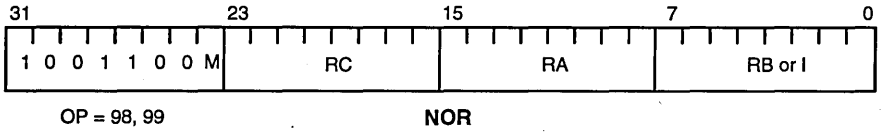
Operation: DEST ← ~(SRCA | SRCB)

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



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

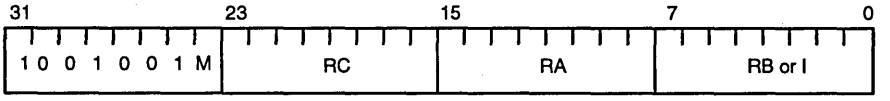
Operation: DEST ← SRCA | SRCB

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.

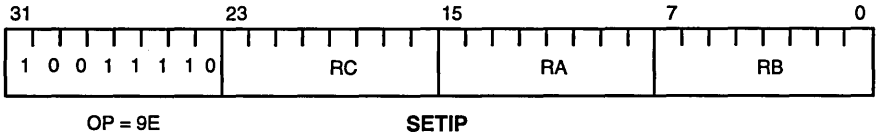
Set Indirect Pointers

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

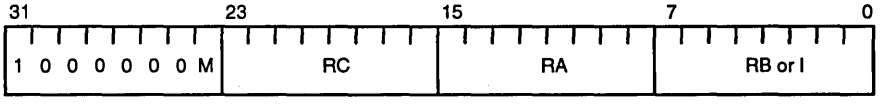
Operation: DEST ← SRCA << SRCB (zero fill)

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

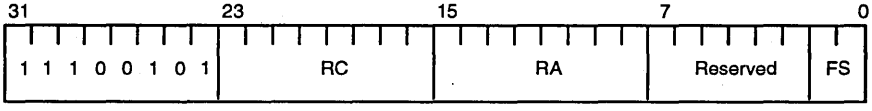
Operands:

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 the 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

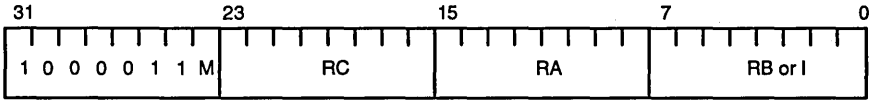
Operation: DEST ← SRCA >> SRCB (sign fill)

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

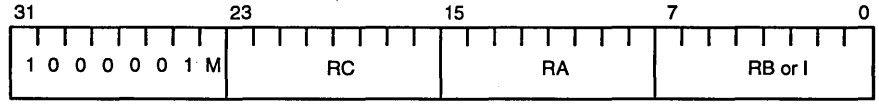
Operation: DEST ← SRCA >> SRCB (zero fill)

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

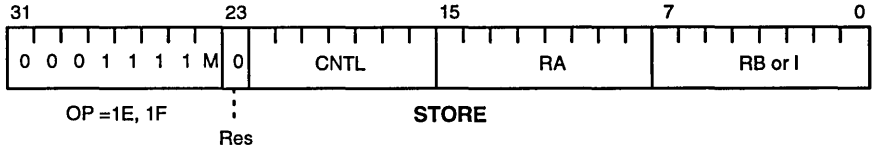
Operation: EXTERNAL WORD [SRCB] ← SRCA

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 access as described in Section 3.3.1.

Store and Lock

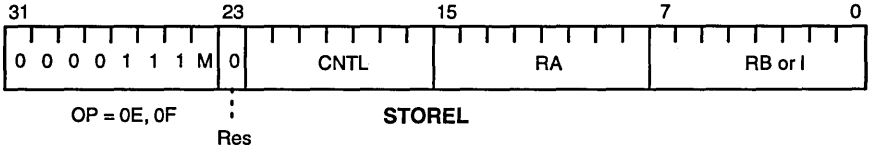
Operation: EXTERNAL WORD [SRCB] ← SRCA

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 access as described in Section 3.3.1.

The store is treated as a noncacheable access that stores only in the external memory. The write buffer is emptied before the store is performed, and, if the associated block is found in the data cache, the block is invalidated.

Store Multiple

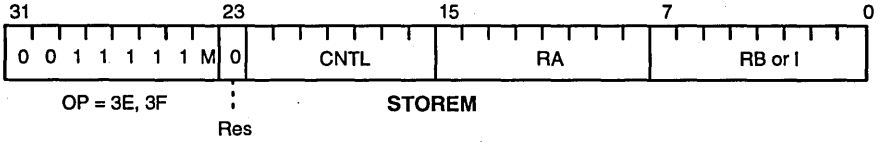
Operation: EXTERNAL WORD [SRCB] ... EXTERNAL WORD [SRCB + (COUNT * 4)]
← SRCA ... SRCA+COUNT

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 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

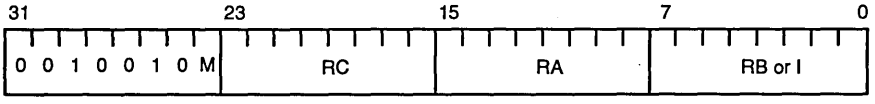
Operation: DEST ← SRCA – SRCB

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



OP = 24, 25 **SUB**

Description: The SRCA operand is added to the two's-complement of the SRCB operand and the result is placed into the DEST location.

Subtract with Carry

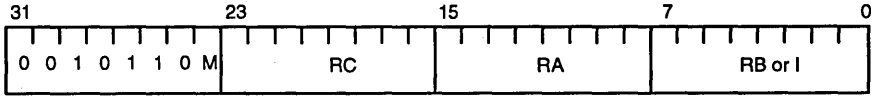
Operation: DEST ← SRCA – SRCB – 1 + C

Assembler Syntax:
SUBC rc, ra, rb
or
SUBC rc, ra, const8

Status: V, N, Z, C

Operands:

SRCA	Content of register RA
SRCB	M = 0: Content of register RB M = 1: I (Zero-extended to 32 bits)
DEST	Register RC



OP = 2C, 2D **SUBC**

Description: The SRCA operand is added to the one's-complement of the SRCB operand and the value of the ALU Status Carry bit. The result is placed into the DEST location.

Subtract with Carry, Signed

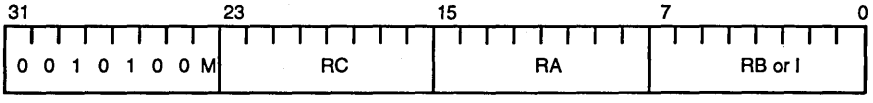
Operation: DEST ← SRCA – SRCB – 1 + C
IF signed overflow THEN Trap (Out of Range)

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. 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

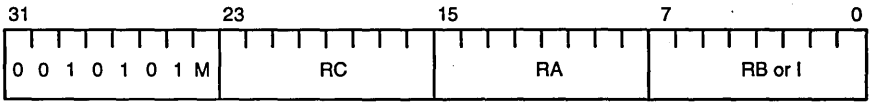
Operation: DEST ← SRCA – SRCB – 1 + C
IF unsigned underflow THEN Trap (Out of Range)

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. 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

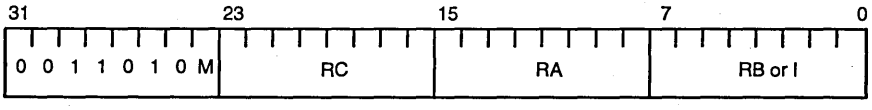
Operation: DEST ← SRCB – SRCA

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

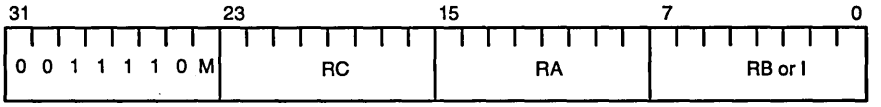
Operation: DEST ← SRCB – SRCA – 1 + C

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. The result is placed into the DEST location.

Subtract Reverse with Carry, Signed

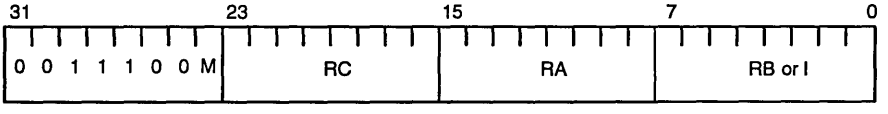
Operation: DEST ← SRCB – SRCA – 1 + C
IF signed overflow THEN Trap (Out of Range)

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. 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

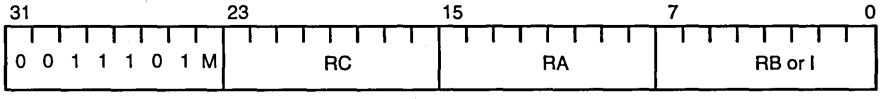
Operation: DEST ← SRCB – SRCA – 1 + C
IF unsigned underflow THEN Trap (Out of Range)

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. 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

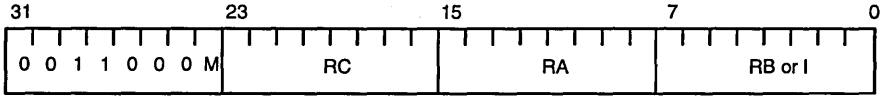
Operation: DEST ← SRCB – SRCA
IF signed overflow THEN Trap (Out of Range)

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

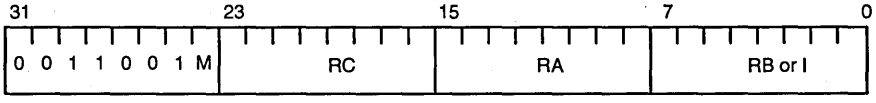
Operation: DEST ← SRCB – SRCA
IF unsigned underflow THEN Trap (Out of Range)

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

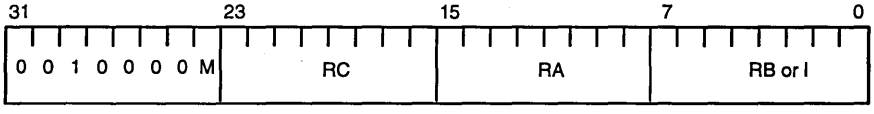
Operation: DEST ← SRCA – SRCB
IF signed overflow THEN Trap (Out of Range)

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

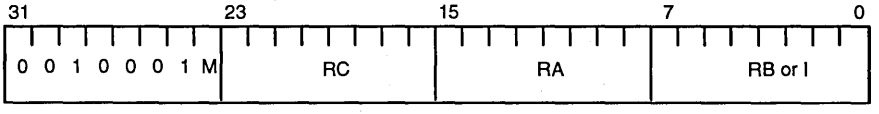
Operation: DEST ← SRCA – SRCB
IF unsigned underflow THEN Trap (Out of Range)

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

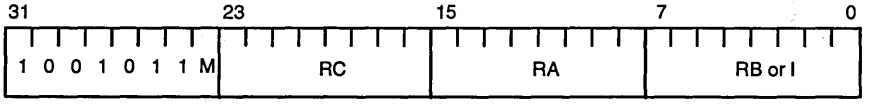
Operation: DEST ← ~ (SRCA ^ SRCB)

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

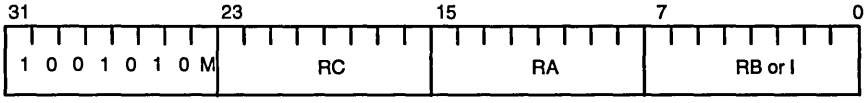
Operation: DEST ← SRCA ^ SRCB

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.

21.4 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 SPECIAL SETTINGS FOR THE Am29240, Am29245, AND Am29243 MICROCONTROLLERS

Am29240 MICROCONTROLLER

Before using the Am29240 microcontroller product, the user should prepare the microcontroller by setting the following field as shown.

- In the DRAM Control Register, set the PCE field to 0.

Am29245 MICROCONTROLLER

Before using the Am29245 microcontroller product, the user should prepare the microcontroller by setting the following fields and signals as shown.

- In the Configuration Register, set the TBO field to 0.
- In the Configuration Register, set the DD field to 1.
- In the DRAM Control Register, set the PCE field to 0.
- Set the RXDB signal to ground or Vcc.

Am29243 MICROCONTROLLER

Before using the Am29243 microcontroller product, the user should prepare the microcontroller by setting the following signals as shown.

- Set the LSYNC and VCLK signals to ground or Vcc.

Appendix

B PROCESSOR REGISTER SUMMARY

Figure B-1 General-Purpose Register Organization

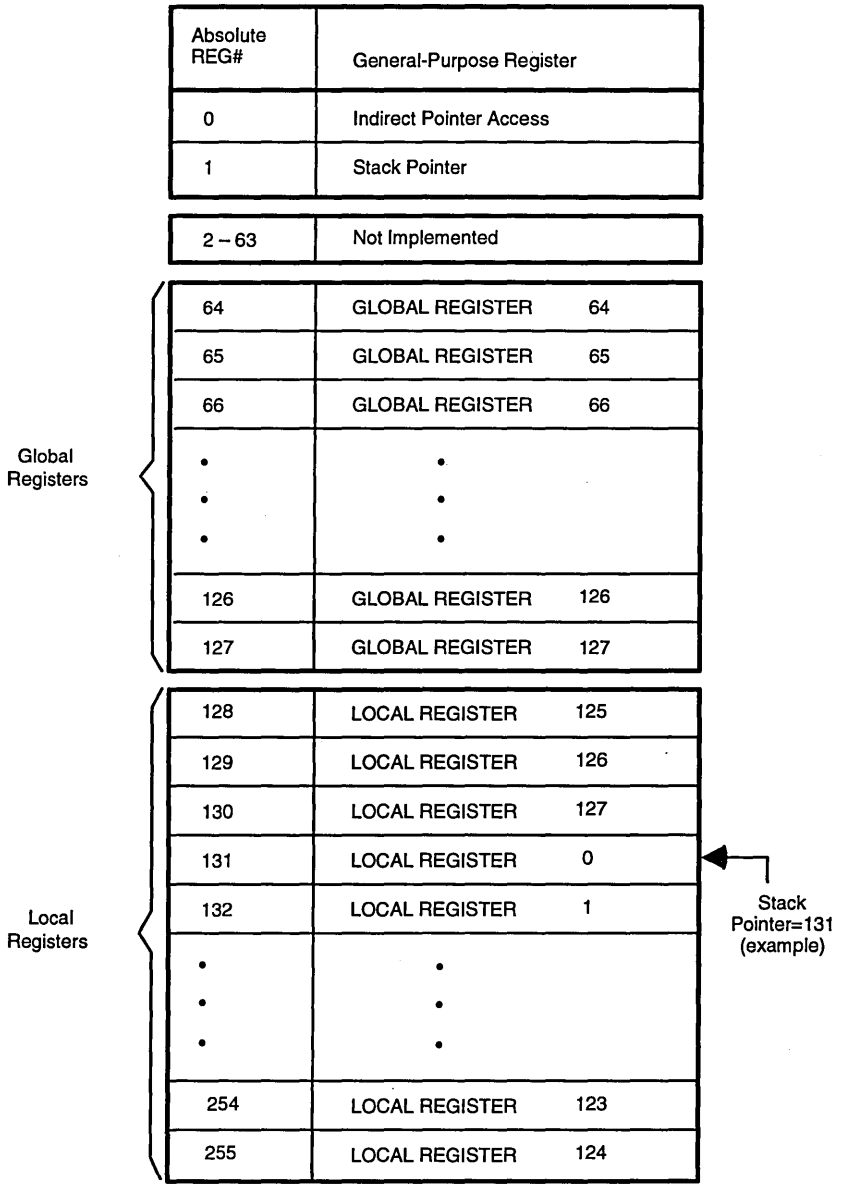


Figure B-2 Register Bank Organization

Register Bank Protect Register Bit	Absolute-Register Numbers	General-Purpose Registers
0	2 through 15	Bank 0 (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 B-3 Special Purpose Registers

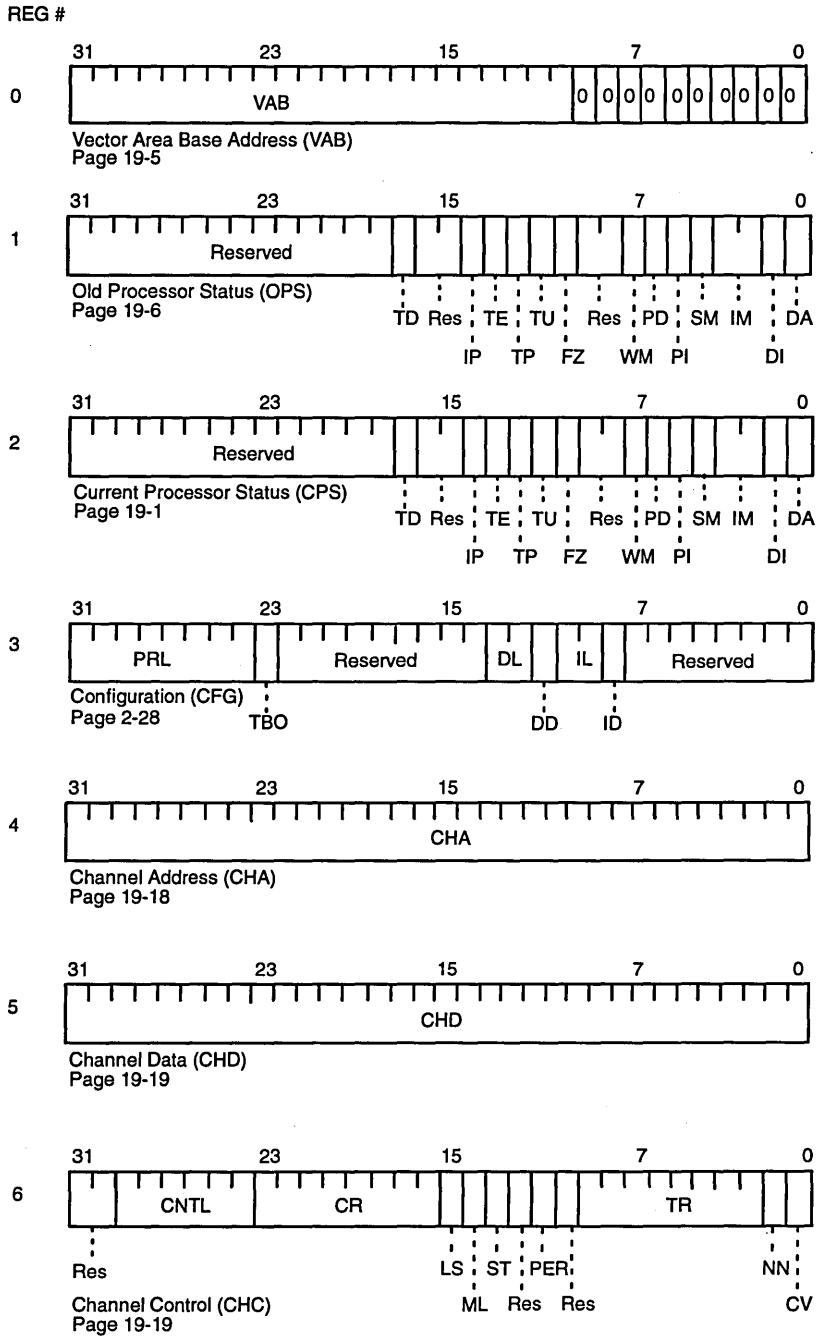
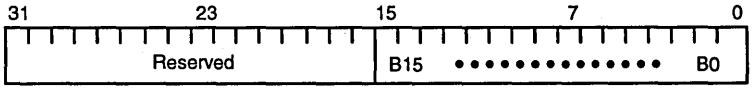


Figure B-3 Special Purpose Registers (continued)

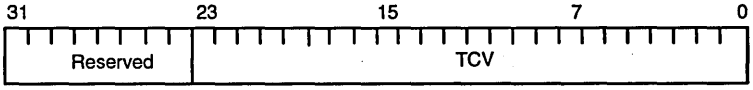
REG #

7



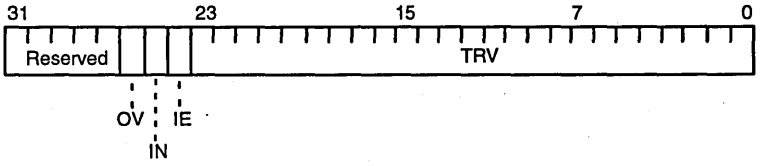
Register Bank Protect (RBP)
Page 6-3

8



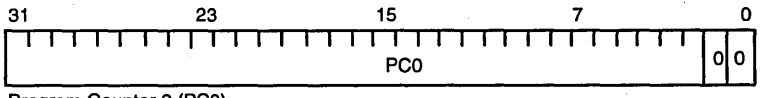
Timer Counter (TMC)
Page 19-23

9



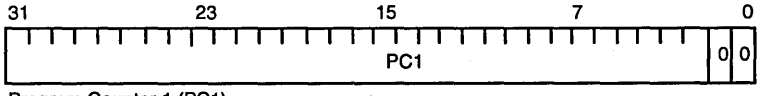
Timer Reload (TMR)
Page 19-24

10



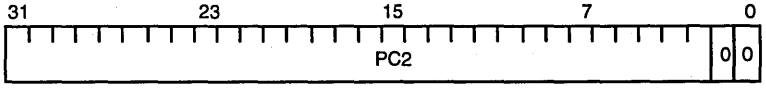
Program Counter 0 (PC0)
Page 19-8

11



Program Counter 1 (PC1)
Page 19-9

12



Program Counter 2 (PC2)
Page 19-10

Figure B-3 Special Purpose Registers (continued)

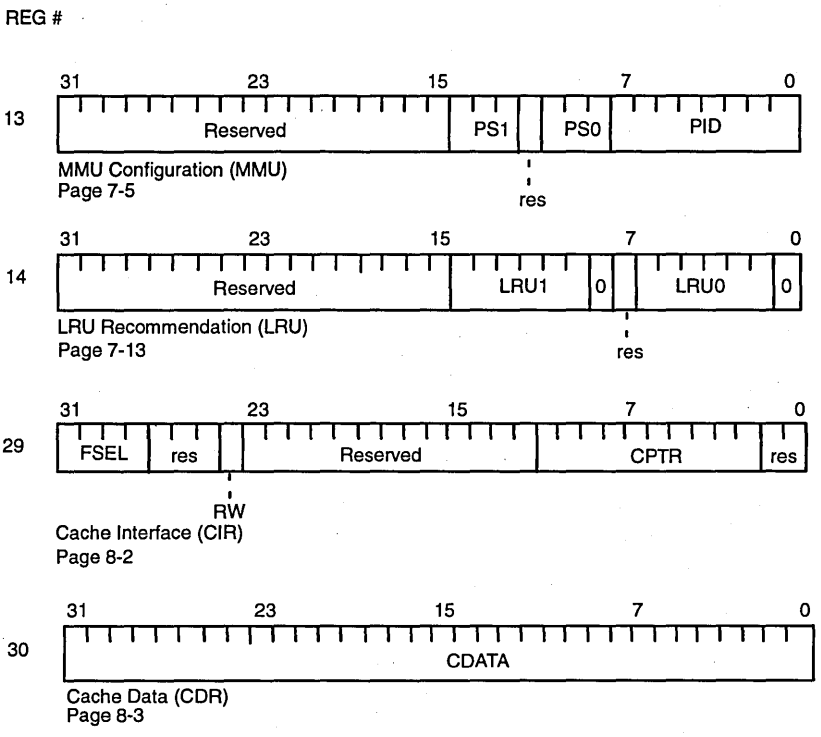


Figure B-3 Special Purpose Registers (continued)

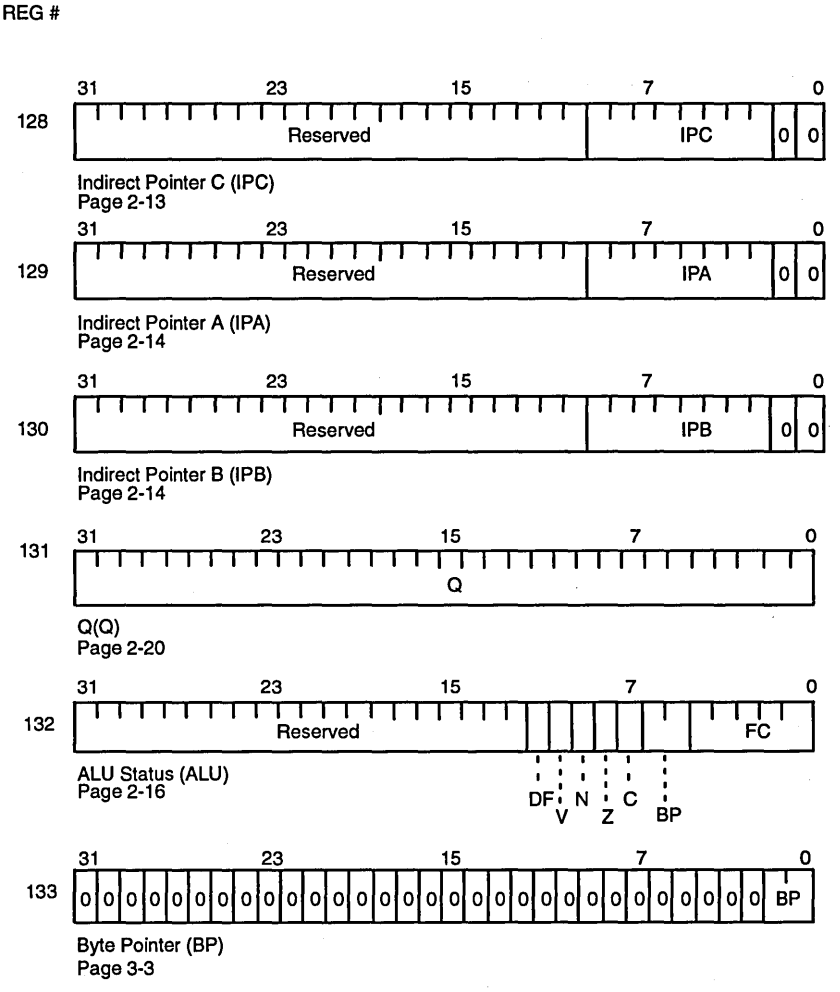


Figure B-3 Special Purpose Registers (continued)

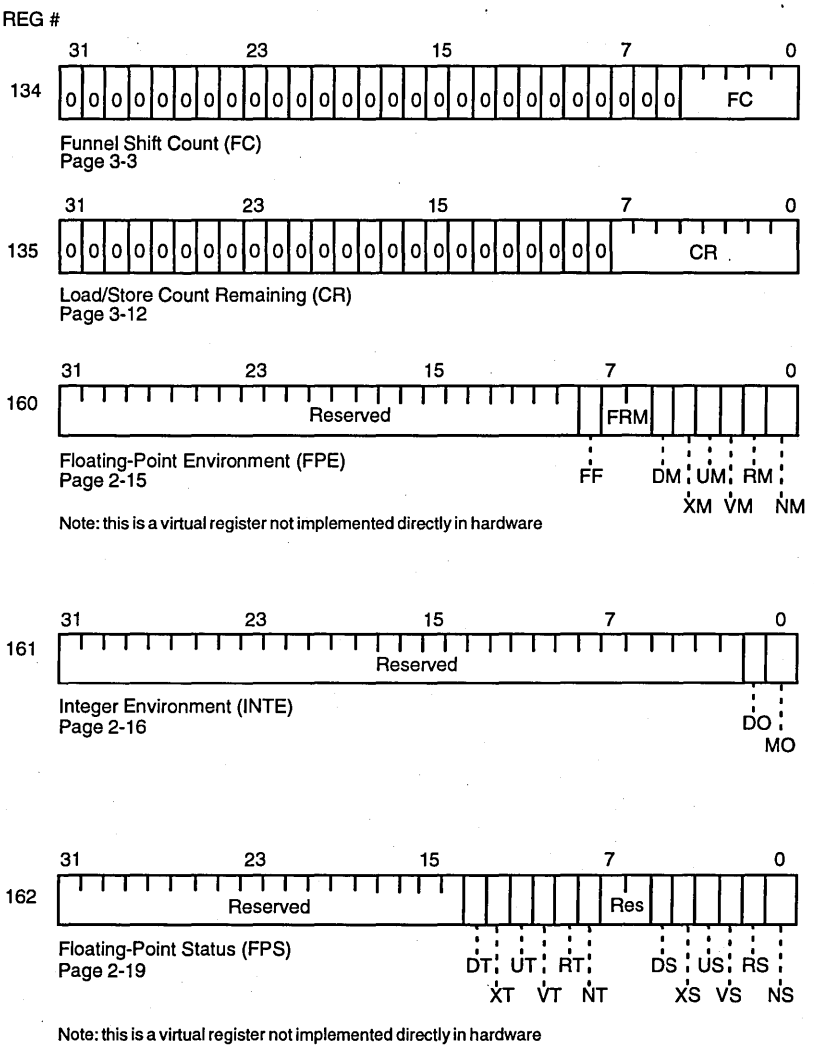


Figure B-4 Translation Look-Aside Buffer Entries

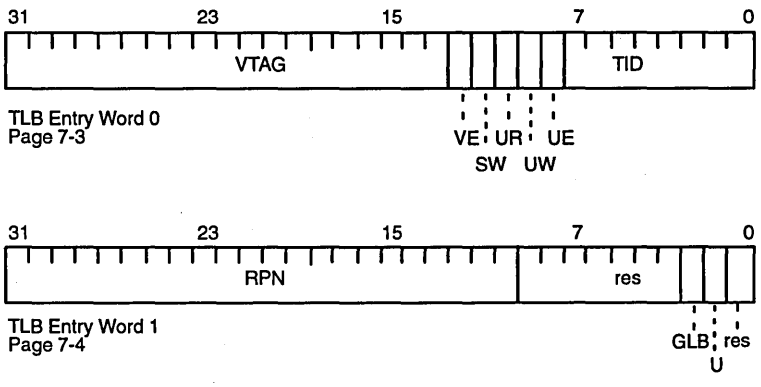


Table B-1 Processor Register Field Summary

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
CDATA	Cache Data	Cache Data	31–0
CHA	Channel Address	Channel Address	31–0
CHD	Channel Data	Channel Data	31–0
CNTL	Control	Channel Control	30–24
CPTR	Cache Pointer	Cache Interface	11–2
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
DD	Data Cache Disable	Configuration	11
DF	Divide Flag	ALU Status	11
DI	Disable Interrupts	Current Processor Status Old Processor Status	1 1
DL	Data Cache Lock	Configuration	13–12
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
FC	Funnel Shift Count	ALU Status Funnel Shift Count	4–0 4–0
FF	Fast Floating-Point Select	Floating-Point Environment	8
FRM	Floating-Point Round Mode	Floating-Point Environment	7–6
FSEL	Cache Field Select	Cache Interface	31–28

Table B-1 Processor Register Field Summary (continued)

FZ	Freeze	Current Processor Status Old Processor Status	10 10
GLB	Global Page	TLB Entry Word 1	2
ID	Instruction Cache Disable	Configuration	8
IE	Interrupt Enable	Timer Reload	24
IL	Instruction Cache Lock	Configuration	10–9
IM	Interrupt Mask	Current Processor Status Old 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
LA	Lock Active	Channel Control	12
LRU0	Least Recently Used Entry, TLB0	LRU Recommendation	6–1
LRU1	Least Recently Used Entry, TLB1	LRU Recommendation	14–9
LS	Load/Store	Channel Control	15
ML	Multiple Operation	Channel Control	14
MO	Integer Multiplication Overflow Exception Mask	Integer Environment	0
N	Negative	ALU Status	9
NM	Floating-Point Invalid Operation Mask	Floating-Point Environment	0
NN	Not Needed	Channel Control	1
NS	Floating-Point Invalid Operation Sticky	Floating-Point Status	0
NT	Floating-Point Invalid Operation Trap	Floating-Point Status	8
OV	Overflow	Timer Reload	26
PC0	Program Counter 0	Program Counter 0	31–2
PC1	Program Counter 1	Program Counter 1	31–2
PC2	Program Counter 2	Program Counter 2	31–2
PD	Physical Addressing/Data	Current Processor Status Old Processor Status	6 6
PER	Parity Error	Channel Control	11
PI	Physical Addressing/Instructions	Current Processor Status Old Processor Status	5 5
PID	Process Identifier	MMU Configuration	7–0
PRL	Processor Release Level	Configuration	31–24
PS0	Page Size, TLB0	MMU Configuration	10–8
PS1	Page Size, TLB1	MMU Configuration	14–12
Q	Quotient/Multiplier	Q Register	31–0
RM	Floating-Point Reserved Operand Mask	Floating-Point Environment	1
RPN	Real Page Number	TLB Entry Word 1	31–10
RS	Floating-Point Reserved Operand Sticky	Floating-Point Status	1

Table B-1 Processor Register Field Summary (continued)

RT	Floating-Point Reserved Operand Trap	Floating-Point Status	9
RW	Read/Write	Cache Interface	24
SM	Supervisor Mode	Current Processor Status Old Processor Status	4 4
ST	Set	Channel Control	13
SW	Supervisor Write	TLB Entry Word 0	11
TBO	Turbo Mode	Configuration	23
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
TID	Task Identifier	TLB Entry Word 0	7–0
TP	Trace Pending	Current Processor Status Old Processor Status	12 12
TR	Target Register	Channel Control	9–2
TRV	Timer Reload Value	Timer Reload	23–0
TU	Trap Unaligned Access	Current Processor Status Old Processor Status	11 11
U	Usage	TLB Entry Word 1	1
UE	User Execute	TLB Entry Word 0	8
UM	Floating-Point Underflow Mask	Floating-Point Environment	3
UR	User Read	TLB Entry Word 0	10
US	Floating-Point Underflow Sticky	Floating-Point Status	3
UT	Floating-Point Underflow Trap	Floating-Point Status	11
UW	User Write	TLB Entry Word 0	9
V	Overflow	ALU Status	10
VAB	Vector Area Base	Vector Area Base Address	31–10
VE	Valid Entry	TLB Entry Word 0	12
VM	Floating-Point Overflow Mask	Floating-Point Environment	2
VS	Floating-Point Overflow Sticky	Floating-Point Status	2
VT	Floating-Point Overflow Trap	Floating-Point Status	10
VTAG	Virtual Tag	TLB Entry Word 0	31–13
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

C PERIPHERAL REGISTER SUMMARY

Figure C-1 On-Chip Peripheral Registers

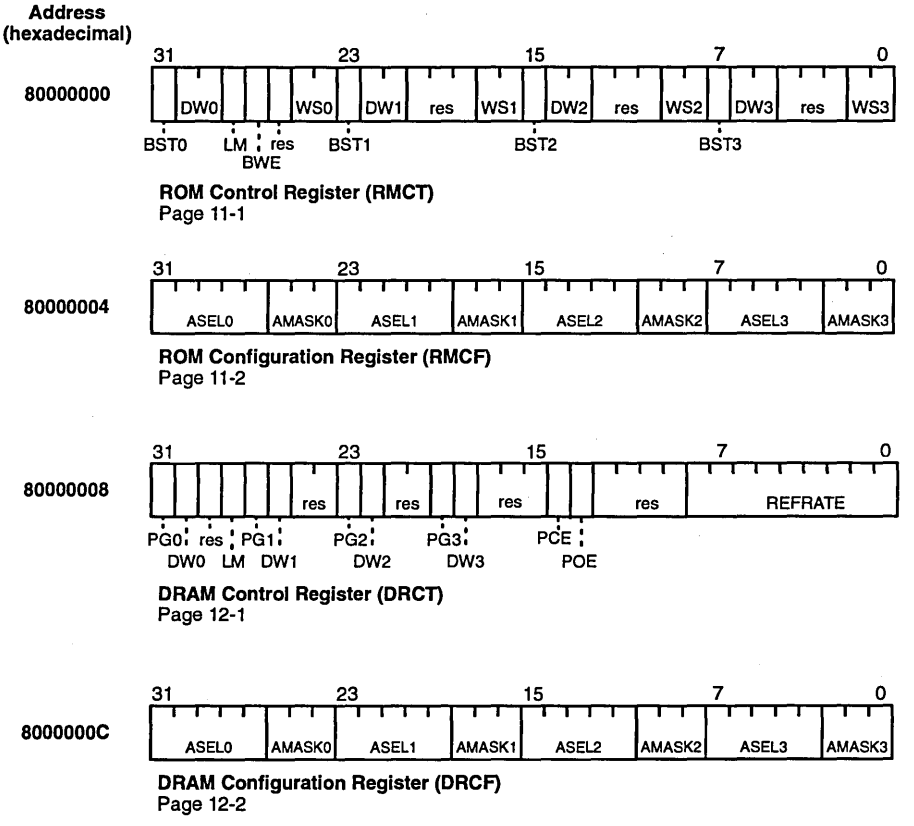


Figure C-1 On-Chip Peripheral Registers (continued)

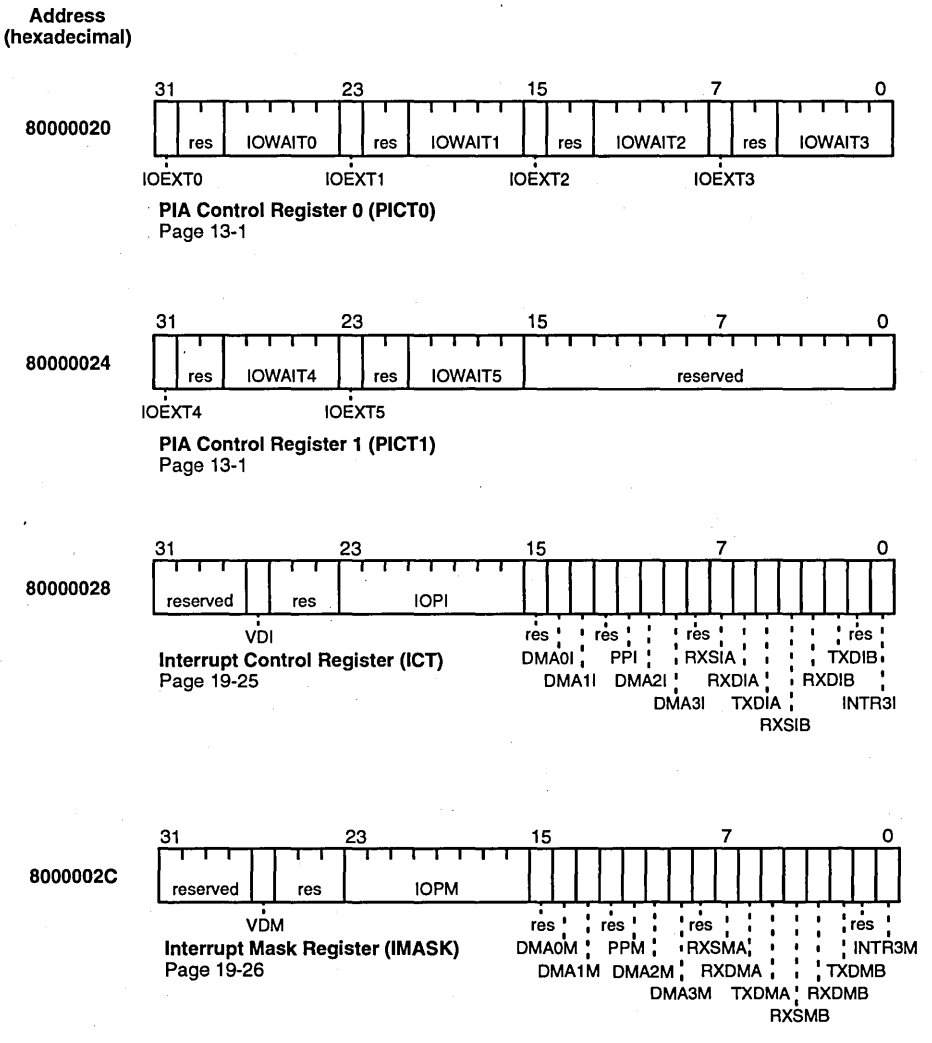


Figure C-1 On-Chip Peripheral Registers (continued)

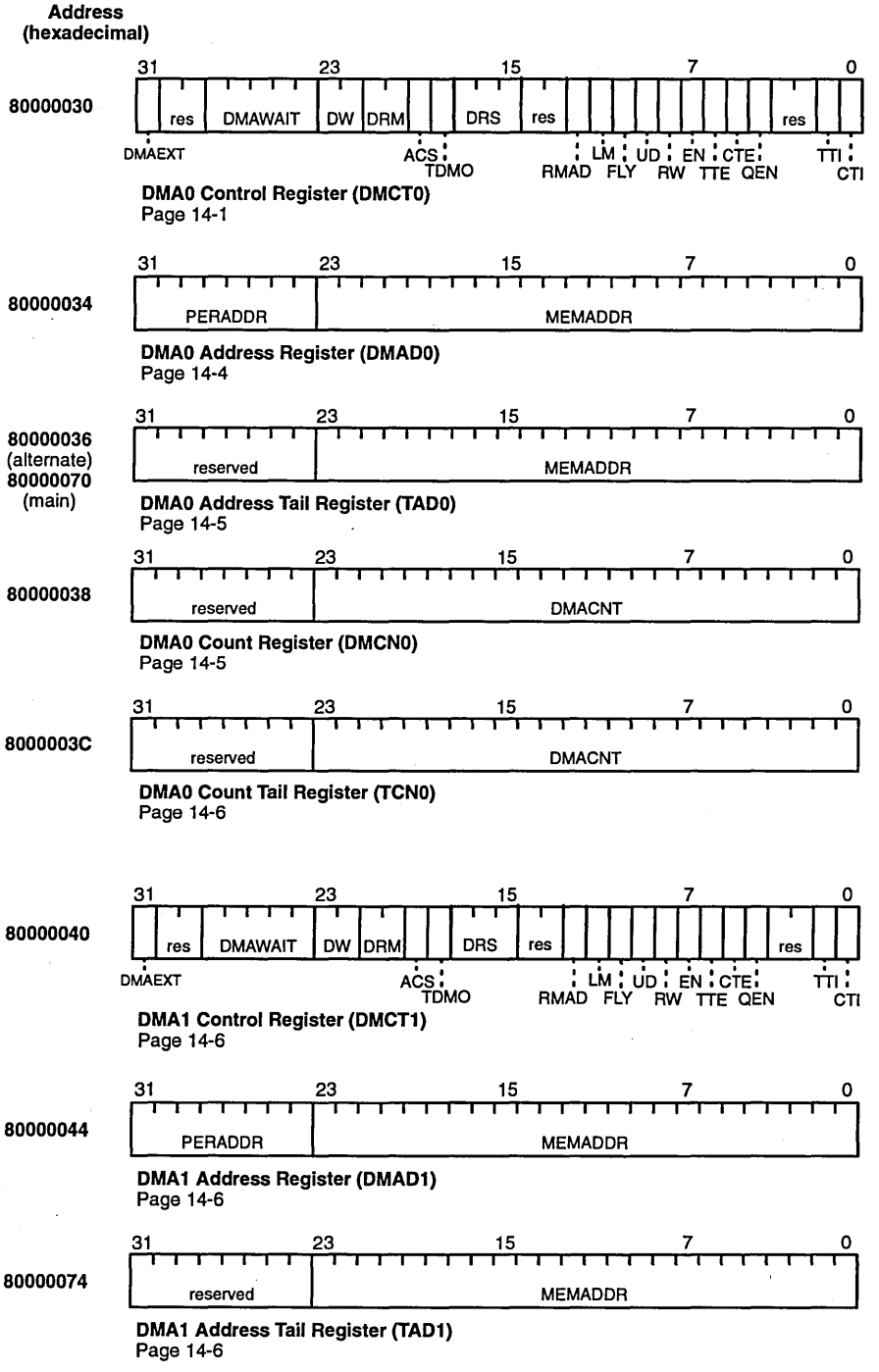


Figure C-1 On-Chip Peripheral Registers (continued)

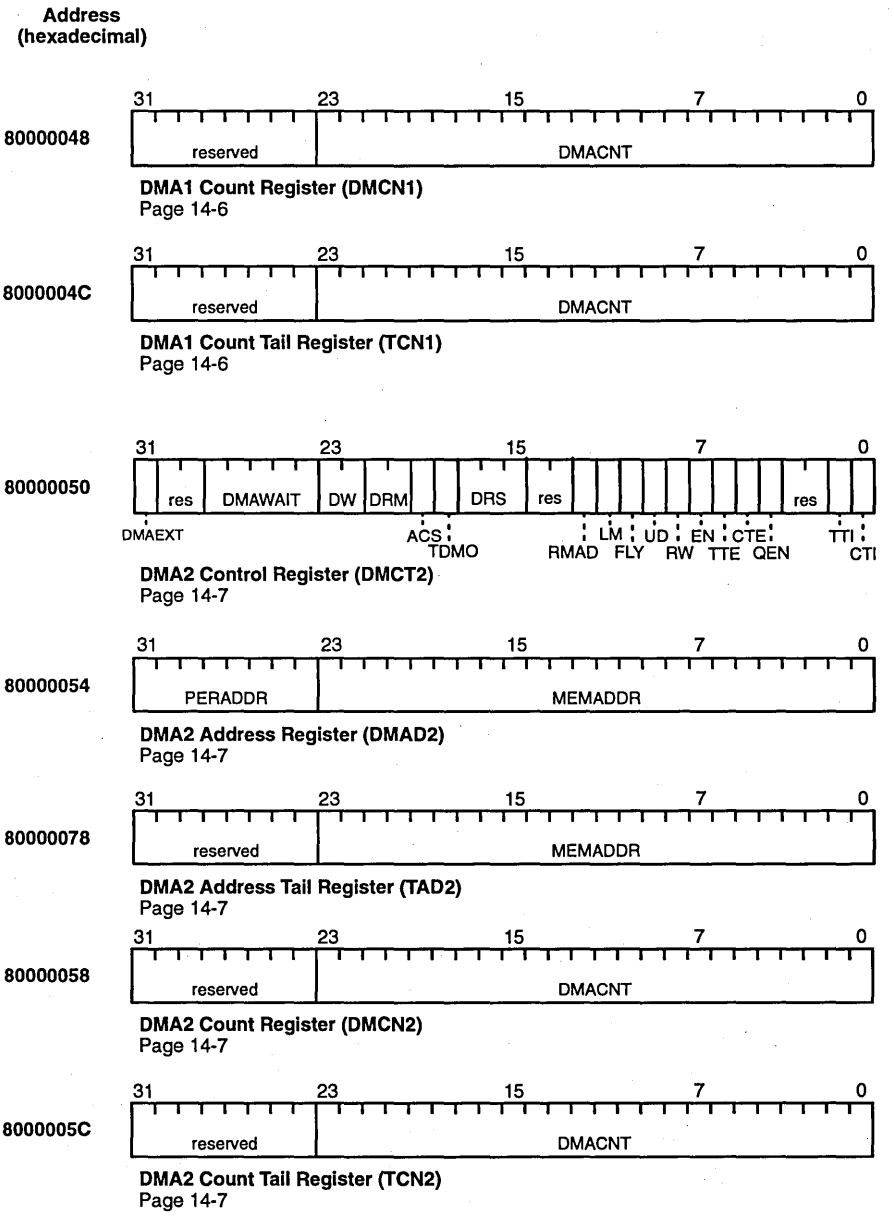
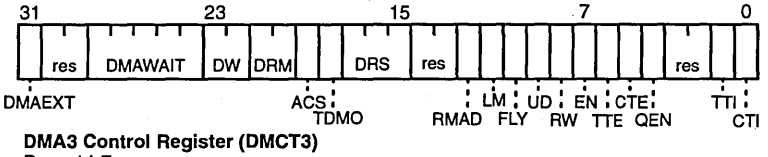


Figure C-1 On-Chip Peripheral Registers (continued)

Address (hexadecimal)

80000060

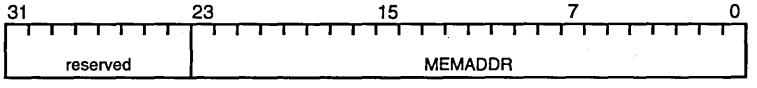


DMA3 Control Register (DMCT3)
Page 14-7

80000064

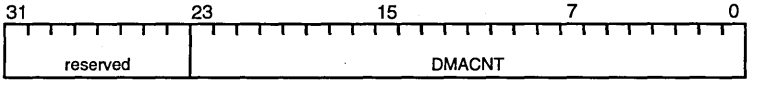
DMA3 Address Register (DMAD3)
Page 14-7

8000007C



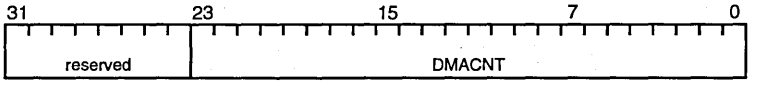
DMA3 Address Tail Register (TAD3)
Page 14-7

80000068



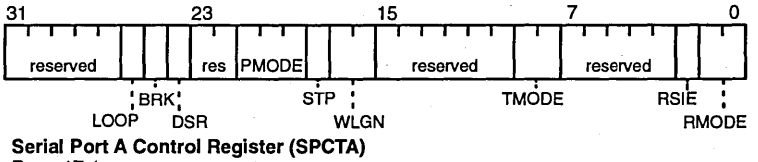
DMA3 Count Register (DMCN3)
Page 14-7

8000006C



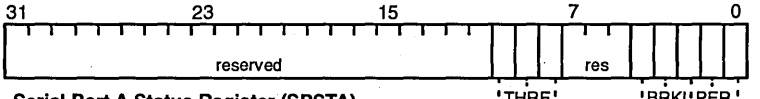
DMA3 Count Tail Register (TCN3)
Page 14-7

80000080



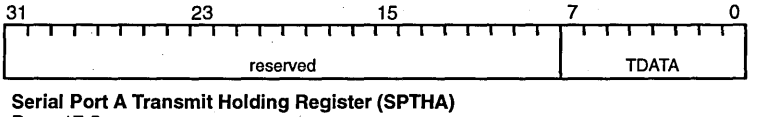
Serial Port A Control Register (SPCTA)
Page 17-1

80000084



Serial Port A Status Register (SPSTA)
Page 17-4

80000088



Serial Port A Transmit Holding Register (SPTHA)
Page 17-5

Figure C-1 On-Chip Peripheral Registers (continued)

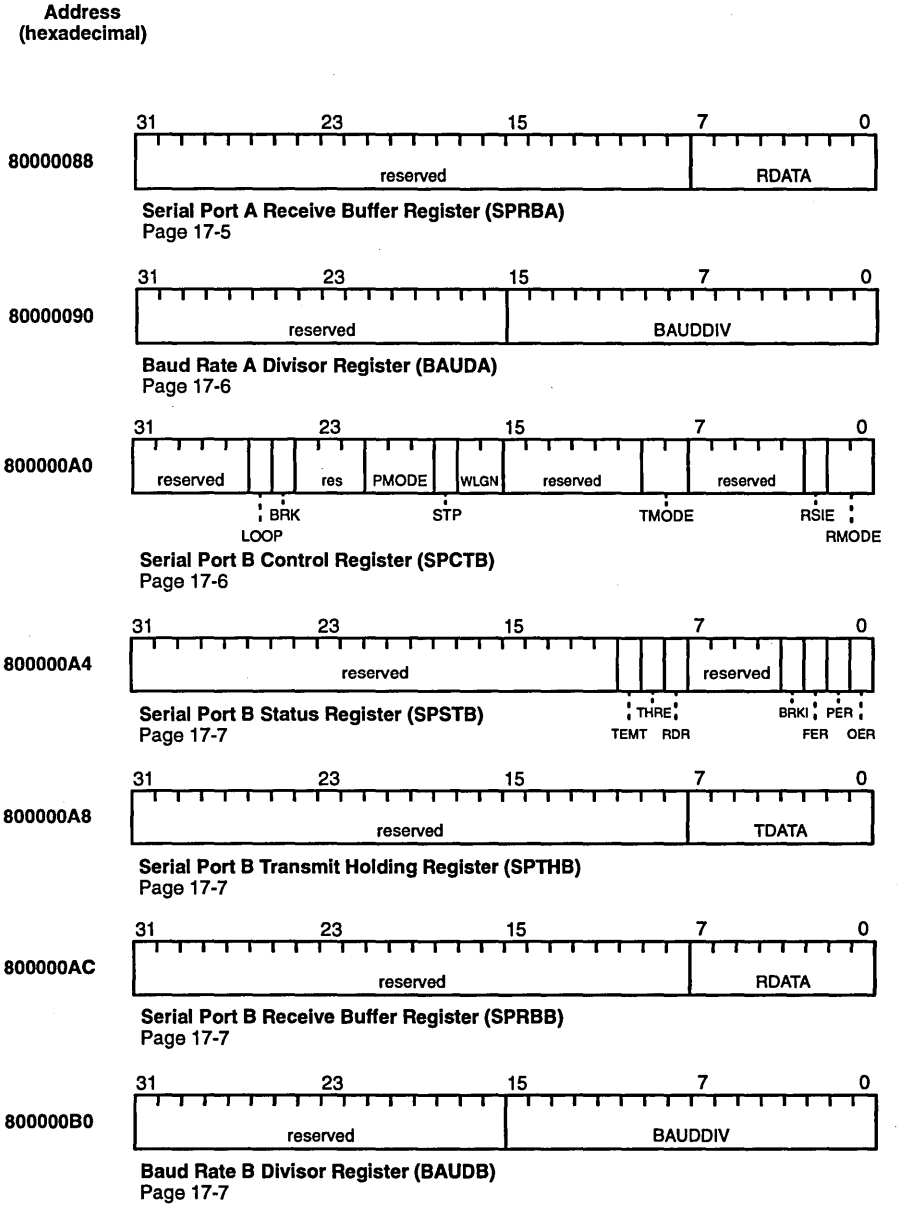


Figure C-1 On-Chip Peripheral Registers (continued)

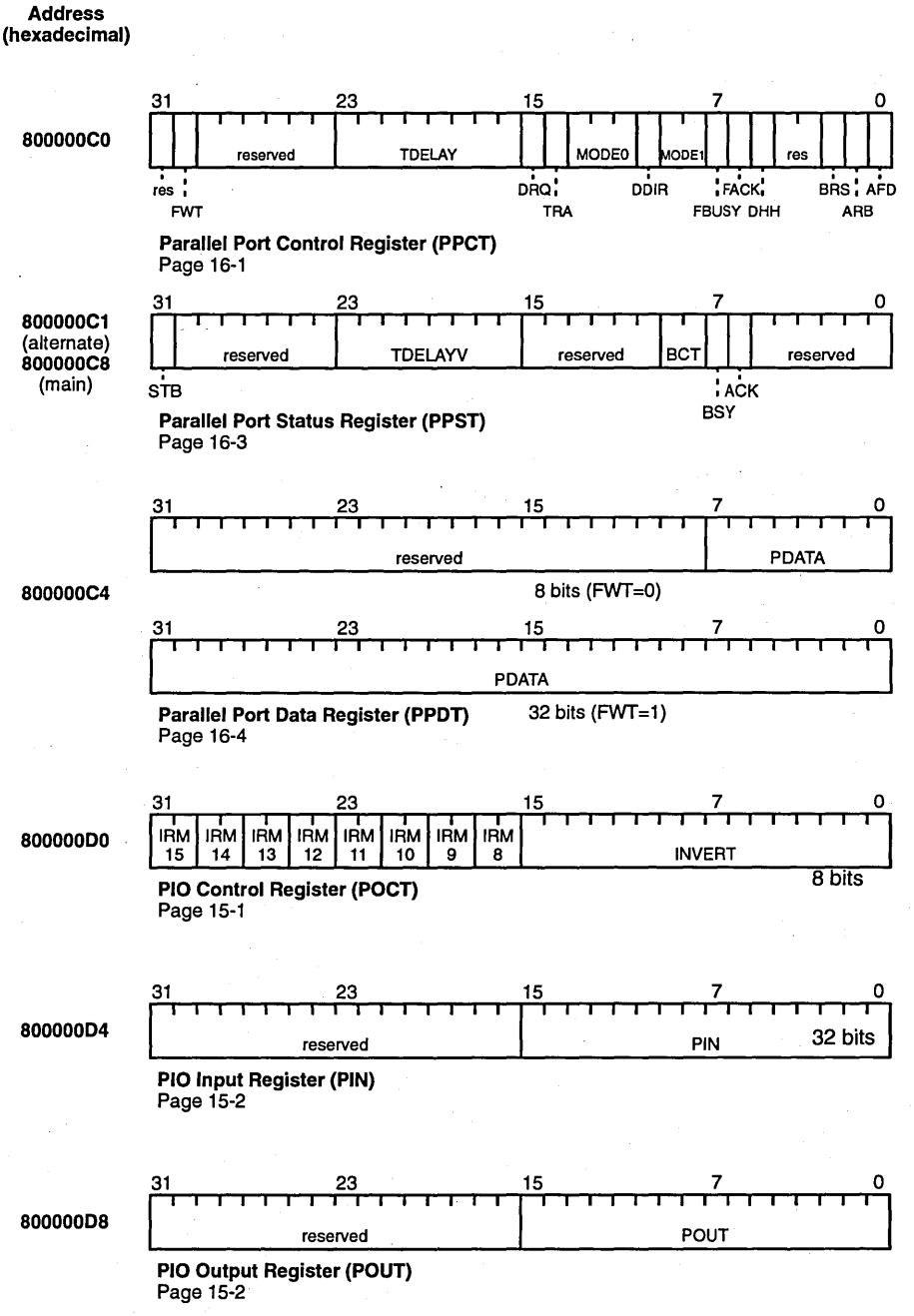


Figure C-1 On-Chip Peripheral Registers (continued)

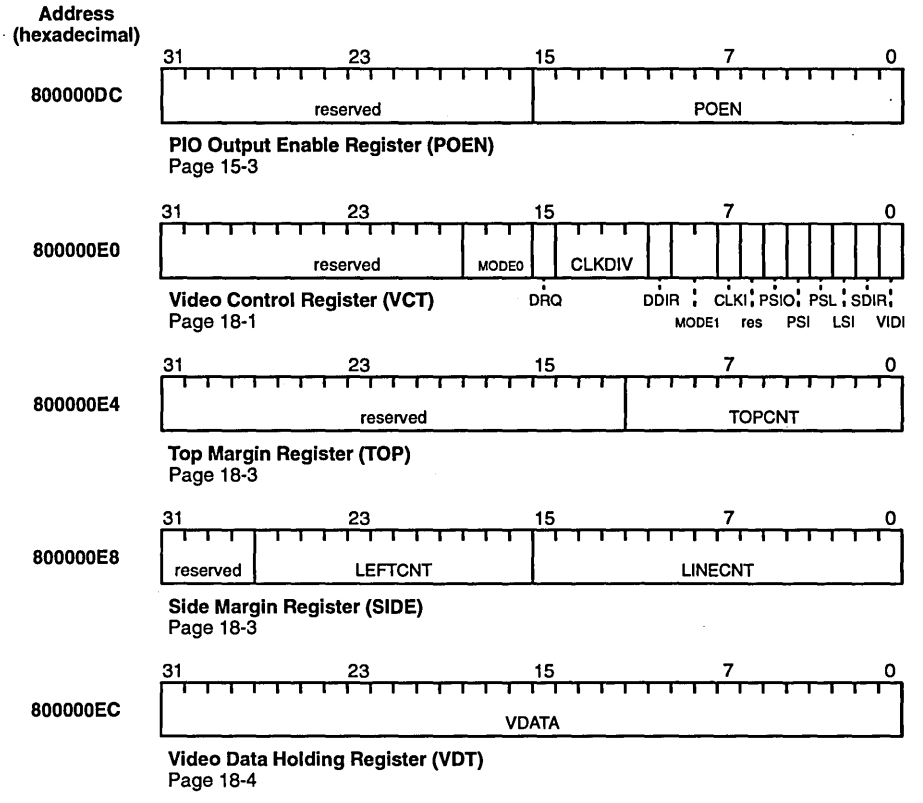


Table C-1 Peripheral Register Field Summary

Label	Field Name	Register	Bit
ACK	PACK Level	Parallel Port Status	6
ACS	Assert Chip Select	DMA0 Control	19
		DMA1 Control	19
		DMA2 Control	19
		DMA3 Control	19
AFD	Autofeed	Parallel Port Control	0
AMASK0	Address Mask, Bank 0	ROM Configuration	26–24
		DRAM Configuration	26–24
AMASK1	Address Mask, Bank 1	ROM Configuration	18–16
AMASK2		DRAM Configuration	18–16
AMASK1	Address Mask, Bank 2	ROM Configuration	10–8
AMASK2		DRAM Configuration	10–8
AMASK3	Address Mask, Bank 3	ROM Configuration	2–0
		DRAM Configuration	2–0
ARB	ACK Relationship to BUSY	Parallel Port Control	1
ASEL0	Address Select, Bank 0	ROM Configuration	31–27
		DRAM Configuration	31–27
ASEL1	Address Select, Bank 1	ROM Configuration	23–19
		DRAM Configuration	23–19
ASEL2	Address Select, Bank 2	ROM Configuration	15–11
		DRAM Configuration	15–11
ASEL3	Address Select, Bank 3	ROM Configuration	7–3
		DRAM Configuration	7–3
BAUDDIV	Baud Rate Divisor	Baud Rate A Divisor	15–0
		Baud Rate B Divisor	15–0
BCT	Byte Count	Parallel Port Status	9–8
BRK	Send Break	Serial Port A Control	25
		Serial Port B Control	25
BRKI	Break Interrupt	Serial Port A Status	3
		Serial Port B 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
BWE	Byte Write Enable	ROM Control	27
CLKDIV	Clock Divide	Video Control	14–11
CLKI	Clock Invert	Video Control	7
CTE	Count Terminate Enable	DMA0 Control	5
		DMA1 Control	5
		DMA2 Control	5
		DMA3 Control	5
CTI	Count Terminate Interrupt	DMA0 Control	0
		DMA1 Control	0
		DMA2 Control	0
		DMA3 Control	0

Table C-1 Peripheral Register Field Summary (continued)

Label	Field Name	Register	Bit
DDIR	Data Direction	Parallel Port Control	10
		Video Control	10
DHH	Disable Hardware Handshake	Parallel Port Control	5
DMA0I	DMA Channel 0 Interrupt	Interrupt Control	14
DMA0M	DMA Channel 0 Mask	Interrupt Mask	14
DMA1I	DMA Channel 1 Interrupt	Interrupt Control	13
DMA1M	DMA Channel 1 Mask	Interrupt Mask	13
DMA2I	DMA Channel 2 Interrupt	Interrupt Control	10
DMA2M	DMA Channel 2 Mask	Interrupt Mask	10
DMA3I	DMA Channel 3 Interrupt	Interrupt Control	9
DMA3M	DMA Channel 3 Mask	Interrupt Mask	9
DMACNT	DMA Count	DMA0 Count	23–0
		DMA0 Count Tail	23–0
		DMA1 Count	23–0
		DMA1 Count Tail	23–0
		DMA2 Count	23–0
		DMA2 Count Tail	23–0
		DMA3 Count	23–0
		DMA3 Count Tail	23–0
DMAEXT	DMA Extend	DMA0 Control	31
		DMA1 Control	31
		DMA2 Control	31
		DMA3 Control	31
DMAWAIT	DMA Wait States	DMA0 Control	28–24
		DMA1 Control	28–24
		DMA2 Control	28–24
		DMA3 Control	28–24
DRM	DMA Request Mode	DMA0 Control	21–20
		DMA1 Control	21–20
		DMA2 Control	21–20
		DMA3 Control	21–20
DRQ	Data Request	Parallel Port Control	15
		Video Control	15
DRS	DMA Request Select	DMA0 Control	17–15
		DMA1 Control	17–15
		DMA2 Control	17–15
		DMA3 Control	17–15
DSR	Data Set Ready	Serial Port A Control	24
		Serial Port B Control	24
DTR	Data Terminal Ready	Serial Port A Status	4
		Serial Port B Status	4
DW	Data Width	DMA0 Control	22–23
		DMA1 Control	22–23
		DMA2 Control	22–23
		DMA3 Control	22–23
DW0	Data Width, Bank 0	ROM Control	30–29
		DRAM Control	30
DW1	Data Width, Bank 1	ROM Control	22–21
		DRAM Control	26
DW2	Data Width, Bank 2	ROM Control	14–13
		DRAM Control	22
DW3	Data Width, Bank 3	ROM Control	6–5
		DRAM Control	18

Table C-1 Peripheral Register Field Summary (continued)

EN	Enable	DMA0 Control DMA1 Control DMA2 Control DMA3 Control	7 7 7 7
FACK	Force ACK	Parallel Port Control	6
FBUSY	Force Busy	Parallel Port Control	7
FER	Framing Error	Serial Port A Status Serial Port B Status	2 2
FLY	Fly-By Transfers	DMA0 Control DMA1 Control DMA2 Control DMA3 Control	10 10 10 10
FWT	Full Word Transfer	Parallel Port Control	30
INTR3I	$\overline{\text{INTR}}$3 Interrupt	Interrupt Control	0
INTR3M	$\overline{\text{INTR}}$3 Mask	Interrupt Mask	0
INVERT	PIO Inversion	PIO Control	15–0
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	23–16
IOPM	I/O Port Mask	Interrupt Mask	23–16
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	29–28
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 DMA0 Control DMA1 Control DMA2 Control DMA3 Control	28 28 11 11 11 11
LOOP	Loopback	Serial Port A Control Serial Port B Control	26 26

Table C-1 Peripheral Register Field Summary (continued)

LSI	Line Sync Invert	Video Control	2
MEMADDR	Memory Address	DMA0 Address	25–0
		DMA0 Address Tail	25–0
		DMA1 Address	25–0
		DMA1 Address Tail	25–0
		DMA2 Address	25–0
		DMA2 Address Tail	25–0
		DMA3 Address	25–0
		DMA3 Address Tail	25–0
MODE0	Parallel Port Mode 0	Parallel Port Control	13–11
	Video Interface Mode 0	Video Control	18–16
MODE1	Parallel Port Mode 1	Parallel Port Control	9–8
	Video Interface Mode 1	Video Control	9–8
OER	Overrun Error	Serial Port A Status	0
		Serial Port B Status	0
PCE	Parity Check Enable	DRAM Control	14
PDATA	Parallel Port Data	Parallel Port Data	7–0
			31–0
PER	Parity Error	Serial Port A Status	1
		Serial Port B Status	1
PERADDR	Peripheral Address	DMA0 Address	31–24
		DMA1 Address	31–24
		DMA2 Address	31–24
		DMA3 Address	31–24
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
PIN	PIO Input	PIO Input	15–0
PMODE	Parity Mode	Serial Port A Control	21–19
		Serial Port B Control	21–19
POE	Parity Odd or Even	DRAM Control	13
POEN	PIO Output Enable	PIO Output Enable	15–0
POUT	PIO Output	PIO Output	15–0
PPI	Parallel Port Interrupt	Interrupt Control	11
PPM	Parallel Port Mask	Interrupt Mask	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
		DMA1 Control	4
		DMA2 Control	4
		DMA3 Control	4
RDATA	Receive Data	Serial Port A Receive Buffer	7–0
		Serial Port B Receive Buffer	7–0
RDR	Receive Data Ready	Serial Port A Status	8
		Serial Port B Status	8
REFRATE	Refresh Rate	DRAM Control	8–0
RMAD	ROM Address	DMA0 Control	12
		DMA1 Control	12
		DMA2 Control	12
		DMA3 Control	12

Table C-1 Peripheral Register Field Summary (continued)

RMODE0	Receive Mode 0	Serial Port A Control Serial Port B Control	7–5 7–5
RMODE1	Receive Mode1	Serial Port A Control Serial Port B Control	1–0 1–0
RSIE	Receive Status Interrupt Enable	Serial Port A Control Serial Port B Control	2 2
RW	Read/Write	DMA0 Control DMA1 Control DMA2 Control DMA3 Control	8 8 8 8
RXDIA	Serial Port A Receive Data Interrupt	Interrupt Control	6
RXDIB	Serial Port B Receive Data Interrupt	Interrupt Control	3
RXDMA	Serial Port A Receive Data Mask	Interrupt Mask	6
RXDMB	Serial Port B Receive Data Mask	Interrupt Mask	3
RXSIA	Serial Port A Receive Status Interrupt	Interrupt Control	7
RXSIB	Serial Port B Receive Status Interrupt	Interrupt Control	4
RXSMA	Serial Port A Receive Status Mask	Interrupt Mask	7
RXSMB	Serial Port B Receive Status Mask	Interrupt Mask	4
SDIR	Shift Direction	Video Control	1
STB	PSTROBE Level	Parallel Port Status	31
STP	Stop Bits	Serial Port A Control Serial Port B Control	18 18
TDATA	Transmit Data	Serial Port A Transmit Holding Serial Port B Transmit Holding	7–0 7–0
TDELAY	Transfer Delay	Parallel Port Control	23–16
TDELAYV	TDELAY Counter Value	Parallel Port Status	23–16
TDMO	TDMA Output	DMA0 Control DMA1 Control DMA2 Control DMA3 Control	18 18 18 18
TEMT	Transmitter Empty	Serial Port A Status Serial Port B Status	10 10
THRE	Transmit Holding Register Empty	Serial Port A Status Serial Port B Status	9 9
TMODE0	Transmit Mode 0	Serial Port A Control Serial Port B Control	15–13 15–13
TMODE1	Transmit Mode 1	Serial Port A Control Serial Port B Control	9–8 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 DMA2 Control DMA3 Control	6 6 6 6
TTI	TDMA Terminate Interrupt	DMA0 Control DMA1 Control DMA2 Control DMA3 Control	1 1 1 1
TXDIA	Serial Port A Transmit Data Interrupt	Interrupt Control	5
TXDIB	Serial Port B Transmit Data Interrupt	Interrupt Control	2
TXDMA	Serial Port A Transmit Data Mask	Interrupt Mask	5

Table C-1 Peripheral Register Field Summary (continued)

TXDMB	Serial Port B Transmit Data Mask	Interrupt Mask	2
UD	Transfer Up/Down	DMA0 Control DMA1 Control DMA2 Control DMA3 Control	9 9 9 9
VIDI	Video Invert	Video Control	0
VDATA	Video Data	Video Data Holding	31–0
VDI	Video Interrupt	Interrupt Control	27
VDM	Video Mask	Interrupt Mask	27
WLGN	Word Length	Serial Port A Control Serial Port B Control	17–16 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

Appendix

D Am29240, Am29245, and Am29243 RISC Microcontrollers Data Sheet

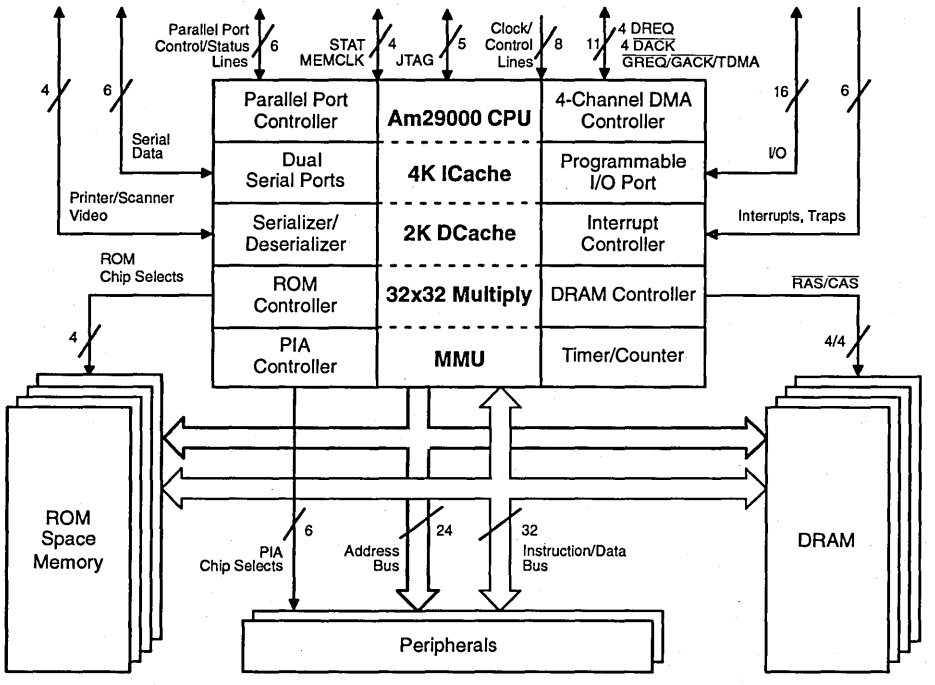
ADVANCE INFORMATION

Am29240™, Am29245™, and Am29243™

High-Performance RISC Microcontrollers

Advanced Micro Devices

Am29240 MICROCONTROLLER BLOCK DIAGRAM



DISTINCTIVE CHARACTERISTICS

Am29240 Microcontroller

- Completely integrated system for embedded applications
- Full 32-bit architecture
- 4-Kbyte two-way set-associative instruction cache
- 2-Kbyte two-way set-associative data cache
- Single cycle 32-bit multiplier for faster integer math; two-cycle Multiply Accumulate (MAC) function
- 16-entry on-chip Memory Management Unit (MMU) with one Translation Look-Aside Buffer
- 4-Gbyte virtual address space, 304-Mbyte physical space implemented
- Glueless system interfaces with on-chip wait state control
- 25 million instructions per second (MIPS) sustained at 33 MHz
- Four banks of ROM, each separately programmable for 8-, 16-, or 32-bit interface
- Four banks of DRAM, each separately programmable for 16- or 32-bit interface
- Single-cycle ROM burst-mode and DRAM page-mode access
- 4-channel double-buffered DMA controller with queued reload
- 6-port peripheral interface adapter
- 16-line programmable I/O port

Publication #: 17787 Rev. B Amendment: /0
Issue Date: July 1993

- **Two serial ports (UARTs)**
- **Bidirectional parallel port controller**
- **Bidirectional bit serializer/deserializer (video interface)**
- **Interrupt controller**
- **Full- and double-speed internal clock**
- **Fully pipelined**
- **Three-address instruction architecture**
- **192 general purpose registers**
- **20-, 25-, and 33-MHz operating frequencies**
- **Traceable Cache™ instruction and data cache tracing feature**
- **IEEE Std. 1149.1–1990 (JTAG) compliant Standard Test Access Port and Boundary Scan Architecture**
- **Binary compatibility with all 29K Family microprocessors and microcontrollers**
- **Fully static system clock capabilities**
- **3.3 V–5 V operating range**
- **CMOS technology/TTL compatible**

Am29245 Microcontroller

The low-cost Am29245 microcontroller is similar to the Am29240 microcontroller, without the data cache and 32-bit multiplier. It includes the following features:

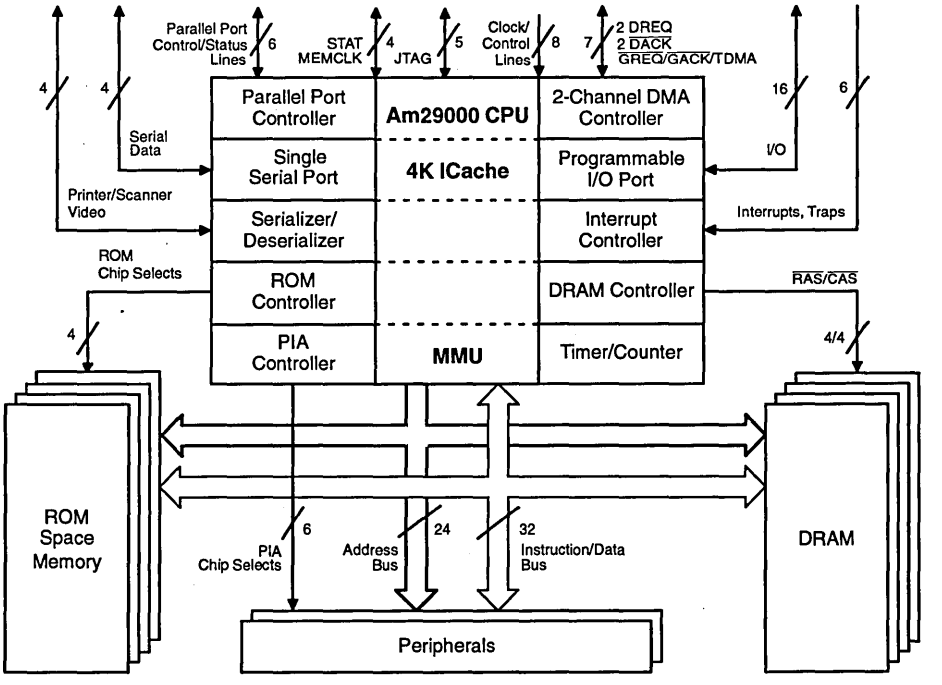
- **One serial port (UART)**
- **Two-channel DMA controller**
- **16-MHz operating frequency**

Am29243 Microcontroller

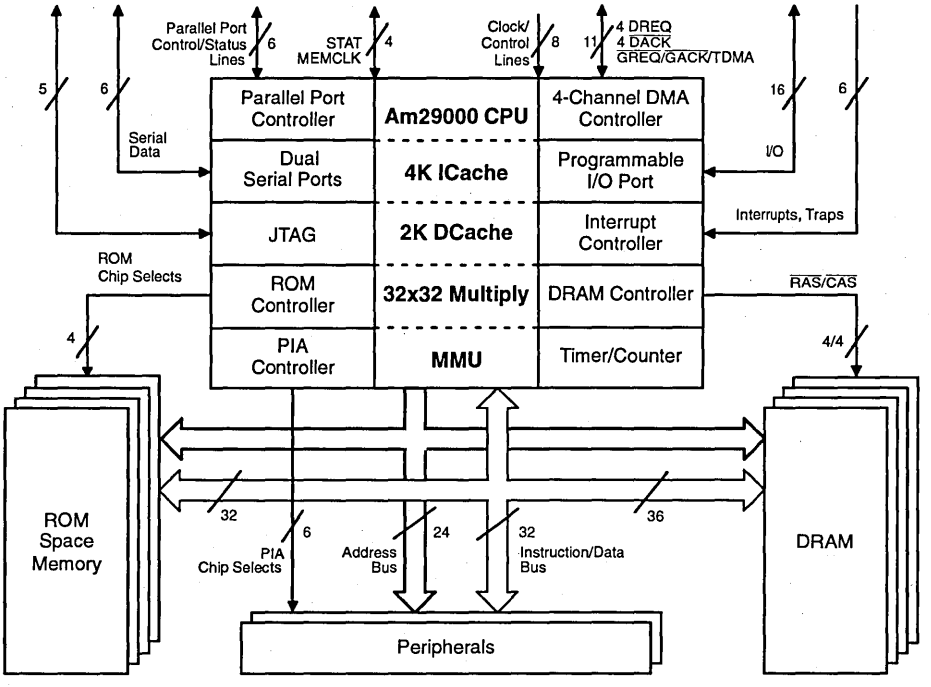
The Am29243 data microcontroller is similar to the Am29240 microcontroller, without the video interface. It includes the following additional features:

- **DRAM parity**
- **32-entry on-chip MMU with dual Translation Look-Aside Buffers (TLBs)**

Am29245 MICROCONTROLLER BLOCK DIAGRAM



Am29243 MICROCONTROLLER BLOCK DIAGRAM



GENERAL DESCRIPTION

The Am29240 microcontroller series is an enhanced bus-compatible extension of the Am29200™ RISC microcontroller family, with two to four times the performance. The Am29240 microcontroller series includes the Am29240 microcontroller, the low-cost Am29245 microcontroller, and the Am29243 data microcontroller. The on-chip caches, MMU, faster integer math, and extended DMA addressing capability of the Am29240 microcontroller series allow the embedded systems designer to provide increasing levels of performance and software compatibility throughout a range of products (see Table 1).

Based on a static low-voltage design, these CMOS-technology devices offer a complete set of system peripherals and interfaces commonly used in embedded applications. Compared to CISC processors, the Am29240 microcontroller series offers better performance, more efficient use of low-cost memories, 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 Am29240 microcontroller series provides the embedded product designer with the cost and performance edge required by today's marketplace.

Am29240 Microcontroller

For general purpose embedded applications, such as mass storage controllers, communications, digital signal processing, networking, industrial control, pen-based systems, and multimedia, the Am29240 microcontroller provides a high-performance solution with a low total system cost. The memory interface of the Am29240 microcontroller provides even faster direct memory access than the Am29200 microcontroller. This performance improvement minimizes the effect of memory latency, allowing designers to use low-cost memory with simpler memory designs. On-chip instruction and data caches provide even better performance for time-critical code. Other on-chip functions include: a ROM controller, DRAM controller, peripheral interface adapter controller, DMA controller, programmable I/O port, parallel port controller, serial ports, and an interrupt controller. For a complete description of the technical features, on-chip peripherals, programming interface, and instruction set, please refer to the *Am29240, Am29245, and Am29243 RISC Microcontrollers User's Manual and Data Sheet* (order #17741C).

The Am29240 microcontroller is available in a 196-pin plastic quad flat-pack (PQFP) package. Of the available 196 pins, 150 are signal inputs and outputs, 36 are power and ground connections, and 10 are no-connects.

Am29245 Microcontroller

The low-cost Am29245 microcontroller is designed for embedded applications in which cost and space constraints, along with increased performance requirements, are primary considerations. In addition, the Am29245 microcontroller provides an easy upgrade path for Am29200 and Am29205™ microcontroller-based products.

The Am29245 microcontroller is available in a 196-pin PQFP package. Of the available 196 pins, 144 are signal inputs and outputs, 36 are power and ground connections, and 16 are no-connects.

Am29243 Microcontroller

With DRAM parity support and a full MMU, the Am29243 data microcontroller is recommended for communications applications that require high-speed data movement and fast protocol processing in a fault-tolerant environment.

Both the Am29243 and Am29240 microcontrollers support fly-by DMA at 100 Mbytes/sec for LANs and switching applications, and a two-cycle Multiply Accumulate function for DSP applications. The low power requirements make either microcontroller a good choice for field-deployed devices.

The Am29243 microcontroller is available in a 196-pin PQFP package. Of the available 196 pins, 150 are signal inputs and outputs, 36 are power and ground connections, and 8 are no-connects.

RELATED AMD PRODUCTS

29K Family Devices

Part No.	Description
Am29000™	32-Bit RISC Microprocessor
Am29005™	Low-Cost 32-Bit RISC Microprocessor with No MMU and No BTC
Am29030™	32-Bit RISC Microprocessor with 8-Kbyte Instruction Cache
Am29035™	32-Bit RISC Microprocessor with 4-Kbyte Instruction Cache
Am29050™	32-Bit RISC Microprocessor with On-Chip Floating Point
Am29200	32-Bit RISC Microcontroller
Am29205	Low-Cost RISC Microcontroller with 16-Bit Bus Interface

29K™ Family Development Support Products

Contact your local AMD representative for information on the complete set of development support tools. The following software and hardware 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
- Execution boards

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 from the AMD Fusion29K Partners include

- Silicon products
- Software generation and debug tools
- Hardware development tools
- Board level products
- Laser printer solutions
- Multiuser, kernel, and real-time operating systems
- Graphics solutions
- Networking and communication solutions
- Manufacturing support
- Custom software consulting, support, and training

Table D-1 Product Comparison—Am29200 Microcontroller Family

FEATURE	Am29205 Microcontroller	Am29200 Microcontroller	Am29245 Microcontroller	Am29240 Microcontroller	Am29243 Microcontroller
Instruction Cache	—	—	4 Kbytes	4 Kbytes	4 Kbytes
Data Cache	—	—	—	2 Kbytes	2 Kbytes
Integer Multiplier	Software	Software	Software	32 x 32-bit	32 x 32-bit
MMU	—	—	1 TLB 16 Entry	1 TLB 16 Entry	2 TLBs 32 Entry
Data Bus Width					
Internal	32 bits	32 bits	32 bits	32 bits	32 bits
External	16 bits	32 bits	32 bits	32 bits	32 bits
ROM Interface					
Banks	3	4	4	4	4
Width	16 bits only	8, 16, 32 bits	8, 16, 32 bits	8, 16, 32 bits	16, 32 bits
ROM Size (Max/Bank)	4 Mbytes	16 Mbytes	16 Mbytes	16 Mbytes	16 Mbytes
Boot-up ROM Width	16 bits	8, 16, 32 bits	8, 16, 32 bits	8, 16, 32 bits	8, 16, 32 bits
Burst-mode access	Not Supported	Supported	Supported	Supported	Supported
DRAM Interface					
Banks	4	4	4	4	4
Width	16 bits only	16, 32 bits	16, 32 bits	16, 32 bits	8, 16, 32 bits
Size: 32-bit mode	—	16 Mbytes/bank	16 Mbytes/bank	16 Mbytes/bank	16 Mbytes/bank
Size: 16-bit mode	8 Mbytes/bank	8 Mbytes/bank	8 Mbytes/bank	8 Mbytes/bank	8 Mbytes/bank
Video DRAM	Not Supported	Supported	Supported	Supported	Not Supported
Initial/Burst Access Cycles	3/2	3/2	2/1	2/1	2/1
On-Chip DMA					
Width (ext. peripherals)	8, 16 bits	8, 16, 32 bits	8, 16, 32 bits	8, 16, 32 bits	8, 16, 32 bits
Externally Controlled	1 Channel	2 Channels	2 Channels	4 Channels	4 Channels
$\overline{\text{GREQ}}$/$\overline{\text{GACK}}$ Access	No	Yes	Yes	Yes	Yes
$\overline{\text{GREQ}}$/$\overline{\text{GACK}}$ Burst	No	No	Yes	Yes	Yes
TDMA	No	Yes	Yes	Yes	Yes
Double-Frequency CPU Option	No	No	No	Yes	Yes
Low Voltage Operation	No	No	Yes	Yes	Yes
PIA					
PIA Ports	2	6	6	6	6
Data Width	8, 16 bits	8, 16, 32 bits	8, 16, 32 bits	8, 16, 32 bits	8, 16, 32 bits
Cycles	3	3	2	2	2
Programmable I/O Port					
Signals	8	16	16	16	16
Serial Ports					
Ports	1 Port	1 Port	1 Port	2 Ports	2 Ports
$\overline{\text{DSR}}$	Not Supported	Supported	Supported	1 Port Supported	1 Port Supported
$\overline{\text{DTR}}$	Not Supported	Supported	Supported	1 Port Supported	1 Port Supported
Interrupt Controller					
External Interrupt Pins	2	4	4	4	4
External Trap and Warn Pins	0	3	3	3	3
Parallel Port Controller	Yes	Yes	Yes	Yes	Yes
Full-Word Transfer	No	Yes	Yes	Yes	Yes
JTAG Testing	No	Yes	Yes	Yes	Yes
Serializer/Deserializer	Yes	Yes	Yes	Yes	No
DRAM Parity	No	No	No	No	Yes
Pin Count and Package	100 PQFP	168 PQFP	196 PQFP	196 PQFP	196 PQFP
Processor Clock Rate	16 MHz	16, 20 MHz	16 MHz	20, 25, 33 MHz	20, 25, 33 MHz

KEY FEATURES AND BENEFITS

The Am29240 microcontroller series extends the line of RISC microcontrollers based on the 29K architecture, providing performance upgrades to the Am29205 and Am29200 microcontrollers. The RISC microcontroller product line allows users to benefit from the very high performance of the 29K architecture, while also capitalizing on the very low system cost made possible by the integration of processor and peripherals.

The Am29240 microcontroller series expands the price/performance range of systems that can be built with the 29K Family. The Am29240 microcontroller series is fully software compatible with the Am29000, Am29005, Am2903, Am29035, and Am29050 microprocessors, as well as the Am29200 and Am29205 microcontrollers. It can be used in existing 29K Family microcontroller applications without software modifications.

On-Chip Caches

The Am29240 microcontroller series incorporates a 4-Kbyte, two-way instruction cache that supplies most processor instructions without wait states at the processor frequency. For best performance, the instruction cache supports critical-word-first reloading with fetch-through, so that the processor receives the required instruction and the pipeline restarts with minimum delay. The instruction cache has a valid bit per word to minimize the reload overhead. All cache array elements are visible to software for testing and preload.

The Am29240 and Am29243 microcontrollers incorporate a 2-Kbyte, two-way set-associative data cache. The data cache appears in the execute stage of the processor pipeline, so that loaded data is available immediately to the next instruction. This provides the maximum performance for loads without requiring load scheduling. The data cache performs critical-word-first, wrap-around, burst-mode refill with load-through. This minimizes the time the processor waits on external data as well as minimizing the reload time. The data cache uses a write-through policy with a two-entry write buffer. Byte, half-word, and word reads and writes are supported. All cache array elements are visible to software for testing and preload.

Single-Cycle Multiplier

The Am29240 and Am29243 microcontrollers incorporate a full combinatorial multiplier that accepts two 32-bit input operands and produces a 32-bit result in a single cycle. The multiplier can produce a 64-bit result in two cycles. The multiplier permits maximum performance without requiring instruction scheduling, since the latency of the multiply is the same as the latency of other integer operations. High-performance multiplication benefits imaging, signal processing, and state modeling applications.

Complete Set of Common System Peripherals

The Am29240 microcontroller series 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, two serial ports, and an interrupt controller. A video interface is also included in the Am29240 and Am29245 microcontrollers for printer, scanner, and other imaging applications. These facilities allow many simple systems to be built using only the Am29240 microcontroller series, external ROM, and/or DRAM memory.

ROM Controller

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

DRAM Controller

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 can appear as a contiguous memory area of up to 64 Mbyte in size. To further enhance the performance, the DRAM controller supports two-cycle accesses, with single-cycle page-mode and burst-mode accesses.

Peripheral Interface Adapter

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.

DMA Controller

The DMA controller provides up to four channels for transferring data between the DRAM and internal or external peripherals. The DMA channels are double buffered to relax constraints on reload time.

I/O Port

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.

Parallel Port

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

Serial Port

The serial port implements up to two full-duplex UARTs.

Serializer/Deserializer

The serializer/deserializer (video interface) permits direct connection to a number of laser marking engines, video displays, or raster input devices such as scanners.

Interrupt Controller

The interrupt controller generates and reports the status of interrupts caused by on-chip peripherals.

Wide Range of Price/Performance Points

To reduce design costs and time-to-market, the product designer can use the Am29200 microcontroller family and one basic system design 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 Am29240 RISC microcontroller series supports this capability through various combinations of on-chip caches, programmable memory widths, programmable wait states, burst-mode and page-mode access support, bus compatibility, and 29K Family software compatibility. A system can be upgraded without hardware and software redesign using various memory architectures.

Within the Am29240 microcontroller series, the external interfaces operate at frequencies in the range of 16 to 25 MHz, and the processor operates at frequencies in the range of 16 to 33 MHz. The internal processor core can operate either at the interface frequency or twice this frequency. For example, the processor can operate at 33 MHz while the interface operates at 16.5 MHz.

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, Am29205, Am29240, Am29245, and Am29243 microcontrollers allow users to address an extremely wide range of cost performance points, with higher performance and lower cost than existing designs based on CISC microprocessors.

Glueless System Interfaces

The Am29240 microcontroller series 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.

Bus- and Software-Compatibility

Compatibility within a processor family is critical for achieving a rational, easy upgrade path. The Am29240 processors are all members of a bus-compatible series of RISC microcontrollers. All members of this family, the Am29205, Am29200, Am29240, Am29245, and Am29243 microcontrollers, allow improvements in price, performance, and system capabilities without requiring that users redesign their system hardware or software. Bus compatibility ensures a convenient upgrade path for future systems.

The Am29240 microcontroller series is available in a 196-pin plastic quad flat-pack (PQFP) package. The Am29240 microcontroller series is signal-compatible with the Am29205 and the Am29200 microcontrollers.

Moreover, the Am29240 microcontroller series is binary compatible with existing RISC microcontrollers and other members of the 29K Family (the Am29000, Am29005, Am29030, Am29035, and Am29050 microprocessors, as well as the Am29205 and Am29200 microcontrollers). The Am29240 microcontroller series provides a migration path to low-cost, high-performance, highly integrated systems from other 29K Family members, without requiring expensive rewrites of application software.

Complete Development and Support Environment

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

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

- High C® 29K optimizing C compiler with assembler, linker, ANSI library functions, and 29K architectural simulator
- XRAY29K™ source-level debugger
- MiniMON29K™ debug monitor
- A complete family of demonstration and development boards

In addition, Advanced Micro Devices has developed a standard host interface (HIF) specification for operating system services, the Universal Debug Interface (UDI) for seamless connection of debuggers to ICEs and target hardware, 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.

PERFORMANCE OVERVIEW

The Am29240 microcontroller series 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 very low cost. This section describes the features of the Am29240 microcontroller series from the point of view of system performance.

Instruction Timing

The Am29240 microcontroller series 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 Am29240 microcontroller series by overlapping them with instruction execution, by taking advantage of pipelining, by an on-chip data cache, and by organizing the flow of external data into the processor so that the impact of external accesses is minimized.

Pipelining

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.

On-Chip Instruction and Data Caches

On chip instruction and data caches satisfy most processor fetches without wait states, even when the processor operates at twice the system frequency. The caches are pipelined for best performance. The reload policies minimize the amount of time spent waiting for reload, while optimizing the benefit of locality of reference.

Burst-Mode and Page-Mode Memories

The Am29240 microcontroller series directly supports burst-mode memories. 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 processor can also 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.

Instruction Set Overview

All 29K Family members employ a three-address instruction set architecture. The compiler or assembly-language programmer is given complete freedom to allocate register usage. There are 192 general-purpose registers, allowing the retention of intermediate calculations and avoiding needless data destruction. Instruction operands may be contained in any of the general-purpose registers, and the results may be stored into any of the general-purpose registers.

The Am29240 microcontroller series instruction set contains 117 instructions that 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.

Data Formats

The Am29240 microcontroller series 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 most significant bit 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 Am29240 microcontroller series.

Protection

The Am29240 microcontroller series offers two mutually exclusive modes of execution, the user and supervisor modes, that 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.

Memory Management Unit

The Am29240 microcontroller series provides a memory-management unit (MMU) for translating virtual addresses into physical addresses. The page size for translation ranges from 1 Kbyte to 16 Mbyte in powers of four. The Am29245 and Am29240 microcontrollers each have a single, 16-entry TLB. The Am29243 microcontroller has dual 16-entry TLBs, each capable of mapping pages of different size.

Interrupts and Traps

When an Am29240 microcontroller series 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 that 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 that 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.

DEBUGGING AND TESTING

The Am29240 microcontroller series 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.

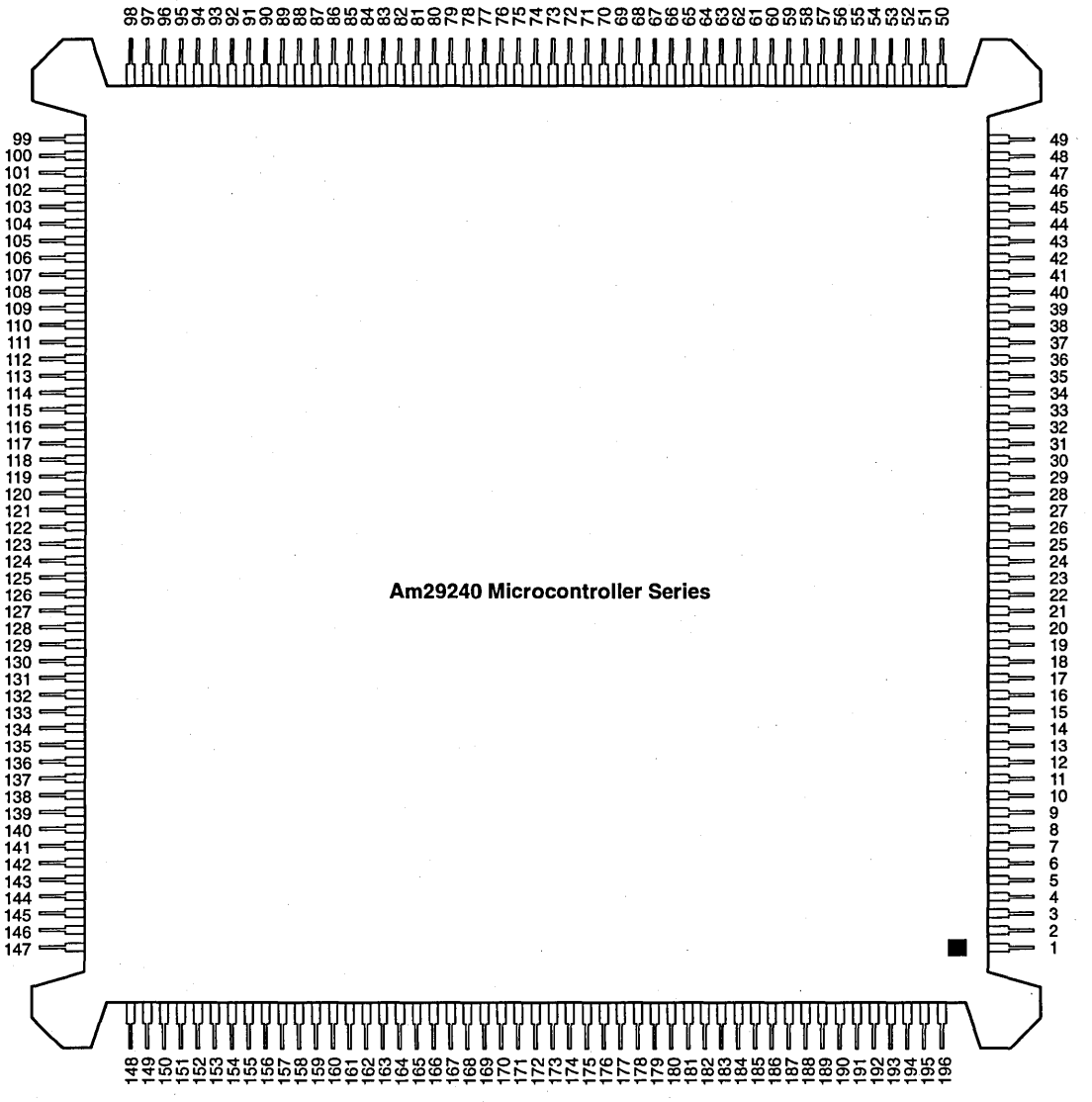
The processor provides several additional features to assist system debugging and testing:

- 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.
- A Traceable Cache feature permits a hardware-development system to track accesses to the on-chip caches, permitting a high level of visibility into processor operation.
- 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 processor and 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.

CONNECTION DIAGRAM

Top Side View

196-Pin PQFP (Plastic Quad Flat Pack) Package



Notes: *Pin 1 marked for orientation.*

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}	50	V_{CC}	99	V_{CC}	148	V_{CC}
2	MEMCLK	51	Reserved	100	Reserved	149	Reserved
3	MEMDRV	52	Reserved	101	Reserved	150	PIO12
4	INCLK	53	TXDB[3]	102	A23	151	PIO11
5	ID31	54	RXDB[3]	103	A22	152	PIO10
6	ID30	55	$\overline{DTRA}$	104	A21	153	PIO9
7	ID29	56	RXDA	105	A20	154	PIO8
8	ID28	57	UCLK	106	A19	155	PIO7
9	ID27	58	$\overline{DSRA}$	107	A18	156	PIO6
10	ID26	59	TXDA	108	A17	157	PIO5
11	ID25	60	$\overline{ROMCS3}$	109	A16	158	PIO4
12	ID24	61	$\overline{ROMCS2}$	110	V_{SS}	159	V_{SS}
13	V_{SS}	62	$\overline{ROMCS1}$	111	V_{CC}	160	V_{CC}
14	V_{CC}	63	$\overline{ROMCS0}$	112	A15	161	PIO3
15	ID23	64	V_{CC}	113	A14	162	PIO2
16	ID22	65	V_{SS}	114	A13	163	PIO1
17	ID21	66	$\overline{BURST}$	115	A12	164	PIO0
18	ID20	67	$\overline{RSWE}$	116	A11	165	TDO
19	ID19	68	$\overline{ROMOE}$	117	A10	166	STAT2
20	ID18	69	$\overline{RAS3}$	118	A9	167	STAT1
21	ID17	70	$\overline{RAS2}$	119	A8	168	STAT0
22	ID16	71	$\overline{RAS1}$	120	V_{SS}	169	VDAT[2]
23	V_{SS}	72	$\overline{RAS0}$	121	V_{CC}	170	PSYNC[2]
24	V_{CC}	73	$\overline{CAS3}$	122	A7	171	V_{SS}
25	ID15	74	$\overline{CAS2}$	123	A6	172	V_{CC}
26	ID14	75	V_{CC}	124	A5	173	$\overline{GREQ}$
27	ID13	76	V_{SS}	125	A4	174	DREQB
28	ID12	77	$\overline{CAS1}$	126	A3	175	DREQA
29	ID11	78	$\overline{CAS0}$	127	A2	176	TDMA
30	ID10	79	$\overline{TR/OE}$	128	A1	177	$\overline{TRAP0}$
31	ID9	80	$\overline{WE}$	129	A0	178	$\overline{TRAP1}$
32	ID8	81	$\overline{GACK}$	130	V_{SS}	179	$\overline{INTR0}$
33	V_{SS}	82	$\overline{PIACS5}$	131	V_{CC}	180	$\overline{INTR1}$
34	V_{CC}	83	$\overline{PIACS4}$	132	BOOTW	181	$\overline{INTR2}$
35	ID7	84	$\overline{PIACS3}$	133	$\overline{WAIT}$	182	$\overline{INTR3}$
36	ID6	85	$\overline{PIACS2}$	134	PAUTOFD	183	V_{SS}
37	ID5	86	V_{CC}	135	PSTROBE	184	V_{CC}
38	ID4	87	V_{SS}	136	$\overline{PWE}$	185	$\overline{WARN}$
39	ID3	88	$\overline{PIACS1}$	137	$\overline{POE}$	186	VCLK[2]
40	ID2	89	$\overline{PIACS0}$	138	PACK	187	LSYNC[2]
41	ID1	90	$\overline{PIAWE}$	139	$\overline{PBUSY}$	188	TMS
42	ID0	91	$\overline{PIAOE}$	140	V_{SS}	189	$\overline{TRST}$
43	V_{SS}	92	$R/\overline{W}$	141	V_{CC}	190	TCK
44	V_{CC}	93	$\overline{DACKB}$	142	PIO15	191	TDI
45	IDP3 [1,3]	94	$\overline{DACKA}$	143	PIO14	192	$\overline{RESET}$
46	IDP2 [1,3]	95	$\overline{DACKD}$[3]	144	PIO13	193	CNTL1
47	IDP1 [1,3]	96	$\overline{DACKC}$[3]	145	DREQD[3]	194	CNTL0
48	IDP0 [1,3]	97	Reserved	146	DREQC[3]	195	$\overline{TRIST}$
49	V_{SS}	98	V_{SS}	147	V_{SS}	196	V_{SS}

Notes: All values are typical and preliminary.

1. *Defined as a no-connect on the Am29240 microcontroller.*
2. *Defined as a no-connect on the Am29243 microcontroller.*
3. *Defined as a no-connect on the Am29245 microcontroller.*

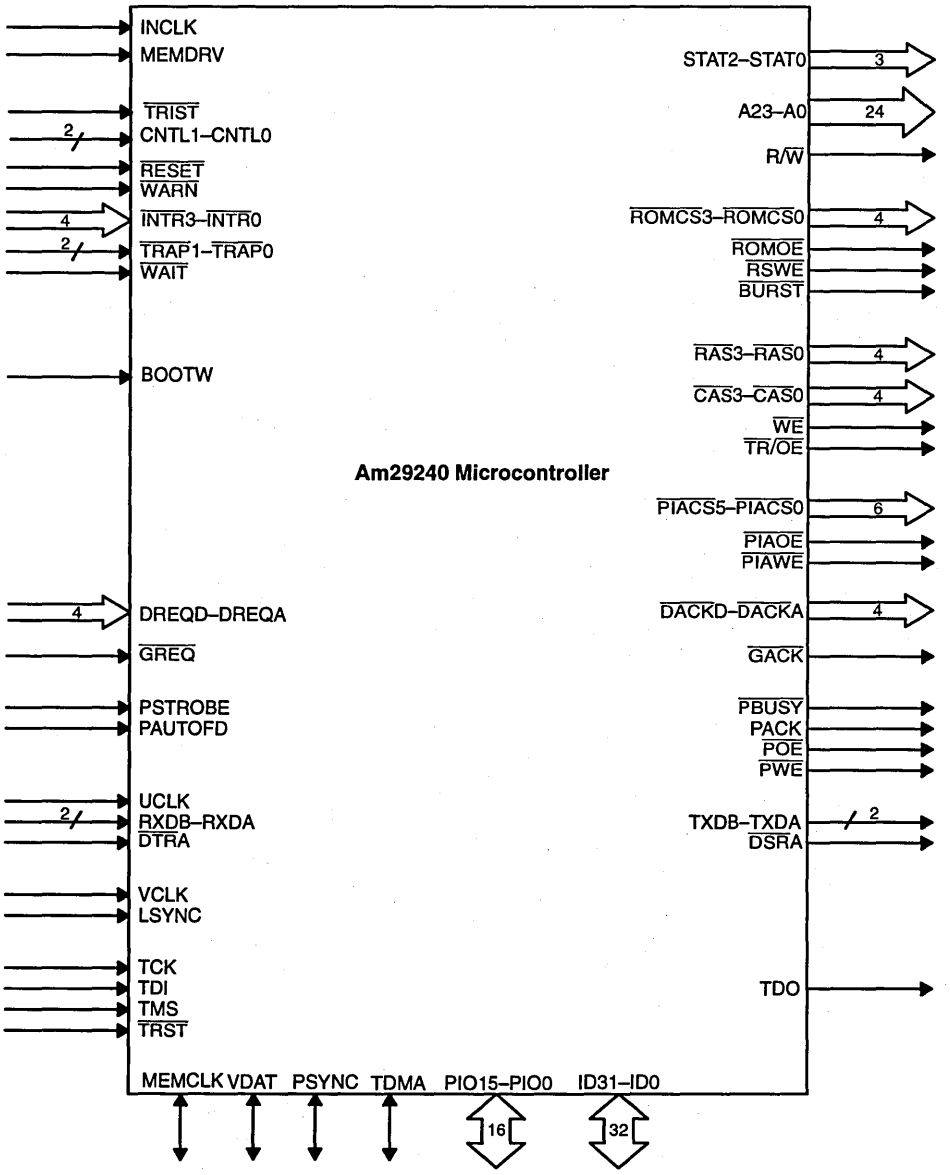
PQFP PIN DESIGNATION – Sorted by Pin NAME

PIN NAME	PIN NO.	PIN NAME	PIN NO.	PIN NAME	PIN NO.	PIN NAME	PIN NO.
A0	129	ID5	37	$\overline{PIAWE}$	90	$\overline{TR/OE}$	79
A1	128	ID6	36	PIO0	164	$\overline{TRAP0}$	177
A2	127	ID7	35	PIO1	163	$\overline{TRAP1}$	178
A3	126	ID8	32	PIO2	162	$\overline{TRIST}$	195
A4	125	ID9	31	PIO3	161	$\overline{TRST}$	189
A5	124	ID10	30	PIO4	158	TXDA	59
A6	123	ID11	29	PIO5	157	TXDB[3]	53
A7	122	ID12	28	PIO6	156	UCLK	57
A8	119	ID13	27	PIO7	155	V_{CC}	1
A9	118	ID14	26	PIO8	154	V_{CC}	14
A10	117	ID15	25	PIO9	153	V_{CC}	24
A11	116	ID16	22	PIO10	152	V_{CC}	34
A12	115	ID17	21	PIO11	151	V_{CC}	44
A13	114	ID18	20	PIO12	150	V_{CC}	50
A14	113	ID19	19	PIO13	144	V_{CC}	64
A15	112	ID20	18	PIO14	143	V_{CC}	75
A16	109	ID21	17	PIO15	142	V_{CC}	86
A17	108	ID22	16	$\overline{POE}$	137	V_{CC}	99
A18	107	ID23	15	PSTROBE	135	V_{CC}	111
A19	106	ID24	12	PSYNC[2]	170	V_{CC}	121
A20	105	ID25	11	$\overline{PWE}$	136	V_{CC}	131
A21	104	ID26	10	$\overline{RAS0}$	72	V_{CC}	141
A22	103	ID27	9	$\overline{RAS1}$	71	V_{CC}	148
A23	102	ID28	8	$\overline{RAS2}$	70	V_{CC}	160
BOOTW	132	ID29	7	$\overline{RAS3}$	69	V_{CC}	172
$\overline{BURST}$	66	ID30	6	Reserved	51	V_{CC}	184
$\overline{CAS0}$	78	ID31	5	Reserved	52	VCLK[2]	186
$\overline{CAS1}$	77	IDP0[1,3]	48	Reserved	97	VDAT[2]	169
$\overline{CAS2}$	74	IDP1[1,3]	47	Reserved	100	V_{SS}	13
$\overline{CAS3}$	73	IDP2[1,3]	46	Reserved	101	V_{SS}	23
CNTL0	194	IDP3[1,3]	45	Reserved	149	V_{SS}	33
CNTL1	193	INCLK	4	$\overline{RESET}$	192	V_{SS}	43
$\overline{DACKA}$	94	$\overline{INTR0}$	179	$\overline{ROMCS0}$	63	V_{SS}	49
$\overline{DACKB}$	93	$\overline{INTR1}$	180	$\overline{ROMCS1}$	62	V_{SS}	65
$\overline{DACKC}$[3]	96	$\overline{INTR2}$	181	$\overline{ROMCS2}$	61	V_{SS}	76
$\overline{DACKD}$[3]	95	$\overline{INTR3}$	182	$\overline{ROMCS3}$	60	V_{SS}	87
DREQA	175	LSYNC[2]	187	$\overline{ROMOE}$	68	V_{SS}	98
DREQB	174	MEMCLK	2	$\overline{RSWE}$	67	V_{SS}	110
DREQC[3]	146	MEMDRV	3	RXDA	56	V_{SS}	120
DREQD[3]	145	PACK	138	RXDB[3]	54	V_{SS}	130
$\overline{DSRA}$	58	PAUTOFD	134	$\overline{R/W}$	92	V_{SS}	140
$\overline{DTRA}$	55	$\overline{PBUSY}$	139	STAT0	168	V_{SS}	147
$\overline{GACK}$	81	$\overline{PIACS0}$	89	STAT1	167	V_{SS}	159
$\overline{GREQ}$	173	$\overline{PIACS1}$	88	STAT2	166	V_{SS}	171
ID0	42	$\overline{PIACS2}$	85	TCK	190	V_{SS}	183
ID1	41	$\overline{PIACS3}$	84	TDO	165	V_{SS}	196
ID2	40	$\overline{PIACS4}$	83	TDI	191	$\overline{WAIT}$	133
ID3	39	$\overline{PIACS5}$	82	TDMA	176	$\overline{WARN}$	185
ID4	38	$\overline{PIAOE}$	91	TMS	188	$\overline{WE}$	80

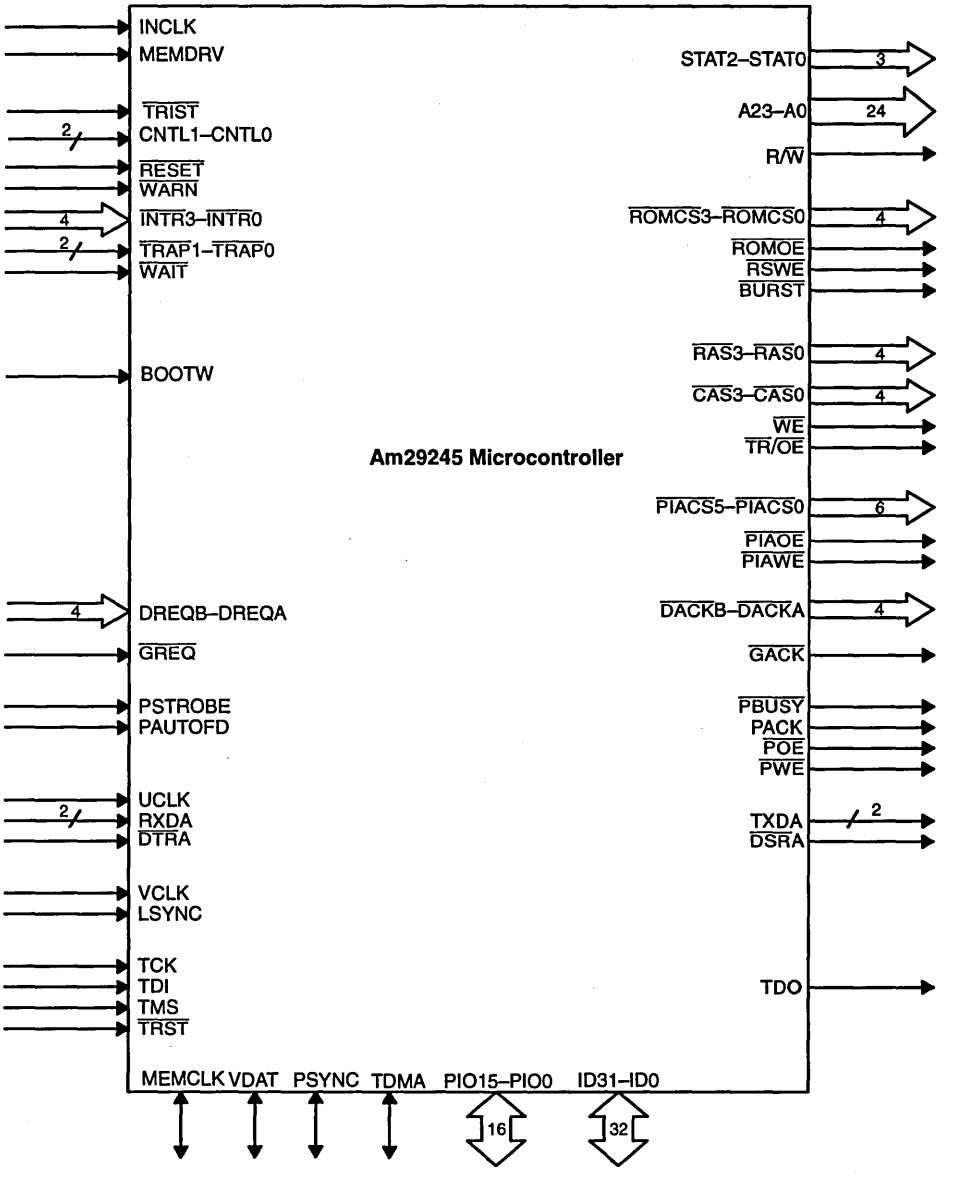
Notes: All values are typical and preliminary.

1. *Defined as a no-connect on the Am29240 microcontroller.*
2. *Defined as a no-connect on the Am29243 microcontroller.*
3. *Defined as a no-connect on the Am29245 microcontroller.*

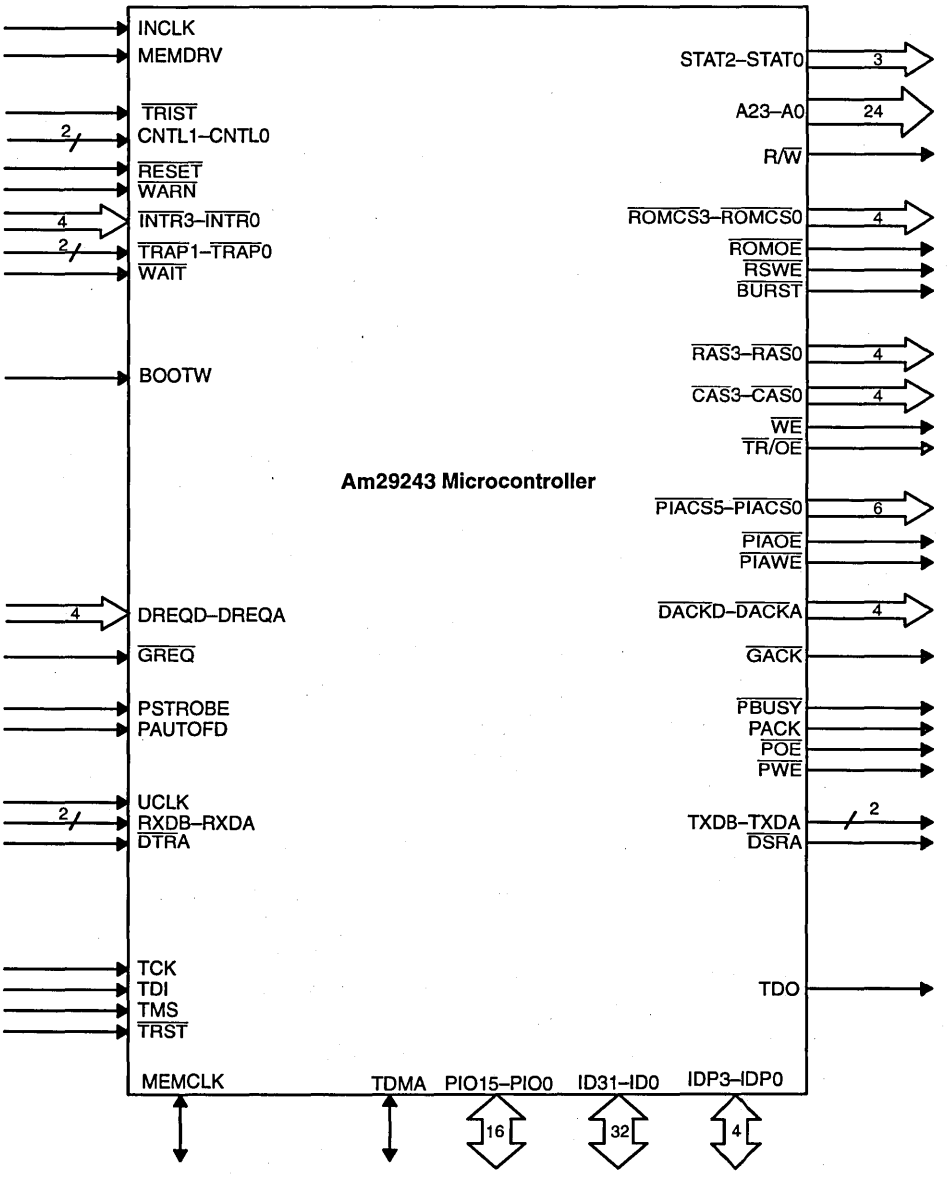
Am29240 MICROCONTROLLER LOGIC SYMBOL



Am29245 MICROCONTROLLER LOGIC SYMBOL



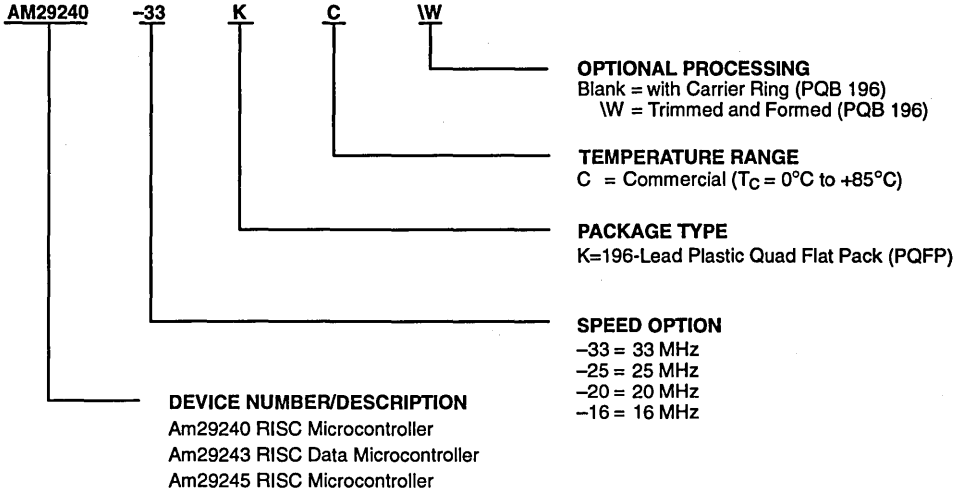
Am29243 MICROCONTROLLER LOGIC SYMBOL



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	
AM29240–20 AM29240–25 AM29240–33	KC, KC\W
AM29243–20 AM29243–25 AM29243–33	KC, KC\W
AM29245–16	KC, KC\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 +125°C
Voltage on any Pin
with Respect to GND ..... –0.5 V to V_{CC} +0.5 V

Stresses above those listed under ABSOLUTE MAXIMUM RATINGS may cause permanent device failure. Functionality at or above these limits is not implied. Exposure to absolute maximum ratings for extended periods may affect device reliability.

OPERATING RANGES

Commercial (C) Devices

Case Temperature (T_C) ..... 0°C to +85°C
Supply Voltage (V_{CC}) ..... +4.75 V to +5.25 V

Operating ranges define those limits between which the functionality of the device is guaranteed.

DC CHARACTERISTICS over COMMERCIAL operating ranges

Symbol	Parameter Description	Test Conditions	Advance Information		Unit
			Min	Max	
V_{IL}	Input Low Voltage		–0.5	0.8	V
V_{IH}	Input High Voltage		2.4	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 with respect to MEMCLK	V_{CC} = 5.25 V, Outputs Floating; Holding $\overline{RESET}$ active at 25 MHz		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
I_{OSGND}	MEMCLK GND Short Circuit Current	V_{CC} = 5.0 V	100		mA
I_{OSVCC}	MEMCLK V_{CC} Short Circuit Current	V_{CC} = 5.0 V	100		mA

Notes: *The Low input leakage current for the inputs* $\overline{CNTL1}$–$\overline{CNTL0}$, $\overline{INTR3}$–$\overline{INTR0}$, $\overline{TRAP1}$–$\overline{TRAP0}$, DREQD–DREQA, TCK, TDI, $\overline{TRST}$, TMS, $\overline{RESET}$, $\overline{WARN}$, $\overline{MEMDRV}$, $\overline{WAIT}$, *and* $\overline{TRIST}$ *is –200 µA. These pins have internal pull-up resistors.*

CAPACITANCE

Symbol	Parameter Description	Test Conditions	Advance Information		Unit
			Min	Max	
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
$C_{I/O}$	I/O Pin Capacitance			20	pF

Notes: *Limits guaranteed by characterization.*

SWITCHING CHARACTERISTICS over COMMERCIAL operating ranges

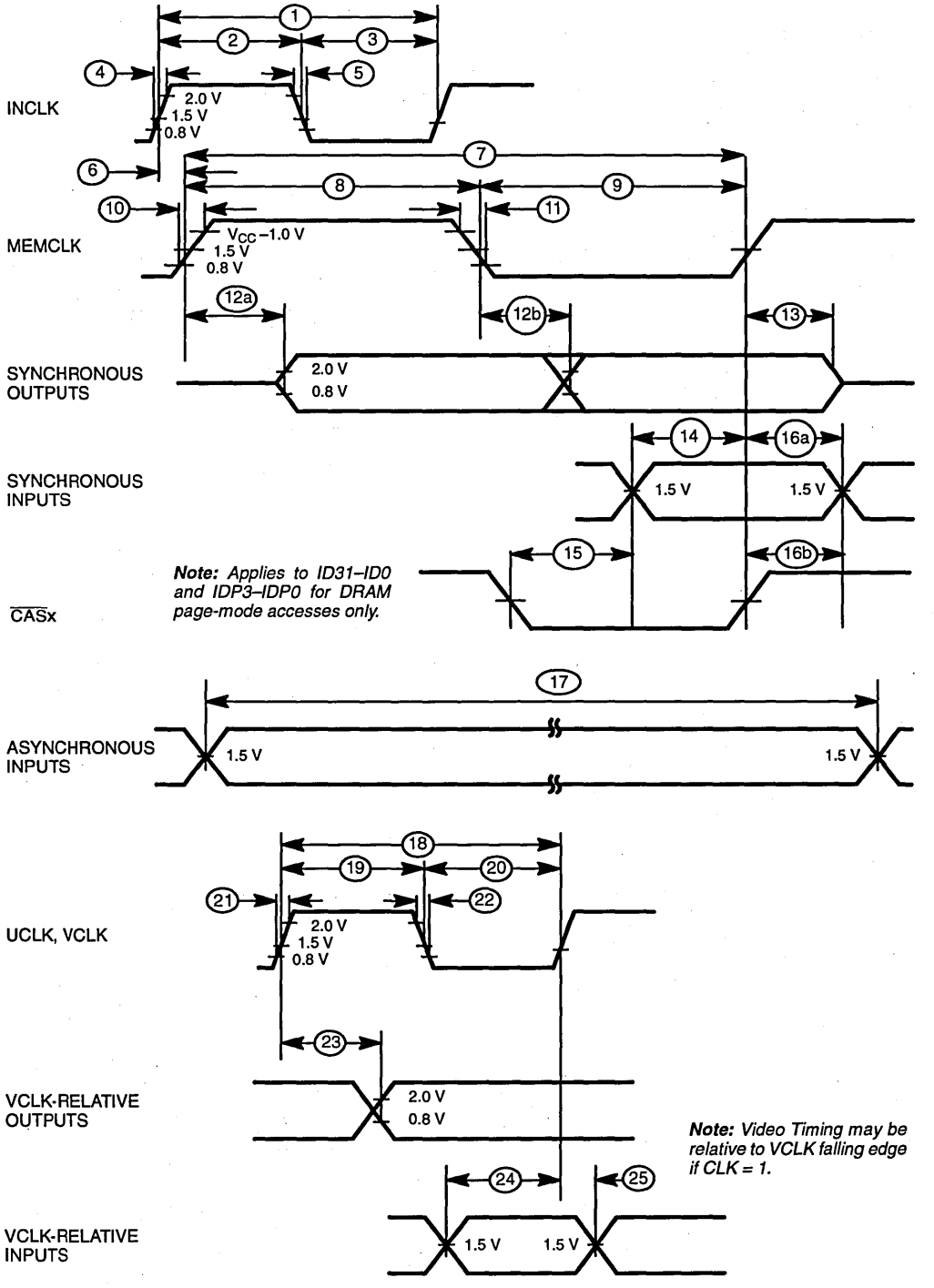
Symbol	Parameter Description	Test Conditions (Note[1])	Advance Information 25 MHz		Unit
			Min	Max	
1	INCLK Period (=0.5T)	Note[2]	20	∞	ns
2	INCLK High Time	Note[2]	6	∞	ns
3	INCLK Low Time	Note[2]	6	∞	ns
4	INCLK Rise Time	Note[2]	0	3	ns
5	INCLK Fall Time	Note[2]	0	3	ns
6	MEMCLK Delay from INCLK	MEMCLK Output[3]	0	5	ns
	MEMCLK Delay (MD) from INCLK	MEMCLK Input	5	10	ns
7	MEMCLK Period (T)	MEMCLK Input	40	∞	ns
8	MEMCLK High Time	MEMCLK Output[3]	0.5T –3	∞	ns
		MEMCLK Input	11	∞	ns
9	MEMCLK Low Time	MEMCLK Output[3]	0.5T –3	∞	ns
		MEMCLK Input	11	∞	ns
10	MEMCLK Rise Time	Note[3]	0	4	ns
11	MEMCLK Fall Time	Note[3]	0	4	ns
12a	Synchronous Output Valid Delay Rise Time from MEMCLK				
	PIO15–PIO0, STAT2–STAT0, and $\overline{\text{PIACS5}}$–$\overline{\text{PIACS0}}$	MEMCLK Output[1A]	1	11	ns
		MEMCLK Input[1A]	1	13+ (MD–5)	ns
	$\overline{\text{RAS3}}$–$\overline{\text{RAS0}}$	MEMCLK Ouput[1B]	1	17	ns
		MEMCLK Input[1B]	1	17+ (MD–5)	ns
	All others	MEMCLK Output[1C]	1	10	ns
		MEMCLK Input[1C]	1	12+ (MD–5)	ns
12b	Synchronous Output Valid Delay Fall Time from MEMCLK				
	PIO15–PIO0, STAT2–STAT0, and $\overline{\text{PIACS5}}$–$\overline{\text{PIACS0}}$	MEMCLK Output[1A]	1	10	ns
		MEMCLK Input[1A]	1	12+ (MD–5)	ns
	$\overline{\text{RAS3}}$–$\overline{\text{RAS0}}$	MEMCLK Output[1B]	1	16	ns
		MEMCLK Input[1B]	1	16+ (MD–5)	ns
	All others	MEMCLK Output[1C]	1	9	ns
		MEMCLK Input[1C]	1	11+ (MD–5)	ns
13	Synchronous Output Disable Delay from MEMCLK Rise	MEMCLK Output	1	10	ns
		MEMCLK Input	1	12+ (MD–5)	ns
14	Synchronous Input Setup Time to MEMCLK		7		ns
15	Available $\overline{\text{CAS}}$ Access Time ($T_{\overline{\text{CAS}}}$–$T_{\text{Setup}}$)	Parity Disabled[4]	0.4T		ns
		Parity Enabled[4]	0.4T –4		ns

Symbol	Parameter Description	Test Conditions (Note[1])	Advance Information 25 MHz		Unit
			Min	Max	
16a	Synchronous Input Hold Time to MEMCLK		0		ns
16b	Synchronous Input Hold Time to $\overline{CAS3}$–$\overline{CAS0}$	Note[4]	3		ns
17	Asynchronous Input Pulse Width				ns
	LSYNC and PSYNC		Note[5]		
	All others		4T		ns
18	UCLK Period	Note[2]	20		ns
	VCLK Period	Note[2]	15		ns
19	UCLK High Time	Note[2]	6		ns
	VCLK High Time	Note[2]	4		ns
20	UCLK Low Time	Note[2]	6		ns
	VCLK Low Time	Note[2]	4		ns
21	UCLK Rise time	Note[2]	0	3	ns
	VCLK Rise time	Note[2]	0	3	ns
22	UCLK Fall Time	Note[2]	0	3	ns
	VCLK Fall Time	Note[2]	0	3	ns
23	Synchronous Output Valid Delay from VCLK Rise and Fall	Note[6]	0	14	ns
24	Input Setup Time to VCLK Rise and Fall	Notes[6,7]	9		ns
25	Input Hold Time to VCLK Rise and Fall	Notes[6,7]	0		ns

Notes:

1. *All outputs driving 80 pF, measured at V_{OL} = 1.5 V and V_{OH} = 1.5 V. For higher capacitance:*
   - *A. Add 1-ns output delay per 15 pF loading up to 150-pF total.*
   - *B. Add 1-ns output delay per 25 pF loading up to 300-pF total. In order to meet the setup time (t_{ASR}) from A23–A0 to $\overline{RAS3}$–$\overline{RAS0}$ for DRAM, the capacitance loading of A23–A0 must not exceed the capacitance loading of $\overline{RAS3}$–$\overline{RAS0}$ by more than 150 pF.*
   - *C. Add 1-ns output delay per 25 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. *MEMCLK can drive an external load of 100 pF.*
4. *Applies to ID31–ID0 and IDP3–IDP0 for DRAM page-mode accesses only.*
5. *LSYNC and PSYNC minimum width is two bit-times. A bit-time is one period of the internal video clock, which is determined by the CLKDIV field in the Video Control Register and VCLK.*
6. *Active VCLK edge depends on the CLKI bit in the Video Control Register.*
7. *LSYNC and PSYNC can be treated as synchronous signals by meeting the setup and hold times, though the synchronization delay still applies.*

SWITCHING WAVEFORMS

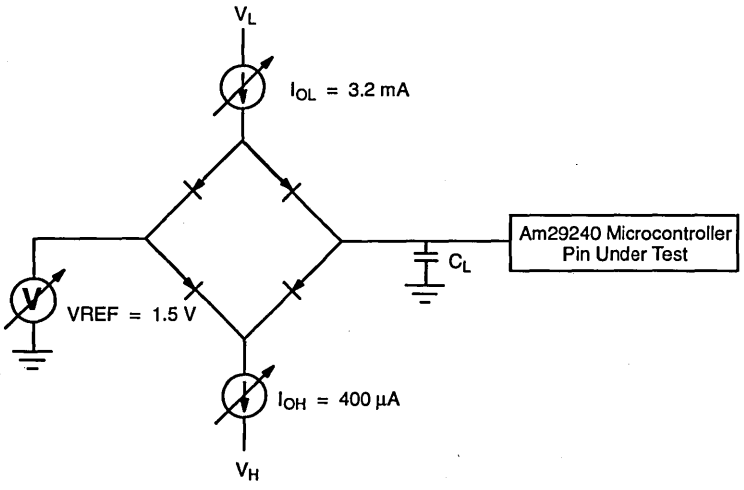


Note: *Applies to ID31–ID0 and IDP3–IDP0 for DRAM page-mode accesses only.*

Note: *Video Timing may be relative to VCLK falling edge if CLK = 1.*

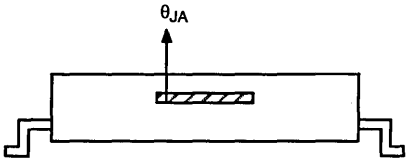
Note: *During AC testing, all inputs are driven at V_{IL} = 0.45 V, V_{IH} = 2.4 V.*

SWITCHING TEST CIRCUIT



THERMAL CHARACTERISTICS

PQFP Package



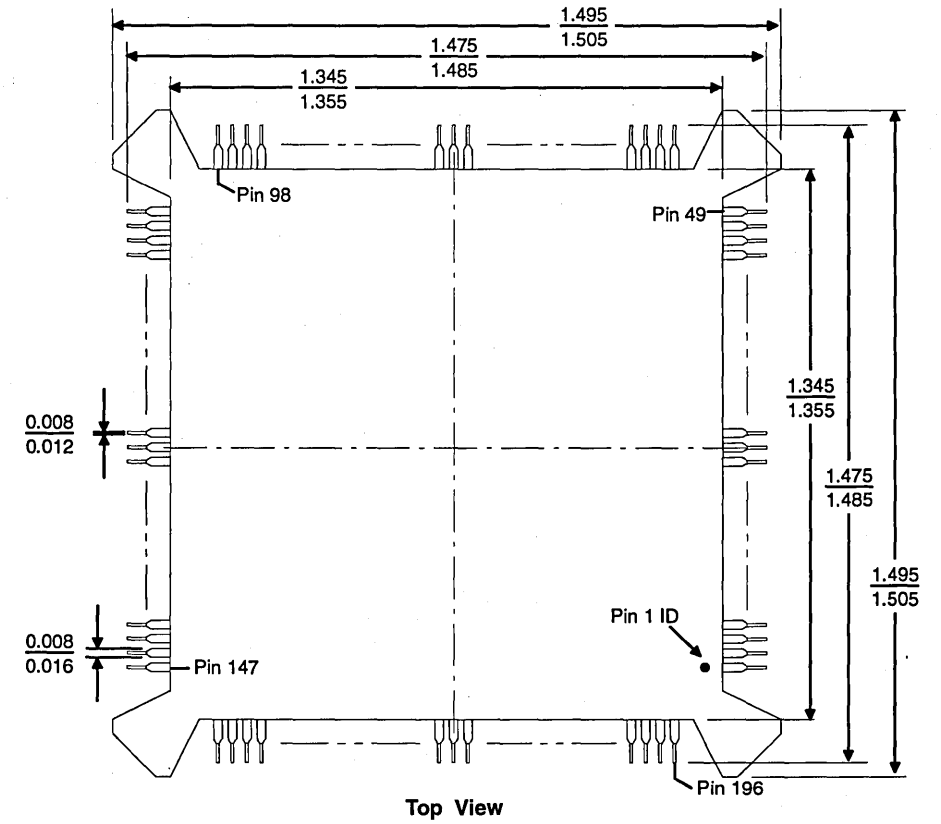
Thermal Resistance – °C/Watt

Parameter	°C/Watt
θ_{JA} Junction-to-Ambient	30
θ_{JC} Junction-to-Case	8
θ_{CA} Case-to-Ambient	22

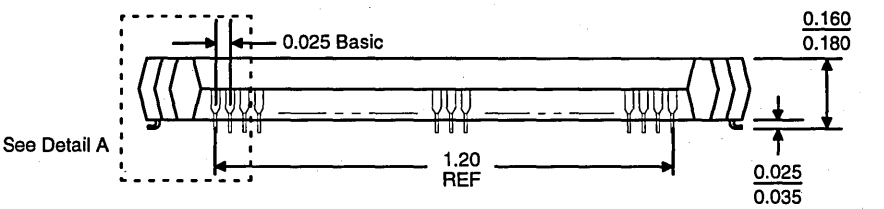
PHYSICAL DIMENSIONS

PQB 196

Plastic Quad Flat Pack; Trimmed and Formed (Measured in inches)



Top View



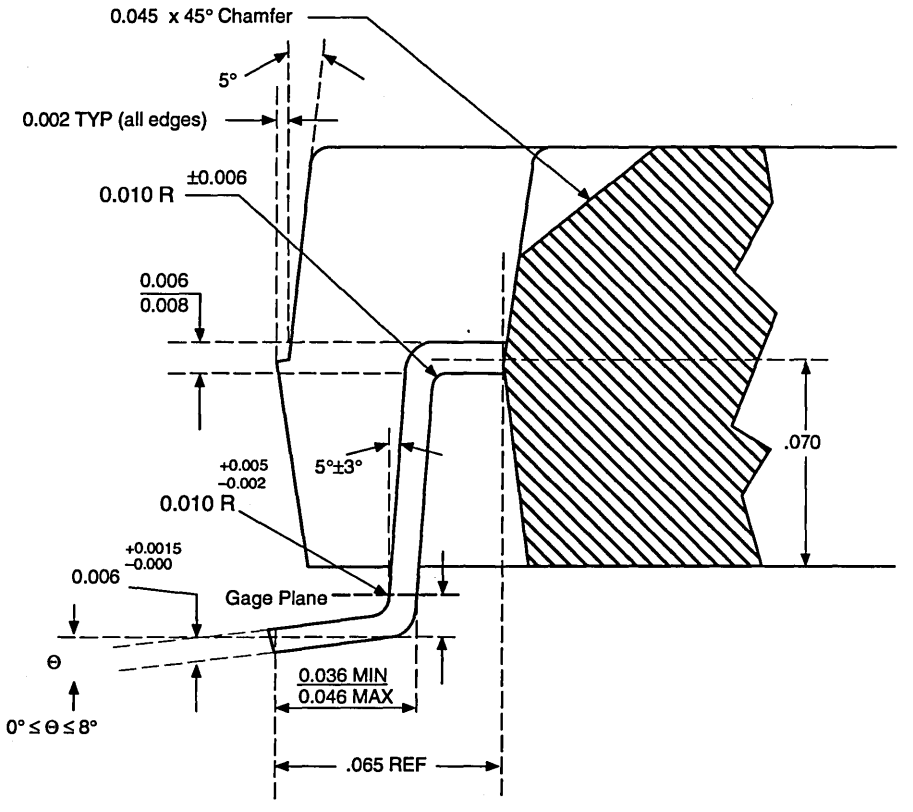
Side View

20012A
CL85
04/9/93 MB

Notes: *For reference only. BSC is an ANSI standard for Basic Space Centering.*

PHYSICAL DIMENSIONS (continued)

PQB 196—Plastic Quad Flat Pack; Trimmed and Formed (continued)



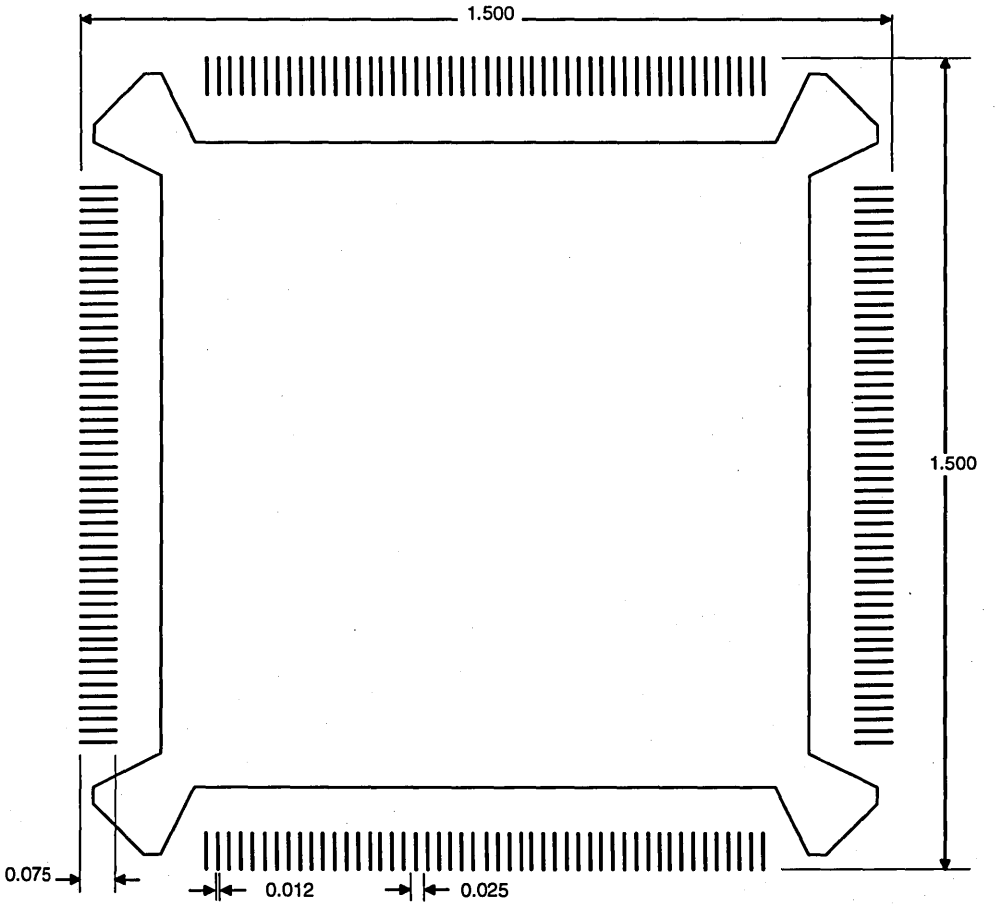
Detail A

Notes:

1. All dimensions are in inches.
2. Dimensions do not include mold protrusion.
3. Coplanarity of all leads will be within 0.004 inches measured from the seating plan. Coplanarity is measured per specification 06–500.
4. Deviation from lead-tip true position shall be within ±0.003 inches.
5. Half span (center of package to lead-tip) shall be within ±0.0085 inches.

PHYSICAL DIMENSIONS (continued)

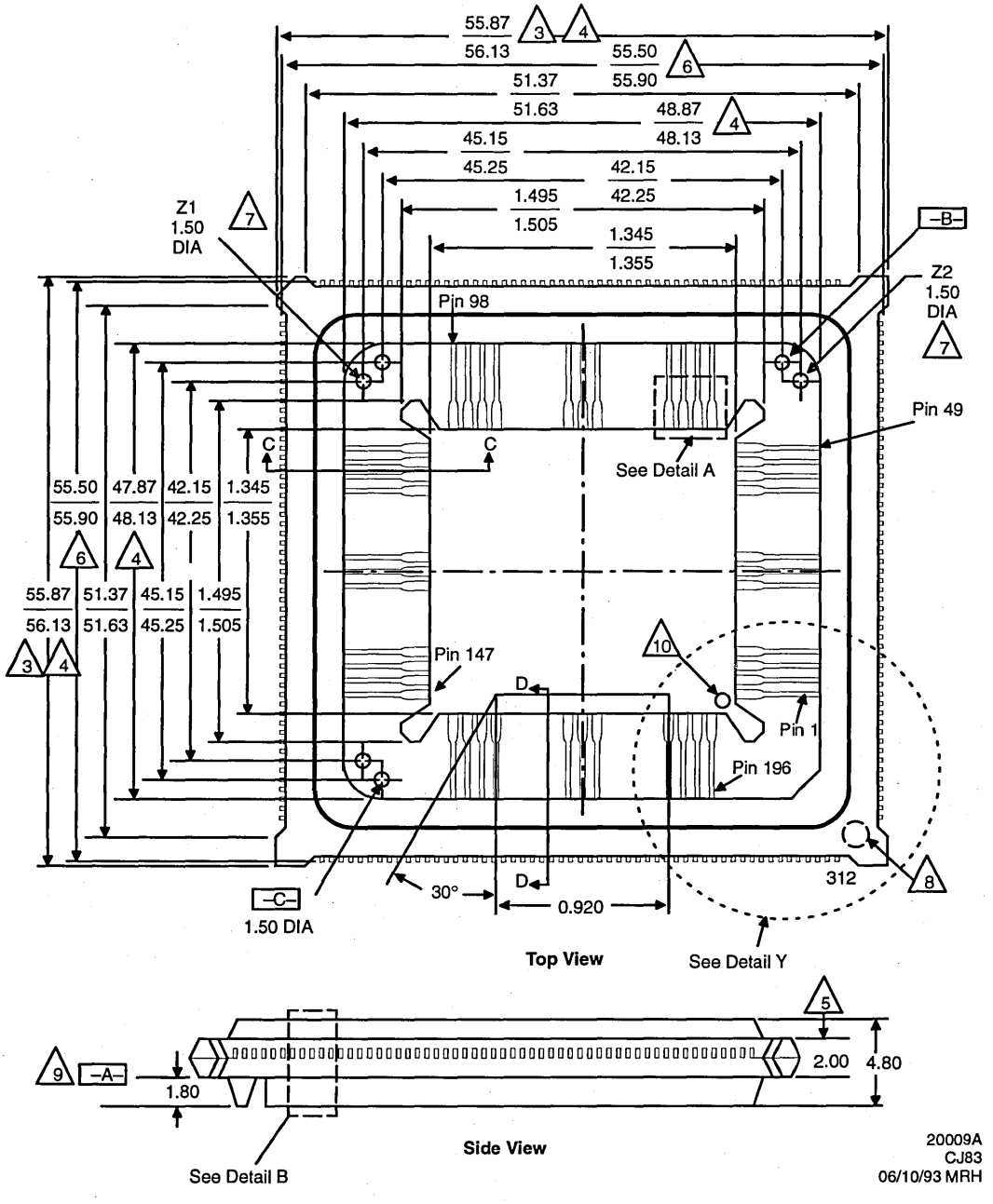
Solder Land Recommendations—196-Lead PQFP



PHYSICAL DIMENSIONS (continued)

PQB 196

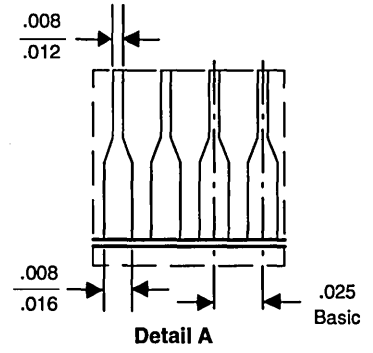
Plastic Quad Flat Pack; Molded Carrier Ring (outer ring measured in millimeters)



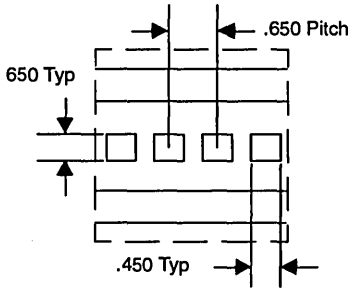
Notes: *For reference only.*

PHYSICAL DIMENSIONS (continued)

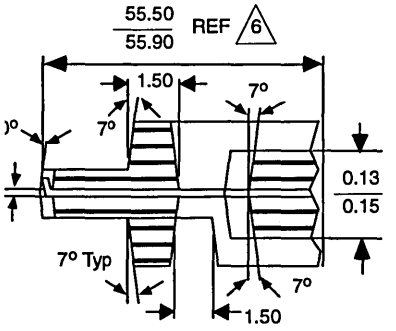
PQB 196—Plastic Quad Flat Pack with Molded Carrier Ring (continued)



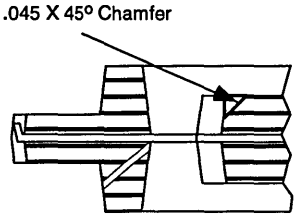
Detail A



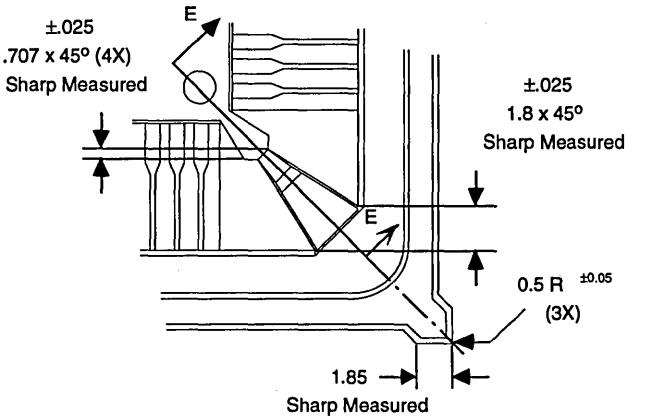
Detail B



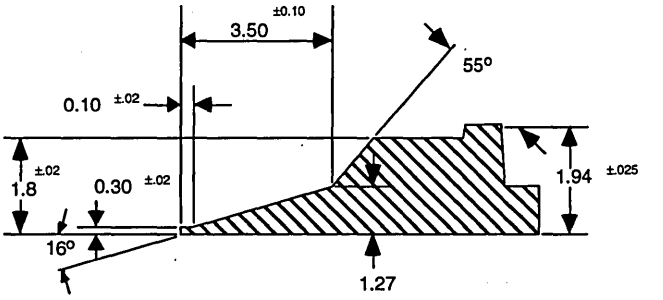
Section C–C



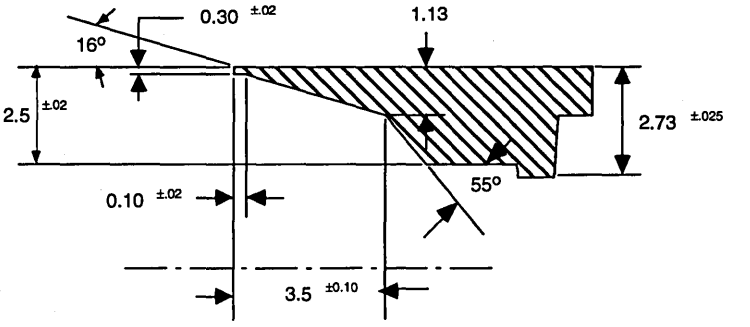
Section D–D



Detail Y



Top Gate



Bottom Gate

Section E–E

PHYSICAL DIMENSIONS (continued)

PQB 196—Plastic Quad Flat Pack with Molded Carrier Ring (continued)

Notes:

1. All dimensions and tolerances conform to ANSI Y14.5M–1982.
2. Controlling dimensions: package is measured in inches and ring is measured in millimeters.
3. These dimensions do not include mold protrusion. Allowable mold protrusion is 0.2 mm per side.
4. These dimensions include mold mismatch and are measured at the parting line.
5. Dimensions are centered about centerline of lead material.
6. These dimensions are from the outside edge to the outside edge of the test points.
7. There are six locating holes in the ring. –B– and –C– datum holes are used for trim form and excise of the molded package only. Holes Z1 and Z2 are used for electrical testing only.
8. This area is reserved for vacuum pickup on each of the four corners of the ring and must be flat within 0.025 mm. No ejector pins in this area.
9. Datum –A– surface for seating in socket applications.
10. Pin one orientation with respect to carrier ring as indicated.

INDEX

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Sales Offices

North American

ALABAMA ..... (205) 882-9122
ARIZONA ..... (602) 242-4400
CALIFORNIA,
- Culver City ..... (310) 645-1524
- Newport Beach ..... (714) 752-6262
- Sacramento(Roseville) ..... (916) 786-6700
- San Diego ..... (619) 560-7030
- San Jose ..... (408) 452-0500
- Woodland Hills ..... (818) 878-9988

CANADA, Ontario,
- Kanata ..... (613) 592-0060
- Willowdale ..... (416) 222-7800

COLORADO ..... (303) 741-2900
CONNECTICUT ..... (203) 264-7800
FLORIDA,
- Clearwater ..... (813) 530-9971
- Boca Raton ..... (407) 361-0050
- Orlando (Longwood) ..... (407) 862-9292

GEORGIA ..... (404) 449-7920
IDAHO ..... (208) 377-0393
ILLINOIS,
- Chicago (Itasca) ..... (708) 773-4422
- Naperville ..... (708) 505-9517

MARYLAND ..... (301) 381-3790
MASSACHUSETTS ..... (617) 273-3970
MINNESOTA ..... (612) 938-0001
NEW JERSEY,
- Cherry Hill ..... (609) 662-2900
- Parsippany ..... (201) 299-0002

NEW YORK,
- Brewster ..... (914) 279-8323
- Rochester ..... (716) 425-8050

NORTH CAROLINA
- Charlotte ..... (704) 875- 3091
- Raleigh ..... (919) 878- 8111

OHIO,
- Columbus (Westerville) ..... (614) 891-6455
- Dayton ..... (513) 439-0268

OREGON ..... (503) 245-0080
PENNSYLVANIA ..... (215) 398-8006
TEXAS,
- Austin ..... (512) 346-7830
- Dallas ..... (214) 934-9099
- Houston ..... (713) 376-8084

International

BELGIUM, Antwerpen ..... TEL ..... (03) 248 43 00
FAX ..... (03) 248 46 42
FRANCE, Paris ..... TEL ..... (1) 49-75-10-10
FAX ..... (1) 49-75-10-13
GERMANY,
- Bad Homburg ..... TEL ..... (06172)-24061
- FAX ..... (06172)-23195
- München ..... TEL ..... (089) 45053-0
- FAX ..... (089) 406490

HONG KONG, ... TEL ..... (852) 865-4525
Wanchai FAX ..... (852) 865-4335
ITALY, Milano ..... TEL ..... (02) 3390541
FAX ..... (02) 38103458
JAPAN,
- Atsugi ..... TEL ..... (0462) 29-8460
- FAX ..... (0462) 29-8458
- Kanagawa ..... TEL ..... (0462) 47-2911
- FAX ..... (0462) 47-1729

International (Continued)

Tokyo ..... TEL ..... (03) 3346-7550
FAX ..... (03) 3342-5196
Osaka ..... TEL ..... (06) 243-3250
FAX ..... (06) 243-3253
KOREA, Seoul ..... TEL ..... (82) 2-784-0030
FAX ..... (82) 2-784-8014
LATIN AMERICA,
Ft. Lauderdale ..... TEL ..... (305) 484-8600
FAX ..... (305) 485-9736
SINGAPORE ..... TEL ..... (65) 3481188
FAX ..... (65) 3480161
SWEDEN,
Stockholm area ..... TEL ..... (08) 98 61 80
(Bromma) FAX ..... (08) 98 09 06
TAIWAN, Taipei ..... TEL ..... (886) 2-7153536
FAX ..... (886) 2-7122183
UNITED KINGDOM,
Manchester area ..... TEL ..... (0925) 830380
(Warrington) FAX ..... (0925) 830204
London area ..... TEL ..... (0483) 740440
(Woking) FAX ..... (0483) 756196

North American Representatives

CANADA
Burnaby, B.C. – DAVETEK MARKETING ..... (604) 430-3680
Kanata, Ontario – VITEL ELECTRONICS ..... (613) 592-0060
Mississauga, Ontario – VITEL ELECTRONICS ..... (416) 564-9720
Lachine, Quebec – VITEL ELECTRONICS ..... (514) 636-5951
ILLINOIS
- Skokie – INDUSTRIAL REPRESENTATIVES,INC ..... (708) 967-8430

IOWA
- LORENZ SALES ..... (319) 377-4666

KANSAS
- Merriam – LORENZ SALES ..... (913) 469-1312
- Wichita – LORENZ SALES ..... (316) 721-0500

MICHIGAN
- Holland – COM-TEK SALES, INC ..... (616) 335-8418
- Brighton – COM-TEK SALES, INC ..... (313) 227-0007

MINNESOTA
- Mel Foster Tech. Sales, Inc. ..... (612) 941-9790

MISSOURI
- LORENZ SALES ..... (314) 997-4558

NEBRASKA
- LORENZ SALES ..... (402) 475-4660

NEW MEXICO
- THORSON DESERT STATES ..... (505) 883-4343

NEW YORK
- East Syracuse – NYCOM, INC ..... (315) 437-8343
- Hauppauge – COMPONENT CONSULTANTS, INC ..... (516) 273-5050

OHIO
- Centerville – DOLFUSS ROOT & CO ..... (513) 433-6776
- Columbus – DOLFUSS ROOT & CO ..... (614) 885-4844
- Westlake – DOLFUSS ROOT & CO ..... (216) 899-9370

PENNSYLVANIA
- RUSSELL F. CLARK CO.,INC. ..... (412) 242-9500

PUERTO RICO
- COMP REP ASSOC, INC ..... (809) 746-6550

UTAH
- Front Range Marketing ..... (801) 288-2500

WASHINGTON
- ELECTRA TECHNICAL SALES ..... (206) 821-7442

WISCONSIN
- Brookfield – INDUSTRIAL REPRESENTATIVES,INC ..... (414) 574-9393

Advanced Micro Devices, Inc. 901 Thompson Place, P.O. Box 3453, Sunnyvale, CA 94088, USA
Tel: (408) 732-2400 • TWX: 910-339-9280 • TELEX: 34-6306 • TOLL FREE: (800) 538-8450
APPLICATIONS HOTLINE & LITERATURE ORDERING • TOLL FREE: (800) 222-9323 • (408) 749-5703

17741C 5/28/93
Ban-21M-6/93-0 Printed in USA

ADVANCED
MICRO
DEVICES, INC.
901 Thompson Place
P.O. Box 3453
Sunnyvale
California 94088-3453
(408) 732-2400
TWX: 910-339-9280
TELEX: 34-6306
TOLL-FREE (800) 538-8450

CORPORATE
APPLICATIONS
HOTLINE
(800) 222-9323
(408) 749-5703

29K LITERATURE
ORDERING
(800) 292-9263
(512) 462-5651

29K SUPPORT PRODUCTS
ENGINEERING HOTLINE
USA (800) 2929-AMD
UK 011-44-256463346
JAPAN 0-031-11-1129

RECYCLED &
RECYCLABLE

Printed in USA
Ban-21M-6/93-0
17741C