AMD5$_K$86™ Processor

Technical Reference Manual

AMD

Contents

5 Bus Interface 5-1

Appendix A Compatibility With the Pentium and 486 Processors A-1

List of Figures

List of Tables

Preface

This manual describes the technical features of the AMD5$_K$86™ processor, and its differences from the Pentium processor, at a level of detail suitable for a hardware designer or system-software developer to implement system boards, core system logic, and system software. Specifically, the manual describes the following aspects of the processor

- Internal architecture
- Software differences from the 486 and Pentium processors
- Performance parameters
- Bus signals functions
- Bus cycle timing
- Design issues for system-board designs
- Test and debugging features

A full description of the x86 programming environment is beyond the scope of this manual. Instead, the software sections describe differences from the 486 processor's programming environment. A list of commercial books that describe the x86 programming environment and other subjects of potential interest appears at the end of this preface.

In addition to descriptions of the AMD5$_K$86 processor's unique internal architecture, the manual incorporates details about the behavior of bus signals and bus cycles that are standard to the x86 processors but that are not fully documented in other x86 manuals.

Notation

The following notation is used in this manual:

b—Binary

d—Decimal

h—Hexadecimal

Set—Written with a value of 1

Clear—Written with a value of 0

GP (0)—General-protection exception (13 decimal) with an error value of 0

EFLAGS.IF—The IF bit in the EFLAGS register

CS:EIP—A logical address, expressed as a segment selector (CS) and offset (EIP)

000F_FFF0h—A physical-memory address using hexadecimal notation

Terminology

The following definitions apply throughout this document:

- *Pin and Signal*—A *pin* is a piece of metal on the processor's package. A *signal* is the information about logical states that a pin carries. Pins have pin numbers; signals have signal names. On processors that multiplex signals, pins can carry more than one signal; the AMD5$_K$86 processor, however, does not multiplex signals in this manner.
- *Assert and Negate*—A signal that is driven or sampled active is *asserted*. A signal that is inactive is *negated*. In general, *asserted* means sampled asserted either by the processor or target logic. Signals that are active in a Low-voltage state, such as $\overline{\text{BRDY}}$, are shown with an overbar. Signals that are active in a High-voltage state, such as INTR, are shown without an overbar. Dual-state signals, such as R/$\overline{\text{S}}$ and WB/$\overline{\text{WT}}$, have two states of assertion and, therefore, the term *asserted* has no meaning; such dual-state signals are *driven* High or Low.
- *Drive and Sample*—A single-state signal is *driven* when it is asserted or negated by a logic device; it is *sampled* when its driven state is detected by another device.
- *Cycle and Clock*—This term commonly refers to at least four different things:
  - *Bus-clock period:* The cycle time of the CLK signal.
  - *Processor-clock period:* The cycle time of the processor's internal clock, which has a frequency relative to CLK that is determined by the state of the BF signal during RESET. Whenever this cycle is meant, such as in the Chapter 4 description of pipeline timing and the instruction latency, the full name, *processor-clock cycle*, is used.
  - *Bus cycle:* A signal protocol on the processor's bus, such as a single-transfer read cycle or a special bus cycle.
  - *Sequence of bus cycles:* One or more contiguous bus cycles. For example, the two bus cycles that constitute an interrupt acknowledgment are called a bus *operation*, so that the constituent bus cycles can be distinguished from the entire operation.

- *Writeback*—This term refers to two related concepts:
  - *Bus Cycle*—A 32-byte burst write cycle to a memory block that has been cached in the *modified* state. Writebacks can be caused by inquire cycles, internal snoops, writeback and invalidate operations (such as $\overline{\text{FLUSH}}$ or the WBINVD instruction), cache-line replacements, or locked operations on cached locations. It is sometimes called a *copyback*.
  - *Cache-Line State*—A cache line in the *modified* or *exclusive* MESI state (modified, exclusive, shared, invalid).
- *Writethrough*—This term refers to two related concepts:
  - *Bus Cycle*—A 1-to-8-byte, single-transfer write cycle caused by write misses or write hits to lines in the *shared* or *exclusive* MESI state.
  - *Cache-Line State*—A cache line in the *shared* MESI state.
- *Flush*—This term commonly refers to at least four things and is usually avoided in favor of the following specific terms:
  - *Pipeline Invalidation:* A pipeline-flush operation invalidates instructions in the pipeline that have not been retired (and, depending on the type of pipeline invalidation, entries in the reorder buffer, entries in the TLB, and/or branch-prediction bits) without writing their state to any storage resource.
  - *Cache Invalidation:* The INVD instruction invalidates the contents of the instruction and data caches, without writing modified data back to memory.
  - *Cache Writeback and Invalidation:* The WBINVD instruction writes *modified* lines in the data cache back to memory while invalidating each line in the instruction and data caches.
  - *FLUSH Operation:* The $\overline{\text{FLUSH}}$ input signal executes the same microcode routine as the WBINVD instruction to write *modified* lines in the data cache back to memory while invalidating each line in the instruction and data caches.
- *Flush Acknowledge Cycle*—This term commonly refers to different types of special bus cycles driven by the processor, and is therefore avoided in favor of the following specific terms:
  - *FLUSH Acknowledge:* A special bus cycle driven after the $\overline{\text{FLUSH}}$ operation completes.
  - *INVD Acknowledge:* A special bus cycle driven after the INVD cache invalidation completes.
  - *WBINVD Acknowledge:* A sequence of two special bus cycles driven after the WBINVD cache writeback and invalidation completes.
- *Snoop*—This term commonly refers to at least three different actions and is therefore avoided in favor of the following specific terms:
  - *Inquire Cycles:* These are bus cycles driven by system logic. They cause the processor to compare the inquire-cycle address with the processor's physical

cache tags. The AMD5$_K$86 and Pentium processors both support inquire cycles.

- *Internal Snooping:* These snoops are initiated by the processor (rather than system logic) during certain types of cache accesses. Both the AMD5$_K$86 and Pentium microprocessors support this type of internal snooping for the purpose of detecting self-modifying code. See page 2-22 for details.
- *Bus Watch:* Some caching devices watch their address and data bus continuously while they are held off the bus. They compare every address driven by another bus master with their internal cache tags, and they may also be able to update their cached lines during writebacks to memory by another bus master. Neither the AMD5$_K$86 nor Pentium microprocessors support bus watching.

- *Cold and Warm Reset*—The terms *cold* or *hard* reset and *warm* or *soft* reset are commonly used to mean three related but different things, and the terms are therefore avoided. A *cold* or *hard* reset typically refers to the assertion of RESET at power-up, but *warm* or *soft* reset can refer either to the assertion of RESET after power-up or to the assertion of INIT.
- *System Logic*—Any logic outside the processor, including a core-logic chipset, another bus master, or separate controllers for L2 cache, memory, interrupts, DMA, communications, video, bus bridging, bus arbitration, or any other system function.

References

Abel, Peter. *IBM PC Assembly Language and Programming*. Englewood Cliffs: Prentice Hall, 1995.

Abramovici, Miron; Melvin A. Breuer; and Arthur D. Friedman. *Digital Systems Testing and Testable Design*. New York: IEEE Press, 1990.

Agarwal, Rakesh. *80x86 Architecture & Programming*. Vols. I and II. Englewood Cliffs: Prentice-Hall, 1991.

Alexandridis, Nikitas. *Design of Microprocessor-Based Systems.* Englewood Cliffs: Prentice-Hall, 1993.

Anderson, Don, and Tom Shanley. *Pentium Processor System Architecture*. Reading: Addison-Wesley, 1995.

Barkakati, Nabajyoti, and Randall Hyde. *Microsoft Macro Assembler Bible*. Carmel: Sams, 1992.

Brey, Barry B. *The Intel 32-Bit Microprocessors*. Englewood Cliffs: Prentice Hall, 1995.

Brown, Ralf, and Jim Kyle. *PC Interrupts, A Programmer's Reference to BIOS, DOS, and Third-Party Calls*. Reading: Addison-Wesley, 1994. For an updated version on the Internet, ftp to `oak.oakland.edu` and get file `/pub/msdos/info/inter41.zip`.

Brumm, Penn, and Don Brumm. *80386/80486 Assembly Language Programming*. Windcrest: McGraw-Hill, 1993.

Chappell, Geoff. *DOS Internals*. Reading: Addison-Wesley, 1994.

Crawford, John H., and Patrick P. Gelsinger. *Programming the 80386*. San Francisco: Sybex, 1987.

Giles, William B. *Assembly Language Programming for the Intel 80xxx Family*. New York: Macmillan, 1991.

Handy, Jim. *The Cache Memory Book*. San Diego: Academic Press, 1993.

Hennessy, John L., and David A. Patterson. *Computer Architecture, A Quantitative Approach*. 2d Ed. San Francisco: Morgan Kaufmann Publishers, 1996.

Hogan, Thom. *The Programmer's PC Sourcebook*. Redmond: Microsoft Press, 1991.

Hwang, Kai. *Advanced Computer Architecture*. New York: McGraw-Hill, 1993.

Institute of Electrical and Electronics Engineers. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Std 754-1985.

Institute of Electrical and Electronics Engineers. *IEEE Standard for Radix-Independent Floating-Point Arithmetic*. ANSI/IEEE Std 854-1987.

Institute of Electrical and Electronics Engineers. *IEEE Standard Glossary of Mathematics of Computing Terminology*. ANSI/IEEE Std 1084-1986. Out of print.

Johnson, Mike. *Superscalar Microprocessor Design*. Englewood Cliffs: Prentice-Hall, 1991.

Katz, Randy H. *Contemporary Logic Design*. Redwood City: Benjamin Cummings, 1994.

Morse, Stephen P.; Eric J. Isaacson; and Douglas J. Albert. *The 80386/387 Architecture*. New York: John Wiley & Sons, 1987.

Norton, Peter; Peter Aitken; and Richard Wilton. *PC Programmer's Bible*. Redmond: Microsoft Press, 1993.

Parker, Kenneth P. *The Boundary-Scan Handbook*. Boston: Kluiver, 1992.

Patterson, David A., and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. San Francisco: Morgan Kaufmann Publishers, 1994.

Phoenix Technical Reference Series. *System BIOS for IBM PCs, Compatibles, and EISA Computers*. Reading: Addison-Wesley, 1991.

Pietrek, Matt. *Windows Internals.* Reading: Addison Wesley, 1993.

Richter, Jeffrey. *Advanced Windows NT.* Redmond: Microsoft Press, 1994.

Ro, Sen-Cuo, and Sheau-Chuen Her. *i386/i486 Advanced Programming.* New York: Van Nostrand Reinhold, 1993.

Slater, Michael. *Microprocessor-Based Design.* Englewood Cliffs: Prentice-Hall, 1989.

Stallings, William. *Operating Systems.* New York: Macmillan, 1992.

Van Gilluwe, Frank. *The Undocumented PC.* Reading: Addison-Wesley, 1994.

Wakerly, John F. *Digital Design Principles and Practices.* Englewood Cliffs: Prentice-Hall, 1994.

Wharton, John. *The Complete x86.* Sebastopol, CA: MicroDesign Resources, 1994.

1

Overview

The AMD5$_K$86™ processor brings superscalar RISC performance to desktop systems running industry-standard x86 software. The processor implements advanced design techniques like instruction pre-decoding, single-cycle internal RISC operations, parallel execution units, out-of-order issue and completion, register renaming, data forwarding, and dynamic branch prediction. The processor's many test and debug features support fast, reliable designs for x86 desktop systems.

AMD's development and support of the popular Am386® and Am486® processors has given it a broad foundation of experience in the x86 architecture. The AMD5$_K$86 processor's binary compatibility with DOS and Windows®-compatible software running on the Pentium processor and all previous x86 processors has been established in extensive testing, using industry-standard test tools. Compatibility and qualification testing has also been provided by leading desktop-system manufacturers, chip-set manufacturers, and the independent XXCAL testing laboratory.

The result can be seen in the AMD5$_K$86 processor's performance. This performance plus its compatibility with an immense library of existing x86 software make the AMD5$_K$86 processor a leading-edge solution for desktop systems.

1.1 Features

- Pentium-Processor Standard
  - Compatible with the Pentium (735\90, 815\100) processor 296-pin socket
  - Compatible with existing Pentium (735\90, 815\100) processor support infrastructure and system designs
  - Compatible with Pentium, 486, and 386 processor software
  - Compatible with x86 DOS, Microsoft® Windows® operating system, and the large installed base of x86 software
  - Compatible with IEEE 854 floating-point standard
  - Selectable bus frequencies
  - Support for multiprocessing
- High-Performance Execution
  - Six execution units (two ALUs, two load/store, one branch, one floating-point)
  - Up to four instructions issued per processor clock
  - Out-of-order issue and completion
  - Speculative execution along three predicted branches
  - Register renaming
  - Data forwarding
  - Predecoder converts x86 instructions to single-cycle RISC operations (ROPs)
  - Fast integer multiply (4-cycle, fully pipelined)
  - Five-stage pipeline
  - Single-cycle cache access
  - Zero-delay branching, 3-clock misprediction penalty (often hidden)
  - No mixed-operand-size penalty
  - No prefix penalty
  - Single-cycle misalignment penalty
  - No instruction-pairing requirements for parallel issue
  - No pipeline invalidation on segment loads
  - Efficient support for 16- and 32-bit code, with mixed operand sizes

- High-Performance Cache and TLBs
  - 16-Kbyte instruction cache supports split-line access
  - 8-Kbyte, dual-ported data cache with MESI cache coherency protocol
  - Dual-tagged (both linear and physical tags)
  - Inquire cycles run in parallel with program cache access
  - 4-Kbyte TLB (128 entries) and 4-Mbyte TLB (4 entries)
- Extended Features
  - Control Register 4 (CR4)
  - CMPXCHG8B instruction
  - CPUID instruction
  - Time stamp counter (TSC)
  - Machine-Specific Registers (MSRs)
  - 4-Mbyte page size
  - Global pages held in TLB during flushes
- Low Power
  - Static, 3.3-V design
  - System Management Mode (SMM) with I/O trapping
  - Low-power halt and stop-clock states
  - Compatible with U.S. Department of Energy's *Energy Star* program
  - Compatible with Microsoft Advanced Power Management specification
- Extensive Test and Debug Features
  - Two built-in self-test (BIST) modes
  - Output-Float Test mode
  - Cache and TLB testing (tags and data)
  - Debug registers, with I/O breakpoint extension
  - Branch tracing
  - Functional-redundancy checking
  - IEEE 1149.1-1990 Test Access Port (TAP) and JTAG boundary-scan testing
  - Hardware Debug Tool (HDT)

2

Internal Architecture

The RISC design techniques used in the processor's internal architecture account, in large part, for its high performance. The following sections summarize the processor's execution pipeline behavior, the hardware aspects of the internal instruction cache and data cache, and the hardware aspects of memory management.

Figure 2-1 shows the major logic blocks that make up the internal architecture. The blocks are organized in the figure by stages of the processor's execution pipeline, which are listed vertically on the right side of the figure. The blocks are explained throughout the section that follows.

In this chapter, the terms *clock* and *cycle* refer to processor-clock cycles. If bus-clock cycles or bus cycles are discussed, they are explicitly named. Processor-clock cycles occur at a multiple of bus-clock (CLK) cycles, as determined by the BF input signal and processor model number.

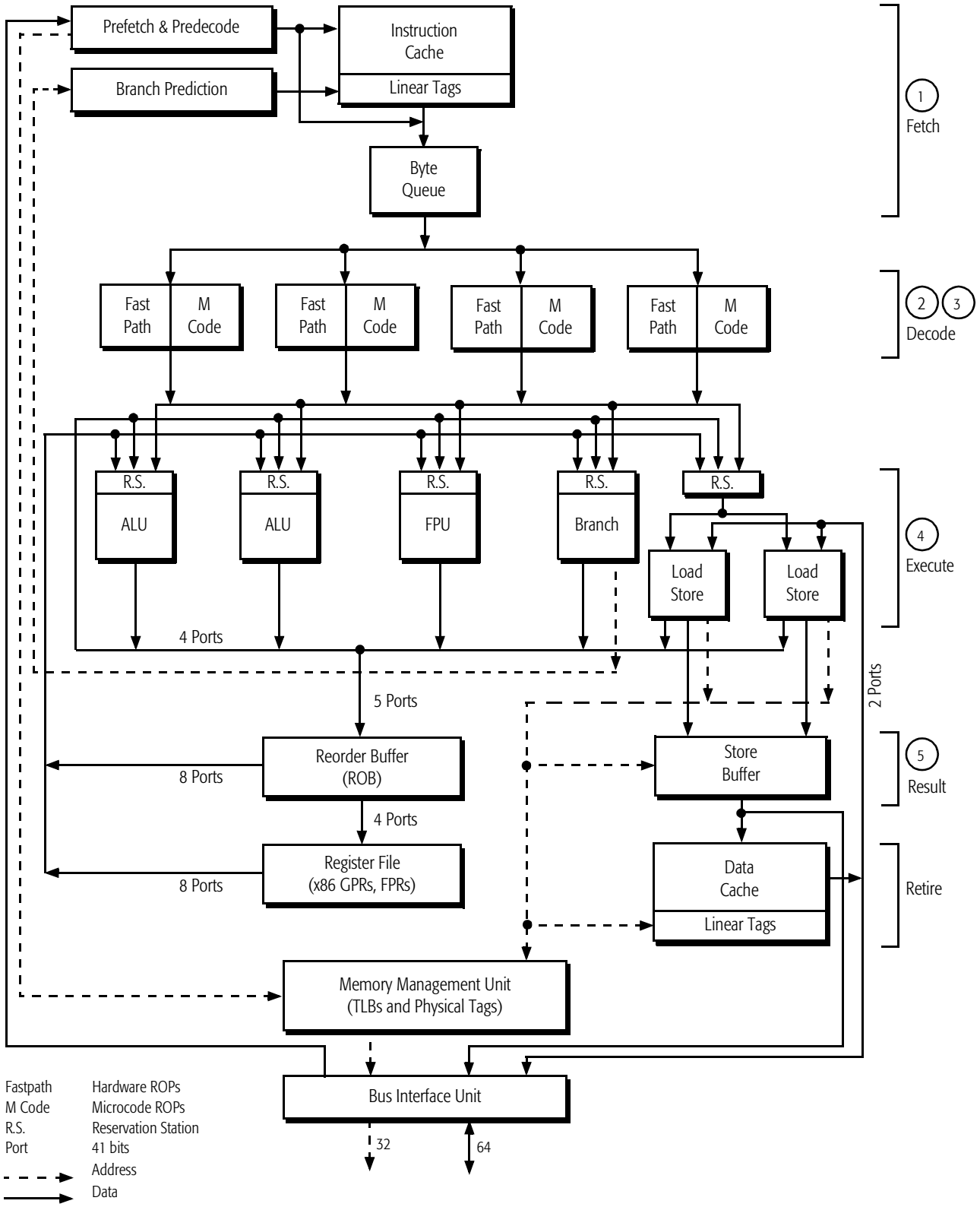


FIGURE 2-1. Internal Architecture, with Pipeline Stage

2.1 Prefetch and Predecode

Figure 2-1 (top-left corner) shows the processor's prefetch and predecode logic being fed with data from the external bus via the memory management unit. Prefetching attempts to keep the instruction cache and prefetch cache filled ahead of the execution pipeline's fetch requirements. The processor only prefetches during fetch-stage misses in the instruction cache, which typically occur during taken branches.

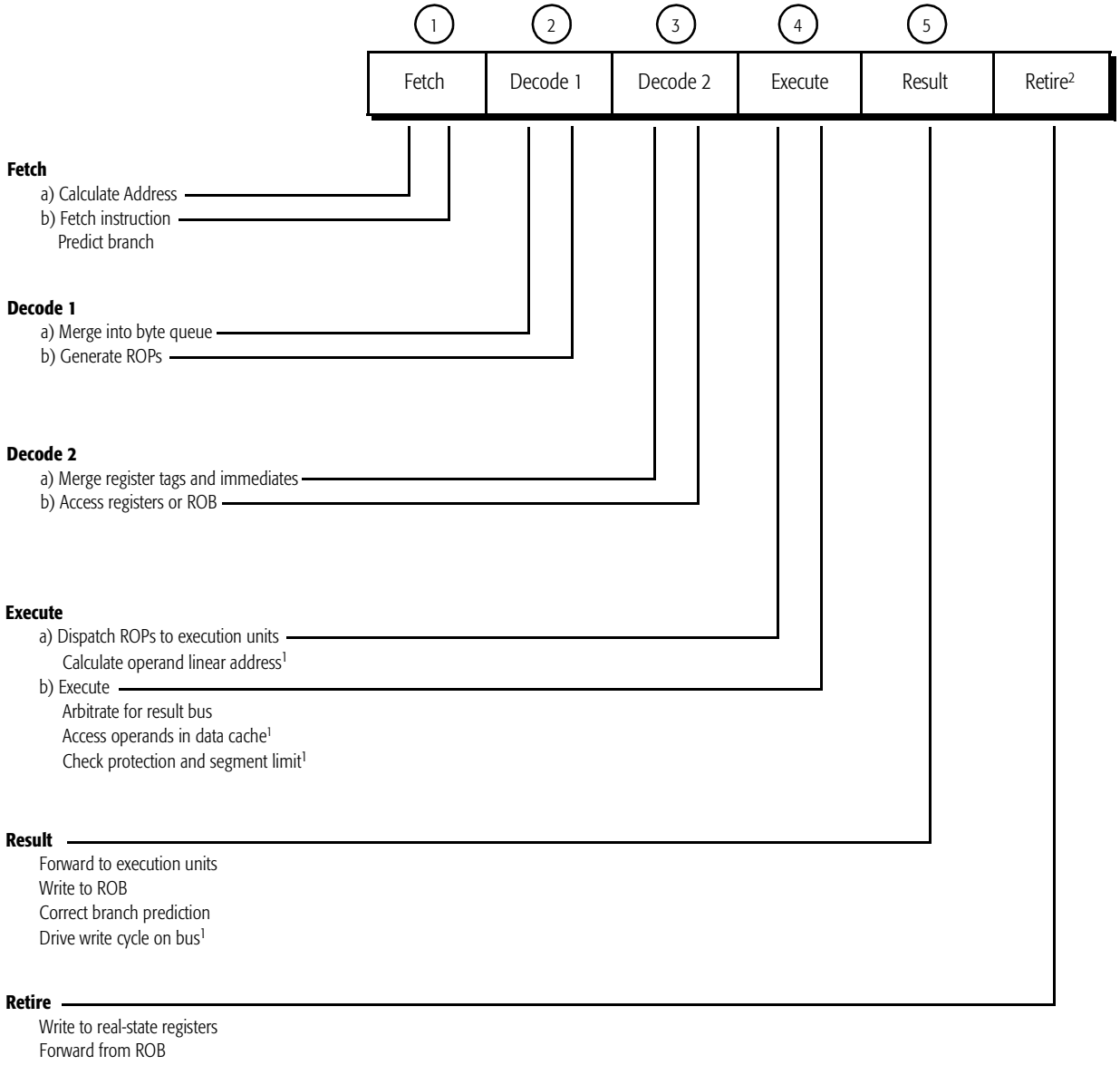
When a miss occurs, the prefetcher initiates a 32-byte burst memory read cycle on the bus to fill a *prefetch cache*. For cacheable accesses, the prefetch cache also fills 32-byte lines in the instruction cache. For non-cacheable accesses, the prefetch cache provides instructions directly to the execution pipeline.

The instruction cache contains a copy of certain fields in the current code-segment descriptor. During a taken branch, the fetch logic adds the code-segment base to the effective address and places the resulting linear address in the prefetch program counter, which then increments as a linear address along a sequential stream. All branches during prefetching are assumed to be not taken.

The processor predecodes its x86-instruction stream in the same clock in which x86 instructions come out of the prefetch cache. An x86 instruction can be from 1 to 15 bytes long. Predecoding annotates each instruction byte with information that later enables the decode stage of the pipeline to perform more efficiently. The predecode information identifies whether the byte is the start and/or end of an x86 instruction, whether it is an opcode byte, and the number of internal RISC operations (ROPs) it will require at the decode stage. The predecode information is stored in the instruction cache with each x86 instruction byte. It is passed during instruction fetching to the decode stage, where it allows multiple x86 instructions to be decoded in parallel. This avoids delaying the decode of one instruction until the decode of the prior instruction has determined its ending byte.

2.2 Execution Pipeline

Figure 2-1 shows the relation between the internal logic and the stages of the execution pipeline. Figure 2-2 shows the functions of the pipeline stages. The first five stages—Fetch, Decode 1, Decode 2, Execute, and Result—affect throughput performance. The sixth stage, Retire, may occur at a variable number of clocks after the Result stage, but the Retire stage does not affect throughput performance when the processor operates in a non-serialized mode, which is typical of most processing. Thus, the pipeline effectively has five stages. Because the pipeline is moderately shallow, penalties associated with mispredicting a branch (three clocks) or clearing the pipeline (variable clocks) are relatively small compared with processors that have deeper pipelines (more pipeline stages).



Notes:

1. *Load/store instructions only.*
2. *The Retire stage may occur one or more clocks after completion, but it does not affect throughput.*

FIGURE 2-2. Pipeline Stage Functions

2.2.1 Fetch

The processor can fetch up to 16 bytes per clock out of the instruction cache. Fetching begins with the calculation of the linear address for the next instruction along a predicted branch of the x86 instruction stream. The address accesses the instruction cache or, during a miss, the prefetch cache. Fetching can occur along a single execution stream with up to three taken branches. Fetches that miss both the instruction cache and prefetch cache are driven to the prefetcher.

In addition to fetching instructions, the fetch logic handles branch predictions and detects conditions requiring pipeline invalidation and restarting, such as context switches or branches into cache lines that do not contain the correct predecode state. Branches are dynamically predicted on a cache-line basis using a 1-bit algorithm. Each of the 1024 instruction-cache lines has a tag that predicts the last byte in the cache line to be executed, whether or not the branch will be taken, and the cache index of the branch target (called the *successor index)*. When the caches are invalidated, all branch predictions are cleared.

During prefetch all branch instructions are predicted as not-taken. Later, if the execution of a branch instruction reveals a misprediction, the fetch unit backs out of the branch by invalidating all speculative states in the prefetch cache, reorder buffer, load/store reservation station, and store buffer. Then, for cacheable instructions, the branch prediction stored in the instruction cache is updated while the correct branch target is fetched. Prediction updates are disabled when the branch instruction is non-cacheable, because no prediction information is saved for non-cacheable instructions.

In typical x86 desktop programs, a branch occurs about once every seven x86 instructions. Without branch prediction, branch targets remain unresolved until the execution phase, which creates pipeline delays. The processor's branch-prediction mechanism accurately predicts 70% to 85% of branches (depending on program behavior) and has a misprediction penalty of only three processor clocks.

2.2.2 Decode

The two-stage decode logic accepts predicted x86 instruction bytes and their predecode bits from the fetch logic, shifts them into a 16-byte FIFO buffer called the *byte queue*, merges register tags and operands, and generates internal RISC operations (ROPs). The decode logic also generates microcode entry points for complex instructions, interrupts and exceptions, and several other functions, and it manages the floating-point stack.

ROPs are fixed-format internal instructions with up to three operands. Most ROPs execute in a single clock. The operands (up to two source and one destination) can be 1-, 2-, or 4-bytes wide, or half of an 8- or 10-byte floating-point operand. ROPs can be combined to perform every function of an x86 instruction. One x86 instruction can be decoded into as few as one ROP (for example, a register-to-register add), or it can be decoded into several ROPs, depending on its complexity.

The processor uses a combination of hardware and microcode to convert x86 instructions into ROPs. The hardware consists of four parallel *fastpath* converters that translate the most commonly used x86 instructions (moves, shifts, branches, ALUs) into one, two, or three ROPs. Translations requiring more than three ROPs (complex instructions, serializing conditions, interrupts and exceptions, etc.) are handled by microcode. Microcode generates the same types of ROPs as the fastpath hardware but in streams longer than three. The predecode information stored with each x86 instruction byte specifies the number of ROPs that instruction requires, or it specifies that microcode is required. The decoder provides the entry point into microcode for complex operations.

Pipeline *serialization* (or *synchronization)* is handled at the decode stage. When the processor decodes a serializing instruction, it stops decoding at that instruction, waits for all previously decoded instructions to retire (described in Section 2.2.5 on page 2-12), then decodes and executes through retirement the serializing instruction before decoding any additional instructions. Thus, the serializing instruction is guaranteed to execute in program order.

The serializing instructions include OUTx, invalidations (INVD, WBINVD, INVLPG), interrupt returns (IRET, IRETD, RSM), descriptor-table-register and task-register loads (LGDT, LLDT, LIDT, LTR), moves to control or debug registers (MOV to CRx or DRx), model-specific register instructions (RDMSR, WRMSR), and CPUID. Special bus cycles and interrupt-acknowledge operations also serialize the pipeline. INx instructions are not executed until the store buffer and write-back buffers are drained of any pending writes.

The four converters that generate fastpath or microcode ROPs *dispatch* up to four ROPs in parallel per clock to the execution unit reservation stations.

2.2.3 Execute

The processor has the following execution units that work in parallel with one another:

- Two ALUs (integer, logic, and shift operations)
- One floating-point unit
- Two load/store units
- One branch unit

Each execution unit has its own FIFO reservation station with two or four entries. ROPs are dispatched to reservation stations in program order. One ROP can be dispatched to a single reservation station in a given clock, thus up to four reservation stations receive an ROP each clock. ROPs are *issued* from a reservation station to its execution unit when all operands are available from the register file, reorder buffer, or prior execution via forwarding (including from data cache loads), and when the execution unit has completed its prior ROP. Issue and dispatch occur in the same clock if the operands are available and the unit is free at dispatch time.

While ROPs are issued in order to a particular execution unit, ROPs go out of order at the point of issue because reservation stations issue ROPs at different times relative to each other. The use of reservation stations and out-of-order execution reduces instruction stalls due to dependencies on execution resources and allows a higher issue rate to be maintained. Multiple values for the same register are resolved by providing tags for each register value (register renaming). True data

dependencies are resolved using forwarding at all execution units. Antidependencies (in which later instructions produce a value that overwrites one used by an earlier instruction) are removed automatically by buffering operands—or tags that point to operands—at reservation stations. Output dependencies (in which later instructions must be seen by software to complete after earlier instructions in order to leave the correct value in a register) are resolved by the reorder buffer.

Reservation stations are supplied with operands over eight 41-bit operand buses. Execution results are sent to the reorder buffer (ROB) over five 41-bit result buses. Tags forwarded to the execution units represent results to watch for on one of the result buses.

No special compiler optimizations are required for high-performance execution on the AMD5$_K$86 processor.

Integer/Shift Units

Two ALUs perform integer, logic, and shift operations. Both ALUs have two-entry reservation stations. Table 2-1 shows the types of ROPs executed by each ALU. Unlike the Pentium processor, the AMD5$_K$86 processor has few restrictions on the pairing of integer instructions needed to use both integer units in parallel.

TABLE 2-1. ALU Instruction Classes

Instruction Class	ALU0	ALU1
Addition	Yes	Yes
Subtraction	Yes	Yes
Logical	Yes	Yes
Compare	Yes	Yes
Packed BCD	Yes	No
Unpacked BCD	Yes	No
Special (ADDC, SUBB)	Yes	No
Shift	No	Yes
Divide	Yes	No

Floating-Point Unit

The IEEE 854-compatible floating-point unit (FPU) can issue pipelined ROPs from its 2-entry reservation station at the rate of one per clock. One ROP can be issued to either the add or multiply pipeline in each clock, even when the operations are separated by an exchange ROP. The add and multiply pipelines use a common pre-detect unit and rounder. The rounder can return one result per clock.

When data is loaded from memory, it is converted to an internal 82-bit extended format before being stored in the stack. The format uses two of the internal 41-bit operand or result buses.

Load/Store Units

Two load/store units read and write data-cache and memory operands. A shared, 4-entry reservation station buffers incoming ROPs, and a shared, 4-entry store buffer accepts outgoing speculative-state operands destined for the data cache or memory. The reservation station is dual-ported and the store buffer is single-ported, so that the processor can perform two loads or one load and one store per clock.

Each unit holds copies of segment-descriptor fields so that it can calculate logical and linear addresses and check protection variables and segment limits. Data loaded by one instruction in a load/store unit can be used by another instruction in another execution unit in the next clock. There is no load-use penalty. The data cache can be accessed in a single clock. These low latencies provide an important performance advantage because a majority of x86 instructions in typical desktop programs involve memory as one of their operands.

The load/store units can service two accesses in parallel (two loads or one load and one store), except a load and store to the same data-cache index and bank, or when one of the accesses is an I/O load, a locked access, a segment-descriptor load, a data breakpoint, or the first half of a misaligned access.

Branch Unit

The branch unit has a 2-entry reservation station and executes correctly predicted branches with zero delay. The unit executes calls, returns, conditional jumps, conditional byte-sets, floating-point exchanges, and microbranches. *Speculative execution* occurs whenever a conditional-branch instruction executes. The branch unit is the only execution unit that decodes condition codes and supports speculative flag input operands.

The branch unit receives branch-prediction information from the decoder. If the branch unit executes a branch differently than predicted, it signals the instruction cache, reorder buffer, and decode logic, and it passes the correct information to the branch-prediction array in the fetch stage.

2.2.4 Result

The processor implements a 16-entry *reorder buffer (ROB)* for speculative-state register renaming, and a 4-entry *store buffer* for speculative-state buffering between the load/store units and the data cache. An ROP is said to *complete* when the result of its execution is written to the ROB or store buffer. Results may be returned out of order. Results written to the ROB are simultaneously forwarded (that is, fed back) to all execution units.

An entry tag is allocated at the top of the ROB for each ROP that is dispatched to a reservation station. Entries for up to four ROPs can be allocated simultaneously. Among other things, the ROB keeps track of the program counter associated with each instruction, resolves ROP-level dependencies, stores speculative results, provides the most recent copy of a register to execution units, recovers from mispredicted branches without altering real state, and provides substitute tags to internal resources when required operands are still outstanding.

The x86 architecture defines only eight general-purpose registers and eight entries in the floating-point stack. This limited set of registers leads to register dependencies and register reuse. The processor overcomes register dependencies by renaming registers in the ROB, and it overcomes register reuse with data forwarding. Data forwarding provides execution results immediately to other instructions without waiting for results to be written to and read back from registers, the data cache, or memory. Multiple speculative-state registers for each real-state register enable different execution units to use the same logical register simultaneously. When the register file detects multiple writes to the same real-state register, only the latest write in program order is performed—all other writes are discarded. Multiple reads of the same real-state register are performed without detection or special handling.

2.2.5 Retire

The processor implements a real-state (non-speculative) *register file* that contains the x86-architecture registers and a real-state 8-Kbyte data cache. While ROPs complete out of order and their results are forwarded to other execution units and to the ROB out of order, their results are always written at retirement time to the real-state x86 registers in program order. Likewise, as results are written from the load/store units to the store buffer out of order, they are always written at retirement time to the data cache and/or memory in program order.

An x86 instruction is said to *retire* when the ROB or store buffer writes the operands for all of its ROPs, in program order, to the x86 real-state registers or the data cache. At the point of retirement, the register file and data cache fully reflect the execution of an instruction. Any associated exceptions are recognized (the ROB facilitates precise exception handling), any external interrupts that were latched or are currently held asserted are recognized, and the instruction pointer is updated. For instructions that store an operand to memory, retirement is the time at which the store is guaranteed to be written externally. When a *pipeline invalidation* (flush) occurs, it does so at the retirement stage, causing all instructions in the pipeline that have not reached the retirement stage to be invalidated.

The retirement stage is also called the *instruction-retirement boundary*, or simply *instruction boundary*. The processor can retire up to four instructions per processor clock. Thus, the next set of up to four instructions that are candidates to retire determines the next instruction boundary at which an external interrupt can be recognized.

Only one store (from the store buffer or from either of the processor's two writeback buffers) can be among the set of up to four instructions that retire simultaneously. Thus, for example, the processor will only finish one in-progress store cycle on the bus before recognizing an asserted HOLD, $\overline{\text{SMI}}$, or $\overline{\text{STPCLK}}$. If the set of retirement candidates in any clock includes more than one store, only those instructions up to (but not including) the second store will retire. The remaining stores occur one at a time, in their queued order, during subsequent retirements.

2.3 Cache Organization and Management

The performance of the execution pipeline is enhanced by the processor's on-chip, 16-Kbyte instruction cache and 8-Kbyte data cache. Both caches are linearly addressed and each has two associated tag directories, one for linear tags and one for physical tags.

Linearly addressed caches avoid linear-to-physical address translation through the TLB and can be faster than physically addressed caches. Cache accesses in the AMD5$_K$86 processor take one clock. The physical tags are only accessed during cache misses and snoops. By comparison, accesses in the Pentium processor's physically tagged caches take one or two clocks, depending on the type of operand being accessed (operands used in address calculations for the next cache access take two clocks). Since most x86 instructions access memory, they benefit greatly by being cached, and the faster cache-access time on the AMD5$_K$86 processor is a performance advantage.

The enabling and operating modes for the caches are software controlled by the CD and NW bits of CR0. When disabled, both caches are *locked*. They are accessed in all operating modes, and the processor can still hit in a cache that has not been invalidated, even if software has turned the caches off. These mechanisms work the same on both the AMD5$_K$86 and Pentium processors.

Any area of memory may be cached. However, the processor prevents caching of locked operations and TLB reads, the operating system can prevent caching of certain pages by setting the PCD bit in page-directory and/or page-table entries, and system logic can prevent caching of certain bus cycles by negating the $\overline{\text{KEN}}$ input signal on the first $\overline{\text{BRDY}}$.

The processor implements a requested-word-first protocol for line fills in both caches. Upon receiving the first 8-byte quadword, execution continues while the remainder of the line is loaded into the cache. Both caches, however, are *blocking*—a read hit or miss after a read miss waits until the prior miss fills the cache. Since read misses are rare, relative to read hits, cache blocking has little effect on overall performance.

The following sections describe the basic architecture and resources of the processor's internal caches. For information about how the system software and hardware control cache configuration and coherency, see Section 6.2 on page 6-8.

2.3.1 Instruction Cache

The instruction cache has the following characteristics:

- 16 Kbytes
- 32-byte line size
- Four-way, set associative
- Dual-tagged (linear and physical)
- Single-clock access
- Supports 16-byte split-line accesses
- Requested-word-first line-fill protocol
- Five predecode bits per instruction byte
- Round-robin replacement policy
- Read-only, invalidate on write hit

Instruction-cache accesses can be to any 16 bytes within a single 32-byte line or they can be split into two 8-byte accesses across two contiguous lines.

Split-line fetches can provide instructions from sequential lines in a single clock. This keeps decode logic supplied with a steady stream of bytes. Instruction fetches can read any 16 bytes of a single line or—in a split-line fetch—the high 8 bytes of the first line and the low 8 bytes of the next sequential line (index + 1 as determined by the A4 address bit), starting on either an odd or even line.

Instruction-cache lines have only two coherency states (valid or invalid) rather than the four MESI *(modified, exclusive, shared, invalid)* coherency states of data-cache lines. Only two states are needed because these lines are only read, never written. In addition to holding instructions, each instruction-cache line holds 5 predecode bits per instruction byte. The information contained in these bits is described in Section 2.1 on page 2-3.

Parts of the current code-segment descriptor are maintained in the instruction cache. This allows the cache to translate logical addresses for branches and other prefetch targets to linear address tags for the incoming cache-line fills.

Details on the instruction-cache storage formats and testing are given in Section 7.4 on page 7-7.

2.3.2 Data Cache

The data cache has the following characteristics:

- 8 Kbytes
- 32-byte line size
- Four-way, set associative
- Four banks
- Dual-tagged (linear and physical)
- Byte-addressable
- Single-clock access
- Two true linear-tag ports—two parallel accesses per clock
- Two logical data ports (one read-only, one read/write)—two parallel accesses per clock, if not to the same bank
- MESI cache-coherency protocol (maintained by physical tags)
- Requested-word-first line-fill protocol
- Pseudo-random replacement policy
- Read/write (writeback or writethrough modes)

The data cache overcomes load/store bottlenecks by supporting simultaneous accesses to two lines in a single clock, if the lines are in separate banks. Each of the four cache banks contains eight bytes, or one-fourth of a 32-byte cache line. They are interleaved on a four-byte boundary. One instruction can be accessing bank 0 (bytes 0–3 and 16–19), while another instruction is accessing bank 1, 2, or 3 (bytes 4–7 and 20–23, 8–11 and 24–27, and 12–15 and 28–31 respectively).

Entries in the data cache are real-state operands. A *load* occurs when one of the load/store units reads an operand from the data cache or memory. A *store* occurs at the retirement pipeline stage when an entry from the speculative-state store

buffer, which resides between the load/store units and the data cache, moves to the real-state data cache or memory.

Details on the data-cache storage formats and testing are given in Section 7.4 on page 7-7.

2.3.3 Cache Tags

The processor's caches are *dual-tagged*. That is, the processor maintains two sets of tags—linear and physical—for each line in the two caches. The linear tags are stored in the instruction and data caches. The physical tags are stored in the memory management unit (MMU), where the TLB is also located. The physical-tag directories for each cache have one port.

Linear tags are read for all accesses to the instruction and data caches. All read misses, memory writes, and snooping—both external inquire cycles and automatic internal snooping—go through the physical tags. The MESI cache-coherency state is recorded in the physical tags.

Accesses to the data-cache physical tags add two clocks to the one-clock linear-tag access. Accesses to the instruction-cache physical tags add three clocks to the one-clock linear-tag access. Thus, physical-tag accesses take a total of three clocks for the data cache or four clocks for the instruction cache, but they occur infrequently. For write hits to the data cache, however, the additional latency for accessing the physical tags (needed to determine the MESI state) is transparent to program execution because write hits are pipelined and can occur at a sustained rate of one per clock.

There is a corresponding physical tag for each linear tag. Two or more linear addresses can be aliased to a single physical address. When the processor detects an aliased access to the store buffer, the TLB and physical tags forward the access directly from the store buffer without depending on a linear-tag match in the data cache.

The linear tags for both caches are invalidated whenever paging is turned on or off, or when CR3 (the page-directory base register) is loaded, except that during x86-architecture task switches, the linear tags are only invalidated if the current and new value for CR3 are different. When linear tags are invali-

dated, many or all of the cached lines may still be valid, but accesses miss in the linear tags and go through the MMU to the physical tags. If an access misses the linear tags but hits in the physical tags, the processor restores the linear tag using the linear address for the access. This is called a *cache-tag recovery.* The revalidation of the linear tag does not add any additional time to that of the physical-tag access itself.

The linear tags for both caches are invalidated during physical-tag invalidation, or when the RESET or INIT input signal is asserted. The linear *and* physical tags for both caches are invalidated when the $\overline{\text{FLUSH}}$ input signal is asserted or when the INVD or WBINVD instruction is executed.

2.3.4 Cache-Line Fills

Memory reads that miss in the instruction or data cache generate read-allocate operations. These begin with an attempt to find an invalid line in one of the four cache ways for the accessed index. If an invalid line cannot be found in one of the four ways for the index, a line is pseudo-randomly selected for replacement from one of the four ways. Then the processor fills the line by driving a four-transfer burst cycle on the bus, aligned on 32-byte boundaries, with the target quadword (qword) delivered first.

Instruction-cache line fills initiate four 8-byte transfers from memory (one burst cycle) on the bus. All 32 bytes go through the prefetch cache (which has two 32-byte lines) to the instruction cache and byte queue, with x86 instruction predecoding performed on the fly.

Data-cache line fills also initiate four 8-byte transfers on the bus. If a *shared* or *exclusive* line is being replaced prior to the line fill, the first two 8-byte qwords fill half of the cache line, while the accessed data item is simultaneously forwarded through the load/store unit to the ROB and execution units. Then the remaining two qwords arrive and fill the other half of the cache line. When the cache line is completely filled, the state of the line is updated. If the line being filled is replacing a *modified* line, the prior contents of the line are copied to a 32-byte writeback (copyback) buffer in the bus interface unit while the new line is being read.

2.3.5 Cache Coherency

The processor's cache-coherency mechanism is based on real (non-speculative) state. Everything that accesses main memory has the same view of that memory, which is never modified speculatively. The contents of the processor's data cache are always real-state. Furthermore, on the AMD5$_K$86 processor, writes to both memory and the data cache are always done in program order, irrespective of the state of the $\overline{\text{EWBE}}$ input signal.

The processor's data cache implements coherency with the MESI *(Modified, Exclusive*, *Shared, Invalid)* protocol. The instruction cache, which is read-only, has no write-related states. The instruction cache implements coherency with only a valid bit, which in effect works like a *shared-invalid* subset of the MESI protocol. The coherency state bits are stored in the physical tags for each cache. The physical tags can be accessed by external logic (using inquire cycles) or the processor (for internal snoops) in parallel with accesses to the linear tag by programs running on the processor.

Table 2-2 shows all possible cache-line states before and after program-generated accesses to individual cache lines. The table includes the correspondence between MESI states and *writethrough* or *writeback* states for lines in the data cache. Table 2-3 shows all possible cache-line states before and after cache snoop or invalidation operations performed with inquire cycles. Together, these tables show all of the conditions for writethroughs and writebacks to memory.

TABLE 2-2. Cache States for Read and Write Accesses

Type		Tags[1]	Cache State Before Access[5]	Access Type[2]	Cache State After Access	
					MESI State[5]	Writeback-Writethrough State
Cache Read	Read Miss	Linear	invalid	single read	invalid	invalid
			invalid[3] (cacheable)	burst read (line fill)	shared or exclusive[4]	writethrough or writeback[4]
	Read Hit	Linear	shared	—	shared	writethrough
			exclusive	—	exclusive	writeback
			modified	—	modified	writeback
Cache Write	Write Miss	Linear	invalid	single write	invalid	invalid
	Write Hit	Linear	shared	cache update and single write	shared or exclusive[4]	writethrough or writeback[4]
			exclusive or modified	cache update	modified	writeback

Notes:

1. *Linear tags are masked by $\overline{A20M}$, physical tags are not.*
2. *Single read, single write, cache update, and writethrough = 1 to 8 bytes. Line fill = 32 bytes.*
3. *If $\overline{CACHE}$ and $\overline{KEN}$ are Low.*
4. *If PWT is Low and WB/$\overline{WT}$ is High.*
5. *MESI state is stored in the physical tags. Instruction-cache state consists only of valid (shared) or invalid, and there are no write-related states.*

— Not applicable or none.

TABLE 2-3. Cache States for Snoops, Invalidation, and Replacements

Type of Operation	Tags[1]	Cache State Before Operation[3]	Memory Access[3]	Cache State After Operation		
				MESI State[5]		Writeback-Writethrough State
Inquire Cycle	Physical	shared or exclusive	—	INV=0	shared	writethrough
				INV=1	invalid	invalid
		modified	burst write (writeback)	INV=0	shared	writethrough
				INV=1	invalid	invalid
Internal Snoop	Physical	shared or exclusive	—	invalid		invalid
		modified	burst write (writeback)			
$\overline{\text{FLUSH}}$ Signal	Physical	shared or exclusive	—	invalid		invalid
		modified	burst write (writeback)			
WBINVD Instruction	Physical	shared or exclusive	—	invalid		invalid
		modified	burst write (writeback)			
INVD Instruction	—	—	—	invalid		invalid
Cache-Line Replacement	Physical	shared or exclusive	—	Depends on replacement-line characteristics		
		modified	burst write (writeback)			

Notes:

1. *Linear tags are masked by $\overline{\text{A20M}}$, physical tags are not.*
2. *Writeback = 32 bytes.*
3. *MESI state is stored in the physical tags. Instruction-cache state consists only of valid (shared) or invalid, and there are no write-related states.*

– *Not applicable or none.*

2.3.6 Snooping

The term *snooping* commonly refers to at least three different actions, only two of which are supported by the AMD5$_K$86 and Pentium processors:

- *Inquire Cycles*—These are bus cycles initiated by external logic that cause the processor to look up an address in its physical cache tags. Both the AMD5$_K$86 and Pentium processors support inquire cycles.
- *Internal Snooping*—This is initiated by the processor (rather than external logic) during certain cache accesses. Internal snooping detects self-modifying code. Both the AMD5$_K$86 and Pentium processors support internal snooping.
- *Bus Watching*—Some caching devices watch their address and data buses while they are held off the bus, comparing addresses driven by another bus master with their internal cache tags and optionally updating their cached lines on the fly during writebacks by the other master. The AMD5$_K$86 and the Pentium processor do not support bus watching.

Table 2-4 shows the conditions under which snooping occurs in the AMD5$_K$86 processor and the resources that are snooped. All such snooping is done in the processor's physical tags, in parallel with the processor's own accesses to the linear tags. Thus, there is no execution-performance penalty for snooping.

Inquire Cycles

In systems with multiple caching masters, external logic maintains cache coherency by driving inquire cycles to the processor. System logic initiates inquire cycles by asserting AHOLD, $\overline{\text{BOFF}}$, or HOLD to obtain control of the address bus, and then driving $\overline{\text{EADS}}$, INV and an inquire address. Such bus cycles cause the processor to compare the physical tags for both its instruction and data caches with the inquire address. If the compare hits a *shared* or *exclusive* line in the data cache or a valid line in the instruction cache, the processor asserts $\overline{\text{HIT}}$. If the compare hits a *modified* line in the data cache, the processor asserts $\overline{\text{HITM}}$.

The resulting state of a cache line that is hit depends on the state of the INV signal at the time of the inquire cycle. If INV is negated, the line remains in or transitions to the *shared* (or valid) state. If INV is asserted, the modified line in the data cache is written back, and the line is invalidated.

TABLE 2-4. Snoop Action

Origin of Snoop	Type of Access		Snooping Action				
			Instructions		Data		
			Instruction Cache	Prefetch Cache	Data Cache	Store Buffer	Writeback Buffers
External	Inquire Cycle		yes[1]	yes	yes[1]	no	yes[1]
Internal	Instruction Cache	Read Miss	—	—	yes[2]	yes[2]	yes[2]
		Read Hit	—	—	no	no	no
	Data Cache	Read Miss	yes[3]	yes[3]	—	—	—
		Read Hit	no	no	—	—	—
		Write Miss	yes[4]	yes[4]	—	—	—
		Write Hit	no	no	—	—	—

Notes:

1. *The processor's response to a snoop hit depends on the state of the INV input signal and the state of the cache line as follows: For instructions if INV is negated, the line remains invalid or shared, but if INV is asserted, the line is invalidated. For data if INV is negated, valid lines remain in or transition to the shared state, a modified data cache line is written back before the line is marked shared (with HITM asserted), invalid lines remain invalid.*
   For data if INV is asserted, the line is marked invalid. Modified lines are written back before invalidation.
2. *If the snoop hits a line in the data cache, store buffer or writeback buffer, the line is written back (if modified) and invalidated. Then the instruction-cache read is performed again. If the line is modified, a copy of the writeback data is passed directly to the instruction cache, thus avoiding a line-fill bus cycle after the writeback bus cycle.*
3. *If the snoop hits a line in the instruction cache, prefetch cache, or line-fill buffer, the line stays valid and the data-cache read is performed again, but as a single, non-cacheable read.*
4. *If the snoop hits a line in the instruction cache, prefetch cache, or line-fill buffer, the line is invalidated and the data-cache write is performed.*

— *Not applicable.*

Internal Snooping

The processor automatically snoops its instruction cache during read or write misses to its data cache, and it snoops its data cache during read misses to its instruction cache. It does this to detect the presence of self-modifying code. Table 2-4 summarizes the actions taken during this internal snooping.

If an internal snoop hits its target, the processor does the following:

- *During Instruction-Cache Read Miss*—The line in the data cache, store buffer, or writeback buffer is written back (if modified) and invalidated, and the instruction-cache read is performed again. If the data-cache line was modified, a copy of the writeback data is passed directly to the instruction cache, thus avoiding a line-fill bus cycle after the writeback bus cycle.
- *During Data-Cache Read Miss*—The line in the instruction cache, prefetch cache, or line-fill buffer stays valid, and the data-cache read is performed as a single, non-cacheable read.
- *During Data-Cache Write Miss*—The line in the instruction cache, prefetch cache, or line-fill buffer is invalidated, the reorder buffer invalidates all instructions in the pipeline following the instruction that initiated the snoop, and the data-cache write is performed.

The AMD5$_K$86 processor, like the 486 processor but unlike the Pentium processor, requires a jump (near or far) after a self-modifying write to clear the prefetch cache. However, both the AMD5$_K$86 and the Pentium processors require a serializing instruction after self-modifying code whose physical address is aliased to multiple linear addresses.

2.3.7 Buffers

Several buffers are associated with the instruction and data caches, as described below.

Line-Fill Buffers

The processor has two 16-byte line-fill buffers in the bus interface unit, one of which is used during instruction-cache line fills and the other during data-cache line fills. The buffer holds half of the 32-byte burst cycle that the processor drives in response to a cacheable fetch miss.

Instruction-cache lines are 16 bytes wide. During fetch misses, the first 16 bytes of the burst go through the prefetch cache to the instruction cache and/or byte queue. The remaining 16 bytes from the 32-byte burst cycle, if they are not used immediately thereafter to fill the prefetch cache, are held in a 16-byte line-fill buffer in the bus interface unit for a possible future

access. As shown in Table 2-4, the line-fill buffer for the instruction cache is snooped internally during read or write misses in the data cache, but it is not snooped during inquire cycles. The line-fill buffer for the data cache, unlike the instruction-cache buffer, is never snooped and for this reason does not appear in Table 2-4.

Prefetch Cache

The processor prefetches instructions during fetch-stage misses in the instruction cache, as described in Section 2.1 on page 2-3. When a miss occurs, the processor initiates a 32-byte access for a 16-byte line fill and additional sequentially addressed bytes to fill the prefetch cache. During non-cacheable accesses, the fetch logic fetches directly from the prefetch cache.

As shown in Table 2-4 on page 2-22, the prefetch cache is snooped internally during read or write misses in the data cache and during inquire cycles.

Store Buffer

The Pentium processor implements a write buffer in which real-state data writes can be buffered, waiting for access to the bus, and in which certain types of cacheable read cycles on the bus are promoted ahead of certain types of write cycles when the $\overline{\text{EWBE}}$ signal is asserted. The AMD5$_K$86 processor has no such real-state write buffer between its data cache and the bus, although it does implement a speculative-state, 4-entry, 4-byte-wide store buffer between the two load/store execution units and the data cache.

The store buffer can contain both speculative- and real-state data. Each entry in the store buffer is in speculative state until the associated ROP is retired, after which the data is transferred to the data cache and/or memory, both of which represent the real (non-speculative) state of data. A *store* occurs at the retirement stage of the pipeline, when the processor writes an entry from the store buffer to the data cache and/or memory. For non-cacheable stores, the processor writes directly from the store buffer to the bus interface, at which point the store becomes real-state.

As shown in Table 2-4 on page 2-22, the store buffer is not snooped during inquire cycles. When external logic drives an inquire cycle, the processor's response depends only on the contents of the data cache at that time (that is, only on its real state). Subsequent stores to that line—be they in the store

buffer, load/store execution units, reservation stations, decode unit, or prefetch cache—are not relevant to an inquire cycle or internal snoop. Such stores are speculative and might never occur, due to a branch misprediction, an interrupt, or other intervening event.

As a buffered store leaves the store buffer to update the data cache and/or memory, the processor checks the location's MESI state in the physical tags and observes the MESI update rules for that state. For example, if a buffered store were going to hit an *exclusive* line in the data cache when first placed in the store buffer, but the line's MESI state was changed from *exclusive* to *shared* by a subsequent inquire cycle while the store waited in the store buffer, the store would see a *shared* state on being transferred to the data cache, and it would become a *writethrough*, going externally to main memory at the same time that it updates the data cache.

Replacement and Invalidation Writeback Buffer

The processor has a 1-entry, 32-byte-wide writeback (copy-back) buffer in the bus interface unit for replacements and invalidations. The buffer is used for writebacks of *modified* data in the data cache due to one of the following:

- Cache-line replacement during data-cache read miss
- WBINVD instruction
- $\overline{\text{FLUSH}}$ signal

During cache-line replacements, the memory read cycle for the new cache line is initiated on the bus before the contents of the *modified* line to be replaced are copied into the writeback buffer. When the cache-line fill is completed, the contents of the writeback buffer are written to memory.

Writethroughs from the data cache do not go through a buffer. These transfers are between 1 and 8 bytes in length and they go directly onto the bus from the store buffer.

As shown in Table 2-4 on page 2-22, the writeback buffer is snooped internally during instruction-cache read misses and during inquire cycles.

Snoop Writeback Buffer

In addition to the replacement and invalidation writeback buffer, the processor also has a 1-entry, 32-byte-wide snoop writeback buffer in the bus interface unit that is used for writebacks due to one of the following:

- Internal snoop during an instruction-cache read miss
- External inquire cycle in which the INV signal is asserted

A modified data-cache line can be replaced in parallel with a snoop-hit invalidation to a modified line because the write-backs go to separate buffers.

2.4 Memory Management Unit (MMU)

The MMU supports standard x86 demand-paged virtual memory by translating linear addresses to physical addresses. To speed this process, the most recently accessed address translations are stored in one of two translation lookaside buffers (TLBs), one for mapping 4-Kbyte pages and another for mapping 4-Mbyte pages. Mappings to 4-Kbyte and 4-Mbyte pages can be intermixed in a given page directory, the base of which is pointed to by the contents of control register 3 (CR3).

During memory accesses, the MMU receives a linear address and searches the TLBs for a corresponding physical address. If found, the physical address is passed to the physical tag directory for a validity check. If the physical address is not present (a TLB miss), the MMU searches the page directory and page tables in memory. If found, the MMU loads the translation into the appropriate TLB. If not found, the processor generates a page fault.

2.4.1 Storage Model

The AMD5$_K$86 processor always observes the *strongly ordered memory-write model*. All writes—whether to cache, memory, or I/O—are performed in program order, regardless of the state of the External Write Buffer Empty ($\overline{\text{EWBE}}$) input signal. The only effect of $\overline{\text{EWBE}}$ on writes is to hold additional writes off when the signal is negated. In particular, assertion of $\overline{\text{EWBE}}$ does not permit the AMD5$_K$86 processor to observe a weakly ordered memory-write model, in which writes to cache may

occur out of program order with respect to writes on the bus to memory.

In a strongly ordered memory-write model, writes to cache and memory always appear in program order. In a weakly ordered memory-write model, writes to cache and memory can occur out of program order (that is, a write to cache can occur before a prior write to memory). Weakly ordered systems may perform better, but they can cause problems in systems with multiple-caching masters. For example, errors may occur in weakly ordered systems when a master that is held off the bus continues writing to *exclusive* or *modified* lines in its internal data cache while another master writes to memory. Nevertheless, the strongly ordered AMD5$_K$86 processor supports high performance without using weakly ordered memory writes by buffering speculative stores in the store buffer.

2.4.2 Read/Write Reordering

The processor reorders certain types of cacheable read cycles on the bus ahead of certain types of write cycles. Specifically, any read that hits in the instruction or data cache is promoted ahead of a write in the store buffer if the read is not from the same location to which a write in the store buffer is to be written. The reordering allows reads, which dominate the processor's use of the bus in Writeback mode, to take precedence over data writes, which normally occur infrequently. The EWBE signal has no effect on this reordering of bus cycles.

2.4.3 Segmentation

The instruction cache contains a copy of certain fields in the current code-segment descriptor. The information is used during prefetch for segment translation (logical-to-linear addresses), thus providing linear-address tags for the instruction-cache entries. Likewise, the load/store units hold the current data-segment descriptors, which are used to generate the linear address and perform protection checks during data-cache accesses. The processor can cache segment descriptors in its data cache.

2.4.4 Paging and the TLBs

The processor supports 4-Kbyte and 4-Mbyte paging with two separate translation lookaside buffers (TLBs) that work in parallel:

- *4-Kbyte Pages*—A 128-entry, four-way, set-associative TLB that can cover 512 Kbytes of memory space
- *4-Mbyte Pages*—A four-entry, fully-associative TLB that can cover 16 Mbytes of memory space

The TLBs are accessed during cache accesses that miss in the linear tags. Each TLB is organized into tag directories (linear-address references) and data arrays (physical-address references). The TLB entries also contain bits used to check privilege and access rights. Because the caches are linearly addressed, however, cache accesses do not go through the TLB. The cache accesses are faster because the TLB is not involved. Copies of the privilege and access bits from the TLB entries are loaded into the caches when the cache lines are filled. If a privilege-level violation is detected during a cache access, the TLB is accessed, and it alone can issue a page-related exception.

TLB invalidations (flushes) are done in the standard ways: a MOV to CR3, which loads a new page-table directory, or the INVLPG instruction, which invalidates a single TLB entry. Both the 4-Kbyte and 4-Mbyte TLBs support global pages, which remain in the TLBs during such TLB invalidations when the global-page extension is enabled.

When a TLB miss or fault occurs during a prefetch, bits reflecting this are passed via the prefetch cache to the decode logic during fetch misses so that microcode can serialize the pipeline and initiate the TLB reload nonspeculatively. TLB replacement is done using a pseudo-random algorithm. The processor never caches TLB loads, regardless of the state of the PCD and PWT bits, and it does not do speculative TLB reloads. A page-fault handler, however, may cache page-table entries in the data cache. During a TLB reload, the physical cache tags are snooped for the page-table entry (PTE). A hit on a *modified* line causes that line to be written back to memory. The TLB then completes the read from memory. The TLB always performs reloads from memory, regardless of whether a page-directory entry (PDE) or page-table entry (PTE) is in the data cache. If

the TLB reload involves a write to memory to set the PDE Accessed or Dirty bit, a hit during the physical-tag snoop causes the cache line to be invalidated.

Details on software configuration for 4-Mbyte paging are given in Section 3.1.2 on page 3-5. The global-page option is described in Section 3.1.3 on page 3-9. Details on the TLB storage formats and their testing are given in Section 7.4 on page 7-7.

3

Software Environment and Extensions

The AMD5$_K$86 processor is compatible with the instruction set, programming model, memory management mechanisms, and other software infrastructure supported by the 486 and Pentium (735\90, 815\100) processors. Operating system and application software that runs on the Pentium processor can be executed on the AMD5$_K$86 processor without modification. Because the AMD5$_K$86 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.

The AMD5$_K$86 processor implements the following extensions to the 486 architecture:

- 4-Mbyte Page Size
- Global Pages
- Protected Virtual Extensions
- Virtual-8086 Mode Extensions (VME)
- Machine-Check Registers and Exceptions
- Model-Specific Registers (MSRs)
- Time Stamp Counter (TSC)
- New Instructions: CPUID, CMPXCHG8B, MOV to and from CR4, RDTSC, RDMSR, WRMSR, and RSM
- I/O Breakpoints

The sections that follow provide details on the architectural extensions visible to system and application software. Some sections include pseudo-code algorithms for suggested BIOS modifications to support the extensions. Architectural extensions visible to debug and test software, such as I/O breakpoints, are described in Chapter 7.

3.1 Control Register 4 (CR4) Extensions

Control register 4 contains bits that enable or specify many of the extensions to the 486 architecture. The majority of the bits in CR4 are reserved. The default state for all bits in CR4 is all zeros. Figure 3-1 shows the format of CR4. Table 3-1 describes the fields in CR4.

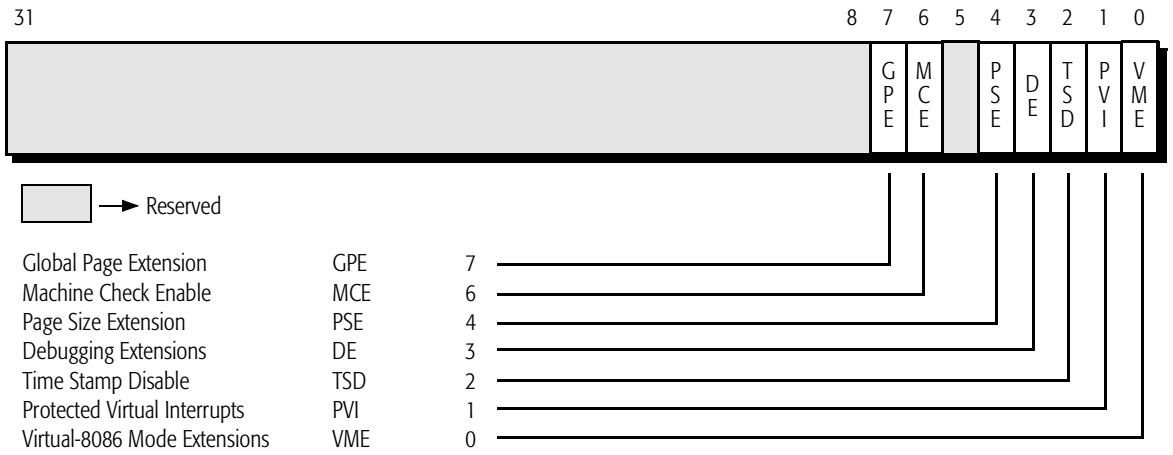


FIGURE 3-1. Control Register 4 (CR4)

TABLE 3-1. 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 Section 3.1.3 on page 3-9 for details.
6	MCE	Machine-Check Enable	Enables machine-check exceptions. 1 = enabled, 0 = disabled. See Section 3.1.1 on page 3-4 for details.
4	PSE	Page Size Extension	Enables 4-Mbyte pages. 1 = enabled, 0 = disabled. See Section 3.1.2 on page 3-5 for details.
3	DE	Debugging Extensions	Enables I/O breakpoints in the DR7–DR0 registers. 1 = enabled, 0 = disabled. See Section 7.5 on page 7-16 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 Section 3.2.3 on page 3-27 for details.
1	PVI	Protected Virtual Interrupts	Enables hardware support for interrupt virtualization in Protected mode. 1 = enabled, 0 = disabled. See Section 3.1.5 on page 3-24 for details.
0	VME	Virtual-8086 Mode Extensions	Enables hardware support for interrupt virtualization in Virtual-8086 mode. 1 = enabled, 0 = disabled. See Section 3.1.4 on page 3-12 for details.

3.1.1 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 in Section 3.3.5 on page 3-35. The format of the registers is shown in Figure 3-8 on page 3-25 and Figure 3-9 on page 3-26.

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, although it still latches the address and cycle type in MCAR and MCTR as described above.

3.1.2 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 AMD5$_K$86 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 3-2 and 3-3 show how 4-Kbyte and 4-Mbyte page translation work.

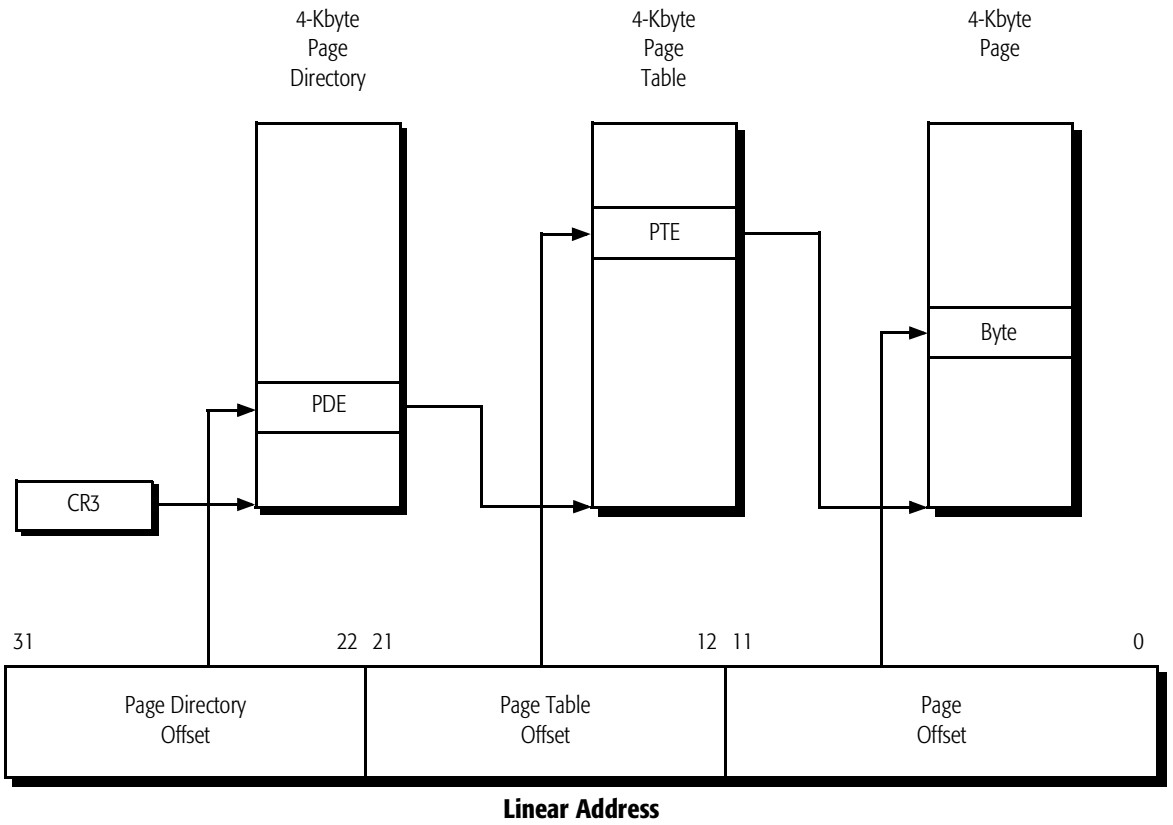


FIGURE 3-2. 4-Kbyte Paging Mechanism

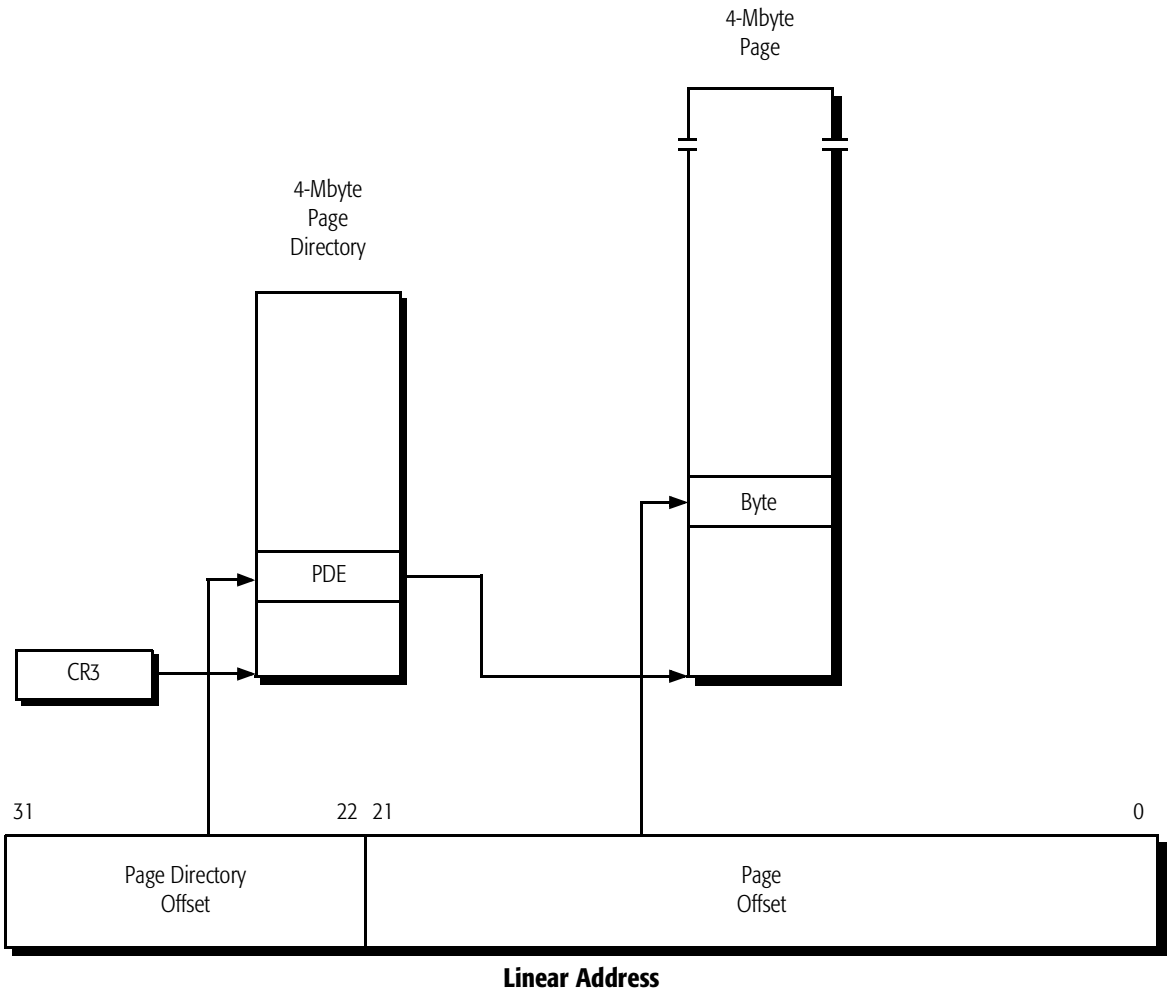


FIGURE 3-3. 4-Mbyte Paging Mechanism

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 3-1 and Table 3-1 show the fields in CR4. Figure 3-4 and Table 3-2 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 3-2)*—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 3-3)*—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.

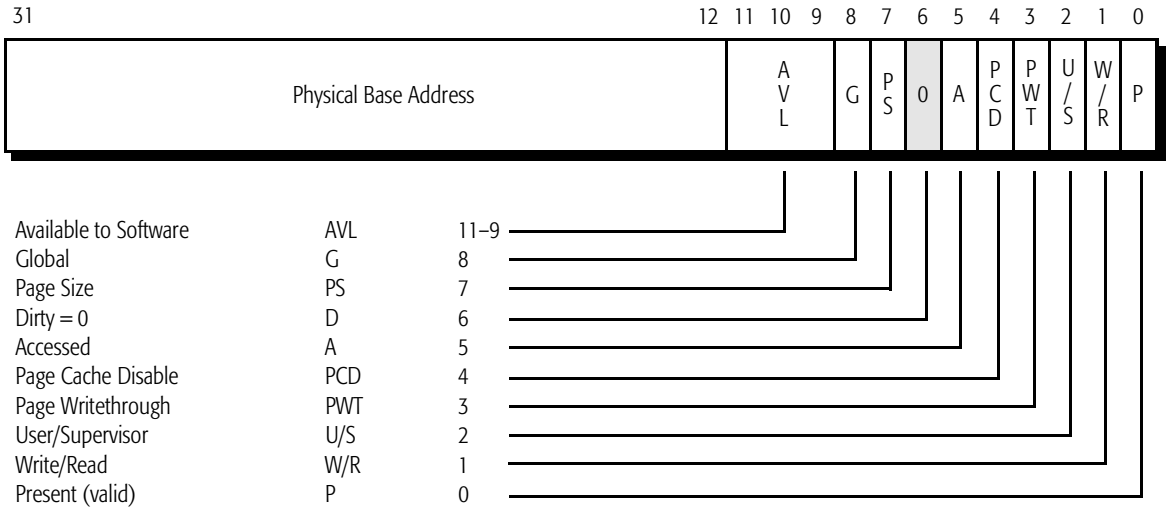


FIGURE 3-4. Page-Directory Entry (PDE)

TABLE 3-2. 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 = writethough 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.

3.1.3 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 3-4 and Table 3-2) and the page-table entry (shown in Figure 3-5 and Table 3-3).

   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 3-4 and Table 3-2).
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.

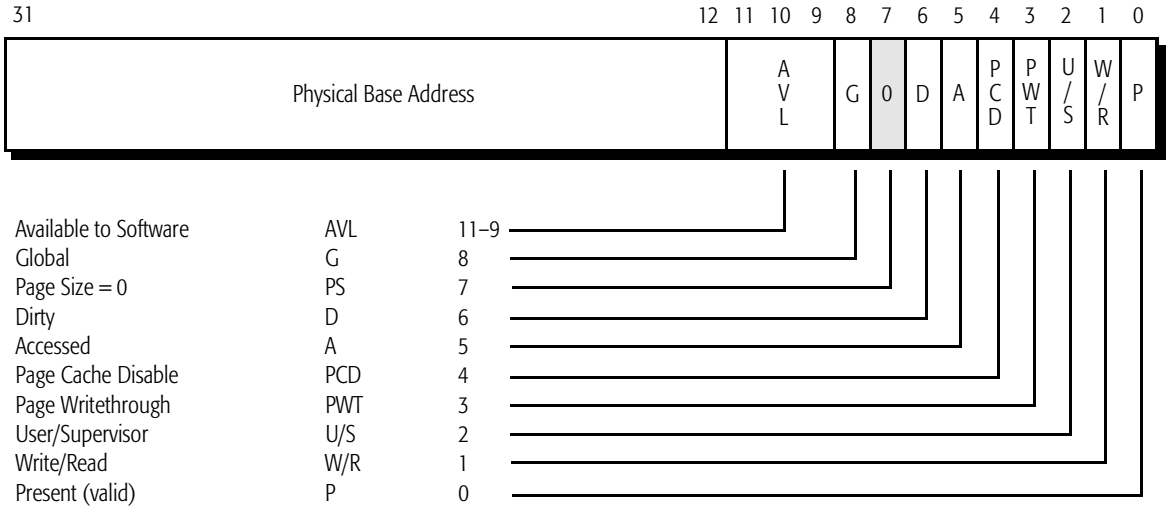


FIGURE 3-5. Page-Table Entry (PTE)

TABLE 3-3. 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 = writethough.
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.

3.1.4 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 AMD5$_K$86 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 3-6 and Table 3-4 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 page 3-21). 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.

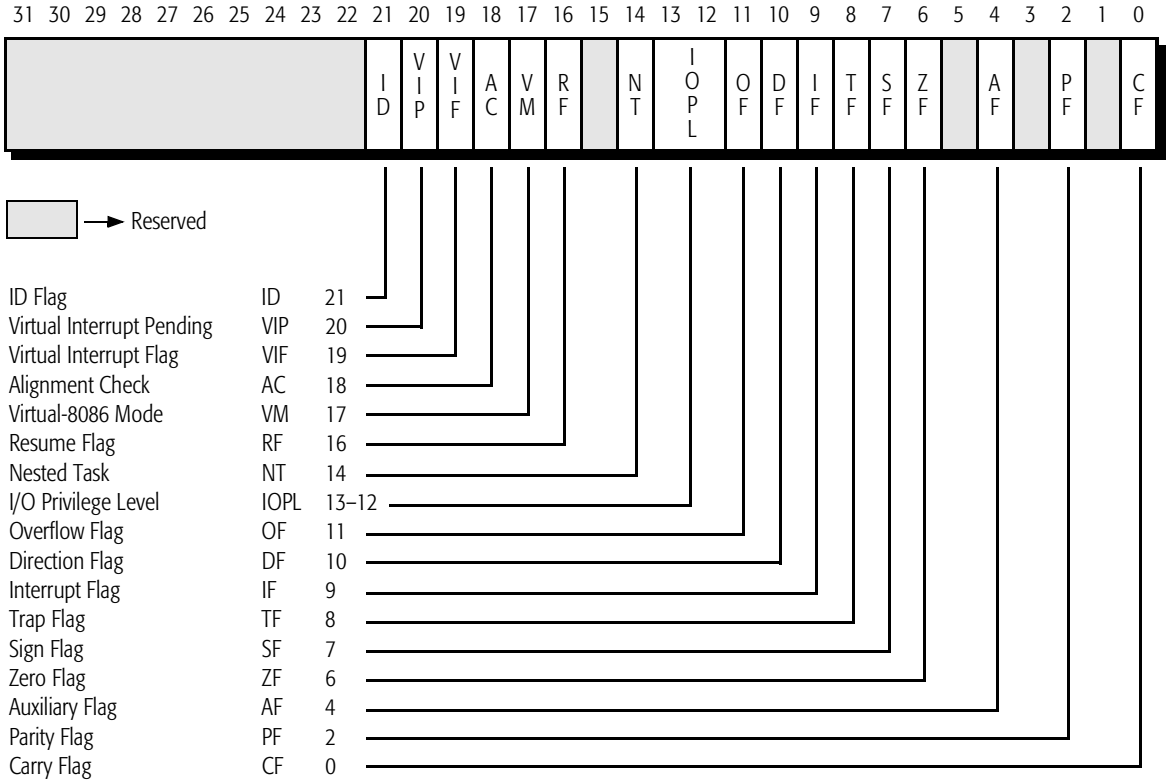


FIGURE 3-6. EFLAGS Register

TABLE 3-4. 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 3-5 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)
- *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 3-5. Instructions that Modify the IF or VIF Flags

Mode	TYPE	PE	VM	VME	PVI	IOPL	GP(0)	IF	VIF
Real Mode[1]	CLI	0	0	0	0	—	No	IF ← 0	—
	STI	0	0	0	0	—	No	IF ← 1	—
	PUSHF	0	0	0	0	—	No	No Change	—
	POPF	0	0	0	0	—	No	IF ← Stack Image	—
	IRET	0	0	0	0	—	No	IF ← Stack Image	—

Notes:

1. *All Virtual-8086 tasks run at CPL = 3.*
2. *INTn handlers and IRETO instructions run at CPL = 0. GP(0) if an attempt is made to set VIF when VIP = 1.*

– Not applicable.

TABLE 3-5. Instructions that Modify the IF or VIF Flags (continued)

Mode	TYPE	PE	VM	VME	PVI	IOPL	GP(0)	IF	VIF
286 Protected Mode	CLI	1	0	0	0	≥CPL	No	IF ← 0	—
	CLI	1	0	0	0	<CPL	Yes	No Change	—
	STI	1	0	0	0	≥CPL	No	IF ← 1	—
	STI	1	0	0	0	<CPL	Yes	No Change	—
	PUSHF	1	0	0	0	≥CPL	No	No Change	—
	PUSHF	1	0	0	0	<CPL	No	No Change	—
	POPF	1	0	0	0	≥CPL	No	IF ← Stack Image	—
	POPF	1	0	0	0	<CPL	No	No Change	—
	IRET	1	0	0	0	≥CPL	No	IF ← Stack Image	—
	IRET	1	0	0	0	<CPL	No	IF ← Stack Image	—

Notes:

1. *All Virtual-8086 tasks run at CPL = 3.*
2. *INTn handlers and IRETO instructions run at CPL = 0. GP(0) if an attempt is made to set VIF when VIP = 1.*

— Not applicable.

TABLE 3-5. Instructions that Modify the IF or VIF Flags (continued)

Mode	TYPE	PE	VM	VME	PVI	IOPL	GP(0)	IF	VIF
386 Virtual-8086 Mode[1]	CLI	1	1	0	—	≥CPL	No	IF ← 0	—
	CLI	1	1	0	—	<CPL	Yes	No Change	—
	STI	1	1	0	—	≥CPL	No	IF ← 1	—
	STI	1	1	0	—	<CPL	Yes	No Change	—
	PUSHF	1	1	0	—	≥CPL	No	No Change	—
	PUSHF	1	1	0	—	<CPL	Yes	No Change	—
	POPF	1	1	0	—	≥CPL	No	IF ← Stack Image	—
	POPF	1	1	0	—	<CPL	Yes	No Change	—
	IRETD	1	1	0	—	≥CPL	No	IF ← Stack Image	—
	IRETD	1	1	0	—	<CPL	Yes	No Change	—

Notes:

1. *All Virtual-8086 tasks run at CPL = 3.*
2. *INTn handlers and IRETO instructions run at CPL = 0. GP(0) if an attempt is made to set VIF when VIP = 1.*

— Not applicable.

TABLE 3-5. Instructions that Modify the IF or VIF Flags (continued)

Mode	TYPE	PE	VM	VME	PVI	IOPL	GP(0)	IF	VIF
Virtual-8086 Mode Extensions (VME)[1, 2]	CLI	1	1	1	—	3	No	IF ← 0	No Change
	CLI	1	1	1	—	0	No	No Change	VIF ← 0
	STI	1	1	1	—	3	No	IF ← 1	No Change
	STI	1	1	1	—	0	No	No Change	VIF ← 1
	PUSHF	1	1	1	—	3	No	Pushed	Not Pushed
	PUSHF	1	1	1	—	0	No	Not Pushed	Pushed into stack IF
	PUSHFD	1	1	1	—	3	No	Pushed	Pushed
	PUSHFD	1	1	1	—	0	Yes	—	—
	POPF	1	1	1	—	3	No	Popped	Not Popped
	POPF	1	1	1	—	0	No	Not Popped	Popped from stack IF
	POPFD	1	1	1	—	3	No	Popped	Not Popped
	POPFD	1	1	1	—	0	Yes	—	—
	IRETD	1	1	1	—	3	No	IF ← Return Stack Image	VIF ← Return Stack Image
	IRETD	1	1	1	—	0	No	IF ← Return Stack Image	VIF ← Return Stack Image

Notes:

1. *All Virtual-8086 tasks run at CPL = 3.*
2. *INTn handlers and IRETO instructions run at CPL = 0. GP(0) if an attempt is made to set VIF when VIP = 1.*

— Not applicable.

TABLE 3-5. Instructions that Modify the IF or VIF Flags (continued)

Mode	TYPE	PE	VM	VME	PVI	IOPL	GP(0)	IF	VIF
Protected Virtual Extensions (PVI)[1, 2]	CLI	1	0	—	1	3	No	IF ← 0	No Change
	CLI	1	0	—	1	0	No	No Change	VIF ← 0
	STI	1	0	—	1	3	No	IF ← 1	No Change
	STI	1	0	—	1	0	No	No Change	VIF ← 1
	PUSHF	1	0	—	1	3	No	Pushed	Not Pushed
	PUSHF	1	0	—	1	0	No	Pushed	Not Pushed
	PUSHFD	1	0	—	1	3	No	Pushed	Pushed
	PUSHFD	1	0	—	1	0	No	Pushed	Pushed
	POPF	1	0	—	1	3	No	Popped	Not Popped
	POPF	1	0	—	1	0	No	Not Popped	Not Popped
	POPFD	1	0	—	1	3	No	Popped	Not Popped
	POPFD	1	0	—	1	0	No	Not Popped	Not Popped
	IRETD	1	0	—	1	3	No	IF ← Return Stack Image	VIF ← Return Stack Image
	IRETD	1	0	—	1	0	No	IF ← Return Stack Image	VIF ← Return Stack Image

Notes:

1. *All Virtual-8086 tasks run at CPL = 3.*
2. *INTn handlers and IRETO instructions run at CPL = 0. 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 3-7 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 on page 3-13 are provided to assist the operating system in this intervention.

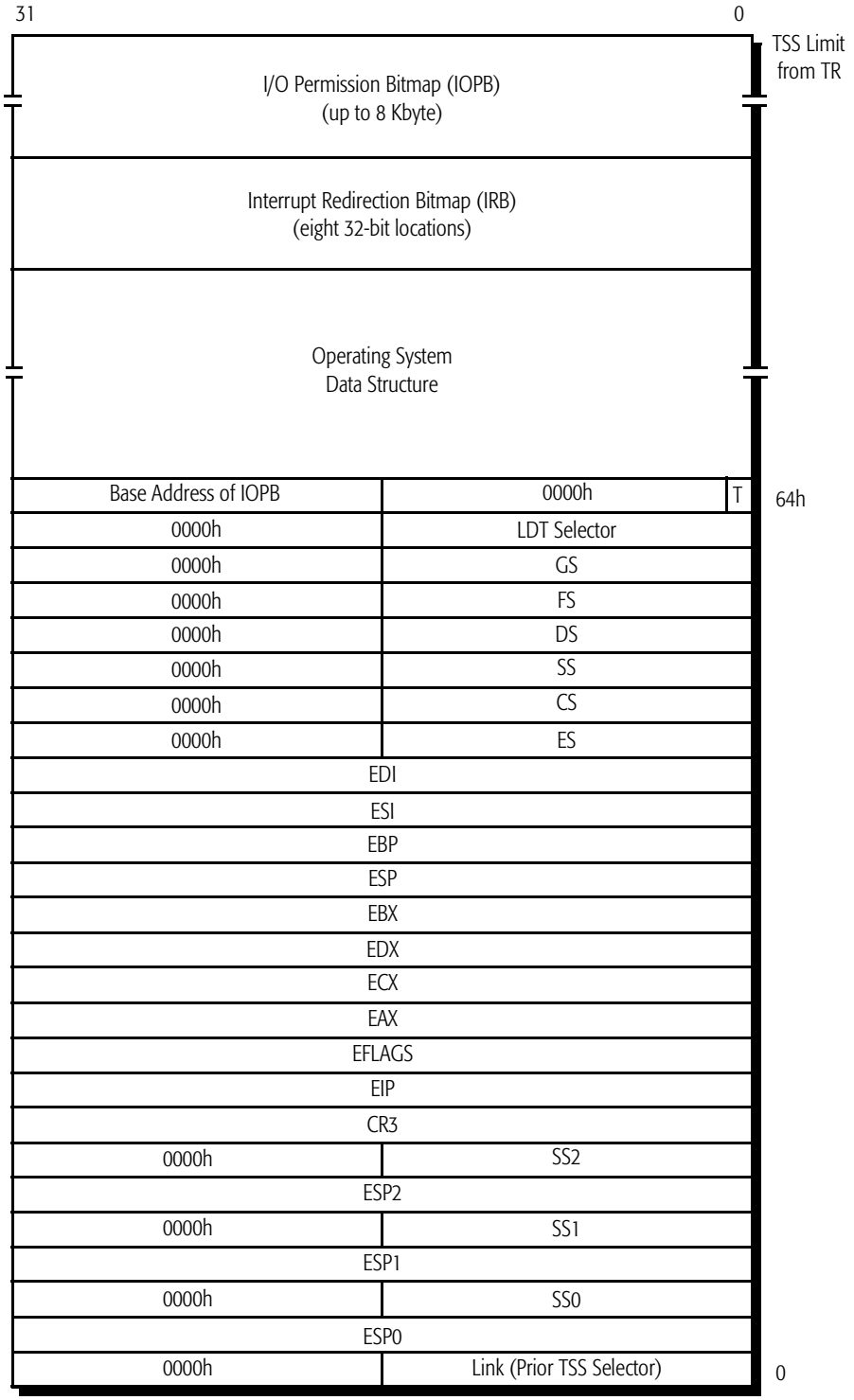


FIGURE 3-7. Task State Segment (TSS)

Table 3-6 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 3-6. 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	—	—	—	3
	Hardware	0	0	0	—	0	—	—	—	3
286 Protected mode	Software	1	0	0	—	—	—	—	3	—
	Hardware	1	0	0	—	—	—	—	3	—
	Software	1	1	0	—	=3	—	No	3	—
386 Virtual-8086 mode[1]	Software	1	1	0	—	<3	—	Yes	3	—
	Hardware	1	1	0	—	—	—	No	3	—
	Software	1	1	1	0	—	0	No		3
Virtual-8086 Mode Extensions (VME)[1]	Software	1	1	1	0	=3	1	No	3	—
	Software	1	1	1	0	<3	1	Yes	3	—
	Hardware	1	1	1	0	—	—	No	3	—
Protected Virtual Extensions (PVI)	Software	1	0	1	1	—	—	No	3	—
	Hardware	1	0	1	1	—	—	No	3	—

Notes:

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

– *Not applicable.*

3.1.5 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.

Tables 3-5 and 3-6 show, among other things, the behavior of hardware and software interrupts, and instructions that affect interrupts, in Protected mode with the PVI extensions enabled.

3.2 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 in Section 3.3.5 on page 3-35. The following sections describe the format of the registers.

3.2.1 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 3-8 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 in Section 3.1.1 on page 3-4.

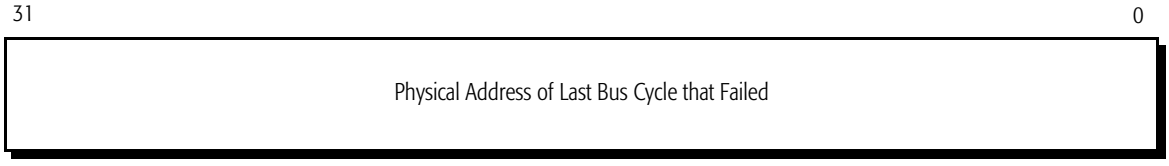


FIGURE 3-8. Machine-Check Address Register (MCAR)

3.2.2 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 (MCAR) 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 3-9 and Table 3-7 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 in Section 3.1.1 on page 3-4.

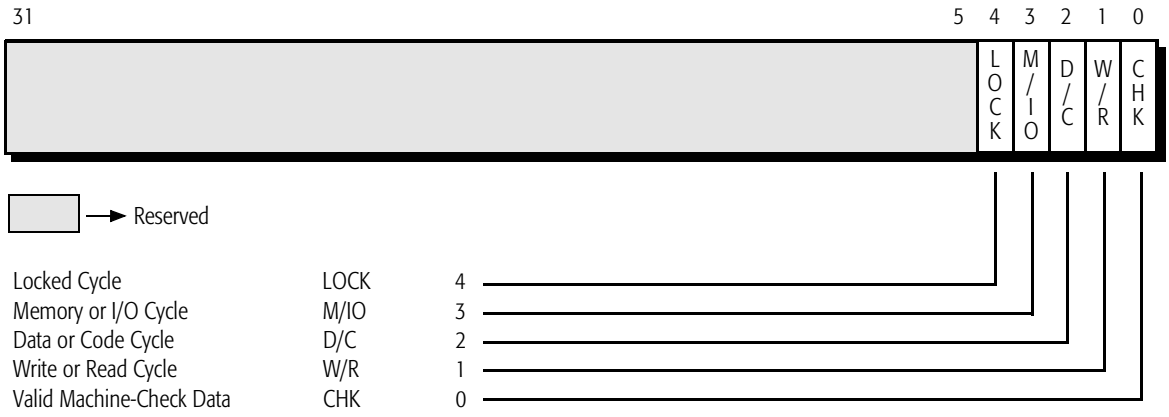


FIGURE 3-9. Machine-Check Type Register (MCTR)

TABLE 3-7. 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.

3.2.3 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 Section 3.3.4 on page 3-34) 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.

3.2.4 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 Section 7.4 on page 7-7.

3.2.5 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 Section 7.1 on page 7-3.

3.3 New Instructions

In addition to supporting all of the 486 processor instructions, the AMD5$_K$86 processor implements the following instructions:

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

3.3.1 CPUID

mnemonic	*opcode*	*description*
CPUID	0FA2h	Identify processor

Privilege:	CPL=0
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.

If 0 is written to EAX, the following values are returned in EAX, EBX, ECX, and EDX:

- EAX: 00000001h
- EBX: 68747541h
- ECX: 444D4163h
- EDX: 69746E65h

These values decode to the ASCII string “AuthenticAMD” when read in the EBX-EDX-ECX registers in least significant byte to most significant byte order.

If 1 is written to EAX, the following value is returned in the bit locations of EAX and EDX:

- EAX:
  - EAX[3–0] Stepping ID
  - EAX[7–4] Model:

    AMD-SSA5 processor (0, 0000b)
    AMD5$_K$86 processor (1, 0001b)
  - EAX[11–8] Family (0101b)
  - EAX[31–12] *reserved*

- EDX:
  - EDX[0] FPU on chip (1 = FPU, 0 = no FPU)
  - EDX[1] Virtual Mode Extensions (1 = support, 0 = no support)
  - EDX[2] I/O Breakpoints (1 = support, 0 = no support)
  - EDX[3] 4-Mbyte Pages (1 = support, 0 = no support)
  - EDX[4] Time Stamp Counter (1 = support, 0 = no support)
  - EDX[5] K86™ Model-Specific Registers (1 = support, 0 = no support)
  - EDX[6] Reserved
  - EDX[7] Support of machine-check exception (1 = supported)
  - EDX[8] Execution of CMPXCHG8B instruction (1 = supported)
  - EDX[9] Global Paging Extension (1 = supported)
  - EDX[31–10] *reserved*

The following pseudo-code illustrates the use of the CPUID instruction:

```
begin
{
if vendor string report desired
      {
      load EAX with 0h
      execute CPUID instruction (opcode = 0Fh 0A2h)
      Result:
      EBX = 'Auth'
      EDX = 'enti'
      ECX = 'cAMD'
      }
else if CPU information desired
      {
      load EAX with 1
      execute CPUID instruction (opcode = 0Fh 0A2h)
      Result:
      EAX[3-0] = stepping ID (contact AMD for specifics)
      EAX[7-4] = Model
            AMD-SSA5 processor -> 0000b
            AMD5K86 processor -> 0001b
      EAX[11-8] = Family
            K5 CPU -> 5
      EAX[31-12] = Reserved
      EBX = 00000000h
      ECX = 00000000h
      EDX[0] = 1b (bit 0==1 indicates FPU present)
      EDX[1] = 1b (bit 1==1 indicates Virtual Mode Extensions)
      EDX[2] = 1b (bit 2==1 indicates I/O Breakpoints)
      EDX[3] = 1b (bit 3==1 indicates 4-Mbyte pages)
      EDX[4] = 1b (bit 4==1 indicates Time Stamp Counter)
```

```
      EDX[5] = 1b (bit 5==1 indicates K86 Model-Specific Registers)
      EDX[6] = 0b Reserved
      EDX[7] = Support of machine check exception (bit 7==1 indicates support)
      EDX[8] = Support of CMPXCHG8B instruction (bit 8==1 indicates support)
      EDX[9] = Support of global paging extension (bit 9==1 indicates support)
      EDX[31-10] = Reserved
      }
}
end
```

3.3.2 CMPXCHG8B

mnemonic	*opcode*	*description*
CMPXCHG8B *r/m64*	0FC7	Compare and exchange 8-byte operand

Privilege:	CPL = 0
Registers Affected:	EAX, EBX, ECX, EDX
Flags Affected:	ZF
Exceptions Generated:	Real, Virtual-8086, Protected mode—GP(0) for all standard cases. 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

3.3.3 MOV to and from CR4

mnemonic	*opcode*	*description*
MOV CR4,*r32*	0F22	Move to CR4 from register
MOV *r32*,CR4	0F20	Move to register from CR4

Privilege:	CPL = 0
Registers Affected:	CR4, 32-bit general-purpose register
Flags Affected:	none
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).

3.3.4 RDTSC

mnemonic	*opcode*	*description*
RDTSC	0F31	Read time stamp counter

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

The 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
```

3.3.5 RDMSR and WRMSR

mnemonic	*opcode*	*description*
RDMSR	0F32	Read model-specific register (MSR)
WRMSR	0F30	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 Section 3.1.1 on page 3-4.
- *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 Section 3.1.1 on page 3-4. 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 Section 3.2.3 on page 3-27.
- *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 Section 7.4 on page 7-7.
- *83h*: Hardware Configuration Register (HWCR). This contains configuration bits that control miscellaneous debugging functions. For details, see Section 7.1 on page 7-3.

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 AMD5$_K$86 processor.

3.3.6 RSM

mnemonic	*opcode*	*description*
RSM	0FAA	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.

3.3.7 Illegal Instruction (Reserved Opcode)

mnemonic	*opcode*	*description*
(none)	0FFF	Illegal instruction (reserved opcode)

Privilege:	none
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.

4

Performance

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.

4.1 Code Optimization

The code optimization suggestions in this section cover both general superscalar optimization (that is, techniques common to both the AMD5$_K$86 and Pentium processors) and techniques specific to the AMD5$_K$86 processor. In general, all optimization techniques used for the Pentium processor apply to any wide-issue x86 processor, but wider-issue designs like the AMD5$_K$86 processor have fewer restrictions.

4.1.1 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 AMD5$_K$86 processor. However, future implementations may have such a penalty to achieve a higher overall clock rate.

4.1.2 Techniques Specific to the AMD5$_K$86 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 Section 4.2 on page 4-5.)
- *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 AMD5$_K$86 processor take 5 cycles. If the semantics of the register-based bit-offset form are desired (where the bit offset can cover a very large bit string in memory), it is better to emulate this with simpler instructions that can be interleaved with independent instructions for greater parallelism.
- *Floating-Point Top-of-Stack Bottleneck*—The AMD5$_K$86 processor has a pipelined floating-point unit. Greater parallelism can be achieved by using FXCH in parallel with floating-point operations to alleviate the top-of-stack bottleneck, as in the Pentium processor. The AMD5$_K$86 processor also permits integer operations (ALU, branch, load/store) in parallel with floating-point operations.
- *Locating Branch Targets*—Performance can be sensitive to code alignment, especially in tight loops. Locating branch targets to the first 17 bytes of the 32-byte cache line maximizes the opportunity for parallel execution at the target. NOPs can be added to adjust this alignment. The AMD5$_K$86 processor executes NOPs (opcode 90h) at the rate of two per cycle. Adding NOPs is even more effective if they execute in parallel with existing code. Other instructions of greater length, such as a register-based TEST instruction, can be used as NOPs to minimize the overhead of such padding.
- *Branch Prediction*—There are two branch prediction bits in a 32-byte instruction cache line. One bit applies to the first 16 bytes of the line and the second bit applies to the second 16 bytes of the line. For effective branch prediction, code should be generated with one branch per 16-byte line half.
- *Address-Generation Interlocks (AGIs)*—The AMD5$_K$86 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.

4.2 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.

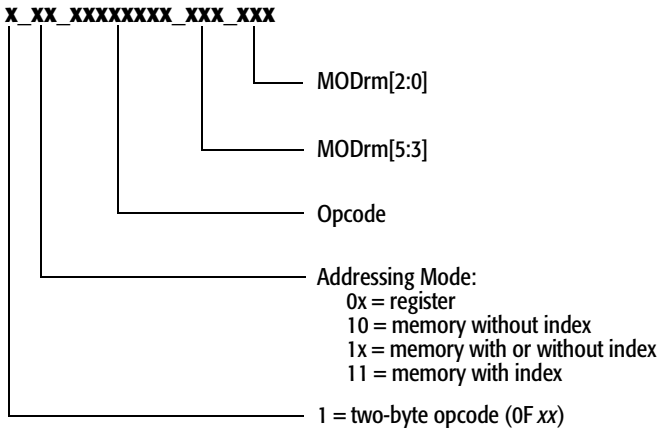
4.2.1 Notation

Table 4-1 on page 4-8 contains the definitions for the integer instructions. Table 4-3 on page 4-19 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 AMD5$_K$86 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 is available to 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.

4.2.2 Integer Instructions

Table 4-1 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 4-1. 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 4-1. 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 4-1. 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 4-1. 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 4-1. 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 4-1. 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 4-1. 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 4-1. 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 4-1. 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 4-1. 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

4.2.3 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 4-2 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 4-2. 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

4.2.4 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 4-3 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 4-3. 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 4-3. 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 4-3. 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 4-3. 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 4-3. 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 4-3. 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 4-3. 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

5

Bus Interface

This chapter describes two closely related subjects, bus signals (Sections 5.1 and 5.2) and the bus-cycle protocols implemented with those signals (Sections 5.3 and 5.4). These sections describe only the architectural characteristics and functions of the signals and bus cycles. The processor data sheet defines the setup and hold times for signals.

Throughout this chapter, unless otherwise stated, the term *clock* refers to bus-clock (CLK) cycles, not processor-clock cycles. The term *cycle* refers to bus cycles not clock cycles. The terms *asserted* and *negated* mean that a signal is sampled asserted or sampled negated by its target on the signal's active (typically rising) clock edge.

5.1 Signal Overview

The signals on the AMD5$_K$86 processor are compatible with the comparable signals on the Pentium (735\90, 815\100) processor 296-pin socket. Appendix A gives a complete list of hardware and software issues relating to this compatibility. The following figures and tables summarize the characteristics and behavior of the AMD5$_K$86 processor's signals:

- Figure 5-1 (Signal Groups) summarizes the processor's signals, showing the functional groups to which each signal belongs (the same figure appears in the introduction to this manual).
- Table 5-1 (Summary of Signal Characteristics) shows each signal's I/O type, when it is sampled, driven, and floated, and its internal resistor (if any).
- Table 5-2 on page 5-9 (Conditions for Driving and Sampling Signals) shows the states and bus cycles during which the processor effectively drives or samples each signal.
- Table 5-3 on page 5-17 (Summary of Interrupts and Exceptions) shows the priority and characteristics of interrupts and exceptions.

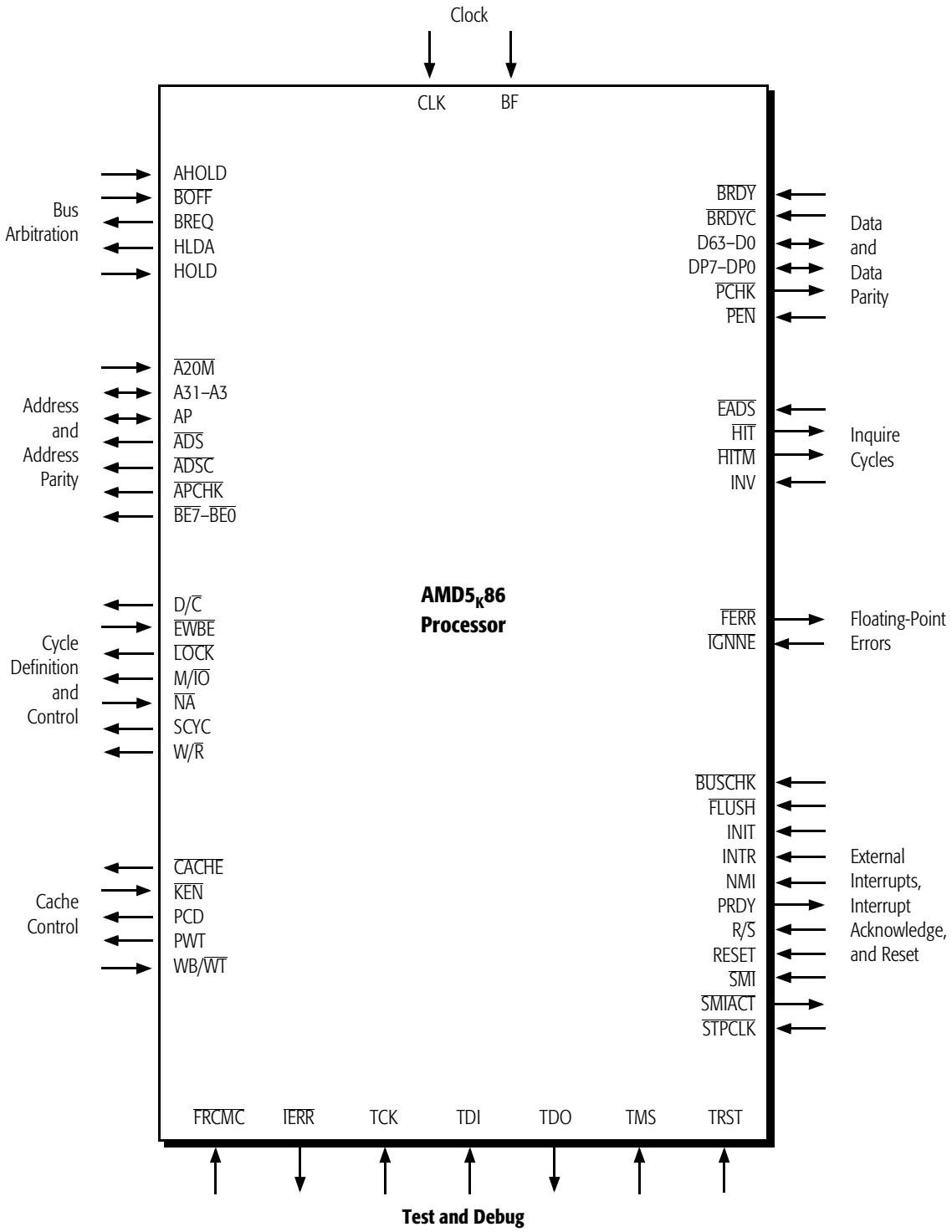


FIGURE 5-1. Signal Groups

5.1.1 Signal Characteristics

TABLE 5-1. Summary of Signal Characteristics

Signal	Type	Sampled (Input) or Asserted (Output)[2]	Internal Resistor	Floated[3]
$\overline{\text{A20M}}$[1]	I	Every clock.		
A31–A3	I/O	*Output:* From $\overline{\text{ADS}}$ until last expected $\overline{\text{BRDY}}$ of the bus cycle. *Input:* Same clock as $\overline{\text{EADS}}$. A4–A3 are disabled for input.		AHOLD +1, $\overline{\text{BOFF}}$ +1 or HLDA
$\overline{\text{ADS}}$	O	First clock of bus cycle.		$\overline{\text{BOFF}}$ +1 or HLDA
$\overline{\text{ADSC}}$	O	First clock of bus cycle.		$\overline{\text{BOFF}}$ +1 or HLDA
AHOLD	I	Every clock.		
AP	I/O	*(same as A31–A3)*		AHOLD +1, $\overline{\text{BOFF}}$ +1 or HLDA
$\overline{\text{APCHK}}$	O	Two clocks after $\overline{\text{EADS}}$, for one clock.		
$\overline{\text{BE7}}$–$\overline{\text{BE0}}$	O	From $\overline{\text{ADS}}$ until the last expected $\overline{\text{BRDY}}$ of the bus cycle.		$\overline{\text{BOFF}}$ +1 or HLDA
BF	I	Falling edge of RESET.	pullup	
$\overline{\text{BOFF}}$	I	Every clock.		
$\overline{\text{BRDY}}$	I	Every clock, from one clock after $\overline{\text{ADS}}$ until the last expected $\overline{\text{BRDY}}$ of the bus cycle.		
$\overline{\text{BRDYC}}$	I	*(same as* $\overline{\text{BRDY}}$*)*	pullup	
BREQ	O	First clock of every bus cycle (same as $\overline{\text{ADS}}$), cache store, cache-tag recovery, and aliased cache load. Asserted continuously while processor is held off bus and needs access to continue.		
$\overline{\text{BUSCHK}}$	I	Every $\overline{\text{BRDY}}$. Recognized at the next instruction boundary.	pullup	
$\overline{\text{CACHE}}$	O	From $\overline{\text{ADS}}$ until the last expected $\overline{\text{BRDY}}$ of the bus cycle. Driven for all reads; only driven for writes during writebacks.		$\overline{\text{BOFF}}$ +1 or HLDA

Notes:

1. *Can be driven asynchronously or synchronously.*
2. *The term clock means bus clock (CLK). "+n" means n CLKs later.*
3. *"+n" means n CLKs after the named signal is sampled active. All outputs and bidirectionals are floated during the float test (FLUSH at RESET).*

TABLE 5-1. Summary of Signal Characteristics (continued)

Signal	Type	Sampled (Input) or Asserted (Output)[2]	Internal Resistor	Floated[3]
CLK	I	Always.		
D/$\overline{C}$	O	From $\overline{ADS}$ until the last expected $\overline{BRDY}$ of the bus cycle.		$\overline{BOFF}$ +1 or HLDA
D63–D0	I/O	*Output (single transfer):* From one clock after $\overline{ADS}$ until $\overline{BRDY}$. *Output (burst transfer):* From one clock after $\overline{ADS}$ until the first $\overline{BRDY}$, and thereafter from one clock after each $\overline{BRDY}$ until the next $\overline{BRDY}$. *Input:* Every $\overline{BRDY}$.		$\overline{BOFF}$ +1 or HLDA
DP7–DP0	I/O	*(same as D63–D0)*		$\overline{BOFF}$ +1 or HLDA
$\overline{EADS}$	I	Every clock while AHOLD, $\overline{BOFF}$ or HLDA is asserted, beginning two clocks after the assertion of AHOLD, two clocks after the assertion of $\overline{BOFF}$, or one clock after the assertion of HLDA; except while the processor drives A31–A3, while it asserts $\overline{HITM}$, and one clock after $\overline{EADS}$.		
$\overline{EWBE}$	I	With $\overline{BRDY}$ of external write cycles and in every clock thereafter until $\overline{EWBE}$ is asserted.		
$\overline{FERR}$	O	Every clock.		
$\overline{FLUSH}$[1]	I	Every clock. Falling-edge-triggered. Recognized at next instruction boundary. Acknowledged with *Flush-Acknowledge* special bus cycle.		
$\overline{FRCMC}$[1]	I	Every clock in which RESET is asserted.		
$\overline{HIT}$	O	Every clock. Changes state two clocks after $\overline{EADS}$ and retains that state until two clocks after next $\overline{EADS}$.		
$\overline{HITM}$	O	Every clock. Changes state two clocks after $\overline{EADS}$ and retains that state until one clock after the last $\overline{BRDY}$ of writeback.		

Notes:

1. *Can be driven asynchronously or synchronously.*
2. *The term clock means bus clock (CLK). "+n" means n CLKs later.*
3. *"+n" means n CLKs after the named signal is sampled active. All outputs and bidirectionals are floated during the float test ($\overline{FLUSH}$ at RESET).*

TABLE 5-1. Summary of Signal Characteristics (continued)

Signal	Type	Sampled (Input) or Asserted (Output)[2]	Internal Resistor	Floated[3]
HLDA	O	From two clocks after last BRDY of an in-progress bus cycle, or two clocks after HOLD, whichever comes last, until two clocks after HOLD is negated.		
HOLD	I	Every clock. Acknowledged with HLDA.		
IERR	O	Every clock, in the Functional-Redundancy Checking mode.		
IGNNE[1]	I	Every clock.		
INIT[1]	I	Every clock. Rising-edge-triggered. Recognized at next instruction boundary.		
INTR[1]	I	Every clock. Level-sensitive. Recognized at next instruction boundary. Acknowledged with an interrupt acknowledge operation.		
INV	I	Every EADS.		
KEN	I	First BRDY or NA of bus cycle, whichever comes first. Recognized only during read cycles.		
LOCK	O	From ADS until last expected BRDY of the bus cycle. Negated for one clock (dead cycle) between sequential locked operations.		BOFF +1 or HLDA
M/IO	O	From ADS until last expected BRDY of the bus cycle.		BOFF +1 or HLDA
NA	I	From one clock after ADS until the first expected BRDY of a bus cycle. The only function of NA is to validate KEN or WB/WT in place of BRDY.		
NMI[1]	I	Every clock. Rising-edge-triggered. Recognized at next instruction boundary.		
PCD	O	From ADS until last expected BRDY of the bus cycle.		BOFF +1 or HLDA
PCHK	O	Two clocks after every BRDY of read cycles.		
PEN	I	Every BRDY of read cycles, and second BRDY of interrupt acknowledge operation.		

Notes:

1. *Can be driven asynchronously or synchronously.*
2. *The term clock means bus clock (CLK). "+n" means n CLKs later.*
3. *"+n" means n CLKs after the named signal is sampled active. All outputs and bidirectionals are floated during the float test (FLUSH at RESET).*

TABLE 5-1. Summary of Signal Characteristics (continued)

Signal	Type	Sampled (Input) or Asserted (Output)[2]	Internal Resistor	Floated[3]
PRDY	O	Every clock, in response to R/$\overline{S}$. Asserted at instruction boundary after R/$\overline{S}$ is sampled Low. Negated in the clock after R/$\overline{S}$ is sampled High.		
PWT	O	From $\overline{ADS}$ until last expected $\overline{BRDY}$ of the bus cycle.		$\overline{BOFF}$ +1 or HLDA
R/$\overline{S}$[1]	I	Every clock. Level-sensitive. Recognized at next instruction boundary. Acknowledged with PRDY.	pullup	
RESET[1]	I	Every clock. Recognized at next instruction boundary.		
SCYC	O	From $\overline{ADS}$ until last expected $\overline{BRDY}$ of the bus cycle.		$\overline{BOFF}$ +1 or HLDA
$\overline{SMI}$[1]	I	Every clock. Falling-edge-triggered. Recognized at next instruction boundary. Acknowledged with $\overline{SMIACT}$.	pullup	
$\overline{SMIACT}$	O	From one clock after the last expected $\overline{BRDY}$ of the bus cycle, while $\overline{EWBE}$ is asserted, until the return from SMM interrupt handler.		
$\overline{STPCLK}$[1]	I	Every clock. Level-sensitive. Recognized at next instruction boundary. Acknowledged with *Stop Grant* special bus cycle.	pullup	
TCK	I	Always.	pullup	
TDI	I	Every rising TCK edge during the *shift_IR* and *shift_DR* states.	pullup	
TDO	O	Every falling TCK edge during the *shift_IR* and *shift_DR* states.		While not in *shift_IR* or *shift_DR* state.
TMS	I	Every rising TCK edge.	pullup	
$\overline{TRST}$	I	Always sampled asynchronously.	pullup	

Notes:

1. *Can be driven asynchronously or synchronously.*
2. *The term clock means bus clock (CLK). "+n" means n CLKs later.*
3. *"+n" means n CLKs after the named signal is sampled active. All outputs and bidirectionals are floated during the float test ($\overline{FLUSH}$ at RESET).*

TABLE 5-1. Summary of Signal Characteristics (continued)

Signal	Type	Sampled (Input) or Asserted (Output)[2]	Internal Resistor	Floated[3]
W/$\overline{R}$	O	From $\overline{ADS}$ until last expected $\overline{BRDY}$ of the bus cycle.		$\overline{BOFF}$ +1 or HLDA
WB/$\overline{WT}$	I	First $\overline{BRDY}$ or $\overline{NA}$ of bus cycle, whichever comes first.		

Notes:

1. *Can be driven asynchronously or synchronously.*
2. *The term clock means bus clock (CLK). "+n" means n CLKs later.*
3. *"+n" means n CLKs after the named signal is sampled active. All outputs and bidirectionals are floated during the float test ($\overline{FLUSH}$ at RESET).*

5.1.2 Conditions for Driving and Sampling Signals

Table 5-2 shows the processor states, signal states, and bus cycles during which the processor can drive or sample each signal. The table indicates when signals can be driven or sampled so that their state has some practical (meaningful) effect on the state of the processor or on the bus cycle being driven or sampled. In Table 5-2, shading indicates signals that are meaningfully driven or sampled. Signals that are not shaded are not driven or sampled or are not meaningful. For details on how each signal behaves, see Section 5.2 starting on page 5-18.

TABLE 5-2. Conditions for Driving and Sampling Signals

Signal		Conditions under which signals are meaningfully driven or sampled																		
		Bus Cycles or Cache Accesses[38]								Arbitration			States and Modes[8]					Reset, Debug		
		Memory Reads[14]	Memory Writes[14]	Cache Hits[39]	Inquire Cycles[3]	I/O Cycles	Locked Cycles	Special Cycles	Interrupt Acknow.	AHOLD Active	$\overline{\text{BOFF}}$ Active	HLDA Active	Shutdown[33]	Halt	Stop Grant	Stop Clock	$\overline{\text{SMIACT}}$ Active	RESET Active	INIT Active	PRDY Active
Bus Arbitration																				
AHOLD	I						23			—										
$\overline{\text{BOFF}}$	I										—									
BREQ	O			38																
HLDA	O			39							35	—								
HOLD	I										35									
Address and Address Parity																				
$\overline{\text{A20M}}$	I	10	10	10	10		10			10	10	10								10
A31–A3[2]	I/O			44	19			19		7	4	4	3	3	3					
AP	I/O			38						7	4	4	3	3	3					
$\overline{\text{ADS}}$	O			38	37					3			3	3	3					
$\overline{\text{ADSC}}$	O			38	37					3			3	3	3					
$\overline{\text{APCHK}}$	O						7			3	3	3	3	3	3		3			3
$\overline{\text{BE7}}$–$\overline{\text{BE0}}$				38	37					16			3	3	3					
Cycle Definition and Control																				
D/$\overline{\text{C}}$	O			38	37					16			3	3	3					
$\overline{\text{EWBE}}$	I				37			26	26				3	3	3					
$\overline{\text{LOCK}}$	O			38	1		—			16										
M/$\overline{\text{IO}}$	O			38	37					16			3	3	3					
$\overline{\text{NA}}$[18]	I	18	18	18						16							18			
SCYC	O	13	13			13				13							13			
W/$\overline{\text{R}}$	O			38	37					16			3	3	3					

TABLE 5-2. Conditions for Driving and Sampling Signals (continued)

Signal		Conditions under which signals are meaningfully driven or sampled																		
		Bus Cycles or Cache Accesses[38]								Arbitration			States and Modes[8]					Reset, Debug		
		Memory Reads[14]	Memory Writes[14]	Cache Hits[39]	Inquire Cycles[3]	I/O Cycles	Locked Cycles	Special Cycles	Interrupt Acknow.	AHOLD Active	BOFF Active	HLDA Active	Shutdown[33]	Halt	Stop Grant	Stop Clock	SMIACT Active	RESET Active	INIT Active	PRDY Active
Cache Control																				
CACHE	O			38	37	25	25	25	25	16			3	3	3					21
KEN[42]	I									16										21
PCD	O			38						16			3	3	3					21
PWT	O			38						16			3	3	3					15
WB/WT	I			38						16										15
Data and Data Parity																				
BRDY	I			38	37					16			3	3	3					
BRDYC	I			38	37					16			3	3	3					
D63–D0	I/O			38	37					16			3	3	3					
DP7–DP0	I/O			38	37					16			3	3	3					
PCHK[42]	O									16										
PEN[42]	I									16										
Inquire Cycles																				
EADS[7]	I	43	43	43		43	1	43	43											
HIT	O						1													
HITM	O						1													
INV	I	43	43	43		43	1	43	43											
Floating-Point Errors																				
FERR	O																			
IGNNE	I																			

TABLE 5-2. Conditions for Driving and Sampling Signals (continued)

Signal		Conditions under which signals are meaningfully driven or sampled																		
		Bus Cycles or Cache Accesses[38]								Arbitration			States and Modes[8]					Reset, Debug		
		Memory Reads[14]	Memory Writes[14]	Cache Hits[39]	Inquire Cycles[3]	I/O Cycles	Locked Cycles	Special Cycles	Interrupt Acknow.	AHOLD Active	BOFF Active	HLDA Active	Shutdown[33]	Halt	Stop Grant	Stop Clock	SMIACT Active	RESET Active	INIT Active	PRDY Active
External Interrupts, Interrupt Acknowledgments, and Reset																				
BUSCHK[29]	I			38	29					16			3	12	12					
FLUSH[27]	I				41					41	41	41			12					
INIT[27]	I				30					30	30	30			12				—	
INTR[5, 28]	I				40					40	40	40								
NMI[27]	I														12		9			
PRDY	O																			—
R/S[28]	I																			31
RESET	I				30					30	30	30						—		17
SMI[27]	I														12					22
SMIACT	O																—			32
STPCLK[28]	I				34					34	34	34		24						
Test and Debug																				
FRCMC	I																			
IERR	O	20	20	20	20	20	20	20	20	20	20	20	20	20	20		20	20	20	
PRDY	O	See "External Interrupts, Interrupt Acknowledgments, and Reset"																		
R/S	I	See "External Interrupts, Interrupt Acknowledgments, and Reset"																		
TCK	I																			
TDI	I																			
TDO	O																			
TMS	I																			
TRST	I																			
Bus and Processor Clock																				
BF	I															11				
CLK	I															11				

Notes to Table 5-2:

– *Shading indicates signals that are meaningfully driven or sampled. Signals that are not shaded are not driven or sampled or are not meaningful.*

1. *Inquire cycles can be driven while $\overline{LOCK}$ is asserted if AHOLD is used to obtain the bus for the inquire cycle. Inquire cycles never hit locations involved in a locked operation because the processor invalidates such locations, if found in the cache, before doing the locked operation. If the inquire cycle hits a modified location that is different than the one involved in the locked operation, the write-back may be done in the middle of the locked operation, between the two locked cycles, with $\overline{LOCK}$ asserted during the writeback.*
2. *A31–A5 are I/O signals (input for inquire cycles), but A4–A3 are output only.*
3. *Sampled or driven during inquire cycles or resulting writebacks.*
4. *Sampled only during inquire cycles, but not driven for resulting writebacks.*
5. *If enabled by the IF flag in EFLAGS.*
6. *Output only.*
7. *If AHOLD is held asserted throughout an inquire cycle and writeback, system logic must use its latched copy of the inquire cycle address for the writeback. By contrast, if system logic always negates AHOLD before the writeback, the processor will drive the write-back address when it asserts $\overline{ADS}$ for the writeback.*
8. *Signal recognition and assertion applies to the actual state, not to the special cycle driven by the processor prior to entering the state.*
9. *During SMM, NMI is recognized only in response to an IRET instruction. After the return from SMM (RSM instruction), a latched NMI will be serviced.*
10. *$\overline{A20M}$ is recognized only in Real mode, and masking is applied to linear addresses. Because the caches are linearly tagged, assertion of $\overline{A20M}$ during Real mode affects all program-generated cache addresses, including cache-line fills (caused by read misses), cache writethroughs (caused by write misses or write hits to lines in the shared state) and cache accesses that occur while the processor does not control the bus. However, $\overline{A20M}$ does not mask inquire cycle addresses or any writebacks caused by inquire cycles; these addresses are looked up only in the physical tags, which are not masked by $\overline{A20M}$.*
11. *CLK can be driven with a different frequency, and/or BF can be changed when CLK is restarted on exit from the Stop-Clock state.*
12. *Latched or (in the case of $\overline{BUSCHK}$) otherwise sampled and held, pending exit from this state.*
13. *SCYC may be asserted during any misaligned memory or I/O cycle, but it is only meaningful during locked cycles.*
14. *Includes Protected, Virtual-8086 and Real modes, unless otherwise indicated.*
15. *During the Hardware Debug Tool (HDT) mode, this signal is only meaningful for cache write misses (PWT=0 and WB/$\overline{WT}$=1 transition a shared line to an exclusive line). The signal is not meaningful during cache read misses in the HDT mode, because the caches are never filled during the HDT mode.*
16. *Sampled or driven only during the completion of a cycle the processor initiated before the assertion of AHOLD, or for writebacks due to inquire cycles.*
17. *Different than the Pentium processor. The system hardware or software must exit the HDT before asserting RESET.*
18. *$\overline{NA}$ acts as an assertion of $\overline{BRDY}$, but only when sampled with $\overline{KEN}$ or WB/$\overline{WT}$. It is valid only for memory reads and writes, including writethroughs during cache hits to shared or exclusive lines. $\overline{NA}$ has no effect on any signals other than $\overline{KEN}$ and WB/$\overline{WT}$, and addresses are not pipelined when $\overline{NA}$ is asserted.*
19. *If an inquire cycle occurs during a Branch-Trace Message special cycle, the branch address information driven by the processor on A31–A3 can be overwritten by the inquiring bus master. In such cases, external logic should latch A31–A3 when $\overline{ADS}$ is asserted (i.e., before asserting AHOLD, $\overline{BOFF}$ or HOLD).*
20. *Used only to report errors in Functional Redundancy Checking mode and driven only by the Checker.*
21. *This signal is not meaningful during cache read misses in the HDT mode, because the caches are never filled in the HDT mode.*
22. *The debugger can force the processor into SMM, but the processor will not recognize $\overline{SMI}$ until PRDY is negated. If $\overline{SMI}$ is asserted while PRDY is asserted, it is latched and acted upon after PRDY is negated.*
23. *During AHOLD, the system must prevent other bus masters from locking the same address that the AMD5$_K$86 processor is locking.*
24. *Different than the Pentium processor, which ignores $\overline{STPCLK}$ in this state.*
25. *Always negated (non-cacheable).*
26. *$\overline{EWBE}$ is not checked prior to running special bus cycles or interrupt acknowledge operations. All special bus cycles (which have W/$\overline{R}$=1) and interrupt acknowledge operations (which have W/$\overline{R}$=0) serialize the pipeline and do not require $\overline{EWBE}$ for this purpose.*
27. *An edge-triggered interrupt. It is latched when sampled and recognized on an instruction boundary.*
28. *A level-sensitive interrupt. It must be held asserted until recognized, which occurs on an instruction boundary.*
29. *Unlike other level-sensitive interrupts, $\overline{BUSCHK}$ is sampled with every $\overline{BRDY}$ and it does not need to be held asserted after sampling. If $\overline{BUSCHK}$ is asserted during a locked operation or inquire cycle, an enabled machine-check exception will not be acted upon until after the last $\overline{BRDY}$ of the locked operation or after a writeback caused by an inquire cycle.*

30. *The first code fetch after register initialization during INIT or RESET does not occur if AHOLD, $\overline{BOFF}$, or HLDA is asserted.*
31. *PRDY is asserted either when R/$\overline{S}$ goes Low or when the Test Access Port (TAP) instruction, USEHDT, is executed. In the latter case, R/$\overline{S}$ is watched for a Low-to-High transition, which takes the processor out of the Hardware Debug Tool (HDT) mode.*
32. *The processor can go into the Hardware Debug Tool (HDT) mode from within SMM either when R/$\overline{S}$ goes Low or when the TAP instruction, USEHDT, is executed (the instruction causes the processor to assert PRDY). In this case, $\overline{SMIACT}$ can be toggled with HDT commands. $\overline{SMIACT}$ selects main or SMM memory.*
33. *Only NMI, INIT, RESET, and $\overline{SMI}$ gets the processor out of the Shutdown state.*
34. *The processor cannot drive the Stop-Grant special bus cycle.*
35. *HOLD is sampled, but the only practical effect is to assert HLDA.*
36. *Writebacks or writethroughs cannot occur when HLDA is asserted.*
37. *During writebacks.*
38. *During writebacks or writethroughs.*
39. *Including writebacks and writethroughs (except for HLDA).*
40. *The processor cannot drive the interrupt acknowledge cycle, and therefore cannot obtain the interrupt vector.*
41. *If $\overline{FLUSH}$ is asserted while AHOLD, $\overline{BOFF}$, or HLDA is asserted, the outcome of the flush depends on whether the flush causes write-backs of modified lines. If no writebacks are needed, the processor invalidates all lines but does not perform the $\overline{FLUSH}$-acknowledge cycle until the processor gets control of the bus again. If a writeback is needed, the processor stops at that writeback without having invalidated any lines, waits until control of the bus is returned to the processor, then completes the $\overline{FLUSH}$ operation.*
42. *Driven or sampled only during reads.*
43. *Sampled after AHOLD or HLDA is asserted, and while the processor completes an in-progress bus cycle.*
44. *Without $\overline{ADS}$ during cache accesses, with $\overline{ADS}$ during cache writethroughs and writebacks.*

5.1.3 External Interrupts

Interrupts and exceptions are often differentiated in x86 documentation as follows: an *interrupt* is the assertion of a hardware input signal and an *exception* is a software event, such as an invalid opcode or execution of an INT*n* instruction. In some documents, however, the terms *interrupt* and *exception* apply to both hardware and software events, which are then differentiated as *external* or *hardware* interrupts or exceptions, and *internal* or *software* interrupts or exceptions, respectively. In still other x86 documents, the term *software interrupt* means an INT*n* instruction that vectors to an interrupt gate. Moreover, some of the old rules commonly applied to interrupts do not apply to the external interrupts defined for the Pentium processor: for example, not all external interrupts alter the program flow, and not all are acknowledged by the processor.

Because these variations in definition are potentially confusing, this document assumes only the following definitions:

- *Interrupt*—The assertion (or in the case of R/$\overline{S}$, the driving Low) of one of eight hardware input signals ($\overline{BUSCHK}$, R/$\overline{S}$, $\overline{FLUSH}$, $\overline{SMI}$, INIT, NMI, INTR, or $\overline{STPCLK}$).
- *Exception*—Any software-initiated event that accesses an entry in the Real mode interrupt vector table (IVT) or in the Protected mode interrupt descriptor table (IDT).
- *External Interrupt*—Same as *interrupt*.
- *Software Interrupt*—In Real mode, any INT*n* instruction. In Protected mode, any INT*n* instruction that vectors to an IDT entry that is an *interrupt gate*, or that is a *task gate* which references a TSS with the interrupt flag (IF) cleared in its EFLAGS image. (INT*n* instructions that vector to a trap gate are not considered software interrupts because the processor does not clear IF in such cases.)

All interrupts are recognized on the next instruction retirement boundary. Most exceptions are recognized at the point in the instruction where they occur, and are not usually deferred to the end of the instruction. All interrupts and exceptions invalidate (flush) the pipeline when recognized (as defined in Section 2.2.5 on page 2-12). All exceptions are handled precisely so that the instruction causing an exception can be restarted after the exception is serviced.

The processor writes (pushes) its current state onto the stack prior to entering the service routine for exceptions and for $\overline{\text{BUSCHK}}$, $\overline{\text{SMI}}$, NMI, and INTR interrupts. Because of these writes, the state of $\overline{\text{EWBE}}$ affects the processor's response to such interrupts and exceptions. For example, if the processor has initiated a write cycle prior to the next instruction retirement boundary on which such an interrupt would otherwise be recognized, the bus cycle completes but the processor does not respond to the interrupt until it samples $\overline{\text{EWBE}}$ asserted so that it can write to the stack. Also, if the processor has written to the stack once and $\overline{\text{EWBE}}$ is not asserted thereafter, the processor does not write again and its response to an interrupt is halted. A negated $\overline{\text{EWBE}}$ also pauses the processor's response to $\overline{\text{FLUSH}}$ if the flush causes writebacks. However, during interrupts that do not write to memory (R/$\overline{\text{S}}$, $\overline{\text{FLUSH}}$ if there are no writebacks, INIT, and $\overline{\text{STPCLK}}$), the state of $\overline{\text{EWBE}}$ has no affect on the processor's recognition of or response to such interrupts.

The processor performs an interrupt by executing a microcode routine. In this sense, an interrupt acts like the execution of a complex instruction and the microcode routine has a completion boundary that acts like an instruction retirement boundary. In effect, the microcode routine for an interrupt begins executing when the interrupt is recognized on an instruction boundary and it finishes executing when an associated interrupt service routine begins or the hardware aspect of the interrupt function otherwise completes. For example, the $\overline{\text{FLUSH}}$ interrupt completes when all modified cache lines have been written back to memory and all cache lines are invalidated, whereas the R/$\overline{\text{S}}$ interrupt completes when the processor negates PRDY, and the $\overline{\text{STPCLK}}$ interrupt completes when the processor drives the Stop Grant special bus cycle.

The four edge-triggered interrupts ($\overline{\text{FLUSH}}$, $\overline{\text{SMI}}$, INIT, and NMI) are latched on one of the edges of CLK when they are asserted and are recognized later, even if they are negated before being recognized. The four level-sensitive interrupts ($\overline{\text{BUSCHK}}$, R/$\overline{\text{S}}$, INTR, and $\overline{\text{STPCLK}}$) must be held asserted until recognized, except that the $\overline{\text{BUSCHK}}$ interrupt is sampled and latched with every $\overline{\text{BRDY}}$.

The processor disables the recognition of interrupts or exceptions in the following cases:

- *INTR Interrupts*—The processor disables INTR interrupts during all *software interrupts* (that is, INT*n* instructions that vector through interrupt gates or through task gates that reference a TSS with IF cleared in its EFLAGS image). It does this by automatically clearing the IF bit in EFLAGS. If system logic can leave the INTR signal asserted after the INTR service routine is entered, the interrupt vector returned by system logic during the Interrupt acknowledge operation must be for an interrupt gate or for a task gate that references a TSS with IF cleared. (Software may set the IF flag again upon entering the service routine.)
- *NMI Interrupts*—The processor disables NMI interrupts until the IRET of the NMI service routine.
- *Debug Breakpoints*—After a debug breakpoint exception, the debug service routine can disable debug exceptions for one instruction by setting the resume flag (RF) in EFLAGS to 1 to prevent restarted instructions from generating another debug fault.

Table 5-3 shows the characteristics of interrupts and exceptions and the priority with which the processor recognizes them. The term *priority* means two things here:

- *Simultaneous Interrupts*—The order in which a single interrupt or exception is selected for recognition if all occur simultaneously, and
- *Latched Interrupts*—The order in which latched interrupts (any of the four edge-triggered interrupts, $\overline{\text{FLUSH}}$, $\overline{\text{SMI}}$, INIT, or NMI) are recognized when the processor becomes interruptible again after it recognizes a prior interrupt or exception. By contrast, the term *priority* does not mean the order in which *level-sensitive interrupts* ($\overline{\text{BUSCHK}}$, R/$\overline{\text{S}}$, INTR, and $\overline{\text{STPCLK}}$) are nested if one such interrupt occurs while the processor is responding to another interrupt.

Interrupts are themselves interruptible only if they have a software component, such as a service routine. All other interrupts complete their action before the processor recognizes another interrupt. Lower-priority interruptible interrupts can be interrupted by higher-priority interrupts or exceptions at their *point of interruptibility*, as shown in the right-most column of Table 5-3, which is always on an instruction boundary.

TABLE 5-3. Summary of Interrupts and Exceptions

Priority	Description	Type	Sampling[5]	Vector[1]	Acknowledgment	Point of Interruptibility[6]
1	INTn instructions and all other software exceptions	exceptions	*internal*	0-255	*none*	Entry to service routine.
2	$\overline{\text{BUSCHK}}$	interrupt	level-sensitive	18[2]	*none*	Entry to service routine.[2]
3	R/$\overline{\text{S}}$	interrupt	level-sensitive	none	PRDY	Negation of PRDY.
4	$\overline{\text{FLUSH}}$	interrupt	edge-triggered[4]	none	$\overline{\text{FLUSH}}$-Acknowledge special bus cycle	$\overline{\text{BRDY}}$ of $\overline{\text{FLUSH}}$ Acknowledge bus cycle.
5	$\overline{\text{SMI}}$	interrupt	edge-triggered[4]	SMM[3]	$\overline{\text{SMIACT}}$	Entry to SMM service routine.[7]
6	INIT	interrupt	edge-triggered[4]	BIOS	*none*	Completion of initialization.
7	NMI	interrupt	edge-triggered[4]	2	*none*	NMI interrupts: IRET from service routine. All others: Entry to service routine.
8	INTR	interrupt	level-sensitive	0-255	Interrupt acknowledge special bus cycle	Entry to service routine.
9	$\overline{\text{STPCLK}}$	interrupt	level-sensitive	none	Stop-Grant special bus cycle	Negation of $\overline{\text{STPCLK}}$.

Notes:

1. *For interrupts with vectors, the processor saves its state prior to accessing service routine and changing program flow. Interrupts without vectors do not change program flow; instead, they simply pause program flow for the duration of the interrupt function and then return to where they left off.*
2. *If the machine check enable (MCE) bit in CR4 is set to 1.*
3. *The entry point for the $\overline{\text{SMI}}$ interrupt handler is at offset 8000h from the SMM Base Address.*
4. *Only the edge-triggered interrupts are latched when asserted. All interrupts are recognized at the next instruction retirement boundary.*
5. *If a bus cycle is in progress, $\overline{\text{EWBE}}$ must be asserted before the interrupt is recognized.*
6. *For external interrupts (most exceptions, by contrast, are recognized when they occur). External interrupts are recognized at instruction boundaries. MOV or POP instructions that load SS delay interruptibility until after the next instruction, thus allowing both SS and the corresponding SP to load.*
7. *After assertion of $\overline{\text{SMI}}$, subsequent assertions of $\overline{\text{SMI}}$ are masked so as to prevent recursive entry into SMM. Other exceptions or interrupts (except INIT and NMI), however, will intervene in the SMM service routine.*

The processor recognizes $\overline{\text{BOFF}}$, HOLD, and AHOLD while any interrupt signal is asserted, and these signals will intervene with their normal timing in the handling of any interrupt or exception. The interrupt or exception continues from where it left off after the intervening signal is negated. For example, if $\overline{\text{BOFF}}$ is asserted while a $\overline{\text{FLUSH}}$ operation is writing modified

cache lines back to memory, an in-progress writeback will be aborted, but it will be restarted after $\overline{\text{BOFF}}$ is negated, and the $\overline{\text{FLUSH}}$ operation will then continue; any writebacks that completed before $\overline{\text{BOFF}}$ was asserted are not affected.

5.1.4 Bus Signal Compatibility with Pentium Processor

The differences in bus signal functions between the AMD5$_K$86 and Pentium processors are described in Section A.1 on page A-2.

5.2 Signal Descriptions

The following pages describe each signal in detail. The bus cycle protocols that use these signals are described in Section 5.3 on page 5-137. Chapter 6 describes the context in which the SMM and clock-control signals are used, and Chapter 7 does the same for the test signals.

5.2.1 $\overline{\text{A20M}}$ (Address Bit 20 Mask)

Input

Summary Assertion of $\overline{\text{A20M}}$ causes the processor to clear bit 20 of the A31–A3 address bus to 0 prior to accessing the cache or memory in Real mode. The clearing of address bit 20 bit maps addresses above 1 Mbyte to addresses below 1 Mbyte.

Sampled The processor samples $\overline{\text{A20M}}$ in every clock during Real mode. System logic can drive the signal either synchronously or asynchronously (see the data sheet for synchronously driven setup and hold times).

$\overline{\text{A20M}}$ is sampled only in Real mode during memory cycles (including cache writethroughs and writebacks) and locked cycles; or while AHOLD, $\overline{\text{BOFF}}$, HLDA, RESET, INIT, or PRDY is asserted. $\overline{\text{A20M}}$ is not sampled when the processor is operating in Protected mode, Virtual-8086 mode or SMM; during I/O cycles, inquire cycles, special bus cycles, or interrupt acknowledge operations; or while the processor is in the Shutdown, Halt, Stop Grant, or Stop Clock states.

Details The action of clearing A20 so that addresses above 1MB wraparound to addresses below 1 Mbyte simulates the behavior of the 8086 processor, allowing the processor to run software designed for DOS. $\overline{\text{A20M}}$ should only be asserted when the processor runs in Real mode.

$\overline{\text{A20M}}$ should not be asserted during the first code fetch following the RESET or INIT cycles because the masking of bit 20 leads to a fetch from an incorrect address. The BIOS and the operating system alone are responsible for controlling the state of $\overline{\text{A20M}}$. After RESET or INIT, they do this by writing to an external I/O port. (I/O ports 60 and 64h, or port 92h, or register-shadowed versions of those ports are commonly used to control the state of $\overline{\text{A20M}}$.) The instruction pipeline is serialized by virtue of writing to the I/O port, thus allowing time for the $\overline{\text{A20M}}$ signal to assert before the next memory or cache access. Advanced operating systems that do not run under DOS, such as Windows NT™ and OS/2 operating systems, do not use Real mode and never assert $\overline{\text{A20M}}$.

Programs running in Virtual-8086 mode run as tasks under Protected mode. The effect of $\overline{\text{A20M}}$ for these Virtual-8086-mode tasks is normally emulated by the operating system using the

paging mechanism. The operating system writes page table entries so as to map all pages required for the Virtual-8086 mode task to addresses below 1 Mbyte.

Unlike the Pentium processor, the AMD5$_K$86 processor ignores $\overline{\text{A20M}}$ in Protected mode, Virtual-8086 mode, and System Management Mode (SMM). The Pentium processor masks the A20 bit if $\overline{\text{A20M}}$ is asserted in Protected mode or Virtual-8086 mode, even though this behavior is undefined and may change in future processors. The AMD5$_K$86 processor simply ignores $\overline{\text{A20M}}$ except when the processor runs in Real mode.

The AMD5$_K$86 processor applies $\overline{\text{A20M}}$ masking to its linear cache tags, through which all programs access the caches. Thus, assertion of $\overline{\text{A20M}}$ affects all program-generated cache addresses, including cache-line fills (caused by read misses), cache writethroughs (caused by write misses or write hits to lines in the *shared* state), and cache accesses that occur while the processor does not control the bus. However, $\overline{\text{A20M}}$ does not mask writebacks or invalidations caused by internal snoops, inquire cycles, the $\overline{\text{FLUSH}}$ signal, or the WBINVD instruction—such addresses are looked up only in the physical tags, which are not masked by $\overline{\text{A20M}}$. (See Table 2-3 on page 2-20 for details.) By contrast, the Pentium processor applies masking only to physical addresses. This difference of masking linear vs. physical addresses is not visible to software because linear and physical addresses are identical in Real mode.

However, the AMD5$_K$86 processor's $\overline{\text{A20M}}$ linear address masking can affect debug software differently than such masking on the Pentium processor. With $\overline{\text{A20M}}$ asserted, the AMD5$_K$86 processor does breakpoint matching (debug-register comparisons) on masked addresses, whereas the Pentium processor does them on unmasked addresses.

5.2.2 A31–A3 (Address Bus)

A31–A5 Bidirectional, A4–A3 Output

Summary

A31–A3 carries the physical address for the current bus cycle. The processor drives addresses on A31–A3 during memory and I/O cycles, and cycle definition information during special bus cycles. It samples addresses on A31–A5 during inquire cycles.

Driven, Sampled, and Floated

As Outputs: The processor drives A31–A3 from the clock in which $\overline{\text{ADS}}$ is asserted until the last expected $\overline{\text{BRDY}}$ of the bus cycle. The processor also drives A31–A3 without $\overline{\text{ADS}}$ during cache accesses. A31–A3 are driven during memory cycles (including cache writethroughs and writebacks), I/O cycles, inquire cycle writebacks, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM, and while PRDY is asserted. During special bus cycles and interrupt acknowledge operations, the address signals simply support bus cycle definition; they do not provide an address.

The processor floats A31–A3 as outputs, one clock after system logic asserts AHOLD or $\overline{\text{BOFF}}$, and in the same clock that the processor asserts HLDA.

As Inputs: While AHOLD, $\overline{\text{BOFF}}$, or HLDA is asserted, the processor samples A31–A5 in the same clock as $\overline{\text{EADS}}$. A31–A5 are sampled in this way during inquire cycles in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM, including during the Shutdown, Halt, and Stop Grant states, and while PRDY is asserted. The A4–A3 signals are not interpreted as part of the inquire cycle address but must nevertheless be driven at valid 0 or 1 logic levels. The processor may again drive A31–A3 in the next clock after system logic negates AHOLD, $\overline{\text{BOFF}}$, or HOLD.

A31–A3 are never driven or sampled in the Stop Clock state, or while RESET or INIT is asserted.

Details

During processor-initiated bus cycles, the processor drives A31–A3 with $\overline{\text{ADS}}$ to define an eight-byte (quadword) starting address in physical memory or I/O space. System logic interprets these addresses in conjunction with the $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ and cycle definition (D/$\overline{\text{C}}$, M/$\overline{\text{IO}}$, and W/$\overline{\text{R}}$) outputs, and with the $\overline{\text{A20M}}$ input. The processor drives $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ to define the validity of each of the eight bytes accessed by the quadword

addresses on A31–A3. In this manner, $\overline{BE7}$–$\overline{BE0}$ replace the function of address bits A2–A0, which do not exist.

When the processor drives burst reads it drives the starting address on A31–A3 (which is the address of the quadword that contains the instruction or data required) and it drives $\overline{BE7}$–$\overline{BE0}$ to specify the required bytes in that quadword. (This addressing method is unlike the 486 processor, which drives separate addresses for each transfer of a burst.) System logic must determine the remaining three quadword addresses as shown in Table 5-4.

When the processor drives burst writes (writebacks), it drives the starting address on A31–A3 in the same manner as for burst reads, but it enables all eight bytes ($\overline{BE7}$–$\overline{BE0}$ = 00h) because it always starts writebacks at 32-byte aligned addresses (address of the first quadword is xxxx_xx00h). Thus, A4–A3 are always 00b for writebacks.

TABLE 5-4. Address-Generation Sequence During Bursts

Address Driven By Processor on A31–A3	Address of Subsequent Quadwords[1] Generated By System Logic		
Quadword 1	**Quadword 2**	**Quadword 3**	**Quadword 4**
...00h	...08h	...10h	...18h
...08h	...00h	...18h	...10h
...10h	...18h	...00h	...08h
...18h	...10h	...08h	...00h

Notes:
1. *quadword = 8 bytes*

System logic can derive memory and I/O port select signals, as well as memory row and address signals, from A31–A3 and the cycle definition signals. Although the processor does not interpret the A4–A3 signals as part of an inquire cycle address, system logic must drive them at valid logic levels (0 or 1) during inquire cycles, and the processor drives both bits to 0 during writebacks.

While system logic has obtained control of the address bus via assertion of AHOLD, $\overline{BOFF}$ or HOLD, the A31–A5 signals become inputs and define a 32-byte, cache-line, inquire cycle address in conjunction with the following signals:

- The $\overline{\text{EADS}}$ input defines the beginning of the inquire cycle and validates the input address on A31–A5.
- The AP input carries the even parity bit for the A31–A5 address.
- The $\overline{\text{APCHK}}$ output indicates a parity error for the inquire cycle address on A31–A5.

During such system-initiated inquire cycles, A31–A5 defines the starting physical address of a 32-byte cache line that is being snooped in the processor's on-chip instruction and data caches. The processor interprets the addresses using its physical address tags, in conjunction with the $\overline{\text{A20M}}$ input, in parallel with the processor's own cache accesses that use its linear cache tags.

If an inquire cycle hits a modified line in the processor's data cache, the processor performs a writeback. During this writeback, A31–A5 defines a 32-byte starting address in physical memory. This address is identified by the processor's assertion of $\overline{\text{ADS}}$, just as with all other processor-initiated bus cycles, and the address must be interpreted by system logic in conjunction with the $\overline{\text{A20M}}$ input.

The processor does not control the complete bus during a writeback caused by an inquire cycle; in these cases, AHOLD, $\overline{\text{BOFF}}$ or HOLD may still be asserted. However, in addition to writebacks caused by inquire cycle hits, writebacks can also occur while the processor controls the bus (by processor-initiated cache-line replacements, internal snoops for self-modifying code, or execution of the WBINVD instruction) or by system-initiated assertion of the $\overline{\text{FLUSH}}$ signal.

If AHOLD is held asserted throughout an inquire cycle and writeback, system logic must latch the inquire cycle address when it asserts $\overline{\text{EADS}}$. This is required so that, if the inquire cycle hits a modified line ($\overline{\text{HITM}}$ asserted), the processor need not drive the writeback address when it asserts $\overline{\text{ADS}}$ for the writeback, which can occur as early as two clocks after the processor asserts $\overline{\text{HITM}}$. Instead, system logic must use its latched copy of the inquire cycle address for the writeback. By contrast, if system logic always negates AHOLD before the writeback, the processor will drive the writeback address when it asserts $\overline{\text{ADS}}$ for the writeback, and system logic need not retain a copy of the inquire cycle address.

If an inquire cycle occurs while the processor is driving a Branch-Trace Message special bus cycle, the branch address information driven by the processor on A31–A3 can be overwritten by the inquiring bus master. In such cases, system logic should latch A31–A3 when $\overline{\text{ADS}}$ is asserted (that is, before asserting AHOLD, $\overline{\text{BOFF}}$ or HOLD).

At the falling edge of RESET, the states of $\overline{\text{BRDYC}}$ and $\overline{\text{BUSCHK}}$ control the drive strength on A21–A3 (not including A31–A22). The drive strength is *weak* for all states of $\overline{\text{BRDYC}}$ and $\overline{\text{BUSCHK}}$ except $\overline{\text{BRDYC}}$ and $\overline{\text{BUSCHK}}$ both Low (0), in which case the drive strength is strong. The A31–A22 signals use the weak drive strength at all times. See the data sheet for details.

Unlike the Pentium processor, pipelined address-data transactions are not supported by the AMD5$_K$86 processor. Thus, the $\overline{\text{NA}}$ input has no effect on the processor's address bus. $\overline{\text{NA}}$ only affects the sampling time for the $\overline{\text{KEN}}$ and WB/$\overline{\text{WT}}$ inputs.

5.2.3 $\overline{\text{ADS}}$ (Address Strobe)

Output

Summary

The processor asserts $\overline{\text{ADS}}$ to specify the beginning of a memory or I/O bus cycle, or a cache writeback to memory. The signal validates the processor's address and cycle definition signals and it can be used by system logic to enable accesses to memory and I/O.

Driven and Floated

During processor-initiated bus cycles, the processor asserts $\overline{\text{ADS}}$ for one clock at the beginning of each bus cycle. During writeback cycles, whether initiated by the processor or by system logic, the processor asserts $\overline{\text{ADS}}$ for one clock as early as two clocks after the processor asserts $\overline{\text{HITM}}$. The processor can assert $\overline{\text{ADS}}$ as early as two clocks after the assertion of $\overline{\text{BRDY}}$ (thus allowing one idle or dead clock between any two bus cycles), and one clock after the negation of AHOLD, $\overline{\text{BOFF}}$, or HLDA.

$\overline{\text{ADS}}$ is driven during memory cycles (including cache writethroughs and writebacks), I/O cycles, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM, or while PRDY is asserted. While AHOLD is asserted, and during the Shutdown, Halt, and Stop Grant states, $\overline{\text{ADS}}$ is driven only for writebacks that result from inquire cycle hits. $\overline{\text{ADS}}$ is not driven during the Stop Clock state, or while $\overline{\text{BOFF}}$, HLDA, RESET, or INIT is asserted.

The processor floats $\overline{\text{ADS}}$ one clock after system logic asserts $\overline{\text{BOFF}}$ and in the same clock that the processor asserts HLDA.

Details

The processor initiates bus cycles for the purpose of reading and writing memory or I/O, and for writebacks of modified cache lines. While the processor controls the bus, or while it is writing back a modified cache line (whether in control of the bus or not), $\overline{\text{ADS}}$ defines the beginning of the cycle. In the clock that it asserts $\overline{\text{ADS}}$, the processor also begins driving the several signals that define and qualify the bus cycle, including A31–A3 (or A31–A5 for writebacks), AP, the cycle definition signals (D/$\overline{\text{C}}$, M/$\overline{\text{IO}}$ and W/$\overline{\text{R}}$), $\overline{\text{BE7}}$–$\overline{\text{BE0}}$, BREQ, $\overline{\text{A20M}}$, $\overline{\text{CACHE}}$, $\overline{\text{LOCK}}$, PCD, PWT and SCYC.

If $\overline{\text{ADS}}$ initiates a cache line fill and all four ways of the cache that could accommodate the incoming line are filled with valid

entries, the processor uses a pseudo-random algorithm to select a line for replacement. If the selected line is cached in the *modified* state, it must be written back to memory. In this case, the order of events is:

1. Complete the burst read, placing the incoming cache line in the processor's line fill buffer.
2. Write the modified line back to memory.
3. Fill the vacated cache line with the contents of the line buffer.

Processor-initiated writebacks can occur during cache line replacement, internal snoops for self-modifying code, and execution of the WBINVD instruction. System-initiated writebacks can occur during inquire cycle hits to modified cache lines (while AHOLD, $\overline{\text{BOFF}}$ or HLDA is asserted) or by assertion of the $\overline{\text{FLUSH}}$ input. The processor drives writebacks by asserting $\overline{\text{ADS}}$ and either reusing the inquire cycle address (if AHOLD is held asserted throughout the writeback) or driving the address itself (if AHOLD is negated for the writeback, or if $\overline{\text{BOFF}}$ or HOLD was used to obtain the bus).

During an inquire cycle that hits a modified cache line, the processor asserts $\overline{\text{ADS}}$ as soon as two clocks after asserting $\overline{\text{HITM}}$, regardless of whether AHOLD is asserted or negated. By contrast, if $\overline{\text{BOFF}}$ or HLDA is asserted instead of AHOLD during an inquire hit, the processor postpones the writeback until after $\overline{\text{BOFF}}$ or HLDA is negated.

During special bus cycles and interrupt acknowledge operations, the processor drives $\overline{\text{ADS}}$ to validate A31–A3, $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ and the cycle definition signals. This use of $\overline{\text{ADS}}$ and A31–A3 simply serves to identify the type of special bus cycle, rather than to address a location in memory or I/O space.

The processor asserts BREQ in the same clock that it asserts $\overline{\text{ADS}}$, although BREQ is also asserted at other times (see the description of BREQ on page 5-46). The processor negates $\overline{\text{ADS}}$ for one clock between any contiguous bus operations, such as between a single-transfer I/O write and a burst read from memory, or between two burst reads. The same is true for contiguous sequences of locked operations (sequences of locked bus cycle pairs). System logic can use the negation of $\overline{\text{ADS}}$ between contiguous bus operations to make the bus available to other

bus masters, thus intervening temporarily in the processor's sequential operations.

If $\overline{\text{BOFF}}$ is asserted while $\overline{\text{ADS}}$ is asserted, $\overline{\text{ADS}}$ remains Low (floats asserted). System logic must consider this when interpreting the state of $\overline{\text{ADS}}$ after negating $\overline{\text{BOFF}}$. In the next clock after $\overline{\text{BOFF}}$ is negated, the processor may reassert $\overline{\text{ADS}}$ to restart a cycle if a cycle was aborted by the assertion of $\overline{\text{BOFF}}$.

If system logic begins driving an inquire cycle by asserting AHOLD or $\overline{\text{BOFF}}$ and then asserting $\overline{\text{EADS}}$ with the inquire address, and the processor is driving a Branch-Trace Message special bus cycle at the same time that AHOLD or $\overline{\text{BOFF}}$ is asserted, the branch address information driven by the processor on A31–A3 can be overwritten by the inquiring bus master. In such cases, system logic should latch A31–A3 when $\overline{\text{ADS}}$ is asserted, before asserting AHOLD or $\overline{\text{BOFF}}$.

At the falling edge of RESET, the states of $\overline{\text{BRDYC}}$ and $\overline{\text{BUSCHK}}$ control the drive strength on the A21–A3 (not including A31–A22), ADS, HITM, and W/R signals. The drive strength is weak for all states of $\overline{\text{BRDYC}}$ and $\overline{\text{BUSCHK}}$ except $\overline{\text{BRDYC}}$ and $\overline{\text{BUSCHK}}$ both Low (0), in which case the drive strength is strong. The A31–A22 signals use the weak drive strength at all times. See the data sheet for details.

5.2.4 $\overline{\text{ADSC}}$ (Address Strobe Copy)

Output

Summary $\overline{\text{ADSC}}$ is an identical copy of $\overline{\text{ADS}}$. In systems that would otherwise place large capacitive loads on $\overline{\text{ADS}}$, the $\overline{\text{ADSC}}$ output can be used instead of $\overline{\text{ADS}}$ to distribute loads, thereby increasing response time.

Driven and Floated $\overline{\text{ADSC}}$ is driven and floated with the same timing as $\overline{\text{ADS}}$. See the description of $\overline{\text{ADS}}$ on page 5-25.

Details See the description of $\overline{\text{ADS}}$ on page 5-25.

5.2.5 AHOLD (Address Hold)

Input

Summary System logic can assert AHOLD to obtain control of the bidirectional A31–A3 address bus and AP address parity signal to drive one or more inquire cycles to the processor.

Sampled The processor samples AHOLD in every clock and responds by floating the bidirectional A31–A3 and AP signals one clock after AHOLD is asserted.

AHOLD is sampled during memory cycles (including cache writethroughs and writebacks), I/O cycles, inquire cycles, locked cycles, writebacks, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM; in the Shutdown, Halt, or Stop Grant states; or while RESET, INIT or PRDY is asserted. AHOLD is sampled but not effective when $\overline{\text{BOFF}}$ or HLDA is asserted. AHOLD is not sampled during the Stop Clock state.

Details The sole function of AHOLD is to support inquire cycles. There are three methods by which system logic can obtain control of the address bus to drive an inquire cycle: AHOLD, $\overline{\text{BOFF}}$, or HOLD. AHOLD obtains control only of the address bus and allows another master or system logic to drive only inquire cycles, whereas $\overline{\text{BOFF}}$ and HOLD obtain control of the full bus (address and data), allowing another master to drive not only inquire cycles but also read and write cycles. AHOLD and HOLD both permit an in-progress bus cycle to complete, but a writeback can occur while AHOLD is asserted, whereas a pending writeback during the assertion of $\overline{\text{BOFF}}$ or HOLD occurs after the $\overline{\text{BOFF}}$ or HOLD is negated.

AHOLD is useful primarily in systems with multiple buses and multiple bus masters, where operations can occur on the separate buses independently and in parallel. This configuration occurs, for example, if the processor shares a bus only with a look-through L2 cache, and other caching masters work in parallel on another bus that is isolated from the processor by system logic. In such designs, system logic may drive separate AHOLD signals to each bus master in the system. For details on how AHOLD can be driven in such configurations, see Section 6.2.5 on page 6-14.

When the processor releases control of A31–A3 and AP in response to AHOLD, the processor still maintains control of the remaining signals on the bus so that it can (a) finish driving a bus cycle it may have begun before AHOLD was asserted, and (b) drive a writeback if an inquire cycle hits a modified line in the processor's data cache. However, the processor cannot begin driving a new bus cycle while AHOLD is asserted because system logic controls the address bus.

System logic drives inquire cycles with the $\overline{\text{EADS}}$, A31–A5, AP and INV inputs. A typical sequence for an inquire cycle is: assert AHOLD; two clocks later, assert $\overline{\text{EADS}}$ and drive A31–A5 and INV; wait two clocks for the processor to assert $\overline{\text{HITM}}$ and/or $\overline{\text{HIT}}$. If $\overline{\text{HITM}}$ remains negated two clocks after $\overline{\text{EADS}}$ is asserted, the inquire cycle ends. If $\overline{\text{HITM}}$ is asserted at that time, the processor begins driving a four-transfer burst writeback as early as two clocks after asserting $\overline{\text{HITM}}$.

AHOLD can be negated as early as one clock after $\overline{\text{EADS}}$ is asserted. If system logic holds AHOLD asserted throughout an inquire cycle and any required writeback, system logic must latch the inquire cycle address when it asserts $\overline{\text{EADS}}$. This is required so that, if the inquire cycle hits a modified line ($\overline{\text{HITM}}$ asserted), the address used for the writeback need not be driven by the processor when the processor asserts $\overline{\text{ADS}}$ for the writeback. Instead, A31–A5 remains an input-only bus and system logic must use its latched copy of the inquire cycle address. By contrast, if system logic always negates AHOLD before the writeback, the processor drives the writeback address when it asserts $\overline{\text{ADS}}$ for the writeback, and system logic need not retain a copy of the inquire cycle address. While the processor drives the writeback address, it drives only the beginning address for the 32-byte transfer on A31–A5. System logic must determine the remaining addresses as shown in Table 5-4 on page 5-22.

If system logic asserts AHOLD while the processor is driving a locked cycle, the system must not allow accesses by other bus masters to lock the same address that the processor is locking.

While AHOLD is asserted (after the completion of any in-progress bus cycle by the processor), the processor continues to execute out of its instruction and data caches, if possible. If the processor can no longer operate out of its caches, it holds

BREQ asserted continuously. For a list of signals recognized while AHOLD is asserted, see Table 5-2 on page 5-9.

The processor may again drive its own cycles with $\overline{\text{ADS}}$ as early as one clock after system logic negates AHOLD. Before negating AHOLD, however, system logic may need to arbitrate among potential contenders for the address bus so as to avoid deadlock contention for the bus.

Ground-bounce spikes can be avoided by following two rules with respect to AHOLD:

- Do not negate AHOLD in the same clock that $\overline{\text{BRDY}}$ is asserted during a write cycle.
- Do not negate AHOLD in the same clock that $\overline{\text{ADS}}$ is asserted during a writeback.

These restrictions must be observed because the processor's 32 address drivers turn on almost immediately after AHOLD is negated. If the processor is driving data with $\overline{\text{BRDY}}$ on the 64-bit data bus at the same time, the processor then drives 96 bits simultaneously and ground-bounce spikes can occur.

5.2.6 AP (Address Parity)

Bidirectional

Summary AP carries the even parity bit for cache line addresses driven and sampled on A31–A5. The processor drives AP when it drives an address for a read or write cycle. The processor samples AP during inquire cycles in order to drive the $\overline{\text{APCHK}}$ output.

Driven, Sampled, and Floated AP is driven, sampled, and floated with the same timing as A31–A3. See the description of A31–A3 on page 5-21.

Details The bit value driven on AP is counted with the bit values driven on A31–A5 to determine address parity. If the total number of 1 bits is even on AP and A31–A5, the address is considered free of error (thus the term *even parity*). If the total number of 1 bits is odd, the address is considered to have an error. The bit values driven on A4–A3 are not counted during the parity checking.

In addition to generating and checking address parity, the processor also generates and checks data parity using the DP7–DP0 and $\overline{\text{PCHK}}$ signals. See page 5-58 and 5-102 for details. Unlike the handling of $\overline{\text{PCHK}}$, however, the processor does not capture the faulty address in a register when it asserts $\overline{\text{APCHK}}$. System logic must handle the error externally. Typical PC systems assert an interrupt signal such as NMI after a parity error is detected.

Systems that do not implement address parity generation and checking should tie AP either High or Low and ignore the $\overline{\text{APCHK}}$ output.

5.2.7 APCHK (Address Parity Check)

Output

Summary The processor asserts APCHK if an even-parity error occurs on A31–A5 during an inquire cycle.

Driven The processor drives APCHK for one clock, two clocks after system logic asserts EADS with an inquire address.

APCHK is driven under the same conditions in which EADS is sampled: See the description of EADS on page 5-59.

Details System logic can use APCHK to initiate a remedy for the error. Typical PC systems assert an interrupt such as NMI if a parity error is detected.

See the description of parity error determination for the AP input on page 5-32. Systems that do not implement address parity checking should ignore APCHK.

5.2.8 $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ (Byte Enables)

Output

Summary

The eight bits of $\overline{\text{BE7}}$–$\overline{\text{BE0}}$, when cleared to 0, validate the eight bytes driven on D63–D0. In this way, $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ expands on the function of address bits A2–A0, which do not exist on the A31–A3 address bus. $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ also help differentiate the special bus cycles.

Driven and Floated

The processor drives $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ from the clock in which $\overline{\text{ADS}}$ is asserted until the last expected $\overline{\text{BRDY}}$ of the bus cycle. The processor floats $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ one clock after system logic asserts $\overline{\text{BOFF}}$ and in the same clock that the processor asserts HLDA.

$\overline{\text{BE7}}$–$\overline{\text{BE0}}$ is driven with the address and cycle definition outputs (D/$\overline{\text{C}}$, M/$\overline{\text{IO}}$ and W/$\overline{\text{R}}$) during memory cycles (including cache writethroughs and writebacks), I/O cycles, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM, or while PRDY is asserted. While AHOLD is asserted, $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ is driven only to complete a bus cycle that had been initiated before AHOLD was asserted, or for inquire cycle writebacks. During the Shutdown, Halt, and Stop Grant states, $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ is driven only for inquire cycle writebacks. $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ is not driven during the Stop Clock state, or while $\overline{\text{BOFF}}$, HLDA, RESET, or INIT is asserted.

Details

Table 5-5 shows the relationship between $\overline{\text{BE7}}$–$\overline{\text{BE0}}$, D63–D0, DP7–DP0, and the effective relationship with A2–A0, the non-existent low address bits. The $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ signals expand on the function of A2–A0; $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ allow the processor to address any or all eight bytes indicated by A31–A3, whereas A2–A0, if they existed, would only address one of eight bytes.

During single-transfer memory cycles and all I/O cycles, the processor drives $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ to identify all of the bytes desired for the transfer. System logic must return valid data in those byte lanes of D63–D0.

During burst reads ($\overline{\text{CACHE}}$ and $\overline{\text{KEN}}$ both asserted with the first $\overline{\text{BRDY}}$ of a memory read), the processor drives $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ with $\overline{\text{ADS}}$ to identify the bytes of the desired instruction or operand. The processor drives $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ with the desired bytes at that time because it does not yet know whether the read will be a single-transfer or a burst—this depends on how system

logic drives KEN with the first BRDY. If system logic negates KEN, it must return as a single transfer only the bytes specified on BE7–BE0. If system logic asserts KEN, it must ignore BE7–BE0 during all transfers of the burst and return all eight bytes for the starting address on A31–A3. BE7–BE0 does not change during the four transfers of the burst. (This behavior is unlike the 486 processor, which drives BE3–BE0 separately for each transfer of a burst.) System logic must determine the successive quadword addresses for each transfer in a burst, depending on the starting address, as shown in Section 5-4 on page 5-22.

During single writes, which include cache writethroughs (1-to-8-byte transfers with CACHE negated) the processor drives the bits of BE7–BE0 to indicate which of the eight bytes on D63–D0 are valid. During writebacks (32-byte, four-transfer bursts with CACHE asserted) the processor drives all bits of BE7–BE0 Low to indicate that all eight bytes on D63–D0 are valid. Writebacks are addressed by A31–A3 but they are always aligned to 32-byte boundaries, so A4–A3 are always 0.

TABLE 5-5. Relation Of BE7–BE0 To Other Signals

Byte Enable Output	Effective Address Bits[1]			Byte On Data Bus	Data Parity Bit
	A2	A1	A0		
BE7	1	1	1	D63–D56	D7
BE6	1	1	0	D55–D48	D6
BE5	1	0	1	D47–D40	D5
BE4	1	0	0	D39–D32	D4
BE3	0	1	1	D31–D24	D3
BE2	0	1	0	D23–D16	D2
BE1	0	0	1	D15–D8	D1
BE0	0	0	0	D7–D0	D0

Notes:

1. *BE7–BE0 expand on the function of A2–A0 by allowing the processor to address any or all eight bytes addressed by A31–A3.*

The processor differentiates special bus cycles using a combination of BE7–BE0, the cycle definition (D/C, M/IO, and W/R) outputs, and A31–A3. The values on the cycle definition signals are the same for all special cycles; only BE7–BE0 and A31–A3 differentiate among those cycles. Table 5-6 shows the relation-

ships. This function of $\overline{BE7}$–$\overline{BE0}$ bears no relationship to the D63–D0 data bus. This is particularly apparent in the case of the Branch-Trace Message special bus cycle, during which the value of $\overline{BE7}$–$\overline{BE0}$ is DFh (1101_1111b) but, in contradiction to the byte-enable bits, the four bytes on D31–D0 carry valid data during both cycles of the operation: during the first cycle, D31–D0 carries the EIP value of the source (branch) instruction; during the second cycle, D31–D0 carries the EIP value of the branch-target instruction.

TABLE 5-6. Encodings For Special Bus Cycles

$\overline{BE7}$–$\overline{BE0}$	A31–A3	Special Bus Cycle[1]	Cause
FEh	...00h	Shutdown	Triple fault
FDh	...00h	Cache Invalidation	INVD instruction
FBh	...10h	Stop Grant	$\overline{STPCLK}$
FBh	...00h	Halt	HLT instruction
F7h	...00h	Cache Writeback and Invalidation	WBINVD instruction
EFh	...00h	$\overline{FLUSH}$ Acknowledge	$\overline{FLUSH}$
DFh	...00h	Branch-Trace Message[2]	Bit 5 = 1 and bits 3–1 = 001 in the Hardware Configuration Register (HWCR). See Section 7.1 on page 7-3 for details.

Notes:

1. *For all special bus cycles, D/$\overline{C}$ = 0, M/$\overline{IO}$ = 0 and W/$\overline{R}$ = 1. System logic must return $\overline{BRDY}$ in response to this cycle.*
2. *The message in a branch-trace message special bus cycle is different in the AMD5$_K$86 and Pentium processors.*

Certain models of the Pentium processor implement $\overline{BE7}$–$\overline{BE5}$ as outputs and $\overline{BE4}$–$\overline{BE0}$ as bidirectional signals. On the AMD5$_K$86 processor, however, all eight $\overline{BE7}$–$\overline{BE0}$ signals are outputs only.

5.2.9 BF (Bus Frequency)

Input

Summary During RESET, BF selects between a high and low multiplication factor for the frequency ratio between the processor's internal clock and the bus clock (CLK).

Sampled The processor samples BF only on the falling edge of RESET. The signal assertion must be stable 10 clocks prior to its sampling. BF has a weak internal pullup resistor; see the data sheet for details.

Details Table 5-7 shows the ratios between the processor clock and the bus clock (CLK) for the High and Low values of BF. BF may be tied High or Low. Due to the internal pullup resistor, the lower ratio is selected if BF is left unconnected.

TABLE 5-7. Processor-to-Bus Clock Ratios

State of BF Input	Processor-Clock to Bus-Clock Ratio
BF = 1	1.5x
BF = 0	2.0x

Notes:

1. *The default processor-to-clock ratios are shown in Table 5-7. Specific models of the AMD5$_K$86 processor may implement different ratios for the High and Low values of BF. For authorative information, see the data sheet for each AMD5$_K$86 processor model.*

5.2.10 $\overline{\text{BOFF}}$ (Backoff)

Input

Summary When system logic asserts $\overline{\text{BOFF}}$, the processor floats the bus and continues to float it until $\overline{\text{BOFF}}$ is negated. If the processor is driving a bus cycle when $\overline{\text{BOFF}}$ is asserted, the cycle is aborted and restarted after $\overline{\text{BOFF}}$ is negated. The processor does not acknowledge $\overline{\text{BOFF}}$. While $\overline{\text{BOFF}}$ is asserted, another bus master can drive cycles on the bus, including inquire cycles to the processor.

Sampled The processor samples $\overline{\text{BOFF}}$ in every clock. When $\overline{\text{BOFF}}$ is asserted, the processor floats the cycle-driving outputs on the bus in the next clock and continues to float them until $\overline{\text{BOFF}}$ is negated.

$\overline{\text{BOFF}}$ is sampled during memory cycles (including cache writethroughs and writebacks), I/O cycles, inquire cycles, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM; in the Shutdown, Halt, or Stop Grant states; or while AHOLD, RESET, INIT, or PRDY is asserted. $\overline{\text{BOFF}}$ is sampled but not effective when HLDA is asserted. $\overline{\text{BOFF}}$ is not sampled during the Stop Clock state.

Details The assertion of $\overline{\text{BOFF}}$, like HOLD but unlike AHOLD, forces the processor to relinquish the full address and data bus to another bus master. The signal can be used for the following purposes:

- *Bus Turnaround*—Another bus master can assert $\overline{\text{BOFF}}$ to the processor to obtain control of the bus, allowing the other bus master to drive any type of bus cycles.
- *Inquire Cycles*—In multi-master systems with shared memory, another bus master typically drives an inquire cycle to the processor or its L2 cache prior to driving a read or write cycle to any memory locations shared by both masters. Such inquire cycles can be driven while $\overline{\text{BOFF}}$ is asserted.
- *Deadlock Resolution*—When an inquire cycle by one master hits a *modified* cache line in another processor, neither master can proceed until the target of the inquire cycle gets the bus. In such a case, system logic would back the inquiring master off the bus by asserting $\overline{\text{BOFF}}$ to it, so that the master with the *modified* line can write it back to memory.

$\overline{\text{BOFF}}$ provides the fastest response of the three bus-hold inputs. Because of its ability to help resolve deadlock problems, it is required in almost all systems with multiple-caching masters. In such designs, system logic typically drives separate $\overline{\text{BOFF}}$ signals to each bus master in the system. See Section 6.2.5 on page 6-14 for system configurations using $\overline{\text{BOFF}}$.

Unlike AHOLD and HOLD, $\overline{\text{BOFF}}$ does not permit an in-progress bus cycle to complete. It forces the processor off the bus in the next clock, aborting any in-progress bus cycle that the processor has begun. A writeback can occur while AHOLD is asserted, but a pending writeback during the assertion of $\overline{\text{BOFF}}$ or HOLD waits until after $\overline{\text{BOFF}}$ or HOLD is negated.

The processor floats the bus one clock after the assertion of $\overline{\text{BOFF}}$. All output and bidirectional signals used for memory or I/O accesses are floated. Table 5-8 shows the signals floated. The same set of signals is floated with HLDA.

TABLE 5-8. Outputs Floated When $\overline{\text{BOFF}}$ is Asserted

Address and Address Parity	Cycle Definition and Control	Data and Data Parity	Cache Control
A31–A3	D/$\overline{\text{C}}$	D63–D0	$\overline{\text{CACHE}}$
$\overline{\text{ADS}}$	$\overline{\text{LOCK}}$	DP7–DP0	PCD
$\overline{\text{ADSC}}$	M/$\overline{\text{IO}}$	N/A	PWT
AP	SCYC	N/A	N/A
$\overline{\text{BE7}}$–$\overline{\text{BE0}}$	W/$\overline{\text{R}}$	N/A	N/A

The processor supports only one in-progress bus cycle, no pending bus cycles are buffered. If the processor is driving a bus cycle when $\overline{\text{BOFF}}$ is asserted the processor retains the data that had been transferred up to the clock in which $\overline{\text{BOFF}}$ was asserted but ignores the data transferred with or after $\overline{\text{BOFF}}$ was asserted. $\overline{\text{BOFF}}$ has no effect on writes to the processor store buffer, except to delay them. (The store buffer is situated between the execution units and the data cache. It is used for speculative stores prior to being written to the data cache.)

The bus master asserting or causing the assertion of $\overline{\text{BOFF}}$ must wait two clocks after asserting $\overline{\text{BOFF}}$ before driving its first bus cycle because the processor does not float its outputs until one clock after the assertion of $\overline{\text{BOFF}}$. System logic or

another bus master may continue asserting $\overline{\text{BOFF}}$ for as long as it wants. The processor has no way of breaking the hold. While the processor is backed off, it continues to execute out of its instruction and data caches, if possible. If it can no longer operate out of its caches, it holds BREQ asserted continuously.

As early as one clock after $\overline{\text{BOFF}}$ is negated, the processor restarts—from the beginning—any bus cycle that was aborted when $\overline{\text{BOFF}}$ was asserted. This is unlike $\overline{\text{BOFF}}$ on the 486 processor, which restarts only the transfers that did not complete when $\overline{\text{BOFF}}$ was asserted. The processor can drive another cycle with $\overline{\text{ADS}}$ as early as two clocks after any aborted cycle completes. This allows one idle clock (also called a dead clock) between any two bus cycles. If $\overline{\text{BOFF}}$ was asserted when $\overline{\text{ADS}}$ was also asserted, however, $\overline{\text{ADS}}$ remains Low (floats asserted) after $\overline{\text{BOFF}}$ is negated. In such a case, system logic must properly interpret the state of $\overline{\text{ADS}}$ when it negates $\overline{\text{BOFF}}$.

If $\overline{\text{BOFF}}$ is asserted during a locked operation, only the cycle(s) aborted before their last $\overline{\text{BRDY}}$ and the cycles not yet run are restarted after $\overline{\text{BOFF}}$ is negated. Thus, system logic must keep track of all cycles in the locked operation that have completed before the assertion of $\overline{\text{BOFF}}$ and must continue the locked operation immediately after $\overline{\text{BOFF}}$ is negated, except that if a writeback is pending when $\overline{\text{BOFF}}$ is negated, the writeback takes precedence over the restarting of the aborted cycles in the locked operation.

The processor responds to inquire cycles while $\overline{\text{BOFF}}$ is asserted and drives $\overline{\text{HIT}}$ and $\overline{\text{HITM}}$ in response to such cycles. During the $\overline{\text{BOFF}}$-initiated inquire cycles, $\overline{\text{BOFF}}$ can be negated as early as one clock after $\overline{\text{EADS}}$ is asserted. If $\overline{\text{HITM}}$ is asserted, which would occur two clocks after $\overline{\text{EADS}}$ is asserted, the writeback is performed after $\overline{\text{BOFF}}$ is negated. If a processor cycle was aborted by the assertion of $\overline{\text{BOFF}}$, that cycle is restarted as soon as $\overline{\text{BOFF}}$ is negated, except that if an inquire cycle hits a *modified* line while $\overline{\text{BOFF}}$ was asserted, the writeback is driven first when $\overline{\text{BOFF}}$ is negated, before an aborted cycle is restarted. Multiple inquire cycles are not permitted to hit modified lines. The processor implements this restriction by ignoring $\overline{\text{EADS}}$ while $\overline{\text{HITM}}$ is asserted; when $\overline{\text{HITM}}$ is asserted, it is held asserted until the last $\overline{\text{BRDY}}$ of the writeback.

If $\overline{\text{BOFF}}$ is asserted when $\overline{\text{BUSCHK}}$ is asserted, $\overline{\text{BOFF}}$ is recognized and $\overline{\text{BUSCHK}}$ is ignored. For a list of signals recognized while $\overline{\text{BOFF}}$ is asserted, see Table 5-2 on page 5-9.

5.2.11 BRDY (Burst Ready)

Input

Summary

For bus cycles that transfer data, system logic must assert BRDY to indicate that it has received a data transfer on D63–D0 during a write and to indicate that it has placed valid data on D63–D0 during a read. Up to eight bytes of data—the width of the D63–D0 data bus—are validated with each BRDY. For special bus cycles, system logic must assert BRDY either to validate data or as a simple handshake.

Sampled

The processor samples BRDY every clock, from one clock after ADS until the last expected BRDY of the bus cycle.

BRDY is sampled during memory cycles (including cache writethroughs and writebacks), I/O cycles, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM, or while PRDY is asserted. While AHOLD is asserted, BRDY is sampled only to complete a bus cycle that had been initiated before AHOLD was asserted, or for inquire cycle writebacks. During the Shutdown, Halt, and Stop Grant states, BRDY is sampled only for inquire cycle writebacks. BRDY is not sampled when the processor is not driving an external bus cycle; or during the Stop Clock state; or while BOFF, HLDA, RESET, or INIT is asserted.

If BRDY is asserted simultaneously with BOFF, BOFF is recognized and BRDY is not, but if BRDY is asserted simultaneously with HOLD, BRDY is recognized and the HOLD waits until the bus cycle associated with the BRDY completes.

Details

BRDY is associated with a *transfer* of one to eight bytes on the D63–D0 data bus. During memory and I/O reads, the processor samples and latches the bytes on D63–D0 and the parity bits on DP7–DP0 that are enabled by BE7–BE0 when system logic asserts BRDY. During memory and I/O writes, the processor waits for system logic to return BRDY before transferring more data on D63–D0 or before starting another bus cycle. Delays in returning the BRDY for a transfer (and delays in returning EWBE for a write cycle) are said to add *wait states* to the transfer, although these states are nothing more than the absence of an expected BRDY.

The processor samples BRDY during all types of bus cycles, including the following:

- Single-transfer reads
- Single-transfer writes (including cache writethroughs)
- Burst reads (cache line fills)
- Burst writebacks
- Special bus cycles
- Interrupt acknowledge cycles

The number of BRDYs expected by the processor depends on the type of bus cycle, as follows:

- One BRDY for an aligned single-transfer cycle, a special bus cycle, or each of two cycles in an interrupt acknowledge operation. Additional BRDYs are needed for misaligned cycles.
- Four BRDYs, one for each data transfer in a burst cycle. BRDY may be held asserted throughout the four transfers of the burst.

All data transfers that are not performed as bursts are performed as one or more single-transfer cycles. For write cycles, EWBE must be asserted either with or after BRDY in order for any further writes or certain other operations to be performed (see the description of EWBE on page 5-63). If system logic returns more BRDYs than the processor expects for a single-transfer cycle or a burst cycle, the processor ignores them.

The processor samples the following inputs in the clock in which system logic asserts BRDY:

- D63–D0—Every BRDY, for all bus cycles.
- DP7–DP0—Every BRDY, for all bus cycles.
- BUSCHK—Every BRDY, for all bus cycles.
- EWBE—Every BRDY, for write cycles.
- KEN—First BRDY or NA, whichever occurs first, for read cycles.
- PEN—Every BRDY for read cycles, and second BRDY of interrupt acknowledge operations.
- WB/WT—First BRDY or NA, whichever occurs first, for read and write cycles.

The assertion of $\overline{\text{NA}}$ acts as an assertion of $\overline{\text{BRDY}}$ only when the processor samples $\overline{\text{KEN}}$ or WB/$\overline{\text{WT}}$.

The processor drives or asserts the following outputs relative to the assertion of $\overline{\text{BRDY}}$:

- D63–D0—For single-transfer write cycles, the processor drives data from one clock after $\overline{\text{ADS}}$ until $\overline{\text{BRDY}}$ is returned. For burst transfers, the processor drives data from one clock after $\overline{\text{ADS}}$ until the first $\overline{\text{BRDY}}$ is returned, and thereafter from each $\overline{\text{BRDY}}$ until the next $\overline{\text{BRDY}}$.
- DP7–DP0—Same as D63–D0.
- $\overline{\text{PCHK}}$—Two clocks after every $\overline{\text{BRDY}}$ for writes.

In addition to the above uses of $\overline{\text{BRDY}}$ on the 486 processor, $\overline{\text{BRDY}}$ on the AMD5$_K$86 and Pentium processors is used for both single-transfer and burst cycles, and it terminates special bus cycles.

Unlike $\overline{\text{BRDY}}$ on the 486 processor, $\overline{\text{BRDY}}$ on the AMD5$_K$86 and Pentium processors is used for both single-transfer and burst cycles, and it terminates special bus cycles. On the 486 processor, single-transfer cycles and special bus cycles use $\overline{\text{RDY}}$; $\overline{\text{BRDY}}$ is used only for burst cycles. The $\overline{\text{BLAST}}$ output on the 486 processor is not implemented on the AMD5$_K$86 and Pentium processors, which instead use the $\overline{\text{CACHE}}$ output to indicate cacheability. However, unlike the 486 processor, which can terminate a burst cycle prematurely by negating $\overline{\text{BLAST}}$, the AMD5$_K$86 and Pentium processors cannot terminate a burst prematurely.

5.2.12 BRDYC (Burst Ready)

Input

Summary BRDYC is an identical copy of BRDY, except that BRDYC has an internal pullup resistor whereas BRDY does not. In systems that would otherwise place large capacitive loads on BRDY, the BRDYC output can be used in place of BRDY to distribute loads, thereby increasing response times.

Sampled BRDYC is sampled with the same timing as BRDY. See the description of BRDY on page 5-42.

Details See the description of BRDY on page 5-42. Unlike BRDY, BRDYC has an internal pullup resistor.

At the falling edge of RESET, the states of BRDYC and BUSCHK control the drive strength on the A21–A3 (not including A31–A22), ADS, HITM, and W/R signals. The drive strength is weak for all states of BRDYC and BUSCHK except when BRDYC and BUSCHK are both Low, in which case the drive strength is strong. The A31–A22 signals use the weak drive strength at all times. See the data sheet for details.

5.2.13 BREQ (Bus Request)

Output

Summary The processor asserts BREQ to indicate that it is either driving a cycle on the bus, performing certain types of cache accesses, or needs access to the bus in order to continue operating.

Driven The processor asserts BREQ on the first clock of every processor-initiated bus cycle, with $\overline{\text{ADS}}$, and in the first clock of every cache store and cache-tag recovery. The processor asserts BREQ continuously while it being held off the bus and can no longer operate out of its cache.

BREQ is driven during memory cycles (including cache writethroughs and writebacks), I/O cycles, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM; or while AHOLD, $\overline{\text{BOFF}}$, HLDA, or PRDY is asserted. BREQ is not driven in the Shutdown, Halt, Stop Grant, or Stop Clock states; or while RESET or INIT is asserted.

Details The processor observes a bus-parking protocol. It continues to drive the bus without an arbitration sequence in the absence of AHOLD, $\overline{\text{BOFF}}$ or HOLD. System logic can use the assertion of BREQ to arbitrate bus access among competing bus masters. If the processor asserts BREQ only on the first clock of a cache access or bus cycle, system logic need not take action, whether or not the processor is being held off the bus. If the processor can no longer operate out of cache, it holds BREQ asserted until system logic negates the signal that is holding it off the bus (AHOLD, $\overline{\text{BOFF}}$, or HOLD). One clock after the negation of that signal, the processor drives a bus cycle with $\overline{\text{ADS}}$.

5.2.14 BUSCHK (Bus Check)

Input

Summary

System logic can assert BUSCHK if it determines that the current bus cycle has or will have any type of error. In response, the processor stores information about the aborted bus cycle and (optionally) generates a machine check exception. If machine check exceptions are not enabled, the processor attempts to continue execution after the assertion of BUSCHK. The signal is also used to set the drive strength of the A21–A3, ADS, HITM, and W/R signals at RESET.

Sampled

The processor samples BUSCHK with every BRDY, including the BRDYs of writeback cycles, and recognizes it at the next instruction boundary. BUSCHK is a level-sensitive interrupt with an internal pullup resistor. However, unlike other level-sensitive interrupts, BUSCHK is sampled with every BRDY and is not acknowledged.

BUSCHK is sampled during memory cycles (including cache writethroughs and writebacks), I/O cycles, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM; or in the Shutdown, Halt, or Stop Grant states. While AHOLD is asserted, the processor samples BUSCHK only to complete a bus cycle that had been initiated before AHOLD was asserted, or during writebacks that result from inquire cycle hits. BUSCHK is not sampled when the processor is not driving an external bus cycle; or during the Stop Clock state; or while BOFF, HLDA, RESET, INIT, or PRDY is asserted.

At the falling edge of RESET, the states of BRDYC and BUSCHK control the drive strength on the A21–A3 (not including A31–A22), ADS, HITM, and W/R signals. The drive strength is weak for all states of BRDYC and BUSCHK except BRDYC and BUSCHK both Low, in which case drive strength is strong. A31–A22 use the weak drive strength at all times. See the data sheet for details.

BUSCHK is the highest-priority external interrupt. For details on its relationship to other interrupts and exceptions, see Section 5.1.3 on page 5-14 and Table 5-3 on page 5-17.

Details

Bus cycle errors such as parity can be reported to the processor on $\overline{\text{BUSCHK}}$ if this reporting is not done on NMI. The $\overline{\text{BUSCHK}}$ signal is not used in most PC systems, although higher-end systems may find uses for it in special situations.

Upon recognizing a $\overline{\text{BUSCHK}}$ interrupt at the instruction boundary, the processor performs the following actions, in the order shown:

1. *Latch Cycle Information*—The processor latches the physical address and cycle definition of the failed bus cycle in its 64-bit machine check address register (MCAR) and 64-bit machine check type register (MCTR). These registers can be read during a service routine with the RDMSR instruction (ECX = 0 for the MCTR, ECX = 1 for the MCTR). See Section 3.3.5 on page 3-35 for details on this instruction.
2. *Machine Check Exception (Optional)*—If system software has set the MCE bit in CR4 to 1, the processor waits for the last $\overline{\text{BRDY}}$ of the failed bus cycle, then invalidates all instructions remaining in the pipeline, saves its state, and generates a machine check exception (12h).

   If the MCE bit is cleared to 0, the processor continues execution with the next instruction.

After asserting $\overline{\text{BUSCHK}}$, system logic must nevertheless return all $\overline{\text{BRDY}}$s that the processor expects for the type of bus cycle that experienced the error: one $\overline{\text{BRDY}}$ for single-transfer cycles; four $\overline{\text{BRDY}}$s for burst cycles. If $\overline{\text{BUSCHK}}$ is asserted during a locked operation or inquire cycle, an enabled machine check exception will not be acted upon until after the last $\overline{\text{BRDY}}$ of the locked operation or after a writeback caused by an inquire cycle. If $\overline{\text{BUSCHK}}$ is asserted during the Halt or Stop Grant state, the signal is sampled with $\overline{\text{BRDY}}$ but held pending until after the processor exits the Halt or Stop Grant state, at which point an enabled machine check exception will be acted upon.

If $\overline{\text{BOFF}}$ is asserted when $\overline{\text{BUSCHK}}$ is asserted, $\overline{\text{BOFF}}$ is recognized and $\overline{\text{BUSCHK}}$ is ignored. The processor does not recognize $\overline{\text{BOFF}}$ or HOLD while $\overline{\text{BUSCHK}}$ is asserted, but it does recognize AHOLD if that signal is asserted for the cycle causing the bus check. The processor latches the assertion of any edge-triggered interrupt ($\overline{\text{FLUSH}}$, $\overline{\text{SMI}}$, INIT, NMI) while

$\overline{\text{BUSCHK}}$ is asserted and recognizes latched interrupts in priority order when $\overline{\text{BUSCHK}}$ is negated.

The MCE bit in CR4, which enables machine check exceptions during $\overline{\text{BUSCHK}}$, also enables machine check exceptions during data parity errors that are indicated on $\overline{\text{PCHK}}$ while $\overline{\text{PEN}}$ is asserted.

5.2.15 $\overline{\text{CACHE}}$ (Cacheable Access)

Output

Summary

The processor drives $\overline{\text{CACHE}}$ to specify that the current bus cycle is a burst cycle. If $\overline{\text{CACHE}}$ is asserted for a read cycle, the cycle is a four-transfer burst and fills a cache line. If $\overline{\text{CACHE}}$ is asserted for a write cycle, the cycle is a four-transfer burst writeback of a *modified* cache line. $\overline{\text{CACHE}}$ is not asserted for writethroughs, so the signal is not asserted for all cycles involving cacheable locations.

Driven and Floated

The processor drives $\overline{\text{CACHE}}$ from $\overline{\text{ADS}}$ until the last expected $\overline{\text{BRDY}}$ of the bus cycle.

$\overline{\text{CACHE}}$ is driven during memory cycles, I/O cycles, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM. $\overline{\text{CACHE}}$ is not driven in the Shutdown, Halt, or Stop Grant states, except for writebacks due to inquire cycles, and $\overline{\text{CACHE}}$ is never driven during the Stop Clock state or while $\overline{\text{BOFF}}$, HLDA, RESET, INIT, or PRDY is asserted.

The processor floats $\overline{\text{CACHE}}$ one clock after system logic asserts $\overline{\text{BOFF}}$ and in the same clock that the processor asserts HLDA.

Details

The processor asserts $\overline{\text{CACHE}}$ for certain types of unlocked memory reads, as specified by the operating system, and for all writebacks (writes of lines cached in the *M* state). The assertion of $\overline{\text{CACHE}}$ indicates the processor's intent to drive the read or write cycle as a 32-byte burst and, in the case of read cycles, to cache the data or instructions. During reads, system logic can use the assertion of $\overline{\text{CACHE}}$ to initiate a table lookup of cacheable addresses. To enable caching in the processor's instruction or data cache, system logic must assert $\overline{\text{KEN}}$ during the first $\overline{\text{BRDY}}$ or $\overline{\text{NA}}$ of the bus cycle, whichever comes first. If either $\overline{\text{CACHE}}$ or $\overline{\text{KEN}}$ is negated when $\overline{\text{KEN}}$ is sampled, the processor performs a non-cacheable, single-transfer read.

The only type of write cycle for which the processor asserts $\overline{\text{CACHE}}$ are 32-byte writebacks of *modified* data. Writebacks can be caused by (a) externally initiated inquire cycles or $\overline{\text{FLUSH}}$ operations, (b) processor-initiated internal snoops or cache line replacements, or (c) program-initiated WBINVD instructions. By contrast, the processor drives writethroughs

during write hits to *shared* cache lines and during write misses, but writethroughs are driven as single transfers of 1 to 8 bytes. $\overline{\text{CACHE}}$ is not asserted during writethroughs.

$\overline{\text{CACHE}}$ is partially determined by the PCD bit maintained by the operating system (in Protected mode, for example, the PCD bit is maintained in the page directory and page table entries for the accessed page). This is the bit that fully determines the processor's page cache disable (PCD) output. PCD indicates a non-cacheable page. Thus, the states of $\overline{\text{CACHE}}$ and PCD are very often the same. $\overline{\text{CACHE}}$ is never asserted when PCD is asserted. PCD indicates the cacheability of an entire page, and $\overline{\text{CACHE}}$ indicates the burstability of a particular bus cycle; burstability is a necessary but insufficient condition for determining cacheability. The cacheability of a particular bus cycle is determined during read cycles when system logic asserts $\overline{\text{KEN}}$ while the processor asserts $\overline{\text{CACHE}}$. $\overline{\text{KEN}}$ is not a factor in determining the state of the PCD or $\overline{\text{CACHE}}$ signals. The processor drives both PCD and $\overline{\text{CACHE}}$ before it knows the state of $\overline{\text{KEN}}$. For details, see the descriptions of $\overline{\text{KEN}}$ and PCD on pages 5-90 and 5-100.

The MESI state of a cache line is determined at the time of the line-fill by the states of the $\overline{\text{CACHE}}$, $\overline{\text{KEN}}$, PWT and WB/$\overline{\text{WT}}$ signals. Table 5-9 shows the relationship between these signals and the data cache MESI states during reads. Read misses with $\overline{\text{CACHE}}$ or $\overline{\text{KEN}}$ negated are non-cacheable and are driven as single-transfer cycles on the bus. Read misses with both $\overline{\text{CACHE}}$ and $\overline{\text{KEN}}$ asserted in the first transfer of the bus cycle are cacheable, are driven as burst cycles on the bus, and have their resulting MESI state determined by PWT and WB/$\overline{\text{WT}}$. Read hits have their resulting MESI state determined entirely by their prior MESI state.

For data cache MESI state transitions during writes, see the description of the WB/$\overline{\text{WT}}$ signal on page 5-134. For more details on data-cache MESI state transitions and control, and the correspondence between MESI states and *writeback* or *writethrough* states, see Section 5.2.56 on page 5-134 and Section 6.2 on page 6-8.

$\overline{\text{CACHE}}$ is not asserted for the following types of memory reads (M/$\overline{\text{IO}}$ = 1):

- Locked reads (that is, while $\overline{\text{LOCK}}$ is asserted)
- TLB reads
- Any read with PCD asserted (PCD is a factor in determining the state of $\overline{\text{CACHE}}$)

TABLE 5-9. MESI-State Transitions for Reads

Signal or Event	Result of Cache Lookup							
	Read Miss					Read Hit		
						shared	exclusive	modified
$\overline{\text{CACHE}}$, PCD[1]	1	—	0	0	0	—	—	—
$\overline{\text{KEN}}$	—	1	0	0	0	—	—	—
PWT	—	—	1	—	0	—	—	—
WB/$\overline{\text{WT}}$	—	—	—	0	1	—	—	—
Cache Line Fill (32 bytes)	no	no	yes	yes	yes	no	no	no
State After Read[2]	—	—	*shared*	*shared*	*exclusive*	*shared*	*exclusive*	*modified*

Notes:

– *Don't care or not applicable.*

1. *The PCD bit is one determinant of the state of $\overline{\text{CACHE}}$.*
2. *Transition occurs after any line fill. Lines in "shared" MESI state are said to be in "writethrough" state. Those in "exclusive" or "modified" MESI states are said to be in "writeback" state.*

On the 486 processor, by comparison, the $\overline{\text{CACHE}}$ output does not exist, but the $\overline{\text{BLAST}}$ output (in conjunction with $\overline{\text{KEN}}$) serves to determine cacheability. Although bursts are typically four 32-bit transfers on the 486 processor, they can be longer with narrower-width memories.

5.2.16 CLK (Bus Clock)

Input

Summary CLK, in conjunction with the state of BF at RESET, determines the frequency of the processor's internal clock.

Sampled The processor always samples CLK. The clock must have begun oscillating prior to the assertion of RESET during power-up.

Details All processor signals are driven and sampled relative to the rising edge of CLK, except the edge-triggered interrupts $\overline{\text{FLUSH}}$ and $\overline{\text{SMI}}$, which are sampled on the falling edge of CLK.

The processor's internal clock runs at a multiple of CLK that is determined by the state of the BF input during RESET. A digital phase-locked loop generates the internal clock from CLK.

Power consumption can be reduced to its minimum when system logic turns CLK off. The processor enters its *Stop Clock state* when system logic asserts $\overline{\text{STPCLK}}$ (thus entering the Stop Grant state) and subsequently turns CLK off (thus entering the Stop Clock state). In the Stop Clock state, the processor's phase-lock loop and I/O buffers are disabled, except for the I/O buffers on CLK and the TAP signals. While the processor is in the Stop Clock state, system logic should not change the state of any signals other than CLK without first restarting CLK. When CLK is restarted, the processor returns to the Stop Grant state and responds to inputs in the next clock, but cannot drive bus cycles until its phase-lock loop is synchronized. The latter takes several clocks (see the data sheet for this specification). For details on $\overline{\text{STPCLK}}$ and the Stop Clock state, see page 5-123.

While the processor operates with the Test Access Port (TAP), all TAP events are timed relative to TCK rather than to CLK.

5.2.17 D/$\overline{C}$ (Data or Code)

Output

Summary

The processor drives D/$\overline{C}$ to indicate whether it is accessing data or executable code on the bus. The signal is driven at the same time as the other two cycle definition signals: M/$\overline{IO}$ and W/$\overline{R}$. A specific encoding of D/$\overline{C}$, M/$\overline{IO}$, and W/$\overline{R}$ identifies one of several special bus cycles.

Driven and Floated

The processor drives D/$\overline{C}$ from $\overline{ADS}$ until the last expected $\overline{BRDY}$ of the bus cycle.

D/$\overline{C}$ is driven with the other cycle definition outputs (M/$\overline{IO}$ and W/$\overline{R}$) and with the $\overline{BE7}$–$\overline{BE0}$ byte-enable outputs during memory cycles (including cache writethroughs and writebacks), I/O cycles, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM, or while PRDY is asserted. While AHOLD is asserted, D/$\overline{C}$ is driven only to complete a bus cycle that had been initiated before AHOLD was asserted, or for inquire cycle writebacks. During the Shutdown, Halt, and Stop Grant states, D/$\overline{C}$ is driven only for inquire cycle writebacks. D/$\overline{C}$ is not driven during the Stop Clock state, or while $\overline{BOFF}$, HLDA, RESET, or INIT is asserted.

The processor floats D/$\overline{C}$ one clock after system logic asserts $\overline{BOFF}$ and in the same clock that the processor asserts HLDA.

Details

The processor drives D/$\overline{C}$ according to whether the access is initiated by the processor's prefetch or branch logic (indicating a code access) or its load/store logic (indicating a data access). In the AMD5$_K$86 processor, code accesses can be done speculatively, but data accesses are not. Only data (not code) can be read from the I/O address space, because the cycle definition for an I/O code read (D/$\overline{C}$ = 0, M/$\overline{IO}$ = 0, W/$\overline{R}$ = 0) defines an interrupt acknowledge cycle.

Before the processor fetches an instruction or reads or writes a data operand, it checks the descriptor for the segment containing the code or data to verify that such action is allowed. The execute (E) bit in the segment descriptor distinguishes between data and code segments. A general-protection exception is generated if the E bit does not match the D/$\overline{C}$ type.

During special bus cycles, the processor drives D/$\overline{C}$ = 0, M/$\overline{IO}$ = 0, and W/$\overline{R}$ = 1. The cycles are then differentiated by $\overline{BE7}$–$\overline{BE0}$ and A31–A3.

5.2.18 D63–D0 (Data Bus)

Bidirectional

Summary

The processor drives and samples up to eight bytes on D63–D0 during memory or I/O accesses. System logic must decode the source and destination of these transfers using the address bus and various control signals.

Driven, Sampled, and Floated

As Outputs: For single-transfer writes (including cache writethroughs), the processor drives D63–D0 valid from one clock after $\overline{\text{ADS}}$ until $\overline{\text{BRDY}}$. For writebacks (the only type of burst write), the processor drives D63–D0 valid from one clock after $\overline{\text{ADS}}$ until the first $\overline{\text{BRDY}}$, and thereafter from one clock after each $\overline{\text{BRDY}}$ until the next $\overline{\text{BRDY}}$ of the bus cycle.

The processor floats D63–D0 one clock after system logic asserts $\overline{\text{BOFF}}$ in the clock that the processor asserts HLDA.

As Inputs: While $\overline{\text{BOFF}}$ or HLDA is asserted, the processor samples D63–D0 with every $\overline{\text{BRDY}}$ of the bus cycle.

D63–D0 is driven or sampled during memory cycles (including cache writethroughs and writebacks), I/O cycles, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM, or while PRDY is asserted. While AHOLD is asserted, D63–D0 is driven or sampled only to complete a bus cycle that had been initiated before AHOLD was asserted, or for inquire cycle writebacks. During the Shutdown, Halt, and Stop Grant states, D63–D0 is driven only for inquire cycle writebacks. D63–D0 is not driven or sampled during the Stop Clock state, or while $\overline{\text{BOFF}}$, HLDA, RESET, or INIT is asserted.

Details

Data is transferred between the processor and memory or I/O on up to eight bytes of the D63–D0 data bus. The $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ byte-enable signals specify the validity of each byte on D63–D0. Table 5-10 shows the relation between D63–D0 and $\overline{\text{BE7}}$–$\overline{\text{BE0}}$. System logic must interpret $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ for data byte validation during single-transfer memory reads and writes and for all I/O reads and writes. However, for burst reads (cache line fills) and writes (cache writebacks)—that is, when the processor asserts $\overline{\text{CACHE}}$—the processor expects data to be valid and will drive valid data on all eight bytes of the data bus without regard to the state of $\overline{\text{BE7}}$–$\overline{\text{BE0}}$.

TABLE 5-10. Relation Between D63–D0, $\overline{BE7}$–$\overline{BE0}$, and DP7–DP0

Byte On Data Bus	Byte Enable Output	Data Parity Bits
D63–D56	$\overline{BE7}$	DP7
D55–D48	$\overline{BE6}$	DP6
D47–D40	$\overline{BE5}$	DP5
D39–D32	$\overline{BE4}$	DP4
D31–D24	$\overline{BE3}$	DP3
D23–D16	$\overline{BE2}$	DP2
D15–D8	$\overline{BE1}$	DP1
D7–D0	$\overline{BE0}$	DP0

During burst reads the processor drives $\overline{BE7}$–$\overline{BE0}$ to identify only the byte address of the next desired operand. The byte indication does not change throughout the burst; it continues to be driven on $\overline{BE7}$–$\overline{BE0}$ during all four transfers. The memory subsystem must ignore $\overline{BE7}$–$\overline{BE0}$ during the second, third, and fourth transfers of a burst and return all eight bytes corresponding to the eight-byte address on A31–A3. Furthermore, the memory subsystem must determine the successive addresses, depending on the starting address that the processor drives on A31–A3, as described in Table 5-4 on page 5-22.

During writebacks the processor drives all bits of $\overline{BE7}$–$\overline{BE0}$ Low to indicate that all eight bytes on D63–D0 are valid. Writebacks are addressed by A31–A3, but they are always aligned to 32-byte boundaries so that A4–A3 are always 0.

If memory reads, memory writes, or I/O reads are misaligned, the Pentium processor transfers the highest-addressed portion followed by the lowest-addressed portion. The AMD5$_K$86 processor runs such cycles in the opposite order from the Pentium processor. I/O writes, however, are performed in the same order on both processors.

5.2.19 DP7–DP0 (Data Parity)

Bidirectional

Summary

DP7–DP0 carry the even-parity bits for each byte driven and sampled on the D63–D0 data bus. While DP7–DP0 are outputs, system logic can use the signals to check parity. While DP7–DP0 are inputs, the processor uses them to determine the state of the $\overline{\text{PCHK}}$ output.

Driven, Sampled, and Floated

DP7–DP0 are driven, sampled, and floated with the same timing as D63–D0. See the description for D63–D0 on page 5-56.

Details

DP7 corresponds to the high byte on the data bus (D63–D56) and DP0 corresponds to the low byte on the data bus (D7–D0). To determine data parity, the bit values driven for each byte on DP7–DP0 are considered with the bit values driven for each byte on D63–D0. For example, if the total number of 1 bits for the byte on D63–D56 is even for DP7 and D63–D56, the address is considered free of error (thus the term *even parity*). If the number of 1 bits is odd, the byte is considered to have an error.

During single-transfer read cycles, parity is only checked for enabled bytes as specified by $\overline{\text{BE7}}$–$\overline{\text{BE0}}$. During burst reads, parity is checked for all eight bytes, regardless of $\overline{\text{BE7}}$–$\overline{\text{BE0}}$. If a parity error is detected on a read, the processor asserts $\overline{\text{PCHK}}$.

Systems that do not implement data parity generation and checking should tie DP7–DP0 either High or Low and ignore the $\overline{\text{PCHK}}$ output. In addition to generating and checking data parity, the processor also generates and checks address parity using the AP and $\overline{\text{APCHK}}$ signals. See page 5-32 and 5-33 for details.

5.2.20 EADS (External Address Strobe)

Input

Summary

While system logic holds the processor off the address bus, system logic can assert EADS and drive a cache line address to initiate an *inquire cycle*. Inquire cycles cause the processor to snoop its internal caches.

Sampled

The processor samples EADS every clock, beginning two clocks after the assertion of AHOLD or BOFF, or one clock after the assertion of HLDA; except while the processor drives A31–A3, while it asserts HITM, and one clock after EADS.

While AHOLD is asserted, EADS is sampled while the processor finishes an in-progress memory cycle (including a cache writethrough or writeback), I/O cycle, locked cycle, special bus cycle, or interrupt acknowledge operation in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM. While AHOLD, BOFF, or HLDA is asserted, EADS is always sampled while the processor operates out of its cache or is idle; or is in the Shutdown, Halt, or Stop Grant state; or while INIT or PRDY is asserted. EADS is not sampled in the Stop Clock state or while RESET is asserted.

If BOFF and EADS are both asserted in the same clock that AHOLD is negated, EADS is not recognized. If EADS is asserted on the same clock that HOLD is negated, both the AMD5$_K$86 and the Pentium processors recognize this as a valid inquire cycle and process it correctly. However, if EADS is asserted on the clock following the negation of HOLD, the AMD5$_K$86 processor does not recognize this as a valid inquire cycle.

Details

Inquire cycles cause the processor to compare a physical address driven by system logic with the processor's physical address tags for its instruction and data caches. Inquire cycles can occur in parallel with the processor's own cache accesses, which are done through a separate set of linear address tags.

Inquire cycles are sometimes called *snoop* cycles, although the term *snoop* means at least three different things: (a) external snoop cycles that are occasionally driven on the bus by system logic, such as an inquire cycle, (b) internal snoops that are done automatically whenever the processor accesses its cache, such as when the processor compares the address of a write to

its data cache with the addresses in the instruction cache, and (c) automatic bus watching, in which a caching device constantly compares addresses being driven by any other device on the address bus with its own cached addresses. The AMD5$_K$86 and Pentium processors only support the first two types of snooping, not the third.

There are three methods by which system logic can obtain control of the address bus prior to running one or more inquire cycles: AHOLD, $\overline{\text{BOFF}}$ or HOLD. While it has control of at least the address bus, system logic can drive inquire cycles using $\overline{\text{EADS}}$, A31–A5, INV, and (optionally) AP.

The system logic's sequence for driving inquire cycles is as follows:

1. Assert AHOLD, $\overline{\text{BOFF}}$, or HOLD to obtain at least the address bus.
2. Assert $\overline{\text{EADS}}$ two clocks after asserting AHOLD or $\overline{\text{BOFF}}$, or one clock after the processor asserts HLDA, and simultaneously drive INV and a cache-line address on A31–A5. The processor latches the address on A31–A5 when $\overline{\text{EADS}}$ is asserted.
3. Wait two clocks, watching for $\overline{\text{HITM}}$ and/or $\overline{\text{HIT}}$ to be asserted:
   - If neither $\overline{\text{HIT}}$ nor $\overline{\text{HITM}}$ are asserted at the end of two clocks, or if only $\overline{\text{HIT}}$ is asserted, the inquire cycle terminates. $\overline{\text{EADS}}$ can be asserted again in the same clock that $\overline{\text{HITM}}$ is negated.
   - If $\overline{\text{HITM}}$ is asserted, a writeback follows and the processor does not recognize $\overline{\text{EADS}}$ again until one clock after the last $\overline{\text{BRDY}}$ of the writeback. The timing of the writeback depends on whether AHOLD, $\overline{\text{BOFF}}$ or HOLD was asserted to gain access to the bus: if AHOLD was used, the processor begins driving the four-transfer burst writeback as early as two clocks after asserting $\overline{\text{HITM}}$, whether or not AHOLD is still asserted. If $\overline{\text{BOFF}}$ or HOLD was used, the processor delays the writeback until after $\overline{\text{BOFF}}$ or HLDA is negated.

To prevent multiple inquire cycles from hitting *modified* lines, and causing a backlog of writebacks, the processor does not recognize another $\overline{\text{EADS}}$ while $\overline{\text{HITM}}$ is asserted. $\overline{\text{HITM}}$ is

negated one clock after the last $\overline{\text{BRDY}}$ of the writeback, at which time another $\overline{\text{EADS}}$ can be asserted.

If AHOLD is held asserted throughout an inquire cycle, system logic must latch the inquire cycle address when $\overline{\text{EADS}}$ is asserted. This is required so that, if the inquire cycle hits a *modified* line, the address used for the writeback need not be driven by the processor when the processor asserts $\overline{\text{ADS}}$ for the writeback; instead, system logic must use its latched copy of the inquire cycle address. By contrast, if system logic always negates AHOLD before the writeback, the processor will drive the writeback address when it asserts $\overline{\text{ADS}}$ for the writeback, and system logic need not latch a copy of the inquire cycle address.

If $\overline{\text{EADS}}$ is asserted in the same clock that HOLD is negated, the processor recognizes this as a valid inquire cycle. However, if $\overline{\text{EADS}}$ is asserted in the clock following the negation of HOLD, the processor does not recognize this as a valid inquire cycle.

Inquire cycles can be implemented for every memory access by another caching master. To do this, system logic can generate $\overline{\text{EADS}}$ to the processor using the equivalent of $\overline{\text{ADS}}$ from the other caching master.

An inquire cycle can hit a line that is in the process of being written back for a reason other than the inquire, such as when the writeback is being done to make room in the cache for a new line (called a replacement writeback) or when the WBINVD (writeback and invalidate) instruction is being executed. If this occurs, the in-progress writeback completes but the system must recognize that this writeback was for the same line that was the subject of the inquire cycle. The processor will not repeat the writeback, but it will assert $\overline{\text{HITM}}$.

If an inquire cycle occurs during a Branch-Trace Message special cycle, the branch-address information driven by the processor on A31–A3 can be overwritten by the inquiring bus master. In such cases, system logic should latch A31–A3 when $\overline{\text{ADS}}$ is asserted (that is, before asserting AHOLD, $\overline{\text{BOFF}}$ or HOLD).

$\overline{\text{EADS}}$ should not be asserted at the same time the processor is running a BIST (INIT asserted on the falling edge of RESET) or

while the TAP instruction, RUNBIST, is executed. The processor accesses the physical tag array during both BISTs and inquire cycles via $\overline{\text{EADS}}$, and these accesses can conflict.

The 486 processor without writeback cache samples $\overline{\text{EADS}}$ in every clock, including while the processor drives the address bus. It can thus support inquire cycles every clock. The AMD5$_K$86 and Pentium processors, by comparison, can sample $\overline{\text{EADS}}$ every other clock, and the maximum inquire or invalidation rate with inquire cycles is one every two clocks, because $\overline{\text{HIT}}$ and $\overline{\text{HITM}}$ change state two clocks after $\overline{\text{EADS}}$, and $\overline{\text{EADS}}$ can be asserted in the same clock in which $\overline{\text{HITM}}$ is negated. The AMD5$_K$86 processor does not sample $\overline{\text{EADS}}$ in the clock after a valid $\overline{\text{EADS}}$ assertion.

5.2.21 EWBE (External Write Buffer Empty)

Input

Summary The processor delays cache writes and certain serializing instructions if system logic negates EWBE during external writes.

Sampled The processor samples EWBE with the BRDY of external write cycles and in every clock thereafter until EWBE is asserted.

Details All writes on the AMD5$_K$86 processor—whether to cache, memory, or I/O—are performed in program order, regardless of the state of EWBE. The only effect of EWBE on writes is to hold off additional writes when the signal is negated.

The processor expects EWBE to be asserted with or after the last BRDY of each write cycle. Thus for writebacks, the processor expects EWBE to be asserted with or after the BRDY of the fourth transfer. System logic should assert EWBE when all external write buffers are empty, thus indicating that the write to memory or I/O has completed and that writes to the cache can take place. Most systems tie EWBE Low (asserted), thus allowing the speed of writes to be controlled only by BRDY.

If EWBE is sampled negated with the BRDY of an external write cycle, the processor does not do any of the following:

- Write store-buffer entry to data cache
- Write to memory (single-transfer or burst), including locked write to Accessed (A) bit after TLB load
- Execute serializing instructions like MOV to CR0, MOV to CR4, WBINVD, INVLPG, and CPUID:
- Respond to the following interrupts:
  - FLUSH
  - SMI
- Respond to any other interrupts or exceptions that cause a write to memory, such as pushing state onto the stack or setting the Accessed bit in a segment descriptor. This may include the BUSCHK, NMI, and INTR interrupts.

For interrupts that do not write to memory (R/S, INIT, and STPCLK), the state of EWBE has no effect on the processor's recognition of or response to such interrupts. The processor

latches any edge-triggered interrupt that may not be recognized while $\overline{\text{EWBE}}$ is negated ($\overline{\text{FLUSH}}$, $\overline{\text{SMI}}$, NMI) and recognizes them in priority order when $\overline{\text{EWBE}}$ is asserted.

If system logic implements memory-mapped I/O as non-cacheable memory (the standard method), $\overline{\text{EWBE}}$ on the AMD5$_K$86 processor has the same effect on writes to memory-mapped I/O as does $\overline{\text{EWBE}}$ on the Pentium processor—neither processor reorders reads ahead of writes.

For more details on the function of $\overline{\text{EWBE}}$, see the following sections:

- $\overline{\text{BRDY}}$—Page 5-42.
- $\overline{\text{HITM}}$—Page 5-74.
- $\overline{\text{SMI}}$—Page 5-117.
- $\overline{\text{SMIACT}}$—Page 5-122.
- $\overline{\text{STPCLK}}$—Page 5-123.

5.2.22 FERR (Floating-Point Error)

Output

Summary The processor asserts FERR to report floating-point errors in a manner compatible with floating-point software written for 287 and 387 coprocessors. The IGNNE input controls the behavior of FERR.

Driven The processor drives FERR every clock during memory cycles (including cache writethroughs and writebacks), cache hits of all types, I/O cycles, and locked cycles in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM. FERR is not driven during the Shutdown, Halt, Stop Grant, or Stop Clock states, or while RESET, INIT, or PRDY is asserted.

Details FERR and IGNNE support backward compatibility with floating-point software designed for 287 and 387 coprocessors on PC systems running DOS. Contemporary floating-point software typically observes these same conventions and requires these signals.

If software has set the numeric error (NE) bit in CR0 to 1, the processor reports unmasked floating-point exception conditions in the way specified for 287 and 387 coprocessors—the processor asserts FERR to report the error externally while internally the processor generates a numeric error exception (10h) while executing the next WAIT instruction or at the beginning of the next computational floating-point instruction. The IGNNE input plays no part in error reporting if the NE bit in CR0 is set to 1.

If software has cleared the numeric error (NE) bit in CR0 to 0, unmasked floating-point exception reporting depends on the state of the IGNNE input, as follows:

- If the IGNNE input is *negated*, the processor reports unmasked floating-point exception conditions in a way that is compatible with IBM-compatible PC/AT systems—the processor asserts FERR to report the error externally to a busy latch (FERR is analogous to the ERROR output on 287 and 387 coprocessors). The external busy latch generates IRQ13 if FERR is asserted, and the service routine for IRQ13 can then assert IGNNE to permit continued processing of floating-point instructions.

- If the $\overline{\text{IGNNE}}$ input is *asserted*, $\overline{\text{FERR}}$ is negated and the processor does not report unmasked floating-point exception conditions externally.

DOS and Windows®-based PCs typically clear the NE bit to 0. Only higher-end operating systems such as the Windows NT™ operating system set the NE bit to 1.

5.2.23 FLUSH (Cache Flush)

Input

Summary

FLUSH causes the processor to writeback (if necessary) and invalidate each line in its data and instruction caches. The processor generates a flush-acknowledge special bus cycle at the end of the entire operation. The signal is also used to invoke an output-float test at RESET.

Sampled and Acknowledged

The processor samples FLUSH every clock and recognizes it at the next instruction boundary. FLUSH is a falling-edge-triggered interrupt and is latched when sampled. When FLUSH is recognized, the processor acknowledges it by driving a flush-acknowledge special bus cycle after all modified lines in the data cache are written back and after all lines in both caches are invalidated.

FLUSH is sampled during memory cycles (including cache writethroughs and writebacks), cache accesses, I/O cycles, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM; or in the Shutdown, Halt, or Stop Grant states; or while AHOLD, BOFF, HLDA, or RESET is asserted. FLUSH is not sampled in the Stop Clock state, or while INIT or PRDY is asserted.

If asserted at the falling edge of RESET, FLUSH invokes the processor's three-state (float) test. System logic can drive the signal either synchronously or asynchronously (see the data sheet for synchronously driven setup and hold times).

FLUSH is the third-highest-priority external interrupt. For details on its relationship to other interrupts and exceptions, see Section 5.1.3 on page 5-14 and Table 5-3 on page 5-17.

Details

FLUSH allows system logic to control the data that the processor sees during cache accesses after changing operating modes or data environments. It also provides control for special cache coherency purposes. For example, FLUSH may be asserted when the processor enters SMM or in systems running extended memory managers if there is any change that may affect physical addresses. Depending on how an L2 cache serves the processor and other caching devices, system logic may want to cause the L2 cache to invalidate its same locations when system logic asserts FLUSH to the processor, or it may

use the Flush-Acknowledge special bus cycle to initiate such action.

Entry into SMM may require the assertion of $\overline{\text{FLUSH}}$. If the SMM physical memory space overlaps physical main memory that is cacheable, $\overline{\text{FLUSH}}$ must be asserted with $\overline{\text{SMI}}$ (the $\overline{\text{FLUSH}}$ will be performed first, because it is a higher-priority interrupt). If this is not done, accesses to the SMM memory space after entering SMM may hit cached locations in the main memory space. In addition, if SMM memory is itself cacheable, the SMM service routine should execute the WBINVD (write-back and invalidate) instruction when leaving SMM, just prior to executing the RSM instruction.

The processor performs the $\overline{\text{FLUSH}}$ operation using the same microcode that executes for the WBINVD (writeback and invalidate) instruction. The only difference is the special bus cycle driven upon completion of the operation. A writeback and invalidation operation can be time consuming because all *modified* lines in the data cache are written back to memory. If writebacks are not required, the INVD instruction or RESET can be used to invalidate all contents of the caches.

When $\overline{\text{FLUSH}}$ is recognized at an instruction boundary, the processor performs the following actions in the order shown:

1. *Flush Pipeline*—The processor invalidates all instructions remaining in the pipeline.
2. *Writeback and Invalidate*—The processor writes back any *modified* lines in the data cache, and then (after all writebacks) simultaneously invalidates all lines in the instruction and data caches. The invalidations are done by clearing the valid bits in both the linear and physical tag directories.
3. *Acknowledge*—After the writeback and invalidation completes, the processor drives a $\overline{\text{FLUSH}}$-acknowledge special bus cycle. This cycle is identified by D/$\overline{\text{C}}$ = 0, M/$\overline{\text{IO}}$ = 0, W/$\overline{\text{R}}$ = 1, $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ = EFh and A31–A3 = 0. System logic must return $\overline{\text{BRDY}}$ in response to this cycle.

AHOLD, $\overline{\text{BOFF}}$, and HOLD are all recognized and behave normally while $\overline{\text{FLUSH}}$ is asserted, and they will intervene in an in-progress $\overline{\text{FLUSH}}$ operation. For example, if $\overline{\text{BOFF}}$ is asserted while a $\overline{\text{FLUSH}}$ operation is writing *modified* lines back to memory, an in-progress writeback will be aborted.

When $\overline{\text{BOFF}}$ is subsequently negated, the writeback is restarted and the $\overline{\text{FLUSH}}$ operation continues from where it left off. Any writebacks that completed before $\overline{\text{BOFF}}$ was asserted are not affected by $\overline{\text{BOFF}}$'s intervention.

If $\overline{\text{FLUSH}}$ is asserted while AHOLD, $\overline{\text{BOFF}}$, or HLDA is asserted, the outcome of the flush depends on whether the flush causes writebacks of modified lines. If no writebacks are needed, the processor invalidates all lines but does not perform the $\overline{\text{FLUSH}}$-acknowledge cycle until the processor gets control of the bus again. If a writeback is needed, the processor stops at that writeback, without having invalidated any lines, waits until control of the bus is returned to the processor, then completes the $\overline{\text{FLUSH}}$ operation. If $\overline{\text{FLUSH}}$ is asserted during the Stop Grant state, the signal is held pending until after the processor exits the Stop Grant state, at which point it is acted upon.

No other interrupt or exception will intervene in a flush operation because such interrupts are not recognized until after the FLUSH-Acknowledge special bus cycle, which occurs at the end of all writebacks and invalidations. The processor latches the assertion of any edge-triggered interrupt ($\overline{\text{FLUSH}}$, $\overline{\text{SMI}}$, INIT, NMI) while $\overline{\text{FLUSH}}$ is asserted and recognizes latched interrupts in priority order when $\overline{\text{FLUSH}}$ is negated.

The Three-State (float) Test mode, entered if $\overline{\text{FLUSH}}$ is asserted during RESET, causes the processor to float all of its output and bidirectional signals. In this isolated state, system board traces and connections can be tested for integrity and driveability. The Float-Test mode can only be exited by asserting RESET again.

On the AMD5$_K$86 and Pentium processors, $\overline{\text{FLUSH}}$ is an edge-triggered interrupt. On the early 486 processors, however, the signal is a level-sensitive input.

5.2.24 FRCMC (Functional-Redundancy Check Master/Checker)

Input

Summary If FRCMC is asserted at RESET, the processor enters Functional-Redundancy Checking mode, as the checker, and reports checking errors on the IERR output. If 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.

Sampled The processor samples FRCMC at the falling edge of RESET. The processor does not sample FRCMC at any other time.

System logic can drive the signal either synchronously or asynchronously (see the data sheet for synchronously driven setup and hold times).

Details 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 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 checker's IERR output. On the AMD5$_K$86 processor, the 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 done in the Hardware Debug Tool (HDT) mode. The assertion of $\overline{\text{FRCMC}}$ is not recognized while PRDY is asserted.

5.2.25 HIT (Inquire-Cycle Hit)

Output

Summary The processor asserts $\overline{\text{HIT}}$ to indicate that an inquire cycle hit a valid line in the processor's instruction or data cache.

Driven The processor drives $\overline{\text{HIT}}$ every clock. The signal changes state two clocks after the assertion of $\overline{\text{EADS}}$ and retains that state until two clocks after the next $\overline{\text{EADS}}$.

$\overline{\text{HIT}}$ is driven at all times, except while the processor is in the Stop Clock state, or while RESET or INIT is asserted.

Details The processor asserts $\overline{\text{HIT}}$ if an inquire cycle address matches the address of a valid line in the processor's instruction cache in the *shared* state, or of a *shared, exclusive,* or *modified* line in the processor's data cache (called a *cache hit)*. The processor holds $\overline{\text{HIT}}$ negated if the inquire cycle address does not match any valid address in either cache (called a *cache miss)*.

Table 5-11 shows the relationship between $\overline{\text{HIT}}$, $\overline{\text{HITM}}$, and INV. Inquire cycle logic in systems with look-aside caches can be simplified by monitoring only $\overline{\text{HITM}}$ and ignoring $\overline{\text{HIT}}$. This works because the resulting state of a hit line is determined *only* by the state of the INV input during the assertion of $\overline{\text{EADS}}$:

- If INV is *negated* during a hit, the hit line—whether *shared, exclusive,* or *modified*—transitions to the *shared* state. Thus, the inquiring master can safely cache the same data in the *shared* state without knowing whether the inquire cycle hit in the processor's cache (and thus, without system logic monitoring $\overline{\text{HIT}}$).
- If INV is *asserted* during a hit, the hit line—whether *shared, exclusive,* or *modified*—transitions to the *invalid* state. If the line was *modified* before the inquire, $\overline{\text{HITM}}$ is also asserted and the line is written back before the invalidation; if the line was *shared* or *exclusive* before the inquire, no writeback occurs before the invalidation.
- If the inquire cycle misses, regardless of the state of INV, the inquiring master can cache the target data in the *shared* state, although it will not have enough information to cache that line in the *exclusive* state (this requires that $\overline{\text{HIT}}$ be monitored).

TABLE 5-11. MESI-State Transitions for Inquire Cycles

Signal or Event	Result of Cache Lookup				
	Inquire Miss	Inquire Hit			
		shared or exclusive		modified	
$\overline{\text{HIT}}$	1	0	0	0	0
$\overline{\text{HITM}}$[1]	1[2]	1	1	0	0
INV	—	1	0	1	0
Write to Memory	no	no	no	writeback (32 bytes)	writeback (32 bytes)
State After Inquire[3]	—	invalid	shared	invalid	shared

Notes:

— Don't care or not applicable.

1. *Asserted only for data cache hits to modified lines. Instruction cache lines can only be in the shared or invalid state.*
2. *$\overline{\text{HITM}}$ is never asserted while $\overline{\text{HIT}}$ is negated.*
3. *Transition occurs after any write to memory. Lines in "shared" MESI state are said to be in "writethrough" state. Those in "exclusive" or "modified" MESI states are said to be in "writeback" state.*

Inquire cycle logic in systems with look-through caches, however, normally monitor both $\overline{\text{HIT}}$ and $\overline{\text{HITM}}$ because such systems often implement the write-once cache protocol. The write-once protocol requires caching in the *exclusive* state at certain transitions, and the *exclusive* state can only be identified if both $\overline{\text{HIT}}$ and $\overline{\text{HITM}}$ are monitored. For details on this protocol, see Section 6.2.6 on page 6-19.

Inquire cycles can be driven while $\overline{\text{LOCK}}$ is asserted, if AHOLD is used to obtain the bus for the inquire cycle. An inquire cycle cannot hit a line that is involved in a locked operation ($\overline{\text{LOCK}}$ asserted). The processor prevents this by always checking its cache tags prior to a locked operation. If the location is cached, it is written back (if necessary) and invalidated prior to the locked operation.

The Pentium processor does not recognize an inquire cycle hit on an in-progress cache line fill prior to the first $\overline{\text{BRDY}}$, and it will cache that line in the *exclusive* state if PWT = 0 and WB/$\overline{\text{WT}}$ = 1. This may cause the line to be cached in the *exclusive* state in two separate caches if the system supports other caching masters. In such cases, the AMD5$_K$86 processor asserts $\overline{\text{HIT}}$ and caches the line in the *shared* state or does not cache it, depending on the state of the INV signal.

5.2.26 HITM (Inquire Cycle Hit To Modified Line)

Output

Summary

The processor asserts HITM to indicate that an inquire cycle hit a *modified* line in the processor's data cache. If this occurs, the processor writes the line back to memory during or after the bus-hold tenure, depending on which signal is holding the processor off the bus. HIT is always asserted whenever HITM is asserted.

Driven

The processor drives HITM every clock. The signal changes state two clocks after the assertion of EADS. If the inquire cycle misses the cache or hits an exclusive or shared line in the cache, the processor holds HITM negated and another inquire cycle can begin in that clock (two clocks after EADS). If the inquire cycle hits a *modified* line in the data cache, the processor asserts HITM and holds it asserted until one clock after the last BRDY of the writeback, then negates it.

HITM is driven at all times, except while the processor is in the Stop Clock state, or while RESET or INIT is asserted.

Details

The processor asserts HITM when an inquire cycle address matches the address of a *modified* line in the processor's data cache. The processor then attempts to drive a four-transfer burst writeback of the *modified* line. If INV was asserted at the time EADS was asserted for the inquire cycle, a hit leaves the written-back line in the *invalid* state. If INV was negated at the time EADS was asserted, a hit leaves the written-back line in the *shared* state. For a comparison of the states that HITM, HIT, and INV can assume, see Table 5-11 on page 5-73.

System logic can use HITM to inhibit access to the bus by other masters (via BOFF or HOLD) until the writeback associated with the hit has completed. The time at which the writeback occurs depends on which input signal was used to hold the processor off the bus for the inquire cycle:

- If AHOLD was used, the processor drives the writeback as early as two clocks after asserting HITM, whether or not AHOLD is still asserted at that time.
- If BOFF or HOLD was used, the processor delays the writeback until after BOFF or HLDA is negated. In the case of BOFF, the writeback is driven before any aborted bus cycle is restarted.

The processor drives writebacks by asserting $\overline{\text{ADS}}$ and either reusing the inquire cycle address (if AHOLD is held asserted throughout the writeback) or driving the address itself (if AHOLD is negated for the writeback, or if $\overline{\text{BOFF}}$ or HOLD was used to obtain the bus). If AHOLD is held asserted throughout an inquire cycle and a subsequent writeback, system logic must latch the inquire cycle address when it asserts $\overline{\text{EADS}}$ and use the latched copy during the writeback. By contrast, if system logic always negates AHOLD before the writeback, the processor drives the writeback address when it asserts $\overline{\text{ADS}}$ for the writeback, and system logic need not latch a copy of the inquire cycle address.

Inquire cycles can be driven while $\overline{\text{LOCK}}$ is asserted, if AHOLD is used to obtain the bus for the inquire cycle. An inquire cycle cannot hit a line involved in a locked operation. Cached locations that are about to be accessed in locked operations are written back and invalidated before the locked operation occurs. If such an inquire cycle hits a *modified* location that is *different* than the one involved in the locked operation, the writeback is done in the middle of the locked operation, between the two locked cycles, and $\overline{\text{LOCK}}$ is asserted during the writeback. This is the only case in which another operation can intervene in a locked operation. System logic must recognize this case and know that the inquire cycle is snooping a different location than the one that is locked.

At the falling edge of RESET, the states of $\overline{\text{BRDYC}}$ and $\overline{\text{BUSCHK}}$ control the drive strength on the A21–A3 (not including A31–A22), $\overline{\text{ADS}}$, $\overline{\text{HITM}}$, and W/$\overline{\text{R}}$ signals. The drive strength is weak for all states of $\overline{\text{BRDYC}}$ and $\overline{\text{BUSCHK}}$ except when $\overline{\text{BRDYC}}$ and $\overline{\text{BUSCHK}}$ are both Low, in which case the drive strength is strong. The A31–A22 signals use the weak drive strength at all times. See the data sheet for details.

5.2.27 HLDA (Bus-Hold Acknowledge)

Output

Summary

When system logic asserts HOLD, the processor completes any in-progress bus cycle, floats its cycle-driving outputs, and asserts HLDA as an acknowledgment. While HLDA is asserted, another bus master can drive cycles on the bus, including inquire cycles to the processor.

Driven

The processor drives HLDA every clock. The processor floats the cycle-driving outputs on the bus and asserts HLDA two clocks after the last $\overline{\text{BRDY}}$ of an in-progress bus cycle, if such a cycle is in progress when HOLD is asserted, or two clocks after the assertion of HOLD, whichever comes last. The processor continues to float the bus and assert HLDA until two clocks after HOLD is negated.

HLDA is driven during cache hits in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM, but writebacks wait until HLDA is negated. HLDA is also driven in the Shutdown, Halt, Stop Grant, and Stop Clock states; or while AHOLD, $\overline{\text{BOFF}}$, RESET, INIT, or PRDY is asserted. HLDA is not driven during processor-originated bus cycles, because any such pending bus cycle completes before the processor asserts HLDA.

Details

HLDA is the processor's acknowledgment to HOLD. HLDA indicates that any in-progress bus cycle has completed and that the output and bidirectional signals used for memory or I/O accesses are floating. Table 5-12 shows the signals floated. The same set of signals is floated with $\overline{\text{BOFF}}$.

TABLE 5-12. Outputs Floated When HLDA is Asserted

Address and Address Parity	Cycle Definition and Control	Data and Data Parity	Cache Control
A31–A3	D/$\overline{\text{C}}$	D63–D0	$\overline{\text{CACHE}}$
$\overline{\text{ADS}}$	$\overline{\text{LOCK}}$	DP7–DP0	PCD
$\overline{\text{ADSC}}$	M/$\overline{\text{IO}}$	$\overline{\text{BE7}}$–$\overline{\text{BE0}}$	PWT
AP	SCYC	N/A	N/A
N/A	W/$\overline{\text{R}}$	N/A	N/A

Unlike $\overline{\text{BOFF}}$, the assertion of HOLD does not abort an in-progress cycle. If the processor is not driving a bus cycle when HOLD is asserted, the bus master asserting or causing the assertion of HOLD can begin driving its first bus cycle in the clock after HLDA is asserted, which occurs two clocks after HOLD is asserted. The processor supports only one in-progress bus cycle. Unlike the Pentium processor, no pending bus cycles are held in write buffers between the data cache and the bus interface on the AMD5$_K$86 processor.

The processor can assert $\overline{\text{ADS}}$ in the clock after HOLD is asserted (but before asserting HLDA) and drive a bus cycle before acknowledging HOLD with HLDA. System logic may assert $\overline{\text{EADS}}$ for an inquire cycle as early as one clock after the processor asserts HLDA.

The processor continues driving HLDA until two clocks after HOLD is negated, at which time the processor may again drive its own cycles with $\overline{\text{ADS}}$ in the next clock after it negates HLDA. The processor responds to inquire cycles while HLDA is asserted and will assert $\overline{\text{HIT}}$ and $\overline{\text{HITM}}$ in response to such cycles. If $\overline{\text{HITM}}$ is asserted, the writeback is performed immediately after HLDA is negated. Multiple inquire cycles are not permitted to hit modified lines. The processor implements this restriction by ignoring $\overline{\text{EADS}}$ while $\overline{\text{HITM}}$ is asserted; when $\overline{\text{HITM}}$ is asserted, it is held asserted until one clock after the last $\overline{\text{BRDY}}$ of the writeback.

For a list of signals recognized while HLDA is asserted, see Table 5-2 on page 5-9. See the description of HOLD on page 5-78 for additional details about the HOLD/HLDA protocol.

5.2.28 HOLD (Bus-Hold Request)

Input

Summary

When system logic asserts HOLD, the processor completes any in-progress bus cycle, floats its cycle-driving outputs, and asserts HLDA to acknowledge the HOLD.

Sampled and Acknowledged

The processor samples HOLD every clock. It acknowledges HOLD by floating the cycle-driving outputs on the bus and asserting HLDA two clocks after the last $\overline{\text{BRDY}}$ of an in-progress bus cycle, if such a cycle is in progress when HOLD is asserted, or two clocks after the assertion of HOLD, whichever comes last. The processor continues to float the bus and assert HLDA until two clocks after HOLD is negated.

HOLD is sampled during memory cycles (including cache writethroughs and writebacks), I/O cycles, inquire cycles, and special bus cycles in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM; in the Shutdown, Halt, Stop Grant, and Stop Clock states; or while AHOLD, $\overline{\text{BOFF}}$, RESET, INIT, or PRDY is asserted. HOLD is not sampled during locked cycles or interrupt acknowledge operations.

Details

The assertion of HOLD, like $\overline{\text{BOFF}}$ but unlike AHOLD, forces the processor to relinquish the full address and data bus to another bus master. The signal can be used for the following purposes:

- *Bus Turnaround*—Another bus master can assert HOLD to the processor to obtain control of the bus, allowing the other bus master to drive any type of bus cycles.
- *Inquire Cycles*—In multi-master systems with shared memory, another bus master typically drives an inquire cycle to the processor or its L2 cache prior to driving a read or write cycle to any memory locations shared by both masters. Such inquire cycles can be driven while HOLD is asserted.

HOLD provides the slowest response of the three bus-hold inputs and is normally useful only in single-bus (non-bridged), single-processor systems with a look-aside L2 cache. For example, a DMA controller may use HOLD to obtain the bus, run inquire cycles, and perform memory reads and writes. See Section 6.2.5 on page 14 for system configurations using HOLD.

Like AHOLD but unlike $\overline{\text{BOFF}}$, HOLD allows the processor to complete an in-progress bus cycle before the processor floats its cycle-driving outputs. Such an in-progress cycle may consist of a single-transfer cycle, burst cycle, sequence of locked cycles (such as an interrupt acknowledge operation), or a special bus cycle. The processor supports only one in-progress bus cycle; no pending bus cycles are buffered. Like $\overline{\text{BOFF}}$, HOLD has no effect on writes to the processor's store buffer, except to delay them. (The store buffer is situated between the execution units and the data cache, and it is used for speculative stores prior to being written in non-speculative state to the data cache.)

When HOLD is asserted, system logic may continue asserting HOLD for as long as it wants. The processor has no way of breaking the hold. The processor continues driving HLDA until two clocks after HOLD is negated, at which time the processor may again drive its own cycles with $\overline{\text{ADS}}$ in the next clock after it negates HLDA. During the time HOLD is asserted, the processor attempts to operate out of its cache. If it can no longer do so, it asserts BREQ continuously.

There are three methods by which system logic can obtain control of the address bus to drive an inquire cycle: AHOLD, $\overline{\text{BOFF}}$, or HOLD. AHOLD obtains control only of the address bus and allows another master to drive only inquire cycles, whereas $\overline{\text{BOFF}}$ and HOLD obtain control of the full bus (address and data), allowing another master to drive not only inquire cycles but also read and write cycles. Unlike $\overline{\text{BOFF}}$, AHOLD and HOLD both permit an in-progress bus cycle to complete, but writebacks can occur while AHOLD is asserted, whereas pending writebacks during the assertion of HOLD occur after HOLD is negated, which is similar to $\overline{\text{BOFF}}$.

If $\overline{\text{EADS}}$ is asserted on the same clock that HOLD is negated, the processor recognizes this as a valid inquire cycle and handles it correctly. However, if $\overline{\text{EADS}}$ is asserted on the clock following the negation of HOLD, the AMD5$_K$86 processor does not recognize this as a valid inquire cycle.

See the description of HLDA on page 5-76 for additional details about the HOLD/HLDA protocol.

5.2.29 $\overline{\text{IERR}}$ (Internal Error)

Output

Summary The processor drives $\overline{\text{IERR}}$ only in Functional-Redundancy Checking mode. If the processor is the checker and it detects a difference in signal outputs between the master and itself, it asserts $\overline{\text{IERR}}$. No other errors are reported with $\overline{\text{IERR}}$.

Driven The processor drives $\overline{\text{IERR}}$ every clock while the processor is operating as the checker in the Functional-Redundancy Checking mode. If an error is detected, $\overline{\text{IERR}}$ is asserted for one clock, starting two clocks after the detection of the error.

$\overline{\text{IERR}}$ is only driven in Functional-Redundancy Checking mode when the processor is the checker, including while PRDY is asserted within this mode.

Details The processor enters Functional-Redundancy Checking mode as the checker if $\overline{\text{FRCMC}}$ is asserted at RESET. In this mode, all of the processor's output and bidirectional signals (except $\overline{\text{IERR}}$ and TDO) are floated and tied to those of the master processor. Both processors execute the same instructions, and 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.

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.

No other errors are reported with $\overline{\text{IERR}}$. Unlike the Pentium processor, the AMD5$_K$86 processor does not report parity errors on $\overline{\text{IERR}}$ for every cache or TLB access. Instead, the AMD5$_K$86 processor fully tests cache parity during the built-in self test (BIST), which is invoked by asserting INIT during RESET.

5.2.30 IGNNE (Ignore Numeric Error)

Input

Summary The IGNNE input, together with the numeric error (NE) bit in CR0, controls the behavior of the FERR output where the processor reports errors for DOS-compatible, floating-point programs.

Sampled The processor samples IGNNE every clock during memory cycles (including cache writethroughs and writebacks), cache hits of all types, I/O cycles, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM; or while AHOLD, BOFF, or HLDA is asserted. IGNNE is not sampled in the Shutdown, Halt, Stop Grant, or Stop Clock states; or while RESET, INIT, or PRDY is asserted.

System logic can drive the signal either synchronously or asynchronously (see the data sheet for synchronously driven setup and hold times).

Details The FERR and IGNNE signals are designed for backward-compatibility with floating-point software designed for 286 and 386 IBM-compatible PC/AT systems running DOS.

If software has cleared the numeric error (NE) bit in CR0 to 0, unmasked floating-point exception reporting depends on the state of the IGNNE input, as follows:

- If the IGNNE input is *negated*, the processor reports unmasked floating-point exception conditions in a way that is compatible with PC/AT systems—the processor asserts FERR to report the error externally to a busy latch. The external busy latch generates IRQ13 if FERR is asserted, and the service routine for IRQ13 can then assert IGNNE to permit continued processing of floating-point instructions.
- If the IGNNE input is *asserted*, FERR is negated and the processor does not report unmasked floating-point exception conditions externally.

IGNNE plays no part in error reporting if the NE bit in CR0 is set to 1. DOS and Windows-compatible PCs typically clear the NE bit to 0. Operating systems like Windows NT set the NE bit to 1. For additional details on the relation between FERR and IGNNE, see the description of the FERR signal on page 5-65.

5.2.31 INIT (Initialization)

Input

Summary The assertion of INIT causes the processor to reinitialize its system registers and certain other resources, but it preserves the contents of the caches, the floating-point state, and certain other resources. If INIT is asserted at RESET, it invokes the processor's built-in self test (BIST).

Sampled The processor samples INIT every clock and recognizes it at the next instruction boundary. INIT is a rising-edge-triggered interrupt and is latched when sampled. However, in order to be recognized reliably, the signal must be negated for two clocks prior to assertion.

INIT is sampled during memory cycles (including cache writethroughs and writebacks), cache accesses, I/O cycles, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086); or in the Shutdown, Halt, or Stop Grant states; or while AHOLD, $\overline{\text{BOFF}}$, HLDA, or RESET is asserted. INIT is not sampled in the Stop Clock state or while PRDY is asserted.

If INIT is asserted on the falling edge of RESET, the processor performs its built-in self test (BIST) before initialization and code fetching begin. System logic can drive the signal either synchronously or asynchronously (see the data sheet for synchronously driven setup and hold times).

If INIT is asserted at the same time as RESET, RESET is recognized but INIT is not. If INIT and NMI are both asserted during the Stop Grant state (not necessarily simultaneously), the AMD5$_K$86 processor recognizes the INIT after leaving the Stop Grant state, then it recognizes the NMI prior to fetching any instructions. The Pentium processor does not recognize the NMI.

INIT is the fifth-highest-priority external interrupt. For details on its relationship to other interrupts and exceptions, see Section 5.1.3 on page 5-14 and Table 5-3 on page 5-17.

Details INIT is typically asserted after power-up in response to a BIOS interrupt that writes to an I/O port. This is often, for example, in response to the operator's pressing Ctrl-Alt-Del. The BIOS

writes to a port (such as port 64h in the keyboard controller) that asserts INIT.

INIT is also used to support 286 software that must return to Real mode after accessing extended memory in Protected mode. The 286 processor does not have an INIT input; a transition from Protected mode to Real mode can only be made on the 286 processor by asserting RESET. With the INIT signal, however, the operating system, through a BIOS interrupt, can cause the transition without loss of cache contents or floating-point state.

Upon recognizing an INIT interrupt at the next instruction retirement boundary, the processor performs the following actions, in the order shown:

1. *Flush Pipeline*—The processor invalidates the:
   - Instruction pipeline
   - Translation look-aside buffer (TLB)
2. *Reinitialize*—The processor reinitializes the following resources to reset values:
   - General-purpose registers
   - System registers
3. *Jump To BIOS*—The processor jumps to the BIOS at address FFFF_FFF0h, the same entry point used after RESET. (See the description of RESET on page 5-110 for details on the aliasing of this boot address.)

Unlike RESET, INIT does not reinitialize the data and instruction caches, floating-point registers, model-specific registers, or cache disable (CD) and not-writethrough (NW) bits in CR0.

$\overline{\text{A20M}}$ should not be asserted during the first code fetch following the INIT cycle. The operating system alone is responsible for controlling the state of $\overline{\text{A20M}}$ by writing to an external register provided for this purpose. (See the description of $\overline{\text{A20M}}$ on page 5-19.)

INIT can only be driven at a predictable time, relative to program order, by using an I/O write. Due to the signal's recognition on an instruction boundary, if initialization is to be performed immediately after an I/O write, INIT must be held

asserted three clocks before the $\overline{\text{BRDY}}$ of that write in order to prevent another cycle from starting.

INIT invokes the processor's built-in self test (BIST) if asserted at the falling edge of RESET. The BIST runs a series of tests on the internal hardware that exercise the following resources—all cache tags (linear and physical) and cache arrays, the entry-point and instruction-decode PLAs, and the microcode ROM. At the end of the BIST, a value representing the result of the tests is stored in the EAX register. Zero means passed and any other value means failed. The processor continues with its normal boot process after the BIST completes, whether the BIST passed or failed.

The processor recognizes $\overline{\text{BOFF}}$, HOLD, AHOLD, and R/$\overline{\text{S}}$ while INIT is asserted, but these signals will not intervene in the initialization process except that they will prevent the first code fetch (jump to BIOS) after the registers are initialized.

No other exceptions or interrupts will intervene in the initialization process. The first code fetch after the registers are initialized will occur before another interrupt or exception is recognized. The processor latches the assertion of any edge-triggered interrupt (FLUSH, SMI, INIT, NMI) while INIT is asserted and recognizes latched interrupts in priority order when INIT is negated. If INIT is asserted during the Stop Grant state, the signal is held pending until after the processor exits the Stop Grant state, at which point it is acted upon.

5.2.32 INTR (Maskable Interrupt)

Input

Summary

The assertion of INTR, if enabled by software (unmasked), causes the processor to acknowledge the interrupt and enter an interrupt service routine. The routine is specified by the vector obtained during the acknowledgment.

Sampled and Acknowledged

The processor samples INTR every clock and recognizes it at the next instruction boundary. INTR is a level-sensitive interrupt and must be held asserted until recognized. When recognized, the processor acknowledges it by driving an interrupt acknowledge bus operation (a cycle pair).

INTR is sampled during memory cycles (including cache writethroughs and writebacks), cache accesses, I/O cycles, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM; or in the Halt state. INTR is not sampled in the Shutdown, Stop Grant, or Stop Clock states; or while AHOLD, $\overline{\text{BOFF}}$, or HLDA, RESET, INIT, or PRDY is asserted.

INTR is the seventh-highest-priority external interrupt. For details on its relationship to other interrupts and exceptions, see Section 5.1.3 on page 5-14 and Table 5-3 on page 5-17.

System logic can drive the signal either synchronously or asynchronously (see the data sheet for synchronously driven setup and hold times).

Details

In typical PC systems, maskable interrupts are driven to the processor on INTR from external interrupt-control logic that prioritizes the interrupts from several I/O devices. The processor only recognizes INTR if it is enabled in software by setting the interrupt flag (IF) in the EFLAGS register to 1.

Upon recognizing an INTR interrupt at the next instruction-retirement boundary, the processor performs the following actions, in the order shown:

1. *Flush Pipeline*—The processor invalidates all instructions remaining in the pipeline.

2. *Acknowledge*—Drives an Interrupt acknowledge operation (a cycle pair) on the bus. System logic must return a $\overline{\text{BRDY}}$ in response to both cycles. Table 5-13 shows the signal values driven during the first and second bus cycles. Both bus cycles are reads, but any data returned on the first cycle is ignored. On the second cycle, the processor samples only the enabled data byte (D7–D0) to obtain the interrupt vector. (The interrupt vector is an offset into an interrupt table containing gate or segment descriptors.) The bus cycles are driven as a locked pair, with a minimum of one idle clock between the cycles and with $\overline{\text{LOCK}}$ asserted throughout. System logic may respond as quickly as it is able; $\overline{\text{BRDY}}$ operates in the normal manner to terminate each of the two cycles. The first cycle is provided only for compatibility with the original protocol; it carries no useful information.

TABLE 5-13. Interrupt Acknowledge Operation Definition

Processor Outputs	First Bus Cycle	Second Bus Cycle
D/$\overline{\text{C}}$	0	0
M/$\overline{\text{IO}}$	0	0
W/$\overline{\text{R}}$	0	0
$\overline{\text{BE7}}$–$\overline{\text{BE0}}$	EFh	FEh (low byte enabled)
A31–A3	0	0
D63–D0	(ignored)	Interrupt vector expected from interrupt controller on D7–D0

3. *Disable Interrupts*—The processor clears the IF bit in the EFLAGS register if (a) the processor is in Real mode, or (b) the processor is in Protected mode and the interrupt vector points to an interrupt gate or to a task gate that references a TSS that has its IF bit cleared. (For details on how the IF bit is managed in Virtual-8086 mode, see page 3-12.)
4. *Service Interrupt*—Using the interrupt vector as an entry point, the processor saves its state and accesses a data structure set up by the operating system. In Real mode, the processor accesses the interrupt vector table (IVT); in Protected mode, it accesses the interrupt descriptor table (IDT). The vector identifies one of 256 gates (descriptors) in

the table. The IDT, for example, can contain interrupt, trap, or task gates, all of which point indirectly to the entry point of an interrupt service routine.

The interrupt service routine, upon entry, may re-enable interrupts by setting the IF bit in the EFLAGS before servicing the interrupt. This is typically done if the routine is lengthy, so that the processor can respond to higher-priority interrupts while the current interrupt is being serviced, thus allowing nested interrupts. Upon return from the service routine via an IRET instruction, the processor pops the contents of the CS, EIP, and EFLAGS registers (at a minimum) from the stack and continues where it left off.

System logic typically is not able to determine the instruction boundary on which the processor recognizes INTR. Thus, as a practical matter, system logic should hold INTR asserted until the beginning of the interrupt acknowledge operation, or until there is some other evidence that the interrupt service routine has been entered (for example, the access to the IDT address).

The processor disables INTR interrupts during all *software interrupts* by clearing the IF bit in EFLAGS. Software may re-enable INTR interrupts by setting IF to 1 again on entering the service routine. In this context, software interrupts include:

- In Real mode, any INT*n* instruction
- In Protected mode, any INT*n* instruction that vectors to an IDT entry that is an *interrupt gate*, or that is a *task gate* which references a TSS with the interrupt flag (IF) cleared in its EFLAGS image. (INT*n* instructions that vector to a trap gate are not considered software interrupts because the processor does not clear IF in such cases).

If system logic can leave the INTR signal asserted after the INTR service routine is entered, the interrupt vector returned by system logic during the interrupt acknowledge operation must (in Protected mode) be for an interrupt gate, or for a task gate that references a TSS with its IF cleared. If the returned vector is not one of these two types, the processor will again respond to INTR prior to executing the first instruction of the service routine, causing an infinite loop.

The processor recognizes $\overline{\text{BOFF}}$, HOLD, and AHOLD while INTR is asserted, and these signals will intervene in the INTR

service routine. Other interrupts can intervene in the INTR interrupt on entry into the INTR service routine.

INTR is not recognized if asserted while AHOLD, $\overline{\text{BOFF}}$, or HLDA is asserted, because the processor cannot drive the interrupt acknowledge operation and therefore cannot obtain the interrupt vector.

5.2.33 INV (Invalidate Cache Line)

Input

Summary During an inquire cycle, the state of INV determines whether the addressed cache line, if found in the processor's instruction or data cache, transitions to the *invalid* or *shared* state.

Sampled INV is sampled with the same timing as $\overline{\text{EADS}}$. See the description of $\overline{\text{EADS}}$ on page 5-59.

Details If INV is asserted when $\overline{\text{EADS}}$ is asserted at the beginning of an inquire cycle, the processor transitions the line (if found) to the *invalid* state, regardless of the state in which the cache line was found; such cycles are sometimes called *invalidate* cycles, or simply *invalidations*. If INV is negated when $\overline{\text{EADS}}$ is asserted, the processor transitions the line (if found) to the *shared* state. In either case, if the line is found in the *modified* state, the processor writes it back to memory before changing its state.

INV is typically asserted during a write by another caching master. In such cases, INV can be generated by watching W/$\overline{\text{R}}$ from another bus master and asserting INV to the processor, along with $\overline{\text{EADS}}$, only on writes. This method invalidates a copy that the processor may have cached, whether *modified* or not, for the same location being written by the other bus master. The processor's assertion of $\overline{\text{HITM}}$ and/or $\overline{\text{HIT}}$ does not influence how INV affects a line found in the cache. Those two outputs simply indicate whether the line was found ($\overline{\text{HIT}}$) and whether a writeback will follow ($\overline{\text{HITM}}$). If INV is asserted during the inquire, the resulting state of the line *(invalid)* is entirely determined by INV, without reference to $\overline{\text{HITM}}$ and/or $\overline{\text{HIT}}$. If INV is negated during the inquire, the resulting state of a hit line *(shared)* is also entirely determined by INV, but system logic will not know whether a writeback is imminent without monitoring $\overline{\text{HITM}}$, and another bus master will not be able to cache the line in the *exclusive* state without monitoring $\overline{\text{HIT}}$.

For a comparison of the states that $\overline{\text{HITM}}$, $\overline{\text{HIT}}$, and INV can assume, see Table 5-11 on page 5-73.

5.2.34 KEN (External Cache Enable)

Input

Summary System logic overrides the cacheability of read cycles with KEN. If KEN is negated during a read cycle, the data returned to the processor will not be cached. If KEN is asserted at that time, cacheability and the MESI state of cached lines depends on the states of the CACHE and PWT outputs and the WB/WT input.

Sampled The processor samples KEN in the same clock as the first BRDY of the read cycle or NA, whichever comes first.

KEN is sampled only during memory reads in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM. KEN is not sampled during memory writes, inquire cycles, I/O cycles, locked cycles, special bus cycles, or interrupt acknowledge operations; during the Shutdown, Halt, Stop Grant, or Stop Clock states; or while BOFF, HLDA, RESET, INIT, or PRDY is asserted. While AHOLD is asserted, KEN is sampled only to complete a bus cycle already begun before the assertion of AHOLD.

Details System logic typically maintains a specification of address cacheability in external registers that are written by BIOS at boot time. The BIOS does this by knowing or determining the address ranges of memory-mapped I/O ports and other locations that should be noncacheable. For example, video and network boards are normally mapped by BIOS to the high-memory area between 640 Kbyte and 1 Mbyte, an area that is noncacheable for both functional and security reasons. (The processor would not be able to detect changes in the state of memory-mapped network or semaphore I/O ports that are cached, and video frames written to a writeback cache would not be visible on a display.)

In Protected mode (paging enabled), the operating system can map linear addresses to physical addresses using pages that it knows to be cacheable or non-cacheable. But in non-paging modes, the operating system has no control over cacheability and the external cacheability registers are the only available mechanism for determining whether an address is cacheable or non-cacheable.

The processor's $\overline{\text{CACHE}}$ output can be used to initiate an address lookup in the external cacheability registers, and the result of the lookup can be used to drive $\overline{\text{KEN}}$.

If the address of an access falls within a cacheable range, $\overline{\text{KEN}}$ must be asserted during the first $\overline{\text{BRDY}}$ or $\overline{\text{NA}}$ of the bus cycle, whichever comes first. If $\overline{\text{KEN}}$ and $\overline{\text{CACHE}}$ are both asserted during a memory read, the processor performs the read cycle as a four-transfer burst that fills a cache line, and four $\overline{\text{BRDY}}$s must be returned with the data. If either $\overline{\text{KEN}}$ or $\overline{\text{CACHE}}$ is negated during the first $\overline{\text{BRDY}}$ or $\overline{\text{NA}}$ of the cycle, the processor ends the cycle at that $\overline{\text{BRDY}}$ or $\overline{\text{NA}}$, with only a single quadword transfer. The processor ignores $\overline{\text{KEN}}$ during writes or while $\overline{\text{CACHE}}$ is negated. For details on data-cache MESI state transitions during reads, see Table 5-9 on page 5-52.

If all of the cache *ways* in which a potential line fill can be cached are already filled with valid entries, the processor selects a line to replace during the line fill. In the data cache, if the selected line is in the *modified* state, the processor writes the modified line back to memory before filling the vacated cache line with the new contents.

If $\overline{\text{BOFF}}$ is asserted after the first eight bytes, $\overline{\text{BRDY}}$ and $\overline{\text{KEN}}$ of a cache-line fill are returned, the processor uses the first eight bytes but it does not cache them, and the line fill is aborted. When $\overline{\text{BOFF}}$ is negated, the entire bus cycle is restarted from the beginning and the system must again drive $\overline{\text{KEN}}$ in the same state that was sampled before the backoff. Thus, system logic cannot use $\overline{\text{BOFF}}$ to change the state of $\overline{\text{KEN}}$ and therefore the cacheability status of a line.

On the 486 processor, $\overline{\text{KEN}}$ is sampled twice (on the first and last transfer of a burst) and must be asserted at both times for a burst read to be treated as a cache-line fill. On the AMD5$_K$86 and Pentium processors, however, $\overline{\text{KEN}}$ is sampled only on the first clock of a transfer, during $\overline{\text{BRDY}}$ or $\overline{\text{NA}}$, whichever is first.

5.2.35 $\overline{\text{LOCK}}$ (Bus Lock)

Output

Summary The processor asserts $\overline{\text{LOCK}}$ during certain sequences of bus cycles that require integrity. To preserve the processor's handling of these sequences, system logic should prevent other bus masters from intervening in locked cycles.

Driven and Floated For locked operations, the processor asserts $\overline{\text{LOCK}}$ with $\overline{\text{ADS}}$ and holds it asserted until the last expected $\overline{\text{BRDY}}$ of the last bus cycle in the locked operation. The processor negates $\overline{\text{LOCK}}$ for at least one clock (called a dead or idle clock) between sequential locked operations.

$\overline{\text{LOCK}}$ is driven during memory cycles and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM. $\overline{\text{LOCK}}$ is not driven or not meaningful during cache writethroughs or writebacks, I/O cycles, or special bus cycles; in the Shutdown, Halt, Stop Grant, or Stop Clock states; or while $\overline{\text{BOFF}}$, HLDA, RESET, INIT, or PRDY is asserted. While AHOLD is asserted, $\overline{\text{LOCK}}$ is driven only to complete a locked cycle that had been initiated before AHOLD was asserted.

The processor floats $\overline{\text{LOCK}}$ one clock after system logic asserts $\overline{\text{BOFF}}$ and in the same clock that the processor asserts HLDA.

Details The processor always locks the following types of memory operations:

- *Interrupt Acknowledge Operations*—These are a pair of read cycles used to obtain an interrupt vector in response to the assertion of INTR.
- *Descriptor-Table Accesses*—These involve segment descriptors in the global descriptor table (GDT), local descriptor table (LDT) or interrupt descriptor table (IDT) and occur in Protected mode. The processor performs them during a segment load to ensure that the Accessed (A) bit in code and data descriptors is set to 1, or to test and set the Busy (B) bit in TSS descriptors. The sequence is as follows: (1) the processor drives an unlocked read of the descriptor to see if the relevant bit is set to 1, (2) if the bit is cleared to 0, the processor then drives a locked read-modify-write to set the bit to 1. During updates to the Accessed and Busy bits, the AMD5$_K$86 processor drives a locked four-byte read and

four-byte write sequence. The Pentium processor, however, drives a locked eight-byte read and one-byte write sequence.

Independent of these actions by the processor, the operating system can clear the Accessed or Busy bits to 0 for bookkeeping purposes. The operating system may do this however it wishes, but if locking is to be used for the memory accesses it is the operating system's responsibility to initiate locking with an XCHG or a LOCK instruction prefix.

- *Page Directory and Page Table Accesses*—The processor performs these accesses during each TLB miss to set the Accessed (A) bit to 1 in the relevant page directory and/or page table entry, and during each write access to set the Dirty (D) bit to 1 in the relevant page table entry, if those bits are not already set. These accesses work in a manner similar to descriptor table accesses, described immediately above, except that the operating system typically clears the Accessed and Dirty bits before the processor sets them, so that the operating system can thereafter identify pages that have been accessed and updated.
- *XCHG Instruction*—When XCHG is used to swap a register with a memory location, the access is unconditionally locked.
- *LOCK Prefix*—Applications programs can add the LOCK prefix to the following instructions if the destination operand resides in memory: ADC, ADD, AND, BT, BTC, BTR, BTS, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, and XCHG (redundant). The locking applies only to the bus cycle generated by that single instruction. Other uses of the LOCK prefix generate an undefined opcode fault.

Locked operations normally consist of pairs of bus cycles, typically read followed by write, except in the case of interrupt acknowledge pairs which are read-read. If the locked cycles are misaligned, the processor runs multiple pairs of bus cycles, during which $\overline{\text{LOCK}}$ and SCYC are both asserted throughout. For example, a misaligned, locked, read-modify-write sequence appears on the bus as two read cycles followed by two write cycles. Thus, up to four bus cycles can occur for misaligned accesses. (The AMD5$_K$86 processor runs certain misaligned bus cycles in the opposite order from the Pentium processor; see the description of SCYC on page 5-115 for details.)

The processor always negates $\overline{\text{LOCK}}$ for at least one idle clock between sequential locked operations. For example, if a read-modify-write is followed by another read-modify-write, there is an unlocked idle clock (sometimes called a dead clock) between the two sequences to allow system logic to reallocate the bus to another bus master. During this idle clock, the processor responds to all signals and pending interrupts.

The processor responds to AHOLD and $\overline{\text{BOFF}}$ while $\overline{\text{LOCK}}$ is asserted, and it recognizes but does not respond to HOLD until the clock after the last $\overline{\text{BRDY}}$ of the locked operation. The processor recognizes all other inputs and outputs used for memory cycles except $\overline{\text{KEN}}$, $\overline{\text{NA}}$, and WB/$\overline{\text{WT}}$.

Inquire cycles can be driven while $\overline{\text{LOCK}}$ is asserted if AHOLD is used to obtain the bus for the inquire cycle. If such an inquire cycle occurs before the last write of the locked operation and the inquire hits a *modified* cache location, the write-back is done in the middle of the locked operation between the two locked cycles with $\overline{\text{LOCK}}$ asserted during the writeback. System logic must recognize this case and know that the inquire cycle is snooping and writing back a different location than the one that is locked.

Locked operations cannot be performed on cached locations, and an inquire cycle cannot hit a line that is involved in the locked operation. The processor prevents this by always checking its cache tags prior to a locked operation. If the location is cached, it is written back (if necessary) and invalidated prior to the locked operation. This policy is necessary to support reliable semaphores for multiple caching devices, because such semaphores must never be cached and should only be accessed using locked operations.

If system logic asserts AHOLD while the processor is completing a locked cycle already begun before the assertion of AHOLD, the system must not allow accesses by other bus masters to lock the same address that the processor is locking.

If $\overline{\text{BOFF}}$ is asserted during a locked operation, only the cycle(s) aborted before their last $\overline{\text{BRDY}}$ and the cycles not yet run are restarted after $\overline{\text{BOFF}}$ is negated. Thus, system logic must keep track of all cycles in the locked operation that have completed before the assertion of $\overline{\text{BOFF}}$ and must continue the locked operation immediately after $\overline{\text{BOFF}}$ is negated, except that if a

writeback is pending when $\overline{\text{BOFF}}$ is negated, the writeback takes precedence over the restarting of the aborted cycles in the locked operation.

For purposes of interrupts and exceptions, locked operations are treated by the processor as if the entire multi-cycle operation were a single instruction. Thus, interrupts and exceptions are not recognized during locked operations. The processor samples $\overline{\text{BUSCHK}}$ if it is asserted with any $\overline{\text{BRDY}}$ of a locked operation, but the processor does not generate an enabled machine check interrupt for the $\overline{\text{BUSCHK}}$ until after the locked operation completes, and thus the exception will not intervene in the locked operation. If an edge-triggered interrupt ($\overline{\text{FLUSH}}$, $\overline{\text{SMI}}$, INIT, or NMI) is asserted during a locked operation, the interrupt is latched and recognized after the locked operation completes, even if the interrupt signal is not held asserted until the locked operation completes.

5.2.36 M/$\overline{IO}$ (Memory or I/O)

Output

Summary The processor drives M/$\overline{IO}$ to indicate whether it is accessing memory or I/O on the bus. The signal is driven at the same time as the other two cycle definition signals, D/$\overline{C}$ and W/$\overline{R}$. A specific encoding of D/$\overline{C}$, M/$\overline{IO}$, and W/$\overline{R}$ identifies one of several special bus cycles.

Driven and Floated M/$\overline{IO}$ is driven and floated with the same timing as D/$\overline{C}$. See the description of D/$\overline{C}$ on page 5-54.

Details The processor accesses I/O when it executes an I/O instruction (any of the INx or OUTx instructions). The processor accesses memory when it fetches instructions or executes an instruction that loads or stores data. Accesses to memory-mapped I/O ports appear on the bus as memory accesses.

Only data (not code) can be read or written from the I/O address space; the cycle definition for an I/O code read (D/$\overline{C}$ = 0, M/$\overline{IO}$ = 0, W/$\overline{R}$ = 0) defines an interrupt acknowledge cycle, and the cycle definition for an I/O code write (D/$\overline{C}$ = 0, M/$\overline{IO}$ = 0, W/$\overline{R}$ = 1) defines a special bus cycle.

The processor specifies all special bus cycles with D/$\overline{C}$ = 0, M/$\overline{IO}$ = 0, and W/$\overline{R}$ = 1. The cycles are then differentiated by $\overline{BE7}$–$\overline{BE0}$ and A31–A3.

5.2.37 $\overline{\text{NA}}$ (Next Address)

Input

Summary The assertion of $\overline{\text{NA}}$ indicates that external memory is prepared for a pipelined cycle.

Sampled The processor samples $\overline{\text{NA}}$ from one clock after $\overline{\text{ADS}}$ until the first expected $\overline{\text{BRDY}}$ of a bus cycle.

$\overline{\text{NA}}$ is sampled during memory cycles and writethroughs in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM. $\overline{\text{NA}}$ is not sampled during writebacks, I/O cycles, locked cycles, special bus cycles, or interrupt acknowledge operations; or in the Shutdown, Halt, Stop Grant, or Stop Clock states; or while $\overline{\text{BOFF}}$, HLDA, RESET, INIT, or PRDY is asserted. While AHOLD is asserted, $\overline{\text{NA}}$ is sampled only to complete a bus cycle already begun before the assertion of AHOLD.

Details $\overline{\text{NA}}$ is an input that is asserted when external memory is prepared to accept a pipelined cycle. The AMD5$_K$86 processor drives the pending $\overline{\text{ADS}}$ two clocks after $\overline{\text{NA}}$ is sampled active. $\overline{\text{NA}}$ does not generate pipelined cycles when $\overline{\text{LOCK}}$ is asserted, during writeback cycles, or when there are no pending internal cycles. Furthermore, locked or writeback cycles are not pipelined. $\overline{\text{KEN}}$ and WB/$\overline{\text{WT}}$ are sampled when $\overline{\text{NA}}$ or $\overline{\text{BRDY}}$ is asserted, whichever comes first.

Refer to the appropriate data sheet for model-specific details regarding the operation of $\overline{\text{NA}}$.

5.2.38 NMI (Non-Maskable Interrupt)

Input

Summary The assertion of NMI causes the processor to enter an interrupt service routine using a predefined interrupt vector.

Sampled The processor samples NMI every clock and recognizes it at the next instruction boundary. NMI is a rising-edge-triggered interrupt and is latched when sampled. The signal must be negated for at least four clocks before being asserted.

NMI is sampled during memory cycles (including cache writethroughs and writebacks), cache accesses, I/O cycles, locked cycles, special bus cycles, or interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM; in the Shutdown, Halt, or Stop Grant states; or while AHOLD, $\overline{\text{BOFF}}$, or HLDA is asserted. NMI is not sampled in the Stop Clock state, or while RESET, INIT, or PRDY is asserted.

If INIT and NMI are both asserted during the Stop Grant state (not necessarily simultaneously), the AMD5$_K$86 processor recognizes the INIT after leaving the Stop Grant state, then it recognizes the NMI prior to fetching any instructions. Current implementations of the Pentium processor do not recognize the NMI in such cases, although future implementations may.

NMI is the sixth-highest-priority external interrupt. For details on its relationship to other interrupts and exceptions, see Section 5.1.3 on page 5-14 and Table 5-3 on page 5-17.

System logic can drive the signal either synchronously or asynchronously (see the data sheet for synchronously driven setup and hold times).

Details NMI is normally used by system software to report errors such as parity, low battery, I/O channel check, board removal, timeout, and other system states that require operator attention. If such an error occurs, system software can, for example, display a screen message and wait for the operator to continue operation, if possible. In this sense, the applications for NMI are similar to those for $\overline{\text{BUSCHK}}$ and the Shutdown state, although the three are not functionally related. In typical PC systems, the signal is controlled by a system software interrupt to BIOS or a write to an I/O port (such as port 61h and/or 92h). In spite of its

name, some PC systems allow the interrupt to be masked with a write to an I/O port (such as port 70h).

Upon recognizing an NMI interrupt at the next instruction retirement boundary, the processor performs the following actions, in the order shown:

1. *Flush Pipeline*—The processor invalidates all instructions remaining in the pipeline.
2. *Service Interrupt*—The processor saves its state and accesses vector 2 in the interrupt vector table (IVT) or interrupt descriptor table (IDT), depending on whether the processor is running in Real mode or Protected mode. The vector identifies a gate descriptor in the table. The IDT, for example, can contain interrupt, trap, or task gates, all of which point indirectly to the entry point of an interrupt service routine.

The processor recognizes $\overline{\text{BOFF}}$, HOLD, and AHOLD while NMI is asserted and these signals will intervene in the NMI service routine. The processor latches the assertion of any edge-triggered interrupt ($\overline{\text{FLUSH}}$, $\overline{\text{SMI}}$, INIT, NMI) while $\overline{\text{BUSCHK}}$ is asserted and recognizes latched interrupts in priority order when $\overline{\text{BUSCHK}}$ is negated. If NMI is asserted during the Stop Grant state, the signal is held pending until after the processor exits the Stop Grant state, at which point it is acted upon.

During SMM, the Pentium processor does not respond to NMI until the beginning of its response to the first INTR or software interrupt (INTn) to occur after entering SMM. NMIs can thus be enabled by using a dummy interrupt. When an INTR or software interrupt is recognized, the processor first responds to a pending NMI interrupt before executing the first instruction of the INTR handler. By contrast, the AMD5$_K$86 processor recognizes a pending NMI interrupt after returning (via the IRET instruction) from a prior interrupt.

5.2.39 PCD (Page Cache Disable)

Output

Summary The processor drives PCD to indicate the operating system's specification of cacheability for the entire current page. System logic can use PCD to control external caching.

Driven and Floated The processor drives PCD from the clock in which $\overline{\text{ADS}}$ is asserted until the last expected $\overline{\text{BRDY}}$ of the bus cycle.

PCD is driven during memory cycles (including cache writethroughs and writebacks) and locked cycles in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM. While AHOLD is asserted, PCD is driven only to complete a bus cycle that had been initiated before AHOLD was asserted. PCD is not driven during special bus cycles, or interrupt acknowledge operations; or in the Shutdown, Halt or Stop Grant states, except for writebacks due to inquire cycles; and PCD is never driven during the Stop Clock state, or while $\overline{\text{BOFF}}$, HLDA, RESET, INIT, or PRDY is asserted.

The processor floats PCD one clock after system logic asserts $\overline{\text{BOFF}}$ and in the same clock that the processor asserts HLDA.

Details If PCD is negated during read misses, the page being accessed may or may not be cacheable, depending on the state of other signals. If PCD is asserted during any type of access, the page is noncacheable. The PCD output affects the processor's caching of data only during read misses. It has no effect on the processor during read hits, write misses, or write hits, as shown in Tables 5-17 and 5-18 on page 5-136.

The state of the PCD output is a page-level specification of cacheability based on the state of several bits written by the operating system. In Protected mode, the PCD output specifies the cacheability of the entire page being accessed. The bits that determine the PCD output are stored in one of the processor's control registers or its TLB. Those bits include the cache disable (CD) bit in CR0, the paging enable (PG) bit in CR0, and the page cache disable (PCD) bit in one of three locations. The selection of bits depends on the processor's operating mode and the type of access, as follows:

- In Real mode, or in Protected and Virtual-8086 modes while paging is disabled (PG bit in CR0 cleared to 0):

  PCD output = CD bit in CR0

  (Thus, whenever the CD bit in CR0 is set to 1, the PCD output is asserted and the access is non-cacheable.)

- In Protected and Virtual-8086 modes while caching is enabled (CD bit in CR0 cleared to 0) and paging is enabled (PG bit in CR0 set to 1):

  For accesses to I/O space, page directory entries, and other non-paged accesses:

  PCD output = PCD bit in CR3

  For accesses to 4-Kbyte page table entries or 4-Mbyte pages:

  PCD output = PCD bit in page directory entry

  For accesses to a 4-Kbyte pages:

  PCD output = PCD bit in page table entry

The method of selecting the PCD bit is similar to that for the PWT bit, described on page 5-106. The cache disable (CD) and not-writethrough (NW) bits in CR0 are cleared to 0 for normal, cacheable operation. If a location is already cached before the operating system sets a PCD bit to 1, any access to that location will hit in the cache regardless of the state of the PCD bit or signal.

$\overline{\text{CACHE}}$ is partially determined by the PCD bit. Thus, the states of $\overline{\text{CACHE}}$ and PCD are very often the same. $\overline{\text{CACHE}}$ is never asserted when PCD is asserted. PCD indicates the cacheability of an entire page, and $\overline{\text{CACHE}}$ indicates the burstability of a particular bus cycle; burstability is a necessary but insufficient condition for determining cacheability. The cacheability of a particular bus cycle is determined during read cycles when system logic asserts $\overline{\text{KEN}}$ while the processor asserts $\overline{\text{CACHE}}$. $\overline{\text{KEN}}$ not a factor in determining the state of the PCD or $\overline{\text{CACHE}}$ signals. The processor drives both PCD and $\overline{\text{CACHE}}$ before it knows the state of $\overline{\text{KEN}}$. For details, see the descriptions of $\overline{\text{CACHE}}$ and $\overline{\text{KEN}}$ on pages 5-50 and 5-90.

5.2.40 PCHK (Parity Status)

Output

Summary The processor asserts PCHK during reads if it detects an even parity error on one or more bytes of D63–D0 during a read cycle.

Driven The processor drives PCHK for one clock, two clocks after each BRDY during read cycles.

PCHK is driven for memory and I/O reads, locked reads, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM, or while PRDY is asserted. PCHK is not driven during any type of write cycles or special bus cycles; or during the Shutdown, Halt, Stop Grant, or Stop Clock states; or while BOFF, HLDA, RESET, or INIT is asserted. While AHOLD is asserted, PCHK is driven only to complete a bus cycle already begun before the assertion of AHOLD.

Details To determine data parity, the bit value driven on DP7–DP0 is considered with the bit values driven on D63–D0. If the total number of 1 bits is even for DP7–DP0 and D63–D0, the byte is considered free of error (thus the term *even parity*). If the number of 1 bits is odd, the byte is considered to have an error. During burst reads, the processor checks all eight bytes of D63–D0 for errors, with respect to the even parity bit sampled on DP7–DP0. During single-transfer reads, only the enable bytes on D63–D0 and the enabled parity bits on DP7–DP0 (as specified by BE7–BE0) are checked.

If PEN is asserted during the BRDY for a read cycle, and the processor reports a data parity error on PCHK for that cycle, the processor latches the physical address and cycle definition of the failed bus cycle and (optionally) generates a machine check exception. See the description of PEN on page 5-103 for details.

If an error is reported on PCHK, the system must nevertheless return all remaining BRDYs for that bus cycle—one BRDY for single-transfer cycles and four BRDYs for burst cycles. Systems that do not implement data parity generation and checking should tie DP7–DP0 either High or Low and ignore the PCHK output.

5.2.41 $\overline{\text{PEN}}$ (Parity Enable)

Input

Summary System logic can assert $\overline{\text{PEN}}$ to enable cycle information latching and (optionally) machine check exception generation for data bus parity errors during read cycles.

Sampled The processor samples $\overline{\text{PEN}}$ every $\overline{\text{BRDY}}$ during read cycles.

$\overline{\text{PEN}}$ is sampled for memory and I/O reads, locked reads, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM, or while PRDY is asserted. $\overline{\text{PEN}}$ is not sampled during any type of write cycles or special bus cycles; or during the Shutdown, Halt, Stop Grant, or Stop Clock states; or while $\overline{\text{BOFF}}$, HLDA, RESET, or INIT is asserted. While AHOLD is asserted, $\overline{\text{PEN}}$ is sampled only to complete a bus cycle already begun before the assertion of AHOLD.

Details If $\overline{\text{PEN}}$ is asserted when a data parity error is reported on $\overline{\text{PCHK}}$, the processor latches the physical address and cycle definition of the failed bus cycle in its 64-bit machine check address register (MCAR) and its 64-bit machine check type register (MCTR). These registers can be read with the RDMSR instruction. See Section 3.3.5 on page 3-35 for details on this instruction.

In addition to latching the cycle address and definition, the processor also generates a machine check exception (12h) if the MCE bit in CR4 is set to 1 while $\overline{\text{PEN}}$ is asserted. System logic must then handle the error externally. Typical PC systems provide a mechanism for asserting NMI during a parity error.

If $\overline{\text{PEN}}$ is negated, neither the address and cycle definition latching nor the machine check exception generation occur.

The MCE bit in CR4 also enables the generation of a machine check exception during bus cycle errors that are indicated on the $\overline{\text{BUSCHK}}$ input. The machine check mechanism is not, however, used for address parity errors indicated on $\overline{\text{APCHK}}$.

5.2.42 PRDY (Probe Ready)

Output

Summary The processor asserts PRDY to acknowledge the system logic's assertion of R/$\overline{S}$ or execution of the Test Access Port (TAP) instruction, USEHDT, and to indicate the processor's entry into the Hardware Debug Tool (HDT) mode for debugging.

Driven The processor drives PRDY every clock in response to either R/$\overline{S}$ or the TAP instruction, USEHDT. The processor asserts PRDY at the next instruction boundary after R/$\overline{S}$ is sampled Low or when the USEHDT instruction is executed. The latter causes the processor to assert PRDY without a transition on R/$\overline{S}$. After PRDY is asserted by either means, the processor negates PRDY on the later of (a) the clearing of the TAP instruction register, (b) a TAP reset, or (c) after a Low-to-High transition on R/$\overline{S}$.

PRDY is driven in memory cycles (including writethroughs and writebacks), cache accesses, and I/O cycles in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM; in the Shutdown, Halt or Stop Grant states; or while AHOLD, $\overline{BOFF}$, HLDA, or RESET is asserted. PRDY is not driven during locked cycles, special bus cycles, or interrupt acknowledge operations; during the Stop Clock state; or while INIT is asserted.

Details The HDT is entered either when external debug logic drives R/$\overline{S}$ Low or loads the TAP instruction register with the USEHDT instruction. If R/$\overline{S}$ is used to initiate the HDT, the debug logic must hold R/$\overline{S}$ Low throughout the debug session. If the USEHDT instruction is used to initiate the HDT, the processor asserts PRDY without a transition on R/$\overline{S}$.

The processor negates PRDY and begins fetching instructions for normal operation one clock after a Low-to-High transition on R/$\overline{S}$, or when the TAP instruction register is cleared, or the TAP is reset.

Debug software can force the processor into SMM, but the processor does not recognize $\overline{SMI}$ or any other interrupts while PRDY is asserted. If system hardware or software wishes to assert RESET, it must exit the HDT before asserting RESET.

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

5.2.43 PWT (Page Writethrough)

Output

Summary

The processor drives PWT to indicate the operating system's specification of *writeback* or *writethrough* state for the entire current page. PWT, together with WB/$\overline{WT}$, specifies the data-cache MESI state of cacheable read misses and write hits.

Driven and Floated

The processor drives PWT from the clock in which $\overline{ADS}$ is asserted until the last expected $\overline{BRDY}$ of the bus cycle.

PWT is driven during memory cycles (including cache writethroughs and writebacks), and locked cycles in the normal operating modes (Real, Protected, and Virtual-8086), and in SMM, and when PRDY is asserted. If AHOLD is asserted, PWT is driven only to complete a bus cycle that had been initiated before AHOLD was asserted. PWT is not driven during special bus cycles or interrupt acknowledge operations; or in the Shutdown, Halt or Stop Grant states, except for writebacks due to inquire cycles; and PWT is never driven during the Stop Clock state, or while $\overline{BOFF}$, HLDA, RESET, or INIT is asserted.

The processor floats PWT one clock after system logic asserts $\overline{BOFF}$ and in the same clock that the processor asserts HLDA.

Details

As Table 5-14 shows, lines in the *modified* or *exclusive* MESI state are said to be in the *writeback* state, which corresponds to PWT = 0. Lines in the *shared* MESI state are said to be in the *writethrough* state, which corresponds to PWT = 1.

TABLE 5-14. PWT, Writeback/Writethrough, and MESI

MESI State	Writeback/Writethrough State	PWT State
modified	writeback	0
exclusive	writeback	0
shared	writethrough	1
invalid	invalid	—

System logic can use PWT output, along with its WB/$\overline{WT}$ input, to determine how the processor will control internal caching. Tables 5-17 and 5-18 on page 5-136 show how the state of PWT and WB/$\overline{WT}$ determine the MESI state of a line in the data cache after a cache-line fill or writeback. If WB/$\overline{WT}$ is Low or PWT is High during a read miss or a write hit to a *shared* line,

the accessed line is cached in, transitions to, or remains in the *shared* state after the access. If PWT is Low and WB/$\overline{WT}$ is High, the accessed line is cached in, transitions to, or remains in the *exclusive* state after a read miss or the first write hit. A subsequent write to an *exclusive* line changes it to *modified*.

The state of the PWT output is based on the state of several bits written by the operating system. In Protected mode, the PWT output applies to the entire current page rather than to the specific bus cycle that the WB/$\overline{WT}$ output applies to, and it is the operating system's (rather than the processor hardware's) determination of *writeback* or *writethrough* state.

The bits that determine the PWT output are stored in a processor control register or the TLB. Those bits include the paging enable (PG) bit in CR0 and the page writethrough (PWT) bit in one of three locations. The selection of bits depends on the processor's operating mode and the type of access, as follows:

- In Real mode, and in Protected and Virtual-8086 modes while paging is disabled (PG bit in CR0 cleared to 0):

  PWT output = Low (writeback)

- In Protected and Virtual-8086 modes while paging is enabled (PG bit in CR0 set to 1):

  For accesses to I/O space, page directory entries, and other non-paged accesses:

  PWT output = PWT bit in CR3

  For accesses to 4-Kbyte page table entries or 4-Mbyte pages:

  PWT output = PWT bit in page directory entry

  For accesses to a 4-Kbyte pages:

  PWT output = PWT bit in page table entry

The method of selecting the PWT bit is similar to that for the PCD bit as described on page 5-100. The cache disable (CD) and not-writethrough (NW) bits in CR0 are cleared to 0 for normal, cacheable operation.

In the Hardware Debug Tool (HDT) mode, PWT is only meaningful for cache write misses (PWT = 0 and WB/$\overline{WT}$ = 1 transition a shared line to an *exclusive* line). The signal is not meaningful during cache read misses in HDT mode, because the caches are never filled during HDT mode.

5.2.44 R/$\overline{S}$ (Run or Stop)

Input

Summary

External hardware and software use R/$\overline{S}$ to control entry into and exit from the Hardware Debug Tool (HDT) mode, which supports access to the processor's DR7–DR0 debug registers through an external debug port. The AMD5$_K$86 processor implements the HDT in a manner different than the Pentium processor's Probe mode.

Sampled and Acknowledged

The processor samples R/$\overline{S}$ every clock and recognizes it at the next instruction boundary. R/$\overline{S}$ is a level-sensitive interrupt with an internal pullup resistor. It must be held asserted until recognized. When recognized, the processor acknowledges R/$\overline{S}$ by asserting PRDY at the next instruction boundary.

R/$\overline{S}$ is sampled during memory cycles (including writethroughs and writebacks), cache accesses, and I/O cycles in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM; in the Shutdown, Halt or Stop Grant states; or while AHOLD, $\overline{BOFF}$, HLDA, RESET, or INIT is asserted. R/$\overline{S}$ is not sampled during locked cycles, special bus cycles, or interrupt acknowledge operations; or during the Stop Clock state.

R/$\overline{S}$ is the second-highest-priority external interrupt. For details on its relationship to other interrupts and exceptions, see Section 5.1.3 on page 5-14 and Table 5-3 on page 5-17.

Test logic can drive the signal either synchronously or asynchronously (see the data sheet for synchronously driven setup and hold times).

Details

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.

At the next instruction retirement boundary after system debug logic drives R/$\overline{S}$ Low or loads the TAP instruction register with the USEHDT instruction, the processor performs the following actions, in the order shown:

1. *Flush Pipeline*—The processor invalidates all instructions remaining in the pipeline.

2. *Acknowledge*—The processor asserts PRDY to acknowledge the interrupt and mark entry into the HDT mode. The processor does not save its state before asserting PRDY because it will continue execution at the next instruction after returning from the debug session, when R/$\overline{S}$ and PRDY are negated.

If R/$\overline{S}$ is used to initiate the HDT, the debug logic must hold R/$\overline{S}$ Low throughout the debug session. The processor negates PRDY and begins fetching instructions for normal operation one clock after a Low-to-High transition on R/$\overline{S}$, or when the TAP instruction register is cleared or the TAP is reset.

The processor recognizes AHOLD, $\overline{BOFF}$, and HOLD while R/$\overline{S}$ is Low, and these signals will intervene in the HDT mode when PRDY is asserted. However, exceptions or interrupts are not recognized in the HDT mode. The processor latches the assertion of any edge-triggered interrupt ($\overline{FLUSH}$, $\overline{SMI}$, INIT, NMI) during the HDT mode and recognizes them in priority order when PRDY is negated. See Table 5-3 on page 5-17 for the priority of interrupts and exceptions.

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

The AMD5$_K$86 processor implements the HDT mode in a manner different than the Pentium processor's Probe mode. For details on the processor's PRDY acknowledgment to R/$\overline{S}$, see page 5-104. For details on TAP testing, see Section 7.8 on page 7-19.

5.2.45 RESET (Reset)

Input

Summary The assertion of RESET initializes the processor to the power-up state.

Sampled The processor samples RESET every clock and recognizes it at the next instruction boundary. The RESET process begins at the falling edge of RESET. To be recognized, RESET must be held asserted for at least 1 ms after V_{CC} and CLK reach specification.

The following inputs are sampled on the falling edge of RESET:

- BF is sampled to select the frequency ratio between the processor's internal clock and the bus clock (CLK).
- If $\overline{\text{FLUSH}}$ is asserted, the processor invokes the three-state (float) test.
- If $\overline{\text{FRCMC}}$ is asserted, the processor enters Functional-Redundancy Checking mode as the checker.
- If INIT is asserted, the processor performs its built-in self test (BIST) before initialization and code fetching begin.

The processor samples RESET at all times, except in the Stop Clock state and while INIT or PRDY is asserted. System logic can drive the signal either synchronously or asynchronously (see the data sheet for synchronously driven setup and hold times).

Details RESET is typically asserted at power-up by a power-good signal from the power supply, which is turned on by a hardware switch. RESET can also be asserted after power-up. For example, pressing a front-panel button can cause a BIOS interrupt to write to an I/O port (such as port 64h in the keyboard controller). After RESET, the operating system usually determines the cause of the reset (reset during or after power-up) with another BIOS interrupt that queries another I/O port (such as location 0Fh in the CMOS memory at ports 70 and 71h), and it uses this information to determine whether a full power-on test (POST) of the system should be run.

Starting at the falling edge of a recognized RESET, the processor performs the following actions, in the order shown:

1. *Flush Pipeline*—The processor invalidates the:
   - Instruction pipeline
2. *Reinitialize*—The processor reinitializes the following resources to reset values:
   - General-purpose registers
   - System registers
   - Floating-point registers
   - Model-specific registers (MSRs)
   - Data-cache tag directory (linear and physical) and data array. No writebacks are performed.
   - Instruction-cache tag directory (linear and physical) and instruction array
   - Translation look-aside buffer (TLB)
   - Branch-prediction bits
   - Clears the interrupt flag (IF) in EFLAGS to 0
3. *Jump To BIOS*—The processor jumps to physical address FFFF_FFF0h, the same entry point used after INIT, where it expects to find the BIOS entry point.

The contents of AMD5$_K$86 processor registers at the conclusion of RESET or INIT is identical to that of the Pentium processor, except that the CPU ID in EDX is 0000_050xh. The upper byte of DX (DH) contains 05h and the lower byte of DX (DL) contains 0xh, the processor's type and stepping identifier.

Table 5-15 shows the contents of registers after RESET or INIT. Table 5-16 shows the state of the processor's outputs after RESET.

TABLE 5-15. Register State After RESET or INIT

Register	Contents (hex)
EIP	FFFF_FFF0
EFLAGS	0000_0002
EAX	0000_0000
EBX	0000_0000
ECX	0000_0000
EDX	0000_050x
ESI	0000_0000

TABLE 5-15. Register State After RESET or INIT (continued)

Register	Contents (hex)
EDI	0000_0000
EBP	0000_0000
ESP	0000_0000
FPU Stack R7–R0	0000_0000_0000_0000_0000
FPU Exception Pointer	0_0000_0000_0000
CS	F000
SS	0000
DS	0000
ES	0000
FS	0000
GS	0000
GDTR	base:0000_0000 limit:0000
IDTR	base:0000_0000 limit:0000
TR	0000
LDTR	0000
CR0	6000_0010
CR2	0000_0000
CR3	0000_0000
CR4	0000_0000
DR7	0000_0400
DR6	FFFF_0FF0
DR3	0000_0000
DR2	0000_0000
DR1	0000_0000
DR0	0000_0000

TABLE 5-16. Outputs at RESET

Output	RESET State
$\overline{\text{ADS}}$	1
A31–A3	Floating
$\overline{\text{APCHK}}$	1
$\overline{\text{BE7}}$–$\overline{\text{BE0}}$	FFh
BREQ	1
$\overline{\text{BRDY}}$	1
$\overline{\text{BRDYC}}$	1
$\overline{\text{CACHE}}$	1
D/$\overline{\text{C}}$	0
D63–D0	Floating
DP7–DP0	00h
$\overline{\text{FERR}}$	1
$\overline{\text{HIT}}$	1
$\overline{\text{HITM}}$	1
HLDA	0
$\overline{\text{LOCK}}$	1
M/$\overline{\text{IO}}$	0
PCD	0
$\overline{\text{PCHK}}$	1
PRDY	0
PWT	0
W/$\overline{\text{R}}$	0

Unlike INIT, RESET reinitializes the processor's entire state. In particular, RESET differs by reinitializing the contents of the caches, floating-point registers, control registers, and model-specific registers, as well as all other states that are reinitialized by INIT.

$\overline{\text{A20M}}$ should not be asserted during RESET. The operating system alone is responsible for controlling the state of $\overline{\text{A20M}}$ by writing to an external register provided for this purpose. (See the description of $\overline{\text{A20M}}$ on page 5-19.)

Because the processor boots in Real mode, the memory address decoder must alias the physical address FFFF_FFF0h to the physical address 000F_FFF0h, which lies within the 1-Mbyte

address limit required in Real mode. (The physical address 000F_FFF0h is sometimes written in the *selector:offset* format as F000:FFF0.) This reset address behavior of the x86 architecture is due to the special way in which segment translation is performed on reset. Normally, a Real-mode 16-bit segment selector is shifted left 4 bits (one hex digit) to form the segment base, and then added to the 16-bit offset. Thus, F000:FFF0 in the *selector:offset* format becomes a segment base of F0000h added to an offset of FFF0h, yielding the physical address 000F_FFF0h. When RESET is asserted, however, the left shift is not done and the high 16 address bits are all set to 1, yielding the physical address FFFF_FFF0h. Thereafter, address translation only begins to work in the normal Real-mode manner when the first far jump is executed. This jump loads the code segment register with a 16-bit segment selector. This code segment load causes the address translation mechanism to begin working normally. The system logic address decoder must make this behavior transparent to software by aliasing the physical address FFFF_FFF0h to the physical address 000F_FFF0h.

The processor recognizes AHOLD, $\overline{\text{BOFF}}$, and HOLD while RESET is asserted, but these signals will not intervene in the initialization process except that they will prevent the first code fetch (jump to BIOS) after the registers are initialized.

While RESET is asserted, the processor recognizes or drives only BF, $\overline{\text{FLUSH}}$, FRCMC, the hold signals (AHOLD, $\overline{\text{BOFF}}$, HOLD, and HLDA), INIT, and R/$\overline{\text{S}}$.

Unlike the Pentium processor, the AMD5$_K$86 processor does not recognize RESET in the Hardware Debug Tool (HDT) mode. System hardware or software must exit the HDT (by driving R/$\overline{\text{S}}$ High) before asserting RESET.

5.2.46 SCYC (Split Cycle)

Output

Summary The processor asserts SCYC during misaligned, locked transfers on the D63–D0 data bus. The processor generates additional bus cycles to complete the transfer of misaligned data.

Driven and Floated The processor drives SCYC from the clock in which $\overline{\text{ADS}}$ is asserted until the last expected $\overline{\text{BRDY}}$ of the bus cycle.

SCYC may be driven during any memory and I/O cycles, whether locked or not, but it is only meaningful during locked memory cycles in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM. SCYC is not driven or is not meaningful during unlocked memory cycles, I/O cycles, inquire cycles, special bus cycles, or interrupt acknowledge operations; in the Shutdown, Halt, Stop Grant, or Stop Clock states; while $\overline{\text{BOFF}}$, HLDA, RESET, or INIT is asserted; or while PRDY is asserted. While AHOLD is asserted, SCYC is driven only to complete a locked memory cycle already begun before the assertion of AHOLD.

The processor floats SCYC one clock after system logic asserts $\overline{\text{BOFF}}$ and in the same clock that the processor asserts HLDA.

Details For purposes of bus cycles, the term *aligned* means:

- 2- and 4-byte transfers lie within 4-byte address boundaries
- 8-byte transfers lie within 8-byte address boundaries

(For purposes of exceptions, the term *aligned* means situated on the natural boundaries of an instruction or operand. Thus, a 2-byte transfer that crosses a 2-byte address boundary may incur an alignment exception, but it will be performed as an aligned bus cycle.)

If data on D63–D0 is misaligned, the processor generates additional bus cycles to complete the transfer. For example, if a 4-byte transfer begins at address x07h, one byte is transferred during the first bus cycle and the remaining three bytes are transferred during a second bus cycle, which normally occurs immediately after the first bus cycle (unless intervened, such as by an interrupt or bus backoff). If the misaligned transfer is run as a locked cycle, the processor asserts both $\overline{\text{LOCK}}$ and SCYC throughout the misaligned sequence of bus cycles.

If memory reads, memory writes, or I/O reads are misaligned, the AMD5$_K$86 processor runs the bus cycles in the opposite order of the Pentium processor. The AMD5$_K$86 processor transfers the low-address portion followed by the high-address portion instead of the high-address portion followed by the low-address portion.

I/O writes, however, are performed in the same order on both processors.

5.2.47 $\overline{\text{SMI}}$ (System Management Interrupt)

Input

Summary

The assertion of $\overline{\text{SMI}}$ causes the processor to enter System Management Mode (SMM). In this mode, which can be transparent to standard system and application software, an SMM interrupt service routine accesses a memory space separate from main memory. SMM is most commonly used for power management, although it is not limited to these functions.

Sampled and Acknowledged

The processor samples $\overline{\text{SMI}}$ every clock and recognizes it at the next instruction boundary. $\overline{\text{SMI}}$ is a falling-edge-triggered interrupt with an internal pullup resistor. It is latched when sampled. When recognized, $\overline{\text{SMI}}$ is acknowledged with $\overline{\text{SMIACT}}$ after the later of (a) the last expected $\overline{\text{BRDY}}$ of any in-progress bus cycle, or (b) the assertion of $\overline{\text{EWBE}}$ with or following the last expected $\overline{\text{BRDY}}$. $\overline{\text{SMI}}$ must be negated for at least four clocks before being asserted. It must be asserted at least three clocks before $\overline{\text{BRDY}}$ if it is to be recognized on the instruction boundary associated with that $\overline{\text{BRDY}}$.

$\overline{\text{SMI}}$ is sampled during memory cycles (including cache writethroughs and writebacks), cache accesses, I/O cycles, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM; in the Shutdown, Halt, or Stop Grant states; or while AHOLD, $\overline{\text{BOFF}}$, or HLDA is asserted. $\overline{\text{SMI}}$ is not sampled in the Stop Clock state, or while RESET, INIT, or PRDY is asserted.

$\overline{\text{SMI}}$ is the fourth-highest-priority external interrupt. For details on its relationship to other interrupts and exceptions, see Section 5.1.3 on page 5-14 and Table 5-3 on page 5-17.

System logic can drive the signal either synchronously or asynchronously (see the data sheet for synchronously-driven setup and hold times).

Details

$\overline{\text{SMI}}$ is typically driven by a power management block of system logic that monitors activity on processor outputs, such as the address and cycle definition signals in conjunction with a timer. An SMM interrupt service routine in firmware controls events during SMM. The most common applications involve power management via clock and/or I/O device control. For example, the external power management logic may notice

that an I/O device has not been accessed for several minutes. The power management logic can then assert $\overline{\text{SMI}}$, and the SMM service routine can obtain relevant information from the power management logic with which to make power-down decisions under program control. These decisions can be communicated back to the power management logic, which in turn can power the I/O device down and assert $\overline{\text{STPCLK}}$ to the processor.

Upon recognizing an $\overline{\text{SMI}}$ interrupt at the next instruction retirement boundary, the processor performs the following actions, in the order shown:

1. *Flush Pipeline*—The processor invalidates all instructions remaining in the pipeline.
2. *Complete In-Progress Cycle*—If the processor had begun a bus cycle when $\overline{\text{SMI}}$ was asserted, the processor completes the bus cycle and waits until the system asserts the last expected $\overline{\text{BRDY}}$ and also asserts $\overline{\text{EWBE}}$.
3. *Acknowledge*—After sampling $\overline{\text{EWBE}}$ asserted, the processor asserts $\overline{\text{SMIACT}}$ to acknowledge the interrupt. At that point, system logic must ensure that all memory accesses during SMM are to the SMM memory space.
4. *Save Processor State*—The processor saves its state in a 512-byte SMM state-save area at the top of the 32-Kbyte SMM memory area, starting at default physical location 0003_FFFFh and filling down.
5. *Disable Interrupts and Debug Traps*—The processor disables maskable interrupts by clearing the interrupt flag (IF) in EFLAGS, disables NMI interrupts, clears the trap flag (TF) in EFLAGS, and clears the DR7–DR6 debug control and status registers.
6. *Service Interrupt*—The processor jumps to the entry point of the SMM service routine at the SMM base physical address, whose default is 0003_8000h in SMM memory. The SMM base address can be rewritten with another address while the processor is in SMM. The new address is written to the SMM base slot in the SMM state-save area and is stored internally in the processor.

The processor does not assert $\overline{\text{SMIACT}}$ until it sees $\overline{\text{EWBE}}$ asserted. This ensures that any write data in external write buffers is written to the proper memory space (main memory,

not SMM memory) before address decoding switches memory references to the SMM memory space. If no bus cycle is in progress when $\overline{\text{SMI}}$ is asserted, or if the system does not implement external write buffers, system logic may assert $\overline{\text{EWBE}}$ at the same time as $\overline{\text{SMI}}$ or at some later time. If a bus cycle is in progress when $\overline{\text{SMI}}$ is asserted, $\overline{\text{EWBE}}$ must be asserted with the last expected $\overline{\text{BRDY}}$ or later.

The default physical location for the 64-Kbyte SMM memory area is between 0003_0000h and 0003_FFFFh, of which a minimum 32-Kbyte region between 0003_8000h and 0003_FFFFh must be populated with RAM. The memory controller normally uses the processor's assertion of $\overline{\text{SMIACT}}$ to enable SMM memory. The BIOS and system logic typically remap the SMM memory area from its default location in low memory to high or extended memory. System logic must ensure that, during SMM, all memory accesses are to this SMM memory area or a remapped location.

In general, system designs that do not overlap the address space of SMM memory and main memory are simpler and may perform better. However, if SMM memory space overlaps main memory space that is cacheable, $\overline{\text{FLUSH}}$ must be asserted when $\overline{\text{SMI}}$ is asserted so that memory accesses in SMM do not hit locations cached from main memory. The $\overline{\text{FLUSH}}$ is performed first, because it is a higher-priority interrupt.

If SMM memory is to be cacheable, $\overline{\text{FLUSH}}$ should also be asserted with $\overline{\text{SMI}}$ when entering SMM, and the SMM service routine should execute the WBINVD instruction to invalidate the caches when leaving SMM, just prior to executing the RSM instruction. If SMM memory is to be noncacheable, $\overline{\text{KEN}}$ must be negated when $\overline{\text{FLUSH}}$ and $\overline{\text{SMI}}$ are asserted.

SMM addresses and operands default to 16 bits, addresses are translated in the same manner as in Real mode, and the full 4 Gbytes can be accessed without a segment limit violation. Unlike the Pentium processor, the AMD5$_K$86 processor does not recognize $\overline{\text{A20M}}$ in SMM. The processor exits SMM (that is, the SMM service routine) when it executes the RSM instruction. This instruction causes the processor to copy the contents of the SMM state-save area into the processor's registers and flush the instruction pipeline. Then, the processor continues executing instructions at the location specified by the CS:EIP value from the state-save area (which will be where the proces-

sor left off when it recognized $\overline{\text{SMI}}$, unless the value is altered by the SMM service routine).

If the assertion of $\overline{\text{SMI}}$ was recognized on the boundary of an I/O instruction, the I/O trap restart feature of SMM can optionally be used to restart the I/O instruction when returning from SMM. The SMM service routine can implement this restart feature by writing the value 00FFh into the I/O trap restart slot of the SMM state-save area. If the value is 00FFh (rather than its default, 0000h) upon return from SMM, the processor decrements the instruction pointer and re-executes the I/O instruction. This is useful, for example, if an I/O write to disk finds the disk powered down. The external power management logic monitoring such an access can assert $\overline{\text{SMI}}$. In this case, the SMM service routine would query power management logic, find a failed I/O write, take action to power up the I/O device, enable the I/O restart feature by writing the value FFh into the I/O trap restart slot, and return.

During a simultaneous $\overline{\text{SMI}}$ I/O trap (for I/O instruction restart) and debug breakpoint trap, the AMD5$_K$86 processor responds to the $\overline{\text{SMI}}$ first and postpones writing the exception-related information to the stack until after the return from SMM via the RSM instruction. (If debug registers DR3–DR0 are used in SMM, they must be saved and restored by the SMM software; the processor automatically saves and restores DR7–DR6.) If the I/O trap restart slot in the SMM state-save area is written with the value FFh when the RSM instruction is executed, the debug trap does not occur until after the I/O instruction is re-executed.

The processor recognizes AHOLD, $\overline{\text{BOFF}}$, and HOLD while $\overline{\text{SMIACT}}$ is asserted and these signals will intervene in the SMM service routine. After assertion of $\overline{\text{SMI}}$, subsequent assertions of $\overline{\text{SMI}}$ are masked so as to prevent recursive entry into SMM. Any other type of exception or interrupt, however, will intervene in the SMM service routine, although the INTR and NMI interrupts are managed in a special way as described in the paragraph below. If $\overline{\text{SMI}}$ is asserted during the Stop Grant state, the signal is held pending until after the processor exits the Stop Grant state, at which point it is acted upon.

When SMM is entered, the processor disables both INTR and NMI interrupts. On both the AMD5$_K$86 and Pentium processors, INTR interrupts are disabled by clearing the IF flag in

EFLAGS. But the mechanism by which NMI interrupts are disabled and subsequently recognized differs between the AMD5$_K$86 and Pentium processors.

During SMM, the Pentium processor does not respond to NMI until the beginning of its response to the first INTR or software interrupt (INTn) to occur after entering SMM. NMIs can thus be enabled by using a dummy interrupt. When an INTR or software interrupt is recognized, the processor first responds to a pending NMI interrupt before executing the first instruction of the INTR handler. By contrast, the AMD5$_K$86 processor recognizes a pending NMI interrupt after returning (via the IRET instruction) from a prior interrupt.

The same dummy interrupt used on the Pentium processor to enable NMI recognition during SMM works on the AMD5$_K$86 processor. The only difference is that the AMD5$_K$86 processor responds to the NMI after the IRET of the dummy interrupt whereas the Pentium processor responds at the beginning of the dummy interrupt.

During debugging using the R/$\overline{S}$ and PRDY protocol, the debugger can force the processor into SMM but the processor will not recognize $\overline{SMI}$ in the Hardware Debug Tool (HDT) mode.

For further details on the System Management Mode, see Chapter 6.

5.2.48 SMIACT (System Management Interrupt Active)

Output

Summary The processor acknowledges the assertion of SMI with the assertion of SMIACT. The acknowledgment signifies the processor's readiness to enter System Management Mode (SMM) and begin executing the service routine for that interrupt mode.

Driven The processor drives SMIACT from after the later of (a) the last expected BRDY of any in-progress bus cycle, or (b) the assertion of EWBE with or following the last expected BRDY, until the return from the SMM interrupt handler via the RSM instruction.

SMIACT is driven during memory cycles (including cache writethroughs and writebacks), cache accesses, I/O cycles, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM; in the Shutdown, Halt, or Stop Grant states; or while AHOLD, BOFF, HLDA, or PRDY is asserted. SMIACT is not driven in the Stop Clock state, or while RESET is asserted.

Details The memory controller normally uses the assertion of SMIACT to enable SMM memory, so that the first memory access in SMM is to the base of the state-save area in the SMM memory space.

The processor remains in SMM, continuing to assert SMIACT, until it executes the RSM instruction. For more information regarding SMM, see the description of SMI on page 5-117, Section 6.1.4 on page 6-5, and Section 6.3 on page 6-23.

5.2.49 STPCLK (Stop Clock)

Input

Summary

The assertion of STPCLK causes the processor to complete any in-progress bus cycle and enter the Stop Grant state (processor's internal clock stopped), from which it can subsequently transition to the Stop Clock state (bus clock stopped). These low-power clock states can be entered from the normal operating modes, system management mode (SMM), or the Halt state.

Sampled and Acknowledged

The processor samples STPCLK every clock and recognizes it at the next instruction boundary. STPCLK is a level-sensitive interrupt with an internal pullup resistor. The signal must be held asserted until recognized. When STPCLK is recognized and EWBE is asserted, the processor acknowledges it by driving a Stop Grant special bus cycle, waits for BRDY, then stops its internal clock and floats D63–D0 and DP7–DP0.

STPCLK is sampled during memory cycles (including cache writethroughs and writebacks), cache accesses, I/O cycles, locked cycles, special bus cycles, and interrupt acknowledge operations in the normal operating modes (Real, Protected, and Virtual-8086) and in SMM; or in the Shutdown, Halt, or Stop Grant states. STPCLK is not sampled in the Stop Clock state, or while RESET, INIT, or PRDY is asserted. STPCLK is not meaningful if it is asserted while AHOLD, BOFF, or HLDA is asserted, because the processor cannot drive the Stop Grant special bus cycle.

STPCLK is the lowest-priority external interrupt. For details on its relationship to other interrupts and exceptions, see Section 5.1.3 on page 5-14 and Table 5-3 on page 5-17.

System logic can drive the signal either synchronously or asynchronously (see the data sheet for synchronously driven setup and hold times).

Details

In typical PC systems that implement power control, the STPCLK, CLK, and SMI signals are driven by external power management logic. This logic monitors activity on the address and cycle definition signals. In a typical case, the power management logic may notice that, after having initiated SMM to power down one or more I/O devices, another several minutes have elapsed without activity. Power management logic can again assert SMI, the SMM service routine would obtain the

relevant information and decide to power itself (the processor) down, and the decision would be communicated to the power management logic, which would assert $\overline{\text{STPCLK}}$ to the processor and, optionally, stop driving CLK to the processor and other logic.

Upon recognizing a $\overline{\text{STPCLK}}$ interrupt at the next instruction retirement boundary, the processor performs the following actions, in the order shown:

1. *Flush Pipeline*—The processor invalidates all instructions remaining in the pipeline.
2. *Complete In-Progress Cycle*—If the processor had begun a bus cycle or locked operation when $\overline{\text{STPCLK}}$ was asserted, the processor completes the bus cycle and waits until the system asserts the last expected $\overline{\text{BRDY}}$ and also asserts $\overline{\text{EWBE}}$. If no bus cycle is in progress, system logic must assert $\overline{\text{EWBE}}$ at the same time or at some time after it asserts $\overline{\text{STPCLK}}$.
3. *Acknowledge*—After sampling both $\overline{\text{EWBE}}$ asserted, the processor drives a Stop Grant special bus cycle. This cycle is identified by D/$\overline{\text{C}}$ = 0, M/$\overline{\text{IO}}$ = 0, W/$\overline{\text{R}}$ = 1, $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ = FBh and A31–A3 = 10h. System logic must respond with $\overline{\text{BRDY}}$.
4. *Stop Internal Clock*—When system logic returns $\overline{\text{BRDY}}$ for the Stop Grant special bus cycle, the processor stops its internal clock and floats D63–D0 and DP7–DP0.
5. *(Optional) Stop Bus Clock*—After returning $\overline{\text{BRDY}}$ in response to the Stop Grant special bus cycle, power management logic can transition to the Stop Clock state by stopping CLK while $\overline{\text{STPCLK}}$ is held asserted. This reduces power consumption to its minimum.

$\overline{\text{STPCLK}}$ must be held asserted throughout the Stop Grant and (if entered) Stop Clock states. Within less than 10 clocks after $\overline{\text{STPCLK}}$ is negated, the processor returns to the state from which it entered Stop Grant and can recognize any latched interrupts or drive $\overline{\text{ADS}}$.

The processor enters the *Halt state* from the normal operating modes (Real, Protected or Virtual-8086) or SMM when it executes the HLT instruction. The processor leaves the Halt state and returns to its prior operating mode when RESET, $\overline{\text{SMI}}$, INIT, NMI, or INTR is asserted. If $\overline{\text{STPCLK}}$ is asserted within

the Halt state, the processor transitions to the Stop Grant state; it then returns to the Halt state when $\overline{\text{STPCLK}}$ is negated. No processor registers are saved before entering the Halt state because the processor returns to the next unexecuted instruction in program order when it returns to its prior operating mode. Within the Halt state, the processor disables the majority of its internal clock distribution and (if $\overline{\text{STPCLK}}$ is asserted) the internal pullup resistor on $\overline{\text{STPCLK}}$. However, its phase-lock loop still runs, its key internal logic is still clocked, most of its inputs and outputs retain their last state (except D63–D0 and DP7–DP0, which are floated), and it still responds to input signals.

The assertion of $\overline{\text{STPCLK}}$ causes the processor to enter the *Stop Grant state*. The processor can enter the Stop Grant state from the normal operating modes (Real, Protected or Virtual-8086), SMM, or the Halt state. When $\overline{\text{STPCLK}}$ is negated, the processor leaves the Stop Grant state and returns to the mode from which it entered. If the Stop Grant state was entered from the Halt state, negation of $\overline{\text{STPCLK}}$ returns the processor to the Halt state. Otherwise, negation of $\overline{\text{STPCLK}}$ or assertion of RESET returns the processor to a normal operating mode (Real, Protected or Virtual-8086) or SMM. If INIT is asserted in the Stop Grant state, the signal is latched and acted upon after $\overline{\text{STPCLK}}$ is negated. No processor registers are saved before entering the Stop Grant state because the processor returns to the next unexecuted instruction in program order when it returns to its prior operating mode. Within the Stop Grant state (as in the Halt state) the processor disables the majority of its internal clock distribution and (if $\overline{\text{STPCLK}}$ is asserted) the internal pullup resistor on $\overline{\text{STPCLK}}$. However, its phase-lock loop still runs, its key internal logic is still clocked, most of its inputs and outputs retain their last state (except D63–D0 and DP7–DP0, which are floated), and it still responds to input signals.

An inquire cycle driven while the processor is in the Stop Grant state or the Halt state causes the processor to transition to the *Stop Grant Inquire state*. As for inquire cycles driven from any other state, system logic must assert AHOLD, $\overline{\text{BOFF}}$, or HOLD to obtain the address bus before driving $\overline{\text{EADS}}$, INV, and the inquire address. The processor responds normally by driving $\overline{\text{HITM}}$ and/or $\overline{\text{HIT}}$ and performing any necessary cache-state transition. If $\overline{\text{HITM}}$ is asserted, the processor drives a nor-

mal writeback (immediately if AHOLD is asserted, or delayed if $\overline{\text{BOFF}}$ or HOLD are asserted) and returns to the state from which it entered the Stop Grant Inquire state in the clock in which it negates $\overline{\text{HITM}}$. If $\overline{\text{HITM}}$ is not asserted, the processor returns two clocks after $\overline{\text{EADS}}$.

The processor enters the *Stop Clock state* when system logic turns off CLK while $\overline{\text{STPCLK}}$ is asserted. This is the minimum-power state and it can only be entered from the Stop Grant state after $\overline{\text{BRDY}}$ has been returned for the Stop Grant special bus cycle. In the Stop Clock state, the processor's phase-lock loop and I/O buffers are disabled, except for the I/O buffers on CLK and the Test Access Port (TAP) signals. System logic should not change the state of any signals, and the processor does not recognize any signal edges in the Stop Clock state. When CLK is restarted, the processor returns to the Stop Grant state, responds to inputs in the next clock, but cannot drive bus cycles until its phase-lock loop is synchronized. The latter takes several clocks (see the data sheet for this specification). The CLK can be driven with a different frequency, and/or the bus-to-processor clock ratio can be changed on the BF input upon restarting CLK.

Thus, when CLK is restarted, the processor can:

- Respond to AHOLD, $\overline{\text{BOFF}}$, or HOLD in the next clock after CLK restarts, and
- Transition to the Stop Grant Inquire state as early as two clocks after the assertion of AHOLD, two clocks after the assertion of $\overline{\text{BOFF}}$, or one clock after the assertion of HLDA (if system logic drives an inquire cycle with $\overline{\text{EADS}}$, INV and an inquire address) and
- Drive $\overline{\text{HITM}}$ and/or $\overline{\text{HIT}}$ two clocks after $\overline{\text{EADS}}$.

However, if the inquire cycle hits a *modified* line, the processor does not drive the writeback until several clocks after CLK restarts (see the data sheet). In this case, the only indication system logic receives of the writeback is the $\overline{\text{ADS}}$ that initiates it.

Thus, the processor recognizes AHOLD, $\overline{\text{BOFF}}$, and HOLD during the Stop Grant and Stop Grant Inquire states but not during the Stop Clock state. When asserted in the Stop Grant state, these signals cause the processor to restart its internal clock and transition to the Stop Grant Inquire state. When the

processor is in the Stop Clock state, however, CLK must be restarted before any other signals are changed.

$\overline{\text{STPCLK}}$ is the lowest-priority interrupt, as shown in Table 5-3 on page 5-17. R/$\overline{\text{S}}$ is the only interrupt or exception that is acted upon while $\overline{\text{STPCLK}}$ is asserted, but R/$\overline{\text{S}}$ is only acted upon in the Stop Grant state, not the Stop Clock state. Edge-triggered interrupts ($\overline{\text{FLUSH}}$, $\overline{\text{SMI}}$, INIT, NMI) are not latched in the Stop Clock state; however, they are latched in the Stop Grant state and are recognized after $\overline{\text{STPCLK}}$ is negated.

The AMD5$_K$86 and Pentium processors differ in their support for $\overline{\text{STPCLK}}$ in the following ways:

- In the Halt state, the AMD5$_K$86 processor responds to $\overline{\text{STPCLK}}$ by entering the Stop Grant state. The Pentium processor ignores $\overline{\text{STPCLK}}$ in the Halt state.
- The Pentium processor guarantees that at least one instruction will be executed between the negation of $\overline{\text{STPCLK}}$ and a subsequent reassertion of $\overline{\text{STPCLK}}$. The AMD5$_K$86 processor does not guarantee this.
- In the Halt or Stop Grant states, the AMD5$_K$86 processor cannot enter a low-power state if it does not have the bus (that is, if AHOLD, $\overline{\text{BOFF}}$ or HLDA is asserted). The same may not be true of the Pentium processor.

For further details on clock control and power management, see Section 6.4 on page 6-33 and Section 6.6 on page 6-40.

5.2.50 TCK (Test Clock)

Input

Summary TCK is the clock for boundary-scan testing using the Test Access Port (TAP).

Sampled The processor always samples TCK, except while RESET or INIT is asserted. The signal has an internal pullup resistor.

Details Data and state definition are clocked into the processor on the rising edge of TCK. The outputs on TDO are driven valid on the falling edge of TCK. When TCK stops on its falling edge, the state of test latches in the processor are held.

Section 7.8 on page 7-19 summarizes the implementation of TAP testing on the AMD5$_K$86 processor. System logic should tie TCK High if TAP testing is not implemented.

See the *IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE 1149.1)* specification for details on how the TAP signals and instructions are used for testing. The TAP is often called the Joint Test Action Group (JTAG) port, after the committee that proposed the IEEE TAP standard.

5.2.51 TDI (Test Data Input)

Input

Summary TDI carries input test data and instructions for testing on the Test Access Port (TAP).

Sampled The processor samples TDI every rising TCK edge, but only during the *shift_IR* and *shift_DR* states. TDI has an internal pullup resistor.

TDI is always sampled, except while RESET or INIT is asserted.

Details Instructions are shifted into the processor on TDI during the *shift_IR* TAP state. Data are shifted into the processor on TDI during the *shift_DR* TAP state.

See the *IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE 1149.1)* specification for a description of how the TAP signals and instructions are used for testing.

5.2.52 TDO (Test Data Output)

Output

Summary TDO carries output data for testing on the Test Access Port (TAP).

Driven and Floated The processor drives TDO every falling TCK edge, but only during the *shift_IR* and *shift_DR* states. It is floated at all other times.

TDO is always driven, except when floated and while RESET or INIT is asserted.

Details Instructions are shifted out of the processor on TDO during the *shift_IR* TAP state. Data are shifted out of the processor on TDO during the *shift_DR* TAP state.

See the *IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE 1149.1)* specification for a description of how the TAP signals and instructions are used for testing.

5.2.53 TMS (Test Mode Select)

Input

Summary TMS specifies the test function and sequence of test changes for testing on the Test Access Port (TAP).

Sampled The processor samples TMS every rising TCK edge. TMS has an internal pullup resistor.

TMS is always sampled, except while RESET or INIT is asserted.

Details If TMS is asserted for five or more clocks, the TAP controller enters its test-reset-logic state, regardless of the controller state. This action is the same as that achieved by asserting TRST.

See the *IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE 1149.1)* specification for a description of how the TAP signals and instructions are used for testing.

5.2.54 TRST (Test Reset)

Input

Summary The assertion of $\overline{\text{TRST}}$ initializes the Test Access Port (TAP) by resetting its state machine.

Sampled $\overline{\text{TRST}}$ is an asynchronous input. Unlike other asynchronous inputs, no synchronous setup and hold time are specified for $\overline{\text{TRST}}$. $\overline{\text{TRST}}$ has an internal pullup resistor.

$\overline{\text{TRST}}$ is always sampled, except while RESET or INIT is asserted.

Details When $\overline{\text{TRST}}$ is asserted, the TAP controller enters its test-reset-logic state, regardless of the controller state. This action is the same as that achieved by holding TMS asserted for five or more clocks. The assertion of $\overline{\text{TRST}}$ is unnecessary at RESET because the processor performs the TAP reset automatically at that point.

See the *IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE 1149.1)* specification for a description of how the TAP signals and instructions are used for testing.

5.2.55 W/$\overline{R}$ (Write or Read)

Output

Summary The processor drives W/$\overline{R}$ to indicate whether it is performing a write or read cycle on the bus. The signal is driven at the same time as the other two cycle definition signals: D/$\overline{C}$ and M/$\overline{IO}$. A specific encoding of D/$\overline{C}$, M/$\overline{IO}$ and W/$\overline{R}$ identifies one of several special bus cycles.

Driven and Floated W/$\overline{R}$ is driven and floated with the same timing as D/$\overline{C}$. See the description of D/$\overline{C}$ on page 5-54.

Details The processor drives W/$\overline{R}$ according to whether the access is initiated by the processor's fetch logic (which can initiate only reads) or its load/store logic (which can initiate reads or writes of operands). Such accesses can be done speculatively. Before the processor fetches an instruction or reads or writes a data operand, it checks the associated code or data segment descriptor to verify that such action is permitted. The execute (E) bit in the segment descriptor maintained by the operating system distinguishes between data and code segments, and the (R/W) bit specifies the segment's read and write properties. Code segments can only be read; data and stack segments can read-only or read-write.

The processor specifies all special bus cycles with D/$\overline{C}$ = 0, M/$\overline{IO}$ = 0 and W/$\overline{R}$ = 1. The cycles are then differentiated by $\overline{BE7}$–$\overline{BE0}$ and A31–A3.

At the falling edge of RESET, the states of $\overline{BRDYC}$ and $\overline{BUSCHK}$ control the drive strength on the A21–A3 (not including A31–A22), $\overline{ADS}$, $\overline{HITM}$, and W/$\overline{R}$ signals. The drive strength is weak for all states of $\overline{BRDYC}$ and $\overline{BUSCHK}$ except when $\overline{BRDYC}$ and $\overline{BUSCHK}$ are both Low, in which case the drive strength is strong. The A31–A22 signals use the weak drive strength at all times. See the data sheet for details.

5.2.56 WB/$\overline{\text{WT}}$ (Writeback or Writethrough)

Input

Summary WB/$\overline{\text{WT}}$, together with PWT, specifies the data-cache MESI state of cacheable read misses and write hits.

Sampled The processor samples WB/$\overline{\text{WT}}$ in the same clock as the first $\overline{\text{BRDY}}$ of a bus cycle or $\overline{\text{NA}}$, whichever comes first.

WB/$\overline{\text{WT}}$ is sampled during memory reads and writes, including writebacks, in the normal operating modes (Real, Protected, and Virtual-8086) and SMM, and when PRDY is asserted. WB/$\overline{\text{WT}}$ is not sampled during I/O cycles, locked cycles, special bus cycles, or interrupt acknowledge operations; or during the Shutdown, Halt, Stop Grant, or Stop Clock states; or while $\overline{\text{BOFF}}$, HLDA, RESET, or INIT is asserted. While AHOLD is asserted, WB/$\overline{\text{WT}}$ is sampled only to complete a bus cycle begun before the assertion of AHOLD.

Details Lines in the *shared* MESI state are said to be in the *writethrough* state. Those in the *exclusive* or *modified* MESI state are said to be in the *writeback* state. When a write access either misses the data cache or hits a *shared* line in the data cache, the processor drives a 1-to-8-byte write cycle (called a *writethrough*) on the bus. When an inquire cycle, internal snoop, $\overline{\text{FLUSH}}$ operation, or WBINVD instruction hits a *modified* line in the data cache, the processor drives a 32-byte burst write cycle (called a *writeback*) on the bus. Table 2-2 on page 2-19 shows the relationships between cache accesses, writethroughs, and writebacks.

WB/$\overline{\text{WT}}$ and PWT determine the MESI state of a cache line after a read miss (and resulting cache-line fill) or a write hit. During read misses, these two signals are interpreted along with the states of the $\overline{\text{CACHE}}$ output and the $\overline{\text{KEN}}$ input. During write hits, WB/$\overline{\text{WT}}$ and PWT alone determine the resulting MESI state of a cache line. Tables 5-17 and 5-18 shows the relationship between WB/$\overline{\text{WT}}$ and PWT for reads (Table 5-17) and writes (Table 5-18). If WB/$\overline{\text{WT}}$ is Low or PWT is High during a read miss or write hit, the accessed line is cached in, transitions to, or remains in the *shared* state after the read or write. If PWT is Low and WB/$\overline{\text{WT}}$ is High, the accessed line is cached in, transitions to, or remains in the *exclusive* state after a read miss or the first write hit to that line. If the line transitions to

the *exclusive* state, a subsequent write hit to the same line transitions the line to the *modified* state. During write hits, the states of PWT and WB/$\overline{WT}$ can only change a line from *shared* to *exclusive;* it cannot change an *exclusive* line to a *shared* line.

TABLE 5-17. MESI-State Transitions for Reads

Signal or Event	Result of Cache Lookup							
	Read Miss					Read Hit		
						shared	exclusive	modified
$\overline{CACHE}$, PCD[1]	1	—	0	0	0	—	—	—
$\overline{KEN}$	—	1	0	0	0	—	—	—
PWT	—	—	1	—	0	—	—	—
WB/$\overline{WT}$	—	—	—	0	1	—	—	—
Cache-Line Fill (32 bytes)	no	no	yes	yes	yes	no	no	no
State After Read[2]	—	—	*shared*	*shared*	*exclusive*	*shared*	*exclusive*	*modified*

Notes:

– *Don't care or not applicable.*

1. *The PCD bit is one determinant of the state of $\overline{CACHE}$.*
2. *Transition occurs after any line fill. Lines in shared MESI state are said to be in writethrough state. Those in exclusive or modified MESI states are said to be in writeback state.*

TABLE 5-18. MESI-State Transitions for Writes

Signal or Event	Result of Cache Lookup				
	Write Miss	Write Hit			
		shared			exclusive or modified
$\overline{\text{CACHE}}$, PCD[1]	—	—	—	—	—
$\overline{\text{KEN}}$	—	—	—	—	—
PWT[2]	—	1	—	0	—
WB/$\overline{\text{WT}}$	—	—	0	1	—
Cache Update	no	yes	yes	yes	yes
Write to Memory	writethrough (1 to 8 bytes)	writethrough (1 to 8 bytes)	writethrough (1 to 8 bytes)	writethrough (1 to 8 bytes)	no
State After Write[3]	—	*shared*	*shared*	*exclusive*	*modified*

Notes:

– *Don't care or not applicable.*

1. *The PCD bit is negated and $\overline{\text{CACHE}}$ is asserted during a write hit, but these states do not affect the hit.*
2. *The PWT bit in the page table entry or CR3.*
3. *Transition occurs after any write to memory. Lines in shared MESI state are said to be in writethrough state. Those in exclusive or modified MESI states are said to be in writeback state.*

In single-processor systems with no other caching master, WB/$\overline{\text{WT}}$ is typically tied High. This allows the processor to cache all cacheable reads in the *exclusive* state, and all cacheable writes update only the cache. In systems with multiple caching masters, WB/$\overline{\text{WT}}$ can be generated after inquire cycles to all other caching masters by the logical OR of $\overline{\text{HIT}}$ from all of the masters. This allows the processor to cache reads in the *exclusive* or *modified* state only if no other master has a copy.

While the writeback configuration usually supports higher performance, the writethrough configuration is required for certain transitions in the write-once cache protocol. For details on this protocol, see Section 6.2.6 on page 6-19.

During the Hardware Debug Tool (HDT) mode, WB/$\overline{\text{WT}}$ is only meaningful for cache write misses (PWT = 0 and WB/$\overline{\text{WT}}$ = 1 transition a shared line to an *exclusive* line). The signal is not meaningful during cache read misses in the HDT mode, because the caches are never filled in the HDT mode.

For more details on data-cache MESI state transitions during reads, see Table 5-9 on page 5-52 and Section 6.2.2 on page 6-9.

5.3 Bus Cycle Overview

The bus signals described in the previous section combine to form various types of bus transactions, or *bus cycles*. This section summarizes the general features of the bus cycles: cycle definition, addressing, alignment, and priorities. Section 5.4 describes the signal timing for specific types of bus cycles.

5.3.1 Cycle Definitions

The processor begins driving a bus cycle when it asserts $\overline{\text{ADS}}$. Concurrent with $\overline{\text{ADS}}$, it drives the set of signals indicated in Table 5-19, which define the type of bus cycle. For memory reads, memory writes, burst reads, and burst writes, D/$\overline{\text{C}}$ specifies whether the bus cycle accesses code (instructions) or data. M/$\overline{\text{IO}}$ specifies whether the cycle accesses memory or an I/O port. W/$\overline{\text{R}}$ specifies whether the cycle is a read or write. The assertion of $\overline{\text{CACHE}}$ indicates that the processor is writing or is prepared to read a burst cycle consisting of four consecutive transfers on the data bus. However, for a read, system logic must confirm the burst by asserting $\overline{\text{KEN}}$, or the bus cycle becomes a single-transfer read. I/O accesses are always non-burst cycles.

TABLE 5-19. Bus Cycle Definitions

Type of Cycle	Signals				Comments
	D/$\overline{\text{C}}$	M/$\overline{\text{IO}}$	W/$\overline{\text{R}}$	$\overline{\text{CACHE}}$	
Single-Transfer Memory Read or Write	0 or 1	1	0 or 1	1	—
Single-Transfer I/O Read or Write	1	0	0 or 1	1	—
Burst Memory Read or Write	0 or 1	1	0 or 1	0	For reads, system logic must assert $\overline{\text{KEN}}$ with $\overline{\text{BRDY}}$.
Interrupt Acknowledge	0	0	0	—	Pair of locked cycles.
Special	0	0	1	—	Several special cycles distinguished by $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ and A31–A3. See Table 5-23 on page 5-181.

Interrupt acknowledge operations consist of a locked pair of read cycles. Special bus cycles are further differentiated by the signals shown in Table 5-23 on page 5-181. In addition to the processor-driven bus cycles shown in Table 5-19, system logic can drive inquire cycles to the processor. These bus cycles are described later, in Section 5.4.4 on page 5-157.

The processor samples $\overline{\text{BRDY}}$ during all bus cycles that it drives. The number of $\overline{\text{BRDY}}$s expected by the processor depends on the type of bus cycle, as follows:

- One $\overline{\text{BRDY}}$ for an aligned single-transfer read or write cycle, a special bus cycle, and each of two bus cycles in an interrupt acknowledge operation. One additional $\overline{\text{BRDY}}$ for each misaligned cycle.
- Four $\overline{\text{BRDY}}$s for burst cycles (one $\overline{\text{BRDY}}$ for each of the four transfers). Burst cycles are always aligned.

The last expected $\overline{\text{BRDY}}$ represents the completion of a processor-initiated bus cycle. The processor guarantees at least one idle clock between consecutive bus cycles, whether unlocked or locked. This means that consecutive locked operations, which consist of consecutive bus cycles, also have at least one idle clock between them.

5.3.2 Addressing

The address for a bus cycle is driven on A31–A3 and $\overline{\text{BE7}}$–$\overline{\text{BE0}}$. A31–A3 carry the upper 29 bits of the address, identifying an aligned 8-byte (quadword) region in memory. $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ identify the accessed bytes in that quadword, in effect indicating the three least-significant bits of the address and the size (in bytes) of the desired transfer. For burst and inquire cycles, A31–A5 are sufficient to identify the memory location of the cache line. For burst reads, which are four-transfer cache-line fills, system logic should watch A4–A3 and return the addressed quadword first, before returning the remainder of the cache line.

More details on burst-cycle addressing are given in Section 5.4.3 on page 5-150.

5.3.3 Alignment

For purposes of bus cycles, the term *aligned* means:

- 2- and 4-byte transfers lie within 4-byte address boundaries
- 8-byte transfers lie within 8-byte address boundaries

(For purposes of exceptions, the term *aligned* means situated on the natural boundaries of an instruction or operand. Thus, a 2-byte transfer that crosses a 2-byte address boundary may incur an alignment exception, but it will be performed as an aligned bus cycle.)

If data on D63–D0 are misaligned, the processor generates additional bus cycles to complete the transfer. For example, if a 4-byte transfer begins at address x07h, one byte will be transferred during the first bus cycle and the remaining three bytes will be transferred during a second bus cycle, which will normally occur immediately after the first bus cycle (unless intervened by an interrupt or bus backoff). If the misaligned transfer is run as a locked cycle, the processor asserts both $\overline{\text{LOCK}}$ and SCYC throughout the misaligned sequence of bus cycles.

If memory reads, memory writes, or I/O reads are misaligned, the AMD5$_K$86 processor runs the bus cycles in the opposite order of the Pentium processor. The AMD5$_K$86 processor transfers the least-significant bytes first followed by the most-significant bytes. I/O writes, however, are performed in the same order on both processors: the most-significant bytes first, followed by the least-significant bytes.

For a misaligned CMPXCHG8B operation (that is, the operand does not lie on an 8-byte quadword boundary), the AMD5$_K$86 processor does two split-cycle reads followed by two split-cycle writes, all with $\overline{\text{LOCK}}$ asserted, for a total of eight bus cycles. The Pentium processor combines the cycles for a maximum of four bus cycles.

5.3.4 Bus Speed and Typical DRAM Timing

The processor can be configured for external bus (CLK) speeds of 50, 60, or 66 MHz. Main DRAM memory can be built from Page-mode or EDO (extended data out) DRAM, although faster memory devices can be used for higher performance.

On a 66-MHz bus, the read cycle time for a DRAM-page hit in EDO DRAM is 7-2-2-2 (7 clocks for the first transfer and 2 clocks for each remaining transfer) and 10-2-2-2 for a DRAM-page miss. The read cycle time for a DRAM-page hit in Page-mode DRAM at 66 MHz is 7-4-4-4 and 10-4-4-4 for a DRAM-page miss. On a 50-MHz bus, there is no change in timing for EDO DRAM, but Page-mode DRAM timing becomes 6-3-3-3 for a DRAM-page hit and 8-3-3-3 for a DRAM-page miss.

5.3.5 Bus-Cycle Priorities

The AMD5$_K$86 processor can support only one on-going bus cycle at a time—pending bus cycles are not buffered. System logic maintains the ultimate control over the bus. The processor asserts BREQ to request control of the bus. System logic asserts AHOLD, $\overline{\text{BOFF}}$, or HOLD to take control of the bus. AHOLD passes control of the address bus to system logic for use in inquire cycles, but permits completion of in-progress cycles on the data bus. $\overline{\text{BOFF}}$ forces an in-progress bus cycle to abort and passes control to system logic. HOLD allows an in-progress bus cycle to complete before passing control to system logic.

5.4 Bus Cycle Timing

The following sections describe and illustrate the timing and relationship of bus signals during various types of bus cycles. Only a representative set of bus cycles are illustrated. Many more combinations are possible.

5.4.1 Timing Diagrams

The timing diagrams show the signals on the external bus as a function of time, as measured by the bus clock (CLK). Throughout this chapter, the term *clock* refers to bus-clock cycles, not processor-clock cycles, and the term *cycle* refers to bus cycles not clocks. A clock extends from one rising CLK edge to the next rising CLK edge. The processor samples and drives most signals relative to the rising edge of CLK. The exceptions to this rule include:

- *$\overline{\text{FLUSH}}$ and $\overline{\text{SMI}}$*—Sampled on the falling edge of CLK
- *BF, $\overline{\text{FLUSH}}$, $\overline{\text{FRCMC}}$, and INIT*—Sampled on the falling edge of RESET
- *TDI, TDO, TMS and $\overline{\text{TRST}}$*—Sampled relative TCK

For each signal in the timing diagrams, the High level represents 1, the Low level represents 0, and the middle level represents the floating (high-impedance) state. When both the High and Low levels are shown, the meaning depends on the signal. For a single signal, it means *don't care*. For a bus, it means that the processor or system logic is driving a value, but this value may or may not be valid (for example, the value on the address bus is valid only during the assertion of $\overline{\text{ADS}}$, although addresses are also driven on the bus at other times).

The value indicated for the address bus represents the value driven on lines A31–A3. This value, multiplied by 8, is the byte address of an 8-byte region in memory. The value for $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ indicates which bytes in that region are to be transferred: the bytes corresponding to the zeros on $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ are transferred.

The timing diagrams given in the following sections assume that the current privilege level (CPL) is always 0.

5.4.2 Single-Transfer Reads and Writes

The single-transfer memory and I/O bus cycles transfer 1, 2, 4, or 8 bytes. Misaligned instructions or operands result in a *split cycle*, which requires multiple transactions on the bus. During single-transfer (non-cacheable) code fetches, the AMD5$_K$86 and Pentium processors read 8 bytes, not 16 bytes as the 486 processor does.

Single-Transfer Memory Read and Write

Figure 5-2 shows a single-transfer doubleword code fetch (read) from memory, followed immediately by a single-transfer doubleword write to memory. For the memory-read cycle, the processor drives A31–A3, $\overline{BE7}$–$\overline{BE0}$ (with AP for parity check), D/$\overline{C}$, W/$\overline{R}$, and M/$\overline{IO}$. Then, somewhat later, it asserts $\overline{ADS}$ and BREQ. $\overline{ADS}$, which is held asserted for only one clock, validates the bus cycle. The processor then waits for system logic to return the data on D63–D0 (with DP7–DP0 for parity check) and assert $\overline{BRDY}$. System logic can return $\overline{BRDY}$ as early as one clock after $\overline{ADS}$, thus supporting very fast memory devices.

During the read cycle, the processor drives PCD, PWT, and $\overline{CACHE}$ to indicate its caching and cache-coherency intent for the access. System logic returns $\overline{KEN}$ and WB/$\overline{WT}$ to either confirm or change this intent. In this example, the processor asserts PCD and negates $\overline{CACHE}$, so the accesses are non-cacheable, even though system logic asserts $\overline{KEN}$ during the $\overline{BRDY}$s to indicate its support for cacheability. The processor (which drives $\overline{CACHE}$) and system logic (which drives $\overline{KEN}$) must agree in order for an access to be cacheable. They must also agree among PWT and WB/$\overline{WT}$ in order for a cacheable line to be cached in the writeback state.

The processor can drive another cycle (in this example, a write cycle) as early as two clocks after the assertion of $\overline{BRDY}$. A dead (or idle) clock is thus guaranteed between any two bus cycles. As in the read cycle, neither the address nor the cycle-definition signals are valid until the processor asserts $\overline{ADS}$, and the value driven on A31–A3 is valid only during the assertion of $\overline{ADS}$.

This example shows a parity error during the read cycle, as indicated by the processor's assertion of $\overline{PCHK}$ two clocks after $\overline{BRDY}$. Because system logic asserts $\overline{PEN}$ during the

$\overline{\text{BRDY}}$, the processor latches the physical address and cycle definition of the failed bus cycle in its 64-bit machine-check address register (MCAR) and its 64-bit machine-check type register (MCTR). For details on such parity errors, see the descriptions of $\overline{\text{PCHK}}$ and $\overline{\text{PEN}}$ on pages 5-102 and 5-103.

While Figure 5-2 shows $\overline{\text{BRDY}}$ returned in the next clock after $\overline{\text{ADS}}$, most DRAM-based systems add wait states (idle clocks) between $\overline{\text{ADS}}$ and $\overline{\text{BRDY}}$, as described in Section 5.3.4 on page 5-140.

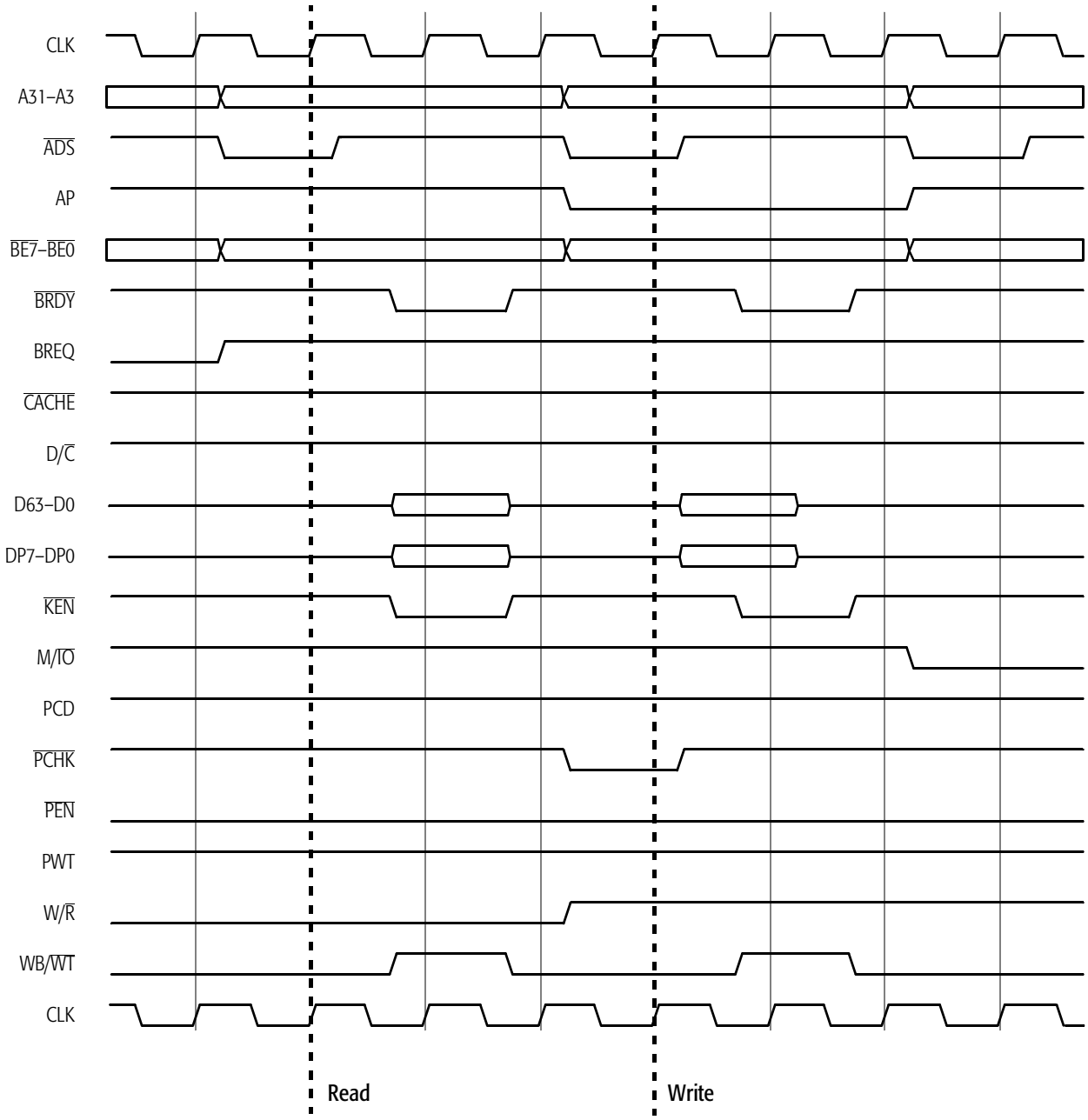


FIGURE 5-2. Single-Transfer Memory Read and Write

Single-Transfer Memory Write Delayed by EWBE Signal

Figure 5-3 shows two consecutive memory writes. The first write fills an external write buffer and the second write is stalled for three clocks by the negation of EWBE.

For writes, system logic can store the address and data in a write buffer, return BRDY, and perform the store to memory later. If the number of outstanding writes exceeds the size of the write buffer, system logic must negate EWBE to prevent the processor from sending additional writes until EWBE is asserted. The advantage of negating EWBE as opposed to not asserting BRDY is that negating EWBE prevents only write requests, but not asserting BRDY stalls the bus and prevents all requests.

More specifically, if EWBE is negated with or after the last BRDY of a write cycle, the processor will not do any of the following:

- Write a store-buffer entry to the data cache
- Write to memory (single-transfer or burst), including locked write to Accessed (A) bit after TLB load
- Write to I/O (OUTx)
- Execute the following instructions:

  MOV to CR0

  MOV to CR4, including during a task switch

  WBINVD

  INVLPG

  CPUID
- Respond to the following instructions:

  FLUSH

  SMI
- Respond to any other interrupts or exceptions that cause a write to memory, such as pushing state onto the stack or setting the Accessed bit in a segment descriptor. This may include the BUSCHK, NMI, and INTR interrupts.

For more details, see the description of EWBE on page 5-63.

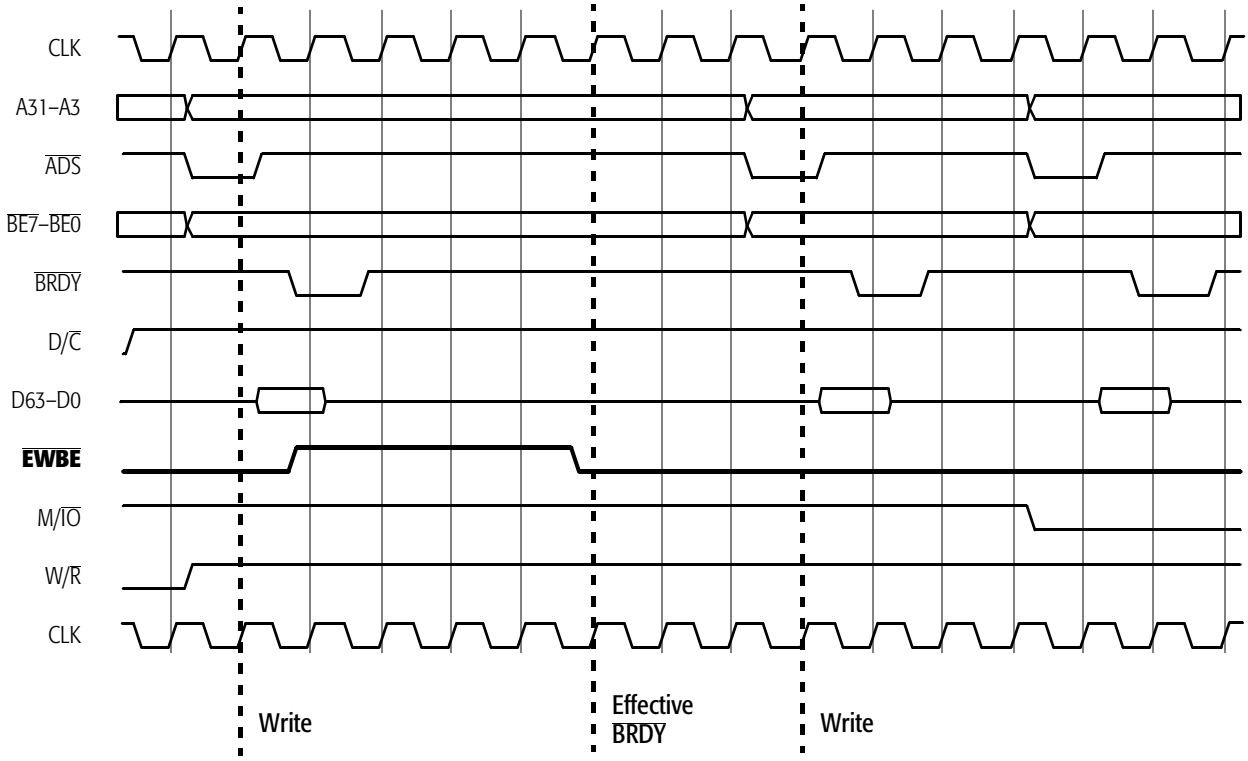


FIGURE 5-3. Single-Transfer Memory Write Delayed by EWBE Signal

I/O Read and Write

Figure 5-4 shows an I/O read followed by an I/O write. The processor accesses I/O when it executes an I/O instruction (any of the INx or OUTx instructions). Accesses to memory-mapped I/O ports appear on the bus as accesses to memory rather than to the I/O address space.

The I/O-cycle protocol is nearly the same as the protocol for read and write accesses to memory, shown in Figure 5-2, except that M/$\overline{IO}$ = 0. Only data (not code) can be read or written from the I/O address space. The cycle definition for an I/O code read (D/$\overline{C}$ = 0, M/$\overline{IO}$ = 0, W/$\overline{R}$ = 0) defines an interrupt acknowledge cycle, and the cycle-definition for an I/O code write (D/$\overline{C}$ = 0, M/$\overline{IO}$ = 0, W/$\overline{R}$ = 1) defines a special bus cycle.

The example in Figure 5-4 shows a single wait state separating $\overline{ADS}$ and $\overline{BRDY}$ for the read. In actual systems, however, the time will typically be longer.

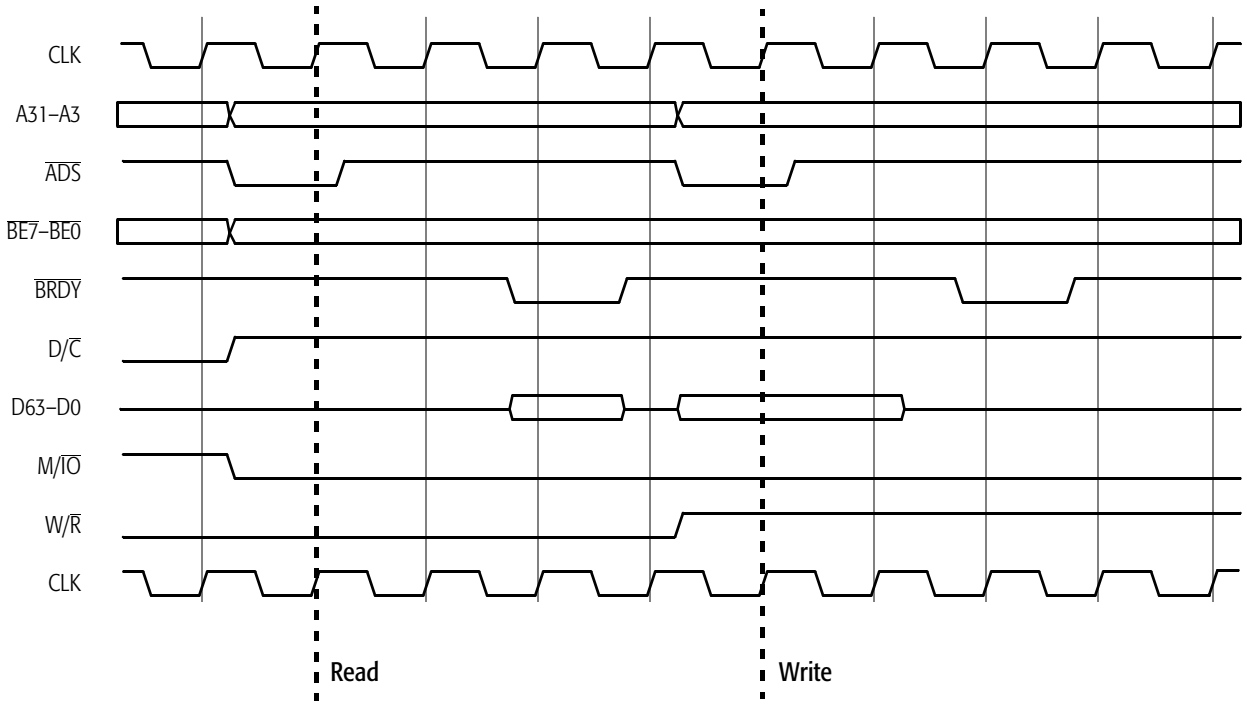


FIGURE 5-4. I/O Read and Write

Single-Transfer Misaligned Memory and I/O Transfers

Figure 5-5 shows a misaligned (split) memory read followed by a misaligned I/O write. (For a definition of *misaligned*, see Section 5.3.3 on page 5-139.) When the processor encounters a misaligned access, it determines the appropriate pair of bus cycles—each with its own $\overline{\text{ADS}}$ and $\overline{\text{BRDY}}$—required to complete the access.

In this example, the first pair of bus cycles represents a memory read of the doubleword at 800Eh. This access crosses a doubleword boundary, so it is misaligned. The processor first reads the word at 800Eh, followed by the word at 8010h. The second pair of bus cycles represents a write of a doubleword to I/O address 8Eh. This transfer also crosses a doubleword boundary, so it is misaligned. The processor writes the word to I/O address 90h, followed by the word to I/O address 8Eh.

The AMD5$_K$86 processor performs misaligned memory read, memory write, and I/O read transfers in the reverse order of the Pentium processor, but misaligned I/O write transfers are performed in the same order on both processors. Table 5-20 shows the order. Thus, in this example, the I/O write accesses the most-significant bytes first followed by the least-significant bytes, the opposite order from the memory accesses and I/O reads.

TABLE 5-20. Bus-Cycle Order During Misaligned Transfers

Type of Access	First K5 Cycle	Second K5 Cycle	Pentium Compatible?
Memory Read	LSBs	MSBs	no
Memory Write	LSBs	MSBs	no
I/O Read	LSBs	MSBs	no
I/O Write	MSBs	LSBs	yes

The SCYC (Split Cycle) output has no meaning in unlocked misaligned transfers. It is only meaningful in locked misaligned transfers.

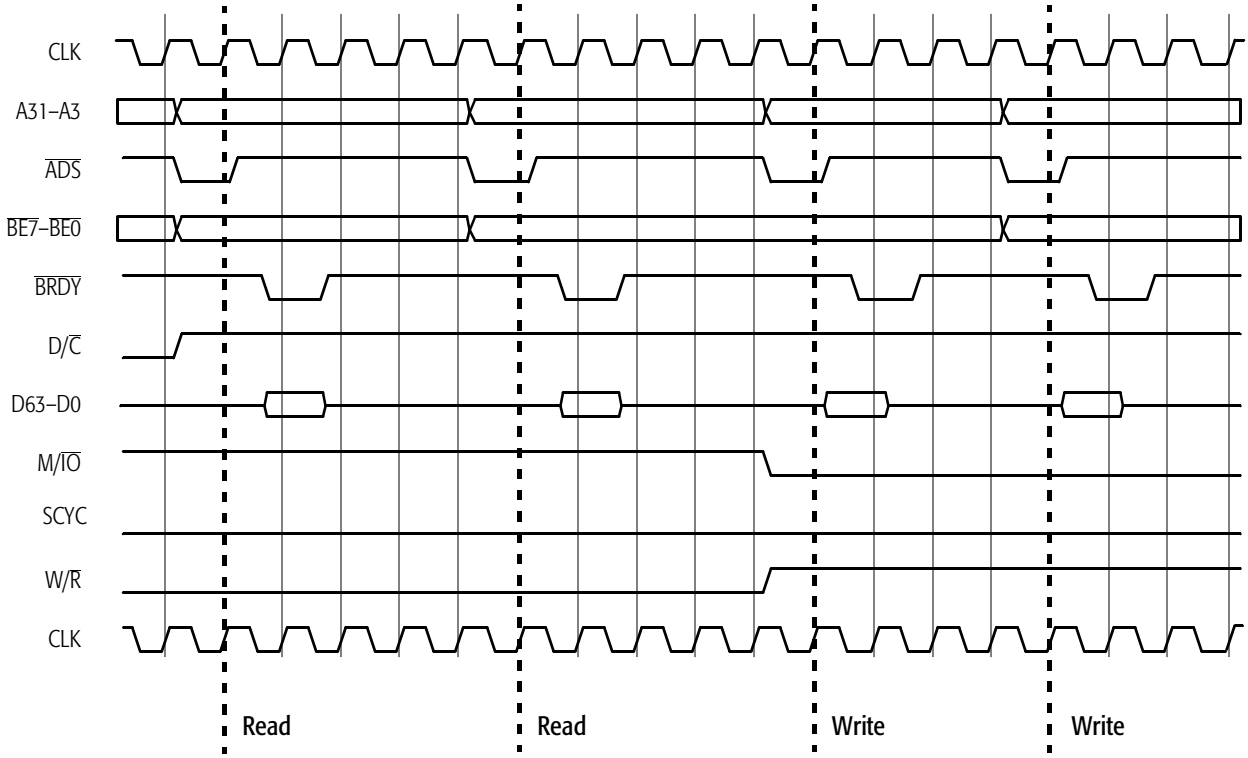


FIGURE 5-5. Single-Transfer Misaligned Memory and I/O Transfers

5.4.3 Burst Cycles

The processor drives burst cycles, which consist of four sequential eight-byte (quadword) transfers on the data bus, only in the following cases:

- *Burst Read*—Cache-line fills from memory. These burst reads occur when the processor asserts $\overline{\text{CACHE}}$ during $\overline{\text{ADS}}$ and system logic asserts $\overline{\text{KEN}}$ during the first $\overline{\text{BRDY}}$ of a read cycle.
- *Burst Write*—Writebacks to memory of *modified* cache lines. Writebacks can be caused by (a) externally initiated inquire cycles or $\overline{\text{FLUSH}}$ operations, (b) processor-initiated internal snoops or cache-line replacements, or (c) program-initiated WBINVD instructions.

Writethroughs to memory, which occur in response to write misses or write hits to *shared* cache lines, are driven as single-transfer bus cycles.

Burst Read

Figure 5-6 shows two consecutive burst reads. During burst reads ($\overline{\text{CACHE}}$ and $\overline{\text{KEN}}$ both asserted with the first $\overline{\text{BRDY}}$ of a memory read), the processor drives $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ with $\overline{\text{ADS}}$ to identify the bytes of the desired instruction or operand. The processor drives $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ with the desired bytes at that time because it does not yet know whether the read will be a single-transfer or a burst—this depends on how system logic drives $\overline{\text{KEN}}$ with the first $\overline{\text{BRDY}}$. If system logic negates $\overline{\text{KEN}}$ it must return, as a single transfer, only the bytes specified on $\overline{\text{BE7}}$–$\overline{\text{BE0}}$. If system logic asserts $\overline{\text{KEN}}$, it must ignore $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ during all transfers of the burst and return all eight bytes for the starting address on A31–A3. $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ does not change during the four transfers of the burst. (This behavior is unlike the 486 processor, which drives $\overline{\text{BE3}}$–$\overline{\text{BE0}}$ separately for each transfer of a burst.) System logic must determine the successive quadword addresses for each transfer in a burst, depending on the starting address, as shown in Table 5-21.

TABLE 5-21. Address-Generation Sequence During Bursts

Address Driven By Processor on A31–A3	Addresses of Subsequent Quadwords[1] Generated By System Logic		
Quadword 1	Quadword 2	Quadword 3	Quadword 4
...00h	...08h	...10h	...18h
...08h	...00h	...18h	...10h
...10h	...18h	...00h	...08h
...18h	...10h	...08h	...00h

Notes:
1. quadword = 8 bytes

In the clock after $\overline{\text{ADS}}$, the processor drives the first of four sequential eight-byte (quadword) transfers on the data bus. The processor holds the first transfer on the bus until system logic returns $\overline{\text{BRDY}}$, then it transfers the next quadword. In this example, system logic returns $\overline{\text{BRDY}}$ with no wait states, and the processor responds by driving the subsequent quadword in the next clock. Typical systems, however, add one or more wait states between the transfers.

For both read cycles, the processor asserts $\overline{\text{CACHE}}$ with $\overline{\text{ADS}}$ and system logic asserts $\overline{\text{KEN}}$ with the $\overline{\text{BRDY}}$ of the first transfer. Thus, $\overline{\text{CACHE}}$ and $\overline{\text{KEN}}$ agree, and the access is cached. This agreement between $\overline{\text{CACHE}}$ and $\overline{\text{KEN}}$ is required in order for a burst read to occur. The processor only drives burst reads if the access is cacheable. If either $\overline{\text{CACHE}}$ or $\overline{\text{KEN}}$ were negated during the $\overline{\text{BRDY}}$ of the first transfer, the read would terminate with the first quadword transfer, thus becoming a single-transfer read.

In this example, the processor negates PWT (indicating write-back state) and system logic drives WB/$\overline{\text{WT}}$ High with the $\overline{\text{BRDY}}$ of the first transfer (also indicating writeback state). Thus, PWT and WB/$\overline{\text{WT}}$ agree, and the cache line becomes a writeback line, which is cached in the *exclusive* MESI state. Details on the writeback/writethrough and MESI cache-coherency state transitions are given in Table 2-2 on page 2-19.

In Figure 5-7, the two consecutive burst reads are identical to those in Figure 5-6, except that system logic asserts $\overline{\text{NA}}$ one clock before it asserts $\overline{\text{BRDY}}$ in the first read cycle of Figure 5-7. This causes $\overline{\text{KEN}}$ and WB/$\overline{\text{WT}}$ to be effective when $\overline{\text{NA}}$

(rather than $\overline{\text{BRDY}}$) is asserted. $\overline{\text{KEN}}$ and WB/$\overline{\text{WT}}$ are validated by either $\overline{\text{NA}}$ or $\overline{\text{BRDY}}$, whichever comes first. $\overline{\text{NA}}$ will not generate a pipelined cycle in the event that there are no pending internal cycles.

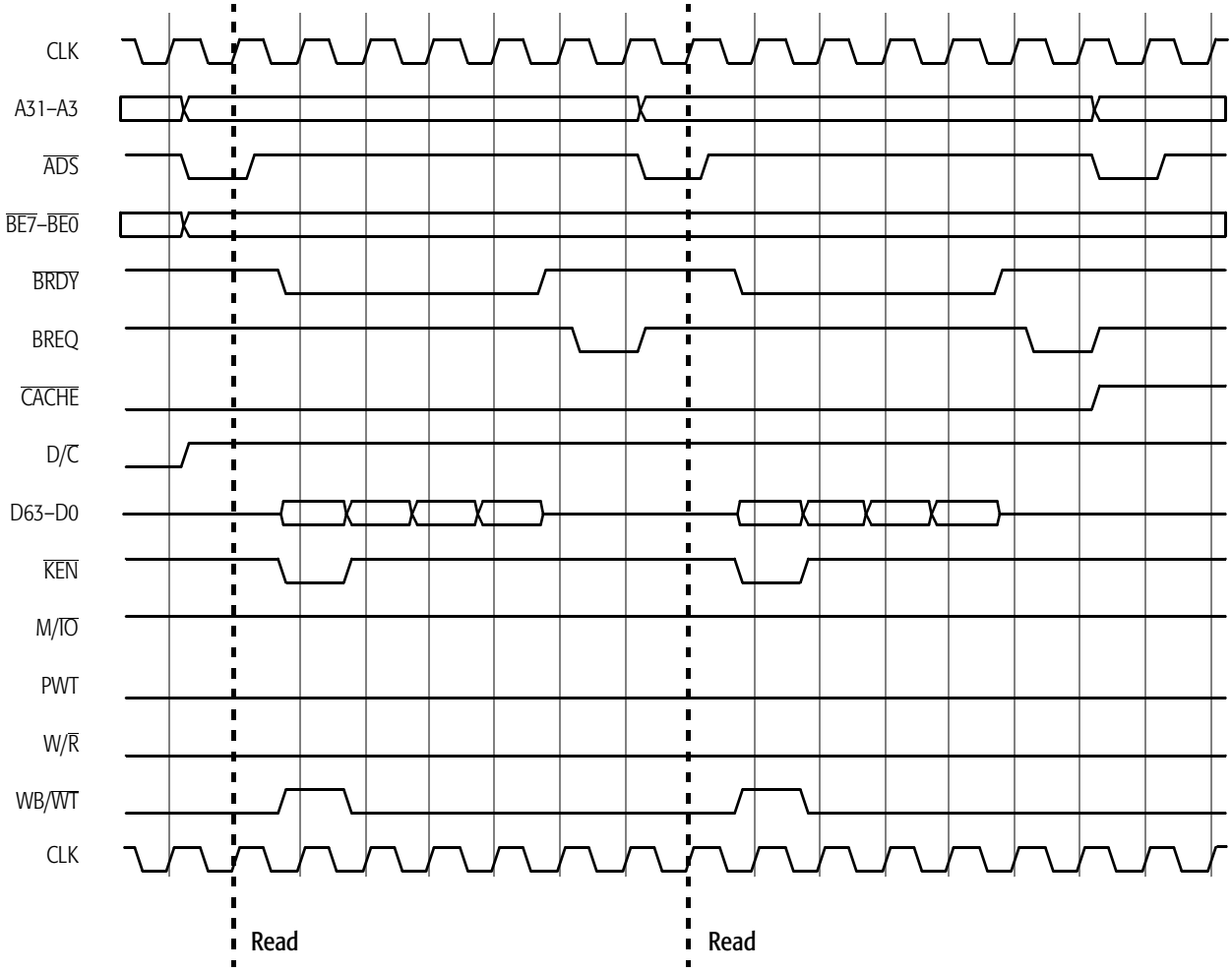


FIGURE 5-6. Burst Reads

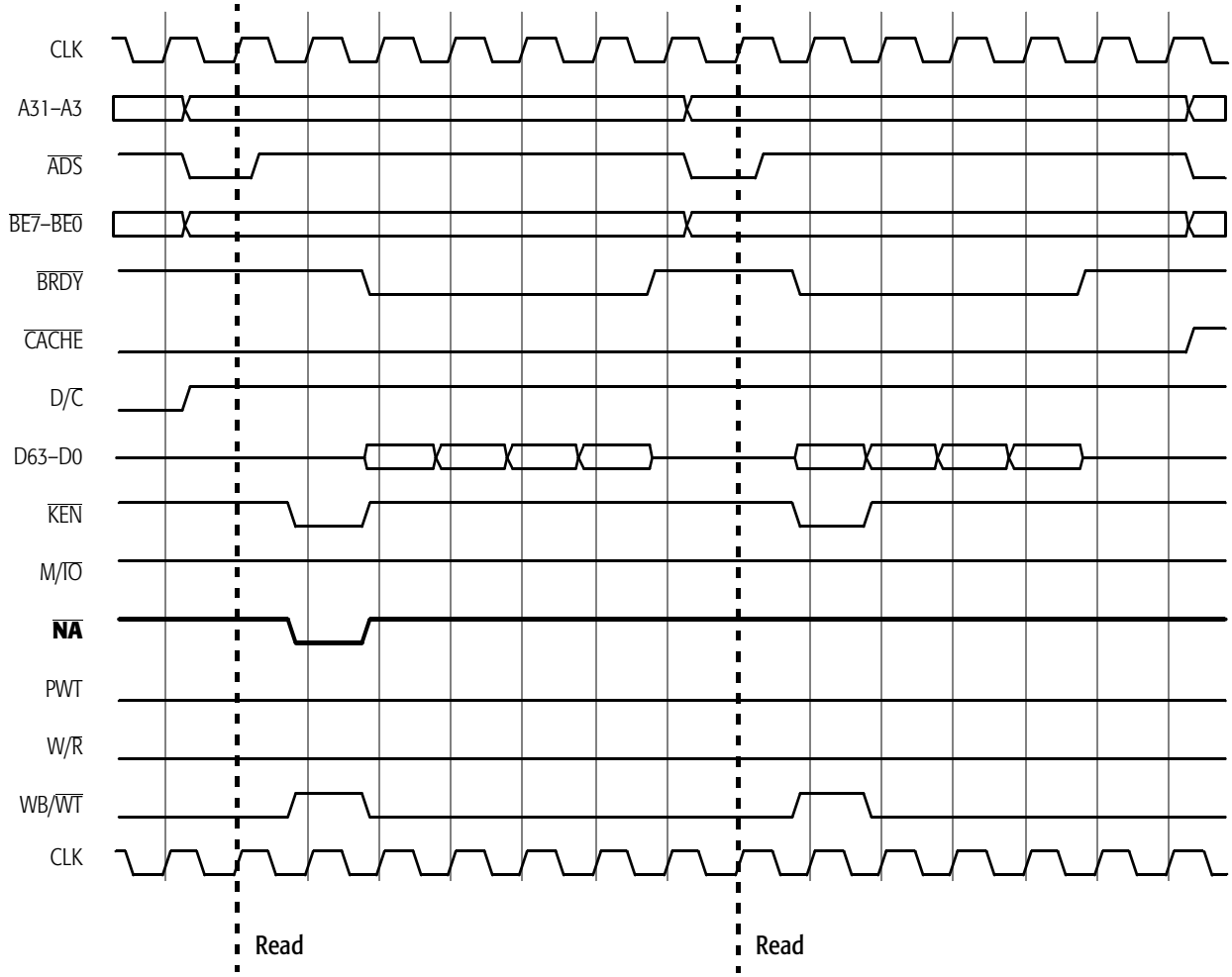


FIGURE 5-7. Burst Read (NA Sampled)

Burst Writeback

Figure 5-8 shows a burst read followed by a writeback. Writebacks are the only type of burst write that the processor performs. They can be initiated by the processor or by system logic in the following cases:

- Processor-Initiated Writebacks:
  - *Replacement*—If a cache-line fill is initiated when all four ways of the cache that could accommodate the incoming line are filled with valid entries, the processor uses a round-robin algorithm to select a line for replacement. Before a replacement is made to a data cache line in the *modified* state, the line is written back to memory.
  - *Internal Snoop*—The processor snoops the data cache whenever an instruction-cache line is read, and it snoops the instruction cache whenever a data cache line is written. This snooping is performed to determine whether the same address is stored in both caches, a situation that is taken to imply the occurrence of self-modifying code. If a snoop hits a data cache line in the *modified* state, the line is written back to memory before being invalidated.
  - *WBINVD Instruction*—When the processor executes a WBINVD instruction, it writes back all modified lines in the data cache and then invalidates all lines in both caches. The action taken in response to the WBINVD instruction is essentially the same as the action taken in response to the $\overline{\text{FLUSH}}$ input signal, except that the acknowledge cycles differ. For details, see page 5-186.
- System-Initiated Writebacks:
  - *Inquire Cycle Hits*—If an inquire cycle hits a *modified* line in the data cache, the processor writes back the line. For details, see page 5-158.
  - *$\overline{\text{FLUSH}}$*—If system logic asserts the $\overline{\text{FLUSH}}$ input, the entire contents of the data cache are written back to memory before the entire contents of both caches are invalidated. The action taken in response to the $\overline{\text{FLUSH}}$ input signal is essentially the same as the action taken in response to the WBINVD instruction, except that the acknowledge cycles differ. For details, see page 5-184.

During all processor-initiated and system-initiated $\overline{\text{FLUSH}}$ writebacks, the processor asserts $\overline{\text{ADS}}$, drives a 32-byte-aligned starting address on A31–A3, and enables all eight bytes

($\overline{BE7}$–$\overline{BE0}$ = 00h). Thus, A4–A3 are always 0 for writebacks. During inquire cycle writebacks, the processor does the same thing, except that if system logic holds AHOLD asserted throughout the writeback, the processor lets system logic provide the address.

The writeback shown in Figure 5-8 is caused by a cache-line replacement, which occurs when an attempted burst read finds that all four cache ways for that address are filled with valid entries. In this case, the processor performs the following sequence:

1. Copies the prior contents of the replacement line to its 32-byte writeback buffer (described in Section 2.3.7 on page 2-23). This is not visible on the bus.
2. Completes the burst read, placing the incoming data into the cache line. This is the first burst cycle in Figure 5-8.
3. Writes the *modified* line back to memory. This is the second burst cycle in Figure 5-8.

During the burst read (Step 2), the states of PWT and WB/$\overline{WT}$ are the same as in Figure 5-6 and Figure 5-7.

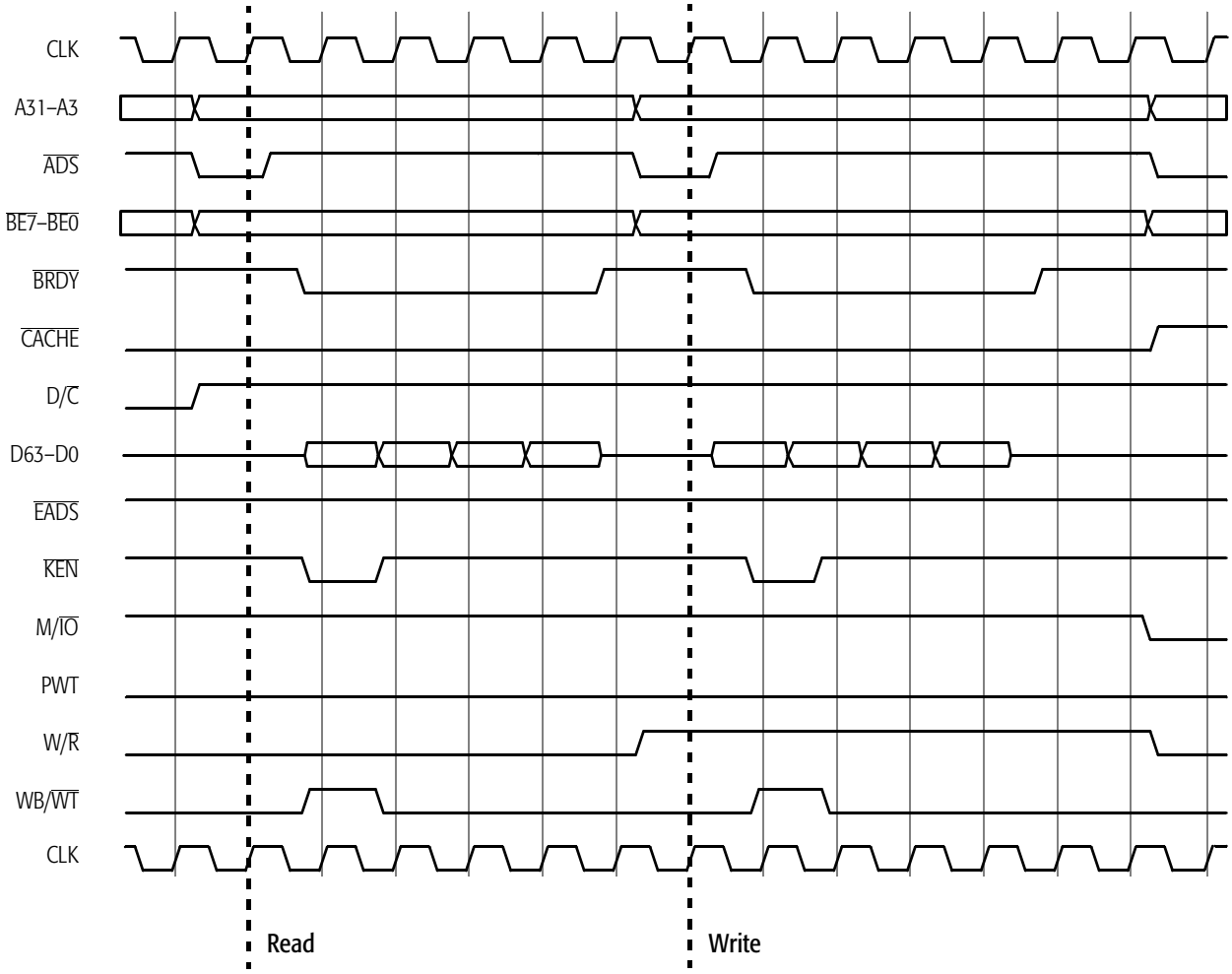


FIGURE 5-8. Burst Writeback Due To Cache-Line Replacement

5.4.4 Bus Arbitration and Inquire Cycles

The processor bus may be required by another bus master, which may need to drive its own cycles on the bus, or by system logic, which may need to drive an inquire cycle to the processor or resolve bus deadlock. One of three signals can be used for these purposes: AHOLD, $\overline{\text{BOFF}}$, or HOLD. AHOLD's sole function is to support inquire cycles. It obtains control only of the address bus and allows another master or system logic to drive only inquire cycles, whereas $\overline{\text{BOFF}}$ and HOLD obtain control of the full bus (address and data), allowing another master to drive not only inquire cycles but also read and write cycles. $\overline{\text{BOFF}}$ provides the fastest access to the bus and it aborts any in-progress cycle by the processor. AHOLD and HOLD both permit an in-progress bus cycle to complete, but a writeback can occur while AHOLD is asserted whereas a pending writeback during the assertion of $\overline{\text{BOFF}}$ or HOLD occurs after the $\overline{\text{BOFF}}$ or HOLD is negated.

In most systems, the choices are between $\overline{\text{BOFF}}$ and AHOLD. Due to its slow response time, HOLD is usually considered only when backward-compatibility with prior-generation subsystems requires it or when the integrity of in-progress bus cycles is of paramount importance. Support for $\overline{\text{BOFF}}$ is usually needed to resolve potential deadlock problems that arise as a result of inquire cycles, and if $\overline{\text{BOFF}}$ is supported, there is usually no reason to support HOLD. The sections that follow further describe these relative advantages and disadvantages.

In systems with multiple caching masters and shared memory, system logic can maintain cache coherency by driving inquire cycles to the processor whenever another bus master accesses shared memory. Such system-initiated bus cycles cause the processor to compare the physical tags for both its instruction and data caches with the inquire address, in parallel with any cache accesses the processor makes via its linear tags. If a match is found, the processor writes the cache line back to memory, if *modified*, and changes the MESI state according to the state of the INV input signal during the inquire cycle.

The system logic's sequence for driving inquire cycles is:

1. Assert AHOLD to obtain control of the address bus, or assert either $\overline{\text{BOFF}}$ or HOLD to obtain control of the entire bus.

2. Two clocks after the assertion of $\overline{\text{BOFF}}$ or AHOLD, or one clock after sampling HLDA asserted when HOLD is used, assert $\overline{\text{EADS}}$ while driving a cache-line address on A31–A5, and assert or negate INV. The processor latches the address when it samples $\overline{\text{EADS}}$ asserted.
3. Wait two clocks, watching for $\overline{\text{HITM}}$ and/or $\overline{\text{HIT}}$ to be asserted:

   If neither $\overline{\text{HIT}}$ nor $\overline{\text{HITM}}$ are asserted at the end of two clocks, or if only $\overline{\text{HIT}}$ is asserted, the inquire cycle terminates.

   If $\overline{\text{HITM}}$ is asserted, a writeback follows and the processor does not recognize $\overline{\text{EADS}}$ again until the last $\overline{\text{BRDY}}$ of the writeback. The timing of the writeback depends on whether AHOLD, $\overline{\text{BOFF}}$, or HOLD was asserted to gain access to the bus. If AHOLD was used, the processor begins driving the four-transfer burst writeback as early as two clocks after asserting $\overline{\text{HITM}}$, whether or not AHOLD is still asserted. If $\overline{\text{BOFF}}$ or HOLD was used, the processor delays the writeback until just after $\overline{\text{BOFF}}$ or HLDA is negated.

The resulting state of a cache line that is hit by an inquire cycle depends on the state of the INV signal at the time of the inquire cycle (see Table 5-11 on page 5-73). If INV is negated, the line remains in or transitions to the *shared* state. If INV is asserted, the line is written back, if *modified*, and transitions to the *invalid* state.

AHOLD-Initiated Inquire Miss

Figure 5-9 shows a burst read, during which system logic asserts AHOLD to acquire the address bus for an inquire cycle. The processor floats the address bus one clock after AHOLD is asserted, although the data bus continues to return data from the in-progress burst read. (The processor supports only one in-progress bus cycle. No pending bus cycles are buffered.) Two clocks after asserting AHOLD, system logic initiates the inquire cycle by asserting $\overline{\text{EADS}}$, driving INV (negated in this example), and driving the inquire address on A31–A5.

Although the inquire cycle misses the cache ($\overline{\text{HIT}}$ is negated two clocks after $\overline{\text{EADS}}$), the processor's assertion of $\overline{\text{APCHK}}$ two clocks after $\overline{\text{EADS}}$ indicates that a parity error occurred on the inquire cycle address. Because of this parity error, system logic should disregard the result of the inquire cycle and perform it again.

For an AHOLD inquire cycle to be recognized, AHOLD must have been asserted continuously for two clocks at the time $\overline{\text{EADS}}$ is asserted. AHOLD and $\overline{\text{BOFF}}$ can be asserted in conjunction with each other without interfering with $\overline{\text{EADS}}$ recognition, as long as the sampling criteria for at least one of the signals (AHOLD or $\overline{\text{BOFF}}$) is met.

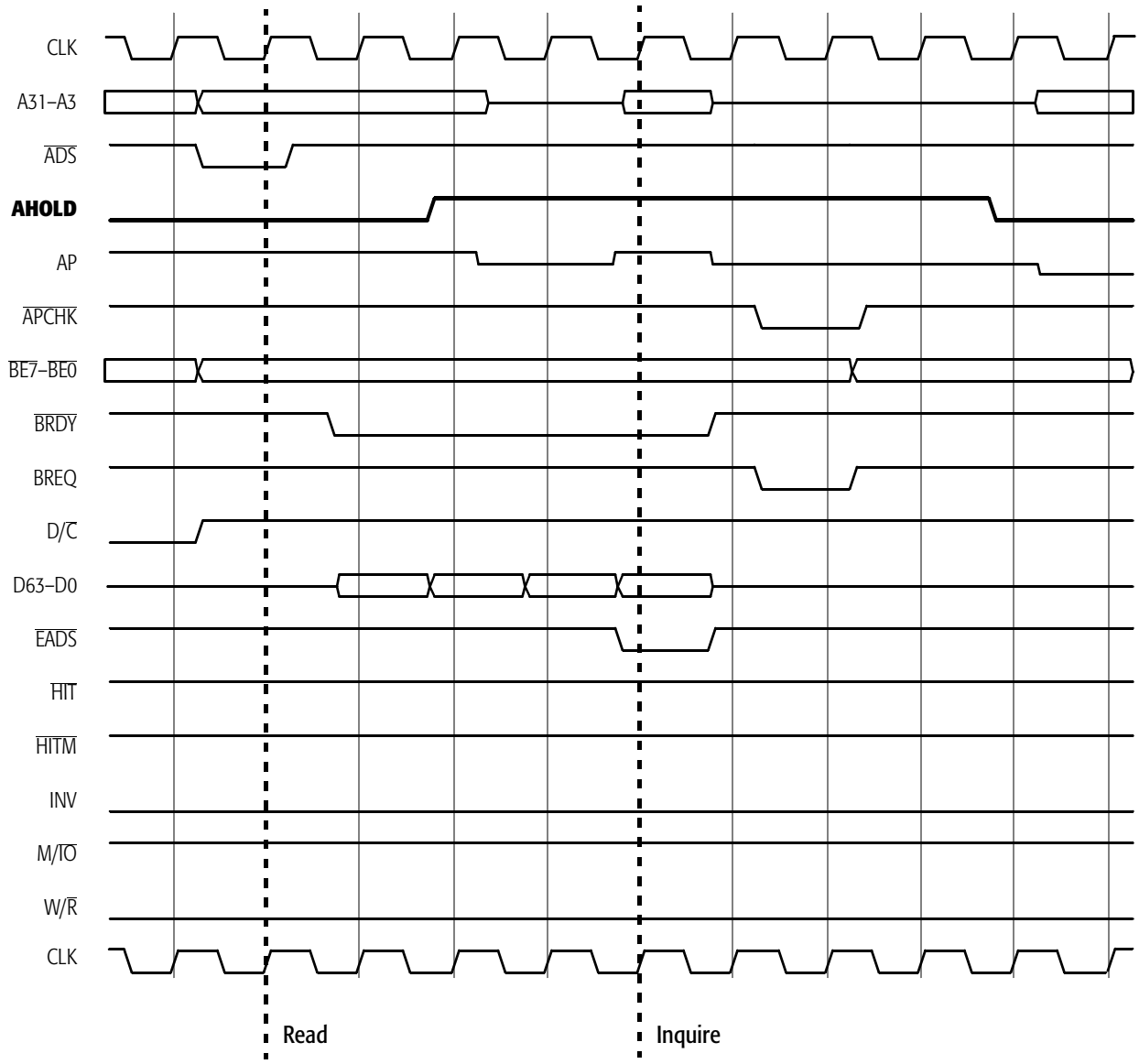


FIGURE 5-9. AHOLD-Initiated Inquire Miss

AHOLD-Initiated Inquire Hit to Shared or Exclusive Line

Figure 5-10 shows an example similar to Figure 5-9, minus the address parity error, but this inquire cycle hits either a *shared* or *exclusive* line in the cache, as indicated by the assertion of HIT and the negation of HITM two clocks after the assertion of EADS. The processor invalidates the cache line because system logic asserts INV with EADS. The processor may drive a new bus cycle as early as one clock after system logic negates AHOLD.

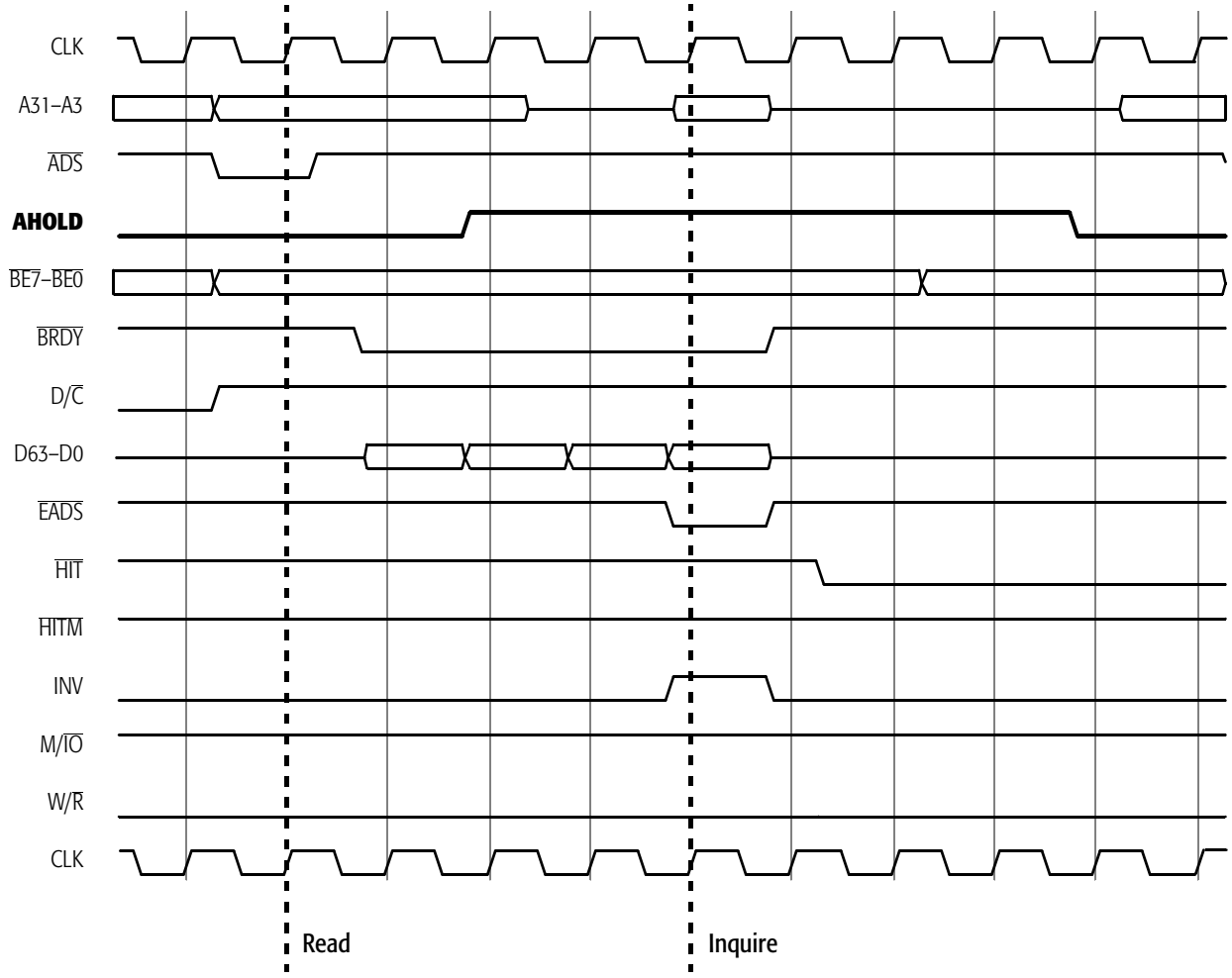


FIGURE 5-10. AHOLD-Initiated Inquire Hit to Shared or Exclusive Line

AHOLD-Initiated Inquire Hit to Modified Line

Figure 5-11 shows the same sequence as in Figure 5-10, but this time the inquire cycle hits a *modified* line. As in Figure 5-10, system logic asserts INV with $\overline{\text{EADS}}$. Two clocks later, the processor asserts both $\overline{\text{HIT}}$ and $\overline{\text{HITM}}$. A few clocks later the processor drives a writeback for the cache line and then invalidates its cached copy. The processor holds $\overline{\text{HITM}}$ asserted until one clock after the last $\overline{\text{BRDY}}$ of the writeback.

If system logic holds AHOLD asserted throughout an inquire cycle and any required writeback, system logic must latch the inquire cycle address when it asserts $\overline{\text{EADS}}$. This is required so that, if the inquire cycle hits a *modified* line, the address used for the writeback need not be driven by the processor when the processor asserts $\overline{\text{ADS}}$ for the writeback. Instead, A31–A5 remains an input-only bus and system logic must use its latched copy of the inquire cycle address. By contrast, if system logic always negates AHOLD before the writeback, the processor drives the writeback address when it asserts $\overline{\text{ADS}}$ for the writeback, and system logic need not retain a copy of the inquire cycle address. While the processor drives the writeback address, it drives only the beginning address for the 32-byte transfer on A31–A5. System logic must determine the remaining addresses as shown in Table 5-21 on page 5-151.

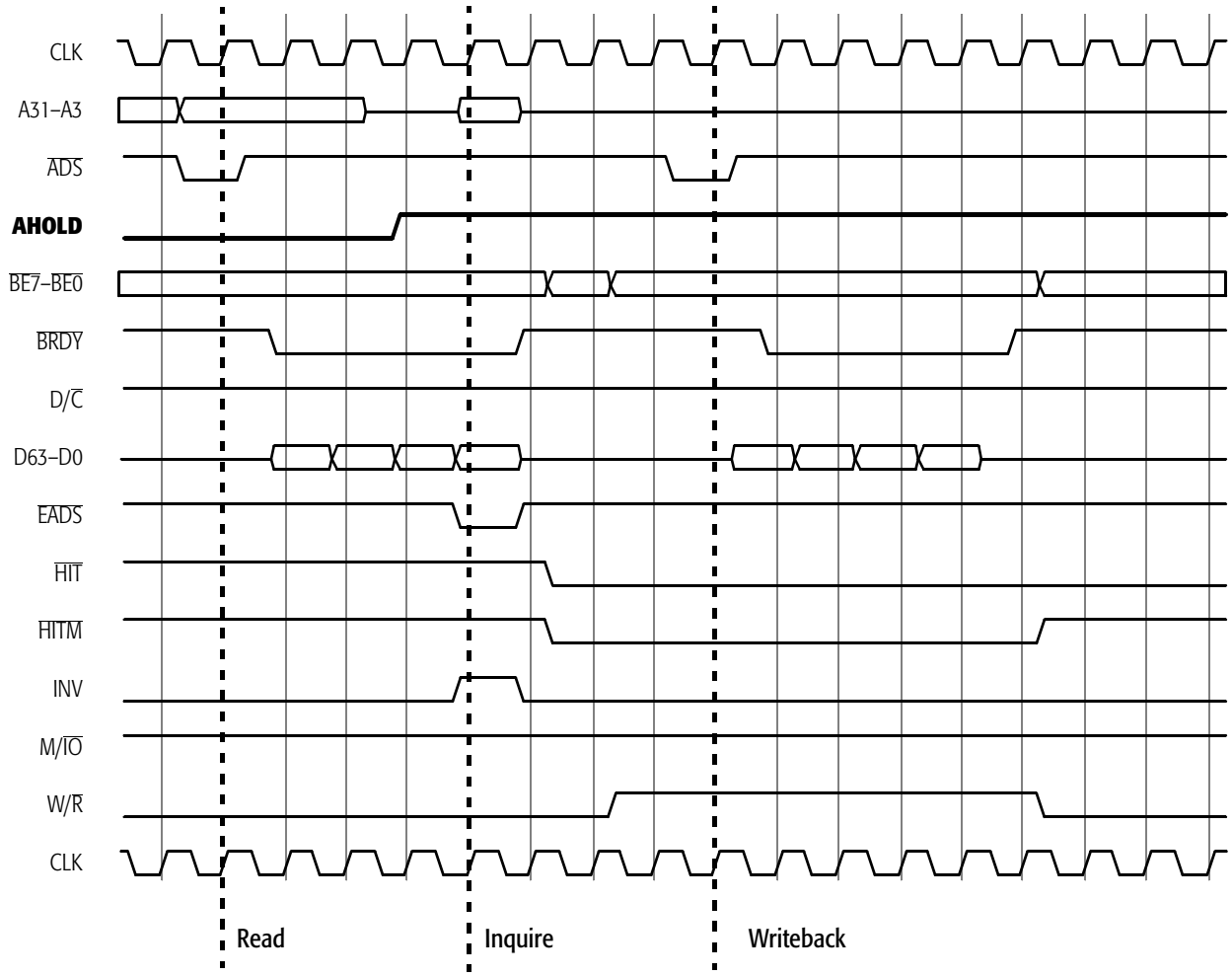


FIGURE 5-11. AHOLD-Initiated Inquire Hit to Modified Line

Bus Backoff (BOFF)

BOFF provides the fastest response of the three bus-hold inputs. Unlike AHOLD and HOLD, BOFF does not permit an in-progress bus cycle to complete. It forces the processor off the bus in the next clock, aborting any in-progress bus cycle that the processor may have begun.

Figure 5-12 shows a burst read interrupted by BOFF. One clock after sampling BOFF asserted, the processor aborts the entire in-progress burst read and floats its bus. All output and bidirectional signals used for memory or I/O accesses are floated. The processor ignores all data and BRDYs returned by the system during the aborted cycle. This is unlike BOFF on the 486 processor, which retains the data that had been transferred up to the clock in which BOFF was asserted. BOFF has no effect on writes to the processor's store buffer, except to delay them. (The store buffer is situated between the execution units and the data cache and is used for speculative stores, prior to being written in non-speculative state to the data cache.)

Another bus master can begin driving cycles as early as two clocks after BOFF is asserted. System logic or another bus master may continue asserting BOFF for as long as it wants. The processor has no way of breaking the hold. While the processor is backed off, it continues to execute out of its instruction and data caches, if possible. If it can no longer operate out of its caches, it holds BREQ asserted continuously.

As early as one clock after BOFF is negated, the processor restarts—from the beginning—any bus cycle that was aborted when BOFF was asserted. This is unlike BOFF on the 486 processor, which restarts only the transfers that did not complete when BOFF was asserted. The processor may drive another cycle with ADS as early as two clocks after any aborted cycle completes. This allows one idle clock (also called a dead clock) between any two bus cycles. If BOFF was asserted when ADS was also asserted, however, ADS remains Low (floats asserted) after BOFF is negated. In such a case, system logic must properly interpret the state of ADS when it negates BOFF.

Because of its ability to help resolve deadlock problems, BOFF is required in virtually all systems with multiple caching masters. In such designs, system logic typically drives separate BOFF signals to each bus master in the system. See Section 6.2.5 on page 6-14 for system configurations using BOFF.

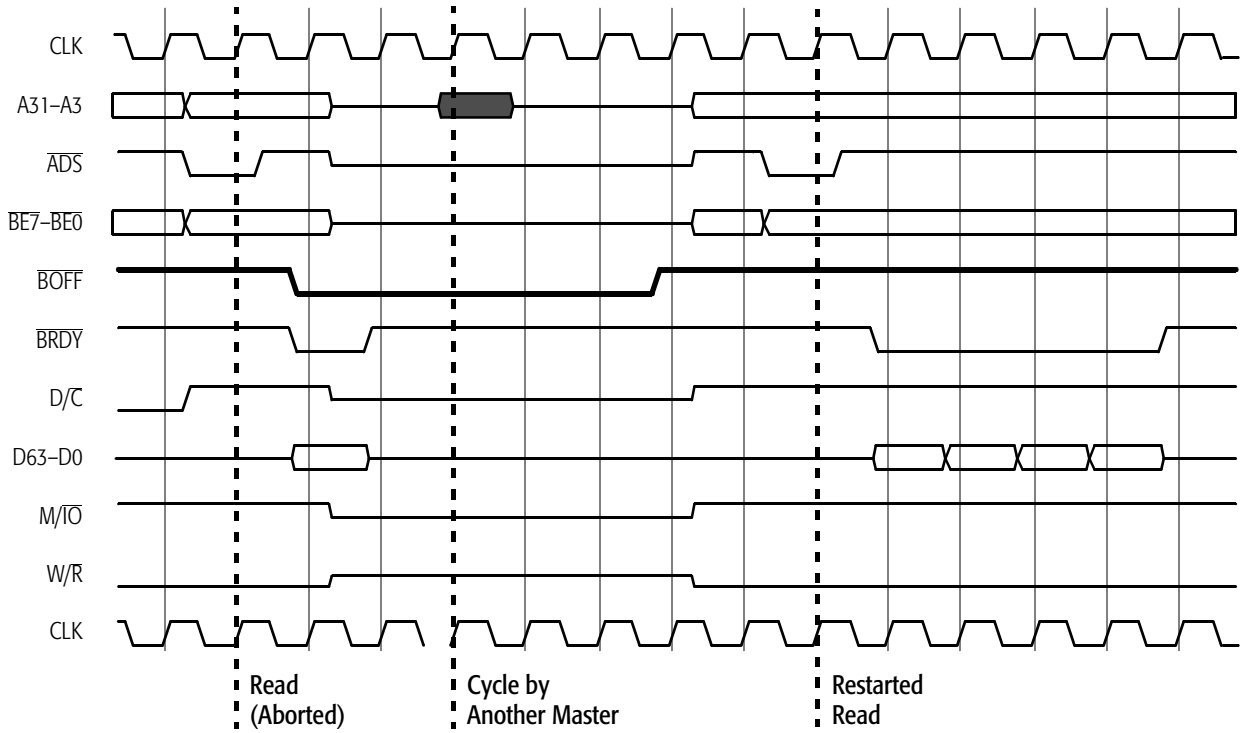


FIGURE 5-12. Basic BOFF Operation

BOFF-Initiated Inquire Hit to Modified Line

Figure 5-13 shows a burst read interrupted by the assertion of BOFF for the purpose of an inquire cycle. One clock after sampling BOFF asserted, the processor aborts the burst read and floats its bus. Two clocks after asserting BOFF, system logic initiates the inquire cycle by asserting EADS and INV, and driving the inquire address on A31–A5. The processor asserts both HIT and HITM two clocks after EADS, thus indicating that the inquire hit a *modified* cache line. The writeback cannot occur while BOFF is asserted, however, because the processor has floated its data and control outputs.

After BOFF is negated, the processor writes back the *modified* cache line, holding HITM asserted until one clock after the last BRDY of the writeback. Because INV was asserted with EADS, the cache line is invalidated after its writeback. Then, the processor restarts—from the beginning—the aborted burst read.

For a BOFF inquire cycle to be recognized, BOFF must have been asserted continuously for two clocks at the time EADS is asserted. AHOLD and BOFF can be asserted in conjunction with each other without interfering with EADS recognition, as long as the sampling criteria for at least one of the signals (AHOLD or BOFF) is met.

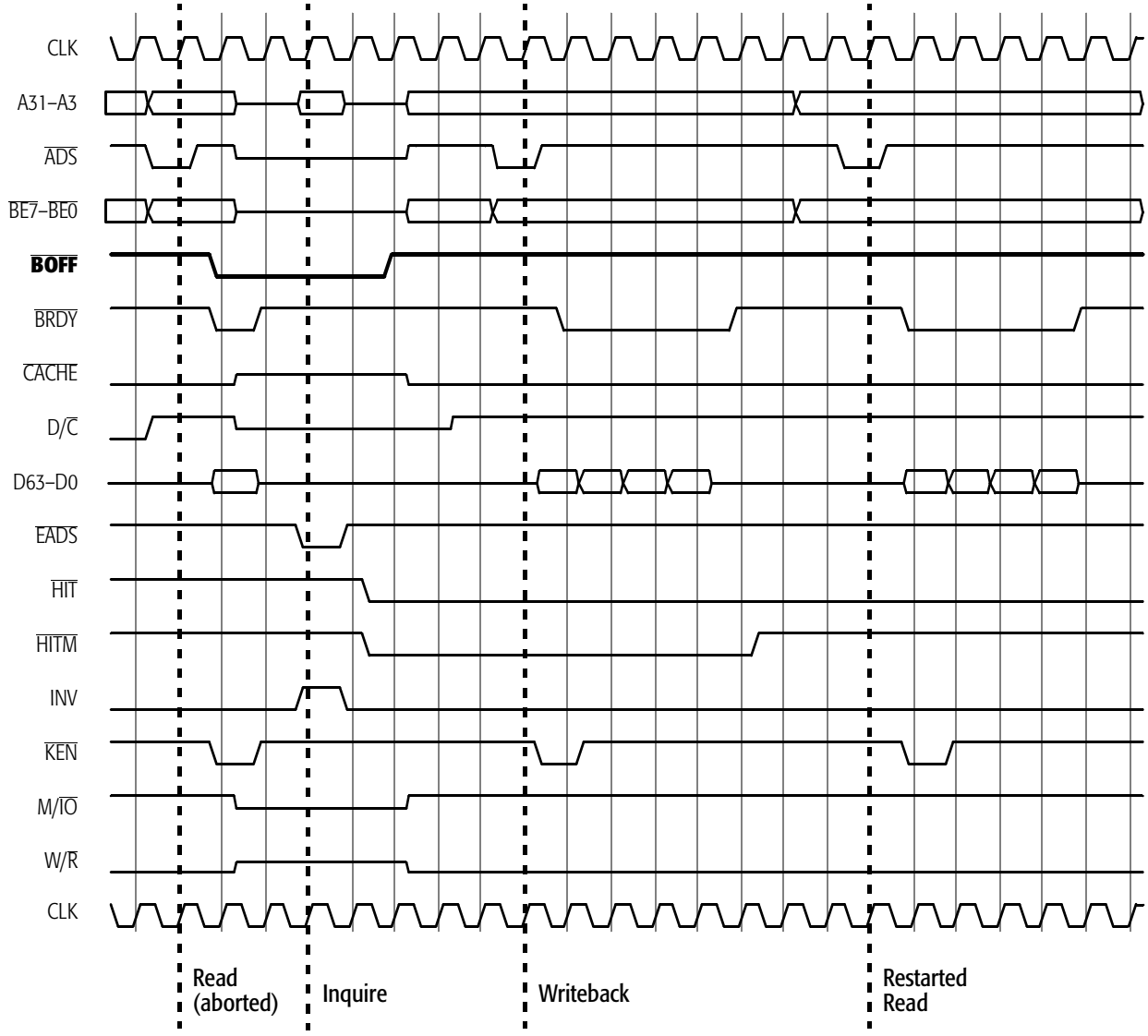


FIGURE 5-13. BOFF-Initiated Inquire Hit to Modified Line

HOLD-Initiated Inquire Hit to Shared or Exclusive Line

Figure 5-14 shows HOLD asserted in the same clock that the processor begins a read cycle. The processor completes the read (which is a burst read) and asserts HLDA two clocks after the last $\overline{\text{BRDY}}$ of the in-progress cycle. It also floats all output and bidirectional signals used for memory or I/O accesses at the same time it asserts HLDA.

In the next clock after sampling HLDA asserted, system logic initiates an inquire cycle by asserting $\overline{\text{EADS}}$ and INV and driving an inquire address on A31–A5. The inquire cycle hits a *shared* or *exclusive* line ($\overline{\text{HIT}}$ asserted and $\overline{\text{HITM}}$ negated two clocks after $\overline{\text{EADS}}$) and the processor invalidates the cache line (not visible on the bus). System logic negates HOLD in the clock after $\overline{\text{EADS}}$, and two clocks later (one clock after $\overline{\text{HIT}}$ and $\overline{\text{HITM}}$ transition) the processor negates HLDA and continues with its other bus cycles.

If $\overline{\text{EADS}}$ is asserted in the same clock that HOLD is negated, the processor recognizes this as a valid inquire cycle and handles it correctly. However, if $\overline{\text{EADS}}$ is asserted in the clock following the negation of HOLD, the processor does not recognize this as a valid inquire cycle.

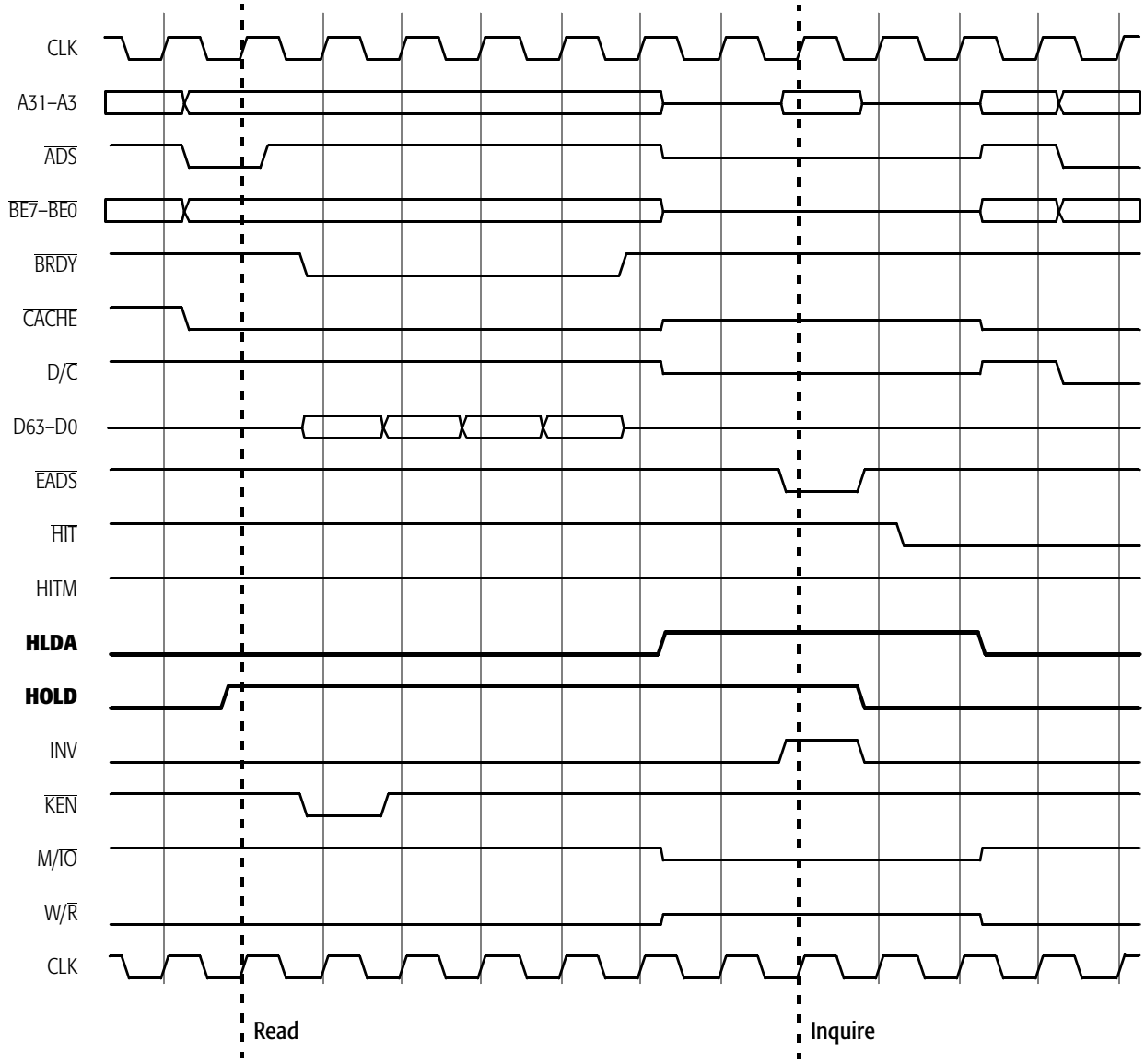


FIGURE 5-14. HOLD-Initiated Inquire Hit to Shared or Exclusive Line

HOLD-Initiated Inquire Hit to Modified Line

Figure 5-15 shows an example similar to the one in Figure 5-14, except that the inquire cycle hits a *modified* line (both $\overline{\text{HIT}}$ and $\overline{\text{HITM}}$ asserted two clocks after $\overline{\text{EADS}}$). System logic negates HOLD in the clock after $\overline{\text{EADS}}$, and two clocks later (one clock after $\overline{\text{HIT}}$ and $\overline{\text{HITM}}$ transition) the processor negates HLDA. As early as one clock after negating HLDA, the processor asserts $\overline{\text{ADS}}$ to drive the writeback, after which the processor invalidates its copy of the line.

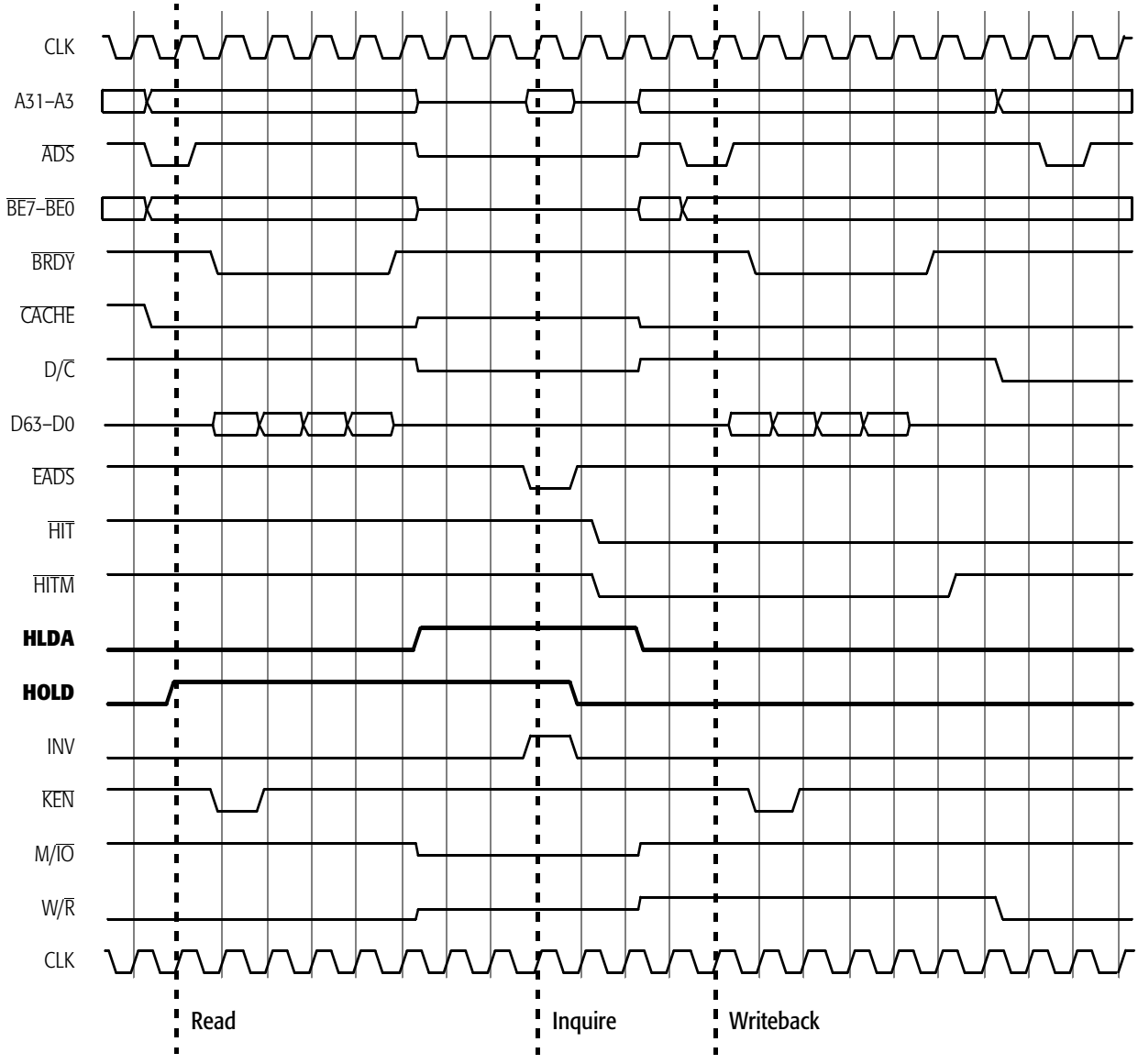


FIGURE 5-15. HOLD-Initiated Inquire Hit to Modified Line

5.4.5 Locked Cycles

The processor asserts $\overline{\text{LOCK}}$ across certain sequences of memory bus cycles that require integrity. These include interrupt acknowledge operations, descriptor-table updates, page-directory and page-table updates, and exchange operations. In addition, the processor asserts $\overline{\text{LOCK}}$ during bus cycles initiated by any instruction that has the LOCK prefix. The processor locks only memory cycles, not I/O cycles.

$\overline{\text{LOCK}}$ is an indication to system logic that it should maintain the integrity of the locked bus cycles, either by never intervening in them or by some other system-level memory protection mechanism that guarantees integrity.

Locked operations generated by the processor typically consist of a read-write pair of bus cycles with an operand modification between the two bus cycles (sometimes called read-modify-write), except that interrupt acknowledge operations, which are also locked, consist of a pair of read cycles with no operand modification between the cycles. Locked operations generated by the LOCK instruction prefix cause $\overline{\text{LOCK}}$ to be asserted only during bus cycles initiated by that single instruction. The processor guarantees at least one idle (or dead) clock between consecutive bus cycles, whether unlocked or locked. This means that consecutive locked operations, which consist of consecutive bus cycles, also have at least one idle clock between them.

Basic Locked Operation

Figure 5-16 shows a pair of read-write bus cycles. The processor asserts $\overline{\text{LOCK}}$ with the $\overline{\text{ADS}}$ of the first bus cycle in the locked operation, and holds it asserted until the last expected $\overline{\text{BRDY}}$ of the last bus cycle in the locked operation. Between the locked operations, the processor negates $\overline{\text{LOCK}}$ for at least one clock.

This example also shows that the value driven on A31–A3 is valid only during the assertion of $\overline{\text{ADS}}$. In the clock immediately preceding the $\overline{\text{ADS}}$ for the write in the first locked operation, the processor changes the address. If system logic reads the address in the clock before $\overline{\text{ADS}}$, an unexpected value may be returned.

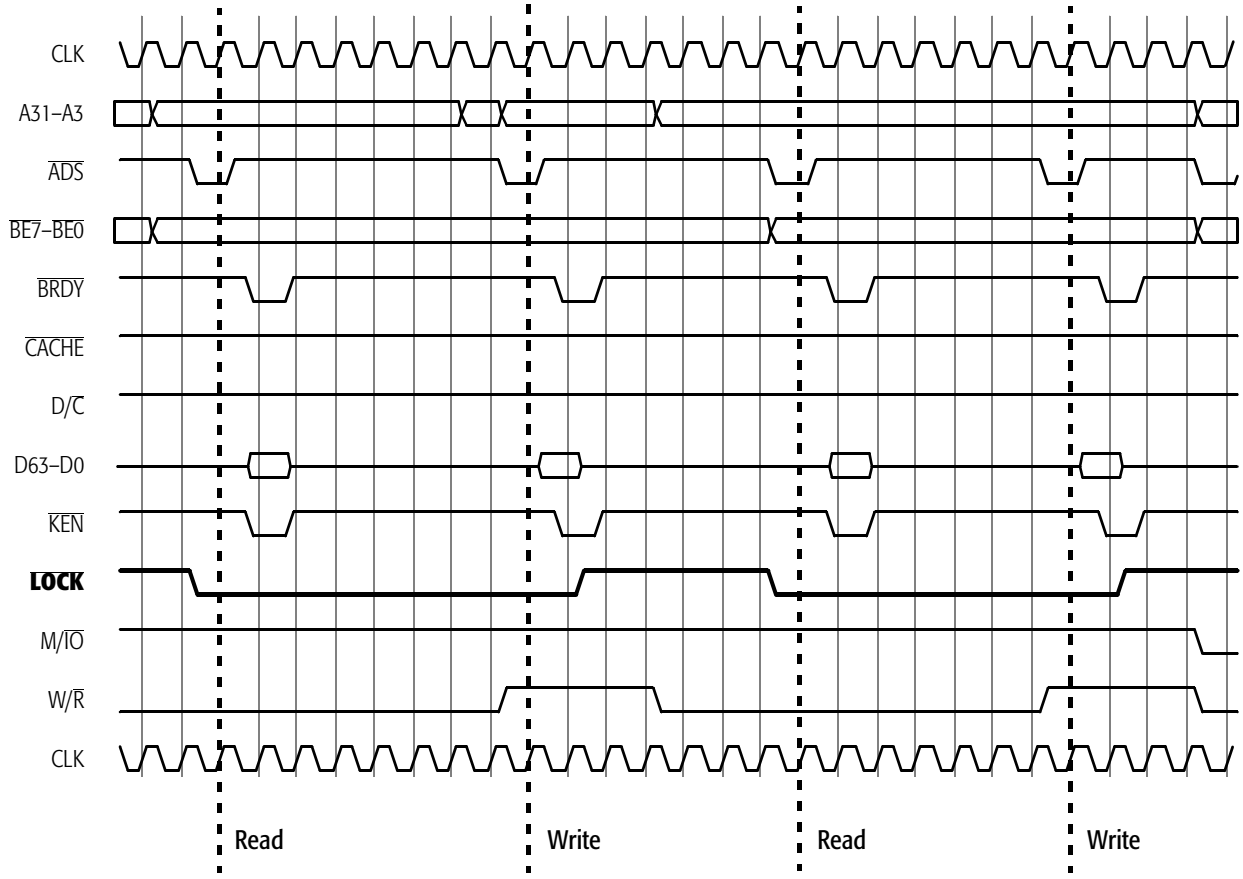


FIGURE 5-16. Basic Locked Operation

TLB Miss (4-Kbyte Page)

Figure 5-17 shows a TLB miss for a 4-Kbyte page. An overview of the 4-Kbyte paging mechanism is illustrated in Figure 3-2 on page 3-5. The paging mechanism for 4-Mbyte pages (Figure 3-3 on page 3-6) is similar but somewhat simpler. The processor has separate TLBs for the two page sizes.

If an address for an access cannot be found in the processor's linearly addressed instruction or data cache, the TLB (which helps translate linear addresses to physical addresses) is searched for the entry associated with the accessed page. A *TLB miss* occurs if the entry cannot be found. For accesses to a 4-Kbyte page that miss the TLB, the processor accesses first the page-directory entry (PDE) in memory and then the page-table entry (PTE) in memory to check, and if necessary set, their Accessed (A) bits. During a write access (not shown in this example), the processor also checks and, if necessary, sets the PTE Dirty (D) bit.

The general sequence, both for PDE and PTE, is as follows for accesses to a 4-Kbyte page:

- The processor drives an unlocked read of the PDE or PTE to see if the relevant bit (A or D) is set.
- If the bit is cleared (0), the processor then drives a locked read-modify-write (four-byte read followed by four-byte write) to set the bit.

The example in Figure 5-17 shows the following specific sequence:

- *Read The PDE*—The A bit in the PDE is set, so nothing further is done with the PDE.
- *Read The PTE*—The A bit in the PTE is cleared, indicating that the page has not been previously accessed since the operating system last cleared the bit
- *Set The Accessed Bit*—The processor performs a locked read-write pair of bus cycles to set the A bit. The diagram shows these cycles as a 4-byte PTE read followed by a 4-byte PTE write. It asserts $\overline{\text{LOCK}}$ with the $\overline{\text{ADS}}$ of the read cycle and holds it asserted until the $\overline{\text{BRDY}}$ of the write cycle.
- *Read The Desired Location (Cache-Line Fill)*—The processor reads the location that caused the TLB miss, filling a cache line as a result of the access.

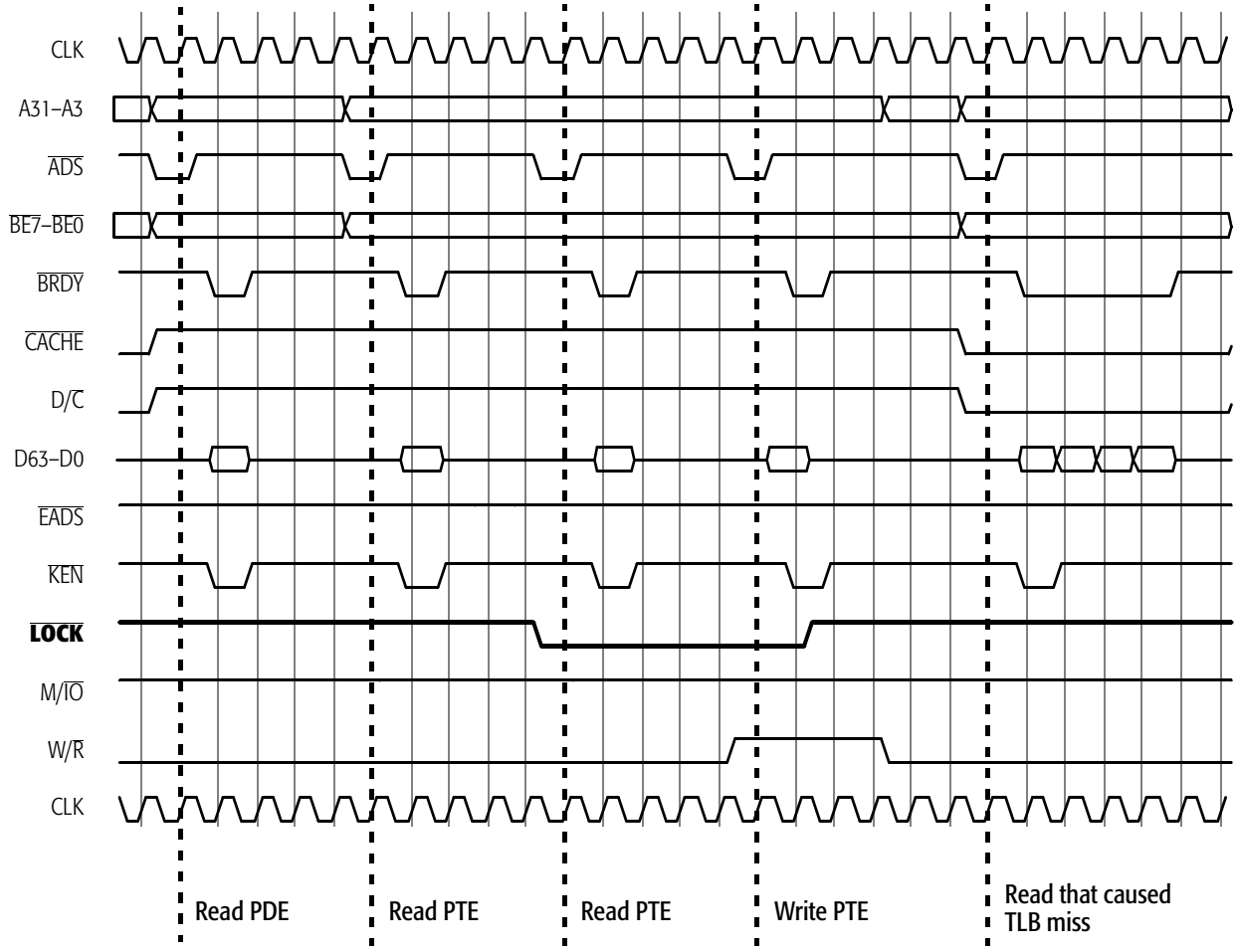


FIGURE 5-17. TLB Miss (4-Kbyte Page)

Locked Operation with $\overline{\text{BOFF}}$ Intervention

Unlike AHOLD and HOLD, $\overline{\text{BOFF}}$ does not permit an in-progress bus cycle to complete. It forces the processor off the bus in the next clock, aborting any in-progress bus cycle that the processor may have begun. If $\overline{\text{BOFF}}$ is asserted during a locked operation, only the cycle(s) aborted before their last $\overline{\text{BRDY}}$ and the cycles not yet run are restarted after $\overline{\text{BOFF}}$ is negated. Thus, system logic must keep track of all cycles in the locked operation that have completed before the assertion of $\overline{\text{BOFF}}$ and must continue the locked operation immediately after $\overline{\text{BOFF}}$ is negated, except that if a writeback is pending when $\overline{\text{BOFF}}$ is negated, the writeback takes precedence over the restarting of the aborted cycles in the locked operation.

Figure 5-18 shows the effect of $\overline{\text{BOFF}}$ intervening in a locked read-write pair of bus cycles. The example begins with the read, while $\overline{\text{LOCK}}$ is asserted. System logic asserts $\overline{\text{BOFF}}$ while the processor is asserting $\overline{\text{ADS}}$ for the write, causing the processor to abort the write and float its bus in the next clock. Another bus master must wait two clocks after the assertion of $\overline{\text{BOFF}}$ before driving its first bus cycle, because the processor does not float its outputs until one clock after the assertion of $\overline{\text{BOFF}}$.

When system logic relinquishes the bus by negating $\overline{\text{BOFF}}$, the processor almost immediately drives the bus again, with $\overline{\text{LOCK}}$ asserted, and restarts the aborted write access by asserting $\overline{\text{ADS}}$ as early as one clock after $\overline{\text{BOFF}}$ is negated (although this example shows two clocks after).

System logic should ensure that the processor results for interrupted and uninterrupted locked cycles are consistent. That is, system logic must guarantee that the memory accessed by the processor is not modified during the time another bus master controls the bus.

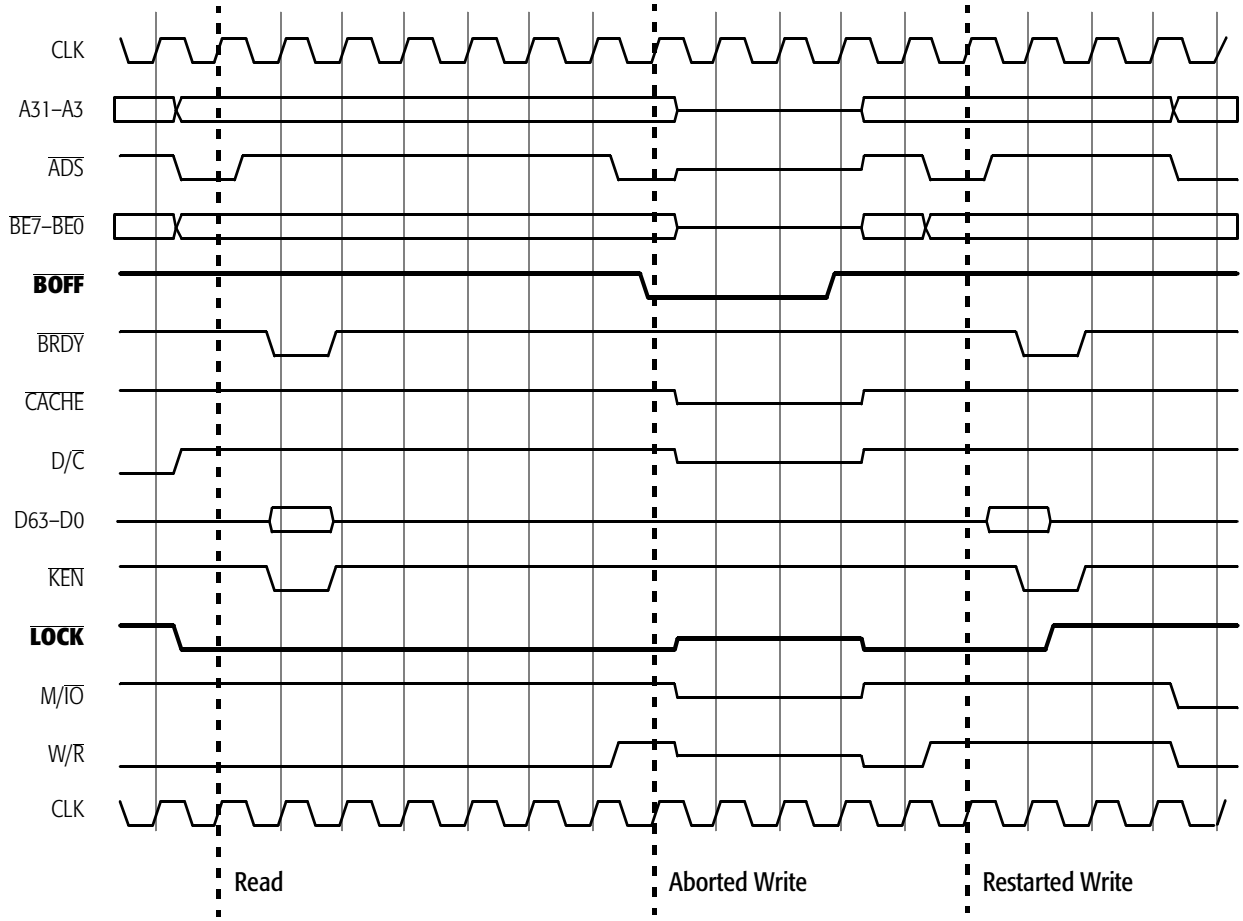


FIGURE 5-18. Locked Operation with BOFF Intervention

Interrupt Acknowledge Operation

Figure 5-19A shows system logic asserting INTR during a burst read. The figure shows the resulting bus behavior, up to the start of the interrupt handler. When the processor recognizes an INTR interrupt at the next instruction-retirement boundary, the processor performs the following actions:

- *Finish In-Progress Bus Cycle*—In Figure 5-19A, a burst read is in progress when system logic asserts INTR. The processor supports only one such in-progress bus cycle.
- *Flush Instruction Pipeline*—This is not visible on the bus.
- *Acknowledge Interrupt*—The interrupt acknowledge operation consists of a locked pair of reads, as shown in Table 5-22. The first read is not functional (a protocol relic). The second read returns the interrupt vector in D7–D0. (The interrupt vector is an offset into an interrupt table.) System logic must return a $\overline{\text{BRDY}}$ in response to both cycles. The processor inserts at least one idle clock between the locked reads.
- System logic will typically not be able to determine the instruction boundary on which the processor recognizes INTR. Thus, as a practical matter, system logic should hold INTR asserted until the beginning of the interrupt acknowledge operation, or until there is some other evidence that the interrupt service routine has been entered (for example, the access to the interrupt-table address).

TABLE 5-22. Interrupt Acknowledge Operation Definition

Processor Outputs	First Bus Cycle	Second Bus Cycle
D/$\overline{\text{C}}$	0	0
M/$\overline{\text{IO}}$	0	0
W/$\overline{\text{R}}$	0	0
$\overline{\text{BE7}}$–$\overline{\text{BE0}}$	EFh	FEh (low byte enabled)
A31–A3	0	0
D63–D0	(ignored)	Interrupt vector expected from interrupt controller on D7–D0

- *Disable Maskable Interrupts*—The processor does this under certain conditions (see Section 5.2.32 on page 5-85 for details), and it is not visible on the bus.

As shown in Figure 5-19B and Figure 5-19C, following the interrupt acknowledge operation and a quiet period during which the processor executes housekeeping microcode, the processor prepares to service the interrupt by performing the following accesses on the bus:

- *IDT Lookup*—Using the interrupt vector and, in Protected mode, the base address of the interrupt descriptor table (IDT), from the interrupt descriptor table register (IDTR), the processor performs a read on the bus to look up the 8-byte IDT entry. In Figure 5-19B, this appears as a burst read, which is cached.
- *GDT Lookup*—Using the segment descriptor from the IDT, the processor performs another read of the global descriptor table (GDT) to look up the 8-byte code segment descriptor. This also appears as a burst read, which is cached. Alternatively, this read can access the local descriptor table rather than the global descriptor table.
- *Write to Stack*—As shown in Figure 5-19C the processor saves the EFLAGS, CS, and EIP registers on the stack. These saves appear as three single writes.
- *Code Fetch for Interrupt Handler*—Finally, using the base address from the GDT descriptor and the offset from the IDT descriptor, the processor locates the interrupt handler in the code segment (CS) and begins fetching the code in cacheable burst reads.

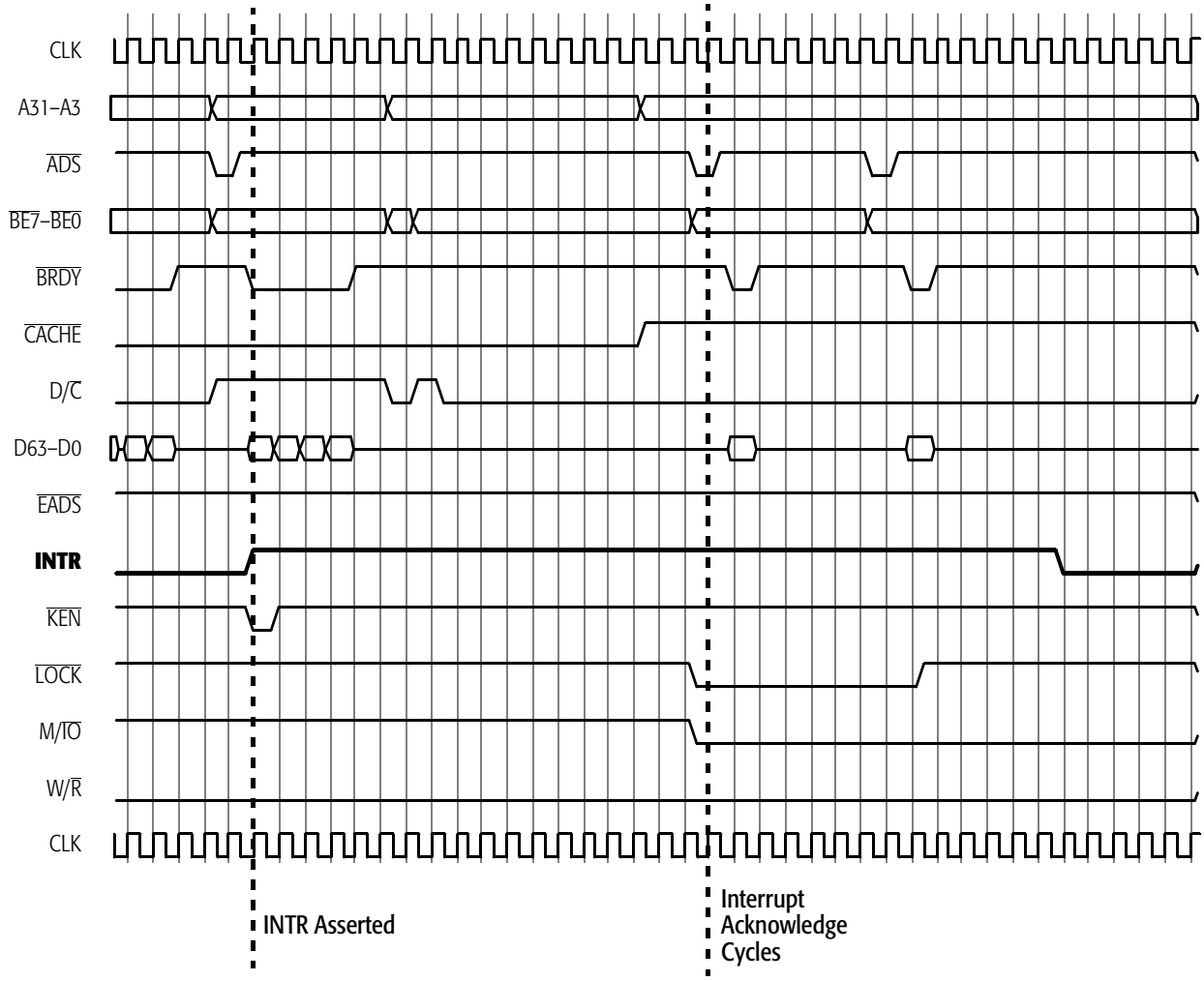


FIGURE 5-19A. Interrupt Acknowledge Operation Part 1

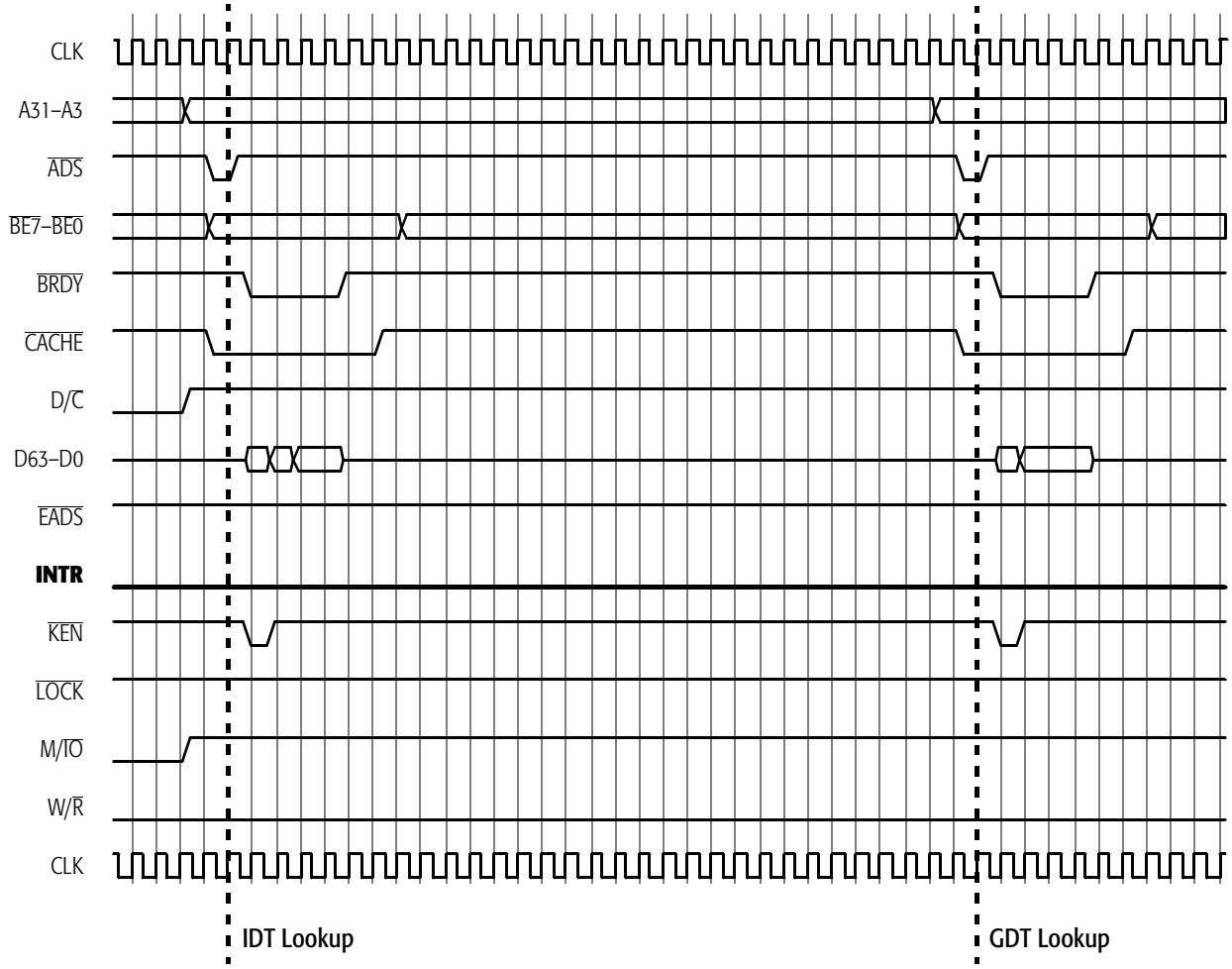


FIGURE 5-19B. Interrupt Acknowledge Operation Part 2

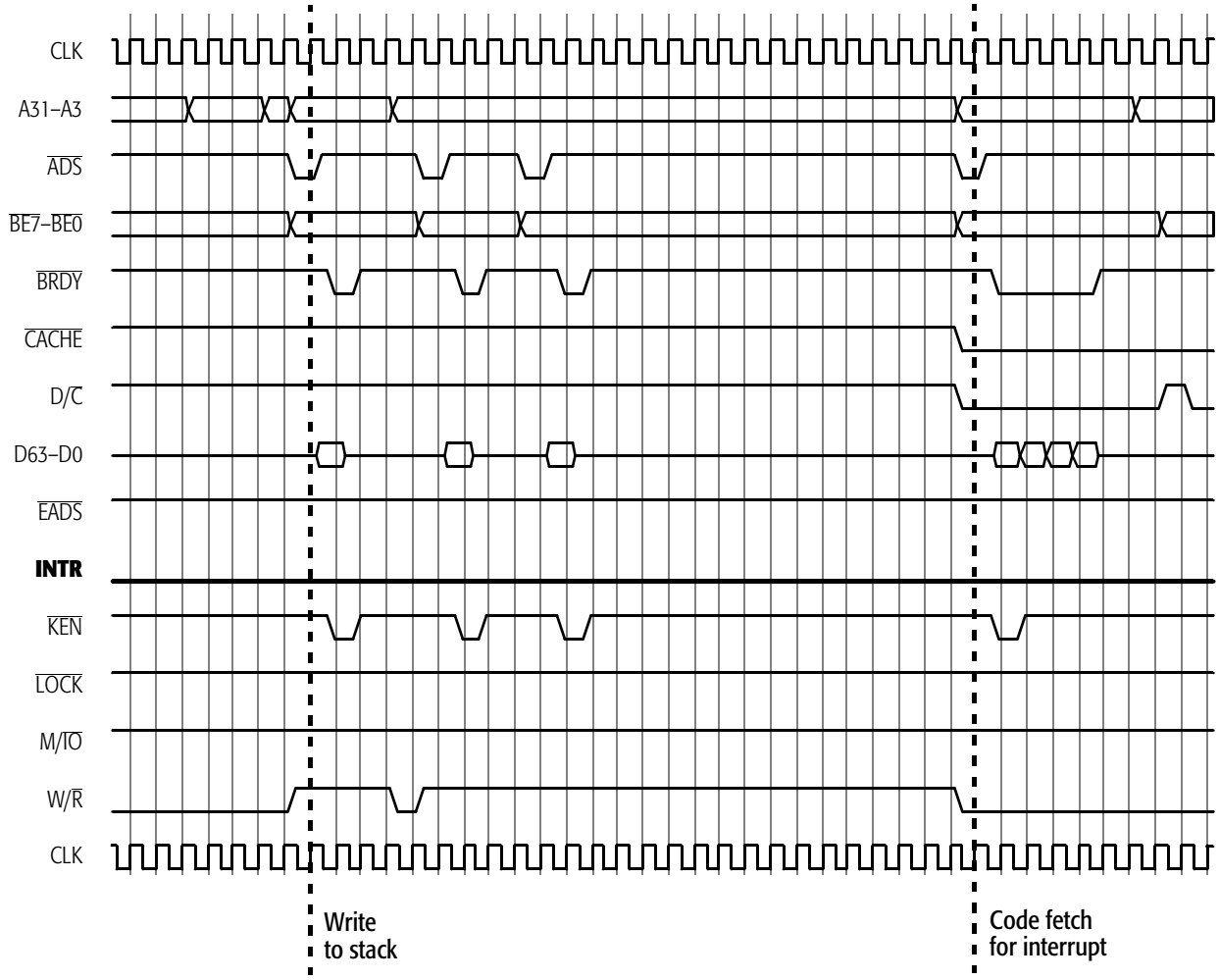


FIGURE 5-19C. Interrupt Acknowledge Operation Part 3

5.4.6 Special Bus Cycles

The processor drives D/$\overline{C}$ = 0, M/$\overline{IO}$ = 0, and W/$\overline{R}$ = 1 to define a special bus cycle. The values of these cycle-definition signals are the same for all special cycles. Only $\overline{BE7}$–$\overline{BE0}$ and A31–A3 differentiate among the special cycles, as shown in Table 5-23.

This function of $\overline{BE7}$–$\overline{BE0}$ bears no relationship to the D63–D0 data bus. It is particularly apparent in the case of the branch-trace message special bus cycle, during which the value of $\overline{BE7}$–$\overline{BE0}$ is DFh (1101_1111b) but, in contradiction to the byte-enable bits, the four bytes on D31–D0 carry valid data during both cycles of the operation. During the first cycle, D31–D0 carries the EIP value of the source (branch) instruction. During the second cycle, D31–D0 carries the EIP value of the branch-target instruction.

TABLE 5-23. Encodings For Special Bus Cycles

$\overline{BE7}$–$\overline{B0}$	A31–A3	Special Bus Cycle[1]	Cause
FEh	...00h	Shutdown	Triple fault
FDh	...00h	Cache Invalidation	INVD instruction
FBh	...10h	Stop Grant	$\overline{STPCLK}$
FBh	...00h	Halt	HLT instruction
F7h	...00h	Cache Writeback and Invalidation	WBINVD instruction
EFh	...00h	$\overline{FLUSH}$ Acknowledge	$\overline{FLUSH}$
DFh	...00h	Branch-Trace Message[2]	Bit 5 = 1 and bits 3–1 = 001 in the hardware configuration register (HWCR). See Section 7.1 on page 7-3 for details.

Notes:

1. *For all special bus cycles, D/$\overline{C}$ = 0, M/$\overline{IO}$ = 0 and W/$\overline{R}$ = 1. System logic must return $\overline{BRDY}$ in response to this cycle.*
2. *The message in a branch-trace message special bus cycle is different in the AMD5$_K$86 and Pentium processors.*

Basic Special Bus Cycle

Figure 5-20 shows a basic special bus cycle, which is defined during $\overline{ADS}$ by D/$\overline{C}$ = 0, M/$\overline{IO}$ = 0, and W/$\overline{R}$ = 1 and differentiated by $\overline{BE7}$–$\overline{BE0}$ and A31–A3. In this example, $\overline{BE7}$–$\overline{BE0}$ = FBh and A31–A3 = 0, so it is the special cycle the processor generates after executing a HLT instruction. System logic must respond with $\overline{BRDY}$.

All special bus cycles serialize the pipeline. $\overline{EWBE}$ is not checked prior to running special bus cycles (all of which have W/$\overline{R}$ = 1), so $\overline{EWBE}$ has no effect on any special bus cycles.

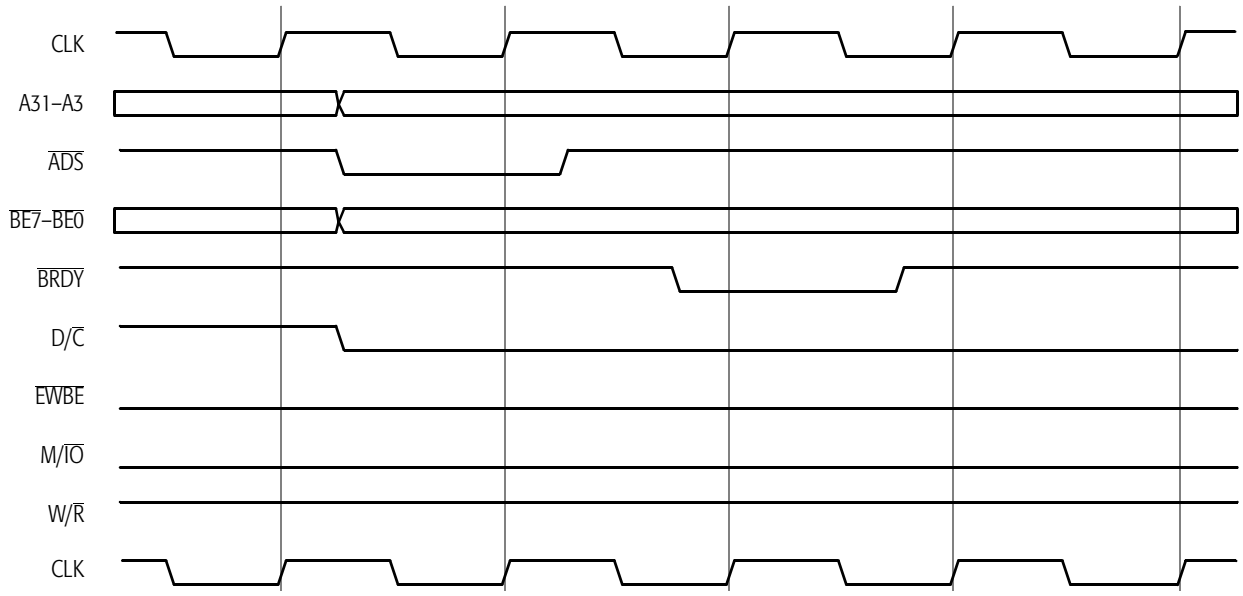


FIGURE 5-20. Basic Special Bus Cycle (Halt Cycle)

Shutdown Cycle

Figure 5-21 shows a shutdown and the special cycle that follows. The processor enters shutdown when an interrupt or exception occurs during the handling of a double fault (vector 8), which amounts to a triple fault. When the processor encounters such a triple fault, it stops its activity on the bus and generates the special bus cycle for shutdown ($\overline{BE7}$–$\overline{BE0}$ = FEh). System logic must respond with $\overline{BRDY}$.

System logic must assert NMI, INIT, RESET, or $\overline{SMI}$ to get the processor out of the Shutdown state.

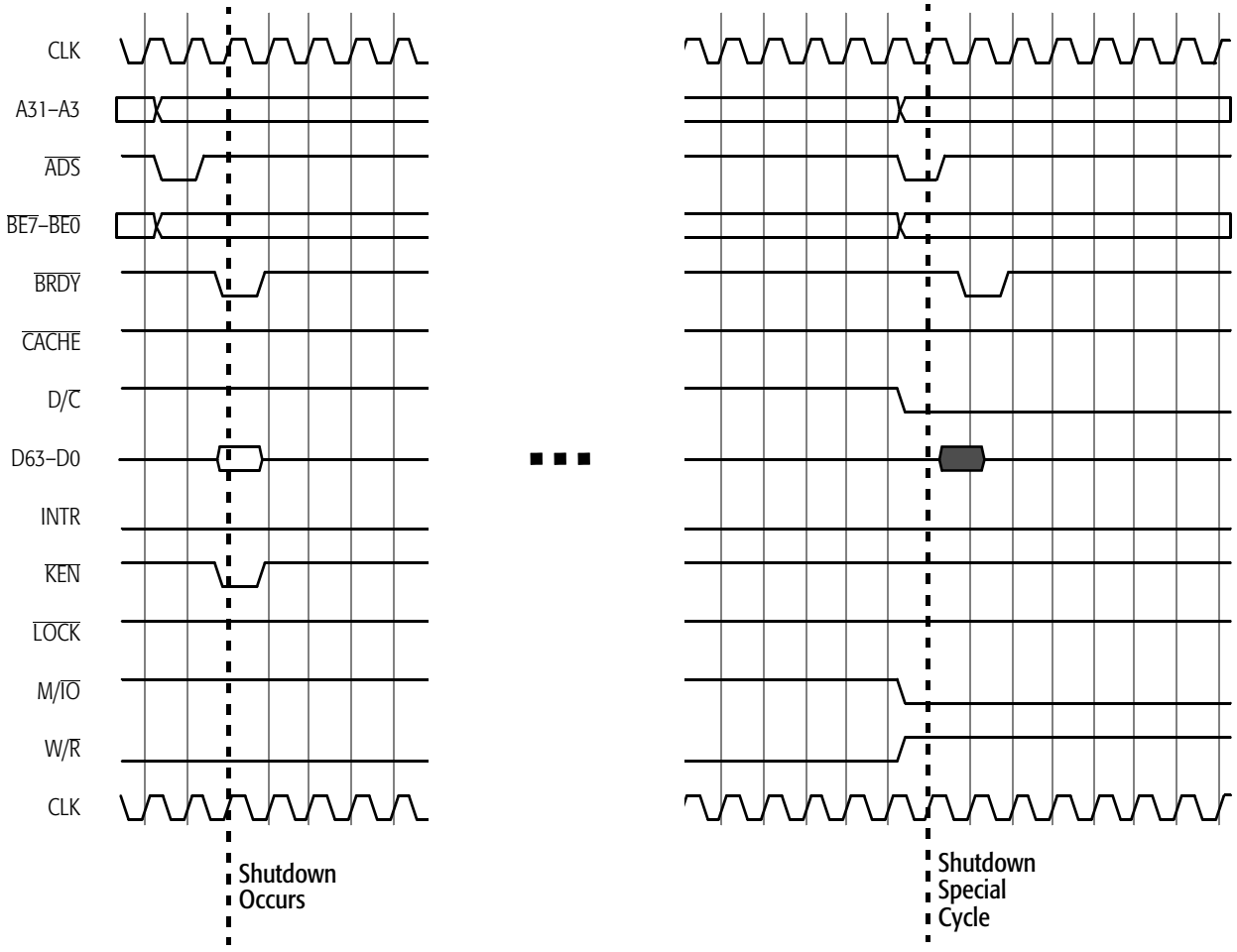


FIGURE 5-21. Shutdown Cycle

FLUSH-Acknowledge Cycle

Figure 5-22 shows the FLUSH-acknowledge special bus cycle, which the processor drives in response to system logic's assertion of FLUSH. This example shows the processor completing other unrelated bus cycles following the assertion of FLUSH. These bus cycles are caused by the execution of instructions earlier in the pipeline, which are completing execution before the processor recognizes FLUSH on the next instruction-retirement boundary.

FLUSH causes the processor to write back all modified lines in its data cache. Only one such writeback is shown in this example. After all writebacks complete, the processor invalidates all lines in both of its caches. Then, the processor generates the FLUSH-acknowledge special bus cycle (BE7–BE0 = EFh) to indicate that the writebacks and invalidation have completed. System logic must respond by asserting BRDY.

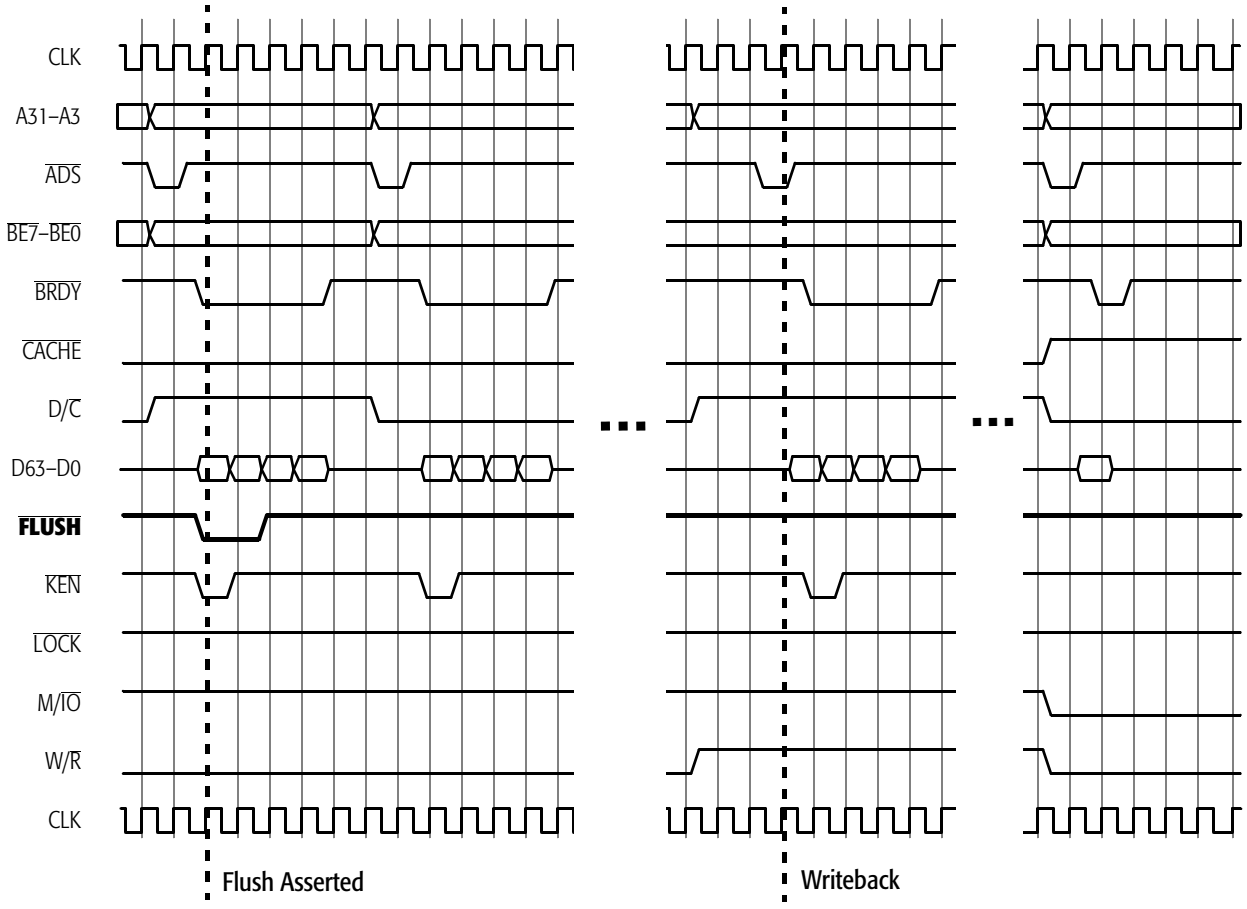


FIGURE 5-22. FLUSH-Acknowledge Cycle

Cache-Invalidation Cycle (INVD Instruction)

Figure 5-23 shows the cache-invalidation special bus cycle, which the processor drives in response to the execution of the INVD instruction. The INVD instruction causes the processor to invalidate each line in its instruction and data caches. Modified lines in the data cache are not written back.

Although the execution of INVD is not visible on the bus, the lack of activity on the bus as the microcode invalidates the lines in the internal cache can be seen. When all lines in both caches are invalidated, the processor drives the cache-invalidation special bus cycle ($\overline{BE7}$–$\overline{BE0}$ = FDh). System logic must respond by asserting $\overline{BRDY}$. When it does, the processor typically begins driving one or more burst reads on the bus to refill its caches.

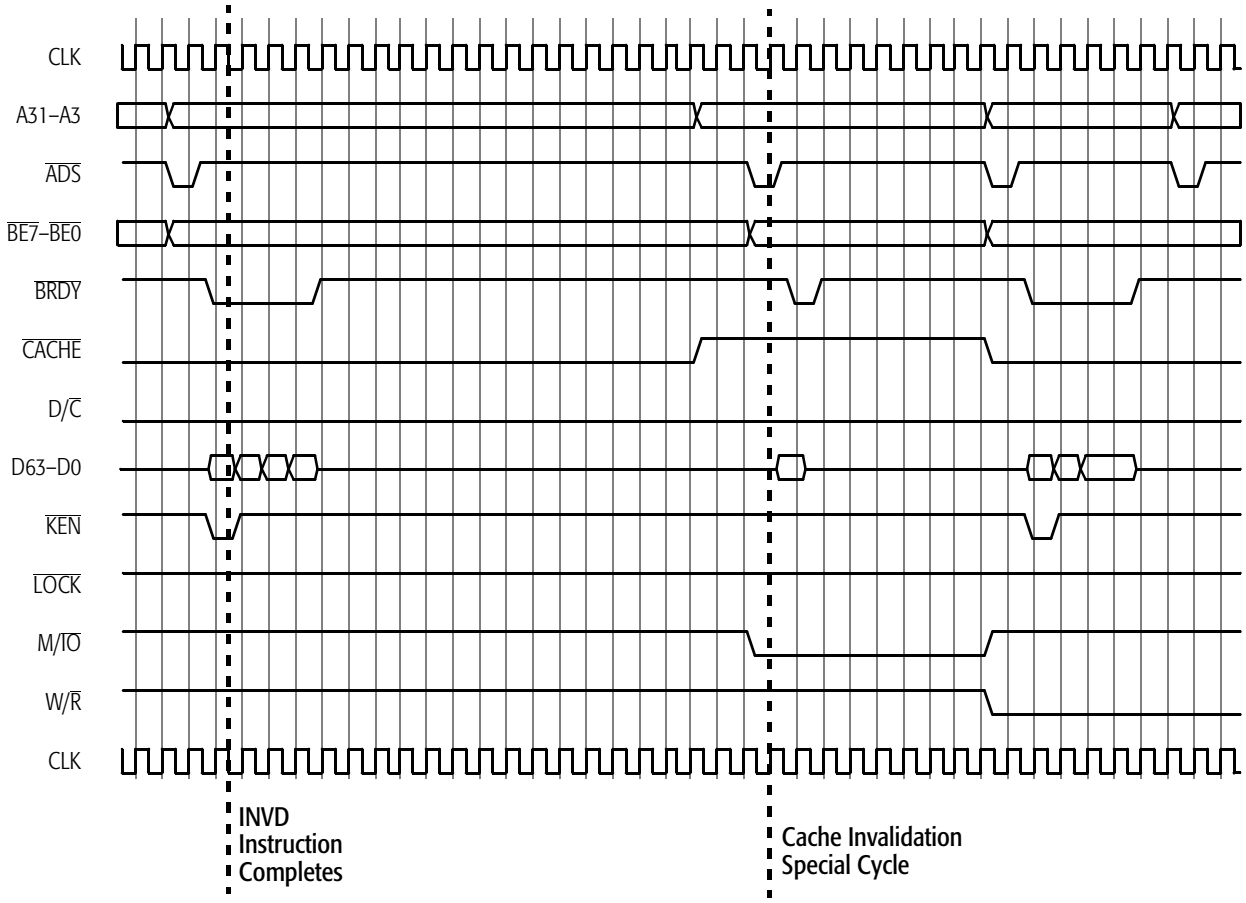


FIGURE 5-23. Cache-Invalidation Cycle (INVD Instruction)

Cache-Writeback and Invalidation Cycle (WBINVD Instruction)

Figure 5-24A and Figure 5-24B show the cache-writeback and invalidation special bus cycle, followed by the cache-invalidation special bus cycle. The processor drives these two special cycles after executing the WBINVD instruction.

The execution of WBINVD causes the processor to invalidate each line in its instruction and data caches. If a data cache line is in the *modified* state, the line is written back immediately before being invalidated. During such writebacks, A31–A5 defines the address of a 32-byte location in memory to which the *modified* cache line will be written back. After all *modified* lines are written back and all lines in both caches are invalidated, the processor first drives the cache-writeback and invalidation special bus cycle ($\overline{BE7}$–$\overline{BE0}$ = F7h) and then the cache-invalidation special bus cycle ($\overline{BE7}$–$\overline{BE0}$ = FDh). System logic must respond by asserting $\overline{BRDY}$ to each of the two special cycles as shown in Figure 5-24B.

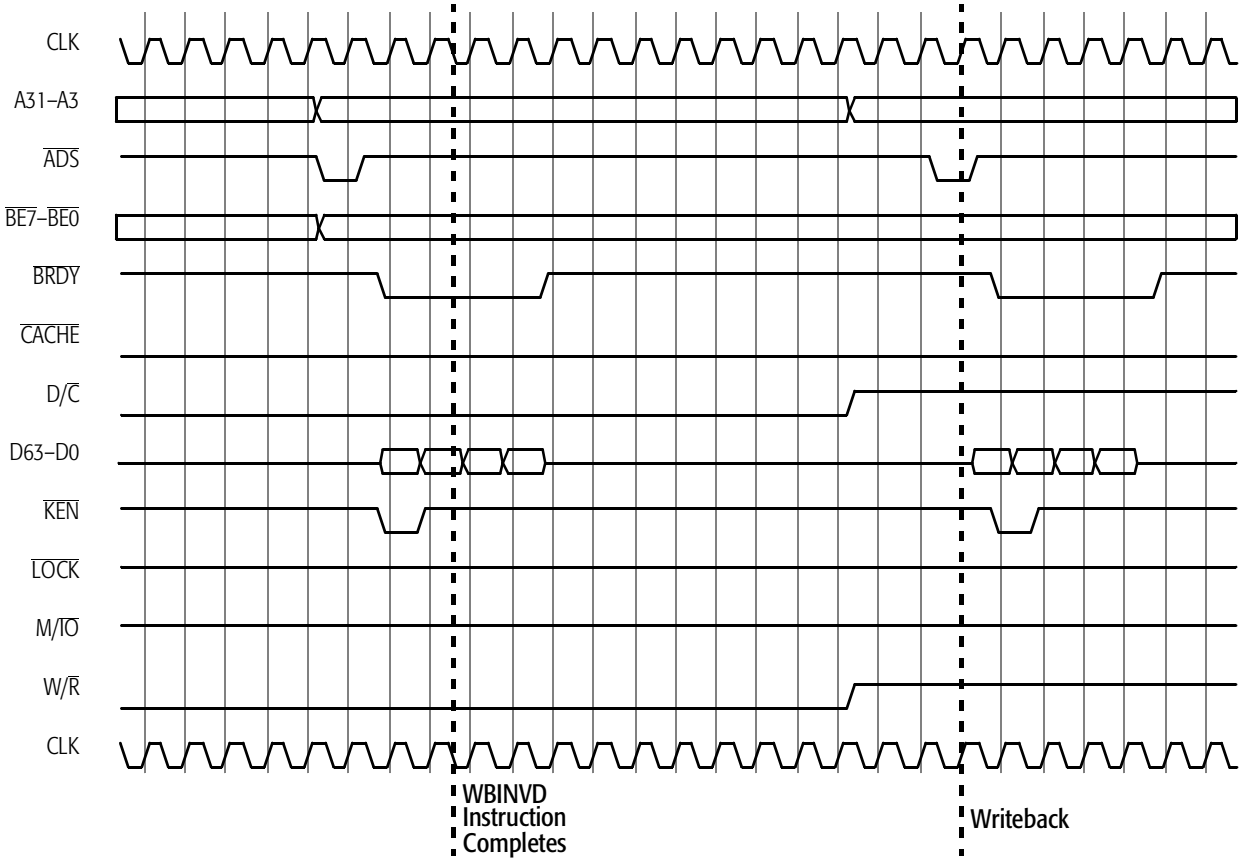


FIGURE 5-24A. Cache-Writeback and Invalidation Cycle (WBINVD Instruction) Part 1

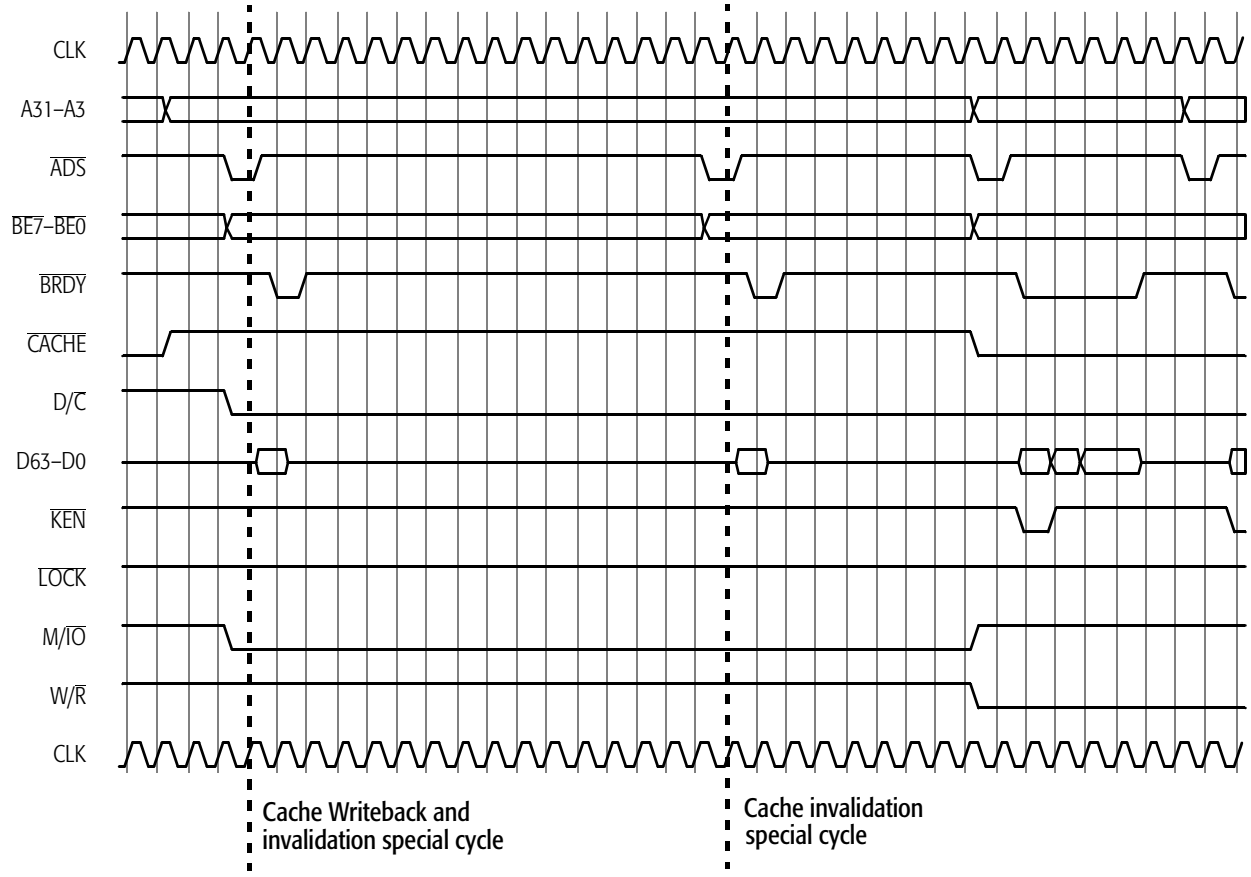


FIGURE 5-24B. Cache-Writeback and Invalidation Cycle (WBINVD Instruction) Part 2

Branch-Trace Message Cycles

Figure 5-25 shows the two branch-trace message special bus cycles that the processor generates for each taken branch when branch tracing is enabled as described in Section 7.6 on page 7-17. System logic can accumulate the address and data bus values for debugging or profiling.

The processor drives these special bus cycles immediately after each taken-branch instruction is executed. Both special bus cycles have a $\overline{BE7}$–$\overline{BE0}$ = DFh, and system logic must respond by asserting $\overline{BRDY}$ to each of the cycles. The first cycle identifies the branch source, and the second identifies the branch target, as shown in Table 5-24.

TABLE 5-24. 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–A0	EIP of branch source.	EIP of branch target.

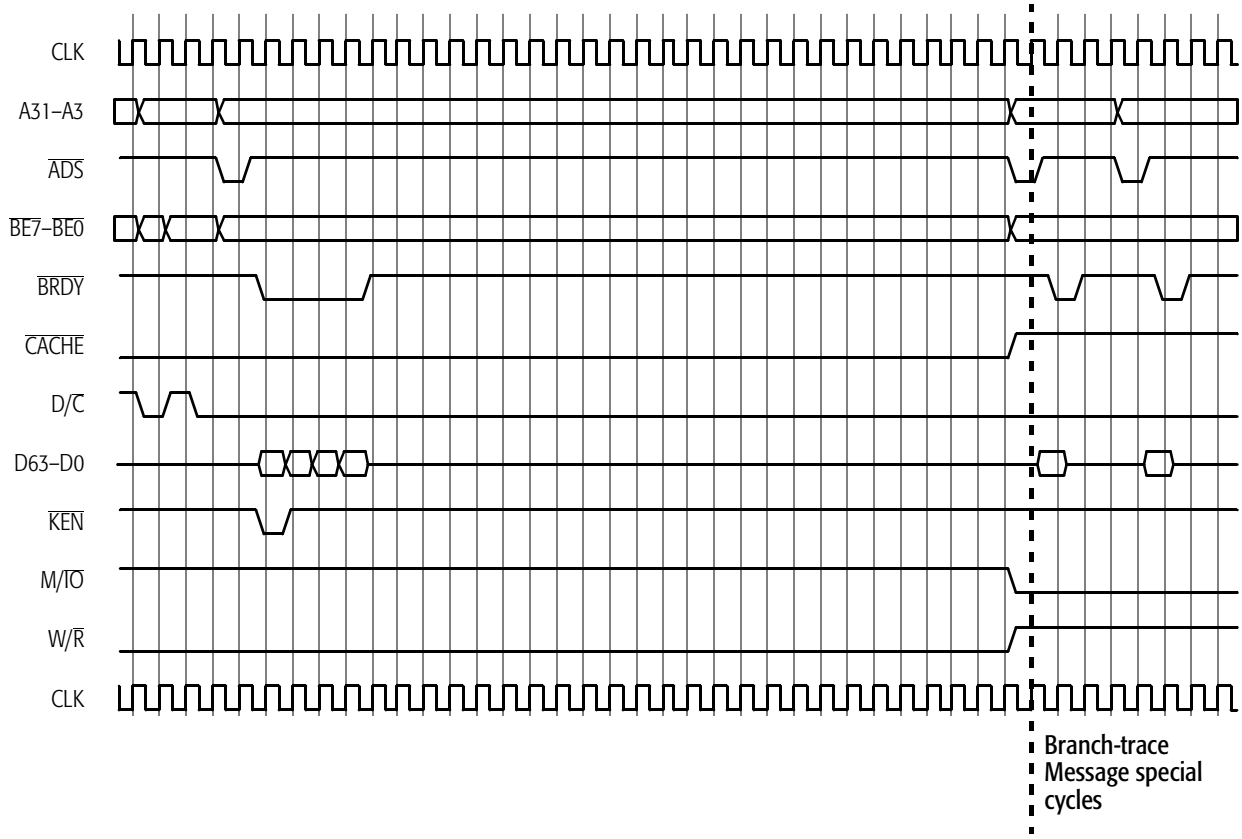


FIGURE 5-25. Branch-Trace Message Cycle

5.4.7 Mode Transitions, Reset, and Testing

System logic can control the system-management, clocking, and initialization states of the processor with SMI, STPCLK, INIT, and RESET. The following examples shows the processor's response to some of the signals.

Transition from Normal Execution to SMM

Figure 5-26A and Figure 5-26B shows the transition from one of the processor's normal operating modes (Real, Protected, or Virtual-8086 mode) to System Management Mode (SMM). System logic causes this transition by asserting SMI.

Upon recognizing an SMI interrupt at the next instruction-retirement boundary, the processor performs the following actions:

1. *Flush Pipeline*—The processor invalidates all instructions remaining in the pipeline. This is not visible on the bus.
2. *Complete In-Progress Cycle*—If the processor had begun a bus cycle when SMI was asserted, the processor completes the bus cycle and waits until the system asserts the last expected BRDY and also asserts EWBE. In Figure 5-26A, a burst read is shown completing after SMI is asserted.
3. *Acknowledge*—After sampling EWBE asserted, the processor asserts SMIACT to acknowledge the interrupt. This is visible on the bus after SMI is recognized. At that point, system logic must ensure that all memory accesses during SMM are to the SMM memory space.
4. *Save Processor State*—The processor saves its state in the SMM state-save area. These saves appear at the far right of the example in Figure 5-26B.
5. *Disable Interrupts and Debug Traps*—The processor disables maskable interrupts by clearing the interrupt flag (IF) in EFLAGS, disables NMI interrupts, clears the trap flag (TF) in EFLAGS, and clears the DR7–DR6 debug control and status registers. This is not visible on the bus.
6. *Service Interrupt*—The processor jumps to the entry point of the SMM service routine at the SMM base physical address, whose default is 0003_8000h in SMM memory.

For details on SMM, see Section 6.3 on page 6-23.

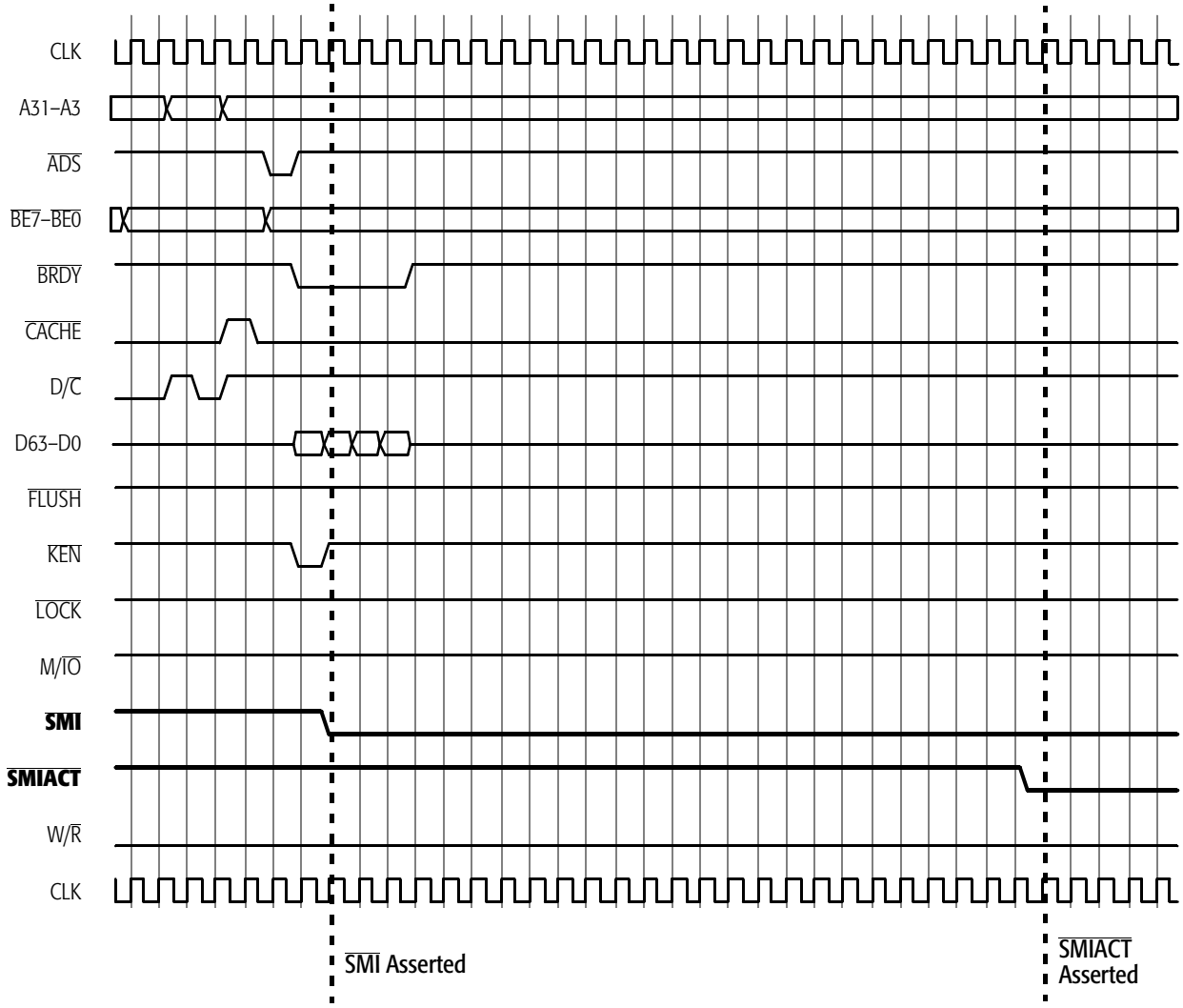


FIGURE 5-26A. Transition from Normal Execution to SMM Part 1

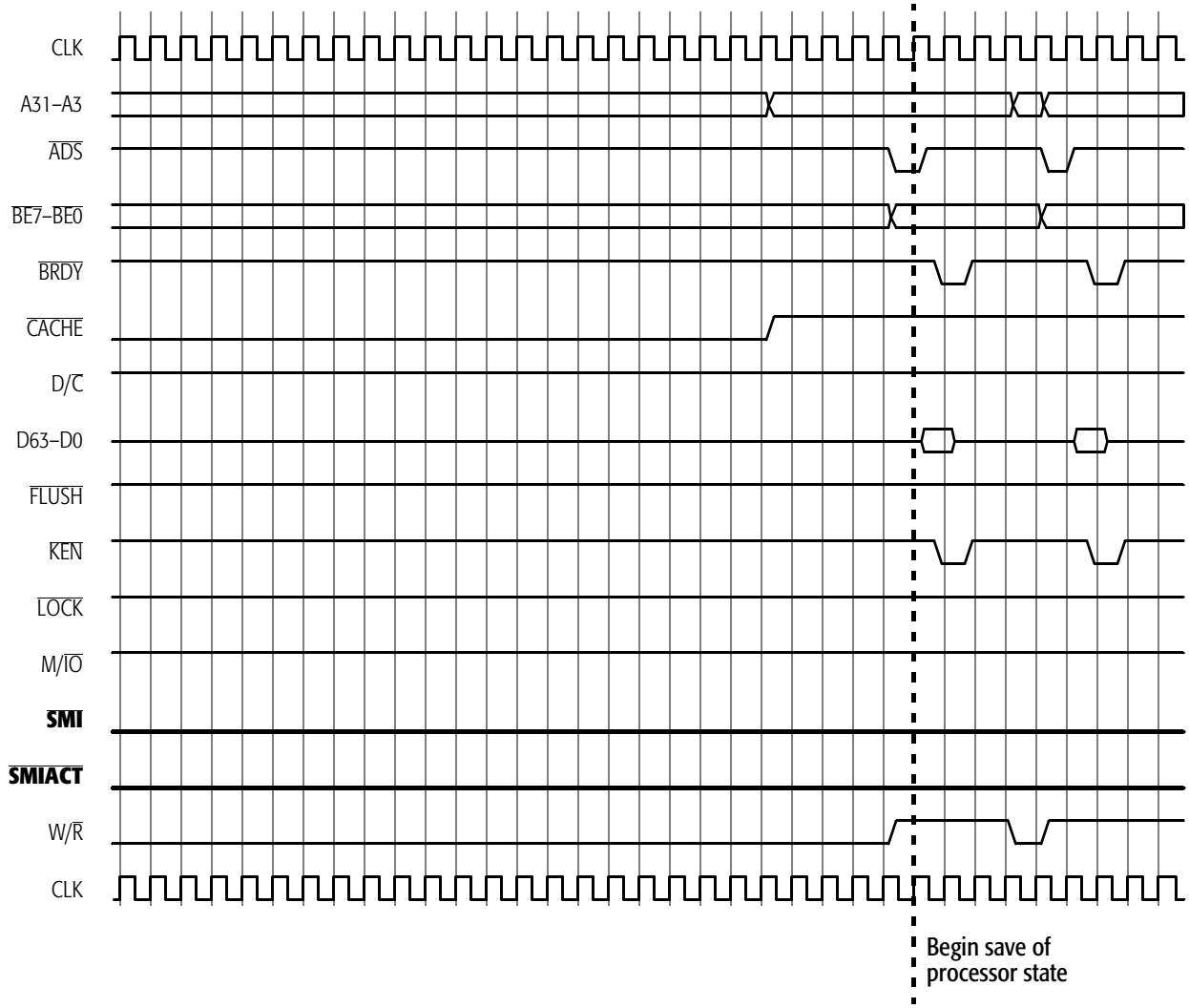


FIGURE 5-26B. Transition from Normal Execution to SMM Part 2

Stop-Grant and Stop-Clock States

Figure 5-27A and Figure 5-27B show the processor's transition from normal execution to the Stop-Grant state, then to the Stop-Clock state, and finally back to normal execution. The series of transitions begins when system logic asserts $\overline{\text{STPCLK}}$. Upon recognizing a $\overline{\text{STPCLK}}$ interrupt at the next instruction-retirement boundary, the processor performs the following actions, in the order shown:

1. *Flush Pipeline*—The processor invalidates all instructions remaining in the pipeline. This is not visible on the bus.
2. *Complete In-Progress Cycle*—If the processor had begun a bus cycle or locked operation when $\overline{\text{STPCLK}}$ was asserted, the processor completes the bus cycle and waits until the system asserts the last expected $\overline{\text{BRDY}}$ and also asserts $\overline{\text{EWBE}}$. If no bus cycle is in progress, system logic must assert $\overline{\text{EWBE}}$ at the same time as, or at sometime after, it asserts $\overline{\text{STPCLK}}$. In Figure 5-27A, a burst read is shown completing after $\overline{\text{STPCLK}}$ is asserted.
3. *Stop-Grant Cycle*—After sampling both $\overline{\text{EWBE}}$ asserted, the processor drives a Stop-Grant special bus cycle. This cycle is identified by D/$\overline{\text{C}}$ = 0, M/$\overline{\text{IO}}$ = 0, W/$\overline{\text{R}}$ = 1, $\overline{\text{BE7}}$–$\overline{\text{BE0}}$ = FBh and A31–A3 = 10h. System logic must respond by asserting $\overline{\text{BRDY}}$. This is visible on the bus, near the middle of Figure 5-27A.
4. *Stop Internal Clock*—When system logic returns $\overline{\text{BRDY}}$ for the Stop-Grant special bus cycle, the processor stops its internal clock and floats D63–D0 and DP7–DP0. This is on the bus between Figure 5-27A and Figure 5-27B immediately after the $\overline{\text{BRDY}}$ of the Stop-Grant special bus cycle.
5. *(Optional) Stop Bus Clock*—After returning $\overline{\text{BRDY}}$ in response to the Stop-Grant special bus cycle, power-management logic can transition to the Stop-Clock state by stopping CLK while $\overline{\text{STPCLK}}$ is held asserted.

$\overline{\text{STPCLK}}$ must be held asserted throughout the Stop-Grant and (if entered) Stop-Clock states.Figure 5-27B shows the processor resuming normal execution after system logic negates $\overline{\text{STPCLK}}$.

For details on clock control, see Section 6.4 on page 6-33.

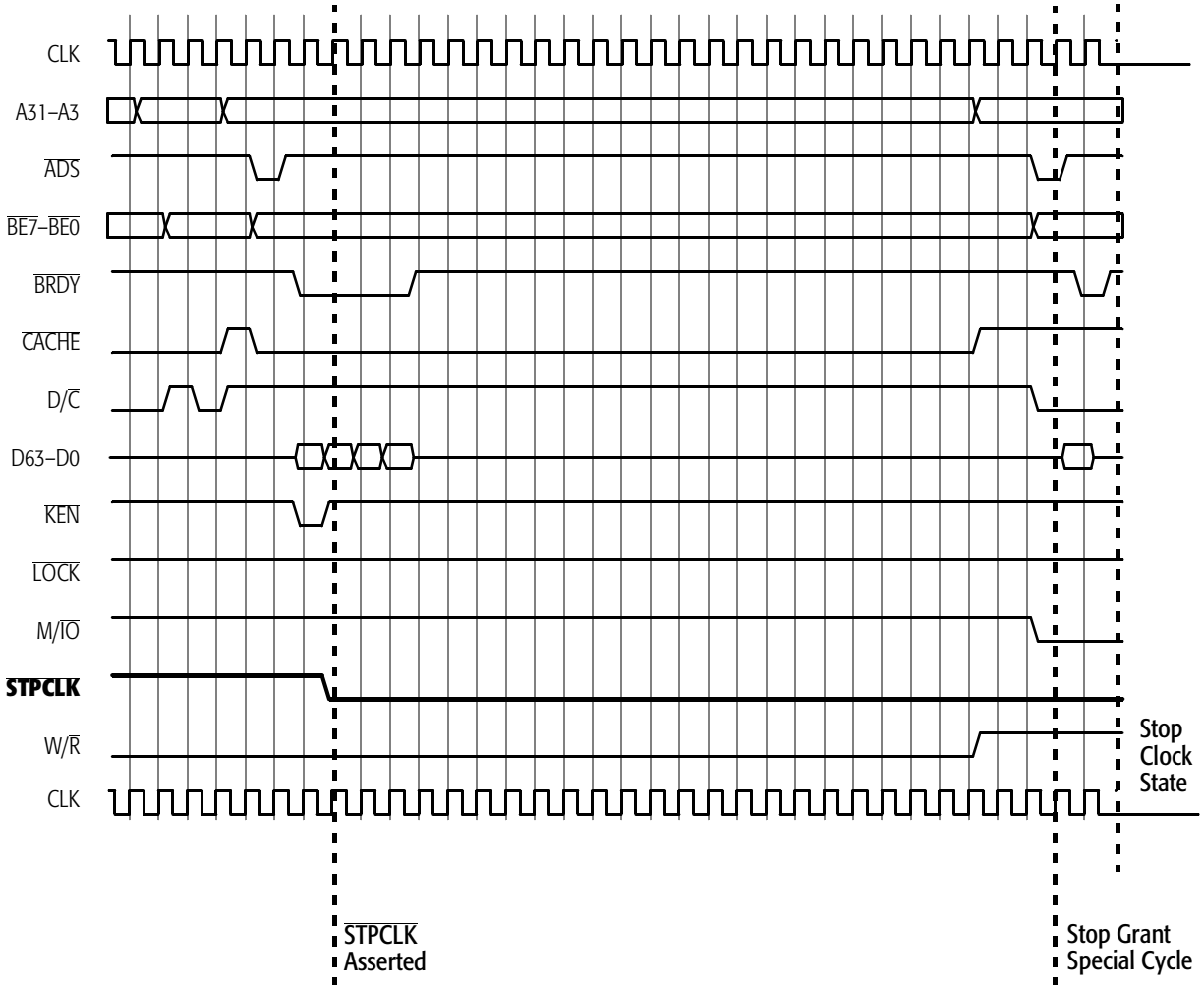


FIGURE 5-27A. Stop-Grant and Stop-Clock Modes Part 1

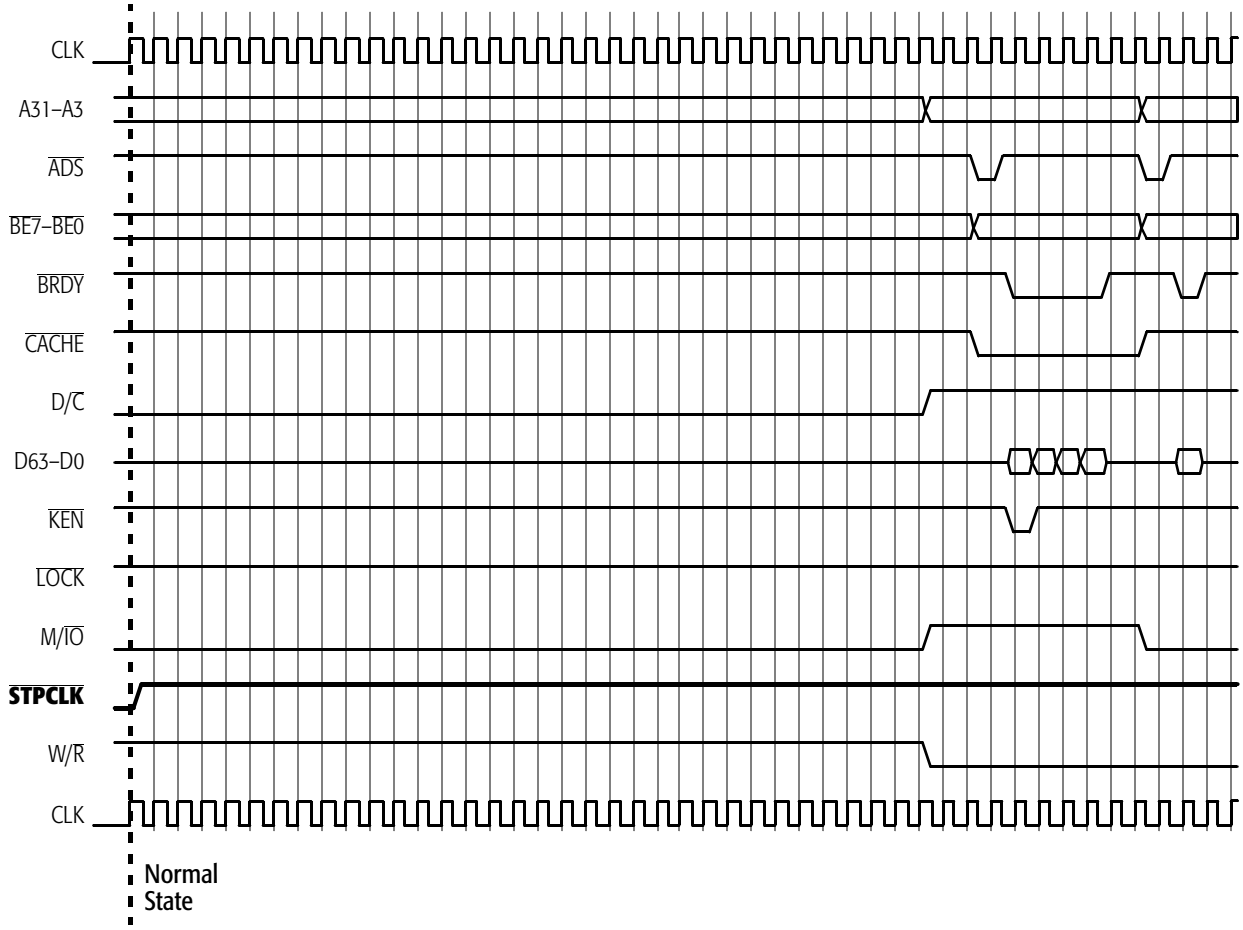


FIGURE 5-27B. Stop-Grant and Stop-Clock Modes Part 2

INIT-Initiated Transition from Protected Mode to Real Mode

Figure 5-28 shows an example in which the operating system writes to an I/O port, causing system logic to assert INIT. The assertion of INIT starts an extended microcode sequence that terminates with a code fetch from the Reset location.

INIT is typically asserted in response to a BIOS interrupt that writes to an I/O port. This is often, for example, in response to the operator's pressing Control-Alt-Del. The BIOS writes to a port (such as port 64h in the keyboard controller) that asserts INIT. INIT is also used to support 286 software that must return to Real mode after accessing extended memory in Protected mode. The 286 processor does not have an INIT input—a transition from Protected mode to Real mode can only be made on the 286 processor by asserting RESET. With the INIT signal, however, the operating system can cause the transition through a BIOS interrupt without loss of cache contents or floating-point state.

Upon recognizing an INIT interrupt at the next instruction-retirement boundary, the processor performs the following actions, in the order shown:

1. *Flush Pipeline*—The processor invalidates the instruction pipeline and TLB. This is not visible on the bus.
2. *Reinitialize*—The processor reinitializes the general-purpose and system registers to their reset values. This is also not visible on the bus, except as an extended period of inactivity.
3. *Jump To BIOS*—The processor jumps to the BIOS at address FFFF_FFF0h, the same entry point used after RESET. This jump is visible on the far-right side of Figure 5-28 as a burst code read.

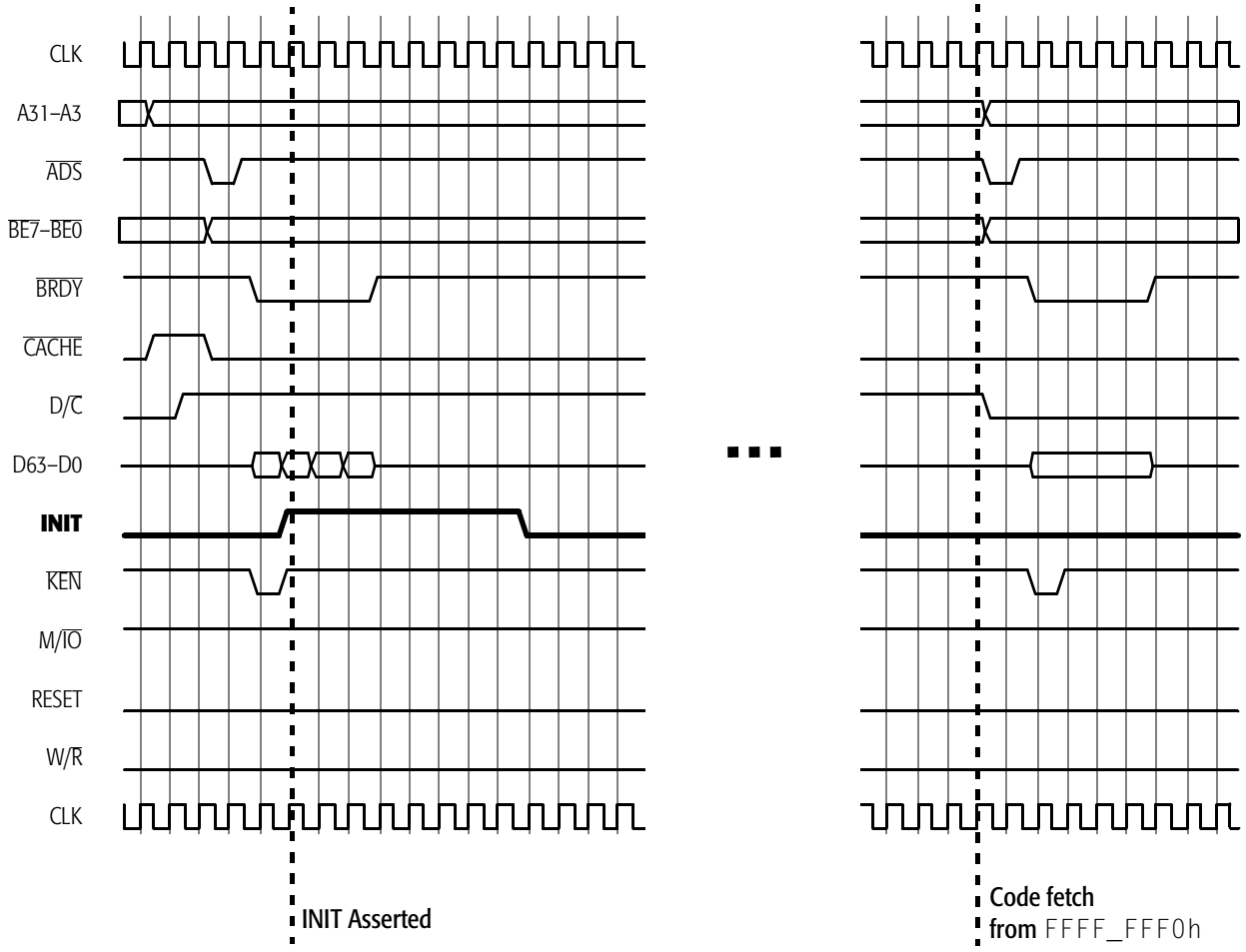


FIGURE 5-28. INIT-Initiated Transition from Protected Mode to Real Mode

6

System Design

This chapter summarizes topics that may be of help to system board designers. The discussions touch on the design of memory, cache, System Management Mode (SMM), clock control (power management), and a few other topics. Many of the details that relate to this subject are also covered in Chapter 5, which describes the processor's signals and bus cycles not only from the processor's view, but also from the system's view.

Throughout this chapter, the term *clock* refers both to the processor's internal clock and to the bus clock (CLK). Thus, each type of clock is explicitly differentiated in the descriptions that follow.

6.1 Memory

The processor can be configured for memory bus speeds of 50, 60, or 66 MHz. Main memory can be built from Page-mode or EDO (extended data out) DRAM. On a 66-MHz bus, the read-cycle time for a page hit in EDO DRAM is 7-2-2-2 (7 clocks for the first transfer and 2 clocks for each remaining transfer) and 10-2-2-2 for a page miss. The read-cycle time for a page-hit Page-mode DRAM at 66 MHz is 7-4-4-4 and 10-4-4-4 for a page miss. On a 50-MHz bus, there is no change in timing for EDO DRAM but Page-mode DRAM timing becomes 6-3-3-3 for a page hit and 8-3-3-3 for a page miss.

6.1.1 Memory Map

Figure 6-1 shows a typical physical memory map for a DOS-based desktop system after DOS boots. Various regions of this memory map to RAM or ROM on the motherboard and adapter boards. The processor hardware imposes only two constraints on the physical memory map implemented by system hardware—the boot address at FFFF_FFF0h, which is accessed when RESET or INIT is asserted, and the default addresses for SMM. However, other physical memory mapping requirements are imposed by BIOS, the operating system, and the specific hardware implemented for the system. In general, the conventions for hardware memory mapping for DOS-based desktop systems include the following:

- Memory-decoder aliasing of boot ROM space
- Cacheable vs. noncacheable address spaces
- SMM memory address space (optional)

Each of these issues is summarized briefly in the sections that follow.

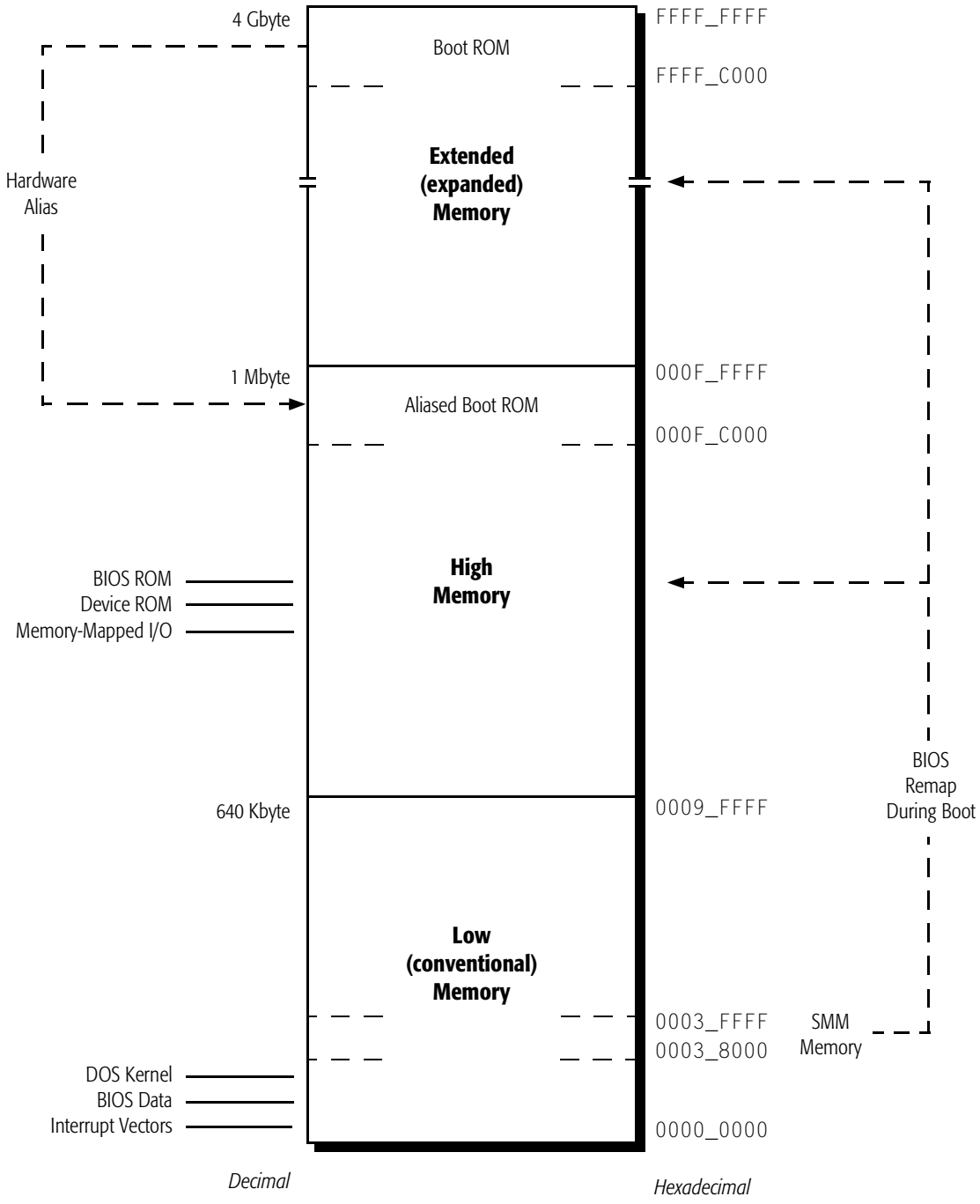


FIGURE 6-1. Typical Desktop-System BIOS Memory Map

6.1.2 Memory-Decoder Aliasing of Boot ROM Space

The processor boots in Real mode at address FFFF_FFF0h. However, because the boot ROM space must be accessed after the first far jump in the processor's Real mode, which generates 20-bit addresses in the space below 1 Mbyte, the address decoder typically aliases the 16-Kbyte physical boot ROM space located between FFFF_FFFFh and FFFF_C000h to the top of the high memory space, between 000F_FFFFh and 000F_C000h, as shown in Figure 6-1.

This reset-address behavior is due to the special way in which segment translation is performed in the x86 architecture when RESET or INIT is asserted. Normally, a Real-mode 16-bit segment selector is shifted left 4 bits to form the segment base, and then added to the 16-bit offset to produce a 20-bit address. Thus, F000:FFF0 in the *selector:offset* format becomes a segment base of 000F_0000h added to an offset of 0000_FFF0h, yielding the physical address 000F_FFF0h. When RESET or INIT is asserted, however, the left-shift is not done and the high 16 address bits are all set to 1, yielding the physical address FFFF_FFF0h. Thereafter, address translation only begins to work in the normal Real-mode manner when the first far jump is executed. This jump loads the code-segment register with a 16-bit segment selector, and this selector-load causes the address-translation mechanism to begin working in its normal Real-mode manner.

The system-logic address decoder must make this behavior transparent to software by aliasing the physical address FFFF_FFF0h to the physical address 000F_FFF0h. As stated above, it normally does this by aliasing the entire 16-Kbyte block between FFFF_FFFFh and FFFF_C000h to between 000F_FFFFh and 000F_C000h.

6.1.3 Cacheable and Noncacheable Address Spaces

When the instruction or data caches are enabled, the processor can fill them with any information found in the system-defined cacheable address space—including code and data for application programs, BIOS, the operation system and its system-level data structures—except that the processor does not fill its instruction or data caches with page directory or page table

entries because these data structures are cached only in CR3 and the TLBs.

System logic normally defines the cacheable address space by implementing external registers which BIOS or other system software initializes during boot with the cacheable (or non-cacheable) ranges of the address space. Lookups in these registers are then used by system logic to control the state of the $\overline{KEN}$ and WB/$\overline{WT}$ input signals. $\overline{KEN}$ controls the caching of memory reads for both the instruction and data caches, and WB/$\overline{WT}$ (together with the PWT bits written by the operating system) controls the MESI state of cacheable read misses and write hits in the data cache.

Most or all of the high memory address range, which lies between 640 Kbyte and 1 Mbyte, is typically specified as non-cacheable by system logic. BIOS ROM is typically hardware-aliased to addresses in this region, and BIOS uses some of the RAM in this region to address locations that should not be cached, such as memory-mapped I/O ports (video, disk, network, and other devices). Thus, system logic typically does not assert $\overline{KEN}$ during accesses to high memory.

System logic can, of course, drive $\overline{KEN}$ so as to specify any other areas of memory as non-cacheable, although this is normally not done.

6.1.4 SMM Memory Space and Cacheability

If the optional System Management Mode (SMM) is implemented, system logic must ensure that, during SMM, all memory accesses are to the SMM memory space rather than to main memory. In general, system designs that do not overlap the address space of SMM memory and main memory are simpler to design and may perform better. Section 6.3 on page 6-23 summarizes the details of SMM. This section deals only with memory usage in SMM.

Figure 6-2 shows the default map of the SMM memory area. It consists of a 64-Kbyte area, between 0003_0000h and 0003_FFFFh, of which the top 32 Kbytes (0003_8000h and 0003_FFFFh) must be populated with RAM. The SMM service-routine entry point is located at 0003_8000h.

During boot, the address decoder must allow BIOS to address the SMM memory area in the main memory address space without entering into the SMM mode in order to initialize it with configuration parameters and the SMM service routine. Thereafter, the BIOS typically remaps the area from its default location in low memory to high or extended memory, as shown in Figure 6-1. After the remapping by BIOS, the address decoder must allow only the processor to access the SMM memory area. Other bus masters must be prevented from accessing it, unless the system design specifically calls for such access.

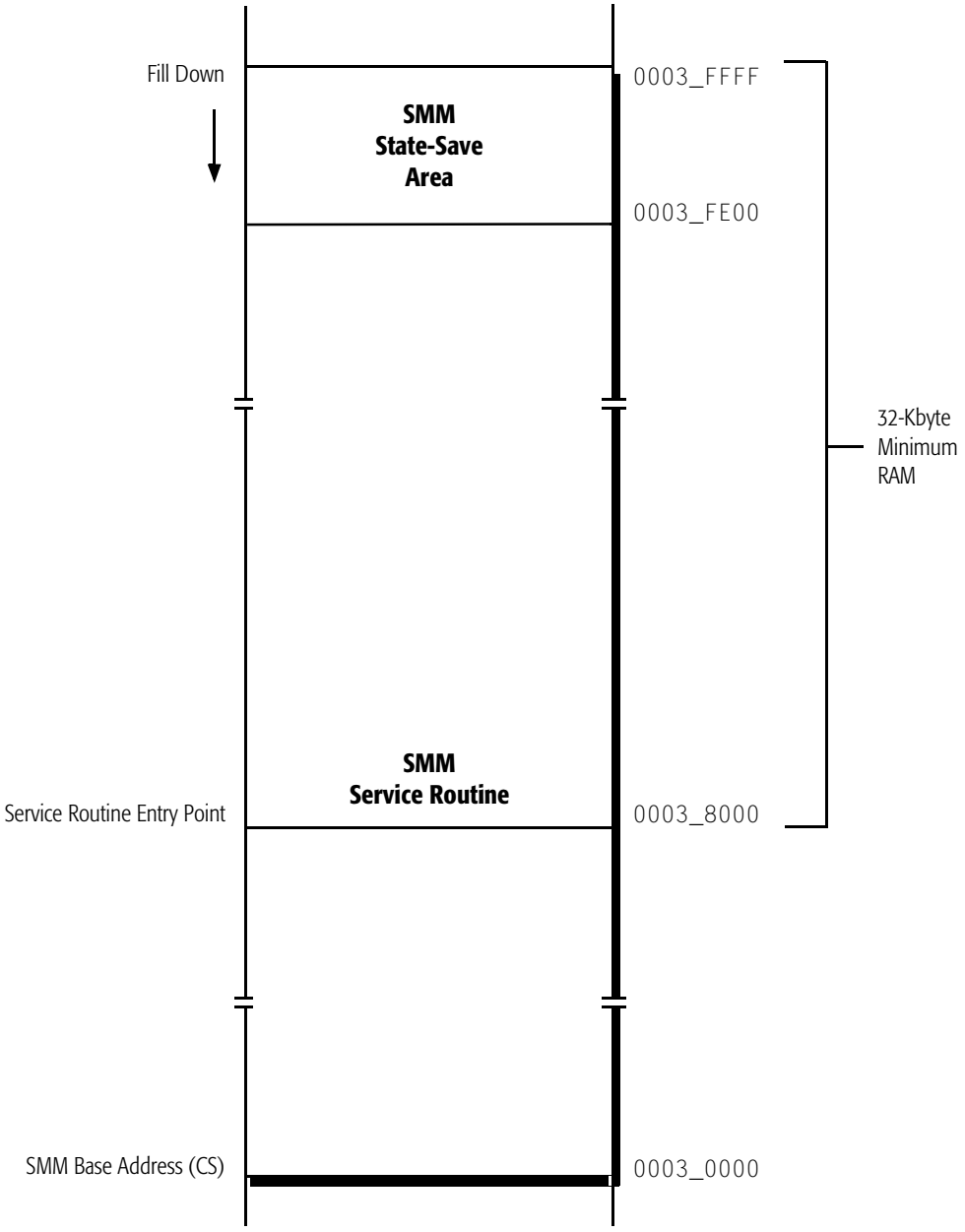


FIGURE 6-2. Default SMM Memory Map

System logic controls the cacheability of SMM memory with $\overline{\text{KEN}}$ in the same way that it controls the cacheability of memory space. If SMM memory is to be non-cacheable, $\overline{\text{KEN}}$ must be held negated from when $\overline{\text{SMI}}$ is asserted until $\overline{\text{SMIACT}}$ is negated. If SMM memory is to be cacheable, $\overline{\text{KEN}}$ must be asserted for cacheable read cycles.

The cacheability of SMM memory has both advantages and disadvantages. By caching SMM memory, the advantage of faster repetitive accesses is offset by delays due to overwriting cache lines that may otherwise be reusable after returning from SMM. If the program that was running prior to entry into SMM ran out of the cache, and the same program continued to run after the return from SMM, the processor would need to refill the caches with the same information after returning from SMM. If an SMM routine frequently accesses the same locations, the delays due to cache refills and writeback-invalidates may be worthwhile. But if an SMM routine seldom accesses the same locations, the speed of returning and continuing on with the prior program might be improved by not caching SMM memory.

If SMM memory space overlaps main memory space that is cacheable, $\overline{\text{FLUSH}}$ must be asserted when $\overline{\text{SMI}}$ is asserted so that memory accesses in SMM do not hit locations cached from main memory. If SMM memory is to be cacheable, $\overline{\text{FLUSH}}$ must also be asserted with $\overline{\text{SMI}}$ when entering SMM, and the SMM service routine must execute the WBINVD instruction to invalidate the caches just prior to executing the RSM instruction, which returns the processor from SMM. The use of $\overline{\text{FLUSH}}$ or WBINVD adds potentially significant time to the entering and leaving of SMM.

6.2 Cache

Systems with multiple bus masters that share cacheable memory require methods for controlling access to the bus and controlling the coherency of shared memory. The sections below summarize certain principles and methods used by system logic, in concert with software, to maintain the coherency of the processor's level-1 (or *L1)* on-chip caches and optional level-2 (or *L2)* external cache.

The internal architecture of the processor's L1 instruction and data caches is described in Section 2.3 on page 2-13. The operating system writes the cache disable (CD) and not-writethrough (NW) bits in CR0 to enable and disable caching, independent of hardware. Thereafter, the operating system may write the PCD and PWT bits in the page directory and

page table entries to control caching properties for specific physical pages. The PCD and PWT bits control the state of the PCD and PWT output signals, which system logic can use to control L2 caching.

6.2.1 L2 Cache

To improve system performance, an L2 cache can be added between the processor and main memory. The L2 cache can be implemented for 3-2-2-2 bursts using 15-ns asynchronous SRAM on a 60-MHz or 66-MHz bus. Faster bursts can be implemented with synchronous SRAM. 9-ns SSRAM can achieve 3-1-1-1 bursts at 66 MHz and 10-ns SSRAM can achieve 2-1-1-1 bursts at 50 MHz.

Most system designs that implement an L2 cache do so using (a) an L2 cache that is significantly larger than the combined sizes of the L1 caches, (b) L2 cache lines that are at least as wide as L1 cache lines (32 bytes or more), and (c) cache-line fills that follow the principle of *inclusion*, which says that any line in the L1 cache is guaranteed to be in the L2 cache.

The first principle (L2 cache bigger) guarantees that the L2 cache will have data that is not already in the L1 cache. The second principle (L2 cache line size greater or equal to L1 cache line size) can simplify and speed up transfers from the L2 cache to the L1 cache. The third principle (inclusion) can simplify and speed up cache-coherency signaling for inquire cycles—if an inquire cycle misses in the L2 cache, the system can safely assume it is not in the L1 cache without having to query the processor directly.

6.2.2 Cacheability and Cache-State Control

The PCD bits maintained by the operating system are a determining factor in the state of the processor's $\overline{\text{CACHE}}$ output signal for each bus cycle. $\overline{\text{CACHE}}$ indicates the processor's intent to drive a read or write cycle as a burst cycle. The signal is only asserted for reads that the operating system determines to be cacheable, and for writebacks of *modified* lines. These writebacks can be caused by inquire cycles, internal snoops, the $\overline{\text{FLUSH}}$ signal, the WBINVD instruction, or cache-line replacements. $\overline{\text{CACHE}}$ is not asserted for cache hits that are

writethroughs, which are driven as single writes rather than burst writes.

From the system's viewpoint, the cacheability of bus cycles is controlled by the $\overline{\text{KEN}}$ and WB/$\overline{\text{WT}}$ inputs, as described in Section 6.1.3 on page 6-4. During reads, system logic can use the assertion of $\overline{\text{CACHE}}$ to initiate a table lookup of cacheable addresses. Such lookups are not normally necessary during writebacks, because the location (having already been cached) is known to be cacheable and $\overline{\text{KEN}}$ has no effect on the processor during writes (only during reads).

The MESI state of a cache-line fill (read miss) or a write hit to a *shared* line is determined by the states of the PWT bits and the WB/$\overline{\text{WT}}$ input signal. The MESI-state transitions for reads and writes are given in Table 2-2 on page 2-19. Complete descriptions of the signals that control cacheability and cache coherency are given on the following pages:

- $\overline{\text{CACHE}}$—Section 5.2.15 on page 5-50
- $\overline{\text{EADS}}$—Section 5.2.20 on page 5-59
- $\overline{\text{HIT}}$—Section 5.2.25 on page 5-72
- $\overline{\text{HITM}}$—Section 5.2.26 on page 5-74
- INV—Section 5.2.33 on page 5-89
- $\overline{\text{KEN}}$—Section 5.2.34 on page 5-90
- PCD—Section 5.2.39 on page 5-100
- PWT—Section 5.2.43 on page 5-106
- WB/$\overline{\text{WT}}$—Section 5.2.56 on page 5-134

6.2.3 Writethrough vs. Writeback Coherency States

The terms *writethrough* and *writeback* apply to two related concepts in a read/write cache like the processor's L1 data cache or an L2 cache. The following conditions apply to both the writethrough and writeback modes:

- *Memory Writes*—There is a relationship between memory writes and their concurrence with cache updates:
  - A memory write that occurs *concurrently* with a cache update to the same location is a *writethrough*. Writethroughs are driven as single cycles on the bus.

- A memory write that occurs *after* a previous cache update to the same location is a *writeback*. Writebacks are driven as burst cycles on the bus.

■ *Coherency State*—There is a relationship between MESI coherency states and writethrough-writeback coherency states of lines in the cache:

- *shared* MESI lines are in the *writethrough* state
- *modified* and *exclusive* MESI lines are in the *writeback* state

Table 2-2 on page 2-19 gives an overview of cache-access states from the viewpoint of both memory writes and coherency state. Chapter 5 deals with memory writes. This section deals with the coherency state of cache lines.

Typically, system logic participates in the coherency control of individual data-cache lines during read misses and write hits to *shared* lines by driving WB/$\overline{\text{WT}}$ as shown in Tables 5-17 and 5-18 on page 5-136. The PWT bit also enters into this control, but it is written by the operating system rather than system logic. Alternatively, system logic can force the on-chip data cache to statically observe a writethrough or a writeback protocol by tying WB/$\overline{\text{WT}}$ as follows:

■ *Writethrough Protocol*—Tie WB/$\overline{\text{WT}}$ Low

■ *Writeback Protocol*—Tie WB/$\overline{\text{WT}}$ High

In the writethrough protocol, a cache line is either in the *shared* or *invalid* state. All write hits to *shared* lines in the data cache also cause 1-to-8-byte writethroughs to memory. Thus, in writethrough cache lines, the MESI protocol is not fully observed—the line never transitions to the *exclusive* or *modified* MESI states. In the writeback protocol, by contrast, a cache line can be in the *shared, exclusive*, *modified,* or *invalid* MESI state. Write hits only cause writethroughs to memory if the hit is to a *shared* line. Writebacks can be caused by inquire cycles, internal snoops, the $\overline{\text{FLUSH}}$ signal, the WBINVD instruction, or cache-line replacements.

The advantages and disadvantages of these modes are as follows:

■ *Writethrough Protocol*:

- Repetitive writes to the same location are slower than in writeback mode.
- No updates to the data cache are hidden from the system.
- When returning from SMM with SMM memory cacheable, there is no need to write back modified lines in the data cache, so the mode transition may be faster. (Both caches, however, must be invalidated.)

■ *Writeback Protocol*:

- Repetitive writes to the same location are faster than in writethrough mode.
- Updates that hit *exclusive* or *modified* lines in the data cache are hidden from the system.
- When returning from SMM, in which SMM memory is cacheable, modified lines in the data cache must be written back before invalidating both caches, so the mode transition may be slower.

In single-processor systems with no other caching master, WB/$\overline{\text{WT}}$ is typically tied High. This allows the processor to cache all cacheable reads in the *exclusive* state, and all cacheable writes update only the cache. In systems with multiple caching masters, WB/$\overline{\text{WT}}$ can be generated after inquire cycles to all other caching masters by the logical OR of $\overline{\text{HIT}}$ from all of the masters. This allows the processor to cache reads in the *exclusive* or *modified* state only if no other master has a copy.

The write-once protocol, as described in Section 6.2.6 on page 6-19, combines the system visibility features of pure writethrough and writeback protocols. While the writeback function can support higher performance in systems with a single caching master, the writethrough function is required for certain transitions in the write-once protocol in systems with multiple caching masters.

6.2.4 Inquire Cycles

System logic maintains coherency between external caching devices and the processor's internal caches by driving inquire cycles to the processor during shared-memory accesses by other caching masters. Inquire cycles are often called *snoops* or *invalidations*, but these terms are too general to clearly differ-

entiate the function of an inquire cycle from the functions of snoops and invalidations that work and/or are initiated in quite different ways (see the preface for a short list of definitions). For example, the AMD5$_K$86 and Pentium processors support only inquire cycles and internal snoops to their L1 cache. They do not support continuous address bus watching.

The processor responds to inquire cycles by looking up the inquire address in its physical tags. The physical-tag lookups are done in parallel with the linear-tag lookups that support program execution, so inquire cycles do not normally affect processor performance. Even when inquire cycles hit *modified* lines, which require writebacks to memory, only the processor's use of the bus is potentially affected. It can normally continue to operate out of its cache during a writeback.

Inquire cycles are initiated with $\overline{\text{EADS}}$, INV, and an inquire address on A31–A5. In response, the processor asserts $\overline{\text{HIT}}$ if the inquire cycle address matches the address of a valid line in the instruction or data cache, or it asserts both $\overline{\text{HIT}}$ and $\overline{\text{HITM}}$ if the address matches a *modified* line in the data cache. If $\overline{\text{HITM}}$ is asserted, the processor writes the *modified* line back to memory. If INV was asserted with $\overline{\text{EADS}}$, a hit invalidates the line. If INV was negated with $\overline{\text{EADS}}$, a hit leaves the line in the *shared* state, or transitions it from the *modified* to *shared* state. On the AMD5$_K$86 processor, the maximum inquire or invalidation rate with inquire cycles is one every two clocks, because $\overline{\text{HIT}}$ and $\overline{\text{HITM}}$ change state two clocks after $\overline{\text{EADS}}$, and $\overline{\text{EADS}}$ can be asserted in the same clock in which $\overline{\text{HITM}}$ is negated.

The MESI-state transitions for inquire cycles, internal snoops, and cache invalidations are given in Table 2-3 on page 2-20 and Table 5-11 on page 5-73.

System logic typically drives inquire cycles to the processor during memory accesses by another bus master. If the processor has a look-through L2 cache, inquire cycles need be driven to the processor only when a prior inquire cycle hits in the processor's L2 cache, or during line replacements in the processor's L2 cache. To implement inquire cycles to the processor or L2 cache for every memory access by another caching master, system logic can generate $\overline{\text{EADS}}$ using the equivalent of $\overline{\text{ADS}}$ from the other caching master.

Inquire cycle logic in systems with look-aside caches can be simplified by monitoring only $\overline{\text{HITM}}$ and ignoring $\overline{\text{HIT}}$. This works because the resulting state of a hit line is determined *only* by the state of INV during the inquire as follows:

- If INV is *negated* during a hit, the hit line—whether *shared, exclusive* or *modified*—transitions to the *shared* state. Thus, the inquiring master can safely cache the same data in the *shared* state without knowing whether the inquire cycle hit in the processor's cache (and thus, without system logic monitoring $\overline{\text{HIT}}$).
- If INV is *asserted* during a hit, the hit line—whether *shared, exclusive* or *modified*—transitions to the *invalid* state. For *modified* lines, the invalidation occurs after a writeback.
- If the inquire cycle misses, irrespective of the state of INV, the inquiring master can cache the target data in the *shared* state, although it will not have enough information to cache that line in the *exclusive* state (this requires that $\overline{\text{HIT}}$ be monitored).

Lookaside caches must implement a signal with which to inform the memory controller that a processor access or an inquire cycle hit the L2 cache, so as to disable the memory from responding. A version of $\overline{\text{HIT}}$ can be implemented for this purpose.

Inquire cycle logic in systems with a look-through L2 cache normally monitor both $\overline{\text{HIT}}$ and $\overline{\text{HITM}}$ from the processor, because such systems often implement the write-once cache protocol. This protocol requires caching in the *exclusive* state at certain transitions, and the *exclusive* state can only be identified if both $\overline{\text{HIT}}$ and $\overline{\text{HITM}}$ are monitored.

6.2.5 Bus Arbitration for Inquire Cycles

Before running an inquire cycle, system logic must obtain control of the address bus by asserting AHOLD, $\overline{\text{BOFF}}$, or HOLD. These signals provide access to the bus with differing conditions and speed.

In most systems, the choices are between $\overline{\text{BOFF}}$ and AHOLD. Due to its slow response time, HOLD is usually considered only when backward compatibility with prior-generation subsystems requires it or when the integrity of in-progress bus

cycles is of paramount importance. Support for $\overline{\text{BOFF}}$ is usually needed to resolve potential deadlock problems that arise as a result of inquire cycles, and if $\overline{\text{BOFF}}$ is supported, there is usually no reason to support HOLD. The sections that follow further describe these relative advantages and disadvantages.

$\overline{\text{BOFF}}$ Arbitration

$\overline{\text{BOFF}}$ obtains control of the full bus (address and data) in the next clock, intervening in any in-progress bus cycle if necessary. It provides the fastest response of the three bus-hold inputs. The processor floats its outputs in the next clock after the assertion of $\overline{\text{BOFF}}$. Thus, the signal can also be used not only for inquire cycles but also to resolve deadlock between two bus masters during inquire cycles.

$\overline{\text{BOFF}}$ is useful, and often necessary, in both single-bus and multiple-bus systems. Because of its ability to help resolve deadlock during shared-memory accesses to cached locations, it is required in virtually all systems with multiple caching masters. For example, if Master A controls the bus and attempts to write a memory location that is cached by Master B in a *modified* state, a shared L2 controller could drive an inquire cycle to Master B, forcing a writeback. But Master B cannot write back until Master A is off the bus. In this case, the L2 controller could use $\overline{\text{HITM}}$ from Master B to gate the assertion of $\overline{\text{BOFF}}$ to Master A.

System logic typically drives separate $\overline{\text{BOFF}}$ signals to each bus master in the system. The assertion by system logic of $\overline{\text{BOFF}}$ to a shared L2 cache for an inquire cycle need not interfere with the processor's continued operation out of its L1 cache. In addition, the assertion by system logic of $\overline{\text{BOFF}}$ to a look-through L2 cache for an inquire cycle need not interfere with the processor's continued accesses to that L2 cache.

Figure 6-3 shows an example of $\overline{\text{BOFF}}$ in a system with two caching masters—a processor and another caching master—sharing the processor bus. A typical sequence for inquire cycles that hit a *modified* line in the processor's cache might be as follows:

1. The other master (or system logic) asserts $\overline{\text{BOFF}}$ to the processor.
2. The other master (or system logic) drives an inquire cycle (represented by $\overline{\text{EADS}}$) to the processor.

3. The processor responds with $\overline{\text{HITM}}$ to system logic.
4. System logic asserts $\overline{\text{BOFF}}$ to the requesting master. ($\overline{\text{HITM}}$ from the processor can be used to generate $\overline{\text{BOFF}}$.)
5. The other master negates $\overline{\text{BOFF}}$ to the processor so that the processor can write back its *modified* line to main memory and the shared L2 cache.

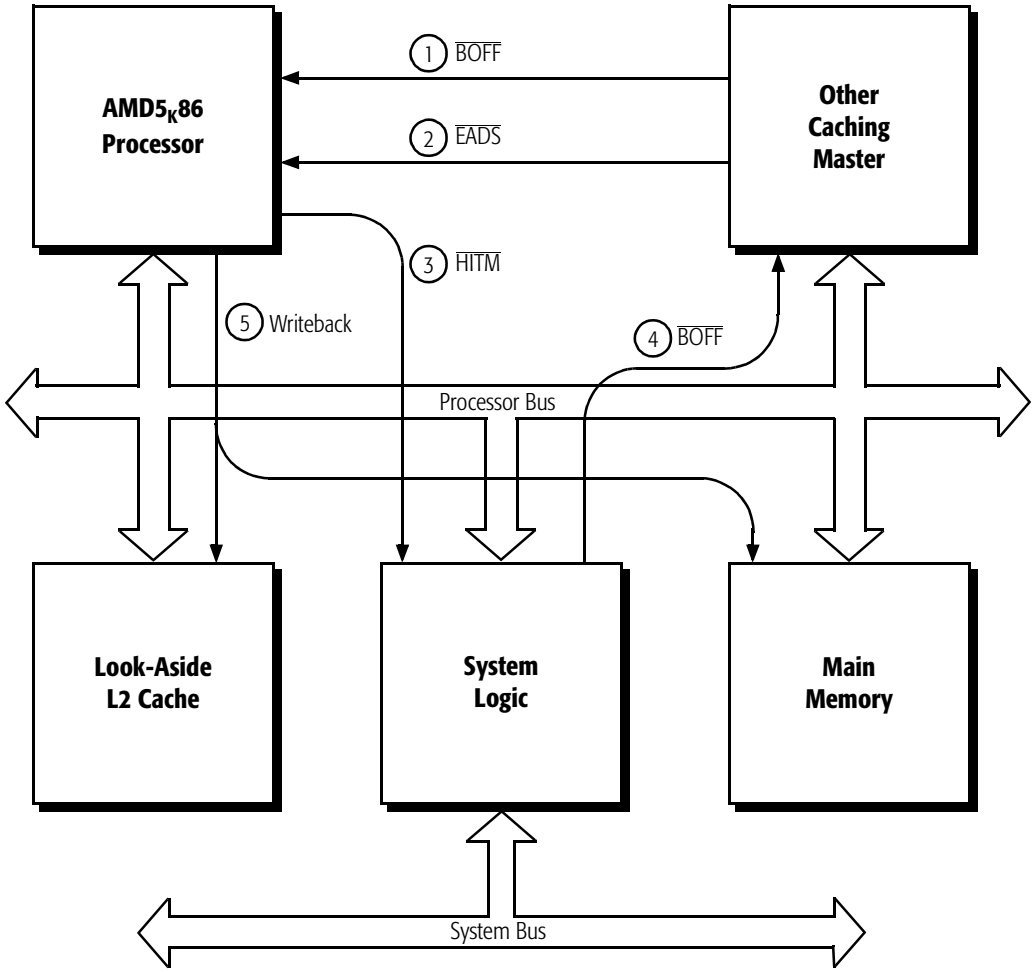


FIGURE 6-3. $\overline{\text{BOFF}}$ Example

A configuration in which both caching masters were on opposite sides of a shared L2 look-through cache would have somewhat similar operations, except that the L2 cache controller would do much of the signalling ascribed to system logic in Figure 6-3.

AHOLD Arbitration

AHOLD's sole function is to support inquire cycles. The assertion of AHOLD by system logic only gets control of the address bus, leaving the data bus available to the processor for the completion of an in-progress bus cycle. If an inquire cycle hits a *modified* line while AHOLD is asserted, the writeback can occur while AHOLD is either asserted or negated.

AHOLD is useful primarily in systems with multiple buses and multiple bus masters, where operations can occur on the separate buses independently and in parallel, and system logic would drive separate AHOLD signals to each caching master. This configuration occurs, for example, if the processor shares its bus only with a look-through L2 cache, and other caching masters work in parallel on a system bus that is isolated by system logic from the L2 cache controller. Figure 6-4 shows such a design.

A typical sequence for inquire cycles that hit *modified* lines in the processor's cache might be as follows:

1. The master on the system bus requests access to memory.
2. System logic responds by asserting $\overline{\text{BOFF}}$ to the processor's L2 cache controller.
3. System logic drives an inquire cycle (represented by $\overline{\text{EADS}}$) to the L2 controller.
4. The L2 controller responds with $\overline{\text{HITM}}$ to system logic (assuming the addressed location is cached by the L2).
5. System logic asserts $\overline{\text{BOFF}}$ to the requesting master on the system bus. ($\overline{\text{HITM}}$ from the L2 controller can be used to generate $\overline{\text{BOFF}}$ to the other master.)
6. The L2 controller asserts AHOLD to the processor.
7. The L2 controller drives an inquire cycle (represented by $\overline{\text{EADS}}$) to the processor.
8. The processor responds with $\overline{\text{HITM}}$ to the L2 controller, indicating that the processor may have a later copy of the location than does the L2 cache.
9. System logic negates $\overline{\text{BOFF}}$ to the L2 cache controller so that the processor can write back its *modified* line to memory and the L2 cache.

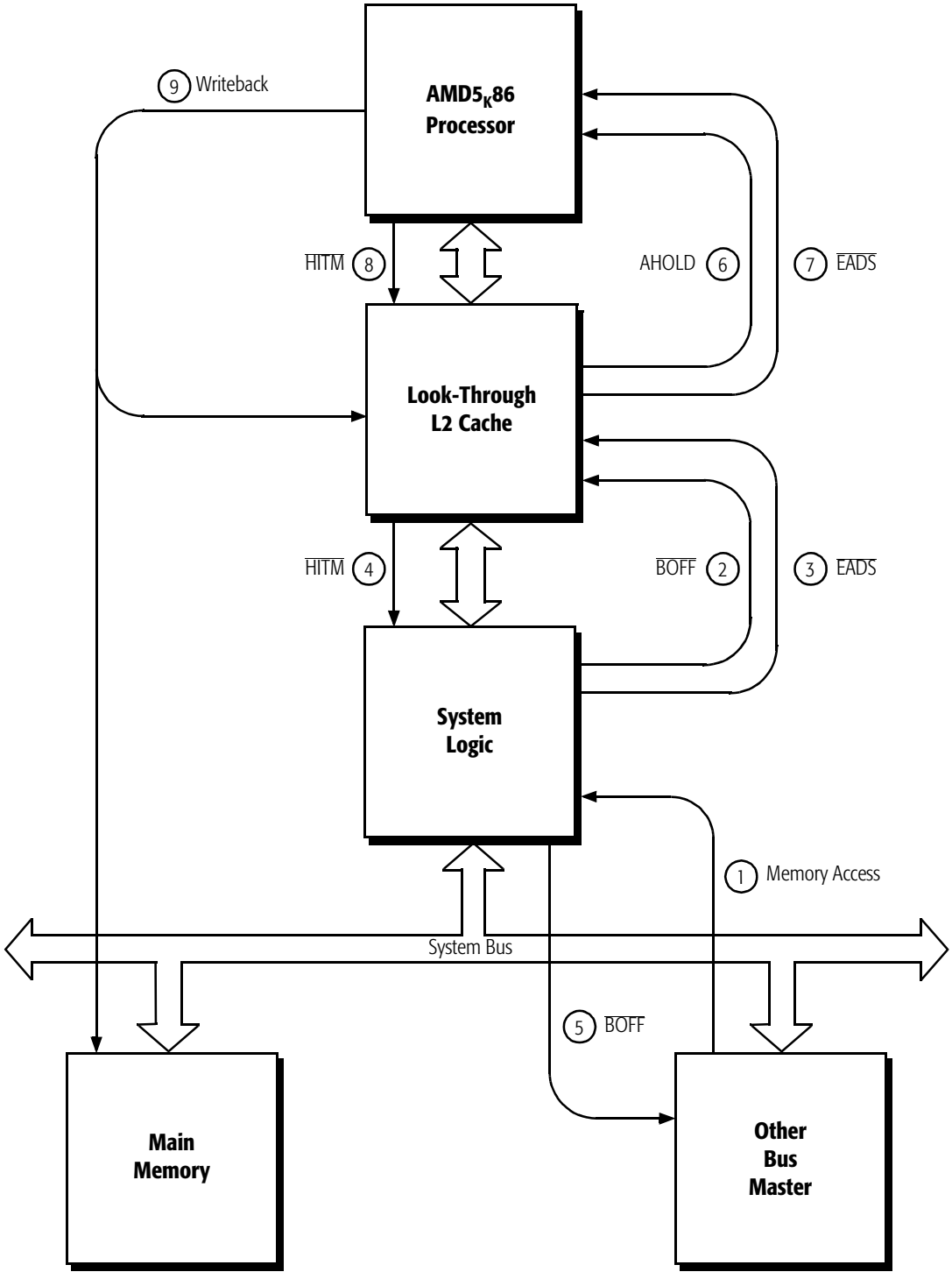


FIGURE 6-4. AHOLD and $\overline{\text{BOFF}}$ Example

HOLD Arbitration System logic can use the HOLD (request) and HLDA (acknowledge) protocol to gain control of the address and data buses. Like $\overline{\text{BOFF}}$, HOLD/HLDA gains control of both the address and data buses but only after the processor completes any in-progress bus cycle or a sequence of cycles, like a locked cycle. However, unlike $\overline{\text{BOFF}}$, the HOLD/HLDA protocol cannot resolve deadlock. In systems where deadlock can occur $\overline{\text{BOFF}}$ must be used, and there is no need to support HOLD/HLDA.

6.2.6 Write-Once Protocol

Among the several write protocols that can be implemented by the L1 and L2 caches, the write-once protocol is of special interest for systems in which the processor has an L2 cache on a separate bus from other caching masters. In such designs, the write-once protocol allows caching masters to simultaneously cache *shared* copies of data until one of the masters writes to that location, at which time the writing master can have the data exclusively and other caching masters must invalidate their copies. The protocol allows other masters to determine whether the processor has a *modified* line in its L1 cache by driving an inquire cycle to the L2 cache, and it allows other masters, via inquire cycles, to intervene in the processor's exclusive use of the data.

Figure 6-5 shows an example. System logic drives separate WB/$\overline{\text{WT}}$ input signals to the L1 and L2 cache. During line fills and writes to the L1 cache, the protocol then works as follows:

1. During a read miss, the processor fills a line in the L1. At the same time, system logic (or the L2) fills a line in the L2 with the same data, and drives the WB/$\overline{\text{WT}}$ input Low (writethrough) to both the L1 and L2. This leaves the L1 and L2 caches as follows:

   L1 cache line in the *shared* state

   L2 cache line in the *shared* state

2. During the first write to that line, the processor updates the *shared* line in the L1 and L2, and writes through to memory. At the same time, system logic drives the L1 WB/$\overline{\text{WT}}$ input Low (writethrough) and the L2 WB/$\overline{\text{WT}}$ input High (write-back). This leaves the L1 and L2 caches as follows:

   L1 cache line in the *shared* state

   L2 cache line in the *exclusive* state

The writethrough to memory must be accompanied by an invalidation of this line in any other caching master's cache.

3. During the second write to that line, the processor updates its *shared* line and writes through to the *exclusive* line of the L2 cache. At the same time system logic drives the L1 WB/$\overline{WT}$ input High (writeback), the L2 WB/$\overline{WT}$ input can also be driven but has no effect. This leaves the L1 and L2 caches as follows:

   L1 cache line in the *exclusive* state

   L2 cache line in the *modified* state

   (If the design of the L2 permits line transitions directly from the *shared* to *modified* state, the state transitions in Step 2 can be skipped.)

4. During the next write to that line, the processor updates its *exclusive* line. The WB/$\overline{WT}$ input has no effect. This leaves the L1 and L2 caches as follows:

   L1 cache line in the *modified* state

   L2 cache line in the *modified* state

5. During all subsequent writes to that line, the processor simply updates its *modified* line.

Inquire cycles to the L2 cache that occur between Steps 1 and 3 get a $\overline{HIT}$ but not a $\overline{HITM}$, thus avoiding the need to drive simultaneous or subsequent inquire cycles to the L1 cache. These inquire cycles to the L2 cache are done in parallel with the processor's L1 and L2 accesses, so they do not reduce the processor's performance when it works out of its caches. However, inquire cycles to the L2 cache that occur after Step 3 get a $\overline{HITM}$. In these cases, the L2 cache drives a subsequent inquire cycle to the L1 cache, which may have updated a *modified* copy after the last update to the L2 cache. These inquire cycles to L1 are done in parallel with the processor's own L1 accesses, but they will block the processor's access to the L2 cache.

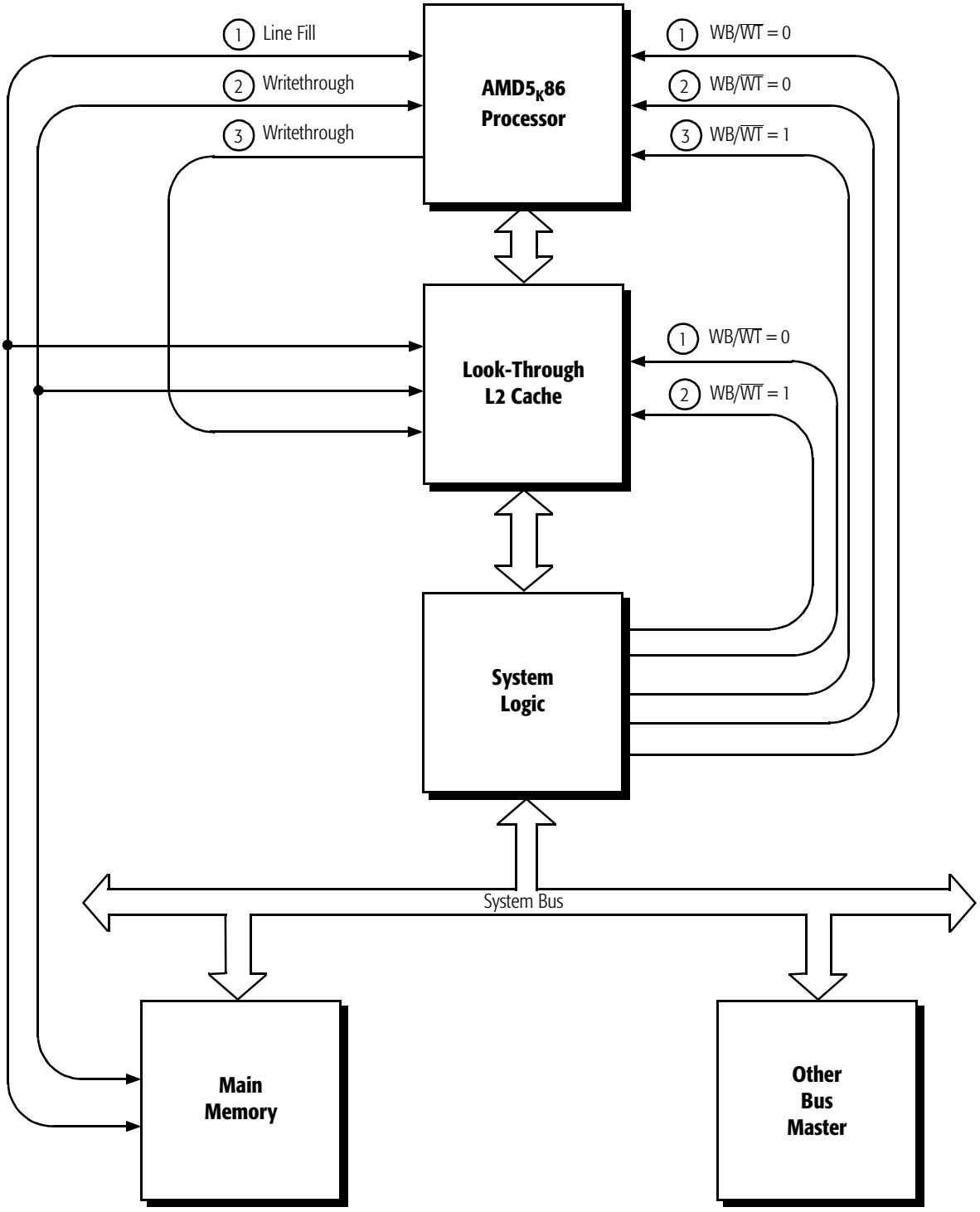


FIGURE 6-5. Write-Once Protocol

6.2.7 Cache Invalidations

The term *invalidation* usually means one of the following things:

- *Individual Cache Lines*—Writebacks and/or invalidations of single lines in the instruction and data caches can be done with inquire cycles (driven by system logic) or internal snoops (initiated by the processor). These invalidations are described in Section 6.2.4 on page 6-12, in the section on Internal Snooping on page 2-22, and elsewhere throughout this manual.
- *Entire Cache Contents*—Writebacks and/or invalidations of the entire contents of the instruction and data caches can be done with the INVD or WBINVD instructions, or with the $\overline{\text{FLUSH}}$ signal. These invalidations are typically performed by the operating system or system logic during task or mode changes. The invalidations are described on pages 5-67 and 5-181.

The MESI-state transitions for cache invalidations are given in Table 2-3 on page 2-20.

6.2.8 $\overline{\text{A20M}}$ Masking of Cache Accesses

The processor samples $\overline{\text{A20M}}$ only in Real mode, and applies $\overline{\text{A20M}}$ masking to its linear cache tags, through which all programs access the caches. Thus, assertion of $\overline{\text{A20M}}$ affects all program-generated cache addresses, including the following:

- Cache-line fills (caused by read misses)
- Cache writethroughs (caused by write misses or write hits to lines in the *shared* state)
- Cache accesses that occur while the processor does not control the bus

However, $\overline{\text{A20M}}$ does not mask writebacks or invalidations caused by the following actions, which are looked up only in the physical (not the linear) tags:

- Internal snoops
- Inquire cycle
- The $\overline{\text{FLUSH}}$ signal

- The WBINVD instruction

Asserting $\overline{\text{A20M}}$ masks Real-mode cache addresses even while the processor does not control the bus. Thus, if another master takes control of the bus and causes the assertion of $\overline{\text{A20M}}$, this masks cache accesses occurring concurrently in the processor. However, it does not affect the correct execution of programs, because linear and physical addresses are identical in Real mode.

The Pentium processor applies masking only to physical addresses, not to linear addresses. This difference between the AMD5$_K$86 and Pentium processors of masking linear vs. physical addresses is not visible to software because linear and physical addresses are identical in Real mode, and the AMD5$_K$86 processor samples $\overline{\text{A20M}}$ only in Real mode.

6.3 System Management Mode (SMM)

SMM is an operating mode entered via an interrupt and performed by an interrupt service routine. It is designed for power management and other system control activities that can occur transparently to conventional operating systems like DOS and Windows. The code and data for SMM are stored in an SMM memory area that should be separate from main memory.

The processor enters SMM when system logic asserts the $\overline{\text{SMI}}$ interrupt and the processor acknowledges it with $\overline{\text{SMIACT}}$, at which point the processor saves its state and jumps to the SMM service routine. The processor returns from SMM when it executes the RSM (resume) instruction from within the SMM service routine. Upon return, the processor picks up where it left off in its prior operating mode, except that special return options are provided when the processor enters SMM from the Halt state or from a trapped I/O instruction, as described in the sections below.

The sections below summarize the SMM state-save area, entry into and exit from SMM, and exceptions and interrupts in SMM. Section 6.1.4 on page 6-5 summarizes memory allocation and addressing in SMM. The $\overline{\text{SMI}}$ and $\overline{\text{SMIACT}}$ signals are described on pages 5-117 and 5-122, respectively.

6.3.1 Operating Mode and Default Register Values

The software environment in SMM has the following features:

- Addressing as in Real mode
- 4-Gbyte segment limit
- Default 16-bit operand, address, and stack size, although instruction prefixes can override these defaults
- Control transfers that do not override the default operand size truncate the EIP to 16 bits
- Far jumps or calls cannot transfer control to a segment with a base address requiring more than 20 bits, as in Real mode segment-base addressing.
- $\overline{\text{A20M}}$ is not recognized (unlike the Pentium processor)
- Interrupt vectors use the Real-mode interrupt vector table (but see Section 6.3.8 on page 6-32)
- The IF flag in EFLAGS is cleared (INTR not recognized)
- The NMI interrupt is disabled
- The TF flag in EFLAGS is cleared (single-step traces disabled)
- Debug register DR7 is cleared (debug traps disabled)

Figure 6-2 on page 6-7 shows the default map of the SMM memory area. It consists of a 64-Kbyte area, between 0003_0000h and 0003_FFFFh, of which the top 32-Kbytes (0003_8000h and 0003_FFFFh) must be populated with RAM. The default code-segment (CS) base address for the area—called the *SMM Base Address*—is at 0003_0000h. The top 512 bytes (0003_FFFFh to 0003_FE00h) contain a fill-down *SMM state-save area*. The default entry point for the SMM service routine is at 0003_8000h.

Table 6-1 shows the initial state of registers when entering SMM.

TABLE 6-1. Initial State of Registers in SMM

Register	Initial Contents			
	Selector	Base	Attributes	Limit
CS	3000h	0003_0000h (see Section 6.3.4)	16-bit, expand-up	4 Gbytes
DS	0000h	0000_0000h	16-bit, expand-up	4 Gbytes
ES	0000h	0000_0000h	16-bit, expand-up	4 Gbytes
FS	0000h	0000_0000h	16-bit, expand-up	4 Gbytes
GS	0000h	0000_0000h	16-bit, expand-up	4 Gbytes
SS	0000h	0000_0000h	16-bit, expand-up	4 Gbytes
General-Purpose	Unmodified			
EFLAGS	0000_0002h			
EIP	0000_8000h			
CR0	Bits 0, 2, 3, 31 cleared (PE, EM, TS, PG). Others are unmodified.			
CR4	0000_0000h			
GDTR	Unmodified			
LDTR	Unmodified			
IDTR	Unmodified			
TR	Unmodified			
DR7	Unmodified			
DR6	Undefined			

6.3.2 SMM State-Save Area

When the processor acknowledges an $\overline{\text{SMI}}$ interrupt by asserting $\overline{\text{SMIACT}}$, it saves its state in the 512-byte SMM state-save area shown in Table 6-2. The save begins at the top of the SMM memory area (SMM Base Address + FFFFh) and fills down to SMM base address + FE00h.

Table 6-2 shows the offsets in the SMM state-save area relative to the SMM base address. The SMM service routine can alter any of the read/write values in the state-save area. The contents of any reserved locations in the state-save area are not necessarily the same between the AMD5$_K$86 processor and the Pentium or 486 processors.

TABLE 6-2. SMM State-Save Area Map

Offset (hex)	Contents	Size (bits)	Type
FFFC	CR0	32	read-only
FFF8	CR3	32	read-only
FFF4	EFLAGS	32	read/write
FFF0	EIP	32	read/write
FFEC	EDI	32	read/write
FFE8	ESI	32	read/write
FFE4	EBP	32	read/write
FFE0	ESP	32	read/write
FFDC	EBX	32	read/write
FFD8	EDX	32	read/write
FFD4	ECX	32	read/write
FFD0	EAX	32	read/write
FFCC	DR6 (FFFF_CFF3h)	32	read-only
FFC8	DR7	32	read-only
FFC4	TR	16 *(upper 16 reserved)*	read-only
FFC0	LDTR	16 *(upper 16 reserved)*	read-only
FFBC	GS	16 *(upper 16 reserved)*	read-only
FFB8	FS	16 *(upper 16 reserved)*	read-only
FFB4	DS	16 *(upper 16 reserved)*	read-only
FFB0	SS	16 *(upper 16 reserved)*	read-only
FFAC	CS	16 *(upper 16 reserved)*	read-only
FFA8	ES	16 *(upper 16 reserved)*	read-only
FFA4	I/O Trap Dword	32 *(See Section 6.3.6)*	read-only
FFA0	reserved	32	—
FF9C	I/O Trap EIP	32	read-only
FF98	reserved	32	—
FF94	reserved	32	—
FF90	IDT Base	32	read-only
FF8C	IDT Limit	16 *(upper 16 reserved)*	read-only
FF88	GDT Base	32	read-only
FF84	GDT Limit	16 *(upper 16 reserved)*	read-only

Notes:

1. Locations marked "reserved" may change in future processors.

2. Writing locations marked as "read-only" has unpredictable results.

TABLE 6-2. SMM State-Save Area Map (continued)

Offset (hex)	Contents	Size (bits)	Type
FF80	TR Attributes	12 *(upper 20 reserved)*	read-only
FF7C	TR Base	32	read-only
FF78	TR Limit	20 *(upper 12 reserved)*	read-only
FF74	LDT Attributes	12 *(upper 20 reserved)*	read-only
FF70	LDT Base	32	read-only
FF6C	LDT Limit	20 *(upper 12 reserved)*	read-only
FF68	GS Attributes	12 *(upper 20 reserved)*	read-only
FF64	GS Base	32	read-only
FF60	GS Limit	20 *(upper 12 reserved)*	read-only
FF5C	FS Attributes	12 *(upper 20 reserved)*	read-only
FF58	FS Base	32	read-only
FF54	FS Limit	20 *(upper 12 reserved)*	read-only
FF50	DS Attributes	12 *(upper 20 reserved)*	read-only
FF4C	DS Base	32	read-only
FF48	DS Limit	20 *(upper 12 reserved)*	read-only
FF44	SS Attributes	12 *(upper 20 reserved)*	read-only
FF40	SS Base	32	read-only
FF3C	SS Limit	20 *(upper 12 reserved)*	read-only
FF38	CS Attributes	12 *(upper 20 reserved)*	read-only
FF34	CS Base	32	read-only
FF30	CS Limit	20 *(upper 12 reserved)*	read-only
FF2C	ES Attributes	12 *(upper 20 reserved)*	read-only
FF28	ES Base	32	read-only
FF24	ES Limit	20 *(upper 12 reserved)*	read-only
FF20	*reserved*	32	—
FF1C	*reserved*	32	—
FF18	*reserved*	32	—
FF14	CR2	32	read-only
FF10	CR4	32	read-only
FF0C	I/O Restart ESI	32	read-only
FF08	I/O Restart ECX	32	read-only

Notes:

1. Locations marked "reserved" may change in future processors.

2. Writing locations marked as "read-only" has unpredictable results.

TABLE 6-2. SMM State-Save Area Map (continued)

Offset (hex)	Contents	Size (bits)	Type
FF04	I/O Restart EDI	32	read-only
FF02	Halt Restart Slot	16 *(See Section 6.3.5)*	read/write
FF00	I/O Trap Restart Slot	16 *(See Section 6.3.7)*	read/write
FEFC	SMM Revision Identifier	32 *(See Section 6.3.3)*	read-only
FEF8	SMM Base Address	32 *(See Section 6.3.4)*	read/write
FE00 - FEF4	*reserved*	32	—

Notes:
1. Locations marked "reserved" may change in future processors.
2. Writing locations marked as "read-only" has unpredictable results.

6.3.3 SMM Revision Identifier

The SMM revision identifier at offset FEFCh in the SMM state-save area specifies the version of SMM and the extensions that are available on the processor. The SMM revision identifier fields are as follows:

- *Bits 31–18—reserved*
- *Bit 17*—SMM base address relocation (always 1 = enabled)
- *Bit 16*—I/O trap restart (always 1 = enabled)
- *Bits 15–0*—SMM revision level = 0000

These fields are the same as in the Pentium processor. Unlike the Pentium processor, however, the I/O trap restart and the SMM base address relocation functions are always enabled in the AMD5$_K$86 processor and do not need to be specifically enabled.

6.3.4 SMM Base Address

During RESET, the processor sets the code-segment (CS) base address for the SMM memory area—the *SMM Base Address*—to its default, 0003_0000h. The SMM base address at offset FEF8 in the SMM state-save area can be changed by the SMM service routine to any address that is aligned to a 32-Kbyte boundary. (Locations not aligned to a 32-Kbyte boundary cause the processor to enter the Shutdown state when executing the RSM instruction.)

If the SMM base address is rewritten, the processor saves its state at the new base address the next time SMM is entered, and each time thereafter, until RESET. The relocated addresses for the SMM memory will then be as follows:

- *SMM base address*—Default: 0003_0000h. Relocated: offset FEF8 in the SMM state-save area (see Table 6-2)
- *Service Routine Entry Point*—SMM base address + 8000h
- *Top*—SMM base address + FFFFh

This SMM base address relocation feature is compatible with the Pentium processor's analogous feature. The following pseudo-code implements a relocatable SMM base address in BIOS:

```
begin
{
if SMI Handler is to be Relocated then
   {
   set SMM Base Address (offset FEF8h) to new value
   resume
   }
else
   {
   SMM execution to begin at relocation area.
   resume
   }
}
end
```

To relocate the SMM base address above the 1-Mbyte limit imposed by Real-mode segment addressing, use the address-override prefix to generate the offset in 32-bit registers. If the SMM base address is relocated to a block below 16 Mbytes, data in the DS segment (which has a segment base of 0000_0000h) can be accessed by the following code:

```
mov ebx,00FExxxxh; 64K segment from 00FE_0000h to 00FE_FFFFh
mov ax, ds:[ebx]
```

6.3.5 Halt Restart Slot

During entry into SMM, the halt restart slot at offset FF02h in the SMM state-save area specifies if SMM was entered from the Halt state. Before returning from SMM, the halt restart slot can be written by the SMM service routine to specify whether the return from SMM should take the processor back to the Halt state or to the instruction-execution state specified by the SMM state-save area.

On entry into SMM, the halt restart slot is configured as follows:

- *Bits 15–1*—Undefined
- *Bit 0*—Point of entry to SMM:

  1 = entered from Halt state.

  0 = not entered from Halt state

Before return from SMM, the halt restart slot can be written as:

- *Bits 15–1*—Undefined
- *Bit 0*—Point of return from SMM

  1 = return to Halt state

  0 = return to state specified by SMM state-save area

The fields of the halt restart slot are the same as in the Pentium processor auto halt restart slot. During entry into and exit from SMM, the processor writes or reads only bit 0 of the 16-bit value although the entire 16 bits can be read or written by the service routine. The Pentium-compatible pseudo-code for implementing the halt restart slot in BIOS is as follows:

```
begin
{
if return to Halt state then
   {
   if SMI# during Halt state then
         set halt restart slot to 00FFh
   }
}end
```

If the return takes the processor back to the Halt state, the HLT instruction is not refetched, but the Halt special bus cycle is driven on the bus after the return.

6.3.6 I/O Trap Dword

If the assertion of $\overline{\text{SMI}}$ is recognized on the boundary of an I/O instruction, the I/O trap dword at offset FFA4h in the SMM state-save area contains information about the instruction. The fields of the I/O trap dword are configured as follows:

- *Bits 31–16*—I/O port address
- *Bits 15–2—reserved*
- *Bit 1*—Valid I/O instruction (1 = valid, 0 = invalid)
- *Bit 0*—Input or output instruction (1 = INx, 0 = OUTx)

The I/O trap dword is related to the I/O trap restart slot, described below. Bit 1 of the I/O trap dword (the valid bit) should be tested if the I/O trap restart slot is to be changed.

6.3.7 I/O Trap Restart Slot

The I/O trap restart slot at offset FF00h in the SMM state-save area specifies whether the assertion of $\overline{\text{SMI}}$ was recognized on the boundary of an I/O instruction, and if so, whether the trapped I/O instruction should be re-executed on return from SMM. This is sometimes called the *I/O-instruction restart* function. Re-executing a trapped I/O instruction is useful, for example, if an I/O write to disk finds the disk powered down. The system logic monitoring such an access can assert $\overline{\text{SMI}}$. Then the SMM service routine would query system logic, find a failed I/O write, take action to power-up the I/O device, enable the I/O trap restart slot feature, and return.

The fields of the I/O trap restart slot are configured as follows:

- *Bits 31–16—reserved*
- *Bits 15–0*—I/O instruction restart on return from SMM:

  0000h = execute the next instruction after the trapped I/O instruction

  00FFh = re-execute the trapped I/O instruction

The processor initializes the I/O trap restart slot to 0000h upon entry into SMM. If SMM was entered due to a trapped I/O instruction, the processor indicates the validity of the I/O instruction by setting or clearing bit 1 of the I/O trap dword at offset FFA4 in the SMM state-save area, as described in Sec-

tion 6.3.6. The SMM service routine should test bit 1 of the I/O trap dword to determine the validity of the I/O instruction before writing the I/O trap restart slot. If the I/O instruction was valid, the SMM service routine can safely rewrite the I/O trap restart slot with the value 00FFh, which causes the processor to re-execute the trapped I/O instruction when the RSM instruction is executed. If the I/O instruction was invalid, writing the I/O trap restart slot has undefined results. If sequential $\overline{\text{SMI}}$ interrupts occur, the second entry into SMM will never have bit 1 of the I/O trap dword set, and the second SMM service routine should not rewrite the I/O trap restart slot.

The pseudo-code for implementing I/O Trap Restart in BIOS is as follows:

```
begin
{
if I/O instruction needs to be restarted then
   {
   if valid I/O instruction (test offset FFA4) then
         set I/O restart slot (offset FF00) to 00FFh
   }
}
end
```

During a simultaneous $\overline{\text{SMI}}$ I/O-instruction trap and debug breakpoint trap, the AMD5$_K$86 processor first responds to the $\overline{\text{SMI}}$ and postpones writing the exception-related information to the stack until after the return from SMM via the RSM instruction. If debug registers DR3–DR0 are used in SMM, they must be saved and restored by the SMM software. The processor automatically saves and restores DR7–DR6. If the I/O trap restart slot in the SMM state-save area is written with the value 00FFh when the RSM instruction is executed, the debug trap does not occur until after the I/O instruction is re-executed.

6.3.8 Exceptions and Interrupts in SMM

When SMM is entered, the processor disables both INTR and NMI interrupts. On both the AMD5$_K$86 and Pentium processors, INTR interrupts are disabled by clearing the IF flag in EFLAGS. But the mechanism by which NMI interrupts are disabled and subsequently recognized differs between the AMD5$_K$86 and Pentium processors.

During SMM, the Pentium processor does not respond to NMI until the beginning of its response to the first INTR or software interrupt (INTn) to occur after entering SMM. NMIs can thus be enabled by using a dummy interrupt. When an INTR or software interrupt is recognized, the processor first responds to a pending NMI interrupt before executing the first instruction of the INTR handler. By contrast, the AMD5$_K$86 processor recognizes a pending NMI interrupt after returning (via the IRET instruction) from a prior interrupt.

The same dummy interrupt used on the Pentium processor to enable NMI recognition during SMM works on the AMD5$_K$86 processor. The only difference is that the AMD5$_K$86 processor responds to the NMI after the IRET of the dummy interrupt whereas the Pentium processor responds at the beginning of the dummy interrupt. All other exceptions and interrupts within SMM are fully compatible with those supported by the Pentium processor in SMM.

The IF flag in EFLAGS is cleared automatically when the processor enters SMM, thus disabling maskable interrupts. The HLT instruction should not be executed in SMM without first setting the IF bit.

Table 5-2 on page 5-9 and Table 5-3 on page 5-17 summarize the behavior of all interrupts in SMM.

6.3.9 SMM Compatibility with Pentium Processor

The differences in SMM functions between the AMD5$_K$86 and Pentium processors are described in Section A.5 on page A-12.

6.4 Clock Control

The processor's consumption of power can be controlled by reducing the frequency of the processor and/or bus clocks when there is no computational or user activity. System logic initiates this control by asserting $\overline{\text{STPCLK}}$, which causes the processor to complete any in-progress bus cycle and enter the Stop Grant state (processor's internal clock stopped), from which system logic can subsequently transition the processor to its Stop Clock state (CLK stopped). These clock control func-

tions can be entered from any of the processor's normal operating modes (Real, Virtual-8086, or Protected mode), from system management mode (SMM), or from the Halt state.

In typical PC systems that implement power control, the $\overline{\text{STPCLK}}$, CLK, and $\overline{\text{SMI}}$ signals are driven by external power management logic that monitors activity on the address and cycle-definition signals. In a typical case, the power management logic may notice that, after having initiated SMM to power down one or more I/O devices, another several minutes have elapsed without activity. Power management logic can again assert $\overline{\text{SMI}}$, the SMM service routine would obtain the relevant information and decide to power itself (the processor) down, and the decision would be communicated to the power management logic, which would assert $\overline{\text{STPCLK}}$ to the processor and, optionally, stop driving CLK to the processor and other logic. For details on $\overline{\text{SMI}}$ and $\overline{\text{STPCLK}}$, see pages 5-117 and 5-123, respectively.

6.4.1 State Transitions

The five states in the processor's clock-control protocol, as shown in Figure 6-6, are as follows:

- *Normal Execution:* Real mode, Virtual-8086 mode, Protected mode, or System Management Mode (SMM). In this state, all clocks run at full speed.
- *Halt State*
- *Stop Grant State*
- *Stop Grant Inquire State*
- *Stop Clock State*

The sections below describe each of the four low-power states.

6.4.2 Halt State

The processor enters the Halt state from the normal operating modes (Real, Protected, or Virtual-8086) or SMM when it executes the HLT instruction. The processor leaves the Halt state and returns to its prior operating mode when RESET, $\overline{\text{SMI}}$, INIT, NMI, or INTR is asserted. If $\overline{\text{STPCLK}}$ is asserted within the Halt state, the processor transitions to the Stop Grant

state, and it returns to the Halt state when $\overline{\text{STPCLK}}$ is negated. No processor registers are saved before entering the Halt state because the processor returns to the next unexecuted instruction in program order when it returns to its prior operating mode. When the processor returns to the Halt state, the HLT instruction is not refetched but the processor drives the Halt special bus cycle on the bus after the return.

Within the Halt state, the processor disables the majority of its internal clock distribution and (if $\overline{\text{STPCLK}}$ is asserted) the internal pullup resistor on $\overline{\text{STPCLK}}$. However, its phase-lock loop still runs, its key internal logic is still clocked, most of its inputs and outputs retain their last state (except D63–D0 and DP7–DP0 which are floated), and it still responds to input signals.

The HLT instruction is commonly executed by modern UNIX-type operating systems as a method of entering an idle loop. The operating system sees that it has no pending processes, therefore nothing to execute, so it executes HLT. Entry into the Halt state achieves the same power-saving effect as entry into the Stop Grant state, but the method is simpler and faster. Entry into the Halt state requires only the execution of the HLT instruction, whereas entry into the Stop Grant state requires that system logic monitor system activity, assert $\overline{\text{STPCLK}}$, and decode the processor's acknowledgment (potentially several clocks later) via the Stop Grant special bus cycle.

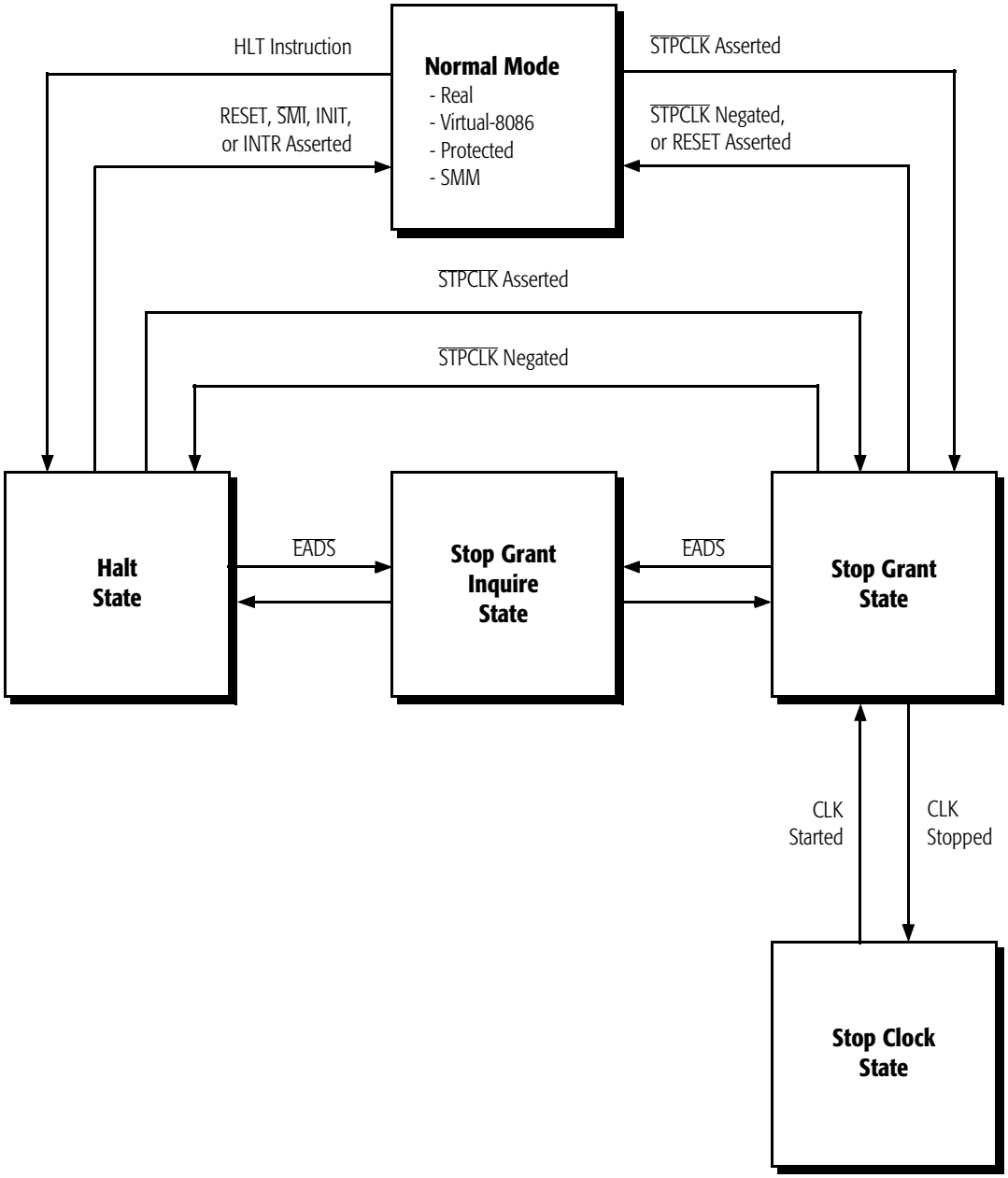


FIGURE 6-6. Clock Control State Transitions

6.4.3 Stop Grant State

The assertion of $\overline{\text{STPCLK}}$ causes the processor to enter the Stop Grant state. The processor can enter the Stop Grant state from the normal operating modes (Real, Protected, or Virtual-8086), SMM, or the Halt state.

When $\overline{\text{STPCLK}}$ is negated, the processor returns to the mode from which it entered. If the processor entered the Stop Grant state from the Halt state, negation of $\overline{\text{STPCLK}}$ returns the processor to the Halt state. Otherwise, negation of $\overline{\text{STPCLK}}$ or assertion of RESET returns the processor to the normal operating mode or SMM, from which it entered. If INIT is asserted in the Stop Grant state, the signal is latched and acted upon after $\overline{\text{STPCLK}}$ is negated. No processor registers are saved before entering the Stop Grant state because the processor returns to the next unexecuted instruction in program order when it returns to its prior operating mode.

Within the Stop Grant state (as in the Halt state) the majority of the processor's internal clock distribution and all internal pullup resistors are disabled. However, its phase-lock loop still runs, its key internal logic is still clocked, most of its inputs and outputs retain their last state (except D63–D0 and DP7–DP0 which are floated), and it still responds to input signals.

6.4.4 Stop Grant Inquire State

An inquire cycle driven while the processor is in the Halt or Stop Grant state causes the processor to transition to the Stop Grant Inquire state. As for inquire cycles driven from any other state, system logic must assert AHOLD, $\overline{\text{BOFF}}$, or HOLD to obtain the address bus before driving $\overline{\text{EADS}}$, INV, and the inquire address.

The processor responds normally to an inquire cycle by driving $\overline{\text{HITM}}$ and/or $\overline{\text{HIT}}$ and performing any necessary cache-state transition. If $\overline{\text{HITM}}$ is asserted, the processor drives a normal writeback (immediately if AHOLD is asserted, or delayed if $\overline{\text{BOFF}}$ or HLDA is asserted) and returns to the state from which it entered the Stop Grant Inquire state in the clock in which it negates $\overline{\text{HITM}}$. If $\overline{\text{HITM}}$ is not asserted, the processor returns from the Stop Grant Inquire state to the state from which it entered, two clocks after $\overline{\text{EADS}}$.

6.4.5 Stop Clock State

The processor enters the Stop Clock state when system logic turns off CLK while $\overline{\text{STPCLK}}$ is asserted. This is the minimum-power state and it can only be entered from the Stop Grant state, after $\overline{\text{BRDY}}$ has been returned for the Stop Grant special bus cycle. In the Stop Clock state, the processor's phase-lock loop and I/O buffers are disabled, except for the I/O buffers on CLK and the JTAG signals. System logic should not change the state of any signals, and the processor does not recognize any signal edges in the Stop Clock state.

When CLK is restarted, the processor returns to the Stop Grant state, responds to inputs in the next clock, but cannot drive bus cycles until its phase-lock loop is synchronized. The latter takes several clocks (see the data sheet for this specification). The CLK can be driven with a different frequency and/or the bus-to-processor clock ratio can be changed on the BF input upon restarting CLK.

6.4.6 Clock Control Compatibility with Pentium Processor

The differences in clock control functions between the AMD5$_K$86 and Pentium processors are described in Section A.5 on page A-12.

6.5 Power and Ground Design

All of the processor input signals operate at 3 V except CLK, which can operate at 3 V or 5 V. Compatible 3-V chipsets are available. If your system operates at 5 V, chipsets that provide 5-V to 3-V voltage translators are available, or you can provide the translators on your system board. (If you use voltage translators, they must be fast enough to support your bus speed.)

Due to the processor's high clock frequency, the package supports many copies of V_{CC} and V_{SS} to prevent power surges when multiple outputs change state simultaneously. In addition, certain precautions must be taken with respect to the AHOLD input. If the processor has a pending bus cycle when AHOLD is negated, all of the address drivers turn on almost

immediately after AHOLD is negated. If the processor is also driving data with $\overline{\text{BRDY}}$ on the data bus at the same time, the processor then drives 96 bits simultaneously and ground-bounce spikes can occur. Such ground-bounce spikes can be avoided by following these two rules with respect to AHOLD:

- Do not negate AHOLD in the same clock that $\overline{\text{BRDY}}$ is asserted during a write cycle.
- Do not negate AHOLD in the same clock that $\overline{\text{ADS}}$ is asserted during a writeback.

In addition to the above restrictions on driving AHOLD, the following general design recommendations apply to power connections between the processor and the system board:

- Connect all V_{CC} pins to a V_{CC} plane on your system board.
- Connect all V_{SS} pins to a GND plane on your system board.
- Do not drive address and data buses into large capacitive loads at high frequencies. This can cause transient power surges.
- Decouple capacitance near the processor.
- Use low-inductance capacitors and circuit paths, and type X7R or better dielectric.
- Use capacitors specifically designed for PGA packages.
- Tie unused inputs High or Low.
- Leave no-connect (NC) pins unconnected.
- Connect active-Low inputs to V_{cc} through a 20-kΩ pullup resistor. This keeps the inputs in a known state while allowing them to be driven during tests.
- Connect active-High inputs to GND through a pulldown resistor.
- Keep trace lengths to a minimum.

6.6 Clock Design

During RESET, the CLK input to the processor should be grounded until V_{cc} has reached its normal operating level and PWRGOOD is asserted. Figure 6-7 shows this timing. After V_{cc} and CLK reach specification, RESET must be asserted for a minimum of 1 ms to allow the digital phase-lock loop to synchronize.

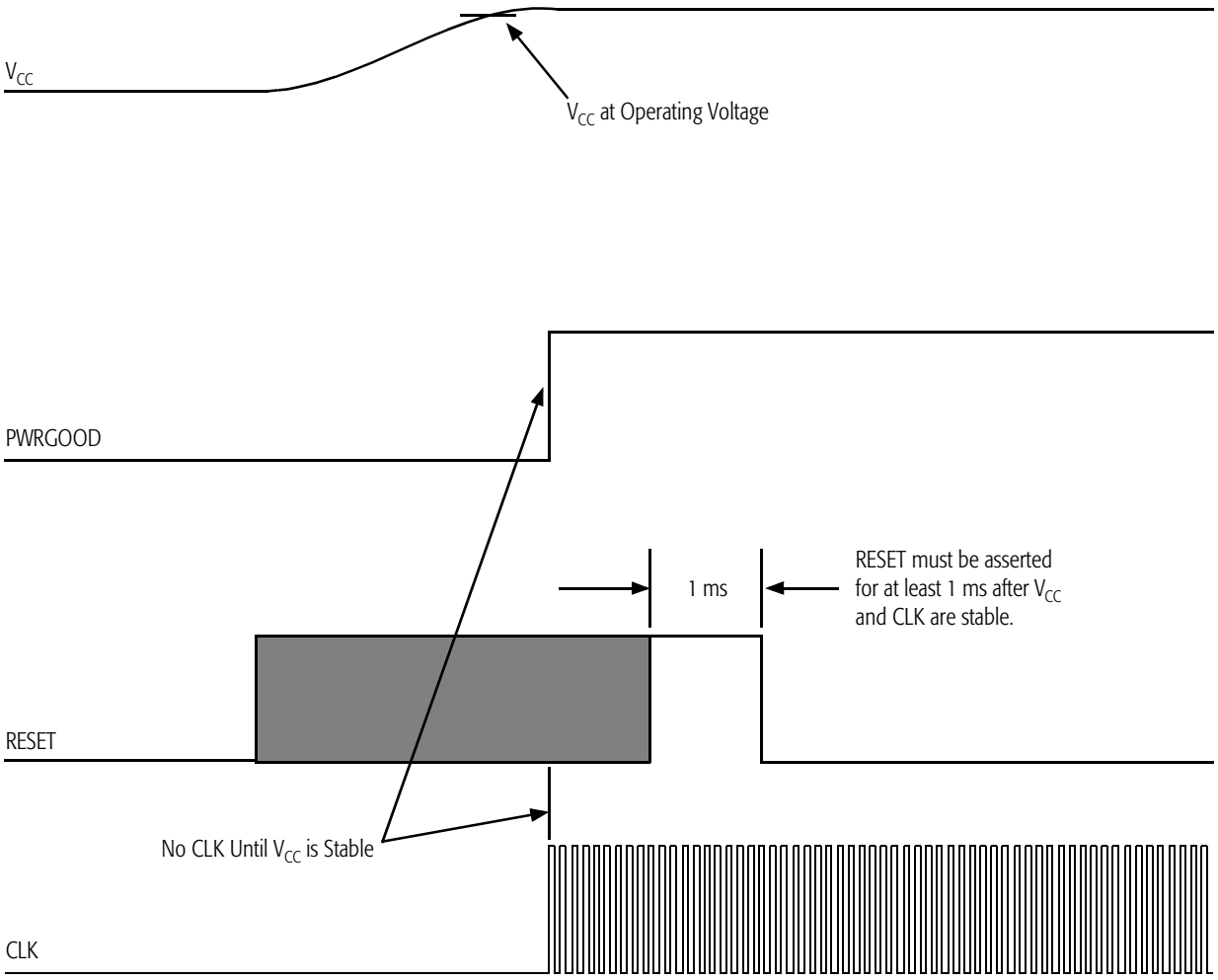


FIGURE 6-7. V_{cc} and CLK

The clock signal to the processor can be gated with one of the following methods:

- *Chipset*—Figure 6-8 illustrates a delay function that gates the system CLK with PWRGOOD to generate the CLK input to the processor (CPUCLK) and RESET. Such a function can easily be implemented by a chipset.
- *Clock Synthesizer with Output Enable*—Figure 6-9 illustrates a clock synthesizer with an $\overline{OE}$ input driven by PWRGOOD.
- *Clock Clamping Circuit*—Figure 6-10 illustrates a clamping circuit that grounds CPUCLK for a predetermined time.

The clock clamping circuit shown in Figure 6-10 has several advantages. In addition to delaying CPUCLK until V_{cc} has reached specification, it also prevents noise glitches on the clock signal from being sensed by the processor during this time. Noise glitches are typically caused by poor design of the clock generator startup circuit, poor layout of the PCB, power supply ringing while V_{cc} is reaching specification, or a long voltage slew rate (such as 100 ms). The integrity of CPUCLK is best maintained by passing CPUCLK directly from the core logic.

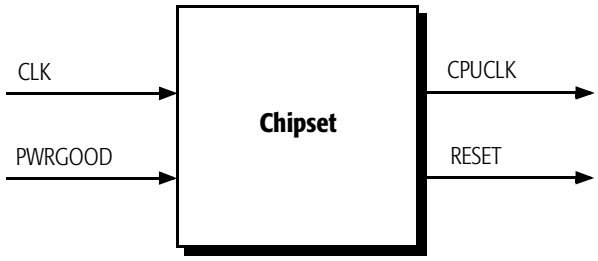


FIGURE 6-8. CLK Delay Function

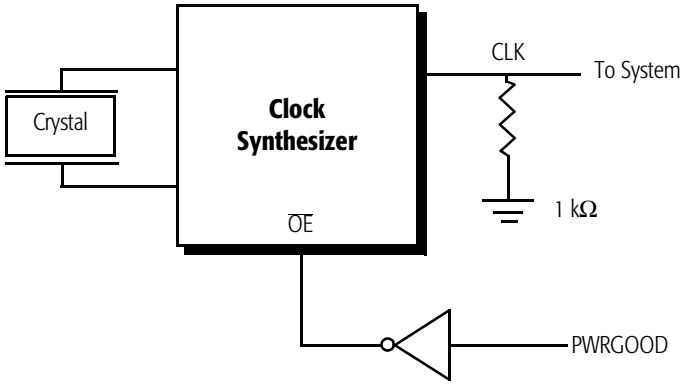


FIGURE 6-9. CLK Synthesizer with Output Enable

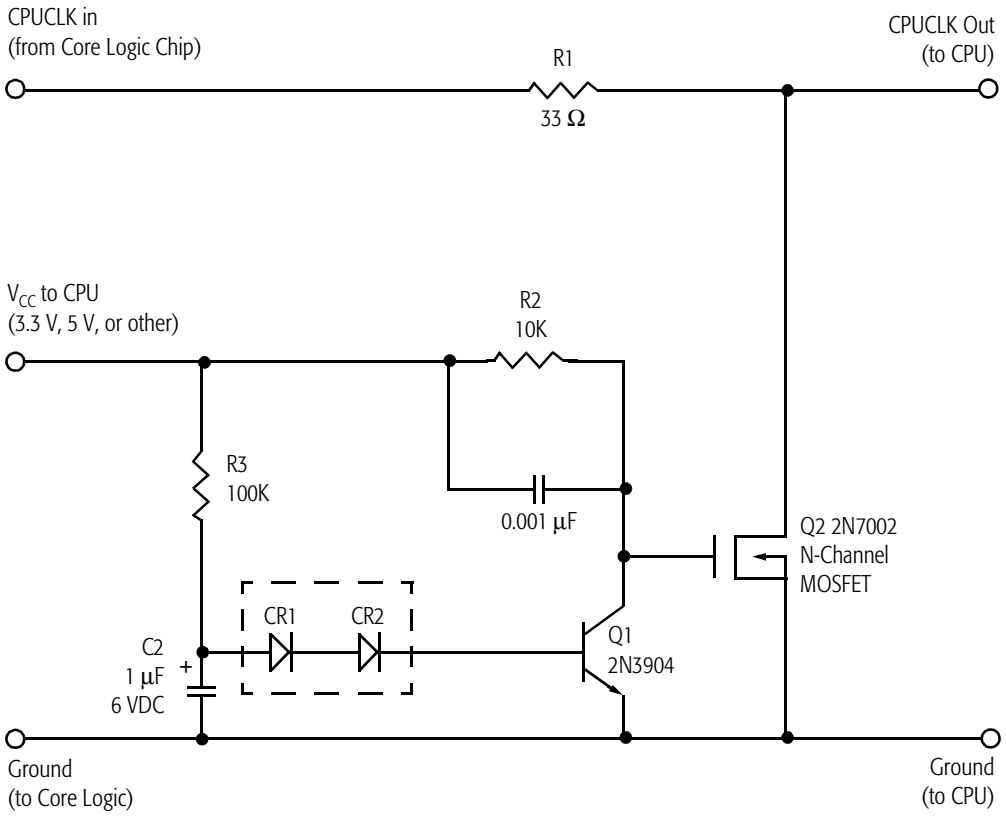


FIGURE 6-10. CPUCLK Clamping Circuit

6.6.1 Noise Reduction

Circuit noise can be minimized by the following design rules:

- *Clock Signal*
  - Place the processor as close as possible to the clock source.
  - Route the CPUCLK signal on a single PCB layer. Do not use vias.
  - Guard-band the CPUCLK signal with twice the minimum pitch width to minimize unwanted cross talk.
- *Capacitors*
  - Place all capacitors as near as possible to the processor.
  - Connect the positive sides of all capacitors through vias directly to the processor power plane.
  - Connect the negative sides of all capacitors through vias to the ground plane.
  - Use tantalum 47 µF and 1 µF capacitors.
  - Use ceramic capacitors with low equivalent series resistance (ESR) ratings at high frequencies and a minimum voltage rating of 6 V for all other capacitor values.
  - Place some capacitors very near to the processor, preferably on the inside perimeter of the processor socket.
  - Connect bypass capacitors on the top side of the PCB directly to the processor's power pins.
- *Multilayer Printed-Circuit Boards*
  - Use a minimum of four layers—one split power plane, one ground plane, two routing planes.
- *Regulator Circuit*
  - Use surface-mounted components placed as near as possible to the processor.
  - Use at least three vias to the +5-V power plane for the input power connection.
  - Use at least three vias to the +3-V processor power plane for the output power connection.

AMD recommends using a split power plane to isolate the processor from the rest of the motherboard. This approach reduces noise without additional PCB planes. The split plane should be made from a portion of copper that is cut out and iso-

lated from the PCB 5-V power plane. This cutout region supplies a separate power source for the processor and allows installation of bypass decoupling capacitors. The capacitors should be placed across the split power plane to provide signal-return termination. The processor power plane should overlap the output pin of the voltage regulator circuit to provide a low-impedance current path.

The ground plane should never be split because it provides a low-impedance current sink and reference. Use generous decoupling to ensure that clean power is supplied to the processor.

6.7 Thermal Design

In virtually all system designs, the processor's case temperature must be kept cool with some type of heatsink device. Typically, the heatsink is combined with an airflow device, such as a fan. In general, the trade-off is heat-sink size and cost versus airflow quantity and temperature. A small, low-cost heat sink requires more airflow than a larger, more efficient heat sink.

Such cooling products are widely available. For detailed specifications and assistance is selecting a product, contact your AMD field application engineer or browse the AMD home page on the World Wide Web (see Section 6.8 for details).

When gluing a heat sink to the processor case, follow these guidelines:

- Use thermal paste. This optimizes heat transfer.
- Apply the thermal paste in a thin, smooth, even layer across the entire processor package. Do not allow air gaps between the processor package and the heatsink. If air gap exits, the heatsink will be ineffective.

In addition to the above guidelines for gluing heatsinks to the processor, observe the following general design guidelines to minimize the adverse effects of system-generated heat on the processor and other heat-sensitive system components:

- Place the power supply as far away from the processor as possible.

- Place linear devices and regulators away from both the processor and the power supply.
- Place high-frequency L2-cache SRAM chips away from both the processor and the power supply.
- Check the specification for any TTL parts on the board for thermal considerations.

6.8 Design Support and Peripheral Products

AMD field application engineers (FAEs) can help you solve system design problems and select peripheral products that are compatible with the AMD5$_K$86 processor. You can locate the FAE nearest you by contacting one of the AMD offices listed in this manual. You can also find support information on AMD's World Wide Web pages. A list of available Web information is given at the AMD home page at the following address:

`http://www.amd.com/`

7

Test and Debug

The AMD5$_K$86 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 AMD5$_K$86 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 using 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 AMD5$_K$86 processor's Test Access Port (TAP) instruction register with the USEHDT instruction.

The test-related signals and their descriptions include the following:

- $\overline{\text{FLUSH}}$—Page 5-67
- $\overline{\text{FRCMC}}$—Page 5-70
- $\overline{\text{IERR}}$—Page 5-80
- INIT—Page 5-82
- PRDY—Page 5-104
- R/$\overline{S}$—Page 5-108
- RESET—Page 5-110
- TCK—Page 5-128
- TDI—Page 5-129
- TDO—Page 5-130
- TMS—Page 5-131
- $\overline{\text{TRST}}$—Page 5-132

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

7.1 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 in Section 3.3.5 on page 3-35. Figure 7-1 and Table 7-1 show the format and fields of the HWCR.

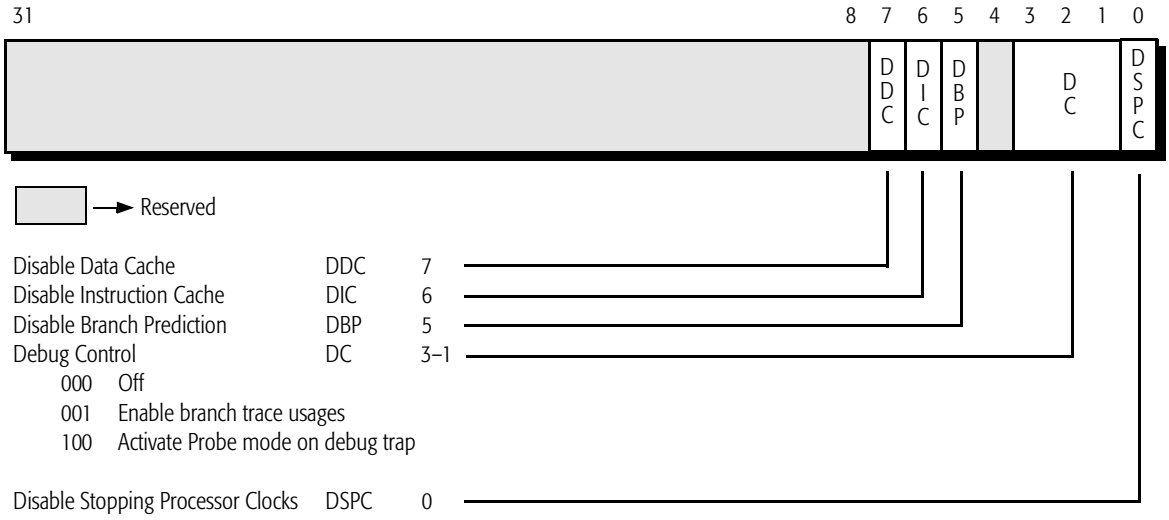


FIGURE 7-1. Hardware Configuration Register (HWCR)

TABLE 7-1. 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 Section 7.6 on page 7-17. 010 reserved 011 reserved 100 reserved 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.

7.2 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 AMD5$_K$86 processor does not report parity errors on $\overline{\text{IERR}}$ for every cache or TLB access. Instead, the AMD5$_K$86 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.

7.2.1 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 7-2. The processor continues with its normal boot process after the BIST completes, whether the BIST passed or failed.

TABLE 7-2. 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

7.2.2 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 Section 7.8 on page 7-19. 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 $\overline{\text{TRST}}$ signal must be asserted or the TAP instruction must be changed in order to exit TAP BIST and return to normal operation.

7.3 Output-Float Test

The Output-Float Test mode is entered if $\overline{\text{FLUSH}}$ 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 AMD5$_K$86 and Pentium processors, $\overline{\text{FLUSH}}$ is an edge-triggered interrupt. On the 486 processor, however, the signal is a level-sensitive input.

7.4 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

7.4.1 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 in Section 3.3.5 on page 3-35. Figure 7-2 shows the format of the AAR.

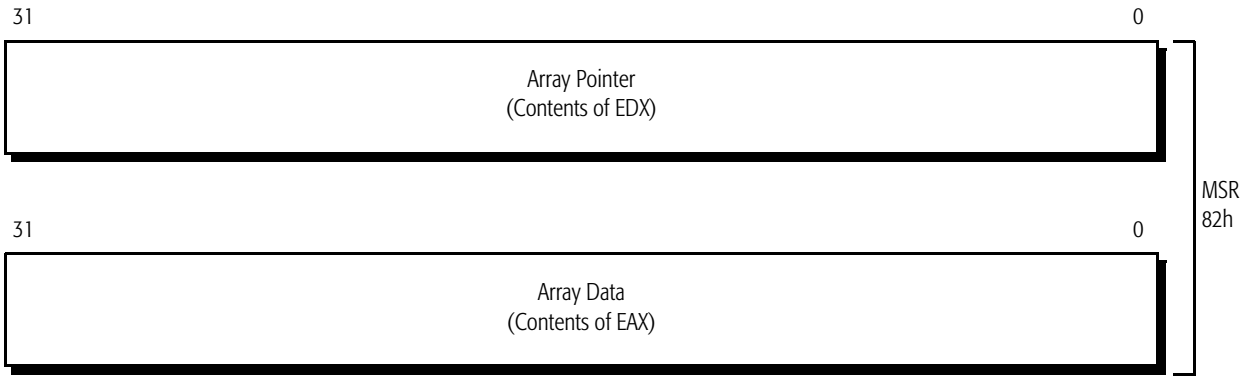


FIGURE 7-2. Array Access Register (AAR)

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 7-3 through 7-8 (top).
3. *EAX*—Read or write 32 bits of *array test data* to or from EAX, as shown in Figures 7-3 through 7-8 (bottom).

7.4.2 Array Pointer

The array pointers entered in EDX (Figures 7-3 through 7-8, 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 further 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. For the 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 7-3. To simplify multiple accesses to an array, the contents of EDX is retained after the RDMSR instruction executes (EDX is normally cleared after a RDMSR instruction).

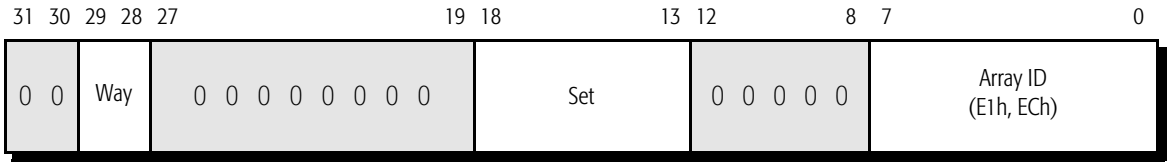
TABLE 7-3. Array IDs in Array Pointers

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

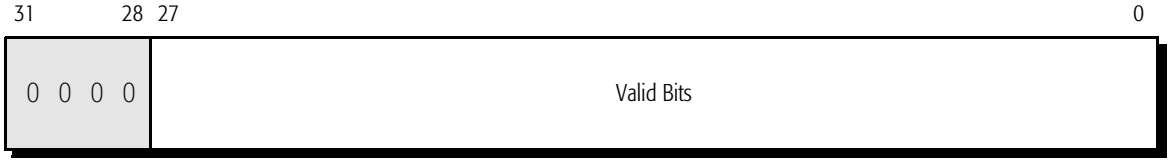
7.4.3 Array Test Data

EAX specifies the test data to be read or written with the RDMSR or WRMSR instruction (see Figures 7-3 through 7-8). For example, in Figure 7-3 (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) were accessed, the data read or written includes the tag and the status bits. The details of the valid fields in EAX are proprietary.

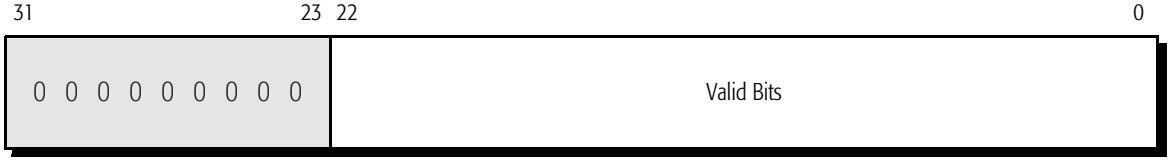
EDX: Array Pointer



EAX: Test Data



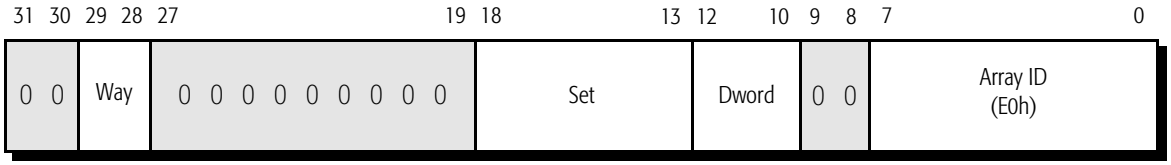
(E1h) Linear Tag



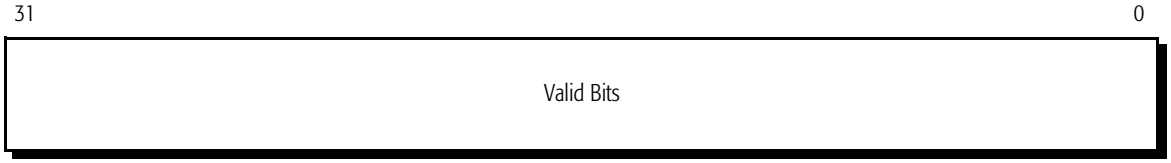
(ECh) Physical Tag

FIGURE 7-3. Test Formats: Data-Cache Tags

EDX: Array Pointer



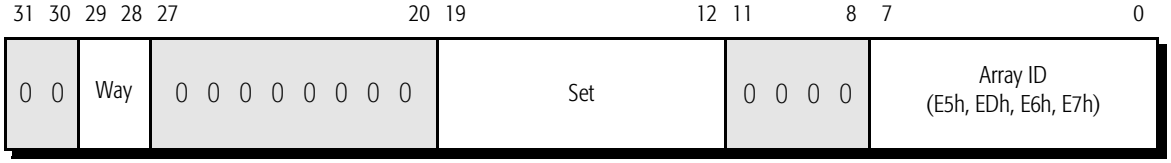
EAX: Test Data



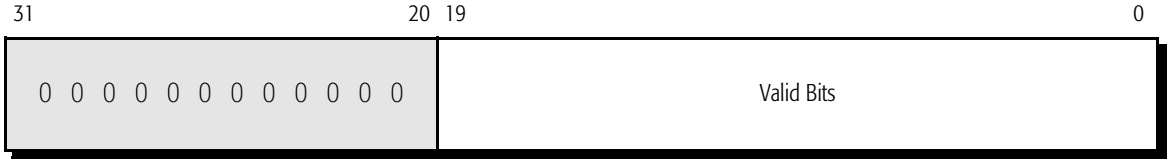
(E0h) Data

FIGURE 7-4. Test Formats: Data-Cache Data

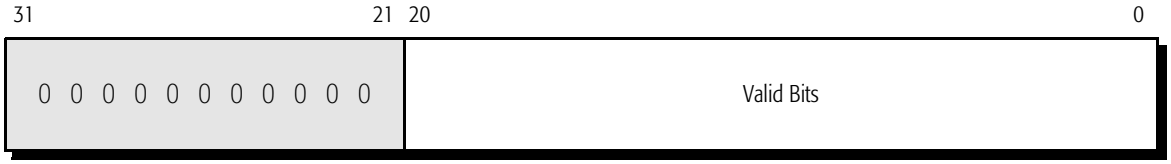
EDX: Array Pointer



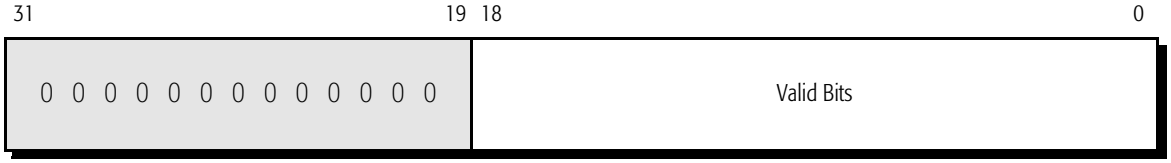
EAX: Test Data



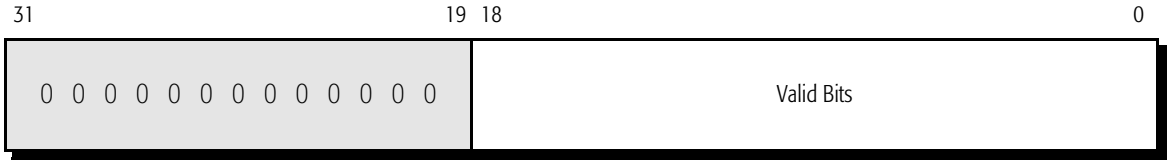
(E5h) Linear Tag



(EDh) Physical Tag



(E6h) Valid Bits

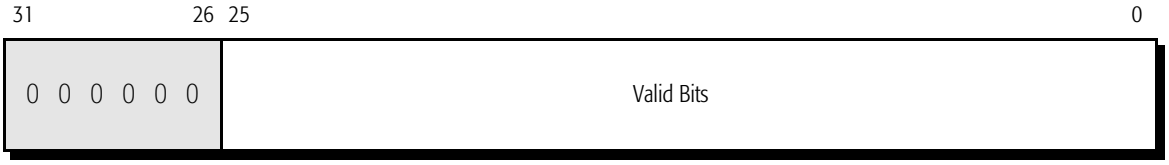


(E7h) Branch-Prediction Bits

FIGURE 7-5. Test Formats: Instruction-Cache Tags

EDX: Array Pointer

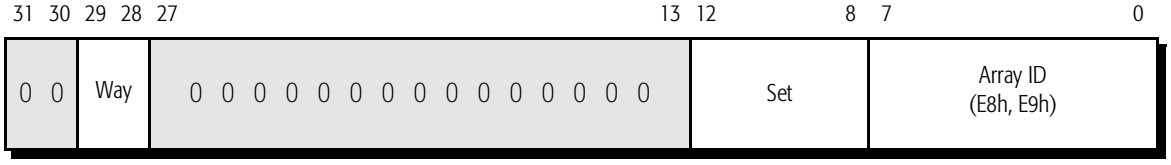
EAX: Test Data



(E4h) Instruction Bytes

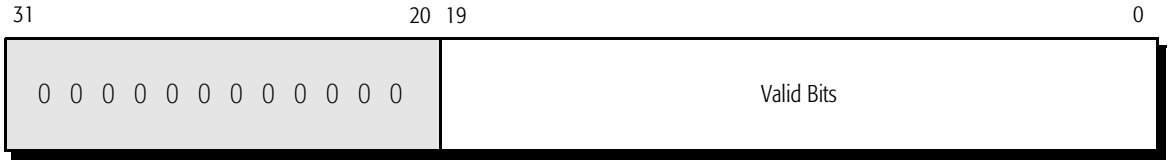
FIGURE 7-6. Test Formats: Instruction-Cache Instructions

EDX: Array Pointer



EAX: Test Data

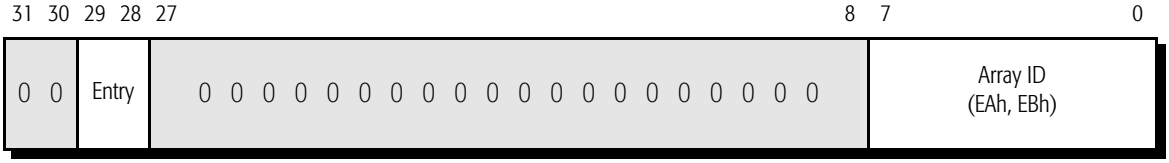
(E8h) 4-Kbyte Page and Status



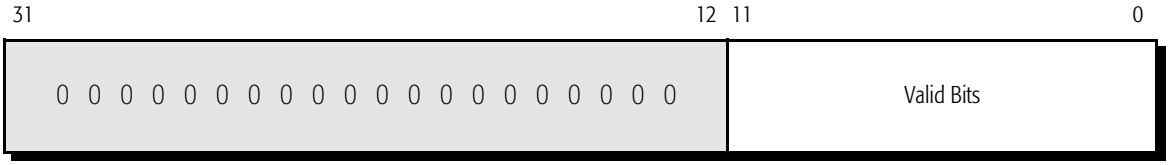
(E9h) 4-Kbyte Linear Tag

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

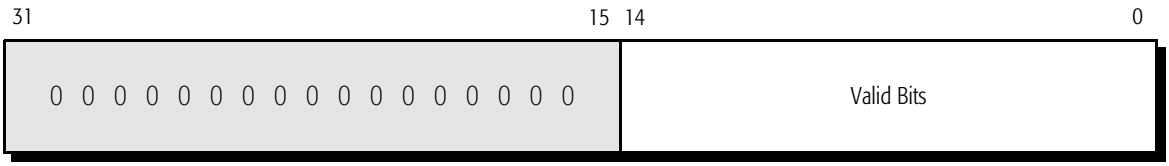
EDX: Array Pointer



EAX: Test Data



(EAh) 4-Mbyte Page and Status



(EBh) 4-Mbyte Linear Tag

FIGURE 7-8. Test Formats: 4-Mbyte TLB

7.5 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.

7.5.1 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.

7.5.2 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 Section 3.1 on page 3-2. 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 AMD5$_K$86 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.

7.5.3 Debug Compatibility with Pentium Processor

The differences in debug functions between the AMD5$_K$86 and Pentium processors are described in Section A.7 on page A-15.

7.6 Branch Tracing

Branch tracing is enabled by writing bits 3–1 with 001b and setting bit 5 to 1 in the Hardware Configuration Register (HWCR), as described in Section 7.1 on page 7-3. 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 7-4.

The branch-trace message special bus cycles are different for the AMD5$_K$86 and Pentium processors, although their $\overline{BE7}$–$\overline{BE0}$ encodings are the same.

TABLE 7-4. 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.

7.7 Functional-Redundancy Checking

If $\overline{\text{FRCMC}}$ 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 AMD5$_K$86 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.

7.8 Boundary-Scan Test Access Port (TAP)

The boundary-scan Test Access Port (TAP)—originally proposed by the Joint European Test Action Group (JETAG) and, later, Joint Test Action Group (JTAG)—is an IEEE standard that defines synchronous scanning test methods for complex logic circuits, such as boards containing a microprocessor. The AMD5$_K$86 processor supports the full TAP standard defined in the *IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE 1149.1-1990)* specification.

The TAP consists of the following:

- *Test Access Port (TAP) Controller*—A synchronous, finite state machine that decodes the inputs on the TMS signal to control a sequence of test operations.
- *Instruction Register (IR)*—Accepts serially shifted instructions from the TDI input. The instructions select the test or debug operation to be performed, the Test Data Register (TDR) to be accessed, or both.
- *Test Data Registers (TDRs)*—Used to process the test data. Each TDR is addressed by an instruction in the Instruction Register (IR). The processor includes the following TDRs:
  - *Boundary Scan Register (BSR)*—Contains cells connected to all of the processor's input and output signals as well as cells for I/O float control. It allows serial data to be written into or read from the processor boundary. The register is controlled with the EXTEST and SAMPLE instructions.
  - *Device Identification Register (DIR)*—Contains the codes for manufacturer's identification, part number, and version.
  - *Bypass Register (BR)*—A path between TDI and TDO used to transfer test data to and from other board components when no test operation is being performed by the processor.
  - *Hardware Debug Tool Register (HDTR)*—Selected by the USEHDT instruction to connect TDI and TDO, allowing HDT instructions to be executed.
  - *Built-In Self-Test Result Register (BISTRR)*—Selected by the RUNBIST instruction to connect TDI and TDO, allowing the result of executing the RUNBIST to be shifted out after the completion of BIST.
- The test signals are as follows:
  - TCK—The clock for all TAP testing
  - TDI—Input test data and instructions
  - TDO—Output data
  - TMS—Test functions and sequence of test changes
  - $\overline{\text{TRST}}$—Test reset

Boundary-scan testing uses shift registers in boundary scan cells located between the processor's internal logic and I/O

buffers to control and observe the behavior of signals at each pin. The boundary scan cells form a serial shift-register chain, called a Boundary Scan Register (BSR), around the processor's internal logic. Test data is shifted through the boundary-scan chain by a test program. If all the components on a board implement this boundary-scan architecture, a single serial path can be used to test component interconnections.

Parallel output registers are fed by the shift registers. Parallel data is loaded into the shift register when the TAP controller exits the capture state *(capture_DR* or *capture_IR)*. The shift registers then shift data from TDI to TDO in the shift state *(shift_DR* or *shift_IR)*. The parallel output registers hold the current data while new data is shifted into the shift registers. The output registers are updated when the controller exits the update state *(update_DR* or *update_IR)*.

The sections below describe only those aspects of the IEEE standard that are implemented uniquely by the AMD5$_K$86 processor. For a description of the IEEE-mandatory TAP functions and the IEEE optional functions implemented by the AMD5$_K$86 processor, see the *IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE 1149.1-1990)* specification.

7.8.1 Device Identification Register

The format of the Device Identification Register (DIR) is shown in Table 7-5. 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 7-5. Test Access Port (TAP) ID Code

Version (Bits 31–28)	Bond Option (Bits 27–26)	Part Number (Bits 25–12)	Manufacturer (Bits 11–0)
Xh	XXb	05XXh	001h

7.8.2 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 AMD5$_K$86 processor (ALL1, ALL0, USEHDT). Table 7-6 shows the complete set of public TAP instructions supported by the processor. In addition, the processor 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 below. Any instruction encodings not shown in Table 7-6 select the BYPASS instruction.

TABLE 7-6. 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 Section 7.9 on page 7-23
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.*

7.9 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 AMD sales representative or field application engineer.

Appendix A

Compatibility With the Pentium and 486 Processors

The AMD5$_K$86 processor is compatible with the existing Pentium-class hardware and software infrastructure, including chipsets, motherboards, operating systems, and applications software. In particular, the following AMD5$_K$86 processor features are compatible with the Pentium processor:

- Package and pinout
- Electrical interface (including bus cycles, AC and DC parameters, interrupt handling, power saving, etc.)
- Instruction set, programming model, memory management, etc.

Because the AMD5$_K$86 processor takes a different approach to implementing the x86 architecture, there are a few subtle differences between the Pentium and AMD5$_K$86 processors. This appendix describes these differences.

A.1 Bus Signals

A.1.1 Signal Comparison

Table A-1 compares the signals on the Pentium processor with those on the AMD5$_K$86 processor, showing which signals are supported on each processor.

TABLE A-1. AMD5$_K$86 and Pentium Processor Signal Comparison

Signal	Pentium (735\90, 815\100)	AMD5$_K$86	Function
$\overline{\text{A20M}}$	x	x	Address Bit 20 Mask
A31–A3	x	x	Address Bus
$\overline{\text{ADS}}$	x	x	Address Strobe
$\overline{\text{ADSC}}$	x	x	Address Strobe
AHOLD	x	x	Address Hold
AP	x	x	Address Parity
$\overline{\text{APCHK}}$	x	x	Address Parity Check
APICEN	x		APIC Enable (High during RESET)
PICD1	x		PIC Data 1
$\overline{\text{BE7}}$–$\overline{\text{BE0}}$	x	x	Byte Enables
Flush(4)	x		Dual-Processor Flush
APICID3–APICID0	x		APIC ID (during reset)
BF	x	x	Bus-to-Core Frequency Ratio
$\overline{\text{BOFF}}$	x	x	Bus Backoff
BP3–BP2	x		Breakpoint 3 to 2
BP1–BP0/ PM1–PM0	x		Breakpoint 1 to 0 or Performance Monitor 1 to 0
$\overline{\text{BRDY}}$	x	x	Burst Ready
$\overline{\text{BRDYC}}$	x		Drive-Strength Control (during RESET)
	x	x	Burst Ready
BREQ	x	x	Bus Request
$\overline{\text{BUSCHK}}$	x	x	Drive-Strength Control (during RESET)
	x	x	Bus Check
$\overline{\text{CACHE}}$	x	x	Cacheable Cycle

TABLE A-1. AMD5$_K$86 and Pentium Processor Signal Comparison (continued)

Signal	Pentium (735\90, 815\100)	AMD5$_K$86	Function
CLK	x	x	System Clock (5 V-tolerant)
CPUTYP	x		Primary or Secondary Processor
D/$\overline{C}$	x	x	Data or Code Cycle
D63–D0	x	x	Data Bus
D/$\overline{P}$	x		Dual or Primary Processor Cycle
DP7–DP0	x	x	Data Parity
$\overline{DPEN}$	x		Dual Processor Present (during RESET)
PCID0	x		PIC Data 0
$\overline{EADS}$	x	x	External Address Strobe
$\overline{EWBE}$	x	x	External Write Buffer Empty
$\overline{FERR}$	x	x	Floating-Point Error
FLUSH	x	x	Float-Test Mode (during RESET)
	x	x	Writeback and Invalidate Caches
$\overline{FRCMC}$	x	x	Functional Redundancy Checking Master/Checker
$\overline{HIT}$	x	x	Inquire Hit
$\overline{HITM}$	x	x	Inquire Hit to Modified Line
HLDA	x	x	Hold Acknowledge
HOLD	x	x	Hold
$\overline{IERR}$	x	x	Internal Error
$\overline{IGNNE}$	x	x	Ignore Numeric Error
INIT	x	x	Execute BIST (during RESET)
	x	x	Initialize (warm start)
INV	x	x	Invalid or Shared After Inquire Cycle
$\overline{KEN}$	x	x	Cache Enable
LINT0/INTR	x		Local Interrupt 0 (APIC enabled)
	x	x	Maskable Interrupt
LINT1/NMI	x		Local Interrupt 1 (APIC enabled)
	x	x	Non-Maskable Interrupt
$\overline{LOCK}$	x	x	Locked Cycle
M/$\overline{IO}$	x	x	Memory or I/O Cycle
$\overline{NA}$	x	x	Next (pipelined) Address
$\overline{PBGNT}$	x		Private Bus Grant

TABLE A-1. AMD5$_K$86 and Pentium Processor Signal Comparison (continued)

Signal	Pentium (735\90, 815\100)	AMD5$_K$86	Function
$\overline{\text{PBREQ}}$	x		Private Bus Request
PCD	x	x	Page Cache Disable
$\overline{\text{PCHK}}$	x	x	Parity Check
$\overline{\text{PEN}}$	x	x	Parity Enable
$\overline{\text{PHIT}}$	x		Private Hit
$\overline{\text{PHITM}}$	x		Private Hit to Modified Line
PICCLK	x		PIC clock, 5 V-Tolerant
PRDY	x	x	Probe Ready
PWT	x	x	Page Writethrough
R/$\overline{\text{S}}$	x	x	Run or Stop
RESET	x	x	Reset
SCYC	x	x	Misaligned Transfer
$\overline{\text{SMI}}$	x	x	System Management Interrupt
$\overline{\text{SMIACT}}$	x	x	System Management Interrupt Active
$\overline{\text{STPCLK}}$	x	x	Stop Clock
TCK	x	x	Test Access Port (TAP) Clock
TDI	x	x	Test Access Port (TAP) Data In
TDO	x	x	Test Access Port (TAP) Data Out
TMS	x	x	Test Access Port (TAP) Test Mode Select
$\overline{\text{TRST}}$	x	x	Test Access Port (TAP) Reset
W/$\overline{\text{R}}$	x	x	Write or Read Cycle
WB/$\overline{\text{WT}}$	x	x	Writeback or Writethrough

A.2 Bus Interface

A.2.1 Updates to Descriptor Accessed and TSS Busy Bits

For updates to the Accessed bit in the data and code segment descriptors, the behavior of the AMD5$_K$86 processor is different than the Pentium processor. In the aligned case, the AMD5$_K$86 processor performs two 4-byte unlocked reads to read in the descriptor. If the Accessed bit needs to be set, a 4-byte locked read and a 4-byte locked write will follow. The Pentium processor performs an 8-byte unlocked read to get the descriptor. If the Accessed bit needs to be set, an 8-byte locked read and a 1-byte locked write will follow.

For the misaligned case, the AMD5$_K$86 processor performs four unlocked reads to get the descriptor. If the Accessed bit needs to be set, two locked reads and two locked writes will follow. The Pentium processor performs two unlocked reads to get the descriptor. If the Accessed bit needs to be set, two locked reads will be followed by one 1-byte locked write.

For updates to the Busy bit in the TSS descriptor, the AMD5$_K$86 processor behaves in the manner described for updates to the Accessed bit. The Pentium processor does not perform the unlocked read to get the descriptor.

A.2.2 Locked and Unlocked CMPXCHG8B Operation

On a locked and misaligned—not on a dword boundary—CMPXCHG8B operation, the AMD5$_K$86 processor performs two split reads followed by two split writes, all under lock, for a total of eight cycles. The Pentium processor combines the split reads and split writes, for a total of four cycles.

On a locked and aligned CMPXCHG8B operation, the AMD5$_K$86 processor performs two reads followed by two writes, for a total of four cycles. The Pentium processor combines one read and one write, for a total of two cycles.

On an unlocked and non-cacheable CMPXCHG8B operation, the misaligned and aligned CMPXCHG8B operations are the

same as the locked misaligned and locked aligned CMPXCHG8B operations, respectively, described above.

On an unlocked and cacheable CMPXCHG8B operation, the AMD5$_K$86 and Pentium processors behave the same.

A.2.3 Bus Cycle Order of Misaligned Memory and I/O Cycles

The AMD5$_K$86 processor performs split (misaligned) memory read, memory write, and I/O read cycles in the reverse order of the Pentium processor. Split I/O write cycles occur in the same order on both processors.

A.2.4 Halt Cycle after FLUSH

When halted, the AMD5$_K$86 processor reruns a Halt special cycle after the Flush Acknowledge special cycle following a cache flush operation. The Pentium processor does not rerun a Halt special cycle.

A.2.5 Selectable Drive Strengths on Output Driver

The AMD5$_K$86 processor supports selectable drive strengths on the following output pins:

- A20–A3
- W/$\overline{R}$
- $\overline{ADS}$
- $\overline{HITM}$

This is the same set of output pins that have selectable drive strengths on the Pentium processor. However, the Pentium processor supports three drive strengths on these pins while the AMD5$_K$86 processor supports two.

The selection of the drive strengths differs between the Pentium processor and the AMD5$_K$86 processor as follows:

Drive Strength	BRDYC	BUSCHK
Pentium		
Strength 1 (weakest)	1	X
Strength 2 (medium)	0	1
Strength 3 (strongest)	0	0
AMD5$_K$86		
Strength 1 (weak)	1	X
Strength 1 (weak)	0	1
Strength 2 (strong)	0	0

Comments

The exact drive characteristics of the two strengths differ from the Intel parts. Those differences are not documented in this functional description. See the AMD5$_K$86 processor data sheet for more information.

A.3 Bus Mastering Operations (including Snooping)

A.3.1 AHOLD Snoop to Linefill Buffer Prior to or Coincident with the Establishment of the Cacheability of the Line

An AHOLD snoop to the linefill buffer occurs during a linefill when the address of the snoop matches the address of the linefill. If the snoop happens before or coincident with the establishment of the cacheability of the line via the $\overline{\text{KEN}}$ pin sampled with the assertion of $\overline{\text{NA}}$ or $\overline{\text{BRDY}}$ (whichever comes first), the AMD5$_K$86 processor treats the snoop as a hit, whereas the Pentium processor treats it as a miss.

Comments

In treating the snoop as a hit, the AMD5$_K$86 processor asserts the $\overline{\text{HIT}}$ pin and also caches the line as either shared or invalid, depending on the state of the INV pin. If $\overline{\text{KEN}}$ is sampled inactive, the line is not cached, regardless of the state of the INV pin.

In treating the snoop as a miss, the Pentium processor deasserts the $\overline{\text{HIT}}$ pin and caches the line based on $\overline{\text{KEN}}$, WB/$\overline{\text{WT}}$, and PWT in the same way it does for linefills with no snoop.

The behavior of snoops to the linefill buffer after cacheability is determined is described in Section A.3.2.

A.3.2 $\overline{\text{BOFF}}$ Asserted before Snoop to Linefill Buffer and after the Cacheability of the Line is Established

A snoop to the linefill buffer occurs during a linefill when the address of the snoop matches the address of the linefill. If $\overline{\text{BOFF}}$ is asserted after the cacheability of the line is determined via the $\overline{\text{KEN}}$ pin being sampled active (with the assertion of $\overline{\text{NA}}$ or $\overline{\text{BRDY}}$, whichever comes first) and a snoop to the linefill buffer occurs with either $\overline{\text{BOFF}}$ or AHOLD or both asserted, the Pentium processor treats the snoop as a hit, whereas the AMD5$_K$86 processor may or may not treat it as a hit. For DCACHE linefills, the AMD5$_K$86 processor treats the snoop as a miss. For ICACHE linefills, the AMD5$_K$86 processor may treat the snoop as a hit or a miss, because the speculative nature of the linefills makes their cacheability dependent on

the code sequence and, therefore, unpredictable from an external system point of view.

Comments

In treating the snoop as a hit, the AMD5$_K$86 and Pentium processors assert the $\overline{\text{HIT}}$ pin and also cache the line as either shared or invalid, depending on the state of the INV pin. The cycle restarts after the deassertion of $\overline{\text{BOFF}}$ and AHOLD.

In treating the snoop as a miss, the AMD5$_K$86 processor deasserts the $\overline{\text{HIT}}$ pin. The state of the line is determined based on $\overline{\text{KEN}}$, WB/$\overline{\text{WT}}$, and PWT when the cycle is restarted after the deassertion of $\overline{\text{BOFF}}$ and AHOLD.

The behavior of snoops to the linefill buffer before cacheability is determined is described in Section A.3.1.

A.3.3 Snoop Before Write Hit to ICACHE Appears on Bus

If a write to a valid ICACHE line occurs and a snoop occurs to the same line before the write appears on the bus, the Pentium processor generates a snoop hit until the write is on the bus. The AMD5$_K$86 processor generates a snoop miss in the window between when the cache is invalidated and the write appears on the bus. The ICACHE line is invalidated in both processors by the time the write appears on the bus.

A.3.4 Invalidations during a $\overline{\text{FLUSH}}$/WBINVD

During a $\overline{\text{FLUSH}}$/WBINVD between a line copyback and the Flush Acknowledge cycle, a subsequent snoop to that line reports a snoop hit modified and generates another copyback. The Pentium processor invalidates lines as they are accessed during $\overline{\text{FLUSH}}$. The AMD5$_K$86 processor invalidates all lines at the end of a $\overline{\text{FLUSH}}$.

Once $\overline{\text{FLUSH}}$/WBINVD has completed, the entire cache is invalid for both the AMD5$_K$86 and Pentium processors.

A.3.5 Cache Line Ownership

When the processor generates a read hit to a line in its own ICACHE, the Pentium processor invalidates the ICACHE and initiates a DCACHE linefill. However, the AMD5$_K$86 processor

keeps the ICACHE valid and a non-cacheable, external read is performed to supply the data.

A.3.6 Write Hit to a Shared Line in the DCACHE

When a write hits a shared line in the DCACHE, the write is passed through to the external bus. The state of the WB/$\overline{\text{WT}}$ pin is sampled with the $\overline{\text{BRDY}}$ (or $\overline{\text{NA}}$) of the write, and if it is High, the line changes state from shared to exclusive. Subsequent writes to the same line change the state of the line from exclusive to modified and do not go external. Both the AMD5$_K$86 and Pentium processors behave in this manner.

However, if two or more writes to different locations within the same cache line are queued up in the store buffer, the line is shared and the WB/$\overline{\text{WT}}$ pin is set High, then the AMD5$_K$86 processor correctly allows the first write to reach the bus and the line transitions to exclusive. The remainder of the writes to that line do not show up on the external bus. In the Pentium processor, the first two or more writes go external. The remainder hit the line in the exclusive state and do not go external.

A.4 Memory Management

A.4.1 Speculative TLB Refills

The Pentium processor performs speculative TLB refills (including setting the accessed bit) for code prefetches. This may result in the accessed bit being set for a page that is not actually used. The AMD5$_K$86 processor does not perform speculative TLB refills.

A.4.2 Page Fault Encountered by a Load/Store Type of Instruction

On a read page fault encountered by a load/op/store type of instruction, the error code reported by a 486 processor indicates a read operation, whereas the Pentium processor indicates a write operation. The AMD5$_K$86 processor reports the same error code as the 486 processor.

A.5 Power Saving Features

A.5.1 $\overline{\text{STPCLK}}$ in Halt State

When in the Halt state, the AMD5$_K$86 processor responds to $\overline{\text{STPCLK}}$ and enters the Stop Grant state. The Pentium processor ignores $\overline{\text{STPCLK}}$ in the Halt state.

A.5.2 $\overline{\text{STPCLK}}$ Pulse does not Guarantee That One Instruction Executes

Unlike the Pentium processor, the AMD5$_K$86 processor does not guarantee that at least one instruction will be executed between the deassertion of $\overline{\text{STPCLK}}$ and a subsequent reassertion of $\overline{\text{STPCLK}}$. On the Pentium processor, at least one instruction is guaranteed to execute.

A.5.3 Simultaneous I/O SMI Trap and Debug Breakpoint Trap

On a simultaneous I/O SMI trap and debug breakpoint trap, the AMD5$_K$86 processor responds to the SMI first and postpones writing the fault frame for the debug trap to the stack until after the resumption of normal execution via RSM. (If debug registers DR3–DR0 are going to be used while in SMM, they must be saved and restored by the SMM software. DR6 and DR7 are automatically saved and restored.) This is similar to the Pentium processor behavior (P54C only) with TR12.ITR set to 1, although the postponing of the debug trap is only accomplished with trapped I/O instructions, where the timing of the SMI met the requirements for SMI I/O trapping.

On the AMD5$_K$86 processor, if, on the RSM, the I/O Restart Flag in the SMM save area is set, the debug trap is cancelled and will be redetected as a result of the reexecution of the I/O instruction.

A.5.4 SMM Save Area

The contents of any reserved locations are not necessarily the same between the AMD5$_K$86, Pentium, and the 486 processors.

A.5.5 NMI Recognition during SMM

When operating in SMM, an NMI request should not be recognized unless an enabled INTR is encountered. Both the AMD5$_K$86 and Pentium processors do this correctly, but in slightly different ways. The Pentium processor takes the NMI request immediately after recognizing the INTR, but before executing any instructions from the interrupt handler. The AMD5$_K$86 processor takes the NMI request upon encountering the IRET in the interrupt handler. (In fact, the AMD5$_K$86 processor unmasks NMI when any IRET is encountered, not just one associated with INTR.)

Comment With both processors, the Intel recommendation of using a fake INTR to unmask NMI while in SMM works correctly.

A.6 Exceptions

A.6.1 Limit Faults on an Invalid Instruction

When executing an instruction that crosses a limit boundary and the instruction is interpreted as invalid, the AMD5$_K$86 processor prioritizes the invalid opcode fault. The Pentium and 486 processors prioritize the limit violation fault.

A.6.2 Task Switch

On a task switch, the AMD5$_K$86 processor sets the busy bit of the incoming task after storing the outgoing TSS according to 486 and Pentium processor documentation. The Pentium processor sets the busy bit before trying to store the outgoing TSS. If a fault occurs while trying to store the TSS, the Pentium processor clears the busy bit. The end result of the instruction is the same on both processors.

A.7 Debug

A.7.1 Proprietary Branch Trace Messages

Branch trace messages are different. The AMD5$_K$86 processor uses the same $\overline{BE}$ pattern for the special bus cycles as the Pentium processor, but the format of decoding information is different.

A.7.2 Multiple Debug Breakpoint Matches

Multiple debug breakpoint matches do not set multiple B bits in DR6 on the AMD5$_K$86 processor.

A.7.3 Simultaneous Debug Trap and Debug Fault

If a debug trap associated with the completion of an instruction (single-step trap or load/store breakpoint) occurs at the same time as a debug fault (instruction breakpoint) on the next instruction, the Pentium processor merges the two conditions into a single call to the debug handler, setting both B bits in the debug status register. The AMD5$_K$86 processor processes the two conditions serially, setting the appropriate B bits for each invocation of the handler.

Index

Numerics

A

B

C

D

E

F

G

H

I

R

S

T

U

V

W