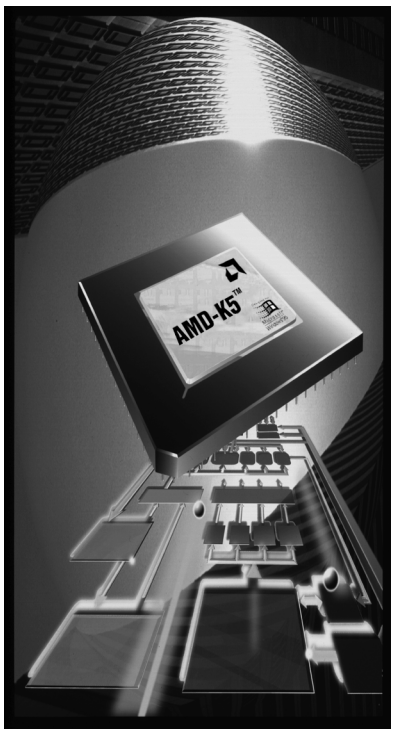
AMD



AMD-K5™

PROCESSOR

Software Development Guide

Publication # **20007** Rev: **D** Amendment/**0**
Issue Date: **September 1996**

Trademarks

AMD, the AMD logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc.
Am486 is a registered trademark, and AMD-K5 is a trademark of Advanced Micro Devices, Inc.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Contents

AMD-K5™ Processor x86 Architecture Extensions

Code Optimization for the AMD-K5 Processor

AMD-K5 Processor Initialization

AMD-K5 Processor Test and Debug

Appendix A Cache

List of Tables

List of Figures

1

AMD-K5™ Processor x86 Architecture Extensions

The AMD-K5™ processor is compatible with the instruction set, programming model, memory management mechanisms, and other software infrastructure supported by the 486 and Pentium (735\90, 815\100) processors. Operating system and application software that runs on the Pentium processor can be executed on the AMD-K5 processor without modification. Because the AMD-K5 processor takes a significantly different approach to implementing the x86 architecture, some subtle differences from the Pentium processor may be visible to system and code developers. These differences are described in Appendix A of the *AMD-K5 Processor Technical Reference Manual,* order# 18524.

Call AMD at 1-800-222-9232 to order AMD-K5 processor support documents.

Before implementing the AMD-K5 processor model-specific features, check CPUID for supported feature flags. See "CPUID" on page 29 for more information.

Additions to the EFLAGS Register

The EFLAGS register on the AMD-K5 processor defines new bits in the upper 16 bits of the register to support extensions to the operating modes. See "Virtual-8086 Mode Extensions (VME)" on page 12 and "CPUID" on page 29 for additional information.

Control Register 4 (CR4) Extensions

Control Register 4 (CR4) was added on the AMD-K5 processor. The bits in this register control the various architectural extensions. The majority of the bits are reserved. The default state of CR4 is all zeros. Figure 1-1 shows the register and describes the bits. The architectural extensions are described in Table 1-1.

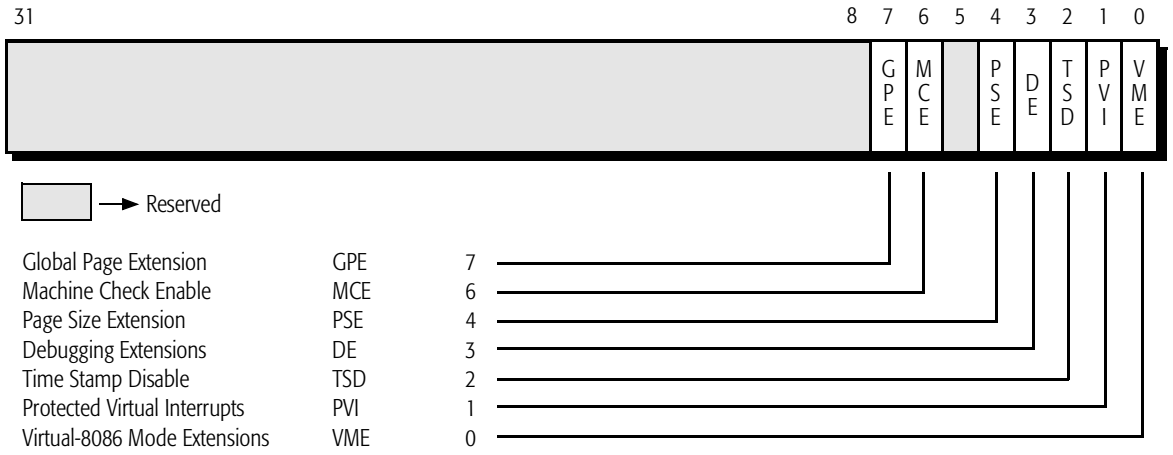


Figure 1-1. Control Register 4 (CR4)

Table 1-1A. Control Register 4 (CR4) Fields

Bit	Mnemonic	Description	Function
7	GPE	Global Page Extension	Enables retention of designated entries in the 4-Kbyte TLB or 4-Mbyte TLB during invalidations. 1 = enabled, 0 = disabled. See "Global Pages" on page 8 for details.
6	MCE	Machine-Check Enable	Enables machine-check exceptions. 1 = enabled, 0 = disabled. See "Machine-Check Exceptions" on page 4 for details.
4	PSE	Page Size Extension	Enables 4-Mbyte pages. 1 = enabled, 0 = disabled. See "4-Mbyte Pages" on page 4 for details.
3	DE	Debugging Extensions	Enables I/O breakpoints in the DR7–DR0 registers. 1 = enabled, 0 = disabled. See "Debug Registers" on page 84 for details.
2	TSD	Time Stamp Disable	Selects privileged (CPL=0) or non-privileged (CPL>0) use of the RDTSC instruction, which reads the Time Stamp Counter (TSC). 1 = CPL must be 0, 0 =any CPL. See "Time Stamp Counter (TSC)" on page 27 for details.
1	PVI	Protected Virtual Interrupts	Enables hardware support for interrupt virtualization in Protected mode. 1 = enabled, 0 = disabled. See "Protected Virtual Interrupt (PVI) Extensions" on page 24 for details.
0	VME	Virtual-8086 Mode Extensions	Enables hardware support for interrupt virtualization in Virtual-8086 mode. 1 = enabled, 0 = disabled. See "Virtual-8086 Mode Extensions (VME)" on page 12 for details.

Machine-Check Exceptions

Bit 6 in CR4, the machine-check enable (MCE) bit, controls generation of machine-check exceptions (12h). If enabled by the MCE bit, these exceptions are generated when either of the following occurs:

- System logic asserts $\overline{\text{BUSCHK}}$ to identify a parity or other type of bus-cycle error
- The processor asserts $\overline{\text{PCHK}}$ while system logic asserts $\overline{\text{PEN}}$ to identify an enabled parity error on the D63–D0 data bus

Whether or not machine-check exceptions are enabled, the processor does the following when either type of bus error occurs:

- Latches the physical address of the failed cycle in its 64-bit machine-check address register (MCAR)
- Latches the cycle definition of the failed cycle in its 64-bit machine-check type register (MCTR)

Software can read the MCAR and MCTR registers in the exception handling routine with the RDMSR instruction, as described on page 34. The format of the registers is shown in Figure 1-8 and Figure 1-9.

If system software has cleared the MCE bit in CR4 to 0 before a bus-cycle error, the processor attempts to continue execution without generating a machine-check exception. It still latches the address and cycle type in MCAR and MCTR as described in this section.

4-Mbyte Pages

The TLBs in the 486 and 386 processors support only 4-Kbyte pages. However, large data structures such as a video frame buffer or non-paged operating system code can consume many pages and easily overrun the TLB. The AMD-K5 processor accommodates large data structures by allowing the operating system to specify 4-Mbyte pages as well as 4-Kbyte pages, and by implementing a four-entry, fully-associative 4-Mbyte TLB which is separate from the 128-entry, 4-Kbyte TLB. From a given page directory, the processor can access both 4-Kbyte pages and 4-Mbyte pages, and the page sizes can be intermixed

within a page directory. When the Page Size Extension (PSE) bit in CR4 is set, the processor translates linear addresses using either the 4-Kbyte TLB or the 4-Mbyte TLB, depending on the state of the page size (PS) bit in the page-directory entry. Figures 1-2 and 1-3 show how 4-Kbyte and 4-Mbyte page translation work.

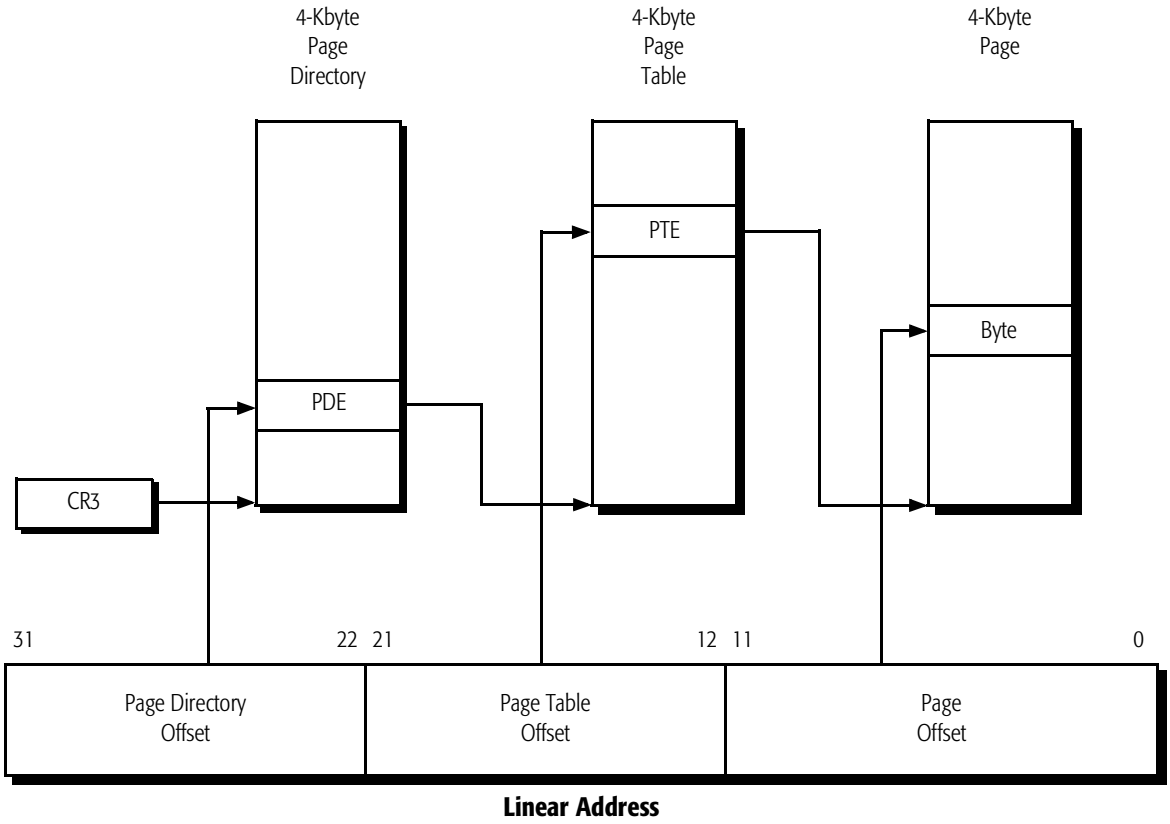


Figure 1-2. 4-Kbyte Paging Mechanism

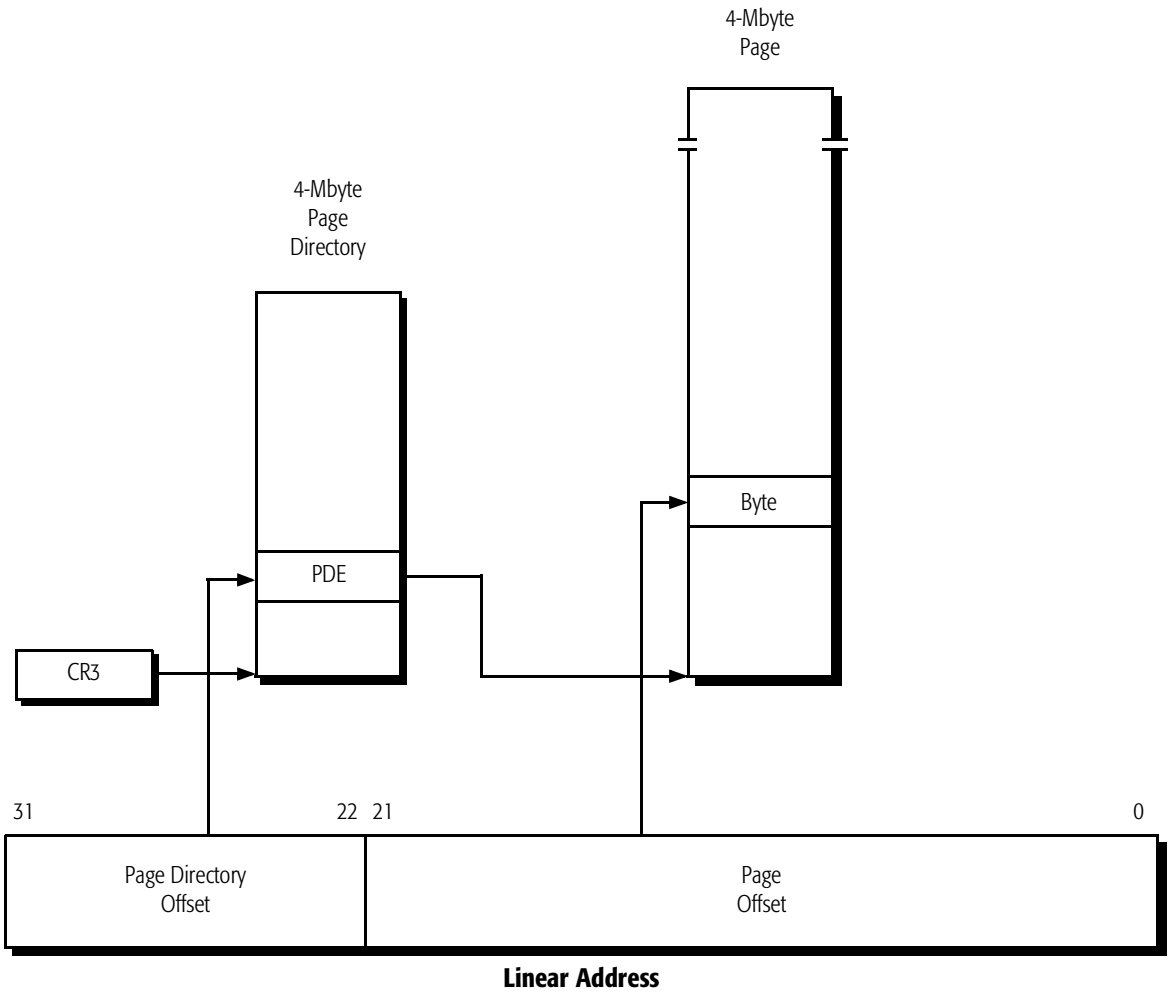


Figure 1-3. 4-Mbyte Paging Mechanism

To enable the 4-Mbyte paging option:

1. Set the Page Size Extension (PSE) bit in CR4 to 1.
2. Set the Page Size (PS) bit in the page-directory entry to 1.
3. Write the physical base addresses of 4-Mbyte pages in bits 31–22 of page-directory entries. (Bits 21–12 of these entries must be cleared to 0 or the processor will generate a page fault.)
4. Load CR3 with the base address of the page directory that contains these page-directory entries.

Figure 1-1 and Table 1-1 show the fields in CR4. Figure 1-4 and Table 1-2 show the fields in a page-directory entry.

4-Kbyte page translation differs from 4-Mbyte page translation in the following ways:

- *4-Kbyte Paging (Figure 1-2)*—Bits 31–22 of the linear address select an entry in a 4-Kbyte page directory in memory, whose physical base address is stored in CR3. Bits 21–12 of the linear address select an entry in a 4-Kbyte page table in memory, whose physical base address is specified by bits 31–22 of the page-directory entry. Bits 11–0 of the linear address select a byte in a 4-Kbyte page, whose physical base address is specified by the page-table entry.
- *4-Mbyte Paging (Figure 1-3)*—Bits 31–22 of the linear address select an entry in a 4-Mbyte page directory in memory, whose physical base address is stored in CR3. Bits 21–0 of the linear address select a byte in a 4-Mbyte page in memory, whose physical base address is specified by bits 31–22 of the page-directory entry. Bits 21–12 of the page-directory entry must be cleared to 0.

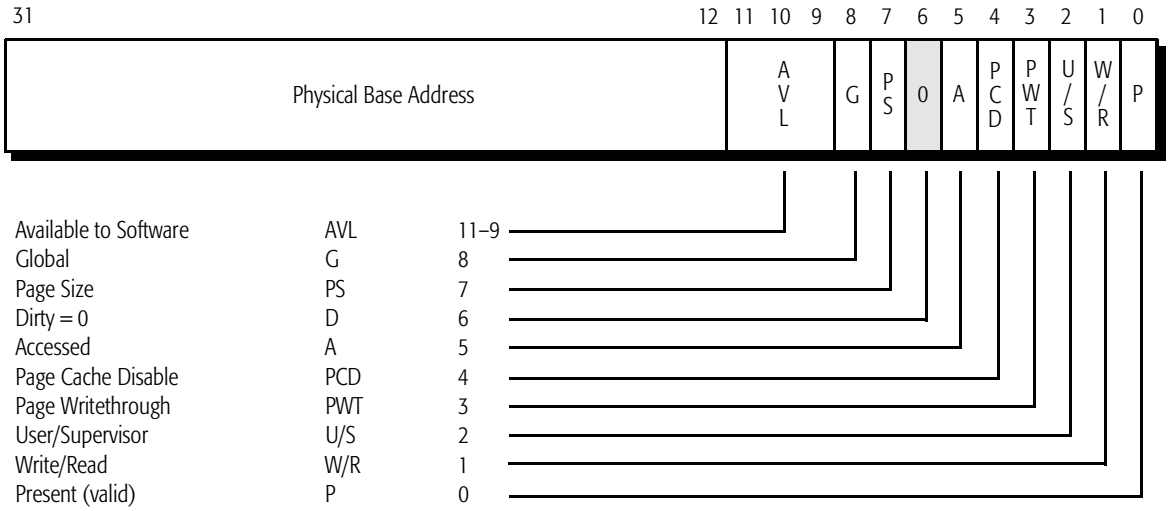


Figure 1-4. Page-Directory Entry (PDE)

Table 1-2A. Page-Directory Entry (PDE) Fields

Bit	Mnemonic	Description	Function
31–12	BASE	Physical Base Address	For 4-Kbyte pages, bits 31–12 contain the physical base address of a 4-Kbyte page table. For 4-Mbyte pages, bits 31–22 contain the physical base address of a 4-Mbyte page and bits 21–12 must be cleared to 0. (The processor will generate a page fault if bits 21–12 are not cleared to 0.)
11–9	AVL	Available to Software	Software may use this field to store any type of information. When the page-directory entry is not present (P bit cleared), bits 31–1 become available to software.
8	G	Global	0 = local, 1 = global.
7	PS	Page Size	0 = 4-Kbyte, 1 = 4-Mbyte.
6	D	Dirty	For 4-Kbyte pages, this bit is undefined and ignored. The processor does not change it. 0 = not written, 1 = written. For 4-Mbyte pages, the processor sets this bit to 1 during a write to the page that is mapped by this page-directory entry. 0 = not written, 1 = written.
5	A	Accessed	The processor sets this bit to 1 during a read or write to any page that is mapped by this page-directory entry. 0 = not read or written, 1 = read or written.
4	PCD	Page Cache Disable	Specifies cacheability for all pages mapped by this page-directory entry. Whether a location in a mapped page is actually cached also depends on several other factors. 0 = cacheable page, 1 = non-cacheable.
3	PWT	Page Writethrough	Specifies writeback or writethrough cache protocol for all pages mapped by this page-directory entry. Whether a location in a mapped page is actually cached in a writeback or writethrough state also depends on several other factors. 0 = writeback page, 1 = writethrough page.
2	U/S	User/Supervisor	0 = user (any CPL), 1 = supervisor (CPL < 3).
1	W/R	Write/Read	0 = read or execute, 1 = write, read, or execute.
0	P	Present	0 = not valid, 1 = valid.

Global Pages

The processor's performance can sometimes be improved by making some pages *global* to all tasks and procedures. This can be done for both 4-Kbyte pages and 4-Mbyte pages.

The processor invalidates (flushes) both the 4-Kbyte TLB and the 4-Mbyte TLB whenever CR3 is loaded with the base address of the new task's page directory. The processor loads CR3 automatically during task switches, and the operating system can load CR3 at any other time. Unnecessary invalidation of certain TLB entries can be avoided by specifying those entries as *global* (a global TLB entry references a *global page)*. This improves performance after TLB flushes. Global entries remain in the TLB and need not be reloaded. For example, entries may reference operating system code and data pages that are always required. The processor operates faster if these entries are retained across task switches and procedure calls.

To specify individual pages as *global:*

1. Set the Global Page Extension (GPE) bit in CR4.
2. (Optional) Set the Page Size Extension (PSE) bit in CR4.
3. Set the relevant Global (G) bit for that page:

   For 4-Kbyte pages—Set the G bit in both the page-directory entry (shown in Figure 1-4 and Table 1-2) and the page-table entry (shown in Figure 1-5 and Table 1-3).

   For 4-Mbyte pages—(Optional) After the PSE bit in CR4 is set, set the G bit in the page-directory entry (shown in Figure 1-4 and Table 1-2).
4. Load CR3 with the base address of the page directory.

The INVLPG instruction clears both the V and G bits for the referenced entry. To invalidate all entries, including global-page entries, in both TLBs:

1. Clear the Global Page Extension (GPE) bit in CR4.
2. Load CR3 with the base address of another (or same) page directory.

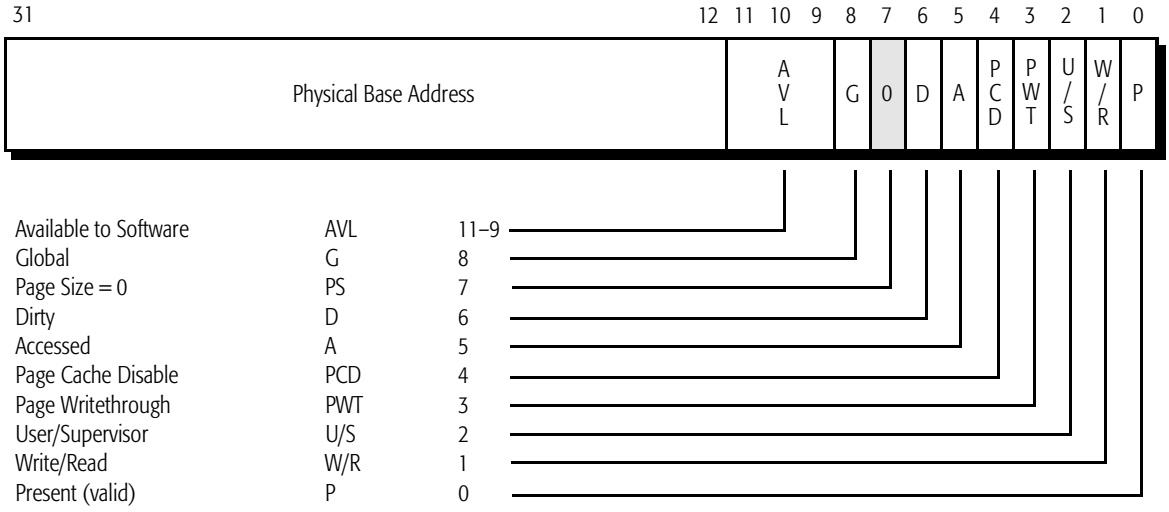


Figure 1-5. Page-Table Entry (PTE)

Table 1-3A. Page-Table Entry (PTE) Fields

Bit	Mnemonic	Description	Function
31–12	BASE	Physical Base Address	The physical base address of a 4-Kbyte page.
11–9	AVL	Available to Software	Software may use the field to store any type of information. When the page-table entry is not present (P bit cleared), bits 31–1 become available to software.
8	G	Global	0 = local, 1 = global.
7	PS	Page Size	This bit is ignored in page-table entries, although clearing it to 0 preserves consistent usage of this bit between page-table and page-directory entries.
6	D	Dirty	The processor sets this bit to 1 during a write to the page that is mapped by this page-table entry. 0 = not written, 1 = written.
5	A	Accessed	The processor sets this bit to 1 during a read or write to any page that is mapped by this page-table entry. 0 = not read or written, 1 = read or written.
4	PCD	Page Cache Disable	Specifies cacheability for all locations in the page mapped by this page-table entry. Whether a location is actually cached also depends on several other factors. 0 = cacheable page, 1 = non-cacheable.
3	PWT	Page Writethrough	Specifies writeback or writethrough cache protocol for all locations in the page mapped by this page-table entry. Whether a location is actually cached in a writeback or writethrough state also depends on several other factors. 0 = writeback, 1 = writethrough.
2	U/S	User/Supervisor	0 = user (any CPL), 1 = supervisor (CPL < 3).
1	W/R	Write/Read	0 = read or execute, 1 = write, read, or execute.
0	P	Present	0 = not valid, 1 = valid.

Virtual-8086 Mode Extensions (VME)

The Virtual-8086 Mode Extensions (VME) bit in CR4 (bit 0) enable performance enhancements for 8086 programs running as protected tasks in Virtual-8086 mode. These extensions include:

- Virtualizing maskable external interrupt control and notification via the VIF and VIP bits in EFLAGS
- Selectively intercepting software interrupts (INT*n* instructions) via the Interrupt Redirection Bitmap (IRB) in the Task State Segment (TSS)

Interrupt Redirection in Virtual-8086 Mode Without VME Extensions

8086 programs expect to have full access to the interrupt flag (IF) in the EFLAGS register, which enables maskable external interrupts via the INTR signal. When 8086 programs run in Virtual-8086 mode on a 386 or 486 processor, they run as protected tasks and access to the IF flag must be controlled by the operating system on a task-by-task basis to prevent corruption of system resources.

Without the VME extensions available on the AMD-K5 processor, the operating system controls Virtual-8086 mode access to the IF flag by trapping instructions that can read or write this flag. These instructions include STI, CLI, PUSHF, POPF, INT*n*, and IRET. This method prevents changes to the real IF when the I/O privilege level (IOPL) in EFLAGS is less than 3, the privilege level at which all Virtual-8086 tasks run. The operating system maintains an image of the IF flag for each Virtual-8086 program by emulating the instructions that read or write IF. When an external maskable interrupt occurs, the operating system checks the state of the IF image for the current Virtual-8086 program to determine whether the program is allowing interrupts. If the program has disabled interrupts, the operating system saves the interrupt information until the program attempts to re-enable interrupts.

The overhead for trapping and emulating the instructions that enable and disable interrupts, and the maintenance of virtual interrupt flags for each Virtual-8086 program, can degrade the processor's performance. This performance can be regained by running Virtual-8086 programs with IOPL set to 3, thus allowing changes to the real IF flag from any privilege level, but with a loss in protection.

In addition to these performance problems caused by virtualization of the IF flag in Virtual-8086 mode, software interrupts (those caused by INT*n* instructions that vector through interrupt gates) cannot be masked by the IF flag or virtual copies of the IF flag, these flags only affect hardware interrupts. Software interrupts in Virtual-8086 mode are normally directed to the Real mode interrupt vector table (IVT), but it may be desirable to redirect interrupts for certain vectors to the Protected mode interrupt descriptor table (IDT).

The processor's Virtual-8086 mode extensions support both of these cases—hardware (external) interrupts and software interrupts—with mechanisms that preserve high performance without compromising protection. Virtualization of hardware interrupts is supported via the Virtual Interrupt Flag (VIF) and Virtual Interrupt Pending (VIP) flag in the EFLAGS register. Redirection of software interrupts is supported with the Interrupt Redirection Bitmap (IRB) in the TSS of each Virtual-8086 program.

Hardware Interrupts and the VIF and VIP Extensions

When VME extensions are enabled, the IF-modifying instructions that are normally trapped by the operating system are allowed to execute, but they write and read the VIF bit rather than the IF bit in EFLAGS. This leaves maskable interrupts enabled for detection by the operating system. It also indicates to the operating system whether the Virtual-8086 program is able to or expecting to receive interrupts.

When an external interrupt occurs, the processor switches from the Virtual-8086 program to the operating system, in the same manner as on a 386 or 486 processor. If the operating system determines that the interrupt is for the Virtual-8086 program, it checks the state of the VIF bit in the program's EFLAGS image on the stack. If VIF has been set by the processor (during an attempt by the program to set the IF bit), the operating system permits access to the appropriate Virtual-8086 handler via the interrupt vector table (IVT). If VIF has been cleared, the operating system holds the interrupt pending. The operating system can do this by saving appropriate information (such as the interrupt vector), setting the program's VIP flag in the EFLAGS image on the stack, and returning to the interrupted program. When the program subsequently attempts to set IF, the set VIP flag causes the processor to inhibit the instruction and generate a general-

protection exception with error code zero, thereby notifying the operating system that the program is now prepared to accept the interrupt.

Thus, when VME extensions are enabled, the VIF and VIP bits are set and cleared as follows:

- *VIF*—This bit is controlled by the processor and used by the operating system to determine whether an external maskable interrupt should be passed on to the program or held pending. VIF is set and cleared for instructions that can modify IF, and it is cleared during software interrupts through interrupt gates. The original IF value is preserved in the EFLAGS image on the stack.
- *VIP*—This bit is set and cleared by the operating system via the EFLAGS image on the stack. It is set when an interrupt occurs for a Virtual-8086 program who's VIF bit is cleared. The bit is checked by the processor when the program subsequently attempts to set VIF.

Figure 1-6 and Table 1-4 show the VIF and VIP bits in the EFLAGS register. The VME extensions support conventional emulation methods for passing interrupts to Virtual-8086 programs, but they make it possible for the operating system to avoid time-consuming emulation of most instructions that write or read the IF.

The VIF and IF flags only affect the way the operating system deals with hardware interrupts (the INTR signal). Software interrupts are handled like machine-generated exceptions and cannot be masked by real or virtual copies of IF (see "Software Interrupts and the Interrupt Redirection Bitmap (IRB) Extension" on page 20). The VIF and VIP flags only ease the software overhead associated with managing interrupts so that virtual copies of the IF flag do not have to be maintained by the operating system. Instead, each task's TSS holds its own copy of these flags in its EFLAGS image.

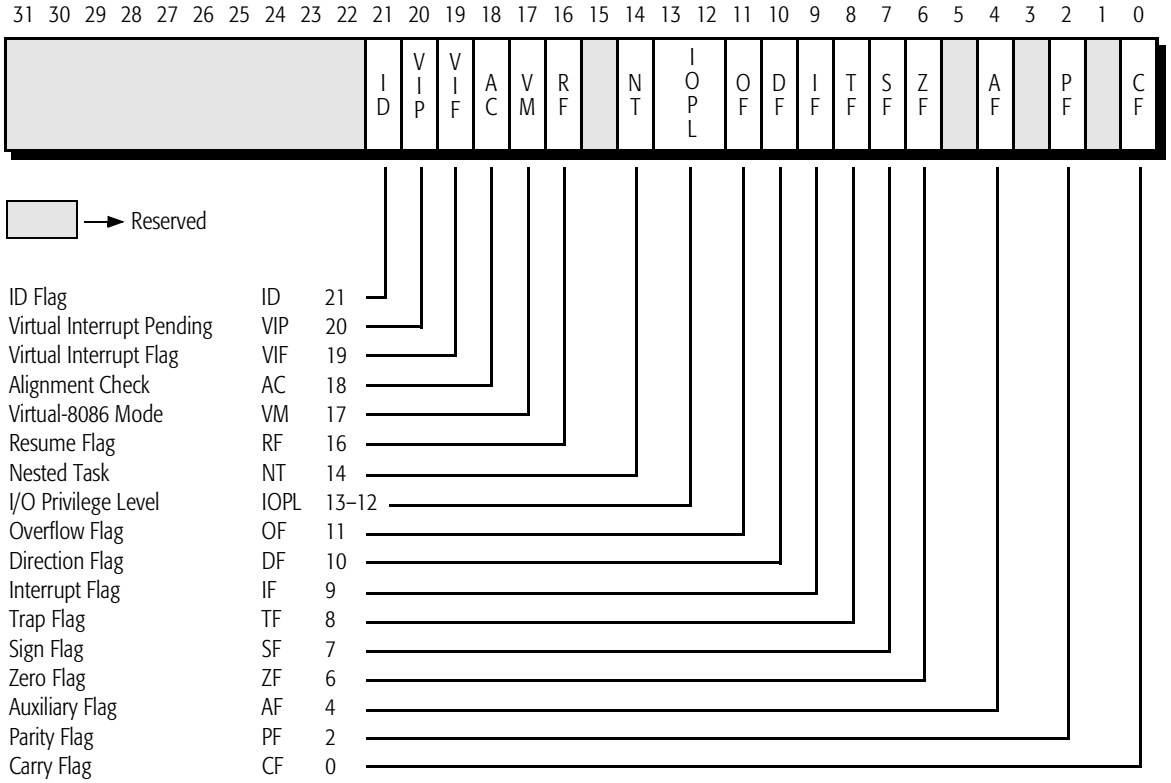


Figure 1-6. EFLAGS Register

Table 1-4A. Virtual-Interrupt Additions to EFLAGS Register

Bit	Mnemonic	Description	Function
20	VIP	Virtual Interrupt Pending	Set by the operating system (via the EFLAGS image on the stack) when an external maskable interrupt (INTR) occurs for a Virtual-8086 program who's VIF bit is cleared. The bit is checked by the processor when the program subsequently attempts to set VIF.
19	VIF	Virtual Interrupt Flag	When the VME bit in CR4 is set, the VIF bit is modified by the processor when a Virtual-8086 program running at less privilege than the IOPL attempts to modify the IF bit. The VIF bit is used by the operating system to determine whether a maskable interrupt should be passed on to the program or held pending.

Table 1-5A through Table 1-5E shows the effects, in various x86-processor modes, of instructions that read or write the IF and VIF flag. The column headings in this table include the following values:

- *PE*—Protection Enable bit in CR0 (bit 0)
- *VM*—Virtual-8086 Mode bit in EFLAGS (bit 17)
- *VME*—Virtual Mode Extensions bit in CR4 (bit 0)
- *PVI*—Protected-mode Virtual Interrupts bit in CR4 (bit 1)
- *IOPL*—I/O Privilege Level bits in EFLAGS (bits 13–12)
- *Handler CPL*—Code Privilege Level of the interrupt handler
- *GP(0)*—General-protection exception, with error code = 0
- *IF*—Interrupt Flag bit in EFLAGS (bit 9)
- *VIF*—Virtual Interrupt Flag bit in EFLAGS (bit 19)

Table 1-5A. Instructions that Modify the IF or VIF Flags—Real Mode

TYPE	PE	VM	VME	PVI	IOPL	GP(0)	IF	VIF
CLI	0	0	0	0	–	No	IF ← 0	–
STI	0	0	0	0	–	No	IF ← 1	–
PUSHF	0	0	0	0	–	No	Pushed	–
POPF	0	0	0	0	–	No	Popped	–
IRET	0	0	0	0	–	No	Popped	–

Notes:
– *Not applicable.*

Table 1-5B. Instructions that Modify the IF or VIF Flags—Protected Mode

TYPE	PE	VM	VME	PVI	IOPL	Handler CPL	GP(0)	IF	VIF
CLI	1	0	–	0	≥ CPL	–	No	IF ← 0	–
CLI	1	0	–	0	< CPL	–	Yes	–	–
STI	1	0	–	0	≥ CPL	–	No	IF ← 1	–
STI	1	0	–	0	< CPL	–	Yes	–	–
PUSHF	1	0	–	0	≥ CPL	–	No	Pushed	–
PUSHF	1	0	–	0	< CPL	–	No	Pushed	–
PUSHFD	1	0	–	0	≥ CPL	–	No	Pushed	Pushed
PUSHFD	1	0	–	0	< CPL	–	No	Pushed	Pushed
POPF	1	0	–	0	≥ CPL	–	No	Popped	–
POPF	1	0	–	0	< CPL	–	No	Not Popped	–
POPFD	1	0	–	0	≥ CPL	–	No	Popped	Not Popped
POPFD	1	0	–	0	< CPL	–	No	Not Popped	Not Popped
IRET	1	0	–	0	–	= 0	No	Popped	–
IRET	1	0	–	0	≥ CPL	> 0	No[1]	Popped	–
IRET	1	0	–	0	< CPL	> 0	No[1]	Not Popped	–
IRETD	1	0	–	0	–	= 0	No	Popped	Popped
IRETD	1	0	–	0	≥ CPL	> 0	No[1]	Popped	Not Popped
IRETD	1	0	–	0	< CPL	> 0	No[1]	Not Popped	Not Popped

Notes:

1. *GP(0) if the CPL of the task executing IRETD is greater than the CPL of the task returned to.*

– *Not applicable.*

Table 1-5C. Instructions that Modify the IF or VIF Flags—Virtual-8086 Mode

TYPE	PE	VM	VME	PVI	IOPL	GP(0)	IF	VIF
CLI	1	1	0	–	3	No	IF ← 0	No Change
CLI	1	1	0	–	< 3	Yes	–	–
STI	1	1	0	–	3	No	IF ← 1	No Change
STI	1	1	0	–	< 3	Yes	–	–
PUSHF	1	1	0	–	3	No	Pushed	–
PUSHF	1	1	0	–	< 3	Yes	–	–
PUSHFD	1	1	0	–	3	No	Pushed	Pushed
PUSHFD	1	1	0	–	< 3	Yes	–	–
POPF	1	1	0	–	3	No	Popped	–
POPF	1	1	0	–	< 3	Yes	–	–
POPFD	1	1	0	–	3	No	Popped	Not Popped
POPFD	1	1	0	–	< 3	Yes	–	–
IRETD[2]	1	1	0	–	–	No	Popped	Popped

Notes:

1. *All Virtual-8086 mode tasks run at CPL = 3.*
2. *All protected virtual interrupt handlers run at CPL = 0.*

– *Not applicable.*

Table 1-5D. Instructions that Modify the IF or VIF Flags—Virtual-8086 Mode Interrupt Extensions (VME)[1]

TYPE	PE	VM	VME	PVI	IOPL	GP(0)	IF	VIF
CLI	1	1	1	–	3	No	IF ← 0	No Change
CLI	1	1	1	–	< 3	No	No Change	VIF ← 0
STI	1	1	1	–	3	No	IF ← 1	No Change
STI	1	1	1	–	< 3	No[3]	No Change	VIF ← 1
PUSHF	1	1	1	–	3	No	Pushed	Not Pushed
PUSHF	1	1	1	–	< 3	No	Not Pushed	Pushed into IF
PUSHFD	1	1	1	–	3	No	Pushed	Pushed
PUSHFD	1	1	1	–	< 3	Yes	–	–
POPF	1	1	1	–	3	No	Popped	Not Popped
POPF	1	1	1	–	< 3	No	Not Popped	Popped from IF
POPFD	1	1	1	–	3	No	Popped	Not Popped
POPFD	1	1	1	–	< 3	Yes	–	–
IRET from V86 Mode	1	1	1	–	3	No	Popped	Not Popped
IRET from V86 Mode	1	1	1	–	< 3	No[3]	Not Popped	Popped from IF
IRETD from V86 Mode	1	1	1	–	3	No	Popped	Not Popped
IRETD from V86 Mode	1	1	1	–	< 3	Yes	–	–
IRETD from Protected Mode[2]	1	1	1	–	–	No[3]	Popped	Popped

Notes:

1. *All Virtual-8086 mode tasks run at CPL = 3.*
2. *All protected virtual interrupt handlers run at CPL = 0.*
3. *GP(0) if an attempt is made to set VIF when VIP = 1.*

– *Not applicable.*

Table 1-5E. Instructions that Modify the IF or VIF Flags—Protected Mode Virtual Interrupt Extensions (PVI)[1]

TYPE	PE	VM	VME	PVI	IOPL	GP(0)	IF	VIF
CLI	1	0	–	1	3	No	IF ← 0	No Change
CLI	1	0	–	1	< 3	No	No Change	VIF ← 0
STI	1	0	–	1	3	No	IF ← 1	No Change
STI	1	0	–	1	< 3	No[3]	No Change	VIF ← 1
PUSHF	1	0	–	1	3	No	Pushed	Not Pushed
PUSHF	1	0	–	1	< 3	No	Pushed	Not Pushed
PUSHFD	1	0	–	1	3	No	Pushed	Pushed
PUSHFD	1	0	–	1	< 3	No	Pushed	Pushed
POPF	1	0	–	1	3	No	Popped	Not Popped
POPF	1	0	–	1	< 3	No	Not Popped	Not Popped
POPFD	1	0	–	1	3	No	Popped	Not Popped
POPFD	1	0	–	1	< 3	No	Not Popped	Not Popped
IRETD[2]	1	0	–	1	–	No[3]	Popped	Popped

Notes:

1. *All Protected mode virtual interrupt tasks run at CPL = 3.*
2. *All protected mode virtual interrupt handlers run at CPL = 0.*
3. *GP(0) if an attempt is made to set VIF when VIP = 1.*

– *Not applicable.*

Software Interrupts and the Interrupt Redirection Bitmap (IRB) Extension

In Virtual-8086 mode, software interrupts (INT*n* exceptions that vector through interrupt gates) are trapped by the operating system for emulation, because they would otherwise clear the real IF. When VME extensions are enabled, these INT*n* instructions are allowed to execute normally, vectoring directly to a Virtual-8086 service routine via the Virtual-8086 interrupt vector table (IVT) at address 0 of the task address space. However, it may still be desirable for security or performance reasons to intercept INT*n* instructions on a vector-specific basis to allow servicing by Protected-mode routines accessed through the interrupt descriptor table (IDT). This is accomplished by an Interrupt Redirection Bitmap (IRB) in the TSS, which is created by the operating system in a manner similar to the IO Permission Bitmap (IOPB) in the TSS.

Figure 1-7 shows the format of the TSS, with the Interrupt Redirection Bitmap near the top. The IRB contains 256 bits, one for each possible software-interrupt vector. The most-

significant bit of the IRB is located immediately below the base of the IOPB. This bit controls interrupt vector 255. The least-significant bit of the IRB controls interrupt vector 0.

The bits in the IRB work as follows:

- *Set*—If set to 1, the INT*n* instruction behaves as if the VME extensions are not enabled. The interrupt vectors to a Protected-mode routine if IOPL = 3, or it causes a general-protection exception with error code zero if IOPL<3.
- *Cleared*—If cleared to 0, the INT*n* instruction vectors directly to the corresponding Virtual-8086 service routine via the Virtual-8086 program's IVT.

Only software interrupts can be redirected via the IRB to a Real mode IVT—hardware interrupts cannot. Hardware interrupts are asynchronous events and do not belong to any current virtual task. The processor thus has no way of deciding which IVT (for which Virtual-8086 program) to direct a hardware interrupt to. Because of this, hardware interrupts always require operating system intervention. The VIF and VIP bits described in "Hardware Interrupts and the VIF and VIP Extensions" on page 13 are provided to assist the operating system in this intervention.

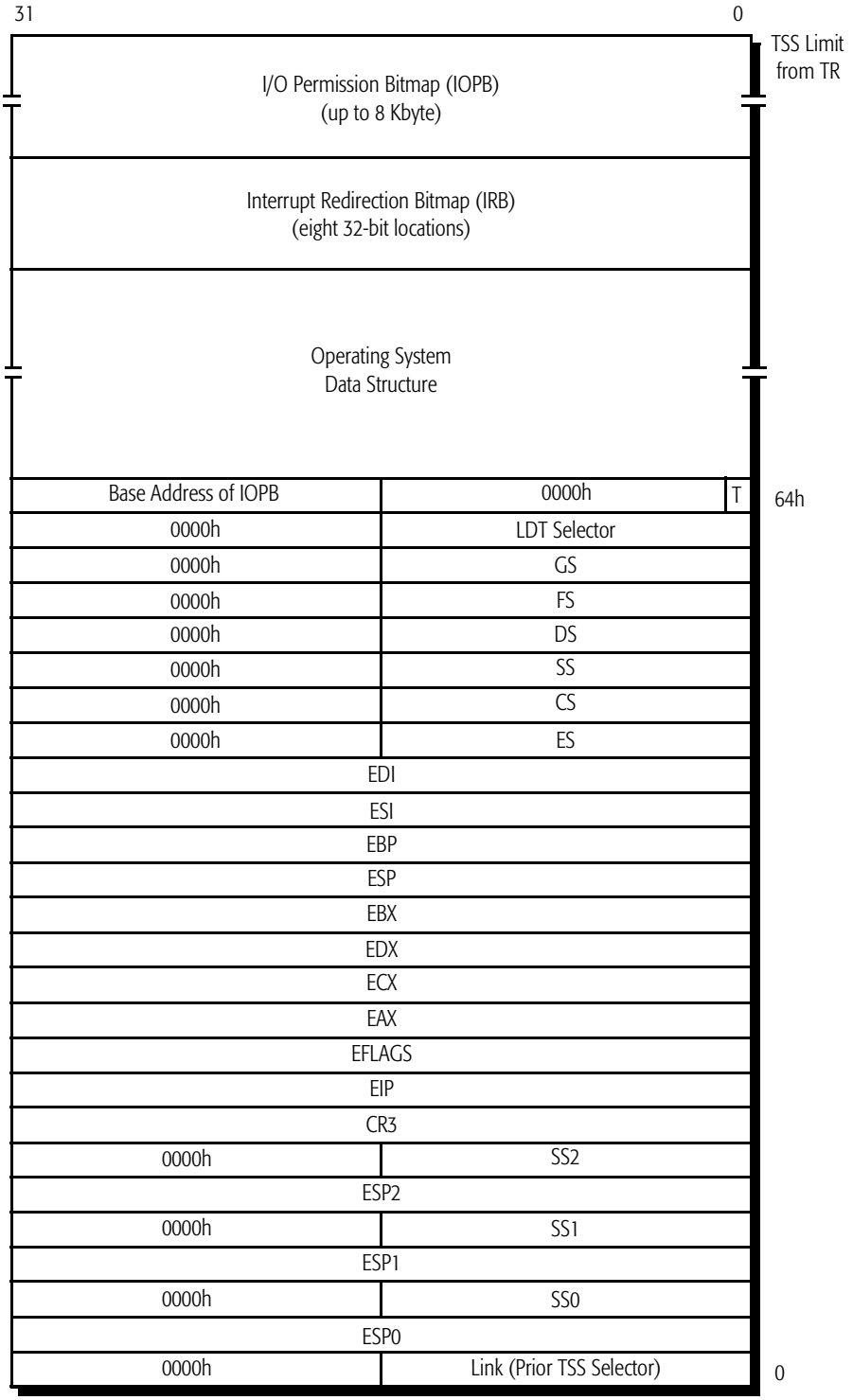


Figure 1-7. Task State Segment (TSS)

Table 1-6 compares the behavior of hardware and software interrupts in various x86-processor operating modes. It also shows which interrupt table is accessed: the Protected-mode IDT or the Real- and Virtual-8086-mode IVT. The column headings in this table include:

- *PE*—Protection Enable bit in CR0 (bit 0)
- *VM*—Virtual-8086 Mode bit in EFLAGS (bit 17)
- *VME*—Virtual Mode Extensions bit in CR4 (bit 0)
- *PVI*—Protected-Mode Virtual Interrupts bit in CR4 (bit 1)
- *IOPL*—I/O Privilege Level bits in EFLAGS (bits 13–12)
- *IRB*—Interrupt Redirection Bit for a task, from the Interrupt Redirection Bitmap (IRB) in the tasks TSS
- *GP(0)*—General-protection exception, with error code = 0
- *IDT*—Protected-Mode Interrupt Descriptor Table
- *IVT*—Real- and Virtual-8086 Mode Interrupt Vector Table

Table 1-6A. Interrupt Behavior and Interrupt-Table Access

Mode	Interrupt Type	PE	VM	VME	PVI	IOPL	IRB	GP(0)	IDT	IVT
Real mode	Software	0	0	0	–	0	–	–	–	✓
	Hardware	0	0	0	–	0	–	–	–	✓
Protected mode	Software	1	0	0	–	–	–	–	✓	–
	Hardware	1	0	0	–	–	–	–	✓	–
Virtual-8086 mode[1]	Software	1	1	0	–	= 3	–	No	✓	–
	Software	1	1	0	–	< 3	–	Yes	✓	–
	Hardware	1	1	0	–	–	–	No	✓	–
Virtual-8086 Mode Extensions (VME)[1]	Software	1	1	1	0	–	0	No	–	✓
	Software	1	1	1	0	= 3	1	No	✓	–
	Software	1	1	1	0	< 3	1	Yes	✓	–
	Hardware	1	1	1	0	–	–	No	✓	–
Protected Virtual Extensions (PVI)	Software	1	0	1	1	–	–	No	✓	–
	Hardware	1	0	1	1	–	–	No	✓	–

Notes:

1. *All Virtual-8086 tasks run at CPL = 3.*

– *Not applicable.*

Protected Virtual Interrupt (PVI) Extensions

The Protected Virtual Interrupts (PVI) bit in CR4 enables support for interrupt virtualization in Protected mode. In this virtualization, the processor maintains program-specific VIF and VIP flags in a manner similar to those in Virtual-8086 Mode Extensions (VME). When a program is executed at CPL = 3, it can set and clear its copy of the VIF flag without causing general-protection exceptions.

The only differences between the VME and PVI extensions are that, in PVI, selective INT*n* interception using the Interrupt Redirection Bitmap in the TSS does not apply, and only the STI and CLI instructions are affected by the extension.

Table 1-5A through Table 1-5E and Table 1-6 show, among other things, the behavior of hardware and software interrupts, and instructions that affect interrupts, in Protected mode with the PVI extensions enabled.

Model-Specific Registers (MSRs)

The processor supports model-specific registers (MSRs) that can be accessed with the RDMSR and WRMSR instructions when CPL = 0. The following index values in the ECX register access specific MSRs:

- *00h:* Machine-Check Address Register (MCAR)
- *01h:* Machine-Check Type Register (MCTR)
- *10h*: Time Stamp Counter (TSC)
- *82h*: Array Access Register (AAR)
- *83h*: Hardware Configuration Register (HWCR)

The RDMSR and WRMSR instructions are described on page 34. The following sections describe the format of the registers.

Machine-Check Address Register (MCAR)

The processor latches the address of the current bus cycle in its 64-bit Machine-Check Address Register (MCAR) when a bus-cycle error occurs. These errors are indicated either by (a) system logic asserting $\overline{\text{BUSCHK}}$, or (b) the processor asserting $\overline{\text{PCHK}}$ while system logic asserts $\overline{\text{PEN}}$.

The MCAR can be read with the RDMSR instruction when the ECX register contains the value 00h. Figure 1-8 shows the format of the MCAR register. The contents of the register can be read with the RDMSR instruction.

If system software has set the MCE bit in CR4 before the bus-cycle error, the processor also generates a machine-check exception as described on page 4.

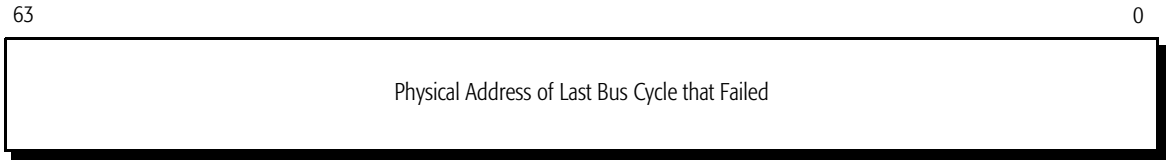


Figure 1-8. Machine-Check Address Register (MCAR)

Machine-Check Type Register (MCTR)

The processor latches the cycle definition and other information about the current bus cycle in its 64-bit Machine-Check Type Register (MTAR) at the same times that the Machine-Check Address Register (MCAR) latches the cycle address: when a bus-cycle error occurs. These errors are indicated either by (a) system logic asserting $\overline{\text{BUSCHK}}$, or (b) the processor asserting $\overline{\text{PCHK}}$ while system logic asserts $\overline{\text{PEN}}$.

The MCTR can be read with the RDMSR instruction when the ECX register contains the value 01h. Figure 1-9 and Table 1-7 show the formats of the MCTR register. The contents of the register can be read with the RDMSR instruction. The processor clears the CHK bit (bit 0) in MCTR when the register is read with the RDMSR instruction.

If system software has set the MCE bit in CR4 before the bus-cycle error, the processor also generates a machine-check exception as described on page 4.

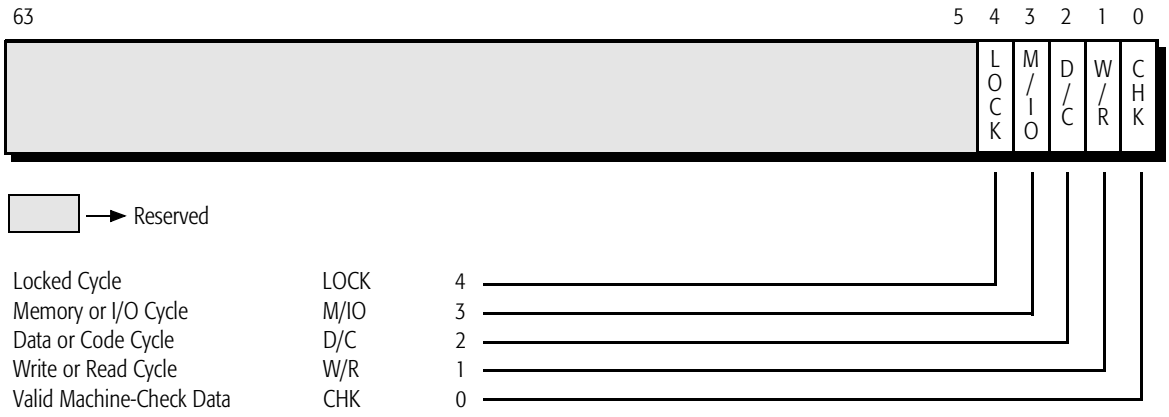


Figure 1-9. Machine-Check Type Register (MCTR)

Table 1-7A. Machine-Check Type Register (MCTR) Fields

Bit	Mnemonic	Description	Function
4	LOCK	Locked Cycle	Set to 1 if the processor was asserting $\overline{\text{LOCK}}$ during the bus cycle.
3	M/$\overline{\text{IO}}$	Memory or I/O	1 = memory cycle, 0 = I/O cycle.
2	D/$\overline{\text{C}}$	Data or Code	1 = data cycle, 0 = code cycle.
1	W/$\overline{\text{R}}$	Write or Read	1 = write cycle, 0 = read cycle.
0	CHK	Valid Machine-Check Data	The processor sets the CHK bit to 1 when both the MCTR and MCAR registers contain valid information. The processor clears the CHK bit to 0 when software reads the MCTR with the RDMSR instruction.

Time Stamp Counter (TSC)

With each processor clock cycle, the processor increments a 64-bit time stamp counter (TSC) model-specific register. The counter can be written or read using the WRMSR or RDMSR instructions when the ECX register contains the value 10h and CPL = 0. The counter can also be read using the RDTSC instruction (see page 33) but the required privilege level for this instruction is determined by the Time Stamp Disable (TSD) bit in CR4. With any of these instructions, the EDX and EAX registers hold the upper and lower double-words (dwords) of the 64-bit value to be written to or read from the TSC, as follows:

- *EDX*—Upper 32 bits of TSC
- *EAX*—Lower 32 bits of TSC

The TSC can be loaded with any arbitrary value.

Array Access Register (AAR)

The Array Access Register (AAR) contains pointers for testing the tag and data arrays for the instruction cache, data cache, 4-Kbyte TLB, and 4-Mbyte TLB. The AAR can be written or read with the WRMSR or RDMSR instruction when the ECX register contains the value 82h.

For details on the AAR, see “Cache and TLB Testing” on page 75.

Hardware Configuration Register (HWCR)

The Hardware Configuration Register (HWCR) contains configuration bits that control miscellaneous debugging functions. The HWCR can be written or read with the WRMSR or RDMSR instruction when the ECX register contains the value 83h.

For details on the HWCR, see "Hardware Configuration Register (HWCR)" on page 71.

New Instructions

In addition to supporting all the 486 processor instructions, the AMD-K5 processor implements the following instructions:

- CPUID
- CMPXCHG8B
- MOV to and from CR4
- RDTSC
- RDMSR
- WRMSR
- RSM
- Illegal instruction (Reserved opcode)

CPUID

mnemonic	*opcode*	*description*
CPUID	0F A2h	Identify processor

Privilege: Any level
Registers Affected: EAX, EBX, ECX, EDX
Flags Affected: none
Exceptions Generated: Real, Virtual-8086 mode—none
Protected mode—none

The CPUID instruction identifies the type of processor and the features it supports. A 0 or 1 value written to the EAX register specifies what information will be returned by the instruction.

The processor implements the ID flag (bit 21) in the EFLAGS register. By writing and reading this bit, software can verify that the processor will execute the CPUID instruction.

For detailed instructions on processor and feature identification see the *AMD Processor Recognition* application note, order# 20734.

Table 1-8 outlines the AMD-K5 processor family codes and model codes with the CPU clock frequencies (MHz), bus frequencies (MHz), and P-Rating strings ("PRxxx").

Table 1-8A. CPU Clock Frequencies, Bus Frequencies, and P-Rating Strings

Family Code	Model Code	CPU Frequency (MHz)	CPU Bus Frequency (MHz)	P-Rating String ("PRxxx")[1]
5	0	75	50	PR75
		90	60	PR90
		100	66	PR100
	1	90	60	PR120
		100	66	PR133
		120	60	PR150
		133	66	PR166

Notes:

1. *The CPUID instruction does not return a P-Rating string.*

– *This table does not constitute product announcements. Instead, the information in the table represents possible product offerings. AMD will announce actual products based on availability and market demand.*

The list below prioritizes the recommended BIOS CPU ID strings. The primary requirement is that if the CPU clock frequency is to be displayed the P-rating *must* also be displayed.

Recommended:

"AMD-K5-PRxxx"	No clock or bus frequency information is displayed.
	OR
"AMD-K5-PRxxx" "yyy MHz" "*zzz Mhz*"	"PRxxx" indicates the P-Rating for the installed K86™ processor. "yyy MHz" indicates the clock frequency of the processor. *"zzz Mhz"* indicates the bus frequency of the processor. Display of the bus frequency is encouraged, but not required.

Acceptable:

"AMD-K5"	The default is recommended if the clock frequency detected is not in the P-Rating table. The actual frequency *should not* be displayed anywhere in the boot-up display.

CMPXCHG8B

mnemonic	*opcode*	*description*
CMPXCHG8B *r/m64*	0F C7h	Compare and exchange 8-byte operand

Privilege:	Any level
Registers Affected:	EAX, EBX, ECX, EDX
Flags Affected:	ZF
Exceptions Generated:	Real, Virtual-8086, Protected mode—GP(0). Invalid opcode if destination is a register. Virtual-8086 mode—Page fault

The CMPXCHG8B instruction is an 8-byte version of the 4-byte CMPXCHG instruction supported by the 486 processor. CMPXCHG8B compares a value from memory with a value in the EDX and EAX register, as follows:

- *EDX*—Upper 32 bits of compare value
- *EAX*—Lower 32 bits of compare value

If the memory value matches the value in EDX and EAX, the ZF flag is set to 1 and the 8-byte value in ECX and EBX is written to the memory location, as follows:

- *ECX*—Upper 32 bits of exchange value
- *EBX*—Lower 32 bits of exchange value

MOV to and from CR4

mnemonic	*opcode*	*description*
MOV CR4,*r32*	0F 22h	Move to CR4 from register
MOV *r32*,CR4	0F 20h	Move to register from CR4

Privilege:	CPL = 0
Registers Affected:	CR4, 32-bit general-purpose register
Flags Affected:	OF, SF, ZF, AF, PF, and CF are undefined
Exceptions Generated:	Real mode—none Virtual-8086 mode—GP(0) Protected mode—GP(0) if CPL not = 0

These instructions read and write control register 4 (CR4).

RDTSC

mnemonic	*opcode*	*description*
RDTSC	0F 31h	Read time stamp counter

Privilege:	Selectable by TSD bit in CR4
Registers Affected:	EAX, EDX
Flags Affected:	none
Exceptions Generated:	Real—none
	Virtual-8086 mode—Invalid Opcode
	Protected mode—GP (0) if CPL not = 0 when CR4.TSD=1

The AMD-K5 processor's 64-bit time stamp counter (TSC) increments on each processor clock. In Real or Protected mode, the counter can be read with the RDMSR instruction and written with the WRMSR instruction when CPL = 0. However, in Protected mode the RDTSC instruction can be used to read the counter at privilege levels higher than CPL = 0.

The required privilege level for using the RDTSC instruction is determined by the Time Stamp Disable (TSD) bit in CR4, as follows:

- *CPL = 0*—Set the TSD bit in CR4 to 1
- *Any CPL*—Clear the TSD bit in CR4 to 0

The RDTSC instruction reads the counter value into the EDX and EAX registers as follows:

- *EDX*—Upper 32 bits of TSC
- *EAX*—Lower 32 bits of TSC

The following example shows how the RDTSC instruction can be used. After this code is executed, EAX and EDX contain the time required to execute the RDTSC instruction.

```
mov ecx,10h       ;Time Stamp Counter Access via MSRs
mov eax,00000000h ;Initialize the Counter to zero
db 0Fh, 30h       ;WRMSR
db 0Fh, 31h       ;RDTSC
db 0Fh, 31h       ;RDTSC
```

RDMSR and WRMSR

mnemonic	*opcode*	*description*
RDMSR	0F 32h	Read model-specific register (MSR)
WRMSR	0F 30h	Write model-specific register (MSR)

Privilege:	CPL=0
Registers Affected:	EAX, ECX, EDX
Flags Affected:	none
Exceptions Generated:	Real—GP(0) for unimplemented MSR address
	Virtual-8086 mode—GP(0)
	Protected mode—GP(0) if CPL not = 0
	Protected mode—GP(0) for unimplemented MSR address

The RDMSR or WRMSR instructions can be used in Real or Protected mode to access several 64-bit, model-specific registers (MSRs). These registers are addressed by the value in ECX, as follows:

- *00h:* Machine-Check Address Register (MCAR). This may contain the physical address of the last bus cycle for which the $\overline{\text{BUSCHK}}$ or $\overline{\text{PCHK}}$ signal was asserted. For details, see "Machine-Check Address Register (MCAR)" on page 25.
- *01h:* Machine-Check Type Register (MCTR). This contains the cycle definition of the last bus cycle for which the $\overline{\text{BUSCHK}}$ or $\overline{\text{PCHK}}$ signal was asserted. For details, see "Machine-Check Type Register (MCTR)" on page 26. The processor clears the CHK bit (bit 0) in MCTR when the register is read with the RDMSR instruction.
- *10h*: Time Stamp Counter (TSC). This contains a time value. The TSC can be initialized to any value with the WRMSR instruction, and it can be read with either the RDMSR or RDTSC instruction. For details, see "Time Stamp Counter (TSC)" on page 27.
- *82h*: Array Access Register (AAR). This contains an array pointer and test data for testing the processor's cache and TLB arrays. For details on the AAR, see "Cache and TLB Testing" on page 75.
- *83h*: Hardware Configuration Register (HWCR). This contains configuration bits that control miscellaneous debugging functions. For details, see "Hardware Configuration Register (HWCR)" on page 71.

The above value in ECX identifies the register to be read or written. The EDX and EAX registers contain the MSR values to be read or written, as follows:

- *EDX*—Upper 32 bits of MSR. For the AAR, this contains the array pointer and (in contrast to all other MSRs) its contents are not altered by a RDMSR instruction.
- *EAX*—Lower 32 bits of MSR. For the AAR, this contains the data to be read/written.

All MSRs are 64 bits wide. However, the upper 32 bits of the AAR are write-only and are not returned on a read. EDX remains unaltered, making it more convenient to maintain the array pointer.

If an attempt is made to execute either the RDMSR or WRMSR instruction when CPL is greater than 0, or to access an undefined model-specific register, the processor generates a general-protection exception with error code zero.

Model-specific registers, as their name implies, may or may not be implemented by later models of the AMD-K5 processor.

RSM

mnemonic	*opcode*	*description*
RSM	0F AAh	Resume execution (exit System Management Mode)

Privilege:	CPL = 0
Registers Affected:	CS, DS, ES, FS, GS, SS, EIP, EFLAGS, LDTR, CR3, EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI
Flags Affected:	none
Exceptions Generated:	Real, Virtual-8086 mode—Invalid opcode if not in SMM Protected mode—Invalid opcode if not in SMM Protected mode—GP(0) if CPL not = 0

The RSM instruction should be the last instruction in any System Management Mode (SMM) service routine. It restores the processor state that was saved when the $\overline{\text{SMI}}$ interrupt was asserted. This instruction is only valid when the processor is in SMM. It generates an invalid opcode exception at all other times.

The processor enters the Shutdown state if any of the following illegal conditions are encountered during the execution of the RSM instruction: the SMM base value is not aligned on a 32-Kbyte boundary, or any reserved bit of CR4 set to 1, or the PG bit is set while the PE is cleared in CR0, or the NW bit it set while the CD bit is cleared in CR0.

Illegal Instruction (Reserved Opcode)

mnemonic	*opcode*	*description*
(none)	0F FFh	Illegal instruction (reserved opcode)

Privilege:	Any level
Registers Affected:	none
Flags Affected:	none
Exceptions Generated:	Real, Virtual-8086 mode—Invalid opcode Protected mode—Invalid opcode Protected mode—Invalid opcode

This opcode always generates an invalid opcode exception. The opcode will not be used in future AMD K86 processors.

2

Code Optimization for the AMD-K5 Processor

This chapter provides information to assist fast execution and details on dispatch and execution timing for x86 instructions. Throughout the chapter, the terms *clock* and *cycle* refer to processor clock cycles, not bus clock (CLK) cycles.

Code Optimization

The code optimization suggestions in this section cover both general superscalar optimization (that is, techniques common to both the AMD-K5 and Pentium processors) and techniques specific to the AMD-K5 processor. In general, all optimization techniques used for the Pentium processor apply to any wide-issue x86 processor, but wider-issue designs like the AMD-K5 processor have fewer restrictions.

General Superscalar Techniques

- *Short Forms*—Use shorter forms of instructions to increase the effective number of instructions that can be examined for decoding at any one time. Use 8-bit displacements and jump offsets where possible.
- *Simple Instructions*—Use simple instructions with hard-wired decode because they often perform more efficiently.

Moreover, future implementations may increase the penalties associated with microcoded instructions.

- *Dependencies*—Spread out true dependencies to increase the opportunities for parallel execution. Antidependencies and output dependencies do not impact performance.
- *Memory Operands*—Instructions that operate on data in memory (load/op/store) can inhibit parallelism. Using separate move and ALU instructions allows independent operations to be performed in parallel. On the other hand, if there are no opportunities for parallel execution, use the load/op/store forms to reduce the number of register spills (storing register values in memory to free registers for other uses) and increase code density.
- *Register Operands*—Maintain frequently used values in registers or on the stack rather than in static storage.
- *Branch Prediction*—Use control-flow constructs that allow effective branch prediction. Although correctly predicted branches have no cost, mispredicted branches incur a three clock penalty.
- *Stack References*—Use ESP for references to the stack so that EBP remains available for general use.
- *Stack Allocation*—When placing outgoing parameters on the stack, allocate space by adjusting the stack pointer (preferably at the same time local storage is allocated on procedure entry) and use moves rather than pushes. This method of allocation allows random access to the outgoing parameters so that they may be set up when they are calculated, instead of having to be held somewhere else until the procedure call. This method also uses fewer execution resources (specifically, fewer register-file write ports when updating ESP).
- *Shifts*—Although there is only one shifter, certain shifts can be done using other execution units: for example, shift left 1 by adding a value to itself. Use LEA index scaling to shift left by 1, 2, or 3.
- *Data Embedded in Code*—When data is embedded in the code segment, align it in separate cache blocks from nearby code to avoid some overhead in maintaining coherency between the instruction and data caches.
- *Undefined Flags*—Do not rely on the behavior of undefined flag results.

- *Loops*—Unroll loops to get more parallelism and reduce loop overhead even with branch prediction. Inline small routines to avoid procedure-call overhead. In both cases, however, consider the cost of possible increased register usage, which might add load/store instructions for register spilling.
- *Indexed Addressing*—There is no penalty for base + index addressing in the AMD-K5 processor. However, future implementations may have such a penalty to achieve a higher overall clock rate.

Techniques Specific to the AMD-K5 Processor

- *Jumps and Loops*—JCXZ requires 1 cycle (correctly predicted) and therefore is faster than a TEST/JZ, in contrast to the Pentium processor in which JCXZ requires 5 or 6 cycles. All forms of LOOP take 2 cycles (correctly predicted), which is also faster than the Pentium processor's 7 or 8 cycles.
- *Multiplies*—Independent IMULs can be pipelined at one per cycle with 4-cycle latency, in contrast to the Pentium processor's serialized 9-cycle time. (MUL has the same latency, although the implicit AX usage of MUL prevents independent, parallel MUL operations.)
- *Dispatch Conflicts*—Load-balancing (that is, selecting instructions for parallel decode) is still important, but to a lesser extent than on the Pentium processor. In particular, arrange instructions to avoid execution-unit dispatching conflicts. (See page 43.)
- *Instruction Prefixes*—There is no penalty for instruction prefixes, including combinations such as segment-size and operand-size prefixes. This is particularly important for 16-bit code. However, future implementations may have penalties for the use of these prefixes.
- *Byte Operations*—For byte operations, the high and low bytes of AX, BX, CX, and DX are effectively independent registers that can be operated on in parallel. For example, reading AL does not have a dependency on an outstanding write to AH.
- *Move and Convert*—MOVZX, MOVSX, CBW, CWDE, CWD, CDQ all take 1 cycle (2 cycles for memory-based input), in contrast to the Pentium processor's 2 or 3 cycles.

- *Bit Scan*—BSF and BSR take 1 cycle (2 cycles for memory-based input), in contrast to the Pentium processor's data-dependent 6 to 34 cycles.
- *Bit Test*—BT, BTS, BTR, and BTC take 1 cycle for register-based operands, and 2 or 3 cycles for memory-based operands with immediate bit-offset, in contrast to the Pentium processor's 4 to 9 cycles. Register-based bit-offset forms on the AMD-K5 processor take 5 cycles. If the semantics of the register-based bit-offset form are desired (where the bit offset can cover a very large bit string in memory), it is better to emulate this with simpler instructions that can be interleaved with independent instructions for greater parallelism.
- *Floating-Point Top-of-Stack Bottleneck*—The AMD-K5 processor has a pipelined floating-point unit. Greater parallelism can be achieved by using FXCH in parallel with floating-point operations to alleviate the top-of-stack bottleneck, as in the Pentium processor. The AMD-K5 processor also permits integer operations (ALU, branch, load/store) in parallel with floating-point operations.
- *Locating Branch Targets*—Performance can be sensitive to code alignment, especially in tight loops. Locating branch targets to the first 17 bytes of the 32-byte cache line maximizes the opportunity for parallel execution at the target. NOPs can be added to adjust this alignment. The AMD-K5 processor executes NOPs (opcode 90h) at the rate of two per cycle. Adding NOPs is even more effective if they execute in parallel with existing code. Other instructions of greater length, such as a register-based TEST instruction, can be used as NOPs to minimize the overhead of such padding.
- *Branch Prediction*—There are two branch prediction bits in a 32-byte instruction cache line. One bit applies to the first 16 bytes of the line and the second bit applies to the second 16 bytes of the line. For effective branch prediction, code should be generated with one branch per 16-byte line half.
- *Address-Generation Interlocks (AGIs)*—The AMD-K5 processor does not suffer from the single-cycle penalty that the 486 and Pentium processors have when a result from execution or from a data-cache access is used to form a cache address, so it is not necessary to avoid these situations.

Dispatch and Execution Timing

This section documents functional unit usage for each instruction, along with relative cycle numbers for dispatch and execution of the associated ROPs for the instruction.

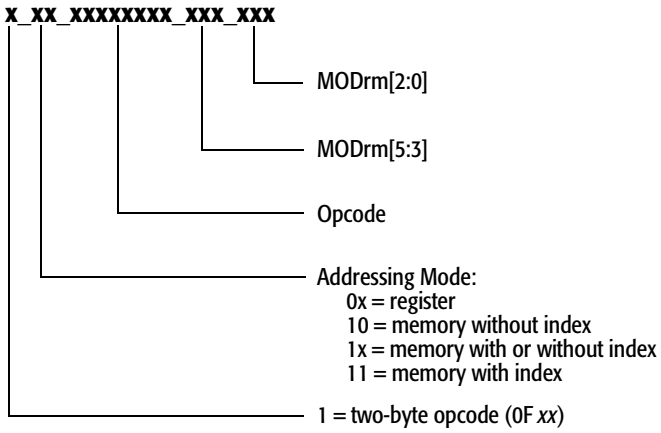
Notation

Table 2-1 contains the definitions for the integer instructions. Table 2-3 contains the definitions for the floating-point instructions. The first column in these tables indicates the instruction mnemonic and operand types. The following notations are used in the AMD-K5 microprocessor documentation:

- *reg*—register
- *mem*—memory location
- *imm*—immediate value
- *int_16*—16-bit integer
- *int_32*—32-bit integer
- *int_64*—64-bit integer
- *real_32*—32-bit floating-point number
- *real_64*—64-bit floating-point number
- *real_80*—80-bit floating-point number

If an operand refers to a specific register, the register name is used (e.g., AX, DX). When the register name is of the form Exx (e.g., EAX, ESI), the width of the register depends on the operand size attribute.

The second column contains an identifier with the following format:



The third column in the tables indicates whether the instruction is Fastpath (F) or Microcoded (M). Fastpath and MROM ROPs cannot both be present in a decode stage at the same time. If a microcoded instruction appears at the head of the byte queue without having been present in the queue on the previous cycle, there is a one-cycle penalty for MROM entry point generation.

Each x86 instruction is converted into one or more ROPs. The fourth column shows the execution unit and timing for each of the ROPs. The ROP types and corresponding execution units are:

- *ld*—load/store
- *st*—load/store
- *alu*—either alu0 or alu1
- *alu0*—alu0 only
- *alu1*—alu1 only
- *brn*—branch
- *fadd*—floating-point add pipe
- *fmul*—floating-point multiply pipe
- *fpmv*—floating-point move and compare pipe
- *fpfill*—floating-point upper half

The *x/y* value following the ROP type indicates the relative dispatch and execution cycle of the opcode, in the absence of any conflicts. The format is:

```
x/y[/z]
```

where:

- x = *Dispatch Cycle*—The relative cycle in which the ROP is dispatched from decode to the reservation station.
- y = *Execution Cycle*—The relative cycle in which the ROP is issued from the reservation station to the execution unit.
- z = *Result Cycle*—The relative cycle in which the result is returned on the result bus. It is indicated only when the latency is greater than one cycle. For stores, it reflects the relative time that a store operand can be forwarded from the store buffer to a dependent load operation.

Using the time that the first ROP of an instruction is dispatched to an execution unit as clock 1, the *x/y* value indicates in which clock each ROP is dispatched and executed relative to clock 1. The execution order and timing does not necessarily match the dispatch order and timing.

If any of the instructions read from or write to memory, it is assumed that the data exists in the cache.

Integer Instructions

Table 2-1 shows the execution-unit usage for each integer instruction, along with relative cycle numbers for dispatch and execution of the associated ROPs for the instruction.

Table 2-1. Integer Instructions

Instruction Mnemonic	Opcode Format	Fastpath or Microcode	Execution Unit Timing
ADD reg, reg	0_0x_000000xx_xxx_xxx	F	alu 1/1
ADD reg, mem	0_1x_0000001x_xxx_xxx	F	ld 1/1 alu 1/2
ADD mem, reg	0_1x_0000000x_xxx_xxx	F	ld 1/1 alu 1/2 st 1/1/3
ADD AL/AX/EAX, imm	0_xx_0000010x_xxx_xxx	F	alu 1/1
ADD reg, imm	0_0x_100000xx_000_xxx	F	alu 1/1
ADD mem, imm	0_1x_100000xx_000_xxx	F	ld 1/1 alu 1/2 st 1/1/3
AND reg, reg	0_0x_001000xx_xxx_xxx	F	alu 1/1
AND reg, mem	0_1x_0010001x_xxx_xxx	F	ld 1/1 alu 1/2
AND mem, reg	0_1x_0010000x_xxx_xxx	F	ld 1/1 alu 1/2 st 1/1/3
AND AL/AX/EAX, imm	0_xx_0010010x_xxx_xxx	F	alu 1/1
AND reg, imm	0_0x_100000xx_100_xxx	F	alu 1/1
AND mem, imm	0_1x_100000xx_100_xxx	F	ld 1/1 alu 1/2 st 1/1/3
BSF reg, reg	1_0x_10111100_xxx_xxx	F	alu1 1/1
BSF reg, mem	1_1x_10111100_xxx_xxx	F	ld 1/1 alu1 1/2
BSR reg, reg	1_0x_10111101_xxx_xxx	F	alu1 1/1
BSR reg, mem	1_1x_10111101_xxx_xxx	F	ld 1/1 alu1 1/2
BSWAP reg	1_xx_11001xxx_xxx_xxx	F	alu1 1/1
BT reg, reg	1_0x_10100011_xxx_xxx	F	alu1 1/1

Table 2-1. Integer Instructions (continued)

Instruction Mnemonic	Opcode Format	Fastpath or Microcode	Execution Unit Timing
BT mem, reg	1_1x_10100011_xxx_xxx	M	alu1 1/1 alu 1/2 alu 2/3 ld 2/4 alu1 3/5
BT reg, imm	1_0x_10111010_100_xxx	F	alu1 1/1
BT mem, imm	1_1x_10111010_100_xxx	F	ld 1/1 alu1 1/2
BTC reg, reg	1_0x_10111011_xxx_xxx	F	alu1 1/1
BTC mem, reg	1_1x_10111011_xxx_xxx	M	alu1 1/1 alu 1/2 alu 2/3 ld 2/4 alu1 3/5 st 3/5/6
BTC reg, imm	1_0x_10111010_111_xxx	F	alu1 1/1
BTC mem, imm	1_1x_10111010_111_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
BTR reg, reg	1_0x_10110011_xxx_xxx	F	alu1 1/1
BTR mem, reg	1_1x_10110011_xxx_xxx	M	alu1 1/1 alu 1/2 alu 2/3 ld 2/4 alu1 3/5 st 3/5/6
BTR reg, imm	1_0x_10111010_110_xxx	F	alu1 1/1
BTR mem, imm	1_1x_10111010_110_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
BTS reg, reg	1_0x_10101011_xxx_xxx	F	alu1 1/1
BTS mem, reg	1_1x_10101011_xxx_xxx	M	alu1 1/1 alu 1/2 alu 2/3 ld 2/4 alu1 3/5 st 3/5/6
BTS reg, imm	1_0x_10111010_101_xxx	F	alu1 1/1

Table 2-1. Integer Instructions (continued)

Instruction Mnemonic	Opcode Format	Fastpath or Microcode	Execution Unit Timing
BTS mem, imm	1_1x_10111010_101_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
CALL near relative	0_xx_11101000_xxx_xxx	M	alu 1/1 st 1/1/2 alu 1/1 brn 1/1
CALL near reg	0_0x_11111111_010_xxx	M	alu 1/1 st 1/1/2 alu 1/1 brn 1/1
CALL near mem	0_1x_11111111_010_xxx	M	alu 1/1 ld 1/1 st 1/1/2 alu 1/1 brn 2/2
CBW/DE	0_xx_10011000_xxx_xxx	F	alu1 1/1
CMP reg, reg	0_0x_001110xx_xxx_xxx	F	alu 1/1
CMP reg, mem	0_1x_0011101x_xxx_xxx	F	ld 1/1 alu 1/2
CMP mem, reg	0_1x_0011100x_xxx_xxx	F	ld 1/1 alu 1/2
CMP AL/AX/EAX, imm	0_xx_0011110x_xxx_xxx	F	alu 1/1
CMP reg, imm	0_0x_100000xx_111_xxx	F	alu 1/1
CMP mem, imm	0_1x_100000xx_111_xxx	F	ld 1/1 alu 1/2
CWD/DQ	0_xx_10011001_xxx_xxx	F	alu1 1/1
DEC reg	0_xx_01001xxx_xxx_xxx	F	alu 1/1
DEC reg	0_0x_1111111x_001_xxx	F	alu 1/1
DEC mem	0_1x_1111111x_001_xxx	F	ld 1/1 alu 1/2 st 1/1/3
IMUL AX, AL, reg	0_0x_11110110_101_xxx	F	fpfill 1/1/4 fmul 1/1/4
IMUL EDX:EAX, EAX, reg	0_0x_11110111_101_xxx	F	fpfill 1/1/4 fmul 1/1/4
IMUL reg, reg	1_0x_10101111_xxx_xxx	F	fpfill 1/1/4 fmul 1/1/4

Table 2-1. Integer Instructions (continued)

Instruction Mnemonic	Opcode Format	Fastpath or Microcode	Execution Unit Timing
IMUL reg, reg, imm	0_0x_011010x1_xxx_xxx	F	fpfill 1/1/4 fmul 1/1/4
IMUL AX, AL, mem	0_1x_11110110_101_xxx	F	ld 1/1 fpfill 1/2/4 fmul 1/2/4
IMUL EDX:EAX, EAX, mem	0_1x_11110111_101_xxx	F	ld 1/1 fpfill 1/2/4 fmul 1/2/4
IMUL reg, mem	1_1x_10101111_xxx_xxx	F	ld 1/1 fpfill 1/2/4 fmul 1/2/4
IMUL reg, reg, mem	0_1x_011010x1_xxx_xxx	F	ld 1/1 fpfill 1/2/4 fmul 1/2/4
INC reg	0_xx_01000xxx_xxx_xxx	F	alu 1/1
INC reg	0_0x_1111111x_000_xxx	F	alu 1/1
INC mem	0_1x_1111111x_000_xxx	F	ld 1/1 alu 1/2 st 1/1/3
Jcc short displacement	0_xx_0111xxxx_xxx_xxx	F	brn 1/1
Jcc long displacement	1_xx_1000xxxx_xxx_xxx	F	brn 1/1
JCXZ short displacement	0_xx_11100011_xxx_xxx	F	brn 1/1
JMP long displacement	0_xx_11101001_xxx_xxx	F	brn 1/1
JMP short displacement	0_xx_11101011_xxx_xxx	F	brn 1/1
JMP reg	0_0x_11111111_100_xxx	F	brn 1/1
JMP mem	0_1x_11111111_100_xxx	F	ld 1/1 brn 1/2
LEA	0_1x_10001101_xxx_xxx	F	ld 1/1
LOOP short displacement	0_xx_11100010_xxx_xxx	F	alu 1/1 brn 1/2
LOOPE short displacement	0_xx_11100001_xxx_xxx	M	alu 1/1 brn 1/2
LOOPNE short displacement	0_xx_11100000_xxx_xxx	M	alu 1/1 brn 1/2
MOV reg, reg	0_0x_100010xx_xxx_xxx	F	alu 1/1
MOV reg, mem	0_1x_1000101x_xxx_xxx	F	ld 1/1
MOV mem, reg	0_10_1000100x_xxx_xxx	F	st 1/1

Table 2-1. Integer Instructions (continued)

Instruction Mnemonic	Opcode Format	Fastpath or Microcode	Execution Unit Timing
MOV mem, reg (base + index addressing)	0_11_1000100x_xxx_xxx	F	ld 1/1 st 1/2/3
MOV AL/AX/EAX, mem	0_xx_1010000x_xxx_xxx	F	ld 1/1
MOV mem, AL/AX/EAX	0_xx_1010001x_xxx_xxx	F	st 1/1
MOV reg, imm	0_0x_1100011x_000_xxx	F	alu 1/1
MOV reg, imm	0_xx_1011xxxx_xxx_xxx	F	alu 1/1
MOV mem, imm	0_10_1100011x_000_xxx	F	alu 1/1 st 1/1
MOV mem, imm (base + index addressing)	0_11_1100011x_000_xxx	F	alu 1/1 ld 1/1 st 1/2/3
MOVSX reg, reg	1_0x_1011111x_xxx_xxx	F	alu1 1/1
MOVSX reg, mem	1_1x_1011111x_xxx_xxx	F	ld 1/1 alu1 1/2
MOVZX reg, reg	1_0x_1011011x_xxx_xxx	F	alu 1/1
MOVZX reg, mem	1_1x_1011011x_xxx_xxx	F	ld 1/1 alu 1/2
MUL AX, AL, reg	0_0x_11110110_100_xxx	F	fpfill 1/1/4 fmul 1/1/4
MUL EDX:EAX, EAX, reg	0_0x_11110111_100_xxx	F	fpfill 1/1/4 fmul 1/1/4
MUL AX, AL, mem	0_1x_11110110_100_xxx	F	ld 1/1 fpfill 1/2/4 fmul 1/2/4
MUL EDX:EAX, EAX, mem	0_1x_11110111_100_xxx	F	ld 1/1 fpfill 1/2/4 fmul 1/2/4
NEG reg	0_0x_1111011x_011_xxx	F	alu 1/1
NEG mem	0_1x_1111011x_011_xxx	F	ld 1/1 alu 1/2 st 1/1/3
NOP (XCHG EAX, EAX)	0_xx_10010000_xxx_xxx	F	alu 1/1
NOT reg	0_0x_1111011x_010_xxx	F	alu 1/1
NOT mem	0_1x_1111011x_010_xxx	F	ld 1/1 alu 1/2 st 1/1/3
OR reg, reg	0_0x_000010xx_xxx_xxx	F	alu 1/1

Table 2-1. Integer Instructions (continued)

Instruction Mnemonic	Opcode Format	Fastpath or Microcode	Execution Unit Timing
OR reg, mem	0_1x_0000101x_xxx_xxx	F	ld 1/1 alu 1/2
OR mem, reg	0_1x_0000100x_xxx_xxx	F	ld 1/1 alu 1/2 st 1/1/3
OR AL/AX/EAX, imm	0_xx_0000110x_xxx_xxx	F	alu 1/1
OR reg, imm	0_0x_100000xx_001_xxx	F	alu 1/1
OR mem, imm	0_1x_100000xx_001_xxx	F	ld 1/1 alu 1/2 st 1/1/3
POP reg	0_xx_01011xxx_xxx_xxx	F	ld 1/1 alu 1/1
POP reg	0_0x_10001111_000_xxx	F	ld 1/1 alu 1/1
POP mem	0_1x_10001111_000_xxx	M	ld 1/1 ld 1/1 st 2/2/3 alu 2/2
PUSH reg	0_xx_01010xxx_xxx_xxx	F	st 1/1 alu 1/1/2
PUSH reg	0_0x_11111111_110_xxx	F	st 1/1 alu 1/1/2
PUSH imm	0_xx_011010x0_xxx_xxx	F	alu 1/1 st 1/1/2 alu 1/1
PUSH mem	0_1x_11111111_110_xxx	M	ld 1/1 st 1/1/2 alu 1/1
RET near	0_xx_11000011_xxx_xxx	F	ld 1/1 alu 1/1 brn 1/2
RET near imm	0_xx_11000010_xxx_xxx	M	ld 1/1 alu 1/1 alu 1/2 brn 1/2
ROL reg, 1	0_0x_1101000x_000_xxx	F	alu1 1/1
ROL mem, 1	0_1x_1101000x_000_xxx	F	ld 1/1 alu1 1/2 st 1/1/3

Table 2-1. Integer Instructions (continued)

Instruction Mnemonic	Opcode Format	Fastpath or Microcode	Execution Unit Timing
ROL reg, imm	0_0x_1100000x_000_xxx	F	alu1 1/1
ROL mem, imm	0_1x_1100000x_000_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
ROL reg, CL	0_0x_1101001x_000_xxx	F	alu1 1/1
ROL mem, CL	0_1x_1101001x_000_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
ROR reg, 1	0_0x_1101000x_001_xxx	F	alu1 1/1
ROR mem, 1	0_1x_1101000x_001_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
ROR reg, imm	0_0x_1100000x_001_xxx	F	alu1 1/1
ROR mem, imm	0_1x_1100000x_001_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
ROR reg, CL	0_0x_1101001x_001_xxx	F	alu1 1/1
ROR mem, CL	0_1x_1101001x_001_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
SAR reg, 1	0_0x_1101000x_111_xxx	F	alu1 1/1
SAR mem, 1	0_1x_1101000x_111_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
SAR reg, mem	0_0x_1100000x_111_xxx	F	alu1 1/1
SAR mem, imm	0_1x_1100000x_111_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
SAR reg, CL	0_0x_1101001x_111_xxx	F	alu1 1/1
SAR mem, CL	0_1x_1101001x_111_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
SETcc reg	1_0x_1001xxxx_xxx_xxx	F	brn 1/1
SETcc mem	1_1x_1001xxxx_xxx_xxx	F	brn 1/1 ld 1/1 st 1/2/3
SHL reg, 1	0_0x_1101000x_1x0_xxx	F	alu1 1/1

Table 2-1. Integer Instructions (continued)

Instruction Mnemonic	Opcode Format	Fastpath or Microcode	Execution Unit Timing
SHL mem, 1	0_1x_1101000x_1x0_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
SHL reg, mem	0_0x_1100000x_1x0_xxx	F	alu1 1/1
SHL mem, imm	0_1x_1100000x_1x0_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
SHL reg, CL	0_0x_1101001x_1x0_xxx	F	alu1 1/1
SHL mem, CL	0_1x_1101001x_1x0_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
SHLD reg, reg, imm	1_0x_10100100_xxx_xxx	F	alu1 1/1 alu1 2/2
SHLD mem, reg, imm	1_1x_10100100_xxx_xxx	M	alu1 1/1 ld 1/1 alu1 2/2 st 2/2/3
SHLD reg, reg, CL	1_0x_10100101_xxx_xxx	F	alu1 1/1 alu1 2/2
SHLD mem, reg, CL	1_1x_10100101_xxx_xxx	M	alu1 1/1 ld 1/1 alu1 2/2 st 2/2/3
SHR reg, 1	0_0x_1101000x_101_xxx	F	alu1 1/1
SHR mem, 1	0_1x_1101000x_101_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
SHR reg, mem	0_0x_1100000x_101_xxx	F	alu1 1/1
SHR mem, imm	0_1x_1100000x_101_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
SHR reg, CL	0_0x_1101001x_101_xxx	F	alu1 1/1
SHR mem, CL	0_1x_1101001x_101_xxx	F	ld 1/1 alu1 1/2 st 1/1/3
SHRD reg, reg, imm	1_0x_10101100_xxx_xxx	F	alu1 1/1 alu1 2/2

Table 2-1. Integer Instructions (continued)

Instruction Mnemonic	Opcode Format	Fastpath or Microcode	Execution Unit Timing
SHRD mem, reg, imm	1_1x_10101100_xxx_xxx	M	alu1 1/1 ld 1/1 alu1 2/2 st 2/2/3
SHRD reg, reg, CL	1_0x_10101101_xxx_xxx	F	alu1 1/1 alu1 2/2
SHRD mem, reg, CL	1_1x_10101101_xxx_xxx	M	alu1 1/1 ld 1/1 alu1 2/2 st 2/2/3
SUB reg, reg	0_0x_001010xx_xxx_xxx	F	alu 1/1
SUB reg, mem	0_1x_0010101x_xxx_xxx	F	ld 1/1 alu 1/2
SUB mem, reg	0_1x_0010100x_xxx_xxx	F	ld 1/1 alu 1/2 st 1/1/3
SUB AL/AX/EAX, imm	0_xx_0010110x_xxx_xxx	F	alu 1/1
SUB reg, imm	0_0x_100000xx_101_xxx	F	alu 1/1
SUB mem, imm	0_1x_100000xx_101_xxx	F	ld 1/1 alu 1/2 st 1/1/3
TEST reg, reg	0_0x_1000010x_xxx_xxx	F	alu 1/1
TEST mem, reg	0_1x_1000010x_xxx_xxx	F	ld 1/1 alu 1/2
TEST reg, imm	0_0x_1111011x_00x_xxx	F	alu 1/1
TEST AL/AX/EAX, imm	0_xx_1010100x_xxx_xxx	F	alu 1/1
TEST mem, imm	0_1x_1111011x_00x_xxx	F	ld 1/1 alu 1/2
XCHG EAX, reg (except EAX)	0_xx_10010xxx_xxx_xxx	F	alu 1/1 alu 1/1 alu 2/2
XCHG reg, reg	0_0x_1000011x_xxx_xxx	F	alu 1/1 alu 1/1 alu 2/2
XCHG mem, reg	0_1x_1000011x_xxx_xxx	F	ld 1/1 st 1/1/2 alu 1/2
XOR reg, reg	0_0x_001100xx_xxx_xxx	F	alu 1/1

Table 2-1. Integer Instructions (continued)

Instruction Mnemonic	Opcode Format	Fastpath or Microcode	Execution Unit Timing
XOR reg, mem	0_1x_0011001x_xxx_xxx	F	ld 1/1 alu 1/2
XOR mem, reg	0_1x_0011000x_xxx_xxx	F	ld 1/1 alu 1/2 st 1/1/3
XOR AL/AX/EAX, imm	0_xx_0011010x_xxx_xxx	F	alu 1/1
XOR reg, imm	0_0x_100000xx_110_xxx	F	alu 1/1
XOR mem, imm	0_1x_100000xx_110_xxx	F	ld 1/1 alu 1/2 st 1/1/3

Integer Dot Product Example

This example illustrates an optimal code sequence for an integer dot product operation that performs multiply/accumulates (MACs) at the rate of one every 3 cycles. In this example, the array size is a constant. The loop is unrolled to perform separate MAC operations in parallel for even and odd elements. The final sum is generated outside the loop (as well as the final iteration for odd-sized arrays).

```
mac_loop:
   MOV   EAX, [ESI][ECX*4]       ;load A(i)
   MOV   EBX, [ESI][ECX*4]+4     ;load A(i+1)
   IMUL  EAX, [EDI][ECX*4]       ;A(i) * B(i)
   IMUL  EBX, [EDI][ECX*4]+4     ;A(i+1) * B(i+1)
   ADD   ECX, 2                  ;increment index
   ADD   EDX, EAX                ;even sum
   ADD   EBP, EBX                ;odd sum
   CMP   ECX, EVEN_ARRAY_SIZE    ;loop control
   JL    mac_loop                ;jump

   ;do final MAC here for odd-sized arrays

   ADD  EDX, EBP                 ;final sum
```

Table 2-2 shows the timing of internal operations from dispatch to retire of each ROP for nearly two iterations of this loop. All memory accesses are assumed to hit in the cache. *EVEN_ARRAY_SIZE* is set to 20.

Table 2-2. Integer Dot Product Internal Operations Timing

Instruction	Cycle													
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
MOV EAX,[ESI][ECX*4]	L	>	-	-	-	!								
MOV EBX,[ESI][ECX*4]+4	L	>	-	-	-	!								
IMUL EAX,[EDI][ECX*4]		L	>	-	-	!								
		-	M	M	M	M	>	!						
IMUL EBX,[EDI][ECX*4]+4			L	>	-	-	-	!						
			-	M	M	M	M	>	!					
ADD ECX,2			A	>	-	-	-	!						
ADD EDX,EAX					-	-	-	A	>	!				
ADD EBP,EBX					-	-	-	A	>	!				
CMP ECX,20						-	-	-	A	>	!			
JL LOOP						-	-	-	-	B	>	!		
MOV EAX,[ESI][ECX*4]							L	>	-	-	-	!		
MOV EBX,[ESI][ECX*4]+4							L	>	-	-	-	!		
IMUL EAX,[EDI][ECX*4]								L	>	-	-	!		
								-	M	M	M	M	>	!
IMUL EAX,[EDI][ECX*4]+4									L	>	-	-	-	!
									-	M	M	M	M	>

Notes:
L— load execute
M— multiply execute
A— ALU execute
B— branch execute
>— result
!— retire (update real state)
- — preceding execute: waiting in the reservation station

Floating-Point Instructions

Floating-point ROPs are always dispatched in pairs to the FPU reservation station. The first ROP conveys the lower halves of the A and B operands, and it always has the *fpfill* ROP type. The second ROP conveys the upper halves of the operands, as well as the numeric opcode. Data from both ROPs is merged in the reservation station and must be converted into an internal floating-point format before it can be issued to the add pipe (*fadd*), multiply pipe (*fmul*), or detect pipe (*fmv*). It takes one cycle to perform the conversion, and this delay is incurred whenever the source of the data is the register file or one of the other functional units (e.g., load/store, ALU). If data is being forwarded from the FPU itself, however, no format conversion is required and operands are fast-forwarded from the back end of a pipe to the front of any other pipe without the one-cycle delay.

The add/subtract/reverse FPU latencies assume that cancellation does not occur in the adder/subtractor. If cancellation does occur, an extra cycle is required to normalize the result.

Table 2-3 shows the execution-unit usage for each floating-point instruction, along with relative cycle numbers for dispatch and execution of the associated ROPs for the instruction.

Table 2-3. Floating-Point Instructions

Instruction Mnemonic	Opcode Format	Fastpath or Microcoded	Execution Unit Timing
FABS	0_0x_11011001_100_xxx	F	fpfill 1/2/4 fmv 1/2/4
FADD ST, ST(i)	0_0x_11011000_000_xxx	F	fpfill 1/2/5 fadd 1/2/5
FADD ST(i), ST	0_0x_11011000_000_xxx	F	fpfill 1/2/5 fadd 1/2/5
FADD real_32	0_1x_11011000_000_xxx	F	ld 1/1 fpfill 1/3/6 fadd 1/3/6
FADD real_64	0_1x_11011100_000_xxx	M	ld 1/1 ld 1/2 fpfill 1/4/7 fadd 1/4/7

Table 2-3. Floating-Point Instructions (continued)

Instruction Mnemonic	Opcode Format	Fastpath or Microcoded	Execution Unit Timing
FADDP ST(i), ST	0_0x_11011110_000_xxx	F	fpfill 1/2/5 fadd 1/2/5
FCHS	0_0x_11011001_100_xxx	F	fpfill 1/2/4 fchs 1/2/4
FCOM ST(i)	0_0x_11011x00_010_xxx	F	fpfill 1/2/4 fcmpst 1/2/4
FCOM real_32	0_1x_11011000_010_xxx	F	ld 1/1 fpfill 1/3/5 fmv 1/3/5
FCOM real_64	0_1x_11011100_010_xxx	M	ld 1/1 ld 1/2 fpfill 1/4/6 fadd 1/4/6
FCOMP ST(i)	0_0x_11011x00_011_xxx	F	fpfill 1/2/4 fmv 1/2/4 alu 1/1
FCOMP real_32	0_1x_11011000_011_xxx	F	ld 1/1 fpfill 1/3/5 fmv 1/3/5
FCOMP real_64	0_1x_11011100_011_xxx	M	ld 1/1 ld 1/2 fpfill 1/4/6 fadd 1/4/6
FCOMPP	0_0x_11011110_011_xxx	F	fpfill 1/2/4 fmv 1/2/4 nop 1/1/2
FDECSTP	0_0x_11011001_110_xxx	M	alu 1/1/2 alu 1/1/2
FIADD int_16	0_1x_11011110_000_xxx	M	ld 1/1 fpfill 1/3/7 fadd 1/3/7 fpfill 2/7/10 fadd 2/7/10
FIADD int_32	0_1x_11011010_000_xxx	M	ld 1/1 fpfill 1/3/7 fadd 1/3/7 fpfill 2/7/10 fadd 2/7/10

Table 2-3. Floating-Point Instructions (continued)

Instruction Mnemonic	Opcode Format	Fastpath or Microcoded	Execution Unit Timing
FICOM int_16	0_1x_11011110_010_xxx	M	ld 1/1 fpfill 1/3/7 fadd 1/3/7 fpfill 2/7/9 fmv 2/7/9
FICOM int_32	0_1x_11011010_010_xxx	M	ld 1/1 fpfill 1/3/7 fadd 1/3/7 fpfill 2/7/9 fmv 2/7/9
FICOMP int_16	0_1x_11011110_011_xxx	M	ld 1/1 fpfill 1/3/7 fadd 1/3/7 fpfill 2/7/9 fmv 2/7/9
FICOMP int_32	0_1x_11011010_011_xxx	M	ld 1/1 fpfill 1/3/7 fadd 1/3/7 fpfill 2/7/9 fmv 2/7/9
FILD int_16	0_1x_11011111_000_xxx	F	ld 1/1 fpfill 1/3/7 fadd 1/3/7
FILD int_32	0_1x_11011011_000_xxx	F	ld 1/1 fpfill 1/3/7 fadd 1/3/7
FILD int_64	0_1x_11011111_101_xxx	M	ld 1/1 ld 1/2 fpfill 1/4/8 fadd 1/4/8
FIMUL int_16	0_1x_11011110_001_xxx	M	ld 1/1 fpfill 1/3/7 fadd 1/3/7 fpfill 2/7/11 fmul 2/7/11
FIMUL int_32	0_1x_11011010_001_xxx	M	ld 1/1 fpfill 1/3/7 fadd 1/3/7 fpfill 2/7/11 fmul 2/7/11

Table 2-3. Floating-Point Instructions (continued)

Instruction Mnemonic	Opcode Format	Fastpath or Microcoded	Execution Unit Timing
FIST int_16	0_1x_11011111_010_xxx	M	ld 1/1 fpfill 1/2/5 fadd 1/2/5 st 1/5/6
FIST int_32	0_1x_11011011_010_xxx	M	ld 1/1 fpfill 1/2/5 fadd 1/2/5 st 1/5/6
FISTP int_16	0_1x_11011111_011_xxx	M	ld 1/1 fpfill 1/2/5 fadd 1/2/5 st 1/5/6
FISTP int_32	0_1x_11011011_011_xxx	M	ld 1/1 fpfill 1/2/5 fadd 1/2/5 st 1/5/6
FISTP int_64	0_1x_11011111_111_xxx	M	ld 1/1 ld 1/2 fpfill 1/2/5 fadd 1/2/5 st 2/3/6 st 2/4/7
FISUB int_16	0_1x_11011110_100_xxx	M	ld 1/1 fpfill 1/3/7 fadd 1/3/7 fpfill 2/7/10 fadd 2/7/10
FISUB int_32	0_1x_11011010_100_xxx	M	ld 1/1 fpfill 1/3/7 fadd 1/3/7 fpfill 2/7/10 fadd 2/7/10
FISUBR int_16	0_1x_11011110_101_xxx	M	ld 1/1 fpfill 1/3/7 fadd 1/3/7 fpfill 2/7/10 fadd 2/7/10

Table 2-3. Floating-Point Instructions (continued)

Instruction Mnemonic	Opcode Format	Fastpath or Microcoded	Execution Unit Timing
FISUBR int_32	0_1x_11011010_101_xxx	M	ld 1/1 fpfill 1/3/7 fadd 1/3/7 fpfill 2/7/10 fadd 2/7/10
FLD real_32	0_1x_11011001_000_xxx	F	ld 1/1 fpfill 1/3/5 fmv 1/3/5
FLD real_64	0_1x_11011101_000_xxx	M	ld 1/1 ld 1/2 fpfill 1/4/6 fmv 1/4/6
FLD real_80	0_1x_11011011_101_xxx	M	ld 1/1 ld 1/2 fpfill 1/6/8 fmv 1/6/8
FLD ST(i)	0_0x_11011001_000_xxx	F	fpfill 1/2/4 fmv 1/2/4 nop 1/1
FMUL ST, ST(i)	0_0x_11011000_001_xxx	F	fpfill 1/2/8 fmul 1/2/8
FMUL ST(i), ST	0_0x_11011100_001_xxx	F	fpfill 1/2/8 fmul 1/2/8
FMUL real_32	0_1x_11011000_001_xxx	F	ld 1/1 fpfill 1/3/7 fmul 1/3/7
FMUL real_64	0_1x_11011100_001_xxx	M	ld 1/1 ld 1/2 fpfill 1/4/10 fmul 1/4/10
FMULP ST, ST(i)	0_0x_11011110_001_xxx	F	fpfill 1/2/8 fmul 1/2/8
FMULP ST(i), ST	0_0x_11011110_001_xxx	F	fpfill 1/2/8 fmul 1/2/8
FNOP	0_0x_11011001_010_xxx	F	alu 1/1/2 alu 1/1/2
FRNDINT	0_0x_11011001_111_xxx	F	fpfill 1/2/9 fadd 1/2/9

Table 2-3. Floating-Point Instructions (continued)

Instruction Mnemonic	Opcode Format	Fastpath or Microcoded	Execution Unit Timing
FSCALE	0_0x_11011001_111_xxx	F	fpfill 1/2/8 fadd 1/2/8
FST real_32	0_1x_11011001_010_xxx	M	ld 1/1 fpfill 1/2/4 fmv 1/2/4 st 1/2/5
FST ST(i)	0_0x_11011101_010_xxx	F	fpfill 1/2/4 fmv 1/2/4
FSTP real_32	0_1x_11011001_011_xxx	M	ld 1/1 fpfill 1/2/4 fmv 1/2/4 st 1/2/5
FSTP real_64	0_1x_11011101_011_xxx	M	ld 1/1 ld 1/2 fpfill 1/2/4 fmv 1/2/4 st 2/3/5 st 2/4/6
FSTP real_80	0_1x_11011011_111_xxx	M	ld 1/1 ld 1/2 fpfill 1/2/4 fmv 1/2/4 st 2/3/5 st 2/4/6
FSTP ST(i)	0_0x_11011x01_011_xxx	F	fpfill 1/2/4 fmv 1/2/4
FSUB ST, ST(i)	0_0x_11011000_100_xxx	F	fpfill 1/2/5 fadd 1/2/5
FSUB ST(i), ST	0_0x_11011100_100_xxx	F	fpfill 1/2/5 fadd 1/2/5
FSUB real_32	0_1x_11011000_100_xxx	F	ld 1/1 fpfill 1/3/6 fadd 1/3/6
FSUB real_64	0_1x_11011100_100_xxx	M	ld 1/1 ld 1/2 fpfill 1/4/7 fadd 1/4/7
FSUBP ST(i), ST	0_0x_11011110_100_xxx	F	fpfill 1/2/5 fadd 1/2/5

Table 2-3. Floating-Point Instructions (continued)

Instruction Mnemonic	Opcode Format	Fastpath or Microcoded	Execution Unit Timing
FSUBR ST, ST(i)	0_0x_11011000_101_xxx	F	fpfill 1/2/5 fadd 1/2/5
FSUBR ST(i), ST	0_0x_11011100_101_xxx	F	fpfill 1/2/5 fadd 1/2/5
FSUBR real_32	0_1x_11011000_101_xxx	F	ld 1/1 fpfill 1/3/6 fadd 1/3/6
FSUBR real_64	0_1x_11011100_101_xxx	M	ld 1/1 ld 1/2 fpfill 1/4/7 fadd 1/4/7
FSUBRP ST(i), ST	0_0x_11011110_101_xxx	F	fpfill 1/2/5 fadd 1/2/5
FTST	0_0x_11011001_100_xxx	F	fpfill 1/2/4 fmv 1/2/4
FUCOM ST(i)	0_0x_11011101_100_xxx	F	fpfill 1/2/4 fmv 1/2/4
FUCOMP ST(i)	0_0x_11011101_101_xxx	F	fpfill 1/2/4 fmv 1/2/4 nop 1/1
FUCOMPP	0_0x_11011010_101_xxx	F	fpfill 1/2/4 fmv 1/2/4 nop 1/1
FWAIT	0_xx_10011011_xxx_xxx	F	alu 1/1
FXAM	0_0x_11011001_100_xxx	F	fpfill 1/2/4 fmv 1/2/4
FXCH ST(i)	0_0x_11011001_001_xxx	F	brn 1/1
FXTRACT	0_0x_11011001_110_xxx	M	fpfill 1/2/4 fmv 1/2/4 fpfill 2/3/11 fadd 2/3/11 fpfill 3/4/6 fmv 3/4/6

3

AMD-K5 Processor Initialization

The internal state of the AMD-K5 processor can be initialized to known values via either the RESET or INIT signal. RESET takes effect immediately, asynchronously to whatever the processor may be doing. INIT is recognized only at the next instruction boundary after assertion. RESET provides a complete initialization, whereas INIT provides only a subset of this. Specifically, INIT does not affect the numeric coprocessor state or the cache contents. The initialized internal state is described in the following paragraphs. Except where explicitly noted, the resulting state is the same for both RESET and INIT.

General Registers

All general registers except EAX and EDX are cleared. EDX is loaded with the processor ID value. This is the value returned by issuing the CPUID instruction with a 1 in EAX (see "CPUID" on page 29). EAX is normally cleared, although if BIST is run along with reset and an error is detected, EAX will be loaded with a BIST error code.

Segment Registers

The selector portion of all segment registers is cleared. The access rights and attribute fields are set up as shown in Table 3-1.

Table 3-1. Segment Register Attribute Fields Initial Values

Attribute Field	Value	Description
G	0	Byte granularity
D/B	0	16-bit
P	1	Present
DPL	0	Privilege level
S	1	Application segment (except LDTR)
Type	2	Data, read/write

The limit fields are set to FFFFh. For CS, the base address is set to FFFF_0000h; for all others the base address is 0. Note that IDTR and GDTR consist of the just base and limit values, which are initialized to 0 and FFFFh, respectively.

EIP and EFLAGS

All bits of EFLAGS are cleared, with the exception of bit 1, which is hardwired to a 1. EIP is set to 0000_FFF0h.

Control and Debug Registers

On RESET, CR0 is initialized to 0600_0010h; the NW and CD bits are set to disable the caches. On INIT, the NW and CD bits retain their prior state. Note that the ET bit is always set. CR2, CR3, and CR4 are cleared. Debug registers 0–3 are cleared. DR6 is set to FFFF_0FF0h, and DR7 is set to 0000_0400h (bit 10 is hardwired to a 1).

Model-Specific Registers

The HWCR (Hardware Configuration Register) is cleared. On RESET, the TSC (Time Stamp Counter) is cleared, although it starts incrementing some clocks before the first instruction is fetched. INIT does not affect the TSC.

Caches and TLB

All TLB entries are invalidated; all cache Tag Valid bits are cleared on RESET. All other cache contents are undefined. On INIT, the Tag Valid bits, as well as all other cache contents, retain their prior state.

Floating-Point Unit

The state of the FPU is initialized by RESET only; it is unaffected by INIT. On RESET, the FP instruction address, data address, opcode, Status Word, and Control Word are all cleared (note that FP Control Word bit 6 is hardwired to 1). The FP Tag Word is set to 5555h. All entries in the FP stack are initialized to 0.

4

AMD-K5 Processor Test and Debug

The AMD-K5 processor has the following modes in which processor and system operation can be tested or debugged:

- *Hardware Configuration Register (HWCR)*—The HWCR is a model-specific register that contains configuration bits that enable cache, branch tracing, debug, and clock control functions.
- *Built-In Self-Test (BIST)*—Both normal and test access port (TAP) BIST.
- *Output-Float Test*—A test mode that causes the AMD-K5 processor to float all of its output and bidirectional signals.
- *Cache and TLB Testing*—The Array Access Register (AAR) supports writes and reads to any location in the tag and data arrays of the processor's on-chip caches and TLBs.
- *Debug Registers*—Standard 486 debug functions, with an I/O-breakpoint extension.
- *Branch Tracing*—A pair of special bus cycles can be driven immediately after taken branches to specify information about the branch instruction and its target. The Hardware Configuration Register (HWCR) provides support for this and other debug functions.
- *Functional Redundancy Checking*—Support for real-time testing that uses two processors in a master-checker relationship.

- *Test Access Port (TAP) Boundary-Scan Testing*—The JTAG test access functions defined by the *IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE 1149.1-1990)* specification.
- *Hardware Debug Tool (HDT)*—The hardware debug tool (HDT), sometimes referred to as the debug port or Probe mode, is a collection of signals, registers, and processor microcode that is enabled when external debug logic drives R/$\overline{S}$ Low or loads the AMD-K5 processor's Test Access Port (TAP) instruction register with the USEHDT instruction.

The test-related signals are described in Chapter 5 of the *AMD-K5 Processor Technical Reference Manual*. The signals include the following:

- $\overline{\text{FLUSH}}$
- $\overline{\text{FRCMC}}$
- $\overline{\text{IERR}}$
- INIT
- PRDY
- R/$\overline{S}$
- RESET
- TCK
- TDI
- TDO
- TMS
- $\overline{\text{TRST}}$

The sections that follow provide details on each of the test and debug features.

Hardware Configuration Register (HWCR)

The Hardware Configuration Register (HWCR) is a model-specific register (MSR) that contains configuration bits that enable cache, branch tracing, debug, and clock control functions. The WRMSR and RDMSR instructions access the HWCR when the ECX register contains the value 83h, as described on page 34. Figure 4-1 and Table 4-1 show the format and fields of the HWCR.

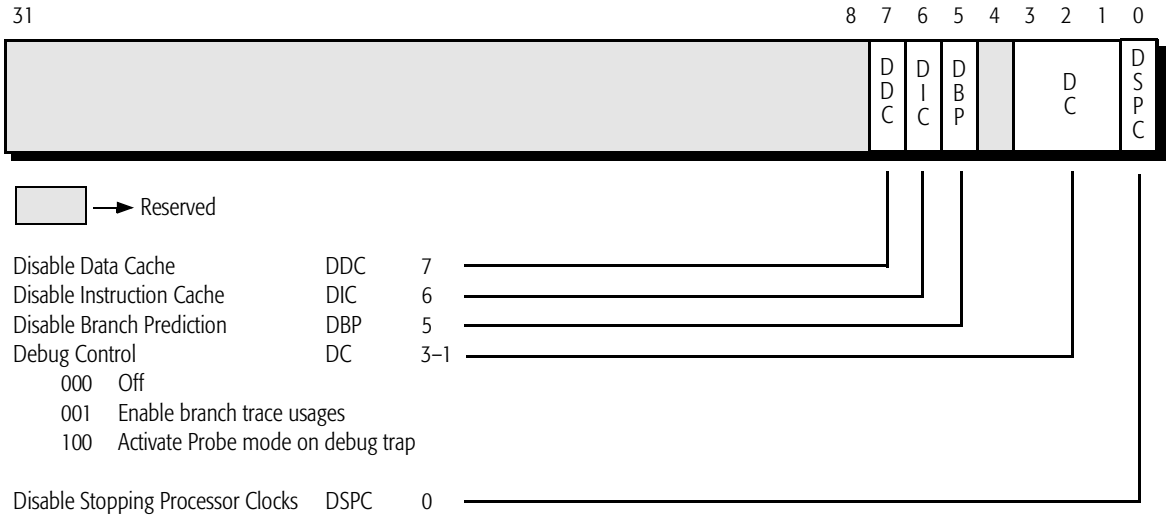


Figure 4-1. Hardware Configuration Register (HWCR)

Table 4-1. Hardware Configuration Register (HWCR) Fields

Bit	Mnemonic	Description	Function
31–8	—	—	*reserved*
7	DDC	Disable Data Cache	Disables data cache. 0 = enabled, 1 = disabled.
6	DIC	Disable Instruction Cache	Disables instruction cache. 0 = enabled, 1 = disabled.
5	DBP	Disable Branch Prediction	Disables branch prediction. 0 = enabled, 1 = disabled.
4	—	—	*reserved*
3–1	DC	Debug Control	Debug control bits: 000 Off (disable HWCR debug control). 001 Enable branch-tracing messages. See "Branch Tracing" on page 85. 010 *reserved* 011 *reserved* 100 HDT trap 101 *reserved* 110 *reserved* 111 *reserved*
0	DSPC	Disable Stopping Processor Clocks	Disables stopping of internal processor clocks in the Halt and Stop Grant states. 0 = enabled, 1 = disabled.

Notes:
Documentation on the Hardware Debug Tool (HDT) is available from AMD under a nondisclosure agreement.

Built-In Self-Test (BIST)

The processor supports the following types of built-in self-test:

- *Normal BIST*—A built-in self-test mode typically used to test system functions after RESET
- *Test Access Port (TAP) BIST*—A self-test mode started by the TAP instruction, RUNBIST

All internal arrays except the TLB are tested in parallel by hardware. The TLB is tested by microcode. Unlike the Pentium processor, the AMD-K5 processor does not report parity errors on $\overline{\text{IERR}}$ for every cache or TLB access. Instead, the AMD-K5 processor fully tests its caches during the BIST. $\overline{\text{EADS}}$ should not be asserted during a BIST. The processor accesses the physical tag array during BISTs, and these accesses can conflict with inquire cycles.

Normal BIST

The normal BIST is invoked if INIT is asserted at the falling edge of RESET. The BIST runs tests on the internal hardware that exercise the following resources:

- Instruction cache:
  - Linear tag directory
  - Instruction array
  - Physical tag directory
- Data cache:
  - Linear tag directory
  - Data array
  - Physical tag directory
- Entry-point and instruction-decode PLAs
- Microcode ROM
- TLB

The BIST runs a linear feedback shift register (LFSR) signature test on the microcode ROM in parallel with a March C test on the instruction cache, data cache, and physical tags. This is followed by the March C test on the TLB arrays and then an

LFSR signature test on the PLA, in that order. Upon completion of the PLA test, the processor transfers the test result from an internal Hardware Debug Test (HDT) data register to the EAX register for external access, resets the internal microcode, and begins normal code fetching.

The result of the BIST can be accessed by reading the lower 9 bits of the EAX register. If the EAX register value is 0000_0000h, the test completed successfully. If the value is not zero, the non-zero bits indicate where the failure occurred, as shown in Table 4-2. The processor continues with its normal boot process after the BIST completes, whether the BIST passed or failed.

Table 4-2. BIST Error Bit Definition in EAX Register

Bit Number	Bit Value	
	0	1
31–9	No Error	Always 0
8	No Error	Data path
7	No Error	Instruction-cache instructions
6	No Error	Instruction-cache linear tags
5	No Error	Data-cache linear tags
4	No Error	PLA
3	No Error	Microcode ROM
2	No Error	Data-cache data
1	No Error	Instruction cache physical tags
0	No Error	Data-cache physical tags

Test Access Port (TAP) BIST

The TAP BIST performs all of the functions of the normal BIST, up to and including the PLA signature test, in the exact manner as the normal BIST. However, after the PLA test, the test result is not transferred to the EAX register.

The TAP BIST is started by loading and executing the RUNBIST instruction in the test access port, as described in "Boundary Scan Architecture Support" on page 87. When the RUNBIST instruction is executed, the processor enters into a reset mode that is identical to that entered when the RESET

signal is asserted. Upon completion of the TAP BIST, the result remains in the BIST result register for shifting out through the TDO signal. The TRST signal must be asserted or the TAP instruction must be changed in order to exit TAP BIST and return to normal operation.

Output-Float Test

The Output-Float Test mode is entered if FLUSH is asserted before the falling edge of RESET. This causes the processor to place all of its output and bidirectional signals in the high-impedance state. In this isolated state, system board traces and connections can be tested for integrity and driveability. The Output-Float Test mode can only be exited by asserting RESET again.

On the AMD-K5 and Pentium processors, FLUSH is an edge-triggered interrupt. On the 486 processor, however, the signal is a level-sensitive input.

Cache and TLB Testing

Cache and TLB testing is often done by the BIOS or operating system during power-up. These arrays can be tested using the Array Access Register (AAR). The following tests can be performed:

- *Data Cache*—8-Kbyte, 4-way, set associative
  - Data array
  - Linear-tag array
  - Physical-tag array
- *Instruction Cache*—16-Kbyte, 4-way, set associative
  - Instruction array
  - Linear-tag array
  - Physical-tag array
  - Valid-bit array
  - Branch-prediction bit array

- *4-Kbyte TLB*—128-entry, 4-way, set associative
  - Linear-tag array
  - Page array
- *4-Mbyte TLB*—4-entry, fully associative
  - Linear-tag array
  - Page array

Note: *For more information on cache arrays, see Appendix A.*

Array Access Register (AAR)

The 64-bit Array Access Register (AAR) is a model-specific register (MSR) that contains a 32-bit *array pointer*, which identifies the array location to be tested, and 32 bits of *array test data* to be read or written. The WRMSR and RDMSR instructions access the AAR when the ECX register contains the value 82h, as described on page 34. Figure 4-2 shows the format of the AAR.

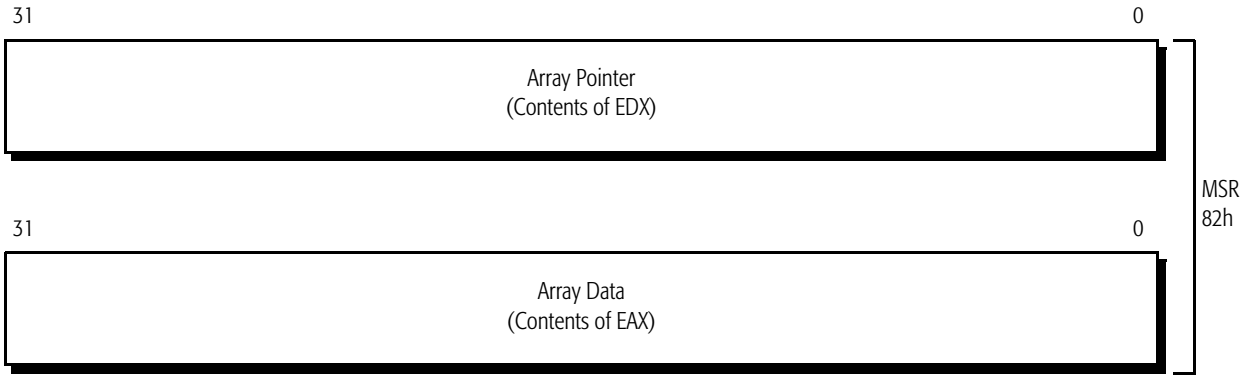


Figure 4-2. Array Access Register (AAR)

To read or write an array location, perform the following steps:

1. *ECX*—Enter 82h into ECX to access the 64-bit AAR.
2. *EDX*—Enter a 32-bit *array pointer* into EDX, as shown in Figures 4-3 through 4-8 (top).
3. *EAX*—Read or write 32 bits of *array test data* to or from EAX, as shown in Figures 4-3 through 4-8 (bottom).

Array Pointer

The array pointers entered in EDX (Figures 4-3 through 4-8, top) specify particular array locations. For example, in the data- and instruction-cache arrays, the way (or column) and set (or index) in the array pointer specifies a cache line in the 4-way, set-associative array. The array pointers for data-cache data and instruction-cache instructions also specify a *dword* location within that cache line. In the data cache, this dword is 32 bits of data, in the instruction cache, this dword is two instruction bytes plus their associated pre-decode bits. For the 4-Kbyte TLB, the way and set specify one of the 128 TLB entries. In 4-Mbyte TLB, one of only four entries is specified.

Bits 7–0 of every array pointer encode the *array ID*, which identifies the array to be accessed, as shown in Table 4-3. To simplify multiple accesses to an array, the contents of EDX are retained after the RDMSR instruction executes (EDX is normally cleared after a RDMSR instruction).

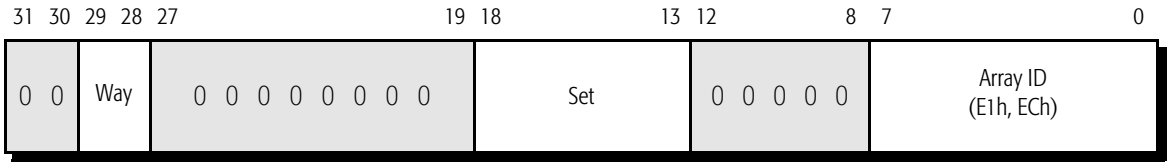
Table 4-3. Array IDs in Array Pointers

Array Pointer Bits 7–0	Accessed Array
E0h	Data Cache: Data
E1h	Data Cache: Linear Tag
ECh	Data Cache: Physical Tag
E4h	Instruction Cache: Instructions
E5h	Instruction Cache: Linear Tag
EDh	Instruction Cache: Physical Tag
E6h	Instruction Cache: Valid Bits
E7h	Instruction Cache: Branch-Prediction Bits
E8h	4-Kbyte TLB: Page
E9h	4-Kbyte TLB: Linear Tag
EAh	4-Mbyte TLB: Page
EBh	4-Mbyte TLB: Linear Tag

Array Test Data

EAX specifies the test data to be read or written with the RDMSR or WRMSR instruction (see Figures 4-3 through 4-8). For example, in Figure 4-3 (top) the array pointer in EDX specifies a way and set within the data-cache linear tag array (E1h in bits 7–0 of the array pointer) or the physical tag array (ECh in bits 7–0 of the array pointer). If the linear tag array (E1h) is accessed, the data read or written includes the tag and the status bits. The details of the valid fields in EAX are shown in Appendix A.

EDX: Array Pointer



EAX: Test Data

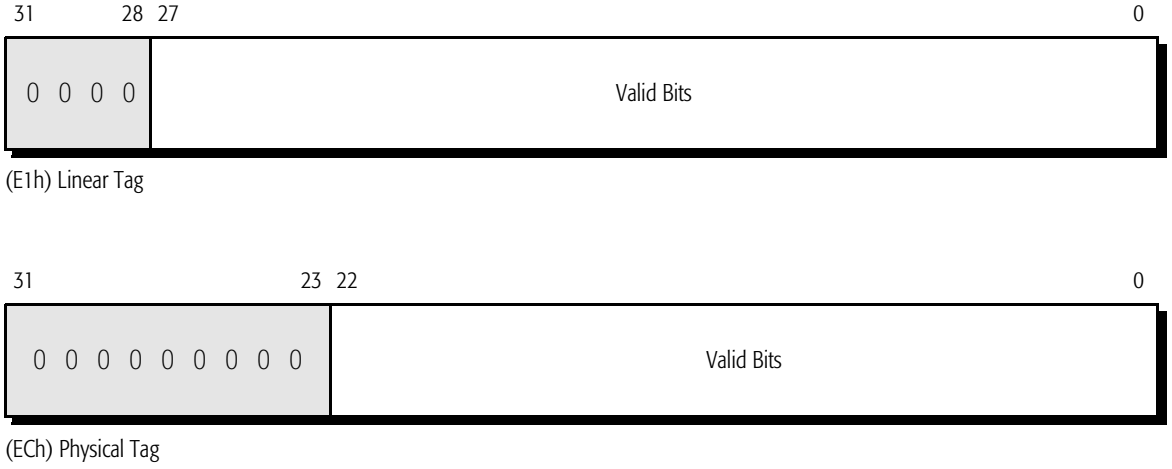
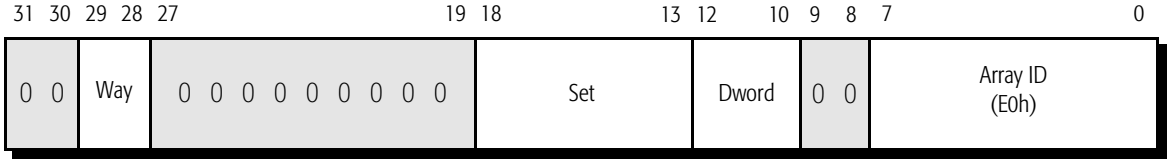
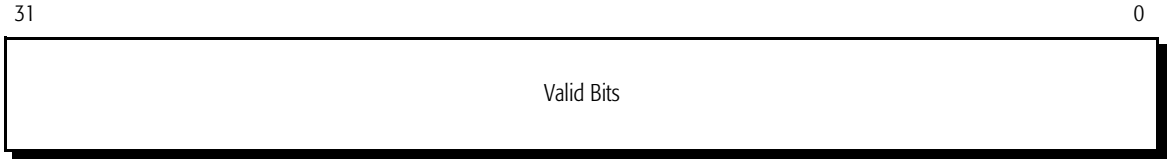


Figure 4-3. Test Formats: Data-Cache Tags

EDX: Array Pointer



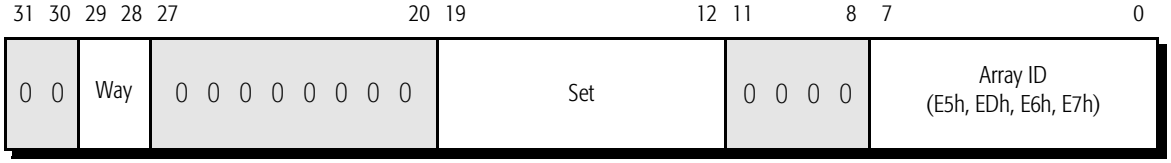
EAX: Test Data



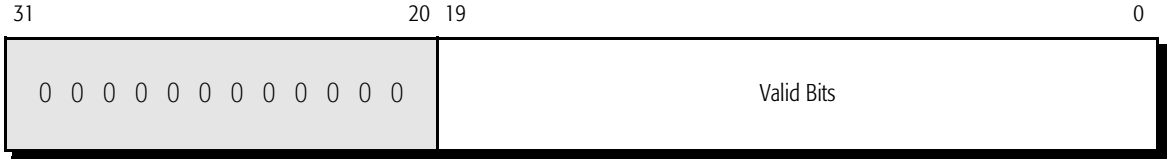
(E0h) Data

Figure 4-4. Test Formats: Data-Cache Data

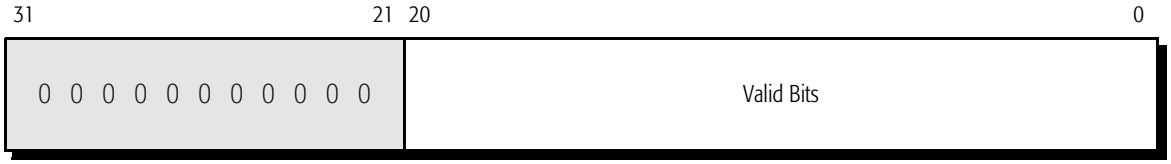
EDX: Array Pointer



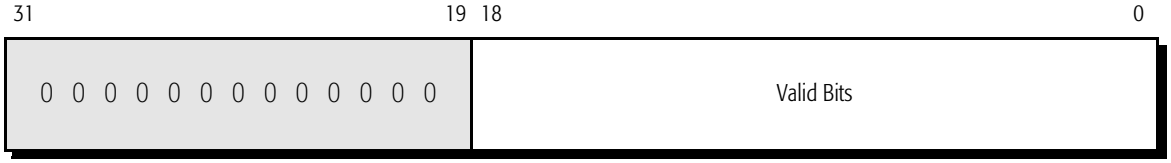
EAX: Test Data



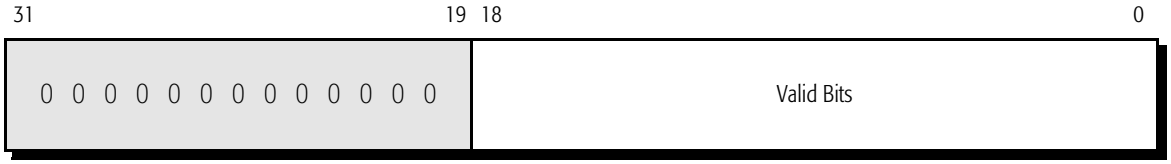
(E5h) Linear Tag



(EDh) Physical Tag



(E6h) Valid Bits

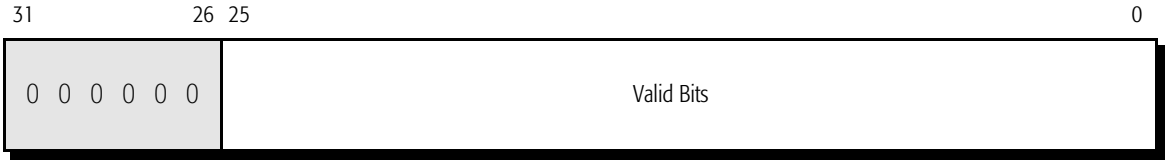


(E7h) Branch-Prediction Bits

Figure 4-5. Test Formats: Instruction-Cache Tags

EDX: Array Pointer

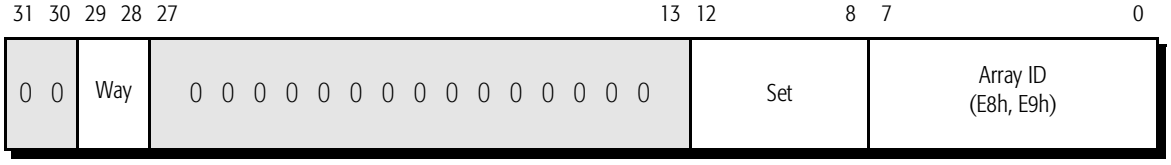
EAX: Test Data



(E4h) Instruction Bytes

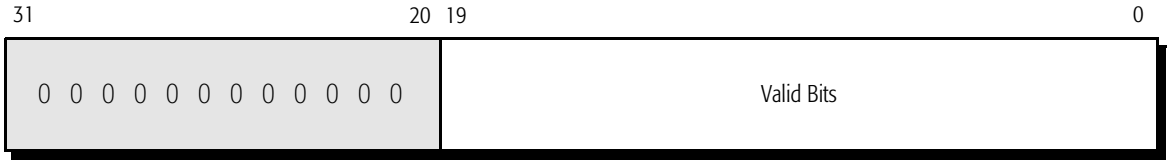
Figure 4-6. Test Formats: Instruction-Cache Instructions

EDX: Array Pointer



EAX: Test Data

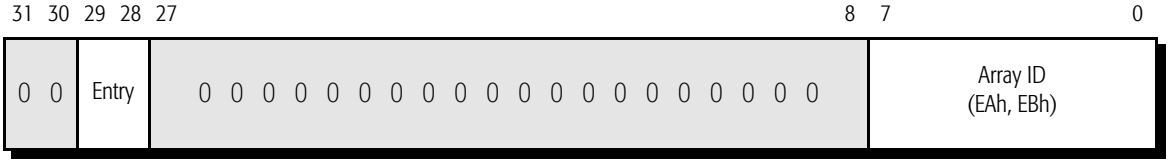
(E8h) 4-Kbyte Page and Status



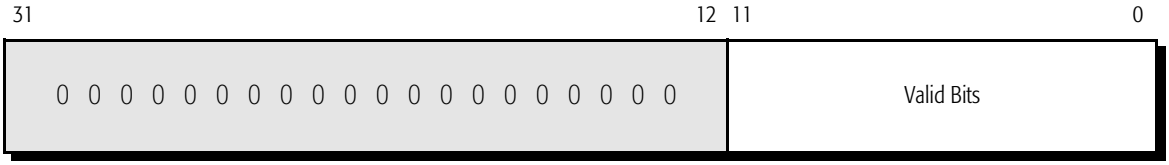
(E9h) 4-Kbyte Linear Tag

Figure 4-7. Test Formats: 4-Kbyte TLB

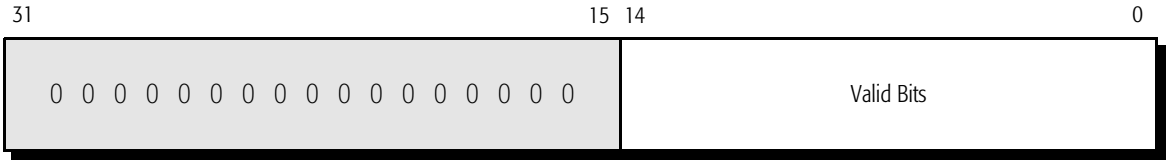
EDX: Array Pointer



EAX: Test Data



(EAh) 4-Mbyte Page and Status



(EBh) 4-Mbyte Linear Tag

Figure 4-8. Test Formats: 4-Mbyte TLB

Debug Registers

The processor implements the standard debug functions and registers—DR7–DR6 and DR3–DR0 (often called DR7–DR0)—that are available on the 486 processor, plus an I/O breakpoint extension.

Standard Debug Functions

The debug functions make the processor's state visible to debug software through four debug registers (DR3–DR0) that are accessed by MOV instructions. Accesses to memory addresses can be set as breakpoints in the instruction flow by invoking one of two debug exceptions (interrupt vectors 1 or 3) during instruction or data accesses to the addresses. The debug functions eliminate the need to embed breakpoints in code and allow debugging of ROM as well as RAM.

For details on the standard 486 debug functions and registers, see the AMD documentation on the Am486® processor or other commercial x86 literature.

I/O Breakpoint Extension

The processor supports an I/O breakpoint extension for breakpoints on I/O reads and writes. This function is enabled by setting bit 3 of CR4, as described in "Control Register 4 (CR4) Extensions" on page 2. When enabled, the I/O breakpoint function is invoked by the following:

- Entering the I/O port number as a breakpoint address (zero-extended to 32 bits) in one of the breakpoint registers, DR3–DR0
- Entering the bit pattern, 10b, in the corresponding 2-bit R/W field in DR7

All data breakpoints on the AMD-K5 processor are precise, including those encountered in repeated string operations, which trap after completing the iteration on which the breakpoint match occurs.

Enabled breakpoints slow the processor somewhat. When a data breakpoint is enabled, the processor disables its dual-issue load/store operations and performs only single-issue load/store operations. When an instruction breakpoint is enabled, instruction issue is completely serialized.

Debug Compatibility with Pentium Processor

The differences in debug functions between the AMD-K5 and Pentium processors are described in Appendix A of the *AMD-K5 Processor Technical Reference Manual,* order# 18524.

Branch Tracing

Branch tracing is enabled by writing bits 3–1 with 001b and setting bit 5 to 1 (disabling branch prediction) in the Hardware Configuration Register (HWCR), as described on page 71. When thus enabled, the processor drives two branch-trace message special bus cycles immediately after each taken branch instruction is executed. Both special bus cycles have a $\overline{BE7}$–$\overline{BE0}$ encoding of DFh (1101_1111b). The first special bus cycle identifies the branch source, the second identifies the branch target. The contents of the address and data bus during these special bus cycles are shown in Table 4-4.

The branch-trace message special bus cycles are different for the AMD-K5 and Pentium processors, although their $\overline{BE7}$–$\overline{BE0}$ encodings are the same.

Table 4-4. Branch-Trace Message Special Bus Cycle Fields

Signals	First Special Bus Cycle	Second Special Bus Cycle
A31	0 = first special bus cycle (source)	1 = second special bus cycle (target)
A30–A29	not valid	Operating Mode of Target: 11 = Virtual-8086 Mode 10 = Protected Mode 01 = not valid 00 = Real Mode
A28	not valid	Default Operand Size of Target Segment: 1 = 32-bit 0 = 16-bit
A27–A20	0	0
A19–A4	Code Segment (CS) selector of Branch Source.	Code Segment (CS) selector of Branch Target.
A3	0	0
D31–D0	EIP of Branch Source.	EIP of Branch Target.

Functional-Redundancy Checking

When $\overline{\text{FRCMC}}$ is asserted at RESET, the processor enters Functional-Redundancy Checking mode, as the checker, and reports checking errors on the $\overline{\text{IERR}}$ output. If $\overline{\text{FRCMC}}$ is negated at RESET, the processor operates normally, although it also behaves as the master in a functional-redundancy checking arrangement with a checker.

In the Functional-Redundancy Checking mode, two processors have their signals tied together. One processor (the master) operates normally. The other processor (the checker) has its output and bidirectional signals (except for TDO and $\overline{\text{IERR}}$) floated to detect the state of the master's signals. The master controls instruction fetching and the checker mimics its behavior by sampling the fetched instructions as they appear on the bus. Both processors execute the instructions in lock step. The checker compares the state of the master's output and bidirectional signals with the state that the checker itself would have driven for the same instruction stream.

Errors detected by the checker are reported on the $\overline{\text{IERR}}$ output of the checker. If a mismatch occurs on such a comparison, the checker asserts $\overline{\text{IERR}}$ for one clock, two clocks after the detection of the error. Both the master and the checker continue running the checking program after an error occurs. No action other than the assertion of $\overline{\text{IERR}}$ is taken by the processor. On the AMD-K5 processor, the $\overline{\text{IERR}}$ output is reserved solely for functional-redundancy checking. No other errors are reported on that output.

Functional-redundancy checking is typically implemented on single-processor, fault-monitoring systems (which actually have two processors). The master processor runs the operational programs and the checker processor is dedicated entirely to constant checking. In this arrangement, the test of accurate operation consists solely of reporting one or more errors. The particular type of error or the instruction causing an error is not reported. The arrangement works because the processor is entirely deterministic. Speculative prefetching, speculative execution, and cache replacement all occur in identical ways and at identical times on both processors if their signals are tied together so that they run the same program.

The Functional-Redundancy Checking mode can only be exited by the assertion of RESET. Functional-redundancy checking cannot be performed in the Hardware Debug Tool (HDT) mode. The assertion of $\overline{\text{FRCMC}}$ is not recognized while PRDY is asserted.

Boundary Scan Architecture Support

The AMD-K5 processor provides test features compatible with the Standard Test Access Port (TAP) and Boundary Scan Test Architecture as defined in the IEEE 1149.1-1990 JTAG Specification. The subsections in this topic include:

- Boundary Scan Test Functional Description
- Boundary Scan Architecture
- Registers
- The Test Access Port (TAP) Controller
- JTAG Register Organization

- JTAG Instructions

The external TAP interface consists of five pins:

- TCK: The Test Clock input provides the clock for the JTAG test logic.
- TMS: The Test Mode Select input enables TAP controller operations.
- TDI: The Test Data Input provides serial input to registers.
- TDO: The Test Data Output provides serial output from the registers; the signal is tri-stated except when in the Shift-DR or Shift-IR controller states.
- $\overline{\text{TRST}}$: The TAP Controller Reset input initializes the TAP controller when asserted Low.

The internal JTAG logic contains the elements listed below:

- The Test Access Port (TAP) Controller—Decodes the inputs on the Test Mode Select (TMS) line to control test operations. The TAP is a general-purpose port that provides access to the test support functions built into the AMD-K5 processor.
- Instruction Register—Accepts instructions from the Test Data Input (TDI) pin. The instruction codes select the specific test or debug operation to be performed or the test data register to be accessed.
- Implemented Test Data Registers—Boundary Scan Register, Device Identification Register, and Bypass Register. See "JTAG Register Organization" on page 91 for more information.

Note: *See Table 4-8 for more information.*

Boundary Scan Test Functional Description

The boundary scan testing uses a shift register, contained in a boundary scan cell, located between the core logic and the I/O buffers adjacent to each component pin. Signals at each input and output pin are controlled and observed using scan testing techniques. The boundary scan cells are interconnected to form a shift register chain. This register chain, called a Boundary Scan Register (BSR), constructs a serial path surrounding the core logic. This enables test data to be shifted through the

boundary scan path. When the system enters the Boundary Scan Test mode, the BSR chain is directed by a test program to pass data along the shift register path.

If all the components used to construct a circuit or PCB contain a boundary scan cell architecture, the resulting serial path can be used to perform component interconnect testing.

Boundary Scan Architecture

Boundary Scan architecture has four basic elements:

- Test Access Port (TAP)
- TAP Controller
- Instruction Register (IR). See"Instruction Register" on page 90 for more information.
- Test Data Registers. See "Registers" on page 90 for more information.

The Instruction and Test Data Registers have separate shift register access paths connected in parallel between the Test Data In (TDI) and Test Data Out (TDO) pins. Path selection and boundary scan cell operation is controlled by the TAP Controller. The controller initializes at start-up, but the Test Reset ($\overline{\text{TRST}}$) input can asynchronously reset the test logic, if required.

All system integrated circuit (IC) I/O signals are shifted in and out through the serial Test Data In and Test Data Out (TDI/TDO) path. The TAP Controller is enabled by the Test Mode Select (TMS) input. The Test Clock (TCK), obtained from a system level bus or Automatic Test Equipment (ATE), supplies the timing signal for data transfer and system architecture operation.

The dedicated TCK input enables the serial test data path between components to be used independently of component-specific system clocks. TCK also ensures that test data can be moved to or from a chip without changing the state of the on-chip system logic.

The TCK signal is driven by an independent 50% duty cycle clock (generated by the Automatic Test Equipment). If the TCK must be stopped (for example, if the ATE must retrieve

data from external memory and is unable to keep the clock running), it can be stopped at 0 or 1 indefinitely, without causing any change to the test logic state.

To ensure race-free operation, changes on the TAP's TMS input are clocked into the test logic. Changes on the TAP's TDI input are clocked into the selected register (Instruction or Test Data Register) on the rising edge of TCK. The contents of the selected register are shifted out onto the TAP output (TDO) on the falling edge of TCK.

Registers

Boundary scan architectural elements include an Instruction Register (IR) and a group of Test Data Registers (TDRs). These registers have separate shift-register-based serial access paths, connected in parallel between the TDI and TDO pins.

The TDRs are internal registers used by the Boundary Scan Architecture to process the test data. Each Test Data Register is addressed by an instruction scanned into the Instruction Register. The AMD-K5 processor includes the following TDRs:

- Bypass Register (BR). See "Bypass Register" on page 92.
- Boundary Scan Register (BSR). See "Boundary Scan Register" on page 91.
- Device Identification Register (DIR). See "Device Identification Register" on page 91.
- Built-In Self-Test Result Register (BISTRR). See "RUNBIST" on page 95.

Instruction Register

The 5-bit Instruction Register (IR) is a serial-in parallel-out register that includes five shift register-based cells for holding instruction data. The instruction determines which test to run, which data register to access, or both. When the TAP controller enters the Capture IR state, the processor loads the IDCODE instruction in the IR. Executing Shift IR starts instructions shifting into the instruction register on the rising edge of TCK. Executing Update-IR loads the instruction from the serial shift register to the parallel register.

The TAP controller is a synchronous, finite state machine that controls the test and debug logic sequence of operations. The TAP controller changes state in response to the rising edge of

TCK and defaults to the test logic reset state at power-up. Reinitialization to the test logic reset state is accomplished by holding the TMS pin High for five TCK periods.

JTAG Register Organization

All registers in the JTAG logic consist of the following two register ranks:

- A shift register
- A parallel output register fed by the shift register

Parallel input data is loaded into the shift register when the TAP controller exits the Capture state (Capture DR or Capture IR). The shift register then shifts data from TDI to TDO when in the Shift state (Shift DR or Shift IR). The output register holds the current data while new data is shifted into the shift register. The contents of the output register are updated when the TAP controller exits the Update state (Update DR or Update IR). The three registers described in this section are:

- Boundary Scan Register
- Device Identification Register
- Bypass Register

Boundary Scan Register

The Boundary Scan Register (BSR) is a 261-bit shift register with cells connected to all input and output pins and containing cells for tri-state I/O control. This enables serial data to be loaded into or read from the processor boundary scan area.

Output cells determine the value of the signal driven on the corresponding pin. Input cells only capture data. The EXTEST and SAMPLE/PRELOAD instructions can operate the BSR.

Device Identification Register

The format of the Device Identification Register (DIR) is shown in Table 4-5. The fields include the following values:

- *Version Number*—This is incremented by AMD manufacturing for each major revision of silicon.
- *Bond Option*—The two bits of the bond option depend on how the part is bonded at the factory.
- *Part Number*—This identifies the specific processor model.

- *Manufacturer*—This is actually only 11 bits (11–1). The least-significant bit, bit 0, is always set to 1, as specified by the IEEE standard.

Table 4-5. Test Access Port (TAP) ID Code

Version (Bits 31–28)	Bond Option (Bit 27	Unused (Bits 26–24)	Part Number (Bits 23–12)	Manufacturer (Bits 11–0)
Xh	Xb	000b	50Xh = Model 0 51Xh = Model 1	001h

Bypass Register

The Bypass Register, a 1-bit shift register, provides the shortest path between TDI and TDO. When the component is not performing a test operation, this path is selected to allow transfer of test data to and from other components on the board. The Bypass Register is also selected during the HIGHZ, ALL1, ALL0, and BYPASS tests and for any unused instruction codes.

Public Instructions

The processor supports all three IEEE-mandatory instructions (BYPASS, SAMPLE/PRELOAD, EXTEST), three IEEE-optional instructions (IDCODE, HIGHZ, RUNBIST), and three instructions unique to the AMD-K5 processor (ALL1, ALL0, USEHDT). Table 4-6 shows the complete set of public TAP instructions supported by the processor. The processor also implements several private manufacturing test instructions.

The IEEE standard describes the mandatory and optional instructions. The ALL1 and ALL0 instructions simply force all outputs and bidirectionals High or Low. The USEHDT instruction is described on page 112. Any instruction encodings not shown in Table 4-6 select the BYPASS instruction.

Table 4-6. Public TAP Instructions

Instruction	Encoding	Register	Description
EXTEST	00000	BSR	As defined by the IEEE standard
SAMPLE/ PRELOAD	00001	BSR	As defined by the IEEE standard
IDCODE	00010	DIR	As defined by the IEEE standard
HIGHZ	00011	BR	As defined by the IEEE standard
ALL1	00100	BR	Forces all outputs and bidirectionals High
ALL0	00101	BR	Forces all outputs and bidirectionals Low
USEHDT	00110	HDTR	Accesses the Hardware Debug Tool (HDT)[1] See page 112
RUNBIST	00111	BISTRR	As defined by the IEEE standard
BYPASS	11111	BR	As defined by the IEEE standard
BYPASS	undefined	BR	Undefined instruction encodings select the BYPASS instruction

Notes:

1. Documentation on the Hardware Debug Tool (HDT) is available from AMD under a nondisclosure agreement.

EXTEST

The EXTEST instruction permits circuits outside the component package to be tested. A common use of the EXTEST instruction is the testing of board interconnects. Boundary scan register cells at output pins are used to apply test stimuli, while those at input pins capture test results. Dependent on the value loaded into their control cell in the boundary scan register, the I/O pins are established as input or output. Inputs to the core logic retain the logic value set prior to execution of the EXTEST instruction. Upon exiting EXTEST, input pins are reconnected to the package pins.

SAMPLE/PRELOAD

There are two functions performed by the SAMPLE/PRELOAD instruction, as follows:

- Capturing an instantaneous picture of the normal operation of the device being tested. This function occurs if the instruction is executed while the TAP controller is in the Capture DR state and causes the Boundary Scan Register to sample the values present at the device pins.
- Preloading data to the device pins to be driven to the board by the EXTEST instruction. This function occurs if the instruction is executed while the TAP controller is in the Update DR state and causes data to be preloaded to the device pins from the Boundary Scan Register.

IDCODE The execution of the IDCODE instruction connects the device identification register between TDI and TDO. Upon such connection, the device identification code can be shifted out of the register.

HIGHZ This instruction forces all output and bidirectional pins into a tri-state condition. When this instruction is selected, the bypass register is selected for shifting between TDI and TDO. A signal called HIZEXT is responsible for forcing the tri-state to occur. This signal is generated in the TAP block, underneath JTAG_BIST, and goes to the PAD_TOP block.

ALL1 This instruction forces all output and bidirectional pins to a High logic level.

The ALL1 instruction, like the HIGHZ instruction selects the bypass register for shifting between TDI and TDO. There is a signal called ALL1 that is responsible for forcing the pins to a High state. This signal is generated in the TAP block underneath JTAG_BIST and goes to the PAD_TOP block. In the PAD_TOP block, this signal goes to boundary scan cells called BSLCD_OUT. The DOUT pins of the BSLCD_OUT cells are forced High when ALL1 is High. The SELPDR signal selects the boundary scan cells as the source for driving the outputs, if the SELPDR signal is High. The SELPDR signal is also generated in the TAP block underneath JTAG_BIST and goes to the PAD_TOP block.

ALL0 This instruction forces all output and bidirectional pins to a Low logic level.

The ALL0 instruction, like the HIGHZ instruction, selects the bypass register for shifting between TDI and TDO. There is a signal called ALL0 that is responsible for forcing the pins to a Low state. This signal is generated in the TAP block underneath JTAG_BIST and goes to the PAD_TOP block. In the PAD_TOP block, this signal goes to boundary scan cells called BSLCD_OUT. The DOUT pins of the BSLCD_OUT cells are forced Low when ALL0 is High. The SELPDR signal selects the boundary scan cells as the source for driving the outputs, if the SELPDR signal is High. The SELPDR signal is also generated in the TAP block underneath JTAG_BIST and goes to the PAD_TOP block.

RUNBIST

This version of BIST is similar to the normal BIST mode, except that it is started by shifting in a TAP instruction. This instruction should behave according to the rules of the IEEE 1149.1 definition of RUNBIST.

When the RUNBIST instruction is updated into the instruction register, a signal from the TAP_RTL block called JTGBIST is asserted High. This signal goes to the PAD_TOP and TESTCTRL blocks. In PAD_TOP, this signal goes to the BRNBIST block and causes both INIT_SAMP and RUNBIST to be asserted. To the rest of the chip, it looks like a normal BIST operation is taking place. The JTGBIST signal also goes to the TESTCTRL block so that the BIST controller knows that the BIST operation was initiated from the TAP controller. This is necessary because the BIST results do not get transferred to the EAX register in this mode of operation. The JTAG_BIST block also asserts the RESET_TAP pin to the CLOCKS block for 15 system clock cycles, in order to fake an external reset.

The pattern that is shifted into the boundary scan ring, prior to the selection of the RUNBIST instruction, is driven at output and bidirectional cells during the duration of the instruction. The results of the execution of RUNBIST are saved in the BIST results register, which is 9 bits long and looks like the least significant 9 bits in the EAX register. This register is selected for shifting between TDI and TDO and can be shifted out after the completion of BIST. Bit 0 (ICACHE data status) is shifted out first. The BIST results should be independent of signals received at non-clock input pins (except for RESET).

BYPASS

The execution of the BYPASS instruction connects the bypass register between TDI and TDO, bypassing the test logic. Because of the pull-up resistor on the TDI input, the bypass register is selected if there is an open circuit in the board-level test data path following an instruction scan cycle. Any unused instruction bit patterns cause the bypass register to be selected for shifting between TDI and TDO.

The control bits listed in Table 4-8 have the characteristics described in Table 4-7.

Table 4-7. Control Bit Definitions

Bit	Definition
144	Controls the direction of the Data bus (D63–D0). If the bit is set to 1, the bus acts as an input. If the bit is set to 0, the bus acts as an output.
213	Controls the direction of the Address bus (A31–A3) and Address Parity (AP). If the bit is set to 1, the bus acts as an input. If the bit is set to 0, the bus acts as an output.
257	Controls pins that can be tri-stated, but these pins never act as inputs. If the bit is set to 1, the pin is tri-stated. If the bit is set to 0, the pin acts as an output.

Table 4-8. Boundary Scan Register Bit Definitions (Model 0)

Bit	Pin Name	Comments
0	DP7	Output Cell: Controlled by bit 144
1	DP7	Input Cell
2	D63	Output Cell: Controlled by bit 144
3	D63	Input Cell
4	D62	Output Cell: Controlled by bit 144
5	D62	Input Cell
6	D61	Output Cell: Controlled by bit 144
7	D61	Input Cell
8	D60	Output Cell: Controlled by bit 144
9	D60	Input Cell
10	D59	Output Cell: Controlled by bit 144
11	D59	Input Cell
12	D58	Output Cell: Controlled by bit 144
13	D58	Input Cell
14	D57	Output Cell: Controlled by bit 144
15	D57	Input Cell
16	D56	Output Cell: Controlled by bit 144
17	D56	Input Cell
18	DP6	Output Cell: Controlled by bit 144
19	DP6	Input Cell
20	D55	Output Cell: Controlled by bit 144

Table 4-8. Boundary Scan Register Bit Definitions (Model 0) (continued)

Bit	Pin Name	Comments
21	D55	Input Cell
22	D54	Output Cell: Controlled by bit 144
23	D54	Input Cell
24	D53	Output Cell: Controlled by bit 144
25	D53	Input Cell
26	D52	Output Cell: Controlled by bit 144
27	D52	Input Cell
28	D51	Output Cell: Controlled by bit 144
29	D51	Input Cell
30	D50	Output Cell: Controlled by bit 144
31	D50	Input Cell
32	D49	Output Cell: Controlled by bit 144
33	D49	Input Cell
34	D48	Output Cell: Controlled by bit 144
35	D48	Input Cell
36	DP5	Output Cell: Controlled by bit 144
37	DP5	Input Cell
38	D47	Output Cell: Controlled by bit 144
39	D47	Input Cell
40	D46	Output Cell: Controlled by bit 144
41	D46	Input Cell
42	D45	Output Cell: Controlled by bit 144
43	D45	Input Cell
44	D44	Output Cell: Controlled by bit 144
45	D44	Input Cell
46	D43	Output Cell: Controlled by bit 144
47	D43	Input Cell
48	D42	Output Cell: Controlled by bit 144
49	D42	Input Cell
50	D41	Output Cell: Controlled by bit 144
51	D41	Input Cell
52	D40	Output Cell: Controlled by bit 144
53	D40	Input Cell
54	DP4	Output Cell: Controlled by bit 144

Table 4-8. Boundary Scan Register Bit Definitions (Model 0) (continued)

Bit	Pin Name	Comments
55	DP4	Input Cell
56	D39	Output Cell: Controlled by bit 144
57	D39	Input Cell
58	D38	Output Cell: Controlled by bit 144
59	D38	Input Cell
60	D37	Output Cell: Controlled by bit 144
61	D37	Input Cell
62	D36	Output Cell: Controlled by bit 144
63	D36	Input Cell
64	D35	Output Cell: Controlled by bit 144
65	D35	Input Cell
66	D34	Output Cell: Controlled by bit 144
67	D34	Input Cell
68	D33	Output Cell: Controlled by bit 144
69	D33	Input Cell
70	D32	Output Cell: Controlled by bit 144
71	D32	Input Cell
72	DP3	Output Cell: Controlled by bit 144
73	DP3	Input Cell
74	D31	Output Cell: Controlled by bit 144
75	D31	Input Cell
76	D30	Output Cell: Controlled by bit 144
77	D30	Input Cell
78	D29	Output Cell: Controlled by bit 144
79	D29	Input Cell
80	D28	Output Cell: Controlled by bit 144
81	D28	Input Cell
82	D27	Output Cell: Controlled by bit 144
83	D27	Input Cell
84	D26	Output Cell: Controlled by bit 144
85	D26	Input Cell
86	D25	Output Cell: Controlled by bit 144
87	D25	Input Cell
88	D24	Output Cell: Controlled by bit 144

Table 4-8. Boundary Scan Register Bit Definitions (Model 0) (continued)

Bit	Pin Name	Comments
89	D24	Input Cell
90	DP2	Output Cell: Controlled by bit 144
91	DP2	Input Cell
92	D23	Output Cell: Controlled by bit 144
93	D23	Input Cell
94	D22	Output Cell: Controlled by bit 144
95	D22	Input Cell
96	D21	Output Cell: Controlled by bit 144
97	D21	Input Cell
98	D20	Output Cell: Controlled by bit 144
99	D20	Input Cell
100	D19	Output Cell: Controlled by bit 144
101	D19	Input Cell
102	D18	Output Cell: Controlled by bit 144
103	D18	Input Cell
104	D17	Output Cell: Controlled by bit 144
105	D17	Input Cell
106	D16	Output Cell: Controlled by bit 144
107	D16	Input Cell
108	DP1	Output Cell: Controlled by bit 144
109	DP1	Input Cell
110	D15	Output Cell: Controlled by bit 144
111	D15	Input Cell
112	D14	Output Cell: Controlled by bit 144
113	D14	Input Cell
114	D13	Output Cell: Controlled by bit 144
115	D13	Input Cell
116	D12	Output Cell: Controlled by bit 144
117	D12	Input Cell
118	D11	Output Cell: Controlled by bit 144
119	D11	Input Cell
120	D10	Output Cell: Controlled by bit 144
121	D10	Input Cell
122	D9	Output Cell: Controlled by bit 144

Table 4-8. Boundary Scan Register Bit Definitions (Model 0) (continued)

Bit	Pin Name	Comments
123	D9	Input Cell
124	D8	Output Cell: Controlled by bit 144
125	D8	Input Cell
126	DP	Output Cell: Controlled by bit 144
127	DP	Input Cell
128	D7	Output Cell: Controlled by bit 144
129	D7	Input Cell
130	D6	Output Cell: Controlled by bit 144
131	D6	Input Cell
132	D5	Output Cell: Controlled by bit 144
133	D5	Input Cell
134	D4	Output Cell: Controlled by bit 144
135	D4	Input Cell
136	D3	Output Cell: Controlled by bit 144
137	D3	Input Cell
138	D2	Output Cell: Controlled by bit 144
139	D2	Input Cell
140	D1	Output Cell: Controlled by bit 144
141	D1	Input Cell
142	D0	Output Cell: Controlled by bit 144
143	D0	Input Cell
144	Control	Direction Control. See Table 4-7.
145	$\overline{\text{STPLK}}$	Input Cell
146	$\overline{\text{FRCMC}}$	Input Cell
147	$\overline{\text{PEN}}$	Input Cell
148	$\overline{\text{IGNNE}}$	Input Cell
149	BF	Input Cell
150	INIT	Input Cell
151	$\overline{\text{SMI}}$	Input Cell
152	R/$\overline{\text{S}}$	Input Cell
153	NMI	Input Cell
154	INTR	Input Cell
155	A21	Output Cell: Controlled by bit 213
156	A21	Input Cell

Table 4-8. Boundary Scan Register Bit Definitions (Model 0) (continued)

Bit	Pin Name	Comments
157	A22	Output Cell: Controlled by bit 213
158	A22	Input Cell
159	A23	Output Cell: Controlled by bit 213
160	A23	Input Cell
161	A24	Output Cell: Controlled by bit 213
162	A24	Input Cell
163	A25	Output Cell: Controlled by bit 213
164	A25	Input Cell
165	A26	Output Cell: Controlled by bit 213
166	A26	Input Cell
167	A27	Output Cell: Controlled by bit 213
168	A27	Input Cell
169	A28	Output Cell: Controlled by bit 213
170	A28	Input Cell
171	A29	Output Cell: Controlled by bit 213
172	A29	Input Cell
173	A30	Output Cell: Controlled by bit 213
174	A30	Input Cell
175	A31	Output Cell: Controlled by bit 213
176	A31	Input Cell
177	A3	Output Cell: Controlled by bit 213
178	A3	Input Cell
179	A4	Output Cell: Controlled by bit 213
180	A4	Input Cell
181	A5	Output Cell: Controlled by bit 213
182	A5	Input Cell
183	A6	Output Cell: Controlled by bit 213
184	A6	Input Cell
185	A7	Output Cell: Controlled by bit 213
186	A7	Input Cell
187	A8	Output Cell: Controlled by bit 213
188	A8	Input Cell
189	A9	Output Cell: Controlled by bit 213
190	A9	Input Cell

Table 4-8. Boundary Scan Register Bit Definitions (Model 0) (continued)

Bit	Pin Name	Comments
191	A10	Output Cell: Controlled by bit 213
192	A10	Input Cell
193	A11	Output Cell: Controlled by bit 213
194	A11	Input Cell
195	A12	Output Cell: Controlled by bit 213
196	A12	Input Cell
197	A13	Output Cell: Controlled by bit 213
198	A13	Input Cell
199	A14	Output Cell: Controlled by bit 213
200	A14	Input Cell
201	A15	Output Cell: Controlled by bit 213
202	A15	Input Cell
203	A16	Output Cell: Controlled by bit 213
204	A16	Input Cell
205	A17	Output Cell: Controlled by bit 213
206	A17	Input Cell
207	A18	Output Cell: Controlled by bit 213
208	A18	Input Cell
209	A19	Output Cell: Controlled by bit 213
210	A19	Input Cell
211	A20	Output Cell: Controlled by bit 213
212	A20	Input Cell
213	Control	Direction Control. See Table 4-7.
214	SCYC	Output Cell: Controlled by bit 257
215	RESET	Input Cell
216	$\overline{BE7}$	Output Cell: Controlled by bit 257
217	$\overline{BE6}$	Output Cell: Controlled by bit 257
218	$\overline{BE5}$	Output Cell: Controlled by bit 257
219	$\overline{BE4}$	Output Cell: Controlled by bit 257
220	$\overline{BE3}$	Output Cell: Controlled by bit 257
221	$\overline{BE2}$	Output Cell: Controlled by bit 257
222	$\overline{BE1}$	Output Cell: Controlled by bit 257
223	$\overline{BE0}$	Output Cell: Controlled by bit 257
224	W/$\overline{R}$	Output Cell: Controlled by bit 257

Table 4-8. Boundary Scan Register Bit Definitions (Model 0) (continued)

Bit	Pin Name	Comments
225	$\overline{\text{HIT}}$	Output Cell
226	CLK	Clock
227	$\overline{\text{ADSC}}$	Output Cell: Controlled by bit 257
228	$\overline{\text{ADS}}$	Output Cell: Controlled by bit 257
229	$\overline{\text{CACHE}}$	Output Cell: Controlled by bit 257
230	$\overline{\text{BRDYC}}$	Input Cell
231	$\overline{\text{BRDY}}$	Input Cell
232	$\overline{\text{EADS}}$	Input Cell
233	PWT	Output Cell: Controlled by bit 257
234	$\overline{\text{LOCK}}$	Output Cell: Controlled by bit 257
235	PCD	Output Cell: Controlled by bit 257
236	WB/$\overline{\text{WT}}$	Input Cell
237	$\overline{\text{HITM}}$	Output Cell
238	$\overline{\text{KEN}}$	Input Cell
239	AHOLD	Input Cell
240	$\overline{\text{BOFF}}$	Input Cell
241	HLDA	Output Cell
242	HOLD	Input Cell
243	$\overline{\text{NA}}$	Input Cell
244	$\overline{\text{EWBE}}$	Input Cell
245	M/$\overline{\text{IO}}$	Output Cell: Controlled by bit 257
246	$\overline{\text{FLUSH}}$	Input Cell
247	$\overline{\text{A20M}}$	Input Cell
248	$\overline{\text{BUSCHK}}$	Input Cell
249	AP	Output Cell: Controlled by bit 213
250	AP	Input Cell
251	D/$\overline{\text{C}}$	Output Cell: Controlled by bit 257
252	BREQ	Output Cell
253	$\overline{\text{SMIACT}}$	Output Cell
254	$\overline{\text{PCHK}}$	Output Cell
255	$\overline{\text{APCHK}}$	Output Cell
256	PRDY	Output Cell
257	Control	Direction Control. See Table 4-7.
258	INV	Input Cell

Table 4-8. Boundary Scan Register Bit Definitions (Model 0) (continued)

Bit	Pin Name	Comments
259	$\overline{\text{FERR}}$	Output Cell
260	$\overline{\text{IERR}}$	Output Cell

Table 4-9. Boundary Scan Register Bit Definitions (Model 1)

Bit	Pin Name	Comments
0	DP7	Output Cell: Controlled by bit 144
1	DP7	Input Cell
2	D63	Output Cell: Controlled by bit 144
3	D63	Input Cell
4	D62	Output Cell: Controlled by bit 144
5	D62	Input Cell
6	D61	Output Cell: Controlled by bit 144
7	D61	Input Cell
8	D60	Output Cell: Controlled by bit 144
9	D60	Input Cell
10	D59	Output Cell: Controlled by bit 144
11	D59	Input Cell
12	D58	Output Cell: Controlled by bit 144
13	D58	Input Cell
14	D57	Output Cell: Controlled by bit 144
15	D57	Input Cell
16	D56	Output Cell: Controlled by bit 144
17	D56	Input Cell
18	DP6	Output Cell: Controlled by bit 144
19	DP6	Input Cell
20	D55	Output Cell: Controlled by bit 144
21	D55	Input Cell
22	D54	Output Cell: Controlled by bit 144
23	D54	Input Cell
24	D53	Output Cell: Controlled by bit 144
25	D53	Input Cell
26	D52	Output Cell: Controlled by bit 144
27	D52	Input Cell

Table 4-9. Boundary Scan Register Bit Definitions (Model 1) (continued)

Bit	Pin Name	Comments
28	D51	Output Cell: Controlled by bit 144
29	D51	Input Cell
30	D50	Output Cell: Controlled by bit 144
31	D50	Input Cell
32	D49	Output Cell: Controlled by bit 144
33	D49	Input Cell
34	D48	Output Cell: Controlled by bit 144
35	D48	Input Cell
36	DP5	Output Cell: Controlled by bit 144
37	DP5	Input Cell
38	D47	Output Cell: Controlled by bit 144
39	D47	Input Cell
40	D46	Output Cell: Controlled by bit 144
41	D46	Input Cell
42	D45	Output Cell: Controlled by bit 144
43	D45	Input Cell
44	D44	Output Cell: Controlled by bit 144
45	D44	Input Cell
46	D43	Output Cell: Controlled by bit 144
47	D43	Input Cell
48	D42	Output Cell: Controlled by bit 144
49	D42	Input Cell
50	D41	Output Cell: Controlled by bit 144
51	D41	Input Cell
52	D40	Output Cell: Controlled by bit 144
53	D40	Input Cell
54	DP4	Output Cell: Controlled by bit 144
55	DP4	Input Cell
56	D39	Output Cell: Controlled by bit 144
57	D39	Input Cell
58	D38	Output Cell: Controlled by bit 144
59	D38	Input Cell
60	D37	Output Cell: Controlled by bit 144
61	D37	Input Cell

Table 4-9. Boundary Scan Register Bit Definitions (Model 1) (continued)

Bit	Pin Name	Comments
62	D36	Output Cell: Controlled by bit 144
63	D36	Input Cell
64	D35	Output Cell: Controlled by bit 144
65	D35	Input Cell
66	D34	Output Cell: Controlled by bit 144
67	D34	Input Cell
68	D33	Output Cell: Controlled by bit 144
69	D33	Input Cell
70	D32	Output Cell: Controlled by bit 144
71	D32	Input Cell
72	DP3	Output Cell: Controlled by bit 144
73	DP3	Input Cell
74	D31	Output Cell: Controlled by bit 144
75	D31	Input Cell
76	D30	Output Cell: Controlled by bit 144
77	D30	Input Cell
78	D29	Output Cell: Controlled by bit 144
79	D29	Input Cell
80	D28	Output Cell: Controlled by bit 144
81	D28	Input Cell
82	D27	Output Cell: Controlled by bit 144
83	D27	Input Cell
84	D26	Output Cell: Controlled by bit 144
85	D26	Input Cell
86	D25	Output Cell: Controlled by bit 144
87	D25	Input Cell
88	D24	Output Cell: Controlled by bit 144
89	D24	Input Cell
90	DP2	Output Cell: Controlled by bit 144
91	DP2	Input Cell
92	D23	Output Cell: Controlled by bit 144
93	D23	Input Cell
94	D22	Output Cell: Controlled by bit 144
95	D22	Input Cell

Table 4-9. Boundary Scan Register Bit Definitions (Model 1) (continued)

Bit	Pin Name	Comments
96	D21	Output Cell: Controlled by bit 144
97	D21	Input Cell
98	D20	Output Cell: Controlled by bit 144
99	D20	Input Cell
100	D19	Output Cell: Controlled by bit 144
101	D19	Input Cell
102	D18	Output Cell: Controlled by bit 144
103	D18	Input Cell
104	D17	Output Cell: Controlled by bit 144
105	D17	Input Cell
106	D16	Output Cell: Controlled by bit 144
107	D16	Input Cell
108	DP1	Output Cell: Controlled by bit 144
109	DP1	Input Cell
110	D15	Output Cell: Controlled by bit 144
111	D15	Input Cell
112	D14	Output Cell: Controlled by bit 144
113	D14	Input Cell
114	D13	Output Cell: Controlled by bit 144
115	D13	Input Cell
116	D12	Output Cell: Controlled by bit 144
117	D12	Input Cell
118	D11	Output Cell: Controlled by bit 144
119	D11	Input Cell
120	D10	Output Cell: Controlled by bit 144
121	D10	Input Cell
122	D9	Output Cell: Controlled by bit 144
123	D9	Input Cell
124	D8	Output Cell: Controlled by bit 144
125	D8	Input Cell
126	DP	Output Cell: Controlled by bit 144
127	DP	Input Cell
128	D7	Output Cell: Controlled by bit 144
129	D7	Input Cell

Table 4-9. Boundary Scan Register Bit Definitions (Model 1) (continued)

Bit	Pin Name	Comments
130	D6	Output Cell: Controlled by bit 144
131	D6	Input Cell
132	D5	Output Cell: Controlled by bit 144
133	D5	Input Cell
134	D4	Output Cell: Controlled by bit 144
135	D4	Input Cell
136	D3	Output Cell: Controlled by bit 144
137	D3	Input Cell
138	D2	Output Cell: Controlled by bit 144
139	D2	Input Cell
140	D1	Output Cell: Controlled by bit 144
141	D1	Input Cell
142	D0	Output Cell: Controlled by bit 144
143	D0	Input Cell
144	Control	Direction Control. See Table 4-7.
145	$\overline{\text{STPLK}}$	Input Cell
146	BF1	Input Cell
147	$\overline{\text{FRCMC}}$	Input Cell
148	$\overline{\text{PEN}}$	Input Cell
149	$\overline{\text{IGNNE}}$	Input Cell
150	BF0	Input Cell
151	INIT	Input Cell
152	$\overline{\text{SMI}}$	Input Cell
153	R/$\overline{\text{S}}$	Input Cell
154	NMI	Input Cell
155	INTR	Input Cell
156	A21	Output Cell: Controlled by bit 213
157	A21	Input Cell
158	A22	Output Cell: Controlled by bit 213
159	A22	Input Cell
160	A23	Output Cell: Controlled by bit 213
161	A23	Input Cell
162	A24	Output Cell: Controlled by bit 213
163	A24	Input Cell

Table 4-9. Boundary Scan Register Bit Definitions (Model 1) (continued)

Bit	Pin Name	Comments
164	A25	Output Cell: Controlled by bit 213
165	A25	Input Cell
166	A26	Output Cell: Controlled by bit 213
167	A26	Input Cell
168	A27	Output Cell: Controlled by bit 213
169	A27	Input Cell
170	A28	Output Cell: Controlled by bit 213
171	A28	Input Cell
172	A29	Output Cell: Controlled by bit 213
173	A29	Input Cell
174	A30	Output Cell: Controlled by bit 213
175	A30	Input Cell
176	A31	Output Cell: Controlled by bit 213
177	A31	Input Cell
178	A3	Output Cell: Controlled by bit 213
179	A3	Input Cell
180	A4	Output Cell: Controlled by bit 213
181	A4	Input Cell
182	A5	Output Cell: Controlled by bit 213
183	A5	Input Cell
184	A6	Output Cell: Controlled by bit 213
185	A6	Input Cell
186	A7	Output Cell: Controlled by bit 213
187	A7	Input Cell
188	A8	Output Cell: Controlled by bit 213
189	A8	Input Cell
190	A9	Output Cell: Controlled by bit 213
191	A9	Input Cell
192	A10	Output Cell: Controlled by bit 213
193	A10	Input Cell
194	A11	Output Cell: Controlled by bit 213
195	A11	Input Cell
196	A12	Output Cell: Controlled by bit 213
197	A12	Input Cell

Table 4-9. Boundary Scan Register Bit Definitions (Model 1) (continued)

Bit	Pin Name	Comments
198	A13	Output Cell: Controlled by bit 213
199	A13	Input Cell
200	A14	Output Cell: Controlled by bit 213
201	A14	Input Cell
202	A15	Output Cell: Controlled by bit 213
203	A15	Input Cell
204	A16	Output Cell: Controlled by bit 213
205	A16	Input Cell
206	A17	Output Cell: Controlled by bit 213
207	A17	Input Cell
208	A18	Output Cell: Controlled by bit 213
209	A18	Input Cell
210	A19	Output Cell: Controlled by bit 213
211	A19	Input Cell
212	A20	Output Cell: Controlled by bit 213
213	A20	Input Cell
214	Control	Direction Control. See Table 4-7.
215	SCYC	Output Cell: Controlled by bit 257
216	RESET	Input Cell
217	$\overline{BE7}$	Output Cell: Controlled by bit 257
218	$\overline{BE6}$	Output Cell: Controlled by bit 257
219	$\overline{BE5}$	Output Cell: Controlled by bit 257
220	$\overline{BE4}$	Output Cell: Controlled by bit 257
221	$\overline{BE3}$	Output Cell: Controlled by bit 257
222	$\overline{BE2}$	Output Cell: Controlled by bit 257
223	$\overline{BE1}$	Output Cell: Controlled by bit 257
224	$\overline{BE0}$	Output Cell: Controlled by bit 257
225	W/$\overline{R}$	Output Cell: Controlled by bit 257
226	$\overline{HIT}$	Output Cell
227	CLK	Clock
228	$\overline{ADSC}$	Output Cell: Controlled by bit 257
229	$\overline{ADS}$	Output Cell: Controlled by bit 257
230	$\overline{CACHE}$	Output Cell: Controlled by bit 257
231	$\overline{BRDYC}$	Input Cell

Table 4-9. Boundary Scan Register Bit Definitions (Model 1) (continued)

Bit	Pin Name	Comments
232	$\overline{\text{BRDY}}$	Input Cell
233	$\overline{\text{EADS}}$	Input Cell
234	PWT	Output Cell: Controlled by bit 257
235	$\overline{\text{LOCK}}$	Output Cell: Controlled by bit 257
236	PCD	Output Cell: Controlled by bit 257
237	WB/$\overline{\text{WT}}$	Input Cell
238	$\overline{\text{HITM}}$	Output Cell
239	$\overline{\text{KEN}}$	Input Cell
240	AHOLD	Input Cell
241	$\overline{\text{BOFF}}$	Input Cell
242	HLDA	Output Cell
243	HOLD	Input Cell
244	$\overline{\text{NA}}$	Input Cell
245	$\overline{\text{EWBE}}$	Input Cell
246	M/$\overline{\text{IO}}$	Output Cell: Controlled by bit 257
247	$\overline{\text{FLUSH}}$	Input Cell
248	$\overline{\text{A20M}}$	Input Cell
249	$\overline{\text{BUSCHK}}$	Input Cell
250	AP	Output Cell: Controlled by bit 213
251	AP	Input Cell
252	D/$\overline{\text{C}}$	Output Cell: Controlled by bit 257
253	BREQ	Output Cell
254	$\overline{\text{SMIACT}}$	Output Cell
255	$\overline{\text{PCHK}}$	Output Cell
256	$\overline{\text{APCHK}}$	Output Cell
257	PRDY	Output Cell
258	Control	Direction Control. See Table 4-7.
259	INV	Input Cell
260	$\overline{\text{FERR}}$	Output Cell
261	$\overline{\text{IERR}}$	Output Cell

Hardware Debug Tool (HDT)

The Hardware Debug Tool (HDT)—sometimes referred to as the debug port or Probe Mode—is a collection of signals, registers, and processor microcode that is enabled when external debug logic drives R/$\overline{S}$ Low or loads the processor's Test Access Port (TAP) instruction register with the USEHDT instruction.

Documentation on the HDT is available under nondisclosure agreement to test and debug developers. For information, contact your local AMD sales representative or field application engineer.

Appendix A

Cache

The individual locations of all SRAM arrays on the AMD-K5 microprocessor are accessible with the RDMSR and WRMSR instructions. To access an array location, set up the Array Access MSR code (82h) in ECX, and the array pointer (described below) in EDX. EAX holds the data to be read or written.

A.1 Array Pointer Formats

Note: *The term "column" in this description refers to the "way"—one of the four blocks in the 4-way associative set at a particular index.*

The array pointer in EDX specifies a particular array, column, index, and possibly word or dword, depending on the array to be accessed.

Table A-1. Cache Array Pointer Formats

Bits	29–28	27–20	19	18–13	12	11	10	9	8	7–0
DCACHE tag array	Column	NA	NA	tag array index	NA	NA	NA	NA	NA	array to be accessed
DCACHE dword and data array index in block	Column	NA	NA	data array index	DCACHE dword index into the block			NA	NA	array to be accessed
ICACHE index and word—Model 0	Column	NA	ICACHE index for all ICACHE arrays			ICACHE word (two instruction bytes + associated prede-code information			NA	array to be accessed
ICACHE index and word—Model 1	Column	NA	ICACHE index for all ICACHE arrays		ICACHE Packet Select	NA	ICACHE word (two instruction bytes + associated prede-code information			array to be accessed
4-Kbyte TLB index	Column	NA	NA	NA	TLB index					array to be accessed
4-Mbyte TLB index	Column	NA	NA	NA	NA	NA	NA	NA	NA	array to be accessed

Notes:
For the instruction cache and data cache, the index/dword/word fields line up with a normal address, except that they are shifted to the left by 8 bits.

Table A-2 defines the array identification value to be used when accessing the various arrays.

Table A-2. Cache Array Identification Values

Bits 7–0 (MSB to LSB)	Array to be Accessed
00h	Data Cache Array
E1h	Data Cache Linear Tag/Status Array
ECh	Data Cache Physical Tag Array
E4h	Instruction Cache Store Array
E5h	Instruction Cache Linear Tag Array
EDh	Instruction Cache Physical Tag Array
E6h	Instruction Cache Valid Bit Array
E7h	Instruction Cache Branch Prediction Array
E8h	Translation Lookaside Buffer 4-Kbyte Page Frame/Status Array

Notes:
Although EDX is normally cleared on RDMSR, it remains intact during array accesses.

Table A-2. Cache Array Identification Values (continued)

Bits 7–0 (MSB to LSB)	Array to be Accessed
E9h	Translation Lookaside Buffer 4-Kbyte Linear Tag Array
Eah	Translation Lookaside Buffer 4–MByte Page Frame/Status Array
Ebh	Translation Lookaside Buffer 4–MByte Virtual Tag Array

Notes:
Although EDX is normally cleared on RDMSR, it remains intact during array accesses.

A.2 AMD-K5 Model 0 Array Data Formats

Table A-3. AMD-K5 Model 0 ICACHE Physical Tags

Bits 31–21	Bit 20	Bits 19–0
0	Valid Bit	Tag (Physical Address 31–12)

Table A-4. AMD-K5 Model 0 DCACHE Physical Tags

Bits 31–23	Bits 22–21	Bits 20–0
0	MESI (00=invalid, 01=shared, 10=modified, 11=exclusive)	Tag (Physical Address 31–11)

Table A-5. AMD-K5 Model 0 DCACHE Data

Bits 31–0
Data

Table A-6. AMD-K5 Model 0 DCACHE Linear Tag

Bit 27	Bit 26	Bit 25	Bit 24	Bit 23	Bit 22	Bit 21	Bits 20–0
PCD	PWT	Dirty Bit	User/Supervisor Bit	R/W Bit	0	Linear Valid Bit	Tag

Table A-7. AMD-K5 Model 0 ICACHE Instructions

Bit 25	Bit 24	Bit 23	Bit 22–21	Bit 20–13	Bit 12	Bit 11	Bit 10	Bit 9–8	Bit 7–0
prefix 1				byte 1	prefix 0				byte 0
start bit	end bit	opcode bit	map (rops/mrom)	byte 1	start bit	end bit	opcode bit	map (rops/mrom)	byte 0

Table A-8. AMD-K5 Model 0 ICACHE Linear Tag

Bits 19–0
Linear Address 31–12

Table A-9. AMD-K5 Model 0 ICACHE Valid Bits

Bits 31–19	Bit 18	Bit 17	Bit 16	Bits 15–0
0	D	linear tag valid bit	user/supervisor	byte-valid bits

Table A-10. AMD-K5 Model 0 ICACHE Branch Prediction

Bits 31–19	Bit 18	Bits 17–14	Bits 13–12	Bits 11–4	Bits 3–0
0	predicted taken	byte offset within block of last byte of predicted branch instruction	column of predicted target	index of predicted target	target byte

Table A-11. AMD-K5 Model 0 TLB 4-Kbyte Linear Tag

Bits 31–20	Bit 19	Bit 18	Bit 17	Bit 16	Bit 15	Bits 14–0
0	global valid bit	dirty bit	user/supervisor bit	read/write bit	valid bit	tag (linear address 31–17)

Table A-12. AMD-K5 Model 0 TLB 4-Kbyte Physical Page Frame

Bits 31–22	Bit 21	Bit 20	Bit 19–0
0	PCD bit	PWT bit	Page frame address (physical address 31–12)

Table A-13. AMD-K5 Model 0 TLB 4-Mbyte Virtual Tag

Bits 31–15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9–0
0	Global valid bit	dirty bit	user/supervisor	read/write bit	valid bit	tag (linear address 31–22)

Table A-14. AMD-K5 Model 0 TLB 4-Mbyte Physical Page Frame

Bits 31–12	Bit 11	Bit 10	Bits 9–0
0	PCD bit	PWT bit	Page frame address (physical address 31–22)

A.3 AMD-K5 Model 1 Array Data Formats

Table A-15. AMD-K5 Model 1 ICACHE Physical Tags

Bits 31–21	Bit 20	Bits 19–0
0	Valid Bit	Tag (Physical Address 31–12)

Table A-16. AMD-K5 Model 1 DCACHE Physical Tags

Bits 31–23	Bits 22–21	Bits 20–0
0	MESI (00=invalid, 01=shared, 10=modified, 11=exclusive)	Tag (Physical Address 31–11)

Table A-17. AMD-K5 Model 1 DCACHE Data

Bits 31–0
Data

Table A-18. AMD-K5 Model 1 DCACHE Linear Tag

Bit 28	Bit 27	Bit 26	Bit 25	Bit 24	Bit 23	Bit 22	Bit 21	Bits 20–0
WB	PCD	PWT	Dirty Bit	User/Supervisor Bit	R/W Bit	0	Linear Valid Bit	Tag

Table A-19. AMD-K5 Model 1 ICACHE Instructions

Bit 25	Bit 24	Bit 23	Bit 22–21	Bit 20–13	Bit 12	Bit 11	Bit 10	Bit 9–8	Bit 7–0
prefix 1				byte (n + 8)	prefix 0				byte (n)
start bit	end bit	opcode bit	map (rops/mrom)	byte (n + 8)	start bit	end bit	opcode bit	map (rops/mrom)	byte (n)

Table A-20. AMD-K5 Model 1 ICACHE Linear Tag

Bit 22	Bit 21	Bit 20	Bits 19–0
D	Linear Valid Bit	User/Supervisor Bit	Linear Address 31–12

Table A-21. AMD-K5 Model 1 ICACHE Valid Bits

Bits 31–0
byte-valid bits

Table A-22. AMD-K5 Model 1 ICACHE Branch Prediction

Bits 31–19	Bit 18	Bits 17–14	Bits 13–12	Bits 11–4	Bits 3–0
0	predicted taken	byte offset within block of last byte of predicted branch instruction	column of predicted target	index of predicted target	target byte

Table A-23. AMD-K5 Model 1 TLB 4-Kbyte Linear Tag

Bits 31–20	Bit 19	Bit 18	Bit 17	Bit 16	Bit 15	Bits 14–0
0	global valid bit	dirty bit	user/supervisor bit	read/write bit	valid bit	tag (linear address 31–17)

Table A-24. AMD-K5 Model 1 TLB 4-Kbyte Physical Page Frame

Bits 31–22	Bit 21	Bit 20	Bit 19–0
0	PCD bit	PWT bit	Page frame address (physical address 31–12)

Table A-25. AMD-K5 Model 1 TLB 4-Mbyte Virtual Tag

Bits 31–15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9–0
0	Global valid bit	dirty bit	user/supervisor	read/write bit	valid bit	tag (linear address 31–22)

Table A-26. AMD-K5 Model 1 TLB 4-Mbyte Physical Page Frame

Bits 31–12	Bit 11	Bit 10	Bits 9–0
0	PCD bit	PWT bit	Page frame address (physical address 31–22)