July 15, 1997 3:24, Updated 5/13/98

Addendums and other updates for this manual can be obtained from Cyrix Web site: www.cyrix.com.

Trademark Acknowledgments:

Cyrix is a registered trademark of Cyrix Corporation.
6x86 and 6x86MX are trademarks of Cyrix Corporation. MMX is a trademark of Intel Corporation.
All other brand or product names are trademarks of their respective companies.

Order Number: 94329-00
Cyrix Corporation
2703 North Central Expressway
Richardson, Texas 75080-2010
United States of America

TABLE OF CONTENTS

LIST OF FIGURES

Figure Name **Page Number**

LIST OF FIGURES (Continued)

Figure Name **Page Number**

LIST OF FIGURES (Continued)

LIST OF TABLES

Table Name **Page Number**

LIST OF TABLES (Continued)

Table Name **Page Number**

LIST OF TABLES (Continued)

Table Name **Page Number**

[redacted]

1. ARCHITECTURE OVERVIEW

The Cyrix 6x86MX™ processor is an enhanced 6x86 processor, that can process 57 new multimedia instructions compatible with MMX™ technology. The processor also operates at a higher frequency, contains a enlarged cache, a two-level TLB, and a improved branch target cache.

The 6x86MX processor is based on the proven 6x86 core that is superscalar in that it contains two separate pipelines that allow multiple instructions to be processed at the same time. The use of advanced processing technology and superpipelining (increased number of pipeline stages) allow the 6x86MX CPU to achieve high clocks rates.

Through the use of unique architectural features, the 6x86MX processor eliminates many data dependencies and resource conflicts, resulting in optimal performance for both 16-bit and 32-bit x86 software.

For maximum performance, the 6x86MX CPU contains two caches, a large unified 64 KByte 4-way set associative write-back cache and a small high-speed instruction line cache.

To provide support for multimedia operations, the cache can be turned into a scratchpad RAM memory on a line by line basis. The cache area set aside as scratchpad memory acts as a private memory for the CPU and does not participate in cache operations.

Within the 6x86MX processor there are two TLBs, the main L1 TLB and the larger L2 TLB. The direct-mapped L1 TLB has 16 entries and the 6-way associative L2 TLB has 384 entries.

The on-chip FPU has been enhanced to process MMX™ instructions as well as the floating point instructions. Both types of instructions execute in parallel with integer instruction processing. To facilitate FPU operations, the FPU features a 64-bit data interface, a four-deep instruction queue and a six-deep store queue.

The CPU operates using a split rail power design. The core runs on a 2.9 volt power supply, to minimize power consumption. External signal level compatibility is maintained by using a 3.3 volt power supply for the I/O interface.

For mobile systems and other power sensitive applications, the 6x86MX processor incorporates low power suspend mode, stop clock capability, and system management mode (SMM).

1.1 Major Differences Between the 6x86MX and the 6x86 Processors

The major differences between the 6x86MX and the 6x86 processors are summarized in Table 1-1.

Table 1-1. The 6x86MX Processor Versus the 6x86 Processor

FEATURE	6x86MX Processor	6x86 Processor	
Pinout	P55C	P54C	
Supply Voltage Core I/O	 2.9 V 3.3 V	6x86: 3.3 or 3.52 V 3.3 V	6x86L: 2.8 V 3.3 V
CPU Primary Cache	64 KBytes	16 KBytes	
Translation Lookaside Buffer (TLB)	L1: 16 entry L2: 384 entry	L1: 128 entry Victim TLB: 8 entry	
Branch Prediction	512 entry branch target cache 1024 entry branch history table	256 entry branch target cache 512 entry branch history table	
MMX	Yes	No	
Performance Monitor including Time Stamp Counter and Model Specific Registers	Yes	No	
Scratchpad RAM in Primary Cache	Yes	No	
Cacheable SMI Code/Data	Yes	No	
Clock Modes	2x, 2.5x, 3x, 3.5x	2x, 3x	

1.2 Major Functional Blocks

The 6x86MX processor consists of four major functional blocks, as shown in the overall block diagram on the first page of this manual:

- Memory Management Unit
- CPU Core
- Cache Unit
- Bus Interface Unit

The CPU contains the superpipelined integer unit, the BTB (Branch Target Buffer) unit and the FPU (Floating Point Unit).

The BIU (Bus Interface Unit) provides the interface between the external system board and the processor's internal execution units. During a memory cycle, a memory location is selected through the address lines (A31-A3 and BE7# -BE0#). Data is passed from or to memory through the data lines (D63-D0).

Each instruction is read into 256-Byte Instruction Line Cache. The Cache Unit stores the most recently used data and instructions to allow fast access to the information by the Integer Unit and FPU.

The CPU core requests instructions from the Cache Unit. The received integer instructions are decoded by either the X or Y processing pipelines within the superpipelined integer unit. If the instruction is a MMX or FPU instruction it is passed to the floating point unit for processing.

As required data is fetched from the 64-KByte unified cache. If the data is not in the cache it is accessed via the bus interface unit from main memory.

The Memory Management Unit calculates physical addresses including addresses based on paging.

Physical addresses are calculated by the Memory Management Unit and passed to the Cache Unit and the Bus Interface Unit (BIU).

1.3 Integer Unit

The Integer Unit (Figure 1-1) provides parallel instruction execution using two seven-stage integer pipelines. Each of the two pipelines, X and Y, can process several instructions simultaneously.

The Integer Unit consists of the following pipeline stages:

- Instruction Fetch (IF)
- Instruction Decode 1 (ID1)
- Instruction Decode 2 (ID2)
- Address Calculation 1 (AC1)
- Address Calculation 2 (AC2)
- Execute (EX)
- Write-Back (WB)

The instruction decode and address calculation functions are both divided into superpipelined stages.

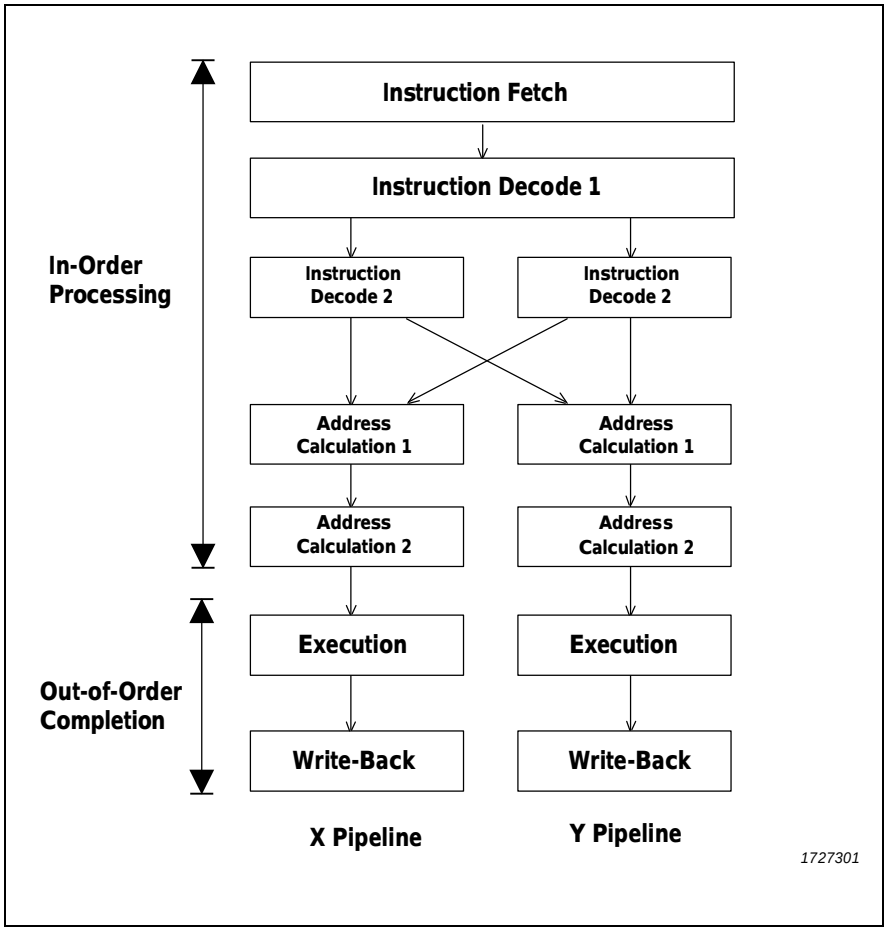


Figure 1-1. Integer Unit

1.3.1 Pipeline Stages

The **Instruction Fetch** (IF) stage, shared by both the X and Y pipelines, fetches 16 bytes of code from the cache unit in a single clock cycle. Within this section, the code stream is checked for any branch instructions that could affect normal program sequencing.

If an unconditional or conditional branch is detected, branch prediction logic within the IF stage generates a predicted target address for the instruction. The IF stage then begins fetching instructions at the predicted address.

The superpipelined **Instruction Decode** function contains the ID1 and ID2 stages. ID1, shared by both pipelines, evaluates the code stream provided by the IF stage and determines the number of bytes in each instruction. Up to two instructions per clock are delivered to the ID2 stages, one in each pipeline.

The ID2 stages decode instructions and send the decoded instructions to either the X or Y pipeline for execution. The particular pipeline is chosen, based on which instructions are already in each pipeline and how fast they are expected to flow through the remaining pipeline stages.

The **Address Calculation** function contains two stages, AC1 and AC2. If the instruction refers to a memory operand, the AC1 calculates a linear memory address for the instruction.

The AC2 stage performs any required memory management functions, cache accesses, and register file accesses. If a floating point instruction is detected by AC2, the instruction is sent to the FPU for processing.

The **Execute** (EX) stage executes instructions using the operands provided by the address calculation stage.

The **Write-Back** (WB) stage is the last IU stage. The WB stage stores execution results either to a register file within the IU or to a write buffer in the cache control unit.

1.3.2 Out-of-Order Processing

If an instruction executes faster than the previous instruction in the other pipeline, the instructions may complete out of order. All instructions are processed in order, up to the EX stage. While in the EX and WB stages, instructions may be completed out of order.

If there is a data dependency between two instructions, the necessary hardware interlocks are enforced to ensure correct program execution. Even though instructions may complete out of order, exceptions and writes resulting from the instructions are always issued in program order.

1.3.3 Pipeline Selection

In most cases, instructions are processed in either pipeline and without pairing constraints on the instructions. However, certain instructions are processed only in the X pipeline:

- Branch instructions
- Floating point instructions
- Exclusive instructions

Branch and floating point instructions may be paired with a second instruction in the Y pipeline.

Exclusive Instructions cannot be paired with instructions in the Y pipeline. These instructions typically require multiple memory accesses. Although exclusive instructions may not be paired, hardware from both pipelines is used to accelerate instruction completion. Listed below are the 6x86MX CPU exclusive instruction types:

- Protected mode segment loads
- Special register accesses (Control, Debug, and Test Registers)
- String instructions
- Multiply and divide
- I/O port accesses
- Push all (PUSHA) and pop all (POPA)
- Intersegment jumps, calls, and returns

1.3.4 Data Dependency Solutions

When two instructions that are executing in parallel require access to the same data or register, one of the following types of data dependencies may occur:

- Read-After-Write (RAW)
- Write-After-Read (WAR)
- Write-After-Write (WAW)

Data dependencies typically force serialized execution of instructions. However, the 6x86MX CPU implements three mechanisms that allow parallel execution of instructions containing data dependencies:

- Register Renaming
- Data Forwarding
- Data Bypassing

The following sections provide detailed examples of these mechanisms.

1.3.4.1 Register Renaming

The 6x86MX CPU contains 32 physical general purpose registers. Each of the 32 registers in the register file can be temporarily assigned as one of the general purpose registers defined by the x86 architecture (EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP). For each register write operation a new physical register is selected to allow previous data to be retained temporarily. Register renaming effectively removes all WAW and WAR dependencies. The programmer does not have to consider register renaming as register renaming is completely transparent to both the operating system and application software.

Example #1 - Register Renaming Eliminates Write-After-Read (WAR) Dependency

A WAR dependency exists when the first in a pair of instructions reads a logical register, and the second instruction writes to the same logical register. This type of dependency is illustrated by the pair of instructions shown below:

X PIPE	**Y PIPE**
(1) MOV BX, AX	(2) ADD AX, CX
BX ← AX	AX ← AX + CX

Note: In this and the following examples the original instruction order is shown in parentheses.

In the absence of register renaming, the ADD instruction in the Y pipe would have to be stalled to allow the MOV instruction in the X pipe to read the AX register.

The 6x86MX CPU, however, avoids the Y pipe stall (Table 1-2). As each instruction executes, the results are placed in new physical registers to avoid the possibility of overwriting a logical register value and to allow the two instructions to complete in parallel (or out of order) rather than in sequence.

Table 1-2. Register Renaming with WAR Dependency

Instruction	Physical Register Contents					Action	
	Reg0	Reg1	Reg2	Reg3	Reg4	Pipe	
(Initial)	AX	BX	CX				
MOV BX, AX	AX		CX	BX		X	Reg3 ← Reg0
ADD AX, CX			CX	BX	AX	Y	Reg4 ← Reg0 + Reg2

Note: The representation of the MOV and ADD instructions in the final column of Table 1-2 are completely independent.

Example #2 - Register Renaming Eliminates Write-After-Write (WAW) Dependency

A WAW dependency occurs when two consecutive instructions perform writes to the same logical register. This type of dependency is illustrated by the pair of instructions shown below:

X PIPE	Y PIPE
(1) ADD AX, BX	(2) MOV AX, [mem]
AX ←AX + BX	AX ← [mem]

Without register renaming, the MOV instruction in the Y pipe would have to be stalled to guarantee that the ADD instruction in the X pipe would write its results to the AX register first.

The 6x86MX CPU uses register renaming and avoids the Y pipe stall. The contents of the AX and BX registers are placed in physical registers (Table 1-3). As each instruction executes, the results are placed in new physical registers to avoid the possibility of overwriting a logical register value and to allow the two instructions to complete in parallel (or out of order) rather than in sequence.

Table 1-3. Register Renaming with WAW Dependency

Instruction	Physical Register Contents				Action	
	Reg0	Reg1	Reg2	Reg3	Pipe	
(Initial)	AX	BX				
ADD AX, BX		BX	AX		X	Reg2 ← Reg0 + Reg1
MOV AX, [mem]		BX		AX	Y	Reg3 ← [mem]

Note: All subsequent reads of the logical register AX will refer to Reg 3, the result of the MOV instruction.

1.3.4.2 Data Forwarding

Register renaming alone cannot remove RAW dependencies. The 6x86MX CPU uses two types of data forwarding in conjunction with register renaming to eliminate RAW dependencies:

- Operand Forwarding
- Result Forwarding

Operand forwarding takes place when the first in a pair of instructions performs a move from register or memory, and the data that is read by the first instruction is required by the second instruction. The 6x86MX CPU performs the read operation and makes the data read available to both instructions simultaneously.

Result forwarding takes place when the first in a pair of instructions performs an operation (such as an ADD) and the result is required by the second instruction to perform a move to a register or memory. The 6x86MX CPU performs the required operation and stores the results of the operation to the destination of both instructions simultaneously.

Example #3 - Operand Forwarding Eliminates Read-After-Write (RAW) Dependency

A RAW dependency occurs when the first in a pair of instructions performs a write, and the second instruction reads the same register. This type of dependency is illustrated by the pair of instructions shown below in the X and Y pipelines:

X PIPE	Y PIPE
(1) MOV AX, [mem]	(2) ADD BX, AX
AX ← [mem]	BX ← AX + BX

The 6x86MX CPU uses operand forwarding and avoids a Y pipe stall (Table 1-4). Operand forwarding allows simultaneous execution of both instructions by first reading memory and then making the results available to both pipelines in parallel.

Table 1-4. Example of Operand Forwarding

Instruction	Physical Register Contents				Action	
	Reg0	Reg1	Reg2	Reg3	Pipe	
(Initial)	AX	BX				
MOV AX, [mem]		BX	AX		X	Reg2 ← [mem]
ADD BX, AX			AX	BX	Y	Reg3 ← [mem] + Reg1

Operand forwarding can only occur if the first instruction does not modify its source data. In other words, the instruction is a move type instruction (for example, MOV, POP, LEA). Operand forwarding occurs for both register and memory operands. The size of the first instruction destination and the second instruction source must match.

Example #4 - Result Forwarding Eliminates Read-After-Write (RAW) Dependency

In this example, a RAW dependency occurs when the first in a pair of instructions performs a write, and the second instruction reads the same register. This dependency is illustrated by the pair of instructions in the X and Y pipelines, as shown below:

X PIPE	Y PIPE
(1) ADD AX, BX	(2) MOV [mem], AX
AX ←AX + BX	[mem] ← AX

The 6x86MX CPU uses result forwarding and avoids a Y pipe stall (Table 1-5). Instead of transferring the contents of the AX register to memory, the result of the previous ADD instruction (Reg0 + Reg1) is written directly to memory, thereby saving a clock cycle.

Table 1-5. Result Forwarding Example

Instruction	Physical Register Contents			Action	
	Reg0	Reg1	Reg2	Pipe	
(Initial)	AX	BX			
ADD AX, BX		BX	AX	X	Reg2 ←Reg0 + Reg1
MOV [mem], AX		BX	AX	Y	[mem] ← Reg0 +Reg1

The second instruction must be a move instruction and the destination of the second instruction may be either a register or memory.

1.3.4.3 Data Bypassing

In addition to register renaming and data forwarding, the 6x86MX CPU implements a third data dependency-resolution technique called data bypassing. Data bypassing reduces the performance penalty of those memory data RAW dependencies that cannot be eliminated by data forwarding.

Data bypassing is implemented when the first in a pair of instructions writes to memory and the second instruction reads the same data from memory. The 6x86MX CPU retains the data from the first instruction and passes it to the second instruction, thereby eliminating a memory read cycle. Data bypassing only occurs for cacheable memory locations.

Example #1- Data Bypassing with Read-After-Write (RAW) Dependency

In this example, a RAW dependency occurs when the first in a pair of instructions performs a write to memory and the second instruction reads the same memory location. This dependency is illustrated by the pair of instructions in the X and Y pipelines as shown below:

X PIPE	**Y PIPE**
(1) ADD [mem], AX	(2) SUB BX, [mem]
[mem] ←[mem] + AX	BX ← BX - [mem]

The 6x86MX CPU uses data bypassing and stalls the Y pipe for only one clock by eliminating the Y pipe's memory read cycle (Table 1-6). Instead of reading memory in the Y pipe, the result of the previous instruction ([mem] + Reg0) is used to subtract from Reg1, thereby saving a memory access cycle.

Table 1-6. Example of Data Bypassing

Instruction	Physical Register Contents			Action	
	Reg0	Reg1	Reg2	Pipe	
(Initial)	AX	BX			
ADD [mem], AX	AX	BX		X	[mem] ← [mem] + Reg0
SUB BX, [mem]	AX		BX	Y	Reg2 ← Reg1 - {[mem] + Reg0}

1.3.5 Branch Control

Branch instructions occur on average every four to six instructions in x86-compatible programs. When the normal sequential flow of a program changes due to a branch instruction, the pipeline stages may stall while waiting for the CPU to calculate, retrieve, and decode the new instruction stream. The 6x86MX CPU minimizes the performance degradation and latency of branch instructions through the use of branch prediction and speculative execution.

1.3.5.1 Branch Prediction

The 6x86MX CPU uses a 512-entry, 4-way set associative Branch Target Buffer (BTB) to store branch target addresses. The 6x86MX CPU has 1024-entry branch history table. During the fetch stage, the instruction stream is checked for the presence of branch instructions. If an unconditional branch instruction is encountered, the 6x86MX CPU accesses the BTB to check for the branch instruction's target address. If the branch instruction's target address is found in the BTB, the 6x86MX CPU begins fetching at the target address specified by the BTB.

In case of conditional branches, the BTB also provides history information to indicate whether the branch is more likely to be taken or not taken. If the conditional branch instruction is found in the BTB, the 6x86MX CPU begins fetching instructions at the predicted target address. If the conditional branch misses in the BTB, the 6x86MX CPU predicts that the branch will not be taken, and instruction fetching continues with the next sequential instruction. The decision to fetch the taken or not taken target address is based on a four-state branch prediction algorithm.

Once fetched, a conditional branch instruction is first decoded and then dispatched to the X pipeline only. The conditional branch instruction proceeds through the X pipeline and is then resolved in either the EX stage or the WB stage. The conditional branch is resolved in the EX stage, if the instruction responsible for setting the condition codes is completed prior to the execution of the branch. If the instruction that sets the condition codes is executed in parallel with the branch, the conditional branch instruction is resolved in the WB stage.

Correctly predicted branch instructions execute in a single core clock. If resolution of a branch indicates that a misprediction has occurred, the 6x86MX CPU flushes the pipeline and starts fetching from the correct target address. The 6x86MX CPU prefetches both the predicted and the non-predicted path for each conditional branch, thereby eliminating the cache access cycle on a misprediction. If the branch is resolved in the EX stage, the resulting misprediction latency is four cycles. If the branch is resolved in the WB stage, the latency is five cycles.

Since the target address of return (RET) instructions is dynamic rather than static, the 6x86MX CPU caches target addresses for RET instructions in an eight-entry return stack rather than in the BTB. The return address is pushed on the return stack during a CALL instruction and popped during the corresponding RET instruction.

1.3.5.2 Speculative Execution

The 6x86MX CPU is capable of speculative execution following a floating point instruction or predicted branch. Speculative execution allows the pipelines to continuously execute instructions following a branch without stalling the pipelines waiting for branch resolution. The same mechanism is used to execute floating point instructions (see Section 1.6) in parallel with integer instructions.

The 6x86MX CPU is capable of up to four levels of speculation (i.e., combinations of four conditional branches and floating point operations). After generating the fetch address using branch prediction, the CPU checkpoints the machine state (registers, flags, and processor environment), increments the speculation level counter, and begins operating on the predicted instruction stream.

Once the branch instruction is resolved, the CPU decreases the speculation level. For a correctly predicted branch, the status of the checkpointed resources is cleared. For a branch misprediction, the 6x86MX processor generates the correct fetch address and uses the checkpointed values to restore the machine state in a single clock.

In order to maintain compatibility, writes that result from speculatively executed instructions are not permitted to update the cache or external memory until the appropriate branch is resolved. Speculative execution continues until one of the following conditions occurs:

1) A branch or floating point operation is decoded and the speculation level is already at four.

2) An exception or a fault occurs.

3) The write buffers are full.

4) An attempt is made to modify a non-checkpointed resource (i.e., segment registers, system flags).

1.4 Cache Units

The 6x86MX CPU employs two caches, the Unified Cache and the Instruction Line Cache (Figure 1-2, Page 1-15). The main cache is a 4-way set-associative 64-KByte unified cache. The unified cache provides a higher hit rate than using equal-sized separate data and instruction caches. While in Cyrix SMM mode both SMM code and data are cacheable.

The instruction line cache is a fully associative 256-byte cache. This cache avoids excessive conflicts between code and data accesses in the unified cache.

1.4.1 Unified Cache

The 64-KByte unified write-back cache functions as the primary data cache and as the secondary instruction cache. Configured as a four-way set-associative cache, the cache stores up to 64 KBytes of code and data in 2048 lines. The cache is dual-ported and allows any

two of the following operations to occur in parallel:

- Code fetch
- Data read (X pipe, Y pipeline or FPU)
- Data write (X pipe, Y pipeline or FPU)

The unified cache uses a pseudo-LRU replacement algorithm and can be configured to allocate new lines on read misses only or on read and write misses.

1.4.2 Instruction Line Cache

The fully associative 256-byte instruction line cache serves as the primary instruction cache. The instruction line cache is filled from the unified cache through the data bus. Fetches from the integer unit that hit in the instruction line cache do not access the unified cache. If an instruction line cache miss occurs, the instruction line data from the unified cache is transferred to the instruction line cache and the integer unit, simultaneously.

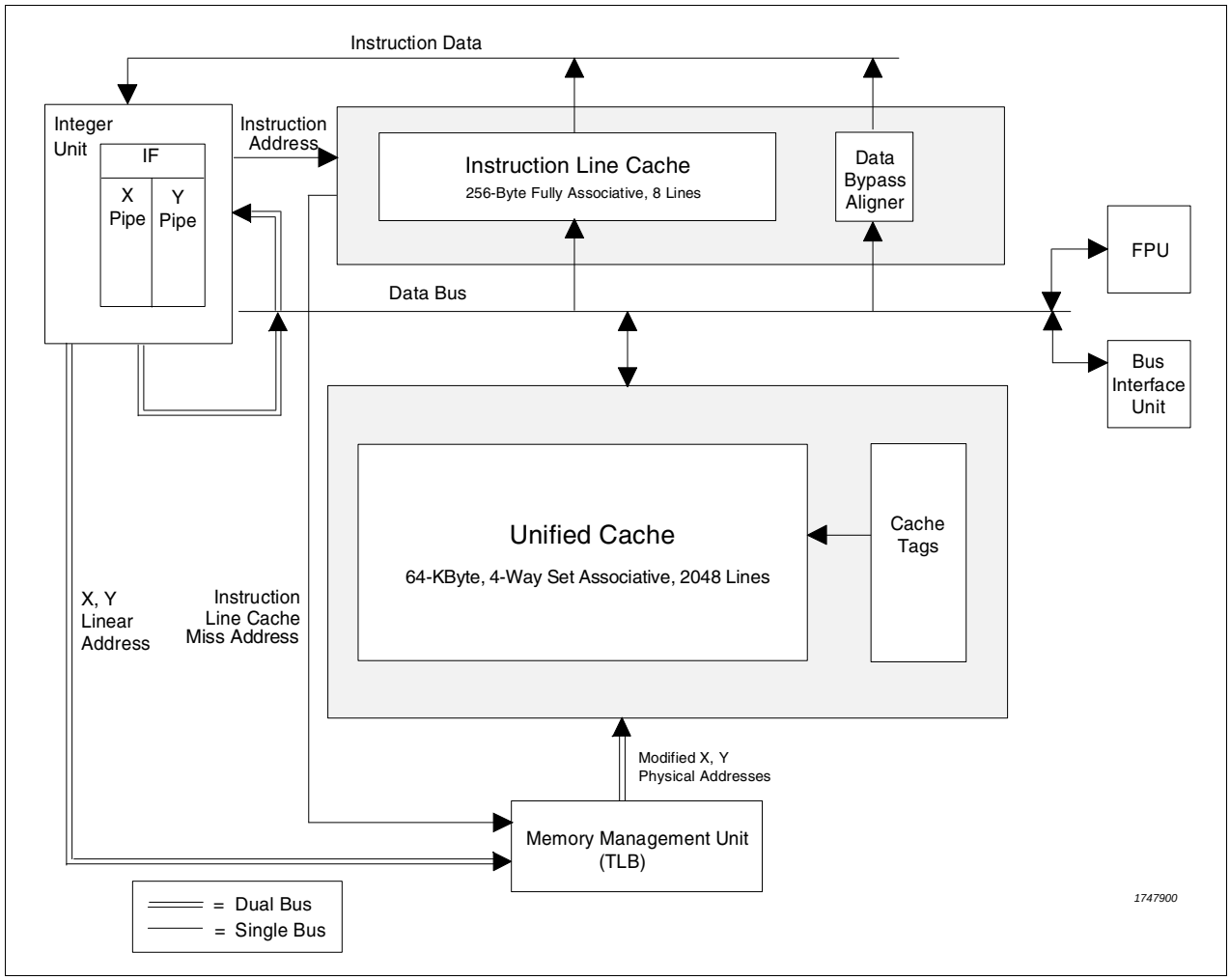


Figure 1-2. Cache Unit Operations

The instruction line cache uses a pseudo-LRU replacement algorithm. To ensure proper operation in the case of self-modifying code, any writes to the unified cache are checked against the contents of the instruction line cache. If a hit occurs in the instruction line cache, the appropriate line is invalidated.

1.5 Memory Management Unit

The Memory Management Unit (MMU), shown in Figure 1-3, translates the linear address supplied by the IU into a physical address to be used by the unified cache and the bus interface. Memory management procedures are x86 compatible, adhering to standard paging mechanisms.

Within the 6x86MX CPU there are two TLBs, the main L1 TLB and the larger L2 TLB. The 16-entry L1 TLB is direct mapped and holds 42 lines. The 384-entry L2 TLB is 6-way associative and hold 384 lines. The DTE is located in memory.

Scratch Pad Cache Memory

The 6x86MX CPU has the capability to "lock down" lines in the L1 cache on a line by line basis. Locked down lines are treated as private memory for use by the CPU. Locked down memory does not participate in hardware--cache coherency protocols.

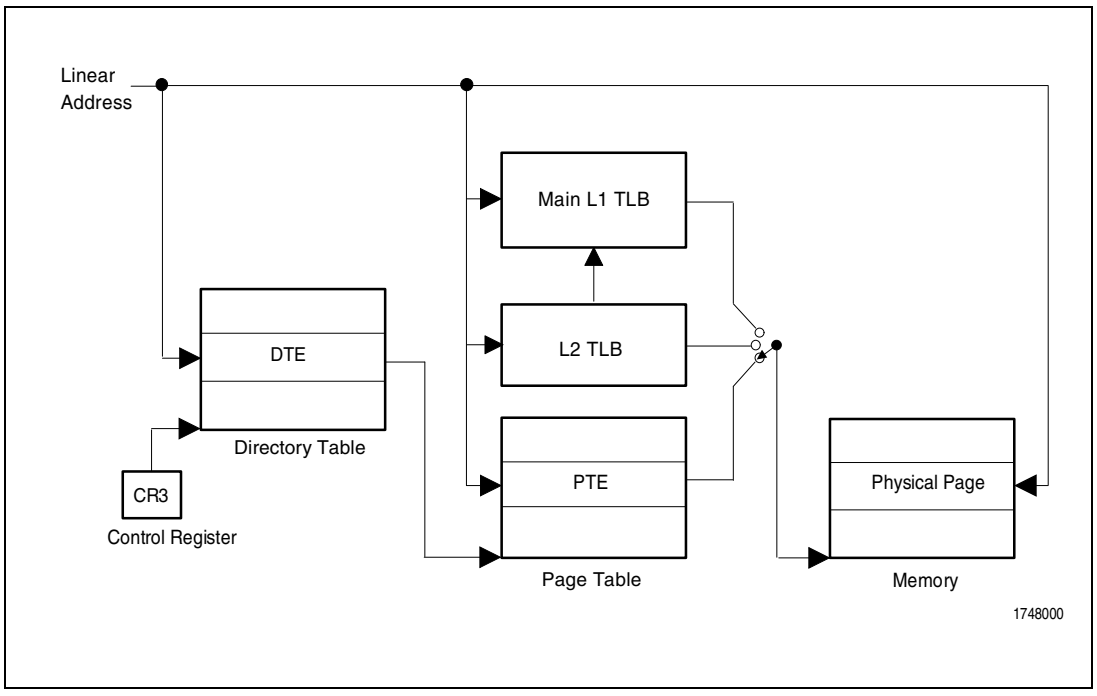


Figure 1-3. Paging Mechanism within the Memory Management Unit

Cache locking is controlled through use of the RDMSR and WRMSR instructions.

1.6 Floating Point Unit

The 6x86MX Floating Point Unit (FPU) processes floating point and MMX instructions. The FPU interfaces to the integer unit and the cache unit through a 64-bit bus. The 6x86MX FPU is x87 instruction set compatible and adheres to the IEEE-754 standard. Since most applications contain FPU instructions mixed with integer instructions, the 6x86MX FPU achieves high performance by completing integer and FPU operations in parallel.

FPU Parallel Execution

The 6x86MX CPU executes integer instructions in parallel with FPU instructions. Integer instructions may complete out of order with respect to the FPU instructions. The 6x86MX CPU maintains x86 compatibility by signaling exceptions and issuing write cycles in program order.

As previously discussed, FPU instructions are always dispatched to the integer unit's X pipeline. The address calculation stage of the X pipeline checks for memory management exceptions and accesses memory operands used by the FPU. If no exceptions are detected, the 6x86MX CPU checkpoints the state of the CPU and, during AC2, dispatches the floating point instruction to the FPU instruction queue. The 6x86MX CPU can then complete any subsequent integer instructions speculatively and out of order relative to the FPU instruction and relative to any potential FPU exceptions which may occur.

As additional FPU instructions enter the pipeline, the 6x86MX CPU dispatches up to four FPU instructions to the FPU instruction queue. The 6x86MX CPU continues executing speculatively and out of order, relative to the FPU queue, until the 6x86MX CPU encounters one of the conditions that causes speculative execution to halt. As the FPU completes instructions, the speculation level decreases and the checkpointed resources are available for reuse in subsequent operations. The 6x86MX FPU also uses a set of six write buffers to prevent stalls due to speculative writes.

1.7 Bus Interface Unit

The Bus Interface Unit (BIU) provides the signals and timing required by external circuitry. The signal descriptions and bus interface timing information is provided in Chapters 3 and 4 of this manual.

[redacted]

2. PROGRAMMING INTERFACE

In this chapter, the internal operations of the 6x86MX CPU are described mainly from an application programmer's point of view. Included in this chapter are descriptions of processor initialization, the register set, memory addressing, various types of interrupts and the shutdown and halt process. An overview of real, virtual 8086, and protected operating modes is also included in this chapter. The FPU operations are described separately at the end of the chapter.

This manual does not—and is not intended to—describe the 6x86MX processor or its operations at the circuit level.

2.1 Processor Initialization

The 6x86MX CPU is initialized when the RESET signal is asserted. The processor is placed in real mode and the registers listed in Table 2-1 (Page 2-2) are set to their initialized values. RESET invalidates and disables the cache and turns off paging. When RESET is asserted, the 6x86MX CPU terminates all local bus activity and all internal execution. During the entire time that RESET is asserted, the internal pipelines are flushed and no instruction execution or bus activity occurs.

Approximately 150 to 250 external clock cycles after RESET is negated, the processor begins executing instructions at the top of physical memory (address location FFFF FFF0h). Typically, an intersegment JUMP is placed at FFFF FFF0h. This instruction will force the processor to begin execution in the lowest 1 MByte of address space.

Note: The actual time depends on the clock scaling in use. Also an additional 2^{20} clock cycles are needed if self-test is requested.

Table 2-1. Initialized Register Controls

REGISTER	REGISTER NAME	INITIALIZED CONTENTS	COMMENTS
EAX	Accumulator	xxxx xxxxh	0000 0000h indicates self-test passed.
EBX	Base	xxxx xxxxh	
ECX	Count	xxxx xxxxh	
EDX	Data	06 + Device ID	Device ID = 51h or 59h (2X clock) Device ID = 55h or 5Ah (2.5X clock) Device ID = 53h or 5Bh (3X clock) Device ID = 54h or 5Ch (3.5X clock)
EBP	Base Pointer	xxxx xxxxh	
ESI	Source Index	xxxx xxxxh	
EDI	Destination Index	xxxx xxxxh	
ESP	Stack Pointer	xxxx xxxxh	
EFLAGS	Flag Word	0000 0002h	
EIP	Instruction Pointer	0000 FFF0h	
ES	Extra Segment	0000h	Base address set to 0000 0000h. Limit set to FFFFh.
CS	Code Segment	F000h	Base address set to FFFF 0000h. Limit set to FFFFh.
SS	Stack Segment	0000h	Base address set to 0000 0000h. Limit set to FFFFh.
DS	Data Segment	0000h	Base address set to 0000 0000h. Limit set to FFFFh.
FS	Extra Segment	0000h	Base address set to 0000 0000h. Limit set to FFFFh.
GS	Extra Segment	0000h	Base address set to 0000 0000h. Limit set to FFFFh.
IDTR	Interrupt Descriptor Table Register	Base = 0, Limit = 3FFh	
GDTR	Global Descriptor Table Register	xxxx xxxxh, xxxxh	
LDTR	Local Descriptor Table Register	xxxx xxxxh, xxxxh	
TR	Task Register	xxxxh	
CR0	Machine Status Word	6000 0010h	
CR2	Control Register 2	xxxx xxxxh	
CR3	Control Register 3	xxxx xxxxh	
CR4	Control Register 4	0000 0000h	
CCR (0-6)	Configuration Control (0-6)	00h CCR(0-3, 5-6) 80h CCR4	
ARR (0-7)	Address Region Registers (0-7)	00h	
RCR (0-7)	Region Control Registers (0-7)	00h	
DR7	Debug Register 7	0000 0400h	

Note: x = Undefined value

2.2 Instruction Set Overview

The 6x86MX CPU instruction set performs ten types of general operations:

- Arithmetic
- Bit Manipulation
- Control Transfer
- Data Transfer
- Floating Point
- High-Level Language Support
- Operating System Support
- Shift/Rotate
- String Manipulation
- MMX Instructions

All 6x86MX CPU instructions operate on as few as zero operands and as many as three operands. An NOP instruction (no operation) is an example of a zero operand instruction. Two operand instructions allow the specification of an explicit source and destination pair as part of the instruction. These two operand instructions can be divided into eight groups according to operand types:

- Register to Register
- Register to Memory
- Memory to Register
- Memory to Memory
- Register to I/O
- I/O to Register
- Immediate Data to Register
- Immediate Data to Memory

An operand can be held in the instruction itself (as in the case of an immediate operand), in one of the processor's registers or I/O ports, or in memory. An immediate operand is prefetched as part of the opcode for the instruction.

Operand lengths of 8, 16, or 32 bits are supported as well as 64-or 80-bit associated with floating point instructions. Operand lengths of 8 or 32 bits are generally used when executing code written for 386- or 486-class (32-bit code) processors. Operand lengths of 8 or 16 bits are generally used when executing existing 8086 or 80286 code (16-bit code). The default length of an operand can be overridden by placing one or more instruction prefixes in front of the opcode. For example, by using prefixes, a 32-bit operand can be used with 16-bit code, or a 16-bit operand can be used with 32-bit code.

Chapter 6 of this manual lists each instruction in the 6x86MX CPU instruction set along with the associated opcodes, execution clock counts, and effects on the FLAGS register.

2.2.1 Lock Prefix

The LOCK prefix may be placed before certain instructions that read, modify, then write back to memory. The prefix asserts the LOCK# signal to indicate to the external hardware that the CPU is in the process of running multiple indivisible memory accesses. The LOCK prefix can be used with the following instructions:

Bit Test Instructions (BTS, BTR, BTC)
Exchange Instructions (XADD, XCHG, CMPXCHG)
One-operand Arithmetic and Logical Instructions (DEC, INC, NEG, NOT)
Two-operand Arithmetic and Logical Instructions (ADC, ADD, AND, OR, SBB, SUB, XOR).

An invalid opcode exception is generated if the LOCK prefix is used with any other instruction, or with the above instructions when no write operation to memory occurs (i.e., the destination is a register). The LOCK# signal can be negated to allow weak-locking for all of memory or on a regional basis. Refer to the descriptions of the NO-LOCK bit (within CCR1) and the WL bit (within RCRx) later in this chapter.

2.3 Register Sets

From the programmer's point of view there are 58 accessible registers in the 6x86MX CPU. These registers are grouped into two sets. The application register set contains the registers frequently used by application programmers, and the system register set contains the registers typically reserved for use by operating system programmers.

The application register set is made up of general purpose registers, segment registers, a flag register, and an instruction pointer register.

The system register set is made up of the remaining registers which include control registers, system address registers, debug registers, configuration registers, and test registers.

Each of the registers is discussed in detail in the following sections.

2.3.1 Application Register Set

The application register set, (Figure 2-1, Page 2-5) consists of the registers most often used by the applications programmer. These registers are generally accessible and are not protected from read or write access.

The **General Purpose Register** contents are frequently modified by assembly language instructions and typically contain arithmetic and logical instruction operands.

Segment Registers in real mode contain the base address for each segment. In protected mode the segment registers contain segment selectors. The segment selectors provide indexing for tables (located in memory) that contain the base address and limit for each segment, as well as access control information.

The **Flag Register** contains control bits used to reflect the status of previously executed instructions. This register also contains control bits that affect the operation of some instructions.

The **Instruction Pointer** register points to the next instruction that the processor will execute. This register is automatically incremented by the processor as execution progresses.

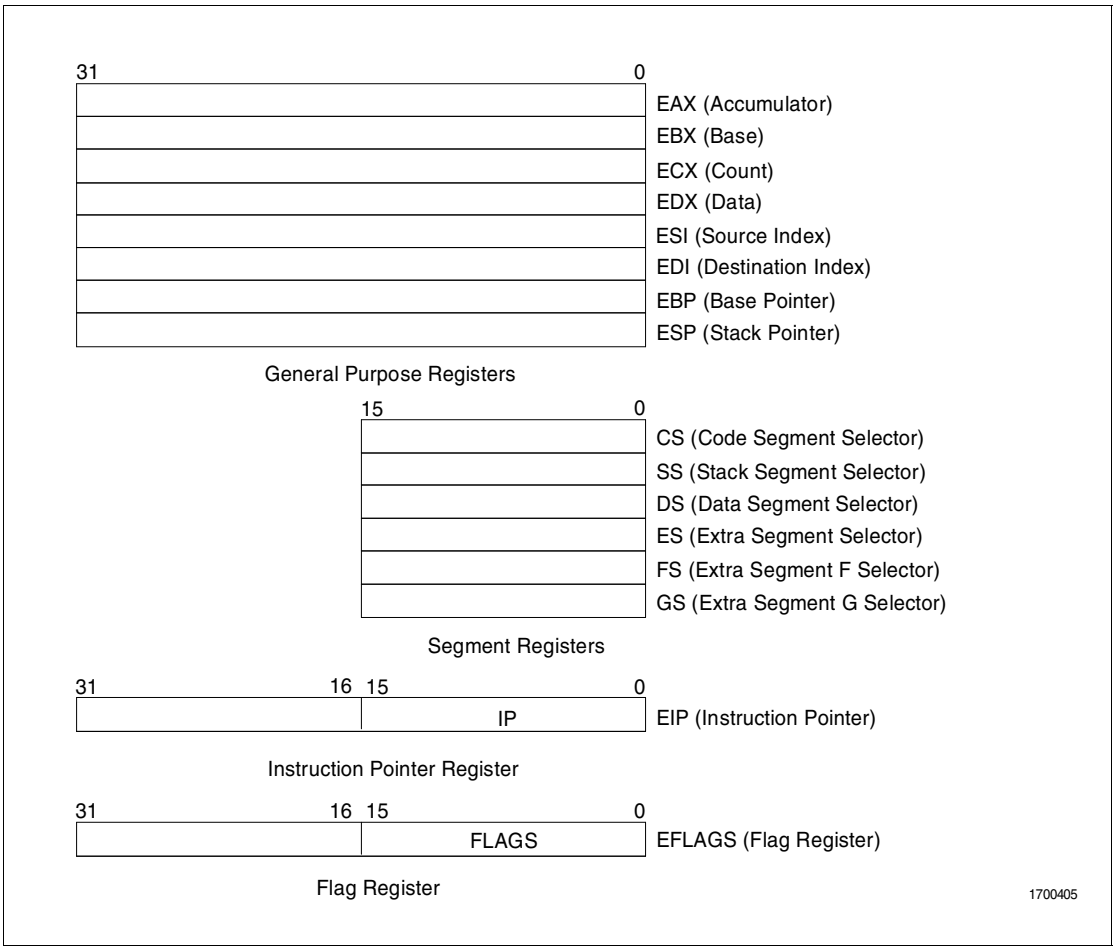


Figure 2-1. Application Register Set

2.3.2 General Purpose Registers

The general purpose registers are divided into four data registers, two pointer registers, and two index registers as shown in Figure 2-2 (Page 2-6).

The **Data Registers** are used by the applications programmer to manipulate data structures and to hold the results of logical and arithmetic operations. Different portions of the general data registers can be addressed by using different names.

An "E" prefix identifies the complete 32-bit register. An "X" suffix without the "E" prefix identifies the lower 16 bits of the register.

The lower two bytes of a data register can be addressed with an "H" suffix (identifies the upper byte) or an "L" suffix (identifies the lower byte). The _L and _H portions of a data registers act as independent registers. For example, if the AH register is written to by an instruction, the AL register bits remain unchanged.

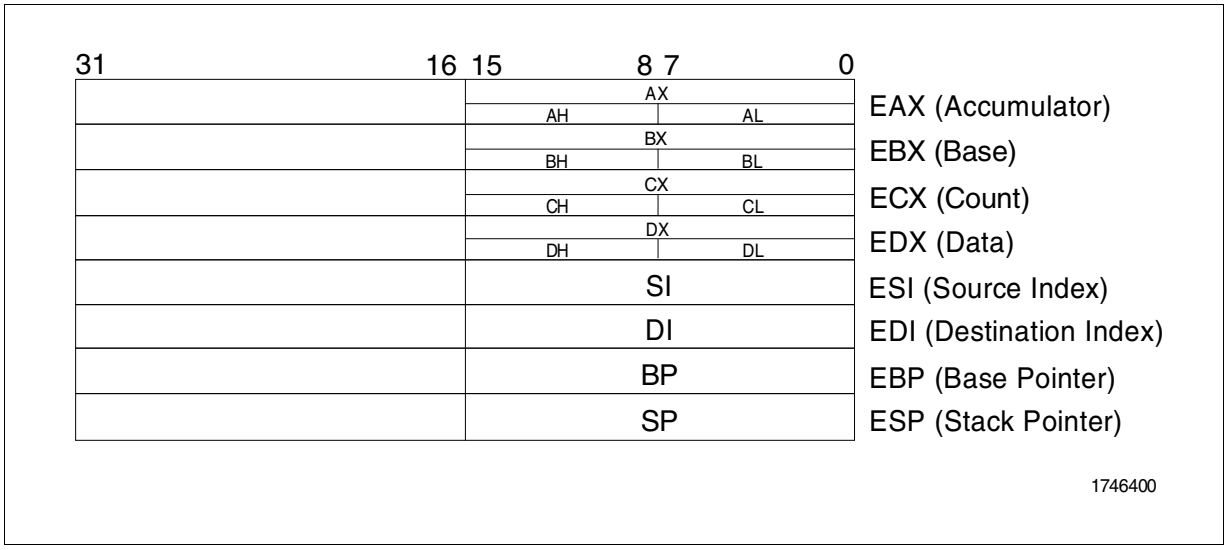


Figure 2-2. General Purpose Registers

The **Pointer and Index Registers** are listed below.

SI or ESI	Source Index
DI or EDI	Destination Index
SP or ESP	Stack Pointer
BP or EBP	Base Pointer

These registers can be addressed as 16- or 32-bit registers, with the "E" prefix indicating 32 bits. The pointer and index registers can be used as general purpose registers, however, some instructions use a fixed assignment of these registers. For example, repeated string operations always use ESI as the source pointer, EDI as the destination pointer, and ECX as the counter. The instructions using fixed registers include multiply and divide, I/O access, string operations, translate, loop, variable shift and rotate, and stack operations.

The 6x86MX processor implements a stack using the ESP register. This stack is accessed during the PUSH and POP instructions, procedure calls, procedure returns, interrupts, exceptions, and interrupt/exception returns.

The microprocessor automatically adjusts the value of the ESP during operation of these instructions.The EBP register may be used to reference data passed on the stack during procedure calls. Local data may also be placed on the stack and referenced relative to BP. This register provides a mechanism to access stack data in high-level languages.

2.3.3 Segment Registers and Selectors

Segmentation provides a means of defining data structures inside the memory space of the microprocessor. There are three basic types of segments: code, data, and stack. Segments are used automatically by the processor to determine the location in memory of code, data, and stack references.

There are six 16-bit segment registers:

CS	Code Segment
DS	Data Segment
ES	Extra Segment
SS	Stack Segment
FS	Additional Data Segment
GS	Additional Data Segment.

In real and virtual 8086 operating modes, a segment register holds a 16-bit segment base. The 16-bit segment is multiplied by 16 and a 16-bit or 32-bit offset is then added to it to create a linear address. The offset size is dependent on the current address size. In real mode and in virtual 8086 mode with paging disabled, the linear address is also the physical address. In virtual 8086 mode with paging enabled, the linear address is translated to the physical address using the current page tables. Paging is described in Section 2.12.4 (Page 2-52).

In protected mode a segment register holds a **Segment Selector** containing a 13-bit index, a Table Indicator (TI) bit, and a two-bit Requested Privilege Level (RPL) field as shown in Figure 2-3.

The **Index** points into a descriptor table in memory and selects one of 8192 (2^{13}) segment descriptors contained in the descriptor table.

A segment descriptor is an eight-byte value used to describe a memory segment by defining the segment base, the segment limit, and access control information. To address data within a segment, a 16-bit or 32-bit offset is added to the segment's base address. Once a segment selector has been loaded into a segment register, an instruction needs only to specify the segment register and the offset.

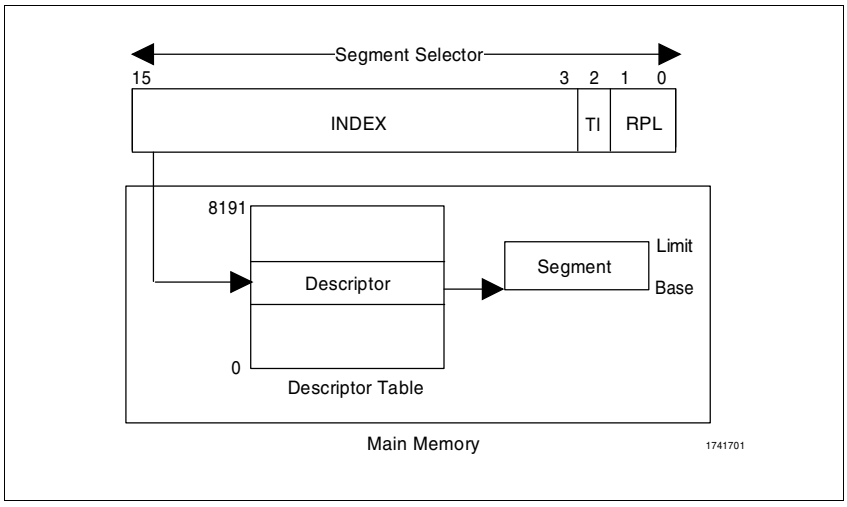


Figure 2-3. Segment Selector in Protected Mode

The **Table Indicator** (TI) bit of the selector defines which descriptor table the index points into. If TI=0, the index references the Global Descriptor Table (GDT). If TI=1, the index references the Local Descriptor Table (LDT). The GDT and LDT are described in more detail in Section 2.4.2 (Page 2-16). Protected mode addressing is discussed further in Sections 2.6.2 (Page 2-52).

The **Requested Privilege Level** (RPL) field in a segment selector is used to determine the Effective Privilege Level of an instruction (where RPL=0 indicates the most privileged level, and RPL=3 indicates the least privileged level).

If the level requested by RPL is less than the Current Program Level (CPL), the RPL level is accepted and the Effective Privilege Level is changed to the RPL value. If the level requested by RPL is greater than CPL, the CPL overrides the requested RPL and Effective Privilege Level remains unchanged.

When a segment register is loaded with a segment selector, the segment base, segment limit and access rights are loaded from the descriptor table entry into a user-invisible or hidden portion of the segment register (i.e., cached on-chip). The CPU does not access the descriptor table entry again until another segment register load occurs. If the descriptor tables are modified in memory, the segment registers must be reloaded with the new selector values by the software.

The processor automatically selects an implied (default) segment register for memory references. Table 2-2 describes the selection rules. In general, data references use the selector contained in the DS register, stack references use the SS register and instruction fetches use the CS register. While some of these selections may be overridden, instruction fetches, stack operations, and the destination write of string operations cannot be overridden. Special segment override instruction prefixes allow the use of alternate segment registers including the use of the ES, FS, and GS segment registers.

Table 2-2. Segment Register Selection Rules

TYPE OF MEMORY REFERENCE	IMPLIED (DEFAULT) SEGMENT	SEGMENT OVERRIDE PREFIX
Code Fetch	CS	None
Destination of PUSH, PUSHF, INT, CALL, PUSHA instructions	SS	None
Source of POP, POPA, POPF, IRET, RET instructions	SS	None
Destination of STOS, MOVS, REP STOS, REP MOVS instructions	ES	None
Other data references with effective address using base registers of:		
EAX, EBX, ECX, EDX, ESI, EDI	DS	CS, ES, FS, GS, SS
EBP, ESP	SS	CS, DS, ES, FS, GS

2.3.4 Instruction Pointer Register

The **Instruction Pointer** (EIP) register contains the offset into the current code segment of the next instruction to be executed. The register is normally incremented with each instruction execution unless implicitly modified through an interrupt, exception or an instruction that changes the sequential execution flow (e.g., JMP, CALL).

2.3.5 Flags Register

The **Flags Register**, EFLAGS, contains status information and controls certain operations on the 6x86MX CPU microprocessor. The lower 16 bits of this register are referred to as the FLAGS register that is used when executing 8086 or 80286 code. The flag bits are shown in Figure 2-4 and defined in Table 2-3 (Page 2-10).

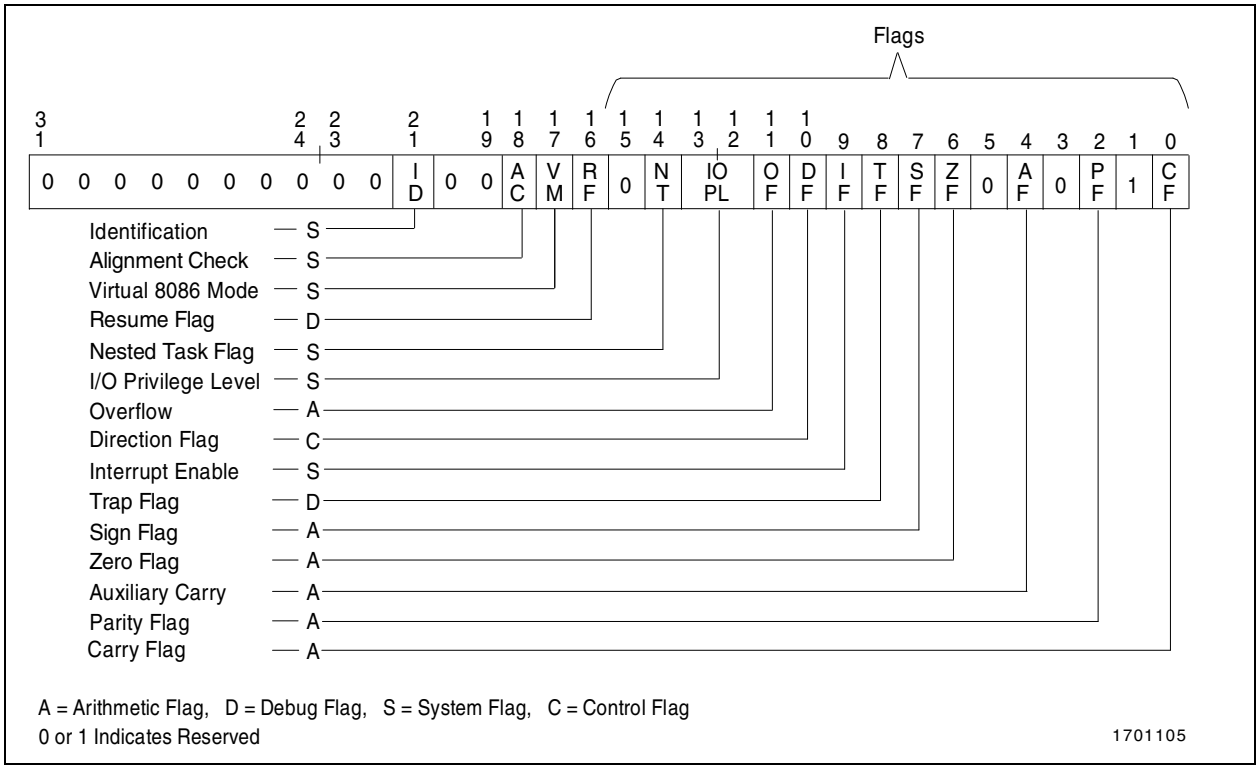


Figure 2-4. EFLAGS Register

Table 2-3. EFLAGS Bit Definitions

BIT POSITION	NAME	FUNCTION
0	CF	Carry Flag: Set when a carry out of (addition) or borrow into (subtraction) the most significant bit of the result occurs; cleared otherwise.
2	PF	Parity Flag: Set when the low-order 8 bits of the result contain an even number of ones; cleared otherwise.
4	AF	Auxiliary Carry Flag: Set when a carry out of (addition) or borrow into (subtraction) bit position 3 of the result occurs; cleared otherwise.
6	ZF	Zero Flag: Set if result is zero; cleared otherwise.
7	SF	Sign Flag: Set equal to high-order bit of result (0 indicates positive, 1 indicates negative).
8	TF	Trap Enable Flag: Once set, a single-step interrupt occurs after the next instruction completes execution. TF is cleared by the single-step interrupt.
9	IF	Interrupt Enable Flag: When set, maskable interrupts (INTR input pin) are acknowledged and serviced by the CPU.
10	DF	Direction Flag: If DF=0, string instructions auto-*in*crement (default) the appropriate index registers (ESI and/or EDI). If DF=1, string instructions auto-*de*crement the appropriate index registers.
11	OF	Overflow Flag: Set if the operation resulted in a carry or borrow into the sign bit of the result but did not result in a carry or borrow out of the high-order bit. Also set if the operation resulted in a carry or borrow out of the high-order bit but did not result in a carry or borrow into the sign bit of the result.
12, 13	IOPL	I/O Privilege Level: While executing in protected mode, IOPL indicates the maximum current privilege level (CPL) permitted to execute I/O instructions without generating an exception 13 fault or consulting the I/O permission bit map. IOPL also indicates the maximum CPL allowing alteration of the IF bit when new values are popped into the EFLAGS register.
14	NT	Nested Task: While executing in protected mode, NT indicates that the execution of the current task is nested within another task.
16	RF	Resume Flag: Used in conjunction with debug register breakpoints. RF is checked at instruction boundaries before breakpoint exception processing. If set, any debug fault is ignored on the next instruction.
17	VM	Virtual 8086 Mode: If set while in protected mode, the microprocessor switches to virtual 8086 operation handling segment loads as the 8086 does, but generating exception 13 faults on privileged opcodes. The VM bit can be set by the IRET instruction (if current privilege level=0) or by task switches at any privilege level.
18	AC	Alignment Check Enable: In conjunction with the AM flag in CR0, the AC flag determines whether or not misaligned accesses to memory cause a fault. If AC is set, alignment faults are enabled.
21	ID	Identification Bit: The ability to set and clear this bit indicates that the CPUID instruction is supported. The ID can be modified only if the CPUID bit in CCR4 is set.

2.4 System Register Set

The system register set, shown in Figure 2-5 (Page 2-12), consists of registers not generally used by application programmers. These registers are typically employed by system level programmers who generate operating systems and memory management programs.

The **Control Registers** control certain aspects of the 6x86MX processor such as paging, coprocessor functions, and segment protection. When a paging exception occurs while paging is enabled, some control registers retain the linear address of the access that caused the exception.

The **Descriptor Table Registers** and the **Task Register** can also be referred to as system address or memory management registers. These registers consist of two 48-bit and two 16-bit registers. These registers specify the location of the data structures that control the segmentation used by the 6x86MX processor. Segmentation is one available method of memory management.

The **Configuration Registers** are used to configure the 6x86MX CPU on-chip cache operation, power management features and System Management Mode. The configuration registers also provide information on the CPU device type and revision.

The **Address Region Registers and Region Control Registers** work together to define address regions which can be given attributes such as cache disable.

The **Model Specific Registers** allow time stamping, event counting, performance monitoring. Cache line locking is also controlled through these registers.

The **Debug Registers** provide debugging facilities to enable the use of data access breakpoints and code execution breakpoints.

The **Test Registers** provide a mechanism to test the contents of both the on-chip 64 KByte cache and the Translation Lookaside Buffer (TLB). In the following sections, the system register set is described in greater detail.

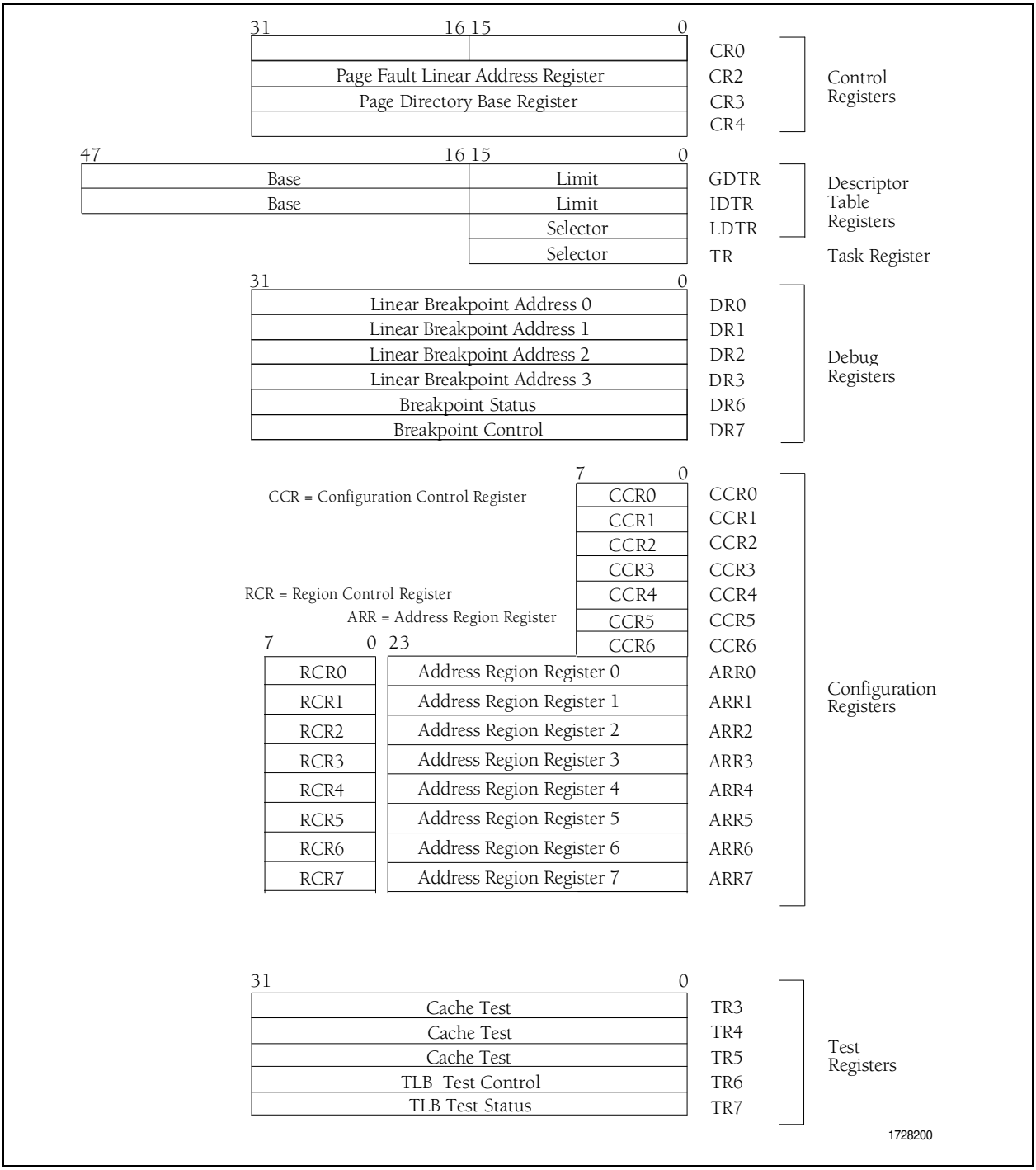


Figure 2-5. System Register Set

2.4.1 Control Registers

The Control Registers (CR0, CR2, CR3 and CR4), are shown in Figure 2-6. (These registers should not be confused with the CCRn registers.) The CR0 register contains system control bits which configure operating modes and indicate the general state of the CPU. The lower 16 bits of CR0 are referred to as the Machine Status Word (MSW). The CR0 bit definitions are described in Table 2-4 and Table 2-5 (Page 2-14). The reserved bits in CR0 should not be modified.

When paging is enabled and a page fault is generated, the CR2 register retains the 32-bit linear address of the address that caused the fault. When a double page fault occurs, CR2 contains the address for the second fault. Register CR3 contains the 20 most significant bits of the physical base address of the page directory. The page directory must always be aligned to a 4-KByte page boundary, therefore, the lower 12 bits of CR3 are not required to specify the base address.

CR3 contains the Page Cache Disable (PCD) and Page Write Through (PWT) bits. During bus cycles that are not paged, the state of the PCD bit is reflected on the PCD pin and the PWT bit is driven on the PWT pin. These bus cycles include interrupt acknowledge cycles and all bus cycles, when paging is not enabled. The PCD pin should be used to control caching in an external cache. The PWT pin should be used to control write policy in an external cache.

Control register CR4 (Table 2-6, Page 2-15) controls usage of the Time Stamp Counter Instruction, Debugging Extensions, Page Global Enable and the RDPMC instruction.

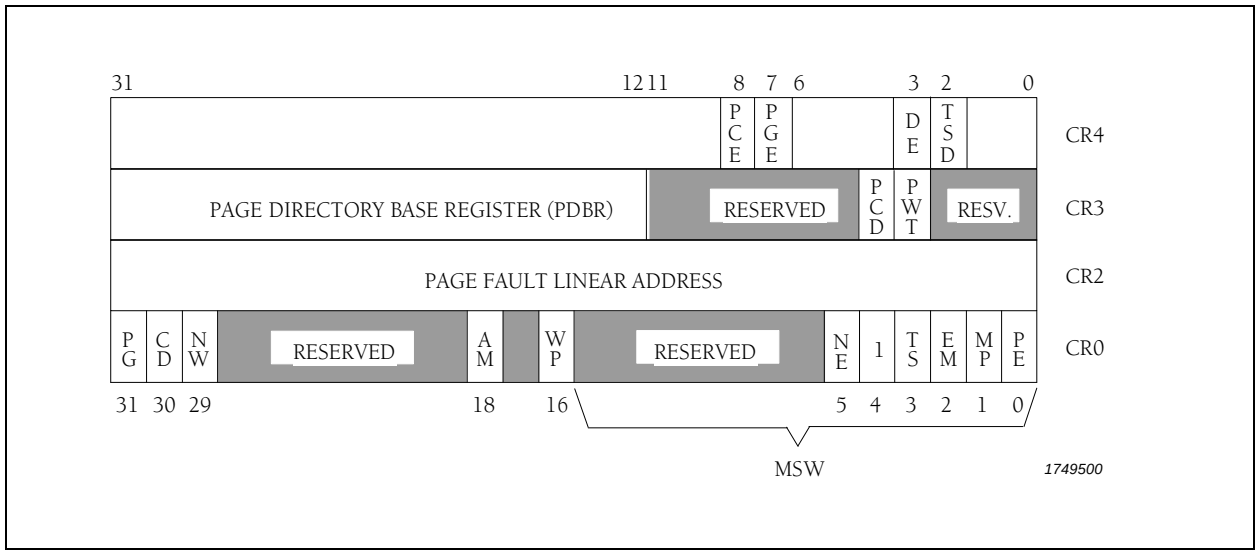


Figure 2-6. Control Registers

Table 2-4. CR0 Bit Definitions

BIT POSITION	NAME	FUNCTION
0	PE	Protected Mode Enable: Enables the segment based protection mechanism. If PE=1, protected mode is enabled. If PE=0, the CPU operates in real mode and addresses are formed as in an 8086-style CPU.
1	MP	Monitor Processor Extension: If MP=1 and TS=1, a WAIT instruction causes Device Not Available (DNA) fault 7. The TS bit is set to 1 on task switches by the CPU. Floating point instructions are not affected by the state of the MP bit. The MP bit should be set to one during normal operations.
2	EM	Emulate Processor Extension: If EM=1, all floating point instructions cause a DNA fault 7.
3	TS	Task Switched: Set whenever a task switch operation is performed. Execution of a floating point instruction with TS=1 causes a DNA fault. If MP=1 and TS=1, a WAIT instruction also causes a DNA fault.
4	1	Reserved: Do not attempt to modify.
5	NE	Numerics Exception. NE=1 to allow FPU exceptions to be handled by interrupt 16. NE=0 if FPU exceptions are to be handled by external interrupts.
16	WP	Write Protect: Protects read-only pages from supervisor write access. WP=0 allows a read-only page to be written from privilege level 0-2. WP=1 forces a fault on a write to a read-only page from any privilege level.
18	AM	Alignment Check Mask: If AM=1, the AC bit in the EFLAGS register is unmasked and allowed to enable alignment check faults. Setting AM=0 prevents AC faults from occurring.
29	NW	Not Write-Back: If NW=1, the on-chip cache operates in write-through mode. In write-through mode, all writes (including cache hits) are issued to the external bus. If NW=0, the on-chip cache operates in write-back mode. In write-back mode, writes are issued to the external bus only for a cache miss, a line replacement of a modified line, or as the result of a cache inquiry cycle.
30	CD	Cache Disable: If CD=1, no further cache line fills occur. However, data already present in the cache continues to be used if the requested address hits in the cache. Writes continue to update the cache and cache invalidations due to inquiry cycles occur normally. The cache must also be invalidated to completely disable any cache activity.
31	PG	Paging Enable Bit: If PG=1 and protected mode is enabled (PE=1), paging is enabled. After changing the state of PG, software must execute an unconditional branch instruction (e.g., JMP, CALL) to have the change take effect.

Table 2-5. Effects of Various Combinations of EM, TS, and MP Bits

CR0 BIT			INSTRUCTION TYPE	
EM	TS	MP	WAIT	ESC
0	0	0	Execute	Execute
0	0	1	Execute	Execute
0	1	0	Execute	Fault 7
0	1	1	Fault 7	Fault 7
1	0	0	Execute	Fault 7
1	0	1	Execute	Fault 7
1	1	0	Execute	Fault 7
1	1	1	Fault 7	Fault 7

Table 2-6. CR4 Bit Definitions

BIT POSITION	NAME	FUNCTION
2	TSD	Time Stamp Counter Instruction If = 1 RDTSC instruction enabled for CPL=0 only; Reset State If = 0 RDTSC instruction enabled for all CPL states
3	DE	Debugging Extensions If = 1 enables I/O breakpoints and R/W bits for each debug register are defined as: 00 -Break on instruction execution only. 01 -Break on data writes only. 10 -Break on I/O reads or writes. 11 -Break on data reads or writes but not instruction fetches. If = 0 I/O breakpoints and R/W bits for each debug register are not enabled.
7	PGE	Page Global Enable If = 1 global page feature is enabled. If = 0 global page feature is disabled. Global pages are not flushed from TLB on a task switch or write to CR3
8	PCE	Performance Monitoring Counter Enable If = 1 enables execution of RDPMC instruction at any protection level. If = 0 RDPMC instruction can only be executed at protection level 0.

2.4.2 Descriptor Table Registers and Descriptors

Descriptor Table Registers

The Global, Interrupt, and Local Descriptor Table Registers (GDTR, IDTR and LDTR), shown in Figure 2-7, are used to specify the location of the data structures that control segmented memory management. The GDTR, IDTR and LDTR are loaded using the LGDT, LIDT and LLDT instructions, respectively. The values of these registers are stored using the corresponding store instructions. The GDTR and IDTR load instructions are privileged instructions when operating in protected mode. The LDTR can only be accessed in protected mode.

The **Global Descriptor Table Register** (GDTR) holds a 32-bit linear base address and 16-bit limit for the Global Descriptor Table (GDT). The GDT is an array of up to 8192 8-byte descriptors. When a segment register is loaded from memory, the TI bit in the segment selector chooses either the GDT or the Local Descriptor Table (LDT) to locate a descriptor. If TI = 0, the index portion of the selector is used to locate the descriptor within the GDT table. The contents of the GDTR are completely visible to the programmer by using a SGDT instruction. The first descriptor in the GDT (location 0) is not used by the CPU and is referred to as the "null descriptor". The GDTR is initialized using a LGDT instruction.

The **Interrupt Descriptor Table Register** (IDTR) holds a 32-bit linear base address and 16-bit limit for the Interrupt Descriptor Table (IDT). The IDT is an array of 256 interrupt descriptors, each of which is used to point to an interrupt service routine. Every interrupt that may occur in the system must have an associated entry in the IDT. The contents of the IDTR are completely visible to the programmer by using a SIDT instruction. The IDTR is initialized using the LIDT instruction.

The **Local Descriptor Table Register** (LDTR) holds a 16-bit selector for the Local Descriptor Table (LDT). The LDT is an array of up to 8192 8-byte descriptors. When the LDTR is loaded, the LDTR selector indexes an LDT descriptor that resides in the Global Descriptor Table (GDT). The base address and limit are loaded automatically and cached from the LDT descriptor within the GDT.

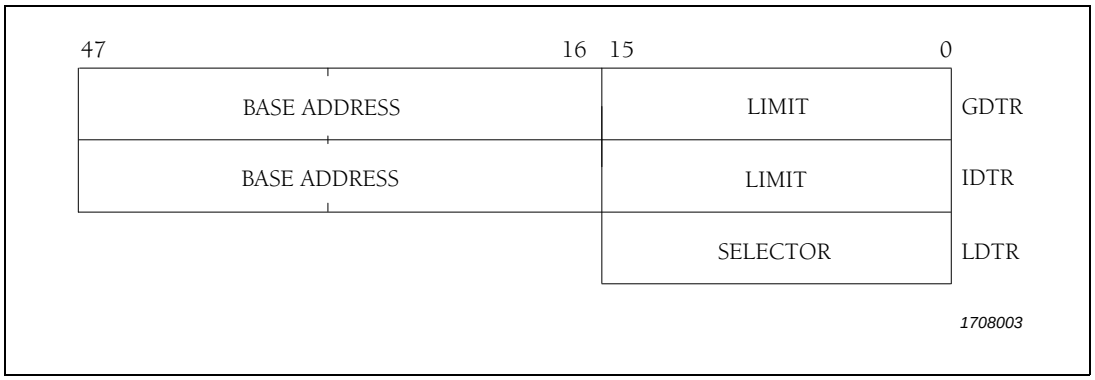


Figure 2-7. Descriptor Table Registers

Subsequent access to entries in the LDT use the hidden LDTR cache to obtain linear addresses. If the LDT descriptor is modified in the GDT, the LDTR must be reloaded to update the hidden portion of the LDTR.

When a segment register is loaded from memory, the TI bit in the segment selector chooses either the GDT or the LDT to locate a segment descriptor. If TI = 1, the index portion of the selector is used to locate a given descriptor within the LDT. Each task in the system may be given its own LDT, managed by the operating system. The LDTs provide a method of isolating a given task's segments from other tasks in the system.

The LDTR can be read or written by the LLDT and SLDT instructions.

Descriptors

There are three types of descriptors:

- Application Segment Descriptors that define code, data and stack segments.
- System Segment Descriptors that define an LDT segment or a Task State Segment (TSS) table described later in this text.
- Gate Descriptors that define task gates, interrupt gates, trap gates and call gates.

Application Segment Descriptors can be located in either the LDT or GDT. System Segment Descriptors can only be located in the GDT. Dependent on the gate type, gate descriptors may be located in either the GDT, LDT or IDT. Figure 2-8 illustrates the descriptor format for both Application Segment Descriptors and System Segment Descriptors. Table 2-7 (Page 2-18) lists the corresponding bit definitions.

Table 2-8. (Page 2-18) and Table 2-9. (Page 2-19) defines the DT field within the segment descriptor.

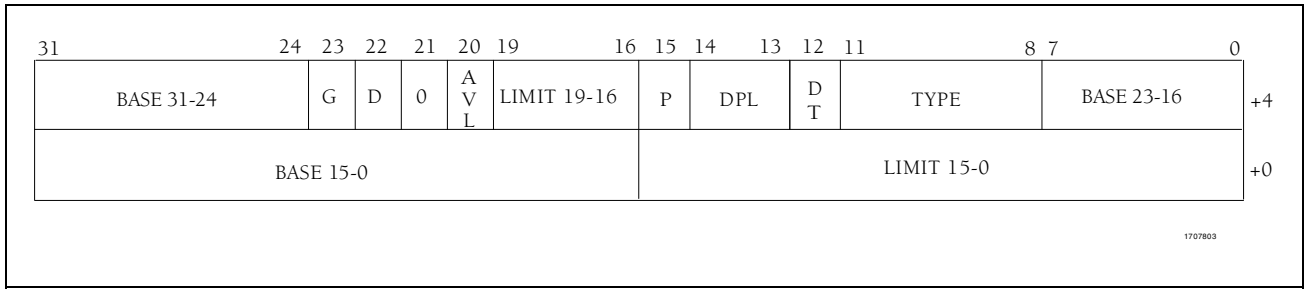


Figure 2-8. Application and System Segment Descriptors

Table 2-7. Segment Descriptor Bit Definitions

BIT POSITION	MEMORY OFFSET	NAME	DESCRIPTION
31-24 7-0 31-16	+4 +4 +0	BASE	Segment base address. 32-bit linear address that points to the beginning of the segment.
19-16 15-0	+4 +0	LIMIT	Segment limit.
23	+4	G	Limit granularity bit: 0 = byte granularity, 1 = 4 KBytes (page) granularity.
22	+4	D	Default length for operands and effective addresses. Valid for code and stack segments only: 0 = 16 bit, 1 = 32-bit.
20	+4	AVL	Segment available.
15	+4	P	Segment present.
14-13	+4	DPL	Descriptor privilege level.
12	+4	DT	Descriptor type: 0 = system, 1 = application.
11-8	+4	TYPE	Segment type. See Tables 2-7 and 2-8.

Table 2-8. TYPE Field Definitions with DT = 0

TYPE (BITS 11-8)	DESCRIPTION
0001	TSS-16 descriptor, task not busy.
0010	LDT descriptor.
0011	TSS-16 descriptor, task busy.
1001	TSS-32 descriptor, task not busy
1011	TSS-32 descriptor, task busy.

Table 2-9. TYPE Field Definitions with DT = 1

TYPE				APPLICATION DECRIPTOR INFORMATION
E	C/D	R/W	A	
0	0	x	x	data, expand up, limit is upper bound of segment
0	1	x	x	data, expand down, limit is lower bound of segment
1	0	x	x	executable, non-conforming
1	1	x	x	executable, conforming (runs at privilege level of calling procedure)
0	x	0	x	data, non-writable
0	x	1	x	data, writable
1	x	0	x	executable, non-readable
1	x	1	x	executable, readable
x	x	x	0	not-accessed
x	x	x	1	accessed

Gate Descriptors provide protection for executable segments operating at different privilege levels. Figure 2-9 illustrates the format for Gate Descriptors and Table 2-10 lists the corresponding bit definitions.

Task Gate Descriptors are used to switch the CPU's context during a task switch. The selector portion of the task gate descriptor locates a Task State Segment. These descriptors can be located in the GDT, LDT or IDT tables.

Interrupt Gate Descriptors are used to enter a hardware interrupt service routine. Trap Gate Descriptors are used to enter exceptions or software interrupt service routines. Trap Gate and Interrupt Gate Descriptors can only be located in the IDT.

Call Gate Descriptors are used to enter a procedure (subroutine) that executes at the same or a more privileged level. A Call Gate Descriptor primarily defines the procedure entry point and the procedure's privilege level.

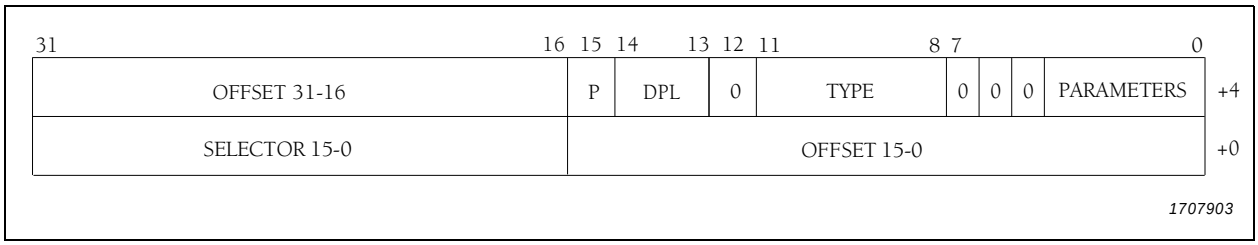


Figure 2-9. Gate Descriptor

Table 2-10. Gate Descriptor Bit Definitions

BIT POSITION	MEMORY OFFSET	NAME	DESCRIPTION
31-16 15-0	+4 +0	OFFSET	Offset used during a call gate to calculate the branch target.
31-16	+0	SELECTOR	Segment selector used during a call gate to calculate the branch target.
15	+4	P	Segment present.
14-13	+4	DPL	Descriptor privilege level.
11-8	+4	TYPE	Segment type: 0100 = 16-bit call gate 0101 = task gate 0110 = 16-bit interrupt gate 0111 = 16-bit trap gate 1100 = 32-bit call gate 1110 = 32-bit interrupt gate 1111 = 32-bit trap gate.
4-0	+4	PARAMETERS	Number of 32-bit parameters to copy from the caller's stack to the called procedure's stack (valid for calls).

2.4.3 Task Register

The **Task Register** (TR) holds a 16-bit selector for the current Task State Segment (TSS) table as shown in Figure 2-10. The TR is loaded and stored via the LTR and STR instructions, respectively. The TR can only be accessed during protected mode and can only be loaded when the privilege level is 0 (most privileged). When the TR is loaded, the TR selector field indexes a TSS descriptor that must reside in the Global Descriptor Table (GDT). The contents of the selected descriptor are cached on-chip in the hidden portion of the TR.

During task switching, the processor saves the current CPU state in the TSS before starting a new task. The TR points to the current TSS. The TSS can be either a 386/486-style 32-bit TSS (Figure 2-11, Page 2-22) or a 286-style 16-bit TSS type (Figure 2-12, Page 2-23). An I/O permission bit map is referenced in the 32-bit TSS by the I/O Map Base Address.

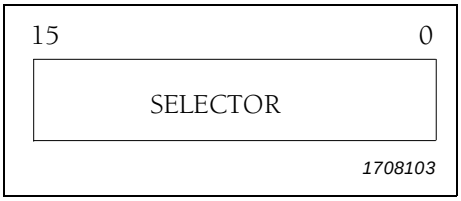


Figure 2-10. Task Register

31 16	15 0	
I/O MAP BASE ADDRESS	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	T +64h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SELECTOR FOR TASK'S LDT	+60h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	GS	+5Ch
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	FS	+58h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	DS	+54h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS	+50h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	CS	+4Ch
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ES	+48h
EDI		+44h
ESI		+40h
EBP		+3Ch
ESP		+38h
EBX		+34h
EDX		+30h
ECX		+2Ch
EAX		+28h
EFLAGS		+24h
EIP		+20h
CR3		+1Ch
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS for CPL = 2	+18h
ESP for CPL = 2		+14h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS for CPL = 1	+10h
ESP for CPL = 1		+Ch
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS for CPL = 0	+8h
ESP for CPL = 0		+4h
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	BACK LINK (OLD TSS SELECTOR)	+0h

0 = RESERVED.

1708203

Figure 2-11. 32-Bit Task State Segment (TSS) Table

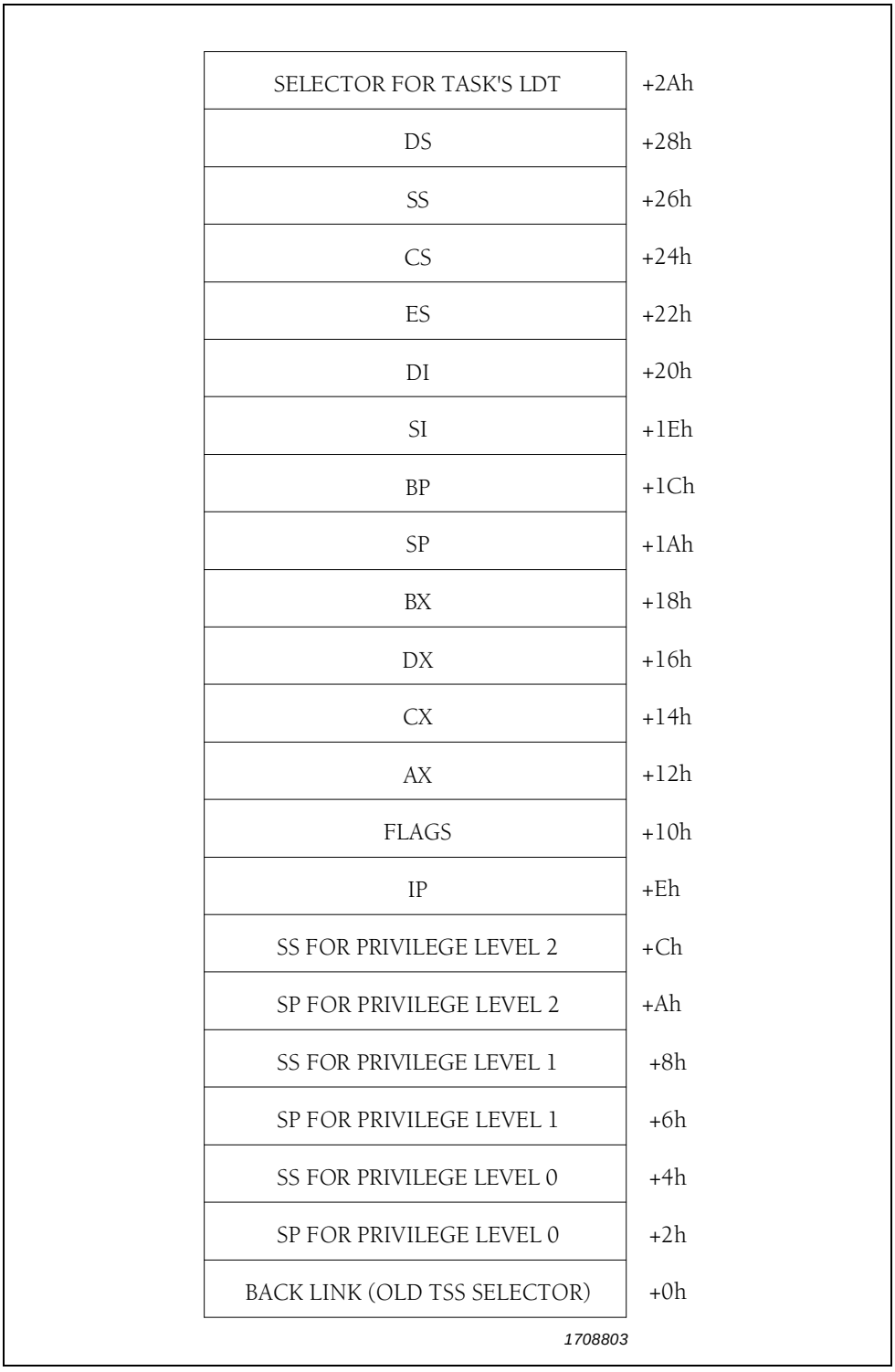


Figure 2-12. 16-Bit Task State Segment (TSS) Table

2.4.4 6x86MX Configuration Registers

The 6x86MX configuration registers are used to enable features in the 6x86MX CPU. These registers assign non-cached memory areas, set up SMM, provide CPU identification information and control various features such as cache write policy, and bus locking control. There are four groups of registers within the 6x86MX configuration register set:

- 7 Configuration Control Registers (CCRx)
- 8 Address Region Registers (ARRx)
- 8 Region Control Registers (RCRx)

Access to the configuration registers is achieved by writing the register index number for the configuration register to I/O port 22h. I/O port 23h is then used for data transfer.

Each I/O port 23h data transfer must be preceded by a valid I/O port 22h register index selection. Otherwise, the current 22h, and the second and later I/O port 23h operations communicate through the I/O port to produce external I/O cycles. All reads from I/O port 22h produce external I/O cycles. Accesses that hit within the on-chip configuration registers do not generate external I/O cycles.

After reset, configuration registers with indexes C0-CFh and FC-FFh are accessible. To prevent potential conflicts with other devices which may use ports 22 and 23h to access their registers, the remaining registers (indexes D0-FBh) are accessible only if the MAPEN(3-0) bits in CCR3 are set to 1h. See Figure 2-16 (Page 2-29) for more information on the MAPEN(3-0) bit locations.

If MAPEN[3-0] = 1h, any access to indexes in the range 00-FFh will not create external I/O bus cycles. Registers with indexes C0-CFh, FC- FFh are accessible regardless of the state of MAPEN[3-0]. If the register index number is outside the C0-CFh or FC-FFh ranges, and MAPEN[3-0] are set to 0h, external I/O bus cycles occur. Table 2-11 (Page 2-25) lists the MAPEN[3-0] values required to access each 6x86MX configuration register. All bits in the configuration registers are initialized to zero following reset unless specified otherwise.

2.4.4.1 Configuration Control Registers

(CCR0 - CCR6) control several functions, including non-cacheable memory, write-back regions, and SMM features. A list of the configuration registers is listed in Table 2-11 (Page 2-25). The configuration registers are described in greater detail in the following pages.

Table 2-11. 6x86MX CPU Configuration Registers

REGISTER NAME	ACRONYM	REGISTER INDEX	WIDTH (Bits)	MAPEN VALUE NEEDED FOR ACCESS
Configuration Control 0	CCR0	C0h	8	x
Configuration Control 1	CCR1	C1h	8	x
Configuration Control 2	CCR2	C2h	8	x
Configuration Control 3	CCR3	C3h	8	x
Configuration Control 4	CCR4	E8h	8	1
Configuration Control 5	CCR5	E9h	8	1
Configuration Control 6	CCR6	EAh	8	1
Address Region 0	ARR0	C4h - C6h	24	x
Address Region 1	ARR1	C7h - C9h	24	x
Address Region 2	ARR2	CAh - CCh	24	x
Address Region 3	ARR3	CDh - CFh	24	x
Address Region 4	ARR4	D0h - D2h	24	1
Address Region 5	ARR5	D3h - D5h	24	1
Address Region 6	ARR6	D6h - D8h	24	1
Address Region 7	ARR7	D9h - DBh	24	1
Region Control 0	RCR0	DCh	8	1
Region Control 1	RCR1	DDh	8	1
Region Control 2	RCR2	DEh	8	1
Region Control 3	RCR3	DFh	8	1
Region Control 4	RCR4	E0h	8	1
Region Control 5	RCR5	E1h	8	1
Region Control 6	RCR6	E2h	8	1
Region Control 7	RCR7	E3h	8	1

Note: x = Don't Care

7	6	5	4	3	2	1	0
Reserved	*Reserved*	*Reserved*	*Reserved*	*Reserved*	*Reserved*	NC1	*Reserved*

Figure 2-13. 6x86MX Configuration Control Register 0 (CCR0)

Table 2-12. CCR0 Bit Definitions

BIT POSITION	NAME	DESCRIPTION
1	NC1	No Cache 640 KByte - 1 MByte If = 1: Address region 640 KByte to 1 MByte is non-cacheable. If = 0: Address region 640 KByte to 1 MByte is cacheable.

Note: Bits 0, 2 through 7 are reserved.

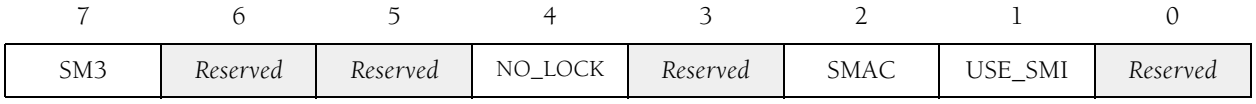


Figure 2-14. 6x86MX Configuration Control Register 1 (CCR1)

Table 2-13. CCR1 Bit Definitions

BIT POSITION	NAME	DESCRIPTION
7	SM3	SMM Address Space Address Region 3 If = 1: Address Region 3 is designated as SMM address space.
4	NO_LOCK	Negate LOCK# If = 1: All bus cycles are issued with LOCK# pin negated except page table accesses and interrupt acknowledge cycles. Interrupt acknowledge cycles are executed as locked cycles even though LOCK# is negated. With NO_LOCK set, previously noncacheable locked cycles are executed as unlocked cycles and therefore, may be cached. This results in higher performance. Refer to Region Control Registers for information on eliminating locked CPU bus cycles only in specific address regions.
2	SMAC	System Management Memory Access If = 1: Any access to addresses within the SMM address space, access system management memory instead of main memory. SMI# input is ignored. Used when initializing or testing SMM memory. If = 0: No effect on access.
1	USE_SMI	Enable SMM and SMIACT# Pins If = 1: SMI# and SMIACT# pins are enabled. If = 0: SMI# pin ignored and SMIACT# pin is driven inactive.

Note: Bits 0, 3, 5 and 6 are reserved.

7	6	5	4	3	2	1	0
USE_SUSP	*Reserved*	*Reserved*	WPR1	SUSP_HLT	LOCK_NW	SADS	*Reserved*

Figure 2-15. 6x86MX Configuration Control Register 2 (CCR2)

Table 2-14. CCR2 Bit Definitions

BIT POSITION	NAME	DESCRIPTION
7	USE_SUSP	Use Suspend Mode (Enable Suspend Pins) If = 1: SUSP# and SUSPA# pins are enabled. If = 0: SUSP# pin is ignored and SUSPA# pin floats.
4	WPR1	Write-Protect Region 1 If = 1: Designates any cacheable accesses in 640 KByte to 1 MByte address region are write protected.
3	SUSP_HLT	Suspend on Halt If = 1: Execution of the HLT instruction causes the CPU to enter low power suspend mode.
2	LOCK_NW	Lock NW If = 1: NW bit in CR0 becomes read only and the CPU ignores any writes to the NW bit. If = 0: NW bit in CR0 can be modified.
1	SADS	If = 1: CPU inserts an idle cycle following sampling of BRDY# and inserts an idle cycle prior to asserting ADS#

Note: Bits 0, 5 and 6 are reserved.

7	6	5	4	3	2	1	0
MAPEN3	MAPEN2	MAPEN1	MAPEN0	*Reserved*	LINBRST	NMI_EN	SMI_LOCK

Figure 2-16. 6x86MX Configuration Control Register 3 (CCR3)

Table 2-15. CCR3 Bit Definitions

BIT POSITION	NAME	DESCRIPTION
7 - 4	MAPEN(3-0)	MAP Enable If = 1h: All configuration registers are accessible. If = 0h: Only configuration registers with indexes C0-CFh, FEh and FFh are accessible.
2	LINBRST	If = 1: Use linear address sequence during burst cycles. If = 0: Use "1 + 4" address sequence during burst cycles. The "1 + 4" address sequence is compatible with Pentium's burst address sequence.
1	NMI_EN	NMI Enable If = 1: NMI interrupt is recognized while servicing an SMI interrupt. NMI_EN should be set only while in SMM, after the appropriate SMI interrupt service routine has been setup.
0	SMI_LOCK	SMI Lock If = 1: The following SMM configuration bits can only be modified while in an SMI service routine: CCR1: USE_SMI, SMAC, SM3 CCR3: NMI_EN CCR6: N, SMM_MODE ARR3: Starting address and block size. Once set, the features locked by SMI_LOCK cannot be unlocked until the RESET pin is asserted.

Note: Bit 3 is reserved.

7	6	5	4	3	2	1	0
CPUID	*Reserved*	*Reserved*	*Reserved*	*Reserved*	IORT2	IORT1	IORT

Figure 2-17. 6x86MX Configuration Control Register 4 (CCR4)

Table 2-16. CCR4 Bit Definitions

BIT POSITION	NAME	DESCRIPTION
7	CPUID	Enable CPUID instruction. If = 1: the ID bit in the EFLAGS register can be modified and execution of the CPUID instruction occurs as documented in section 6.3. If = 0: the ID bit in the EFLAGS register can not be modified and execution of the CPUID instruction causes an invalid opcode exception.
2 - 0	IORT(2-0)	I/O Recovery Time Specifies the minimum number of bus clocks between I/O accesses: 0h = 1 clock delay 1h = 2 clock delay 2h = 4 clock delay 3h = 8 clock delay 4h = 16 clock delay 5h = 32 clock delay (default value after RESET) 6h = 64 clock delay 7h = no delay

Note: Bits 3 - 6 are reserved.

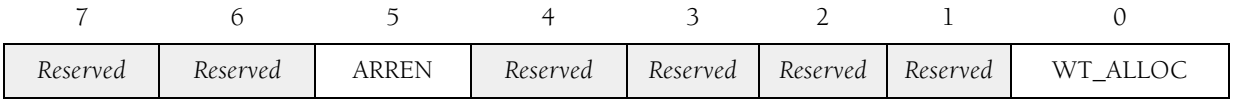


Figure 2-18. 6x86MX Configuration Control Register 5 (CCR5)

Table 2-17. CCR5 Bit Definitions

BIT POSITION	NAME	DESCRIPTION
5	ARREN	Enable ARR Registers If = 1: Enables all ARR registers. If = 0: Disables the ARR registers. If SM3 is set, ARR3 is enabled regardless of the setting of ARREN.
0	WT_ALLOC	Write-Through Allocate If = 1: New cache lines are allocated for read and write misses. If = 0: New cache lines are allocated only for read misses.

Note: Bits 1 - 3 and 6 - 7 are reserved.

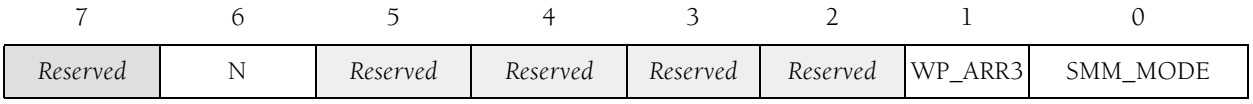


Figure 2-19. 6x86MX Configuration Control Register 6 (CCR6)

Table 2-18. CCR6 Bit Definitions

BIT POSITION	NAME	DESCRIPTION
6	N	Nested SMI Enable bit: If operating in Cyrix enhanced SMM mode and: If = 1: Enables nesting of SMI's If = 0: Disable nesting of SMI's. This bit is automatically CLEARED upon entry to every SMM routine and is SET upon every RSM. Therefore enabling/disabling of nested SMI can only be done while operating in SMM mode.
1	WP_ARR3	If = 1: Memory region defined by ARR3 is write protected when operating outside of SMM mode. If = 0: Disable write protection for memory region defined by ARR3. Reset State = 0.
0	SMM_MODE	If = 1: Enables Cyrix Enhanced SMM mode. If = 0: Disables Cyrix Enhanced SMM mode.

Note: Bit 1 is reserved.

2.4.4.2 Address Region Registers

The Address Region Registers (ARR0 - ARR7) (Figure 2-20) are used to specify the location and size for the eight address regions.

Attributes for each address region are specified in the Region Control Registers (RCR0-RCR7). ARR7 and RCR7 are used to define system main memory and differ from ARR0-6 and RCR0-6.

With non-cacheable regions defined on-chip, the 6x86MX CPU delivers optimum performance by using advanced techniques to eliminate data dependencies and resource conflicts in its execution pipelines. If KEN# is active for accesses to regions defined as non-cacheable by the RCRs, the region is not cached. The RCRs take precedence in this case.

A register index, shown in Table 2-19 (Page 2-34) is used to select one of three bytes in each ARR.

The starting address of the ARR address region, selected by the START ADDRESS field, must be on a block size boundary. For example, a 128 KByte block is allowed to have a starting address of 0 KBytes, 128 KBytes, 256 KBytes, and so on.

The SIZE field bit definition is listed in (Page 2-34). If the SIZE field is zero, the address region is of zero size and thus disabled.

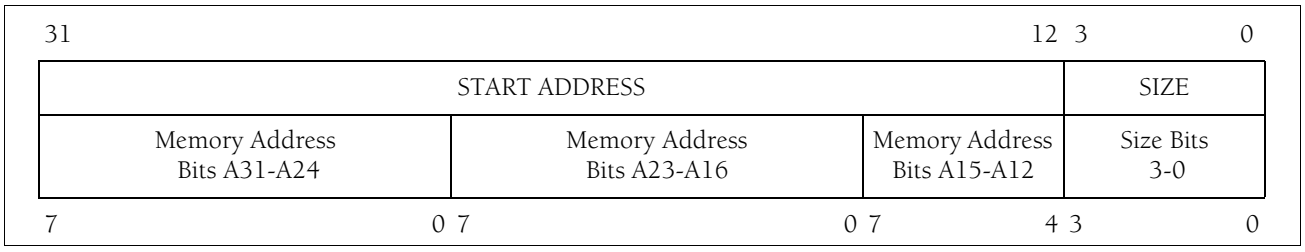


Figure 2-20. Address Region Registers (ARR0 - ARR7)

Table 2-19. ARR0 - ARR7 Register Index Assignments

ARR Register	Memory Address (A31 - A24)	Memory Address (A23 - A16)	Memory Address (A15 - A12)	Address Region Size (3 - 0)
ARR0	C4h	C5h	C6h	C6h
ARR1	C7h	C8h	C9h	C9h
ARR2	CAh	CBh	CCh	CCh
ARR3	CDh	CEh	CFh	CFh
ARR4	D0h	D1h	D2h	D2h
ARR5	D3h	D4h	D5h	D5h
ARR6	D6h	D7h	D8h	D8h
ARR7	D9h	DAh	DBh	DBh

Table 2-20. Bit Definitions for SIZE Field

SIZE (3-0)	BLOCK SIZE ARR0-6	BLOCK SIZE ARR7
0h	Disabled	Disabled
1h	4 KBytes	256 KBytes
2h	8 KBytes	512 KBytes
3h	16 KBytes	1 MBytes
4h	32 KBytes	2 MBytes
5h	64 KBytes	4 MBytes
6h	128 KBytes	8 MBytes
7h	256 KBytes	16 MBytes

SIZE (3-0)	BLOCK SIZE ARR0-6	BLOCK SIZE ARR7
8h	512 KBytes	32 MBytes
9h	1 MBytes	64 MBytes
Ah	2 MBytes	128 MBytes
Bh	4 MBytes	256 MBytes
Ch	8 MBytes	512 MBytes
Dh	16 MBytes	1 GBytes
Eh	32 MBytes	2 GBytes
Fh	4 GBytes	4 GBytes

2.4.4.3 Region Control Registers

The Region Control Registers (RCR0 - RCR7) specify the attributes associated with the ARRx address regions. The bit definitions for the region control registers are shown in Figure 2-21 (Page 2-36) and in Table 2-21 (Page 2-36). Cacheability, weak locking, write gathering, and cache write through policies can be activated or deactivated using the attribute bits.

If an address is accessed that is not in a memory region defined by the ARRx registers, the following conditions will apply:

- If the memory address is cached, write-back is enabled if WB/WT# is returned high.
- Writes are not gathered
- Strong locking takes place
- The memory access is cached, if KEN# is returned asserted.

Overlapping Conditions Defined. If two regions specified by ARRx registers overlap and conflicting attributes are specified, the following attributes take precedence:

- Write-back is disabled
- Writes are not gathered
- Strong locking takes place
- The overlapping regions are non-cacheable.

7	6	5	4	3	2	1	0
Reserved	INV_RGN	*Reserved*	WT	WG	WL	*Reserved*	CD

Figure 2-21. Region Control Registers (RCR0-RCR7)

Table 2-21. RCR0-RCR7 Bit Definitions

BIT POSITION	NAME	DESCRIPTION
6	INV_RGN	**Inverted Region**. If =1, applies controls specified in RCRx to all memory addresses outside the region specified in corresponding ARR. Applicable to RCR0-RCR6 only.
4	WT	**Write-Through.** If =1, defines the address region as write-through instead of write-back.
3	WG	**Write Gathering.** If =1, enables write gathering for the associated address region.
2	WL	**Weak Locking.** If =1, enables weak locking for that address region.
0	CD	**Cache Disable.** If =1, defines the address region as non-cacheable.

Note: Bits 1, 5 and 7 are reserved.

Inverted Region (INV_RGN). Setting INV-RGN applies the controls in RCRx to all the memory addresses outside the specified address region ARRx. This bit effects RCR0-RCR6 and not RCR7.

Write Through (WT). Setting WT defines the address region as write-through instead of write-back, assuming the region is cacheable. Regions where system ROM are loaded (shadowed or not) should be defined as write-through. This bit works in conjunction with the CR0_NW and PWT bits and the WB/WT# pin to determine write-through or write-back cacheability.

Write Gathering (WG). Setting WG enables write gathering for the associated address region. Write gathering allows multiple byte, word, or Dword sequential address writes to accumulate in the on-chip write buffer. As instructions are executed, the results are placed in a series of output buffers. These buffers are gathered into the final output buffer.

When access is made to a non-sequential memory location or when the 8-byte buffer becomes full, the contents of the buffer are written on the external 64-bit data bus. Performance is enhanced by avoiding as many as seven memory write cycles.

WG should not be used on memory regions that are sensitive to write cycle gathering. WG can be enabled for both cacheable and non-cacheable regions.

Weak Locking (WL). Setting WL enables weak locking for the associated address region. During weak locking all bus cycles are issued with the LOCK# pin negated (except when page table access occur and during interrupt acknowledge cycles.)

Interrupt acknowledge cycles are executed as locked cycles even though LOCK# is negated. With WL set previously non-cacheable locked cycles are executed as unlocked cycles and therefore, may be cached, resulting in higher CPU performance.

Note that the NO_LOCK bit globally performs the same function that the WL bit performs on a single address region. The NO_LOCK bit of CCR1 enables weak locking for the entire address space. The WL bit allows weak locking only for specific address regions. WL is independent of the cacheability of the address region.

Cache Disable (CD). Cache Disable - If set, defines the address region as non-cacheable. This bit works in conjunction with the CR0_CD and PCD bits and the KEN# pin to determine line cacheability. Whenever possible, the ARR/RCR combination should be used to define non-cacheable regions rather than using external address decoding and driving the KEN# pin as the M II can better utilize its advanced techniques for eliminating data dependencies and resource conflicts with non-cacheable regions defined on-chip.

2.5 Model Specific Registers

The CPU contains several Model Specific Registers (MSRs) that provide time stamp, performance monitoring and counter event functions. Access to a specific MSR through an index value in the ECX register as shown in Table 2-22 below.

Table 2-22. Model Specific Registers

REGISTER DESCRIPTION	ECX VALUE
Test Data	3h
Test Address	4h
Command/Status	5h
Time Stamp Counter (TSC)	10h
Counter Event Selection and Control Register	11h
Performance Counter #0	12h
Performance Counter #1	13h

The MSR registers can be read using the RDMSR instruction, opcode 0F32h. During a MSR register read, the contents of the particular MSR register, specified by the ECX register, is loaded into the EDX:EAX registers.

The MSR registers can be written using the WRMSR instruction, opcode 0F30h. During a MSR register, write the contents of EDX:EAX are loaded into the MSR register specified in the ECX register.

The RDMSR and WRMSR instructions are privileged instructions and are also used to setup scratch pad lock (Page 2-61)..

2.6 Time Stamp Counter

The Time Stamp Counter (TSC) Register MSR(10) is a 64-bit counter that counts the internal CPU clock cycles since the last reset. The TSC uses a continous CPU core clock and will continue to count clock cycles even when the 6x86MX is suspend mode or shutdown.

The TSC can be accessed using the RDMSR and WRMSR instructions. In addition, the TSC can be read using the RDTSC instruction, opcode 0F31h. The RDTSC instruction loads the contents of the TSC into EDX:EAX. The use of the RDTSC instruction is restricted by the Time Stamp Disable, (TSD) flag in CR4. When the TSD flag is 0, the RDTSC instruction can be executed at any privilege level. When the TSD flag is 1, the RDTSC instruction can only be executed at privilege level 0.

2.7 Performance Monitoring

Performance monitoring allows counting of over a hundred different event occurrences and durations. Two 48-bit counters are used: Performance Monitor Counter 0 and Performance Monitor Counter 1. These two performance monitor counters are controlled by the Counter Event Control Register MSR(11). The performance monitor counters use a continuous CPU core clock and will continue to count clock cycles even when the 6x86MX CPU is in suspend mode or shutdown.

2.8 Performance Monitoring Counters 1 and 2

The 48-bit Performance Monitoring Counters (PMC) Registers MSR(12), MSR(13) count events as specified by the counter event control register.

The PMCs can be accessed by the RDMSR and WRMSR instructions. In addition, the PMCs can be read by the RDPMC instruction, opcode 0F33h. The RDPMC instruction loads the contents of the PMC register specified in the ECX register into EDX:EAX. The use of RDPMC instructions is restricted by the Performance Monitoring Counter Enable, (PCE) flag in C4.

When the PCE flag is set to 1, the RDPMC instruction can be executed at any privilege level. When the PCE flag is 0, the RDPMC instruction can only be executed at privilege level 0.

2.8.1 Counter Event Control Register

Register MSR (11) controls the two internal counters, #0 and #1. The events to be counted have been chosen based on the micro-architecture of the 6x86MX processor. The control register for the two event counters is described in Figure 2-21 (Page 2-36) and Table 2-23 (Page 2-40).

2.8.1.1 PM Pin Control

The Counter Event Control register MSR(11) contains PM control fields that define the PM0 and PM1 pins as counter overflow indicators or counter event indicators. When defined as event counters, the PM pins indicate that one or more events occurred during a particular clock cycle and do not count the actual events.

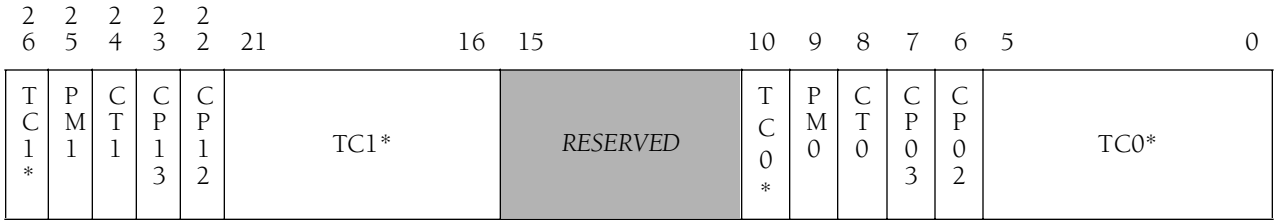
When defined as overflow indicators, the event counters can be preset with a value less the 2^{48}-1 and allowed to increment as events occur. When the counter overflows the PM pin becomes asserted.

2.8.1.2 Counter Type Control

The Counter Type bit determines whether the counter will count clocks or events. When counting clocks the counter operates as a timer.

2.8.1.3 CPL Control

The Current Privilege Level (CPL) can be used to determine if the counters are enabled. The CP02 bit in the MSR (11) register enables counting when the CPL is less than three, and the CP03 bit enables counting when CPL is equal to three. If both bits are set, counting is not dependent on the CPL level; if neither bit is set, counting is disabled.



*Note: Split Fields

Figure 2-22. Counter Event Control Register

Table 2-23. Counter Event Control Register Bit Definitions

BIT POSITION	NAME	DESCRIPTION
25	PM1	Define External PM1 Pin If = 1: PM1 pin indicates counter overflows If = 0: PM1 pin indicates counter events
24	CT1	Counter #1 Counter Type If = 1: Count clock cycles If = 0: Count events (reset state).
23	CP13	Counter #1 CPL 3 Enable If = 1: Enable counting when CPL=3. If = 0: Disable counting when CPL=3. (reset state)
22	CP12	Counter #1 CPL Less Than 3 Enable If = 1: Enable counting when CPL < 3. If = 0: Disable counting when CPL < 3. (reset state)
26, 21 - 16	TC1(5-0)	Counter #1 Event Type Reset state = 0
9	PM0	Define External PM0 Pin If = 1: PM0 pin indicates counter overflows If = 0: PM0 pin indicates counter events
8	CT0	Counter #0 Counter Type If = 1: Count clock cycles If = 0: Count events (reset state).
7	CP03	Counter #0 CPL 3 Enable If = 1: Enable counting when CPL=3. If = 0: Disable counting when CPL=3. (reset state)
6	CP02	Counter #0 CPL Less Than 3 Enable If = 1: Enable counting when CPL < 3. If = 0: Disable counting when CPL < 3. (reset state)
10, 5 - 0	TC0(5-0)	Counter #0 Event Type Reset state = 0

Note: Bits 10 - 15 are reserved.

2.8.2 Event Type and Description

The events that can be counted by the performance monitoring counters are listed in Table 2-24. Each of the 127 event types is assigned an event number.

A particular event number to be counted is placed in one of the MSR(11) Event Type fields. There is a separate field for counter #0 and #1.

The events are divided into two groups. The occurrence type events and duration type events. The occurrence type events, such as hardware interrupts, are counted as single events. The duration type events such as "clock while bus cycles are in progress" count the number of clock cycles that occur during the event.

During occurrence type events, the PM pins are configured to indicate the counter has incremented The PM pins will then assert every time the counter increments in regards to an occurrence event. Under the same PM control, for a duration event the PM pin will stay asserted for the duration of the event.

Table 2-24. Event Type Register

NUMBER	COUNTER 0	COUNTER 1	DESCRIPTION	TYPE
00h	yes	yes	Data Reads	Occurrence
01h	yes	yes	Data Writes	Occurrence
02h	yes	yes	Data TLB Misses	Occurrence
03h	yes	yes	Cache Misses: Data Reads	Occurrence
04h	yes	yes	Cache Misses: Data Writes	Occurrence
05h	yes	yes	Data Writes that hit on Modified or Exclusive Liens	Occurence
06h	yes	yes	Data Cache Lines Written Back	Occurrence
07h	yes	yes	External Inquiries	Occurrence
08h	yes	yes	External Inquires that hit	Occurrence
09h	yes	yes	Memory Accesses in both pipes	Occurrence
0Ah	yes	yes	Cache Bank conflicts	Occurrence
0Bh	yes	yes	Misaligned data references	Occurrence
0Ch	yes	yes	Instruction Fetch Requests	Occurrence
0Dh	yes	yes	L2 TLB Code Misses	Occurrence
0Eh	yes	yes	Cache Misses: Instruction Fetch	Occurrence
0Fh	yes	yes	Any Segment Register Load	Occurrence
10h	yes	yes	Reserved	Occurrence
11h	yes	yes	Reserved	Occurrence
12h	yes	yes	Any Branch	Occurrence

Table 2-24. Event Type Register (Continued)

NUMBER	COUNTER 0	COUNTER 1	DESCRIPTION	TYPE
13h	yes	yes	BTB hits	Occurrence
14h	yes	yes	Taken Branches or BTB hits	Occurrence
15h	yes	yes	Pipeline Flushes	Occurrence
16h	yes	yes	Instructions executed in both pipes	Occurrence
17h	yes	yes	Instructions executed in Y pipe	Occurrence
18h	yes	yes	Clocks while bus cycles are in progress	Duration
19h	yes	yes	Pipe Stalled by full write buffers	Duration
1Ah	yes	yes	Pipe Stalled by waiting on data memory reads	Duration
1Bh	yes	yes	Pipe Stalled by writes to not-Modified or not-Exclusive cache lines.	Duration
1Ch	yes	yes	Locked Bus Cycles	Occurrence
1Dh	yes	yes	I/O Cycles	Occurrence
1Eh	yes	yes	Non-cacheable Memory Requests	Occurrence
1Fh	yes	yes	Pipe Stalled by Address Generation Interlock	Duration
20h	yes	yes	Reserved	
21h	yes	yes	Reserved	
22h	yes	yes	Floating Point Operations	Occurrence
23h	yes	yes	Breakpoint Matches on DR0 register	Occurrence
24h	yes	yes	Breakpoint Matches on DR1 register	Occurrence
25h	yes	yes	Breakpoint Matches on DR2 register	Occurrence
26h	yes	yes	Breakpoint Matches on DR3 register	Occurrence
27h	yes	yes	Hardware Interrupts	Occurrence
28h	yes	yes	Data Reads or Data Writes	Occurrence
29h	yes	yes	Data Read Misses or Data Write Misses	Occurrence
2Bh	yes	no	MMX Instruction Executed in X pipe	Occurrence
2Bh	no	yes	MMX Instruction Executed in Y pipe	Occurrence
2Dh	yes	no	EMMS Instruction Executed	Occurrence
2Dh	no	yes	Transition Between MMX Instruction and FP Instructions	Occurrence
2Eh	no	yes	Reserved	
2Fh	yes	no	Saturating MMX Instructions Executed	Occurrence
2Fh	no	yes	Saturations Performed	Occurrence
30h	yes	no	Reserved	
31h	yes	no	MMX Instruction Data Reads	Occurrence
32h	yes	no	Reserved	
32h	no	yes	Taken Branches	Occurrence
33h	no	yes	Reserved	
34h	yes	no	Reserved	
34h	no	yes	Reserved	
35h	yes	no	Reserved	

Table 2-24. Event Type Register (Continued)

NUMBER	COUNTER 0	COUNTER 1	DESCRIPTION	TYPE
35h	no	yes	Reserved	
36	yes	no	Reserved	
36	no	yes	Reserved	
37	yes	no	Returns Predicted Incorrectly	Occurrence
37	no	yes	Return Predicted (Correctly and Incorrectly)	Occurrence
38	yes	no	MMX Instruction Multiply Unit Interlock	Duration
38	no	yes	MODV/MOVQ Store Stall Due to Previous Operation	Duration
39	yes	no	Returns	Occurrence
39	no	yes	RSB Overflows	Occurrence
3A	yes	no	BTB False Entries	Occurrence
3A	no	yes	BTB Miss Prediction on a Not-Taken Back	Occurrence
3B	yes	no	Number of Clock Stalled Due to Full Write Buffers While Executing	Duration
3B	no	yes	Stall on MMX Instruction Write to E or M Line	Duration
3C - 3Fh	yes	yes	Reserved	Duration
40h	yes	yes	L2 TLB Misses (Code or Data)	Occurrence
41h	yes	yes	L1 TLB Data Miss	Occurrence
42h	yes	yes	L1 TLB Code Miss	Occurrence
43h	yes	yes	L1 TLB Miss (Code or Data)	Occurrence
44h	yes	yes	TLB Flushes	Occurrence
45h	yes	yes	TLB Page Invalidates	Occurrence
46h	yes	yes	TLB Page Invalidates that hit	Occurrence
47h	yes	yes	Reserved	
48h	yes	yes	Instructions Decoded	Occurrence
49h	yes	yes	Reserved	

2.9 Debug Registers

Six debug registers (DR0-DR3, DR6 and DR7), shown in Figure 2-23, support debugging on the 6x86MX CPU. The bit definitions for the debug registers are listed in Table 2-25 (Page 2-45).

Memory addresses loaded in the debug registers, referred to as "breakpoints", generate a debug exception when a memory access of the specified type occurs to the specified address. A data breakpoint can be specified for a particular kind of memory access such as a read or a write. Code breakpoints can also be set allowing debug exceptions to occur whenever a given code access (execution) occurs.

The size of the debug target can be set to 1, 2, or 4 bytes. The debug registers are accessed via MOV instructions which can be executed only at privilege level 0.

The Debug Address Registers (DR0-DR3) each contain the linear address for one of four possible breakpoints. Each breakpoint is further specified by bits in the Debug Control Register (DR7). For each breakpoint address in DR0-DR3, there are corresponding fields L, R/W, and LEN in DR7 that specify the type of memory access associated with the breakpoint.

The R/W field can be used to specify instruction execution as well as data access breakpoints. Instruction execution breakpoints are always taken before execution of the instruction that matches the breakpoint.

The Debug Status Register (DR6) reflects conditions that were in effect at the time the debug exception occurred. The contents of the DR6 register are not automatically cleared by the processor after a debug exception occurs and, therefore, should be cleared by software at the appropriate time.

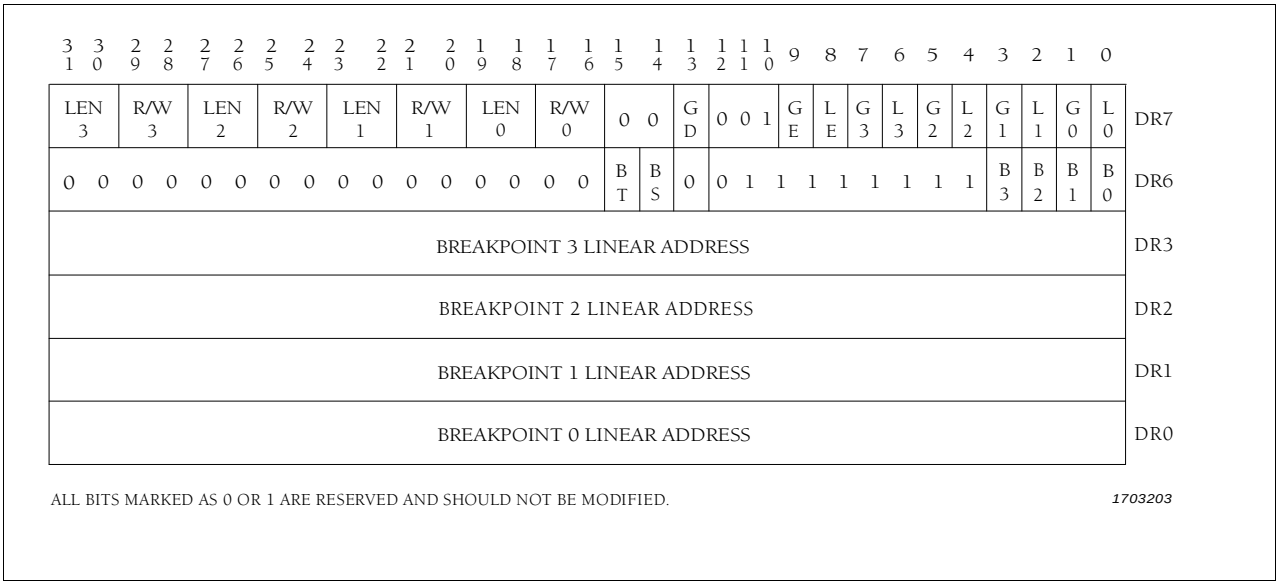


Figure 2-23. Debug Registers

Code execution breakpoints may also be generated by placing the breakpoint instruction (INT 3) at the location where control is to be regained. Additionally, the single-step feature may be enabled by setting the TF flag in the EFLAGS register. This causes the processor to perform a debug exception after the execution of every instruction.

Table 2-25. DR6 and DR7 Debug Register Field Definitions

REGISTER	FIELD	NUMBER OF BITS	DESCRIPTION
DR6	Bi	1	Bi is set by the processor if the conditions described by DRi, R/Wi, and LENi occurred when the debug exception occurred, even if the breakpoint is not enabled via the Gi or Li bits.
	BT	1	BT is set by the processor before entering the debug handler if a task switch has occurred to a task with the T bit in the TSS set.
	BS	1	BS is set by the processor if the debug exception was triggered by the single-step execution mode (TF flag in EFLAGS set).
DR7	R/Wi	2	Specifies type of break for the linear address in DR0, DR1, DR3, DR4: 00 - Break on instruction execution only 01 - Break on data writes only 10 - Not used 11 - Break on data reads or writes.
	LENi	2	Specifies length of the linear address in DR0, DR1, DR3, DR4: 00 - One byte length 01 - Two byte length 10 - Not used 11 - Four byte length.
	Gi	1	If set to a 1, breakpoint in DRi is globally enabled for all tasks and is not cleared by the processor as the result of a task switch.
	Li	1	If set to a 1, breakpoint in DRi is locally enabled for the current task and is cleared by the processor as the result of a task switch.
	GD	1	Global disable of debug register access. GD bit is cleared whenever a debug exception occurs.

2.10 Test Registers

The test registers can be used to test the on-chip unified cache and to test the main TLB.

Test registers TR3, TR4, and TR5 are used to test the unified cache. Use of these registers is described with the memory caches later in this chapter in section 2.13.1.1 on page 2-58.

Test registers TR6 and TR7 are used to test the TLB. Use of these test registers is described in section 2.12.4.1 on page 2-54.

2.11 Address Space

The 6x86MX CPU can directly address 64 KBytes of I/O space and 4 GBytes of physical memory (Figure 2-24).

Memory Address Space. Access can be made to memory addresses between 0000 0000h and FFFF FFFFh. This 4 GByte memory space can be accessed using byte, word (16 bits), or doubleword (32 bits) format. Words and doublewords are stored in consecutive memory bytes with the low-order byte located in the lowest address. The physical address of a word or doubleword is the byte address of the low-order byte.

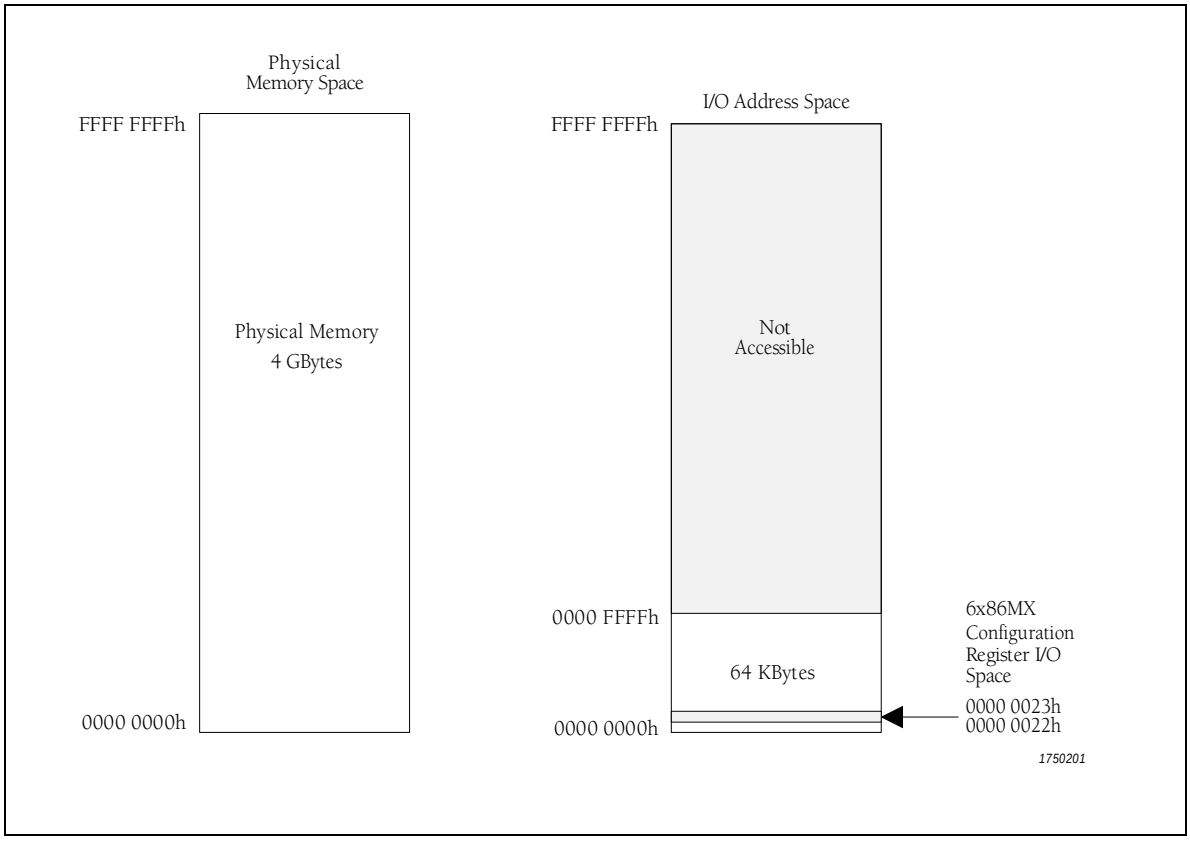


Figure 2-24. Memory and I/O Address Spaces

I/O Address Space

The 6x86MX I/O address space is accessed using IN and OUT instructions to addresses referred to as "ports". The accessible I/O address space size is 64 KBytes and can be accessed through 8-bit, 16-bit or 32-bit ports. The execution of any IN or OUT instruction causes the M/IO# pin to be driven low, thereby selecting the I/O space instead of memory space.

The accessible I/O address space ranges between locations 0000 0000h and 0000 FFFFh (64 KBytes). The I/O locations (ports) 22h and 23h can be used to access the 6x86MX configuration registers.

2.12 Memory Addressing Methods

With the 6x86MX CPU, memory can be addressed using nine different addressing modes (Table 2-26, Page 2-49). These addressing modes are used to calculate an offset address often referred to as an effective address. Depending on the operating mode of the CPU, the offset is then combined using memory management mechanisms to create a physical address that actually addresses the physical memory devices.

Memory management mechanisms on the 6x86MX CPU consist of segmentation and paging. Segmentation allows each program to use several independent, protected address spaces. Paging supports a memory subsystem that simulates a large address space using a small amount of RAM and disk storage for physical memory. Either or both of these mechanisms can be used for management of the 6x86MX CPU memory address space.

2.12.1 Offset Mechanism

The offset mechanism computes an offset (effective) address by adding together one or more of three values: a base, an index and a displacement. When present, the base is the value of one of the eight 32-bit general registers. The index if present, like the base, is a value that is in one of the eight 32-bit general purpose registers (not including the ESP register). The index differs from the base in that the index is first multiplied by a scale factor of 1, 2, 4 or 8 before the summation is made. The third component added to the memory address calculation is the displacement. The displacement is a value of up to 32-bits in length supplied as part of the instruction. Figure 2-25 illustrates the calculation of the offset address.

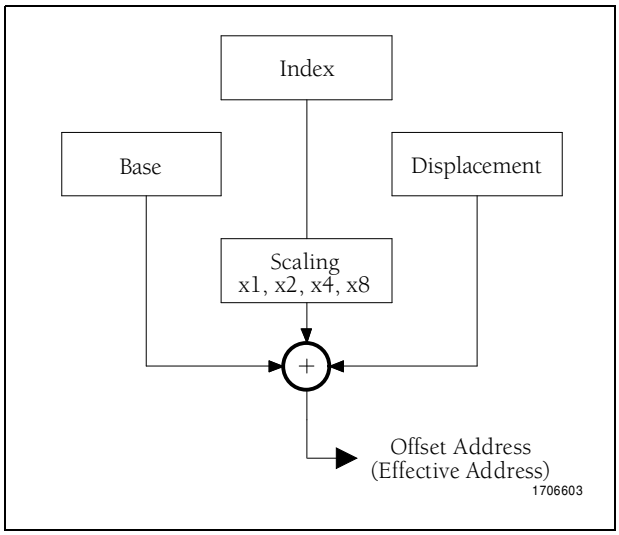


Figure 2-25. Offset Address Calculation

Nine valid combinations of the base, index, scale factor and displacement can be used with the 6x86MX CPU instruction set. These combinations are listed in Table 2-26. The base and index both refer to contents of a register as indicated by [Base] and [Index].

Table 2-26. Memory Addressing Modes

ADDRESSING MODE	BASE	INDEX	SCALE FACTOR (SF)	DISPLACEMENT (DP)	OFFSET ADDRESS (OA) CALCULATION
Direct				x	OA = DP
Register Indirect	x				OA = [BASE]
Based	x			x	OA = [BASE] + DP
Index		x		x	OA = [INDEX] + DP
Scaled Index		x	x	x	OA = ([INDEX] * SF) + DP
Based Index	x	x			OA = [BASE] + [INDEX]
Based Scaled Index	x	x	x		OA = [BASE] + ([INDEX] * SF)
Based Index with Displacement	x	x		x	OA = [BASE] + [INDEX] + DP
Based Scaled Index with Displacement	x	x	x	x	OA = [BASE] + ([INDEX] * SF) + DP

2.12.2 Memory Addressing

Real Mode Memory Addressing

In real mode operation, the 6x86MX CPU only addresses the lowest 1 MByte of memory. To calculate a physical memory address, the 16-bit segment base address located in the selected segment register is multiplied by 16 and then the 16-bit offset address is added. The resulting 20-bit address is then extended. Three hexadecimal zeros are added as upper address bits to create the 32-bit physical address. Figure 2-26 illustrates the real mode address calculation.

The addition of the base address and the offset address may result in a carry. Therefore, the resulting address may actually contain up to 21 significant address bits that can address memory in the first 64 KBytes above 1 MByte.

Protected Mode Memory Addressing

In protected mode three mechanisms calculate a physical memory address (Figure 2-27, Page 2-51).

- **Offset Mechanism** that produces the offset or effective address as in real mode.
- **Selector Mechanism** that produces the base address.
- Optional **Paging Mechanism** that translates a linear address to the physical memory address.

The offset and base address are added together to produce the linear address. If paging is not enabled, the linear address is used as the physical memory address. If paging is enabled, the paging mechanism is used to translate the linear address into the physical address. The offset mechanism is described earlier in this section and applies to both real and protected mode. The selector and paging mechanisms are described in the following paragraphs.

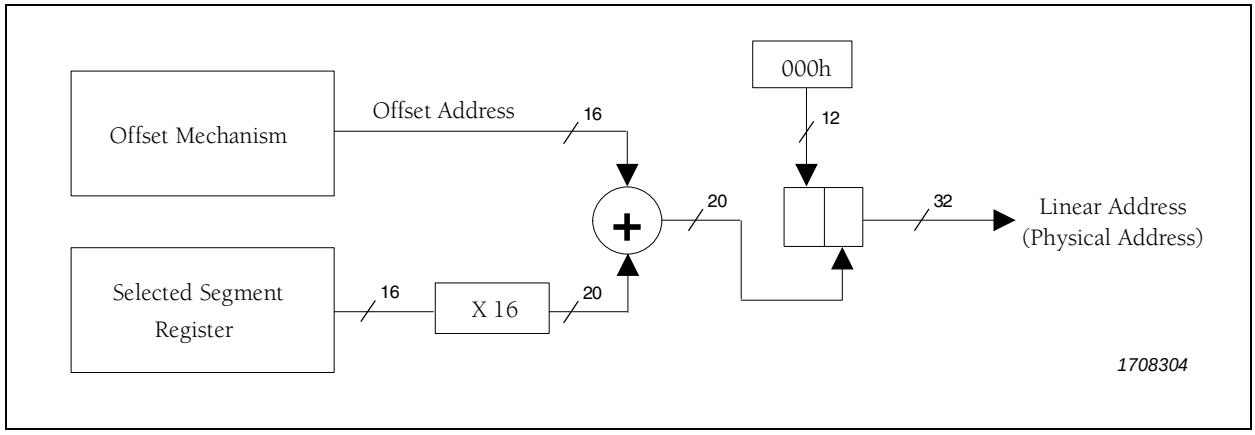


Figure 2-26. Real Mode Address Calculation

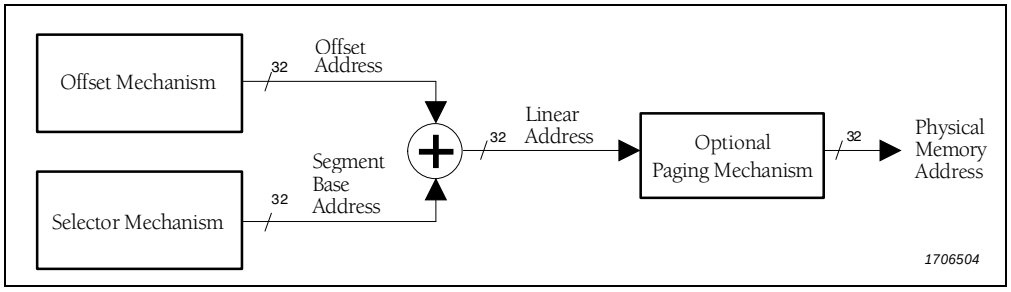


Figure 2-27. Protected Mode Address Calculation

2.12.3 Selector Mechanism

Using segmentation, memory is divided into an arbitrary number of segments, each containing usually much less than the 2^{32} byte (4 GByte) maximum.

The six segment registers (CS, DS, SS, ES, FS and GS) each contain a 16-bit selector that is used when the register is loaded to locate a segment descriptor in either the global descriptor table (GDT) or the local descriptor table (LDT). The segment descriptor defines the base address, limit, and attributes of the selected segment and is cached on the 6x86MX CPU as a result of loading the selector. The cached descriptor contents are not visible to the programmer. When a memory reference occurs in protected mode, the linear address is generated by adding the segment base address in the hidden portion of the segment register to the offset address. If paging is not enabled, this linear address is used as the physical memory address. Figure 2-28 illustrates the operation of the selector mechanism.

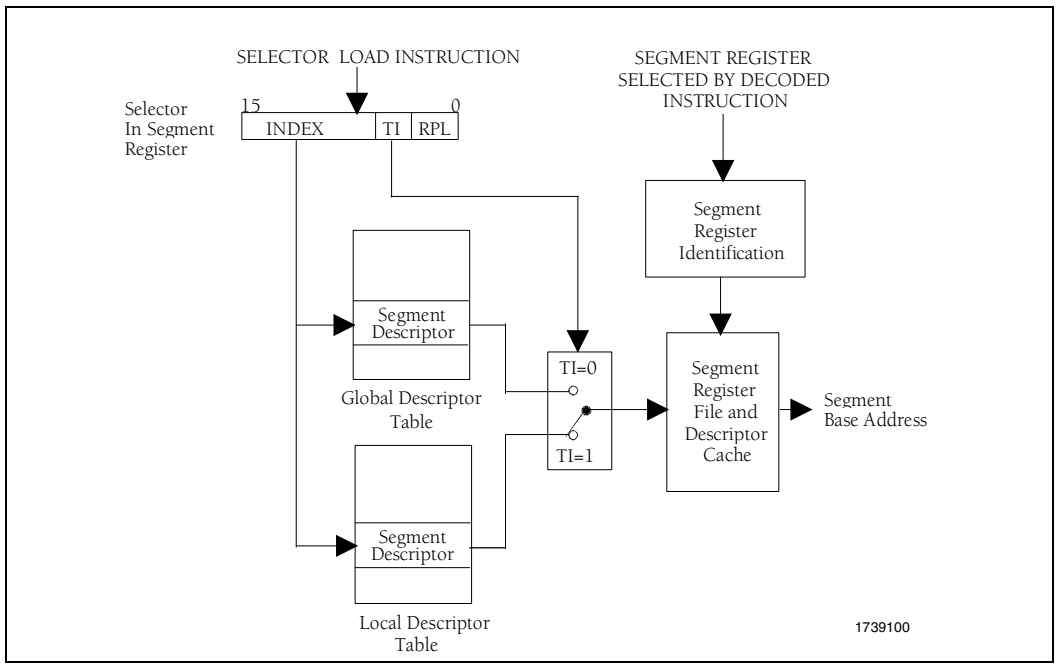


Figure 2-28. Selector Mechanism

2.12.4 Paging Mechanism

The paging mechanism translates linear addresses to their corresponding physical addresses. The page size is always 4 KBytes. Paging is activated when the PG and the PE bits within the CR0 register are set.

The paging mechanism translates the 20 most significant bits of a linear address to a physical address. The linear address is divided into three fields DTI, PTI, PFO (Figure 2-29, Page 2-53). These fields respectively select:

- an entry in the directory table,
- an entry in the page table selected by the directory table
- the offset in the physical page selected by the page table

The directory table and all the page tables can be considered as pages as they are 4 KBytes in size and are aligned on 4 KByte boundaries. Each entry in these tables is 32 bits in length. The fields within the entries are detailed in Figure 2-30 (Page 2-53) and Table 2-27 (Page 2-54).

A single page directory table can address up to 4 GBytes of virtual memory (1,024 page tables—each table can select 1,024 pages and each page contains 4 KBytes).

Translation Lookaside Buffer (TLB) is made up of two caches (Figure 2-29, Page 2-53).

- the L1 TLB caches page tables entries
- the L2 TLB stores PTEs that have been evicted from the L1 TLB

The L1 TLB is a 16-entry direct-mapped dual ported cache. The L2 TLB is a 384 entry, 6-way, dual ported cache.

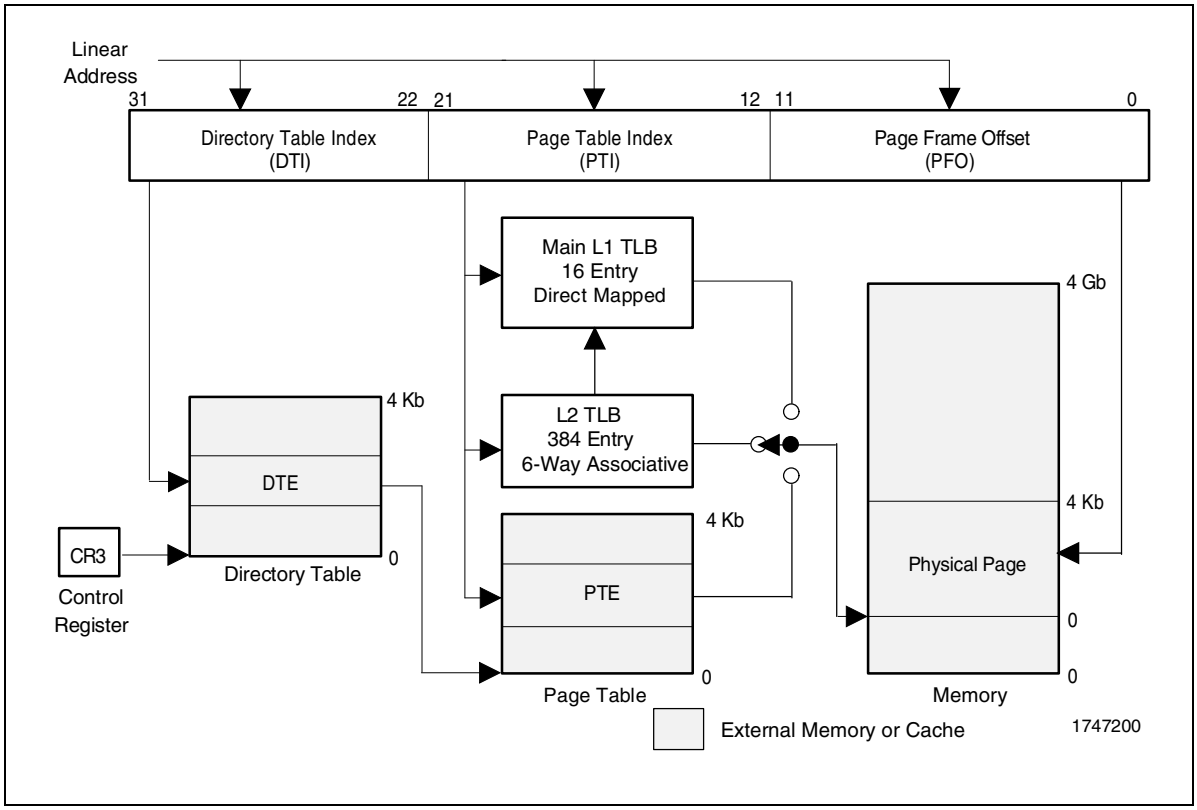


Figure 2-29. Paging Mechanism

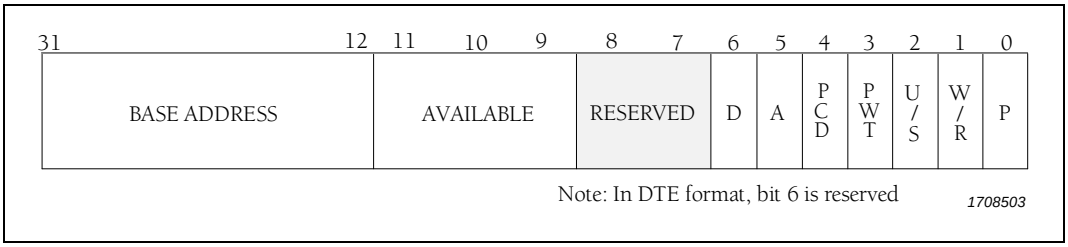


Figure 2-30. Directory and Page Table Entry (DTE and PTE) Format

Table 2-27. Directory and Page Table Entry (DTE and PTE) Bit Definitions

BIT POSITION	FIELD NAME	DESCRIPTION
31-12	BASE ADDRESS	Specifies the base address of the page or page table.
11-9	--	Undefined and available to the programmer.
8-7	--	Reserved and not available to the programmer.
6	D	Dirty Bit. If set, indicates that a write access has occurred to the page (PTE only, undefined in DTE).
5	A	Accessed Flag. If set, indicates that a read access or write access has occurred to the page.
4	PCD	Page Caching Disable Flag. If set, indicates that the page is not cacheable in the on-chip cache.
3	PWT	Page Write-Through Flag. If set, indicates that writes to the page or page tables that hit in the on-chip cache must update both the cache and external memory.
2	U/S	User/Supervisor Attribute. If set (user), page is accessible at privilege level 3. If clear (supervisor), page is accessible only when CPL ≤ 2.
1	W/R	Write/Read Attribute. If set (write), page is writable. If clear (read), page is read only.
0	P	Present Flag. If set, indicates that the page is present in RAM memory, and validates the remaining DTE/PTE bits. If clear, indicates that the page is not present in memory and the remaining DTE/PTE bits can be used by the programmer.

For a TLB hit, the TLB eliminates accesses to external directory and page tables.

The L1 TLB is a small cache optimized for speed whereas the L2 TLB is a much larger cache optimized for capacity. The L2 TLB is a proper superset of the L1 TLB.

The TLB must be flushed by the software when entries in the page tables are changed. Both the L1 and L2 TLBs are flushed whenever the CR3 register is loaded. A particular page can be flushed from the TLBs by using the INVLPG instruction.

2.12.4.1 Translation Lookaside Buffer Testing

The L1 and L2 Translation Lookaside Buffers (TLBs) can be tested by writing, then reading from the same TLB location. The operation to be performed is determined by the command (CMD) field (Table 2-28, Page 2-54) in the TR6 register.

Table 2-28. CMD Field

CMD	OPERATION	LINEAR ADDRESS BITS
x00	Write to L1	15 - 12
x01	Write to L2	17 - 12
010	Read from L1 X port	15 -12
011	Read from L2 X port	17 -12
110	Read from L1 Y port	15 - 12
110	Read from L2 Y port	17 - 12

TLB Write

To perform a write to the 6x86MX TLBs, the TR7 register (Figure 2-31) is loaded with the desired physical address as well as the PCD and PWT bits. For a write to the L2 TLB, the SET field of TR7 must be also specified. The H1, H2, and HSET fields of TR7 are not used. The TR6 register is then loaded with the linear address, V, D, U, W and A fields and the appropriate CMD. For a L1 TLB write, the TLB entry is selected by bits 15-12 of the linear address. For a L2 TLB write, the TLB entry is selected by bits 17-12 of the linear address and the SET field of TR7.

TLB Read

For a L1 LTB read, the TR6 register is loaded with the linear address and the appropriate CMD. The L1 TLB entry selected by bits 15-12 of the linear address will then be accessed. The linear address, V, D, PG, U, W and A fields of TR6 and the physical address, PCD and PWT fields of TR7 are loaded from the specified L1 entry. The H1 bit of TR7 will indicate if the specified linear address hit in the L1 TLB.

For a L2 TLB read, the TR7 register is loaded with the desired SET. The TR6 register is then loaded with the linear address and the appropriate CMD. The L2 TLB entry selected by bits 17-12 of the linear address and the SET field in TR7 will then be accessed. The linear address, V,D, PG, V, W, and A fields of TR6 and the physical address, PCD and PWT fields of TR7 are loaded from the specified L2 entry. The H2 bit of TR7 will indicate if the specified linear address hit in the L2 TLB. If there was an L2 hit, the HSET field of TR7 will indicate which SET hit.

The TLB test register fields are defined in Table 2-29. (Page 2-56).

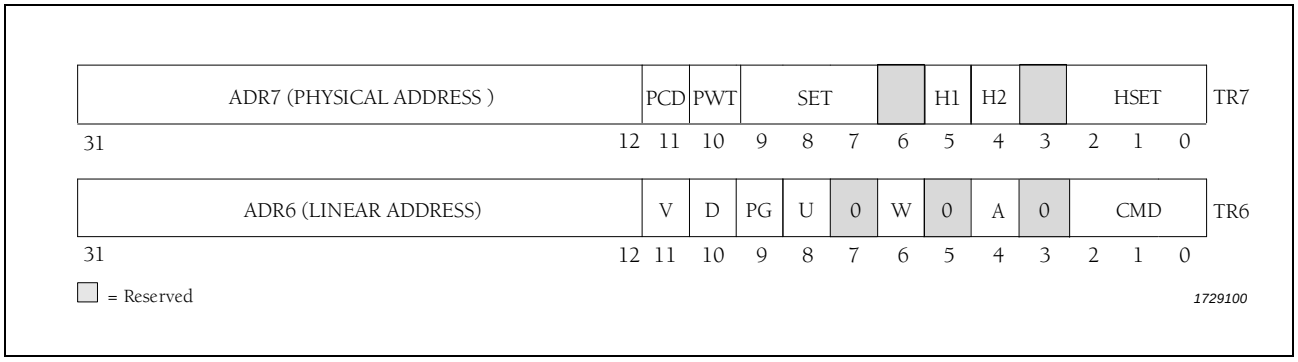


Figure 2-31. TLB Test Registers

Table 2-29. TLB Test Register Bit Definitions

REGISTER NAME	NAME	RANGE	DESCRIPTION
TR7	ADR7	31-12	Physical address or variable page size mechanism mask. TLB lookup: data field from the TLB. TLB write: data field written into the TLB.
	PCD	11	Page-level cache disable bit (PCD). Corresponds to the PCD bit of a page table entry.
	PWT	10	Page-level cache write-through bit (PWT). Corresponds to the PWT bit of a page table entry.
	SET	9-7	L2 TLB Set Selection (0h - 5h)
	H1	5	Hit in L1 TLB
	H2	4	Hit in L2 TLB
	HSET	2-0	L2 Set Selection when L2 TLB hit occurred (0h - 5h)
TR6	ADR6	31-12	Linear Address. TLB lookup: The TLB is interrogated per this address. If one and only one match occurs in the TLB, the rest of the fields in TR6 and TR7 are updated per the matching TLB entry. TLB write: A TLB entry is allocated to this linear address.
	V	11	PTE Valid. TLB write: If set, indicates that the TLB entry contains valid data. If clear, target entry is invalidated.
	D	10	Dirty Attribute Bit
	PG	9	Page Global
	U	8	User/Supervisor Attribute Bit
	W	6	Write Protect bit.
	CMD	2-0	Array Command Select. Determines TLB array command. Refer to Table 2-28, Page 2-54.

2.13 Memory Caches

The 6x86MX CPU contains two memory caches as described in Chapter 1. The Unified Cache acts as the primary data cache, and secondary instruction cache. The Instruction Line Cache is the primary instruction cache and provides a high speed instruction stream for the Integer Unit.

The unified cache is dual-ported allowing simultaneous access to any two unique banks. Two different banks may be accessed at the same time permitting any two of the following operations to occur in parallel:

- Code fetch
- Data read (X pipe, Y pipe or FPU)
- Data write (X pipe, Y pipe or FPU).

2.13.1 Unified Cache MESI States

The unified cache lines are assigned one of four MESI states as determined by MESI bits stored in tag memory. Each 32-byte cache line is divided into two 16-byte sectors. Each sector contains its own MESI bits. The four MESI states are described below:

Modified MESI cache lines are those that have been updated by the CPU, but the corresponding main memory location has not yet been updated by an external write cycle. Modified cache lines are referred to as dirty cache lines.

Exclusive MESI lines are lines that are exclusive to the 6x86MX CPU and are not duplicated within another caching agent's cache within the same system. A write to this cache line may be performed without issuing an external write cycle.

Shared MESI lines may be present in another caching agent's cache within the same system. A write to this cache line forces a corresponding external write cycle.

Invalid MESI lines are cache lines that do not contain any valid data.

2.13.1.1 Unified Cache Testing

The TR3, TR4, and TR5 test registers allow testing the unified cache. These registers can also be accessed as Model Specific Registers MSR(3), MSR(4), and MSR(5) using the RDMSR and WRMSR instructions. The data placed in the MSR registers determine which areas will be tested.

Cache Organization. The 64 KByte Unified Cache (Figure 2-32) is a 4-way set associative cache divided into 2,048 lines. There are 512 cache lines in each of the four sets. Each cache line is 32 bytes wide.

Memory address bits A13-A5 address sequential cache lines, repeating the same sequence in each set. Since each cache line represents any memory location with the same A13-A5 bits, the upper address bits A31-A14 are stored in the cache tag line. Memory address bits A4-A2 are used to select a particular 4-byte entry (ENT) within the cache line.

Test Initiation. A test register operation is initiated by writing to the TR5 register shown in Figure 2-33 (Page 2-59) using a special MOV instruction. The TR5 CTL field, detailed in Table 2-30 (Page 2-59), determines the function to be performed. For cache writes, the registers TR4 and TR3 must be initialized before a write is made to TR5. Eight 4-byte accesses are required to access a complete cache line.

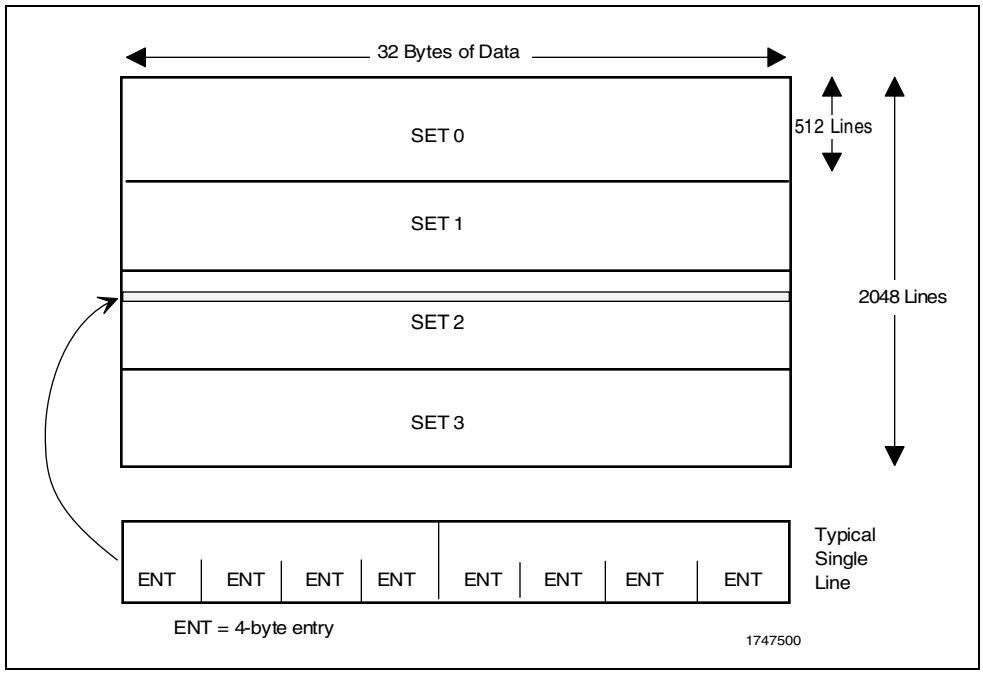


Figure 2-32. Unified Cache

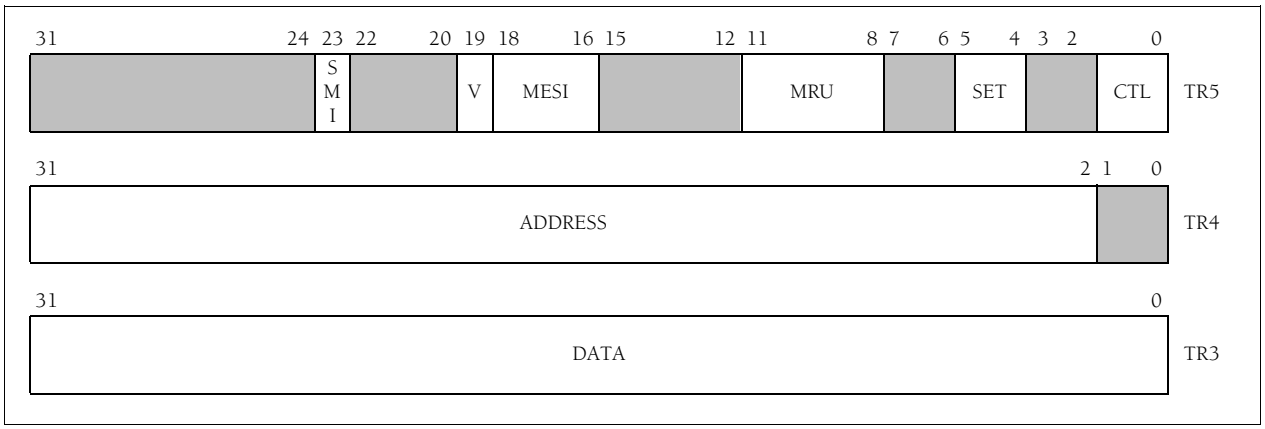


Figure 2-33. Cache Test Registers

Table 2-30. Cache Test Register Bit Definitions

REGISTER NAME	FIELD NAME	RANGE	DESCRIPTION
TR5	SMI	23	SMI Address Bit. Selects separate/cacheable SMI code/data space
	V, MESI	19 - 16	Valid, MESI Bits* If = 1000, Modified If = 1001, Shared If = 1010, Exclusive If = 0011, Invalid If = 1100, Locked Valid If = 0111, Locked Invalid Else = Undefined
	MRU	11 - 8	Used to determine the Least Recently Used (LRU) line.
	SET	5 - 4	Cache Set. Selects one of four cache sets to perform operation on.
	CTL	1 - 0	Control field If = 00: flush cache without invalidate If = 01: write cache If = 10: read cache If = 11: no cache or test register modification
TR4	ADDRESS	31 - 2	Physical Address
TR3	DATA	31 - 0	Data written or read during a cache test.

*Note: All 32 bytes should contain valid data before a line is marked as valid.

Write Operations. During a write, the TR3 DATA (32-bits) and TAG field information is written to the address selected by the ADDRESS field in TR4 and the SET field in TR5.

Read Operations. During a read, the cache address selected by the ADDRESS field in TR4 and the SET field in TR5. The TVB, MESI and MRU fields in TR5 are updated with the information from the selected line. TR3 holds the selected read data.

Cache Flushing. A cache flush occurs during a TR5 write if the CTL field is set to zero. During flushing, the CPU's cache controller reads through all the lines in the cache. "Modified" lines are redefined as "shared" by setting the shared MESI bit. Clean lines are left in their original state.

2.13.2 RAM Cache Locking

RAM cache locking (was called Scratch Pad Memory) sets up a private area of memory that can be assigned within the 6x86MX unified cache. Cached locked RAM is read/writable and is NOT kept coherent with the rest of the system. Scratch Pad Memory is a seperate memory on certain Cyrix CPUs.

Cache locking may be implemented differently on different processors. On the 6x86MX CPU, the cache locking RAM may be assigned on a cache line granularity.

RDMSR and WRMSR instructions (Page 2-39) with indices 03h to 05h are used to assign scratch pad memory. These instructions access the cache test registers. See section 2.13.1.1 (Page 2-58) for detailed description of cache test register operation. The cache line is assigned into Scratch Pad RAM by setting its MESI state to "locked valid."

When locking physical addresses into the cache (Table 2-31), the programmer should be aware of several issues:

1) Locking all sets of the cache should not be done. It is required that one set always be available for general purpose caching. 2) Care must be taken by the programmer not to create synonyms. This is done by first checking to see if a particular address is locked before attempting to lock the address. If synonyms are created, 6x86MX CPU operation will be undefined.

When ever possible, it is recommended to flush the cache before assigning locked memory areas. Locked areas of the cache are cleared on reset, and are unaffected by warm reset and FLUSH#, or the INVD and WBINVD instructions.

Table 2-31. RAM Cache Locking Operations

Read/Write	ECX	EDX	EAX	Operation
Read/Write	03h	----	Data to be read or written from/to the cache.	Loads or stores data to/from TR3.
Write	04h	----	32 bits of address	Address in EAX is loaded into TR4. This address is the cache line address that will be locked.
Read	04h	----	32 bits of address	Stores the contents of TR4 in EAX
Write	05h	----	Data to be written into TR5	Performs operation specified in CTL field of TR5.
Read	05h	----	Data in TR5 register	Reads data in TR5 and stores in EAX.

2.14 Interrupts and Exceptions

The processing of an interrupt or an exception changes the normal sequential flow of a program by transferring program control to a selected service routine. Except for SMM interrupts, the location of the selected service routine is determined by one of the interrupt vectors stored in the interrupt descriptor table.

Hardware interrupts are generated by signal sources external to the CPU. All exceptions (including so-called software interrupts) are produced internally by the CPU.

2.14.1 Interrupts

External events can interrupt normal program execution by using one of the three interrupt pins on the 6x86MX CPU.

- Non-maskable Interrupt (NMI pin)
- Maskable Interrupt (INTR pin)
- SMM Interrupt (SMI# pin).

For most interrupts, program transfer to the interrupt routine occurs after the current instruction has been completed. When the execution returns to the original program, it begins immediately following the last completed instruction.

With the exception of string operations, interrupts are acknowledged between instructions. Long string operations have interrupt windows between memory moves that allow interrupts to be acknowledged.

The **NMI interrupt** cannot be masked by software and always uses interrupt vector 2 to locate its service routine. Since the interrupt vector is fixed and is supplied internally, no interrupt acknowledge bus cycles are performed. This interrupt is normally reserved for unusual situations such as parity errors and has priority over INTR interrupts.

Once NMI processing has started, no additional NMIs are processed until an IRET instruction is executed, typically at the end of the NMI service routine. If NMI is re-asserted prior to execution of the IRET instruction, one and only one NMI rising edge is stored and processed after execution of the next IRET. During the NMI service routine, maskable interrupts may be enabled (unmasked). If an unmasked INTR occurs during the NMI service routine, the INTR is serviced and execution returns to the NMI service routine following the next IRET. If a HALT instruction is executed within the NMI service routine, the 6x86MX CPU restarts execution only in response to RESET, an unmasked INTR or an SMM interrupt. NMI does not restart CPU execution under this condition.

The **INTR interrupt** is unmasked when the Interrupt Enable Flag (IF) in the EFLAGS register is set to 1. When an INTR interrupt

occurs, the CPU performs two locked interrupt acknowledge bus cycles. During the second cycle, the CPU reads an 8-bit vector that is supplied by an external interrupt controller. This vector selects one of the 256 possible interrupt handlers which will be executed in response to the interrupt.

The **SMM interrupt** has higher priority than either INTR or NMI. After SMI# is asserted, program execution is passed to an SMI service routine that runs in SMM address space reserved for this purpose. The remainder of this section does not apply to the SMM interrupts. SMM interrupts are described in greater detail later in this chapter.

2.14.2 Exceptions

Exceptions are generated by an interrupt instruction or a program error. Exceptions are classified as traps, faults or aborts depending on the mechanism used to report them and the restartability of the instruction that first caused the exception.

A **Trap Exception** is reported immediately following the instruction that generated the trap exception. Trap exceptions are generated by execution of a software interrupt instruction (INTO, INT 3, INT n, BOUND), by a single-step operation or by a data breakpoint.

Software interrupts can be used to simulate hardware interrupts. For example, an INT n instruction causes the processor to execute the interrupt service routine pointed to by the nth vector in the interrupt table. Execution of the interrupt service routine occurs regardless of the state of the IF flag in the EFLAGS register.

The one byte INT 3, or breakpoint interrupt (vector 3), is a particular case of the INT n instruction. By inserting this one byte instruction in a program, the user can set breakpoints in the code that can be used during debug.

Single-step operation is enabled by setting the TF bit in the EFLAGS register. When TF is set, the CPU generates a debug exception (vector 1) after the execution of every instruction. Data breakpoints also generate a debug exception and are specified by loading the debug registers (DR0-DR7) with the appropriate values.

A **Fault Exception** is reported prior to completion of the instruction that generated the exception. By reporting the fault prior to instruction completion, the CPU is left in a state that allows the instruction to be restarted and the effects of the faulting instruction to be nullified. Fault exceptions include divide-by-zero errors, invalid opcodes, page faults and coprocessor errors. Instruction breakpoints (vector 1) are also handled as faults. After execution of the fault service routine, the instruction pointer points to the instruction that caused the fault.

An **Abort Exception** is a type of fault exception that is severe enough that the CPU cannot restart the program at the faulting instruction. The double fault (vector 8) is the only abort exception that occurs on the 6x86MX CPU.

2.14.3 Interrupt Vectors

When the CPU services an interrupt or exception, the current program's FLAGS, code segment and instruction pointer are pushed onto the stack to allow resumption of execution of the interrupted program. In protected mode, the processor also saves an error code for some exceptions. Program control is then transferred to the interrupt handler (also called the interrupt service routine). Upon execution of an IRET at the end of the service routine, program execution resumes by popping from the stack, the instruction pointer, code segment, and FLAGS.

Interrupt Vector Assignments

Each interrupt (except SMI#) and exception is assigned one of 256 interrupt vector numbers Table 2-32, (Page 2-65). The first 32 interrupt vector assignments are defined or reserved. INT instructions acting as software interrupts may use any of the interrupt vectors, 0 through 255.

Table 2-32. Interrupt Vector Assignments

INTERRUPT VECTOR	FUNCTION	EXCEPTION TYPE
0	Divide error	FAULT
1	Debug exception	TRAP/FAULT*
2	NMI interrupt	
3	Breakpoint	TRAP
4	Interrupt on overflow	TRAP
5	BOUND range exceeded	FAULT
6	Invalid opcode	FAULT
7	Device not available	FAULT
8	Double fault	ABORT
9	Reserved	
10	Invalid TSS	FAULT
11	Segment not present	FAULT
12	Stack fault	FAULT
13	General protection fault	TRAP/FAULT
14	Page fault	FAULT
15	Reserved	
16	FPU error	FAULT
17	Alignment check exception	FAULT
18-31	Reserved	
32-255	Maskable hardware interrupts	TRAP
0-255	Programmed interrupt	TRAP

*Note: Data breakpoints and single-steps are traps. All other debug exceptions are faults.

In response to a maskable hardware interrupt (INTR), the 6x86MX CPU issues interrupt acknowledge bus cycles to read the vector number from external hardware. These vectors should be in the range 32 - 255 as vectors 0 - 31 are reserved.

Interrupt Descriptor Table

The interrupt vector number is used by the 6x86MX CPU to locate an entry in the interrupt descriptor table (IDT). In real mode, each IDT entry consists of a four-byte far pointer to the beginning of the corresponding interrupt service routine. In protected mode, each IDT entry is an eight-byte descriptor. The Interrupt Descriptor Table Register (IDTR) specifies the beginning address and limit of the IDT. Following reset, the IDTR contains a base address of 0h with a limit of 3FFh.

The IDT can be located anywhere in physical memory as determined by the IDTR register. The IDT may contain different types of descriptors: interrupt gates, trap gates and task gates. Interrupt gates are used primarily to enter a hardware interrupt handler. Trap gates are generally used to enter an exception handler or software interrupt handler. If an interrupt gate is used, the Interrupt Enable Flag (IF) in the EFLAGS register is cleared before the interrupt handler is entered. Task gates are used to make the transition to a new task.

2.14.4 Interrupt and Exception Priorities

As the 6x86MX CPU executes instructions, it follows a consistent policy for prioritizing exceptions and hardware interrupts. The priorities for competing interrupts and exceptions are listed in Table 2-33 (Page 2-67). Debug traps for the previous instruction and the next instructions always take precedence. SMM interrupts are the next priority. When NMI and maskable INTR interrupts are both detected at the same instruction boundary, the 6x86MX processor services the NMI interrupt first.

The 6x86MX CPU checks for exceptions in parallel with instruction decoding and execution. Several exceptions can result from a single instruction. However, only one exception is generated upon each attempt to execute the instruction. Each exception service routine should make the appropriate corrections to the instruction and then restart the instruction. In this way, exceptions can be serviced until the instruction executes properly.

The 6x86MX CPU supports instruction restart after all faults, except when an instruction causes a task switch to a task whose task state segment (TSS) is partially not present. A TSS can be partially not present if the TSS is not page aligned and one of the pages where the TSS resides is not currently in memory.

Table 2-33. Interrupt and Exception Priorities

PRIORITY	DESCRIPTION	NOTES
0	Warm Reset	Caused by the assertion of WM_RST.
1	Debug traps and faults from previous instruction.	Includes single-step trap and data breakpoints specified in the debug registers.
2	Debug traps for next instruction.	Includes instruction execution breakpoints specified in the debug registers.
3	Hardware Cache Flush	Caused by the assertion of FLUSH#.
4	SMM hardware interrupt.	SMM interrupts are caused by SMI# asserted and always have highest priority.
5	Non-maskable hardware interrupt.	Caused by NMI asserted.
6	Maskable hardware interrupt.	Caused by INTR asserted and IF = 1.
7	Faults resulting from fetching the next instruction.	Includes segment not present, general protection fault and page fault.
8	Faults resulting from instruction decoding.	Includes illegal opcode, instruction too long, or privilege violation.
9	WAIT instruction and TS = 1 and MP = 1.	Device not available exception generated.
10	ESC instruction and EM = 1 or TS = 1.	Device not available exception generated.
11	Floating point error exception.	Caused by unmasked floating point exception with NE = 1.
12	Segmentation faults (for each memory reference required by the instruction) that prevent transferring the entire memory operand.	Includes segment not present, stack fault, and general protection fault.
13	Page Faults that prevent transferring the entire memory operand.	
14	Alignment check fault.	

2.14.5 Exceptions in Real Mode

Many of the exceptions described in Table 2-33 (Page 2-67) are not applicable in real mode. Exceptions 10, 11, and 14 do not occur in real mode. Other exceptions have slightly different meanings in real mode as listed in Table 2-34.

Table 2-34. Exception Changes in Real Mode

VECTOR NUMBER	PROTECTED MODE FUNCTION	REAL MODE FUNCTION
8	Double fault.	Interrupt table limit overrun.
10	Invalid TSS.	x
11	Segment not present.	x
12	Stack fault.	SS segment limit overrun.
13	General protection fault.	CS, DS, ES, FS, GS segment limit overrun.
14	Page fault.	x

Note: x = does not occur

2.14.6 Error Codes

When operating in protected mode, the following exceptions generate a 16-bit error code:

Double Fault	Invalid TSS
Alignment Check	Segment Not Present
Page Fault	Stack Fault
	General Protection Fault

The error code is pushed onto the stack prior to entering the exception handler. The error code format is shown in Figure 2-34 and the error code bit definitions are listed in Table 2-35. Bits 15-3 (selector index) are not meaningful if the error code was generated as the result of a page fault. The error code is always zero for double faults and alignment check exceptions.

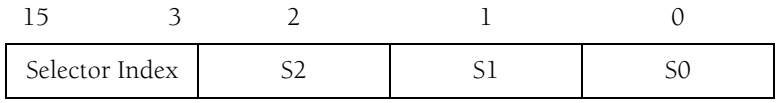


Figure 2-34. Error Code Format

Table 2-35. Error Code Bit Definitions

FAULT TYPE	SELECTOR INDEX (BITS 15-3)	S2 (BIT 2)	S1 (BIT 1)	S0 (BIT 0)
Double Fault or Alignment Check	0	0	0	0
Page Fault	Reserved.	Fault caused by: 0 = not present page 1 = page-level protection violation.	Fault occurred during: 0 = read access 1 = write access.	Fault occurred during: 0 = supervisor access. 1 = user access.
IDT Fault	Index of faulty IDT selector.	Reserved.	1	If = 1, exception occurred while trying to invoke exception or hardware interrupt handler.
Segment Fault	Index of faulty selector.	TI bit of faulty selector.	0	If =1, exception occurred while trying to invoke exception or hardware interrupt handler.

2.15 System Management Mode

System Management Mode (SMM) is a distinct CPU mode that differs from normal CPU x86 operating modes (real mode, V86 mode, and protected mode) and is most often used to perform power management.

The 6x86MX CPU is backward compatible with the SL-compatible SMM found on previous Cyrix microprocessors. On the 6x86MX SMM has been enhanced to optimized software emulation of multimedia and I/O peripherals.

The Cyrix Enhanced SMM provides new features:

- Cacheability of SMM memory
- Support for nesting of multiple SMIs
- Improved SMM entry and exit time.

Overall Operation

The overall operation of a SMM operation is shown in (Figure 2-35). SMM is entered using the System Management Interrupt (SMI) pin. SMI interrupts have higher priority than any other interrupt, including NMI interrupts. SMM can also be entered using software by using an SMINT instruction.

Upon entering SMM mode, portions of the CPU state are automatically saved in the SMM address memory space header. The CPU enters real mode and begins executing the SMI service routine in SMM address space.

Execution of a SMM routine starts at the base address in SMM memory address space. Since the SMM routines reside in SMM memory space, SMM routines can be made totally transparent to all software, including protected-mode operating systems.

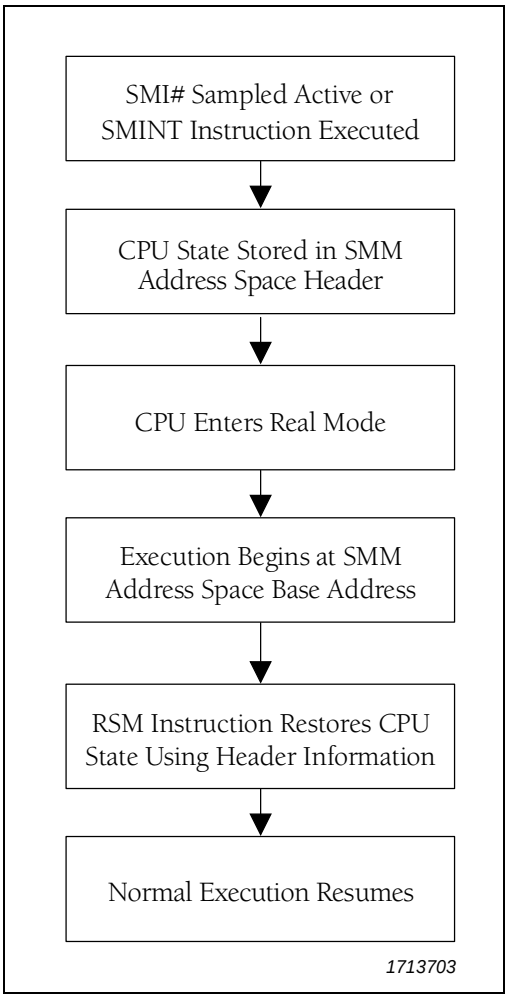


Figure 2-35. SMI Execution Flow Diagram

2.15.1 SMM Memory Space

SMM memory must reside within the bounds of physical memory and not overlap with system memory. SMM memory space (Figure 2-36) is defined by setting the SM3 bit in CCR1 and specifying the base address and size of the SMM memory space in the ARR3 register.

The base address must be a multiple of the SMM memory space size. For example, a 32 KByte SMM memory space must be located on a 32 KByte address boundary. The memory space size can range from 4 KBytes to 4 GBytes SMM accesses ignore the state of the A20M# input pin and drive the A20 address bit to the unmasked value.

SMM memory space can be accessed while in normal mode by setting the SMAC bit in the CCR1 register. This feature may be used to initialize SMM memory space.

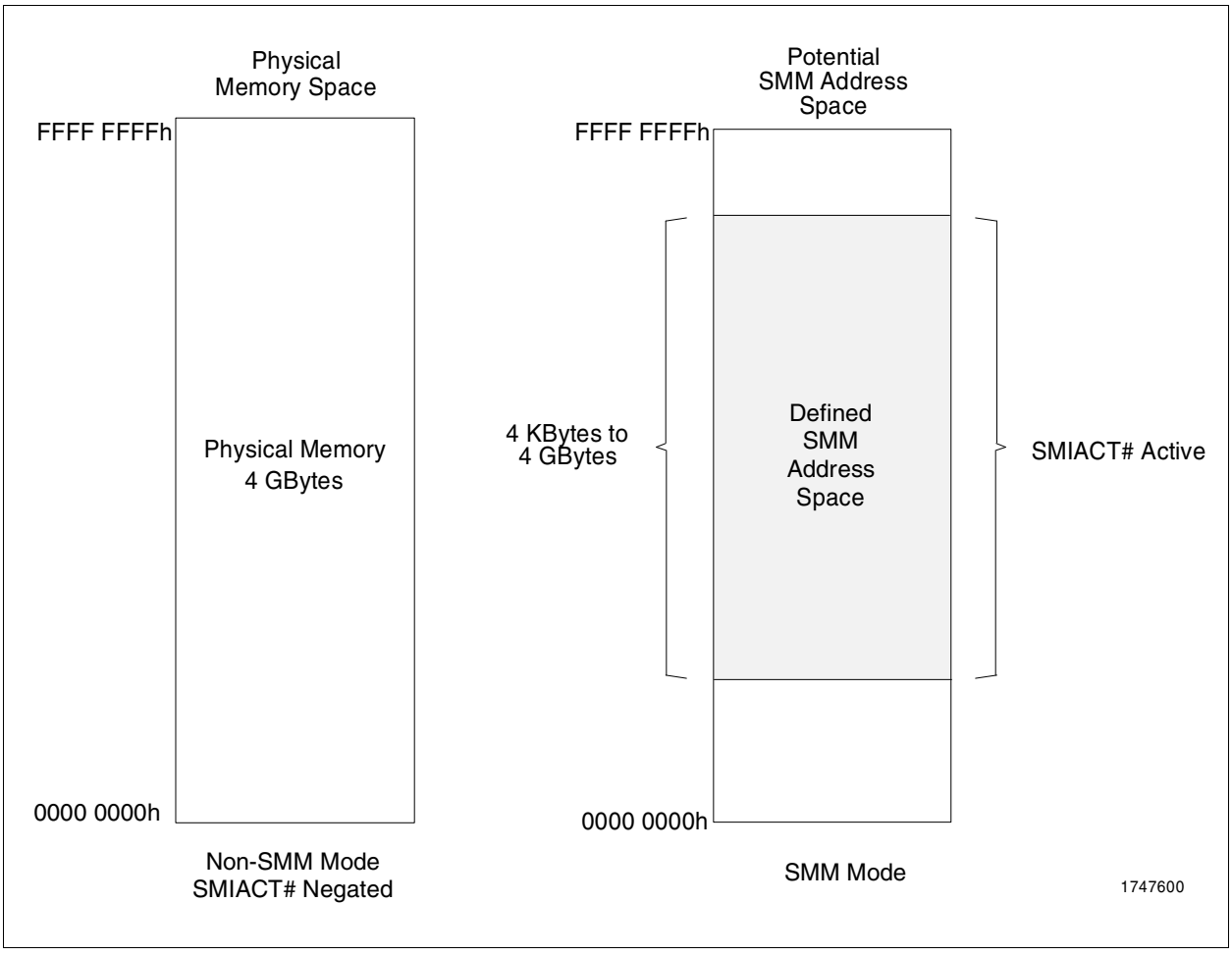


Figure 2-36. System Management Memory Space

2.15.2 SMM Memory Space Header

The SMM Memory Space Header (Figure 2-37) is used to store the CPU state prior to starting an SMM routine. The fields in this header are described in Table 2-36 (Page 2-73). After the SMM routine has completed, the header information is used to restore the original CPU state. The location of the SMM header is determined by the SMM Header Address Register (SMHR).

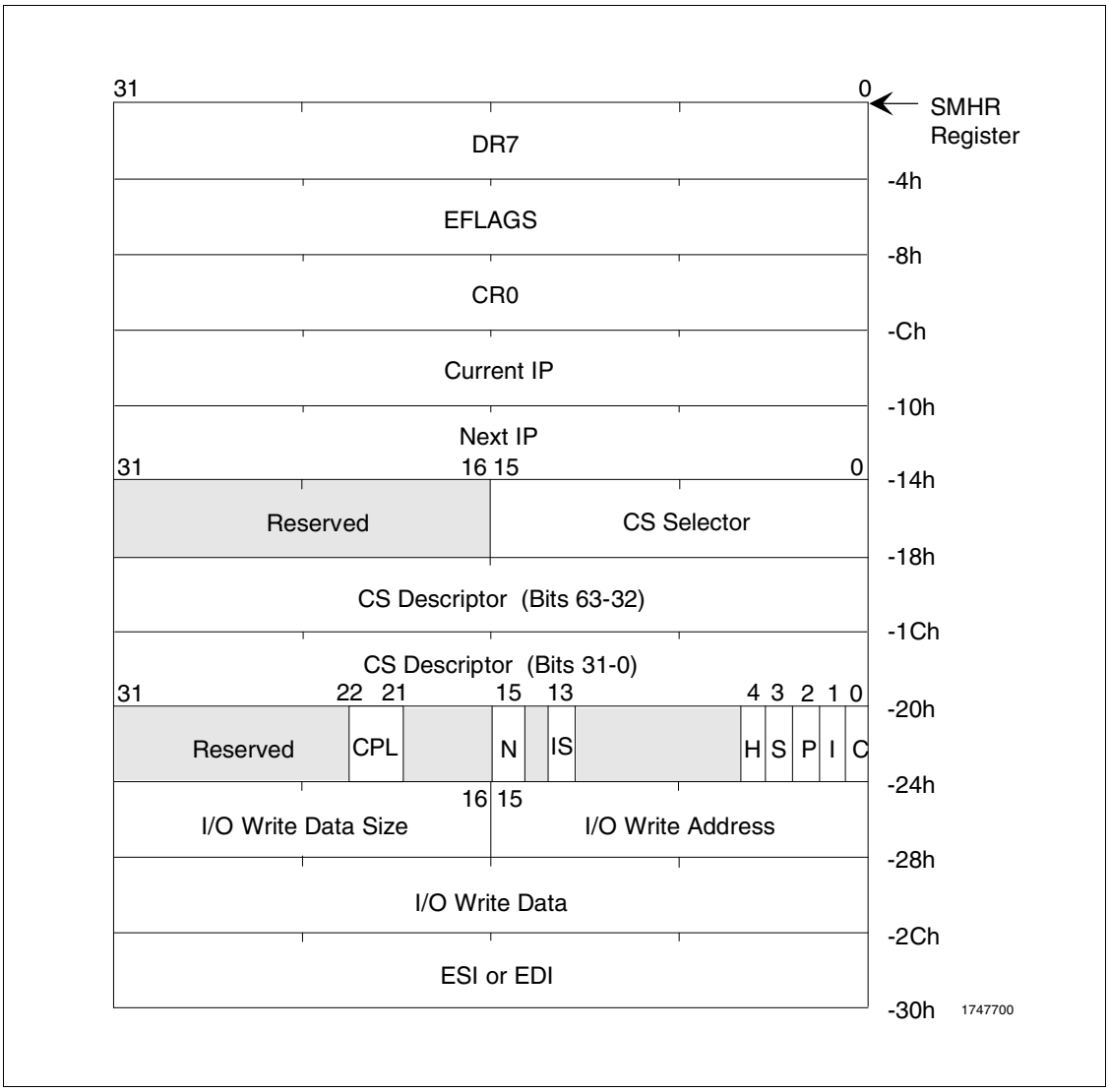


Figure 2-37. SMM Memory Space Header

Table 2-36. SMM Memory Space Header

NAME	DESCRIPTION	SIZE
DR7	The contents of Debug Register 7.	4 Bytes
EFLAGS	The contents of Extended Flags Register.	4 Bytes
CR0	The contents of Control Register 0.	4 Bytes
Current IP	The address of the instruction executed prior to servicing SMI interrupt.	4 Bytes
Next IP	The address of the next instruction that will be executed after exiting SMM mode.	4 Bytes
CS Selector	Code segment register selector for the current code segment.	2 Bytes
CS Descriptor	Code segment register descriptor for the current code segment.	8 Bytes
CPL	Current privilege level for current code segment.	2 Bits
N	Nested SMI Indicator If N = 1: current SMM is being serviced from within SMM mode. If N = 0: current SMM is not being serviced from within SMM mode.	1 Bit
IS	Internal SMI Indicator If IS =1: current SMM is the result of an internal SMI event. If IS =0: current SMM is the result of an external SMI event.	1 Bit
H	SMI during CPU HALT state indicator If H = 1: the processor was in a halt or shutdown prior to servicing the SMM interrupt.	1 Bit
S	Software SMM Entry Indicator. If S = 1: current SMM is the result of an SMINT instruction. If S = 0: current SMM is not the result of an SMINT instruction.	1 Bit
P	REP INSx/OUTSx Indicator If P = 1: current instruction has a REP prefix. If P = 0: current instruction does not have a REP prefix.	1 Bit
I	IN, INSx, OUT, or OUTSx Indicator If I = 1: if current instruction performed is an I/O WRITE. If I = 0: if current instruction performed is an I/O READ.	1 Bit
C	Code Segment writable Indicator If C = 1: the current code segment is writable. If C = 0: the current code segment is not writable.	1 Bit
I/O	Indicates size of data for the trapped I/O write: 01h = byte 03h = word 0Fh = dword	2 Bytes
I/O Write Address	I/O Write Address Processor port used for the trapped I/O write.	2 Bytes
I/O Write Data	I/O Write Data Data associated with the trapped I/O write.	4 Bytes
ESI or EDI	Restored ESI or EDI value. Used when it is necessary to repeat a REP OUTSx or REP INSx instruction when one of the I/O cycles caused an SMI# trap.	4 Bytes

Note: INSx = INS, INSB, INSW or INSD instruction.
Note: OUTSx = OUTS, OUTSB, OUTSW and OUTSD instruction.

Current and Next IP Pointers

Included in the header information are the Current and Next IP pointers. The Current IP points to the instruction executing when the SMI was detected and the Next IP points to the instruction that will be executed after exiting SMM.

Normally after an SMM routine is completed, the instruction flow begins at the Next IP address. However, if an I/O trap has occurred, instruction flow should return to the Current IP to complete the I/O instruction.

If SMM has been entered due to an I/O trap for a REP INSx or REP OUTSx instruction, the Current IP and Next IP fields contain the same address.

If an entry into SMM mode was caused by an I/O trap, the port address, data size and data value associated with that I/O operation are stored in the SMM header. Note that these values are only valid for I/O operations. The I/O data is not restored within the CPU when executing a RSM instruction.

Under these circumstances the I and P bits, as well as ESI/EDI field, contain valid information.

Also saved are the contents of debug register 7 (DR7), the extended flags register (EFLAGS), and control register 0 (CR0).

If the S bit in the SMM header is set, the SMM entry resulted from an SMINT instruction.

SMM Header Address Pointer

The SMM Header Address Pointer Register (SMHR) (Figure 2-38) contains the 32-bit SMM Header pointer. The SMHR address is dword aligned, so the two least significant bits are ignored.

The SMHR valid bit (bit 0) is cleared with every write to ARR3 and during a hardware RESET. Upon entry to SMM, the SMHR valid bit is examined before the CPU state is saved into the SMM memory space header. When the valid bit is reset, the SMM header pointer will be calculated (ARR3 base field + ARR3 size field) and loaded into the SMHR and the valid bit will be set.

If the desired SMM header location is different than the top of SMM memory space, as may be the case when nesting SMI's, then the SMHR register must be loaded with a new value and valid bit from within the SMI routine before nesting is enabled.

The SMM memory space header can be relocated using the new RDSHR and WRSHR instructions.

Figure 2-38. SMHR Register

Table 2-37. SMHR Register Bits

BIT POSITION	DESCRPTION
31 - 2	SMHR header pointer address.
1	Reserved
0	Valid Bit

2.15.3 SMM Instructions

After entering the SMI service routine, the MOV, SVDC, SVLDT and SVTS instructions (Table 2-38) can be used to save the complete CPU state information. If the SMI service routine modifies more than what is automatically saved or forces the CPU to power down, the complete CPU state information must be saved. Since the CPU is a static device, its internal state is retained when the input clock is stopped. Therefore, an entire CPU state save is not necessary prior to stopping the input clock.

Table 2-38. SMM Instruction Set

INSTRUCTION	OPCODE	FORMAT	DESCRIPTION
SVDC	0F 78 [mod sreg3 r/m]	SVDC mem80, sreg3	*Save Segment Register and Descriptor* Saves reg (DS, ES, FS, GS, or SS) to mem80.
RSDC	0F 79 [mod sreg3 r/m]	RSDC sreg3, mem80	*Restore Segment Register and Descriptor* Restores reg (DS, ES, FS, GS, or SS) from mem80. Use RSM to restore CS. Note: Processing "RSDC CS, Mem80" will produce an exception.
SVLDT	0F 7A [mod 000 r/m]	SVLDT mem80	*Save LDTR and Descriptor* Saves Local Descriptor Table (LDTR) to mem80.
RSLDT	0F 7B [mod 000 r/m]	RSLDT mem80	*Restore LDTR and Descriptor* Restores Local Descriptor Table (LDTR) from mem80.
SVTS	0F 7C [mod 000 r/m]	SVTS mem80	*Save TSR and Descriptor* Saves Task State Register (TSR) to mem80.
RSTS	0F 7D [mod 000 r/m]	RSTS mem80	*Restore TSR and Descriptor* Restores Task State Register (TSR) from mem80.
SMINT	0F 38	SMINT	*Software SMM Entry* CPU enters SMM mode. CPU state information is saved in SMM memory space header and execution begins at SMM base address.
RSM	0F AA	RSM	*Resume Normal Mode* Exits SMM mode. The CPU state is restored using the SMM memory space header and execution resumes at interrupted point.
RDSHR	0F 36	RDSHR ereg/mem32	*Read SMM Header Pointer Register* Saves SMM header pointer to extended register or memory.
WRSHR	0F 37	WRSHR ereg/mem32	*Write SMM Header Pointer Register* Load SMM header pointer register from extended register or memory.

Note: mem32 = 32-bit memory location
mem80 = 80-bit memory location

The SMM instructions listed in Table 2-38, (except the SMINT instruction) can be executed only if:

1) ARR3 Size > 0
2) Current Privilege Level =0
3) SMAC bit is set or the CPU is executing an SMI service routine.
4) USE_SMI (CCR1- bit 1) = 1
5) SM3 (CCR1-bit 7) = 1

If the above conditions are not met and an attempt is made to execute an SVDC, RSDC, SVLDT, RSLDT, SVTS, RSTS, SMINT, RSM, RDSHR, or WDSHR instruction, an invalid opcode exception is generated. These instructions can be executed outside of defined SMM space provided the above conditions are met.

The SMINT instruction allows software entry into SMM. The SVDC, RSDC, SVLDT, RSLDT, SVTS and RSTS instructions save or restore 80 bits of data, allowing the saved values to include the hidden portion of the register contents.

The WRSHR instruction loads the contents of either a 32-bit memory operand or a 32-bit register operand into the SMHR pointer register based on the value of the mod r/m instruction byte. Likewise the RDSHR instruction stores the contents of the SMHR pointer register to either a 32 bit memory operand or a 32 bit register operand based on the value of the mod r/m instruction byte.

2.15.4 SMM Operation

This section details the SMM operations.

Entering SMM

Entering SMM requires the assertion of the SMI# pin or execution of an SMINT instruction. SMI interrupts have higher priority than any interrupt including NMI interrupts.

For the SMI# or SMINT instruction to be recognized, the following configuration register bits must be set as shown in Table 2-39.

Table 2-39. Requirements for Recognizing SMI# and SMINT

REGISTER (Bit)		SMI#	SMINT
SMI	CCR1 (1)	1	1
SMAC	CCR1 (2)	0	1
ARR3	SIZE (3-0)	> 0	> 0
SM3	CCR1 (7)	1	1

Upon entry into SMM, after the SMM header has been saved, the CR0, EFLAGS, and DR7 registers are set to their reset values. The Code Segment (CS) register is loaded with the base, as defined by the ARR3 register, and a limit of 4 GBytes. The SMI service routine then begins execution at the SMM base address in real mode.

Saving the CPU State

The programmer must save the value of any registers that may be changed by the SMI service routine. For data accesses immediately after entering the SMI service routine, the programmer must use CS as a segment override. I/O port access is possible during the routine but care must be taken to save registers modified by the I/O instructions. Before using a segment register, the register and the register's descriptor cache contents should be saved using the SVDC instruction. While executing in the SMM space, execution flow can transfer to normal memory locations.

Program Execution

Hardware interrupts, (INTRs and NMIs), may be serviced during a SMI service routine. If interrupts are to be serviced while executing in the SMM memory space, the SMM memory space must be within the 0 to 1 MByte address range to guarantee proper return to the SMI service routine after handling the interrupt.

INTRs are automatically disabled when entering SMM since the IF flag is set to its reset value. Once in SMM, the INTR can be enabled by setting the IF flag. NMI is also automatically disable when entering SMM. Once in SMM, NMI can be enabled by setting NMI_EN in CCR3. If NMI is not enabled, the CPU latches one NMI event and services the interrupt after NMI has been enabled or after exiting SMM through the RSM instruction.

Within the SMI service routine, protected mode may be entered and exited as required, and real or protected mode device drivers may be called.

Exiting SMM

To exit the SMI service routine, a Resume (RSM) instruction, rather than an IRET, is executed. The RSM instruction causes the 6x86MX processor to restore the CPU state using the SMM header information and resume execution at the interrupted point. If the full CPU state was saved by the programmer, the stored values should be reloaded prior to executing the RSM instruction using the MOV, RSDC, RSLDT and RSTS instructions.

When the RSM instruction is executed at the end of the SMI handler, the EIP instruction pointer is automatically read from the NEXT IP field in the SMM header.

When restarting I/O instructions, the value of NEXT IP may need modification. Before executing the RSM instruction, use a MOV instruction to move the CURRENT IP value to the NEXT IP location as the CURRENT IP value is valid if an I/O instruction was executing when the SMI interrupt occurred. Execution is then returned to the I/O instruction, rather than to the instruction after the I/O instruction.

A set H bit in the SMM header indicates that a HLT instruction was being executed when the SMI occurred. To resume execution of the HLT instruction, the NEXT IP field in the SMM header should be decremented by one before executing RSM instruction.

2.15.5 SL and Cyrix SMM Operating Modes

There are two SMM modes, SL-compatible mode (default) and Cyrix SMM mode.

2.15.5.1 SL-Compatible SMM Mode

While in SL-compatible mode, SMM memory space accesses can only occur during an SMI service routine. While executing an SMI service routine SMIACT# remains asserted regardless of the address being accessed. This includes the time when the SMI service routine accesses memory outside the defined SMM memory space.

SMM memory caching is not supported in SL-compatible SMM mode. If a cache inquiry cycle occurs while SMIACT# is active, any resulting write-back cycle is issued with SMIACT# asserted. This occurs even though the write-back cycle is intended for normal memory rather than SMM memory. To avoid this problem it is recommended that the internal caches be flushed prior to servicing an SMI event. Of course in write-back mode this could add an indeterminate delay to servicing of SMI.

An interrupt on the SMI# input pin has higher priority than the NMI input. The SMI# input pin is falling edge sensitive and is sampled on every rising edge of the processor input clock.

Asserting SMI# forces the processor to save the CPU state to memory defined by SMHR register and to begin execution of the SMI service routine at the beginning of the defined SMM memory space. After the processor internally acknowledges the SMI# interrupt, the SMIACT# output is driven low for the duration of the interrupt service routine.

When the RSM instruction is executed, the CPU negates the SMIACT# pin after the last bus cycle to SMM memory. While executing the SMM service routine, one additional SMI# can be latched for service after resuming from the first SMI.

During RESET, the USE_SMI bit in CCR1 is cleared. While USE_SMI is zero, SMIACT# is always negated. SMIACT# does not float during bus hold states.

2.15.5.2 Cyrix Enhanced SMM Mode

The Cyrix SMM Mode is enabled when bit 0 in the CCR6 (SMM_MODE) is set. Only in Cyrix enhanced SMM mode can:

- SMM memory be cached
- SMM interrupts be nested

Pin Interface

The SMI# and SMIACT# pins behave differently in Cyrix Enhanced SMM mode.

In Cyrix Enhanced SMM mode SMI# is level sensitive. As a level sensitive signal software can process SMI interrupts until all sources in the chipset have been cleared.

While operating in this mode, SMIACT# output is not used to indicate that the CPU is operating in SMM mode. This is left to the SMM driver.

In Cyrix enhanced SMM, SMIACT# is asserted for every SMM memory bus cycle and is de-asserted for every non-SMM bus cycle. In this mode the SMIACT# pin meets the timing of D/C# and W/R#.

During RESET, the USE_SMI bit in CCR1 is cleared. While USE_SMI is zero, SMIACT# is always negated. SMIACT# does float during bus hold states.

Cacheability of SMM Space

In SL-compatible SMM mode, caching is not available, but in Cyrix SMM mode, both code and data caching is supported. In order to cache SMM data and avoid coherency issues the processor assumes no overlap of main memory with SMM memory. This implies that a section of main memory must be dedicated for SMM.

The on-chip cache sets a special ID bit in the cache tag block for each line that contains SMM code data. This ID bit is then used by the bus controller to regulate assertion of the SMIACT# pin for write-back of any SMM data.

Nested SMI

Only in the Cyrix Enhanced SMM mode is nesting of SMI interrupts supported. This is important to allow high priority events such as audio emulation to interrupt lower priority SMI code. In the case of nesting, it is up to the SMM driver to determine which SMM event is being serviced, which to prioritize, and perform all SMM interrupt control functions.

Software enables and disables SMI interrupts while in SMM mode by setting and clearing the nest-enable bit (N bit, bit 6 of CCR6). By default the CPU automatically disables SMI interrupts (clears the N bit) on entry to SMM mode, and re-enables them (sets the N bit) when exiting SMM mode (i.e., RSM). The SMI handler can optionally enable nesting to allow higher priority SMI interrupts to occur while handling the current SMI event.

The SMI handler is responsible for managing the SMHR pointer register when processing nested SMI interrupts. Before nested SMI's can be serviced the current SMM handler must save the contents of the SMHR pointer register and then load a new value into the SMHR register for use by a subsequent nested SMI event.

Prior to execution of a RSM instruction the contents of the old SMHR pointer register must be restored for proper operation to continue. Prior to restoring the contents of old SMHR pointer register one should disable additional SMI's. This should be done so that the CPU will not inadvertently receive and service an SMI event after the old SMHR contents have been restored but before the RSM instruction is executed.

2.15.6 Maintaining the FPU and MMX States

If power will be removed from the CPU or if the SMM routine will execute MMX or FPU instructions, then the MMX or FPU state should be maintained for the application running before SMM was entered. If the MMX or FPU state is to be saved and restored from within SMM, there are certain guidelines that must be followed to make SMM completely transparent to the application program.

The complete state of the FPU can be saved and restored with the FNSAVE and FNRSTOR instructions. FNSAVE is used instead of the FSAVE because FSAVE will wait for the FPU to check for existing error conditions before storing the FPU state. If there is a unmasked FPU exception condition pending, the FSAVE instruction will wait until the exception condition is serviced. To maintain transparency for the application program, the SMM routine should not service this exception. If the FPU state is restored with the FNRSTOR instruction before returning to normal mode, the application program can correctly service the exception. FPU instructions can be executed within SMM once the FPU state has been saved.

The information saved with the FSAVE instruction varies depending on the operating mode of the CPU. To save and restore all FPU information, the 32-bit protected mode version of the FPU save and restore instruction should be used.

CPU States Related to SMM and Suspend Mode

The state diagram shown in Figure 2-39 (Page 2-81) illustrates the various CPU states associated with SMM and suspend mode. While in the SMI service routine, the 6x86MX CPU can enter suspend mode either by (1) executing a halt (HLT) instruction or (2) by asserting the SUSP# input.

During SMM operations and while in SUSP# initiated suspend mode, an occurrence of SMI#, NMI, or INTR is latched. (In order for INTR to be latched, the IF flag must be set.) The INTR or NMI is serviced after exiting suspend mode.

If suspend mode is entered via a HLT instruction from the operating system or application software, the reception of an SMI# interrupt causes the CPU to exit suspend mode and enter SMM.

2.16 Shutdown and Halt

The **Halt Instruction** (HLT) stops program execution and prevents the processor from using the local bus until restarted. The 6x86MX CPU then issues a special Stop Grant bus cycle and enters a low-power suspend mode if the SUSP_HLT bit in CCR2 is set. SMI, NMI, INTR with interrupts enabled (IF bit in EFLAGS=1), WM_RST or RESET forces the CPU out of the halt state. If interrupted, the saved code segment and instruction pointer specify the instruction following the HLT.

Shutdown occurs when a severe error is detected that prevents further processing. An NMI input can bring the processor out of shutdown if the IDT limit is large enough to contain the NMI interrupt vector and the stack has enough room to contain the vector and flag information. Otherwise, shutdown can only be exited by a processor reset.

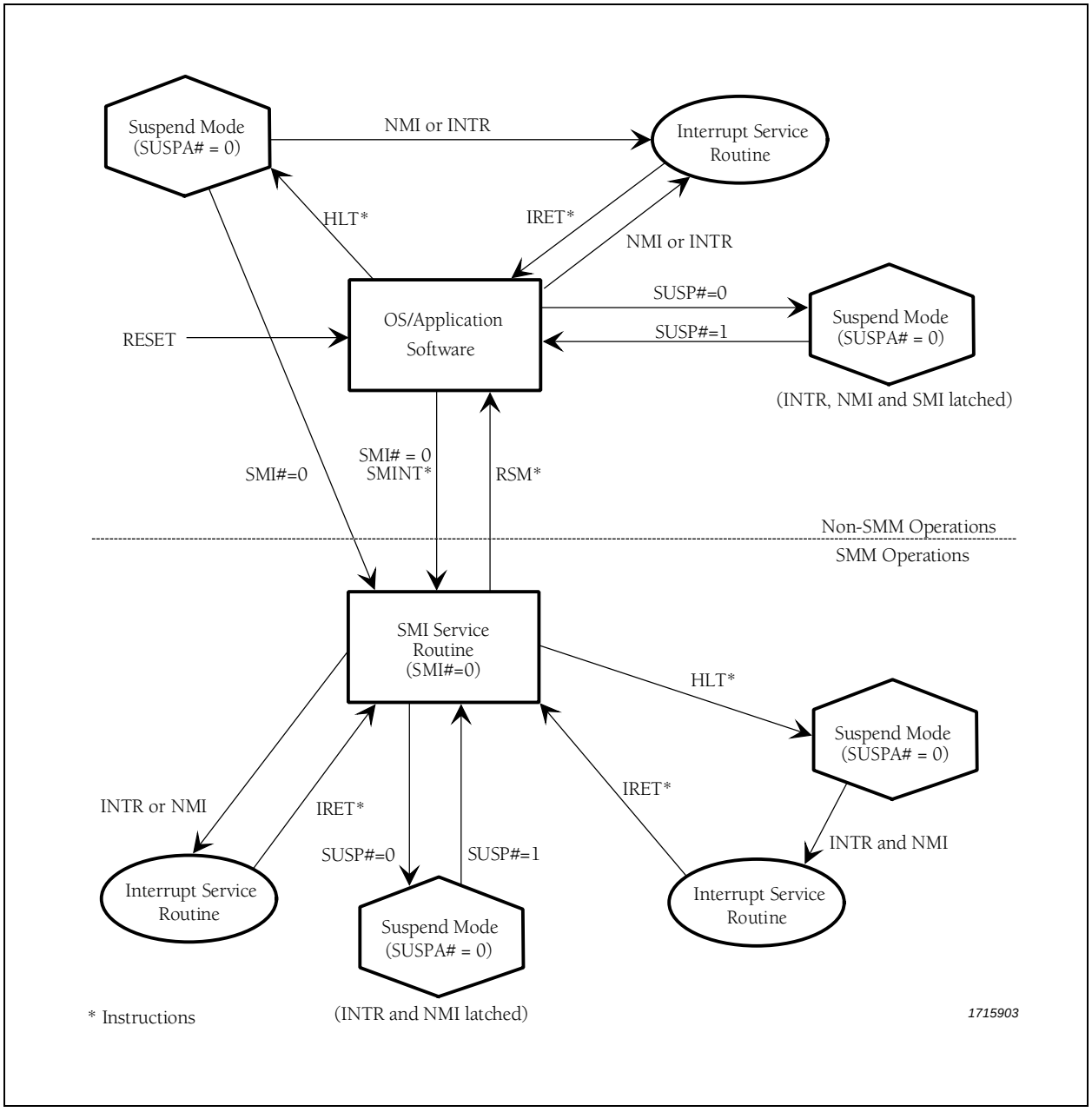


Figure 2-39. SMM and Suspend Mode State Diagram

2.17 Protection

Segment protection and page protection are safeguards built into the 6x86MX CPU protected mode architecture which deny unauthorized or incorrect access to selected memory addresses. These safeguards allow multitasking programs to be isolated from each other and from the operating system. Page protection is discussed earlier in this chapter. This section concentrates on segment protection.

Selectors and descriptors are the key elements in the segment protection mechanism. The segment base address, size, and privilege level are established by a segment descriptor. Privilege levels control the use of privileged instructions, I/O instructions and access to segments and segment descriptors. Selectors are used to locate segment descriptors.

Segment accesses are divided into two basic types, those involving code segments (e.g., control transfers) and those involving data accesses. The ability of a task to access a segment depends on the:

- Segment type
- Instruction requesting access
- Type of descriptor used to define the segment
- Associated privilege levels (described below).

Data stored in a segment can be accessed only by code executing at the same or a more privileged level. A code segment or procedure can only be called by a task executing at the same or a less privileged level.

2.17.1 Privilege Levels

The values for privilege levels range between 0 and 3. Level 0 is the highest privilege level (most privileged), and level 3 is the lowest privilege level (least privileged). The privilege level in real mode is effectively 0.

The **Descriptor Privilege Level** (DPL) is the privilege level defined for a segment in the segment descriptor. The DPL field specifies the minimum privilege level needed to access the memory segment pointed to by the descriptor.

The **Current Privilege Level** (CPL) is defined as the current task's privilege level. The CPL of an executing task is stored in the hidden portion of the code segment register and essentially is the DPL for the current code segment.

The **Requested Privilege Level** (RPL) specifies a selector's privilege level and is used to distinguish between the privilege level of a routine actually accessing memory (the CPL), and the privilege level of the original requestor (the RPL) of the memory access. The lesser of the RPL and CPL is called the effective privilege level (EPL). Therefore, if RPL = 0 in a segment selector, the effective privilege level is always determined by the CPL. If RPL = 3, the effective privilege level is always 3 regardless of the CPL.

For a memory access to succeed, the effective privilege level (EPL) must be at least as privileged as the descriptor privilege level (EPL ≤ DPL). If the EPL is less privileged than the DPL (EPL > DPL), a general protection fault is generated. For example, if a segment has a DPL = 2, an instruction accessing the segment only succeeds if executed with an EPL ≤ 2.

2.17.2 I/O Privilege Levels

The I/O Privilege Level (IOPL) allows the operating system executing at CPL=0 to define the least privileged level at which IOPL-sensitive instructions can unconditionally be used. The IOPL-sensitive instructions include CLI, IN, OUT, INS, OUTS, REP INS, REP OUTS, and STI. Modification of the IF bit in the EFLAGS register is also sensitive to the I/O privilege level. The IOPL is stored in the EFLAGS register.

An I/O permission bit map is available as defined by the 32-bit Task State Segment (TSS). Since each task can have its own TSS, access to individual processor I/O ports can be granted through separate I/O permission bit maps.

If CPL ≤ IOPL, IOPL-sensitive operations can be performed. If CPL > IOPL, a general protection fault is generated if the current task is associated with a 16-bit TSS. If the current task is associated with a 32-bit TSS and CPL > IOPL, the CPU consults the I/O permission bitmap in the TSS to determine on a port-by-port basis whether or not I/O instructions (IN, OUT, INS, OUTS, REP INS, REP OUTS) are permitted, and the remaining IOPL-sensitive operations generate a general protection fault.

2.17.3 Privilege Level Transfers

A task's CPL can be changed only through intersegment control transfers using gates or task switches to a code segment with a different privilege level. Control transfers result from exception and interrupt servicing and from execution of the CALL, JMP, INT, IRET and RET instructions.

There are five types of control transfers that are summarized in Table 2-40 (Page 2-84). Control transfers can be made only when the operation causing the control transfer references the correct descriptor type. Any violation of these descriptor usage rules causes a general protection fault.

Any control transfer that changes the CPL within a task results in a change of stack. The initial values for the stack segment (SS) and stack pointer (ESP) for privilege levels 0, 1, and 2 are stored in the TSS. During a CALL control transfer, the SS and ESP are loaded with the new stack pointer and the previous stack pointer is saved on the new stack. When returning to the original privilege level, the RET or IRET instruction restores the less-privileged stack

Table 2-40. Descriptor Types Used for Control Transfer

TYPE OF CONTROL TRANSFER	OPERATION TYPES	DESCRIPTOR REFERENCED	DESCRIPTOR TABLE
Intersegment within the same privilege level.	JMP, CALL, RET, IRET*	Code Segment	GDT or LDT
Intersegment to the same or a more privileged level. Interrupt within task (could change CPL level).	CALL	Gate Call	GDT or LDT
	Interrupt Instruction, Exception, External Interrupt	Trap or Interrupt Gate	IDT
Intersegment to a less privileged level (changes task CPL).	RET, IRET*	Code Segment	GDT or LDT
Task Switch via TSS	CALL, JMP	Task State Segment	GDT
Task Switch via Task Gate	CALL, JMP	Task Gate	GDT or LDT
	IRET**, Interrupt Instruction, Exception, External Interrupt	Task Gate	IDT

* NT (Nested Task bit in EFLAGS) = 0
** NT (Nested Task bit in EFLAGS) = 1

Gates

Gate descriptors provide protection for privilege transfers among executable segments. Gates are used to transition to routines of the same or a more privileged level. Call gates, interrupt gates and trap gates are used for privilege transfers within a task. Task gates are used to transfer between tasks.

Gates conform to the standard rules of privilege. In other words, gates can be accessed by a task if the effective privilege level (EPL) is the same or more privileged than the gate descriptor's privilege level (DPL).

2.17.4 Initialization and Transition to Protected Mode

The 6x86MX processor switches to real mode immediately after RESET. While operating in real mode, the system tables and registers should be initialized. The GDTR and IDTR must point to a valid GDT and IDT, respectively. The GDT must contain descriptors which describe the initial code and data segments.

The processor can be placed in protected mode by setting the PE bit in the CR0 register. After enabling protected mode, the CS register should be loaded and the instruction decode queue should be flushed by executing an intersegment JMP. Finally, all data segment registers should be initialized with appropriate selector values.

2.18 Virtual 8086 Mode

Both real mode and virtual 8086 (V86) mode are supported by the 6x86MX CPU allowing execution of 8086 application programs and 8086 operating systems. V86 mode allows the execution of 8086-type applications, yet still permits use of the 6x86MX CPU paging mechanism. V86 tasks run at privilege level 3. When loaded, all segment limits are set to FFFFh (64K) as in real mode.

2.18.1 V86 Memory Addressing

While in V86 mode, segment registers are used in an identical fashion to real mode. The contents of the segment register are multiplied by 16 and added to the offset to form the segment base linear address. The 6x86MX CPU permits the operating system to select which programs use the V86 address mechanism and which programs use protected mode addressing for each task.

The 6x86MX CPU also permits the use of paging when operating in V86 mode. Using paging, the 1-MByte memory space of the V86 task can be mapped to anywhere in the 4-GByte linear memory space of the 6x86MX CPU.

The paging hardware allows multiple V86 tasks to run concurrently, and provides protection and operating system isolation. The paging hardware must be enabled to run multiple V86 tasks or to relocate the address space of a V86 task to physical address space greater than 1 MByte.

2.18.2 V86 Protection

All V86 tasks operate with the least amount of privilege (level 3) and are subject to all of the 6x86MX CPU protected mode protection checks. As a result, any attempt to execute a privileged instruction within a V86 task results in a general protection fault.

In V86 mode, a slightly different set of instructions are sensitive to the I/O privilege level (IOPL) than in protected mode. These instructions are: CLI, INT n, IRET, POPF, PUSHF, and STI. The INT3, INTO and BOUND variations of the INT instruction are not IOPL sensitive.

2.18.3 V86 Interrupt Handling

To fully support the emulation of an 8086-type machine, interrupts in V86 mode are handled as follows. When an interrupt or exception is serviced in V86 mode, program execution transfers to the interrupt service routine at privilege level 0 (i.e., transition from V86 to protected mode occurs) and the VM bit in the EFLAGS register is cleared. The protected mode interrupt service routine then determines if the interrupt came from a protected mode or V86 application by examining the VM bit in the EFLAGS image stored on the stack. The interrupt service routine may then choose to allow the 8086 operating system to handle the interrupt or may emulate the function of the interrupt handler. Following completion of the interrupt service routine, an IRET

instruction restores the EFLAGS register (restores VM=1) and segment selectors and control returns to the interrupted V86 task.

2.18.4 Entering and Leaving V86 Mode

V86 mode is entered from protected mode by either executing an IRET instruction at CPL = 0 or by task switching. If an IRET is used, the stack must contain an EFLAGS image with VM = 1. If a task switch is used, the TSS must contain an EFLAGS image containing a 1 in the VM bit position. The POPF instruction cannot be used to enter V86 mode since the state of the VM bit is not affected. V86 mode can only be exited as the result of an interrupt or exception. The transition out must use a 32-bit trap or interrupt gate which must point to a non-conforming privilege level 0 segment (DPL = 0), or a 32-bit TSS. These restrictions are required to permit the trap handler to IRET back to the V86 program.

2.19 Floating Point Unit Operations

The 6x86MX CPU includes an on-chip FPU that provides the user access to a complete set of floating point instructions (see Chapter 6). Information is passed to and from the FPU using eight data registers accessed in a stack-like manner, a control register, and a status register. The 6x86MX CPU also provides a data register tag word which improves context switching and performance by maintaining empty/non-empty status for each of the eight data registers. In addition, registers in the CPU contain pointers to (a) the memory location containing the current instruction word and (b) the memory location containing the operand associated with the current instruction word (if any).

FPU Tag Word Register. The 6x86MX CPU maintains a tag word register (Figure 2-40 (Page 2-87)) comprised of two bits for each physical data register. Tag Word fields assume one of four values depending on the contents of their associated data registers, Valid (00), Zero (01), Special (10), and Empty (11). Note: Denormal, Infinity, QNaN, SNaN and unsupported formats are tagged as “Special”. Tag values are maintained transparently by the 6x86MX CPU and are only available to the programmer indirectly through the FSTENV and FSAVE instructions.

FPU Control and Status Registers. The FPU circuitry communicates information about its status and the results of operations to the programmer via the status register. The FPU status register is comprised of bit fields that reflect exception status, operation execution status, register status, operand class, and comparison results. The FPU status register bit definitions are shown in Figure 2-41 (Page 2-87) and Table 2-41 (Page 2-87).

The FPU Mode Control Register (MCR) is used by the CPU to specify the operating mode of the FPU. The MCR contains bit fields which specify the rounding mode to be used, the precision by which to calculate results, and the exception conditions which should be reported to the CPU via traps. The user controls precision, rounding, and exception reporting by setting or clearing appropriate bits in the MCR. The FPU mode control register bit definitions are shown in Figure 2-42 (Page 2-88) and Table 2-42 (Page 2-88).

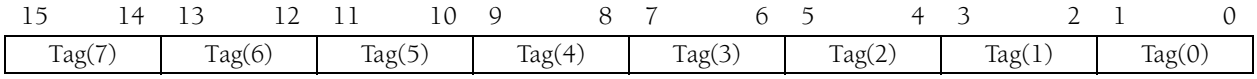


Figure 2-40. FPU Tag Word Register

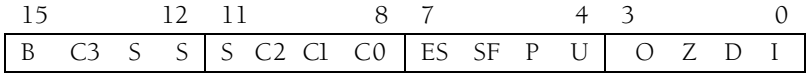


Figure 2-41. FPU Status Register

Table 2-41. FPU Status Register Bit Definitions

BIT POSITION	NAME	DESCRIPTION
15	B	Copy of the ES bit. (ES is bit 7 in this table.)
14, 10 - 8	C3 - C0	Condition code bits.
13 - 11	SSS	Top of stack register number which points to the current TOS.
7	ES	Error indicator. Set to 1 if an unmasked exception is detected.
6	SF	Stack Fault or invalid register operation bit.
5	P	Precision error exception bit.
4	U	Underflow error exception bit.
3	O	Overflow error exception bit.
2	Z	Divide by zero exception bit.
1	D	Denormalized operand error exception bit.
0	I	Invalid operation exception bit.

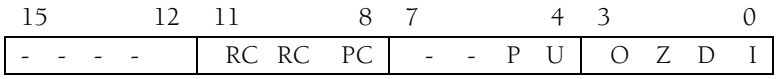


Figure 2-42. FPU Mode Control Register

Table 2-42. FPU Mode Control Register Bit Definitions

BIT POSITION	NAME	DESCRIPTION
11 - 10	RC	Rounding Control bits: 00 Round to nearest or even 01 Round towards minus infinity 10 Round towards plus infinity 11 Truncate
9 - 8	PC	Precision Control bits: 00 24-bit mantissa 01 Reserved 10 53-bit mantissa 11 64-bit mantissa
5	P	Precision error exception bit mask.
4	U	Underflow error exception bit mask.
3	O	Overflow error exception bit mask.
2	Z	Divide by zero exception bit mask.
1	D	Denormalized operand error exception bit mask.
0	I	Invalid operation exception bit mask.

2.20 MMX Operations

The 6x86MX CPU provides user access to the MMX instruction set. MMX data is configured in one of four MMX data formats. During operations eight 64-bit MMX registers are utilized.

2.20.1 MMX Data Formats

The MMX instructions operate on 64-bit data groups called "packed data." A single packed data group can be interpreted as a:

- Packed byte (8 bytes)
- Packed word (4 words)
- Packed doubleword (2 doublewords)
- Quadword (1 quadword)

The packed data types supported are signed and unsigned integer.

2.20.2 MMX Registers

The MMX instruction set operates on eight 64-bit, general-purpose registers (MM0-MM7). These registers are overlayed with the floating point register stack, so no new architectural state is defined by the MMX instruction set. Existing mechanisms for saving and restoring floating point state automatically work for saving and restoring MMX state.

2.20.3 MMX Instruction Set

The MMX instructions operate on all the elements of a signed or unsigned packed data group. All data elements (bytes, words, doublewords or a quadword) are operated on separately in parallel. For example, eight bytes in one packed data group can be added to another packed data group, such that eight independent byte additions are performed in parallel.

2.20.4 Instruction Group Overview

The 57 MMX instructions are grouped into seven categories:

- Arithmetic Instructions
- Comparison Instructions
- Conversion Instructions
- Logical Instructions
- Shift Instructions
- Data Transfer Instructions
- Empty MMX State (EMMS) Instruction

2.20.5 Saturation Arithmetic

For saturating MMX instructions, a ceiling is placed on an overflow and a floor is placed on an underflow. When the result of an operation exceeds the range of the data-type it saturates to the maximum value of the range. Conversely, when a result that is less than the range of a data type, the result saturates to the minimum value of the range.

The saturation limits are shown in Table 2-43.

Table 2-43. Saturation Limits

DATA TYPE	LOWER LIMIT		UPPER LIMIT	
Signed Byte	80h	-128	7Fh	127
Signed Word	8000h	-32,768	7FFFh	32,767
Unsigned Byte	00h	0	FFh	256
Unsigned Word	0000h	0	FFFFh	65,535

MMX instructions do not indicate overflow or underflow occurrence by generating exceptions or setting flags.

2.20.6 EMMS Instruction

The EMMS Instruction clears the TOS pointer and sets the entire FPU tag word as empty. An EMMS instruction should be executed at the end of each MMX routine.

3.0 6x86MX BUS INTERFACE

The signals used in the 6x86MX CPU bus interface are described in this chapter. Figure 3-1 shows the signal directions and the major signal groupings. A description of each signal and their reference to the text are provided in Table 3-1 (Page 3-2).

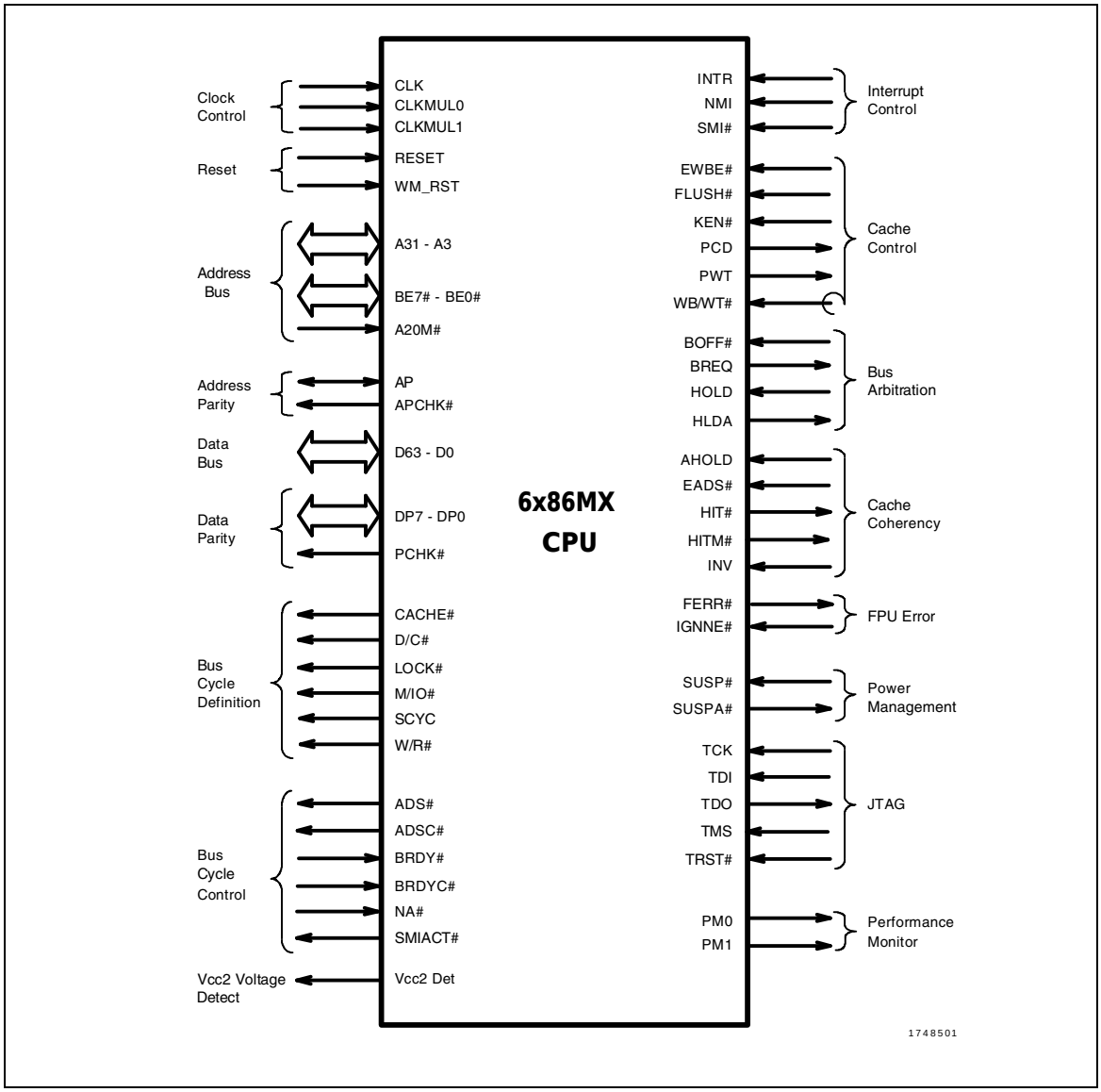


Figure 3-1. 6x86MX CPU Functional Signal Groupings

3.1 Signal Description Table

The Signal Summary Table (Table 3-1) describes the signals in their active state unless otherwise mentioned. Signals containing slashes (/) have logic levels defined as "1/0." For example the signal W/R#, is defined as write when W/R#=1, and as read when W/R#=0. Signals ending with a "#" character are active low.

Table 3-1. 6x86MX CPU Signals Sorted by Signal Name

Signal Name	Description	I/O	Reference
A20M#	**A20 Mask** causes the CPU to mask (force to 0) the A20 address bit when driving the external address bus or performing an internal cache access. A20M# is provided to emulate the 1 MByte address wrap-around that occurs on the 8086. Snoop addressing is not affected.	Input	Page 3-9
A31-A3	The **Address Bus**, in conjunction with the Byte Enable signals (BE7#-BE0#), provides addresses for physical memory and external I/O devices. During cache inquiry cycles, A31-A5 are used as inputs to perform cache line invalidations.	3-state I/O	Page 3-9
ADS#	**Address Strobe** begins a memory/I/O cycle and indicates the address bus (A31-A3, BE7#-BE0#) and bus cycle definition signals (CACHE#, D/C#, LOCK#, M/IO#, PCD, PWT, SCYC, W/R#) are valid.	Output	Page 3-13
ADSC#	**Cache Address Strobe** performs the same function as ADS#.	Output	Page 3-13
AHOLD	**Address Hold** allows another bus master access to the 6x86MX CPU address bus for a cache inquiry cycle. In response to the assertion of AHOLD, the CPU floats AP and A31-A3 in the following clock cycle.	Input	Page 3-18
AP	**Address Parity** is the even parity output signal for address lines A31-A5 (A4 and A3 are excluded). During cache inquiry cycles, AP is the even-parity input to the CPU, and is sampled with EADS# to produce correct parity check status on the APCHK# output.	3-state I/O	Page 3-10
APCHK#	**Address Parity Check Status** is asserted during a cache inquiry cycle if an address bus parity error has been detected. APCHK# is valid two clocks after EADS# is sampled active. APCHK# will remain asserted for one clock cycle if a parity error is detected.	Output	Page 3-10
BE7#-BE0#	The **Byte Enables**, in conjunction with the address lines, determine the active data bytes transferred during a memory or I/O bus cycle.	3-state I/O	Page 3-9
BOFF#	**Back-Off** forces the 6x86MX CPU to abort the current bus cycle and relinquish control of the CPU local bus during the next clock cycle. The 6x86MX CPU enters the bus hold state and remains in this state until BOFF# is negated.	Input	Page 3-16
BRDY#	**Burst Ready** indicates that the current transfer within a burst cycle, or the current single transfer cycle, can be terminated. The 6x86MX CPU samples BRDY# in the second and subsequent clocks of a bus cycle. BRDY# is active during address hold states.	Input	Page 3-13
BRDYC#	**Cache Burst Ready** performs the same function as BRDY# and is logically ORed with BRDY# within the 6x86MX CPU.	Input	Page 3-13

Table 3-1. 6x86MX CPU Signals Sorted by Signal Name (Continued)

Signal Name	Description	I/O	Reference
BREQ	**Bus Request** is asserted by the 6x86MX CPU when an internal bus cycle is pending. The 6x86MX CPU always asserts BREQ, along with ADS#, during the first clock of a bus cycle. If a bus cycle is pending, BREQ is asserted during the bus hold and address hold states. If no additional bus cycles are pending, BREQ is negated prior to termination of the current cycle.	Output	Page 3-16
CACHE#	**Cacheability Status** indicates that a read bus cycle is a potentially cacheable cycle; or that a write bus cycle is a cache line write-back or line replacement burst cycle. If CACHE# is asserted for a read cycle and KEN# is asserted by the system, the read cycle becomes a cache line fill burst cycle.	Output	Page 3-11
CLK	**Clock** provides the fundamental timing for the 6x86MX CPU. The frequency of the 6x86MX CPU input clock determines the operating frequency of the CPU's bus. External timing is defined referenced to the rising edge of CLK.	Input	Page 3-7
CLKMUL1-CLKMUL0	The **Clock Multiplier** inputs are sampled during RESET to determine the 6x86MX CPU core operating frequency. If = 00 core/bus ratio is 2.5 If = 01 core/bus ratio is 3.0 If = 10 core/bus ratio is 2.0 (default) If = 11 core/bus ratio is 3.5	Input	Page 3-7
D63-D0	**Data Bus** signals are three-state, bi-directional signals which provide the data path between the 6x86MX CPU and external memory and I/O devices. The data bus is only driven while a write cycle is active (state=T2).	3-state I/O	Page 3-10
D/C#	**Data/Control Status**. If high, indicates that the current bus cycle is an I/O or memory data access cycle. If low, indicates a code fetch or special bus cycle such as a halt, prefetch, or interrupt acknowledge bus cycle. D/C# is driven valid in the same clock as ADS# is asserted.	Output	Page 3-11
DP7-DP0	**Data Parity** signals provide parity for the data bus, one data parity bit per data byte. Even parity is driven on DP7-DP0 for all data write cycles. DP7-DP0 are read by the 6x86MX CPU during read cycles to check for even parity. The data parity bus is only driven while a write cycle is active (state=T2).	3-state I/O	Page 3-10
EADS#	**External Address Strobe** indicates that a valid cache inquiry address is being driven on the 6x86MX CPU address bus (A31-A5) and AP. The state of INV at the time EADS# is sampled active determines the final state of the cache line. A cache inquiry cycle using EADS# may be run while the 6x86MX CPU is in the address hold or bus hold state.	Input	Page 3-18
EWBE#	**External Write Buffer Empty** indicates that there are no pending write cycles in the external system. EWBE# is sampled only during I/O and memory write cycles. If EWBE# is negated, the 6x86MX CPU delays all subsequent writes to on-chip cache lines in the "exclusive" or "modified" state until EWBE# is asserted.	Input	Page 3-15
FERR#	**FPU Error Status** indicates an unmasked floating point error has occurred. FERR# is asserted during execution of the FPU instruction that caused the error. FERR# does not float during bus hold states.	Output	Page 3-19

Table 3-1. 6x86MX CPU Signals Sorted by Signal Name (Continued)

Signal Name	Description	I/O	Reference
FLUSH#	**Cache Flush** forces the 6x86MX CPU to flush the cache. External interrupts and additional FLUSH# assertions are ignored during the flush. Cache inquiry cycles are permitted during the flush.	Input	Page 3-15
HIT#	**Cache Hit** indicates that the current cache inquiry address has been found in the cache (modified, exclusive or shared states). HIT# is valid two clocks after EADS# is sampled active, and remains valid until the next cache inquiry cycle.	Output	Page 3-18
HITM#	**Cache Hit Modified Data** indicates that the current cache inquiry address has been found in the cache and dirty data exists in the cache line (modified state). The 6x86MX CPU does not accept additional cache inquiry cycles while HITM# is asserted. HITM# is valid two clocks after EADS#.	Output	Page 3-18
HLDA	**Hold Acknowledge** indicates that the 6x86MX CPU has responded to the HOLD input and relinquished control of the local bus. The 6x86MX CPU continues to operate during bus hold as long as the on-chip cache can satisfy bus requests.	Output	Page 3-17
HOLD	**Hold Request** indicates that another bus master has requested control of the CPU's local bus.	Input	Page 3-16
IGNNE#	**Ignore Numeric Error** forces the 6x86MX CPU to ignore any pending unmasked FPU errors and allows continued execution of floating point instructions.	Input	Page 3-19
INTR	**Maskable Interrupt** forces the processor to suspend execution of the current instruction stream and begin execution of an interrupt service routine. The INTR input can be masked (ignored) through the IF bit in the Flags Register.	Input	Page 3-14
INV	**Invalidate Request** is sampled with EADS# to determine the final state of the cache line in the case of a cache inquiry hit. An asserted INV directs the processor to change the state of the cache line to "invalid". A negated INV directs the processor to change the state of the cache line to "shared."	Input	Page 3-18
KEN#	**Cache Enable** allows the data being returned during the current cycle to be placed in the CPU's cache. When the 6x86MX CPU is performing a cacheable code fetch or memory data read cycle (CACHE# asserted), and KEN# is sampled asserted, the cycle is transformed into a 32-byte cache line fill. KEN# is sampled with the first asserted BRDY# or NA# for the cycle.	Input	Page 3-15
LOCK#	**Lock Status** indicates that other system bus masters are denied access to the local bus. The 6x86MX CPU does not enter the bus hold state in response to HOLD while LOCK# is asserted.	Output	Page 3-11
M/IO#	**Memory/IO Status**. If high, indicates that the current bus cycle is a memory cycle (read or write). If low, indicates that the current bus cycle is an I/O cycle (read or write, interrupt acknowledge, or special cycle).	Output	Page 3-11

Table 3-1. 6x86MX CPU Signals Sorted by Signal Name (Continued)

Signal Name	Description	I/O	Reference
NA#	**Next Address** requests the next pending bus cycle address and cycle definition information. If either the current or next bus cycle is a locked cycle, a line replacement, a write-back cycle, or if there is no pending bus cycle, the 6x86MX CPU does not start a pipelined bus cycle regardless of the state of NA#.	Input	Page 3-13
NMI	**Non-Maskable Interrupt Request** forces the processor to suspend execution of the current instruction stream and begin execution of an NMI interrupt service routine.	Input	Page 3-14
PCD	**Page Cache Disable** reflects the state of the PCD page attribute bit in the page table entry or the directory table entry. If paging is disabled, or for cycles that are not paged, the PCD pin is driven low. PCD is masked by the cache disable (CD) bit in CR0, and floats during bus hold states.	Output	Page 3-15
PCHK#	**Data Parity Check** indicates that a data bus parity error has occurred during a read operation. PCHK# is only valid during the second clock immediately after read data is returned to the 6x86MX CPU (BRDY# asserted) and is inactive otherwise. Parity errors signaled by a logic low on PCHK# have no effect on processor execution.	Output	Page 3-10
PM0-PM1	**Performance Monitor** indicate an at least one overflow or event occurred in the associated Performance Monitor Register (0-1).	Output	Page 3-20
PWT	**Page Write-Through** reflects the state of the PWT page attribute bit in the page table entry or the directory table entry. PWT pin is negated during cycles that are not paged, or if paging is disabled. PWT takes priority over WB/WT#.	Output	Page 3-15
RESET	**Reset** suspends all operations in progress and places the 6x86MX CPU into a reset state. Reset forces the CPU to begin executing in a known state. All data in the on-chip caches is invalidated.	Input	Page 3-7
SCYC	**Split Locked Cycle** indicates that the current bus cycle is part of a misaligned locked transfer. SCYC is defined for locked cycles only. A misaligned transfer is defined as any transfer that crosses an 8-byte boundary.	Output	Page 3-11
SMI#	**SMM Interrupt** forces the processor to save the CPU state to the top of SMM memory and to begin execution of the SMI service routine at the beginning of the defined SMM memory space. An SMI is a higher-priority interrupt than an NMI.	Input	Page 3-14
SMIACT#	**SMM Interrupt Active** indicates that the processor is operating in System Management Mode. SMIACT# does not float during bus hold states.	Output	Page 3-13
SUSP#	**Suspend Request** requests that the CPU enter suspend mode. SUSP# is ignored following RESET and is enabled by setting the SUSP bit in CCR2.	Input	Page 3-19
SUSPA#	**Suspend Acknowledge** indicates that the 6x86MX CPU has entered low-power suspend mode. SUSPA# floats following RESET and is enabled by setting the SUSP bit in CCR2.	Output	Page 3-19
TCK	**Test Clock** (JTAG) is the clock input used by the 6x86MX CPU's boundary scan (JTAG) test logic.	Input	Page 3-22

Table 3-1. 6x86MX CPU Signals Sorted by Signal Name (Continued)

Signal Name	Description	I/O	Reference
TDI	**Test Data In** (JTAG) is the serial data input used by the 6x86MX CPU's boundary scan (JTAG) test logic.	Input	Page 3-22
TDO	**Test Data Out** (JTAG) is the serial data output used by the 6x86MX CPU's boundary scan (JTAG) test logic.	Output	Page 3-22
TMS	**Test Mode Select** (JTAG) is the control input used by the 6x86MX CPU's boundary scan (JTAG) test logic.	Input	Page 3-22
TRST#	**Test Mode Reset** (JTAG) initializes the 6x86MX CPU's boundary scan (JTAG) test logic.	Input	Page 3-22
VCC2DET	**Vcc2 Detect** is always driven low by the CPU to indicate that the 6x86MX processor requires two different Vcc voltages.	Output	
WB/WT#	**Write-Back/Write-Through** is sampled during cache line fills to define the cache line write policy. If high, the cache line write policy is write-back. If low, the cache line write policy is write-through. (PWT forces write-through policy when PWT=1.)	Input	Page 3-16
WM_RST	**Warm Reset** forces the 6x86MX CPU to complete the current instruction and then places the 6x86MX CPU in a known state. Once WM_RST is sampled active by the CPU, the reset sequence begins on the next instruction boundary. WM_RST does not change the state of the configuration registers, the on-chip cache, the write buffers and the FPU registers. WM_RST is sampled during reset.	Input	Page 3-9
W/R#	**Write/Read Status**. If high, indicates that the current memory, or I/O bus cycle is a write cycle. If low, indicates that the current bus cycle is a read cycle.	Output	Page 3-11

3.2 Signal Descriptions

The following paragraphs provide additional information about the 6x86MX CPU signals. For ease of this discussion, the signals are divided into 16 functional groups as illustrated in Figure 3-1 (Page 3-1).

3.2.1 Clock Control

The **Clock Input (CLK)** signal, supplied by the system, is the timing reference used by the 6x86MX CPU bus interface. All external timing parameters are defined with respect to the CLK rising edge. The CLK signal enters the 6x86MX CPU where it is multiplied to produce the 6x86MX CPU internal clock signal. During power on, the CLK signal must be running even if CLK does not meet AC specifications.

The **Clock Multiplier (CLKMUL0, CLMUL1)** inputs are sampled during RESET to determine the CPU's core operating frequency (Table 3-2).

Table 3-2. Clock Control

CLKMUL1	CLKMUL0	CORE TO BUS CLOCK RATIO
0	0	2.5
0	1	3.0
1	0	2.0 (Default)
1	1	3.5

The CLKMUL pins have internal pull-up and pull down resistors to define the default ratio. Therefore the default setting indicates which mode the CPU will operate in if the CLKMUL are not driven and left floating.

3.2.2 Reset Control

The 6x86MX CPU output signals are initialized to their reset states during the CPU reset sequence, as shown in Table 3-4 (Page 3-8). The signal states given in Table 3-4 assume that HOLD, AHOLD, and BOFF# are negated.

Asserting **RESET** suspends all operations in progress and places the 6x86MX CPU in a reset state. RESET is an asynchronous signal but must meet specified setup and hold times to guarantee recognition at a particular clock edge.

On system power-up, RESET must be held asserted for at least 1 msec after Vcc and CLK have reached specified DC and AC limits. This delay allows the CPU's clock circuit to stabilize and guarantees proper completion of the reset sequence.

During normal operation, RESET must be asserted for at least 15 CLK periods in order to guarantee the proper reset sequence is executed. When RESET negates (on its falling edge), the pins listed in Table 3-3 determine if certain 6x86MX CPU functions are enabled

Table 3-3. Pins Sampled During RESET

SIGNAL NAME	DESCRIPTION
FLUSH#	If = 0, three-state test mode enabled.
WM_RST	If = 1, built-in self test initiated.

Table 3-4. Signal States During RESET

SIGNAL LINE	STATE
A20M#	Ignored
A31-A3	Undefined until first ADS#
ADS#	1
ADSC#	1
AHOLD	Recognized
AP	Undefined until first ADS#
APCHK#	1
BE7#-BE0#	Undefined until first ADS#
BOFF#	Recognized
BRDY#	Ignored
BRDYC#	Ignored
BREQ	0
CACHE#	Undefined until first ADS#
D(63-0)	Float
D/C#	Undefined until first ADS#
DP(7-0)	Float
EADS#	Ignored
EWBE#	Ignored
FERR#	1
FLUSH#	Initiates three-state test mode
HIT#	1
HITM#	1
HLDA	Responds to HOLD
HOLD	Recognized
IGNNE#	Ignored

SIGNAL LINE	STATE
INTR	Ignored
INV	Ignored
KEN#	Ignored
LOCK#	1
M/IO#	Undefined until first ADS#
NA#	Ignored
NMI	Ignored
PCD	Undefined until first ADS#
PCHK#	1
PWT	Undefined until first ADS#
RESET	1
SCYC	Undefined until first ADS#
SMI#	Ignored
SMIACT#	1
SUSP#	Ignored
SUSPA#	Float
TCK	Recognized
TDI	Recognized
TDO	Responds to TCK, TDI, TMS, TRST#
TMS	Recognized
TRST#	Recognized
W/R#	Undefined until first ADS#
WB/WT#	Ignored
WM_RST	Initiates self-test

Warm Reset (WM_RST) allows the 6x86MX CPU to complete the current instruction and then places the 6x86MX CPU in a known state. WM_RST is an asynchronous signal, but must meet specified setup and hold times in order to guarantee recognition at a particular CLK edge. Once WM_RST is sampled active by the CPU, the reset sequence begins on the next instruction boundary.

WM_RST differs from RESET in that the contents of the on-chip cache, the write buffers, the configuration registers and the floating point registers contents remain unchanged.

Following completion of the internal reset sequence, normal processor execution begins even if WM_RST remains asserted. If RESET and WM_RST are asserted simultaneously, WM_RST is ignored and RESET takes priority. If WM_RST is asserted at the falling edge of RESET, built-in self test (BIST) is initiated.

3.2.3 Address Bus

The **Address Bus (A31-A3)** lines provide the physical memory and external I/O device addresses. A31-A5 are bi-directional signals used by the 6x86MX CPU to drive addresses to both memory devices and I/O devices. During cache inquiry cycles the 6x86MX CPU receives addresses from the system using signals A31-A5.

Using signals A31-A3, the 6x86MX CPU can address a 4-GByte memory address space. Using signals A15-A3, the 6x86MX CPU can address a 64-KByte I/O space through the processor's I/O ports. During I/O accesses, signals A31-A16 are driven low. A31-A3 float during bus hold and address hold states.

The **Byte Enable (BE7#-BE0#)** lines are bi-directional signals that define the valid data bytes within the 64-bit data bus. The correlation between the enable signals and data bytes is shown in Table 3-5.

Table 3-5. Byte Enable Signal to Data Bus Byte Correlation

BYTE ENABLE	CORRESPONDING DATA BYTE
BE7#	D63-D56
BE6#	D55-D48
BE5#	D47-D40
BE4#	D39-D32
BE3#	D31-D24
BE2#	D23-D16
BE1#	D15-D8
BE0#	D7-D0

During a cache line fill, (burst read or "1+4" burst read) the 6x86MX CPU expects data to be returned as if all data bytes are enabled, regardless of the state of the byte enables. BE7#-BE0# float during bus hold and byte enable hold states.

Address Bit 20 Mask (A20M#) is an active low input which causes the 6x86MX CPU to mask (force low) physical address bit 20 when driving the external address bus or when performing an internal cache access. Asserting A20M# emulates the 1 MByte address wrap-around that occurs on the 8086. The A20 signal is never masked during write-back cycles, inquiry cycles, system management address space accesses or when paging is enabled, regardless of the state of the A20M# input.

3.2.4 Address Parity

Address Parity (AP) is a bi-directional signal which provides the parity associated with address lines A31-A5. (A4 and A3 are not included in the parity determination.) During 6x86MX CPU generated bus cycles, while the address bus lines are driven, AP becomes an output supplying even address parity. During cache inquiry cycles, AP becomes an input and is sampled by EADS#. During cache inquiry cycles, even-parity must be placed on the AP line to guarantee an accurate result on the APCHK# (Address Parity Check Status) pin.

Address Parity Check Status (APCHK#) is driven active by the CPU when an address bus parity error has been detected for a cache inquiry cycle. APCHK# is asserted two clocks after EADS# is sampled asserted, and remains valid for one clock only. Address parity errors signaled by APCHK# have no effect on processor execution.

3.2.5 Data Bus

Data Bus (D63-D0) lines carry three-state, bi-directional signals between the 6x86MX CPU and the system (i.e., external memory and I/O devices). The data bus transfers data to the 6x86MX CPU during memory read, I/O read, and interrupt acknowledge cycles. Data is transferred from the 6x86MX CPU during memory and I/O write cycles.

Data setup and hold times must be met for correct read cycle operation. The data bus is driven only while a write cycle is active.

3.2.6 Data Parity

The **Data Parity Bus (DP7-DP0)** provides and receives parity data for each of the eight data bus bytes (Table 3-6). The 6x86MX CPU generates even parity on the bus during write cycles and accepts even parity from the system during read cycles. DP7-DP0 is driven only while a write cycle is active.

Table 3-6. Parity Bit to Data Byte Correlation

PARITY BIT	DATA BYTE
DP7	D63-D56
DP6	D55-D48
DP5	D47-D40
DP4	D39-D32
DP3	D31-D24
DP2	D23-D16
DP1	D15-D8
DP0	D7-D0

Parity Check (PCHK#) is asserted when a data bus parity error is detected. Parity is checked during code, memory and I/O reads, and the second interrupt acknowledge cycle. Parity is not checked during the first interrupt acknowledge cycle.

Parity is checked for only the active data bytes as determined by the active byte enable signals except during a cache line fill (burst read or "1+4" burst read). During a cache line fill, the 6x86MX CPU assumes all data bytes are valid and parity is checked for all data bytes regardless of the state of the byte enables.

PCHK# is valid only during the second clock immediately after read data is returned to the 6x86MX CPU (BRDY# asserted). At other times PCHK# is not active. Parity errors signaled by the assertion of PCHK# have no effect on processor execution.

3.2.7 Bus Cycle Definition

Each bus cycle is assigned a bus cycle type. The bus cycle types are defined by six three-state outputs: CACHE#, D/C#, LOCK#, M/IO#, SCYC, and W/R# as listed in Table 3-7 (Page 3-12).

These bus cycle definition signals are driven valid while ADS# is active. D/C#, M/IO#, W/R#, SCYC and CACHE# remain valid until the clock following the earliest of two signals: NA# asserted, or the last BRDY# for the cycle.

LOCK# continues asserted until after BRDY# is returned for the last locked bus cycle. The bus cycle definition signals float during bus hold states.

Cache Cycle Indicator (CACHE#) is an output that indicates that the current bus cycle is a potentially cacheable cycle (for a read), or indicates that the current bus cycle is a cache line write-back or line replacement burst cycle (for a write). If CACHE# is asserted for a read cycle and the KEN# input is returned active by the system, the read cycle becomes a cache line fill burst cycle.

Data/Control (D/C#) distinguishes between data and control operations. When high, this signal indicates that the current bus cycle is a data transfer to or from memory or I/O. When low, D/C# indicates that the current bus cycle involves a control function such as a halt, interrupt acknowledge or code fetch.

Bus Lock (LOCK#) is an active low output which, when asserted, indicates that other system bus masters are denied access to control of the CPU bus. The LOCK# signal may be explicitly activated during bus operations by including the LOCK prefix on certain instructions. LOCK# is also asserted during descriptor updates, page table accesses, interrupt acknowledge sequences and when executing the XCHG instruction. However, if the NO_LOCK bit in CCR1 is set, LOCK# is asserted only during page table accesses and interrupt acknowledge sequences. The 6x86MX CPU does not enter the bus hold state in response to HOLD while the LOCK# output is active.

Memory/IO (M/IO#) distinguishes between memory and I/O operations. When high, this signal indicates that the current bus cycle is a memory read or memory write. When low, M/IO# indicates that the current bus cycle is an I/O read, I/O write, interrupt acknowledge cycle or special bus cycle.

Split Cycle (SCYC) is an active high output that indicates that the current bus cycle is part of a misaligned locked transfer. SCYC is defined for locked cycles only. A misaligned transfer is defined as any transfer that crosses an 8-byte boundary.

Write/Read (W/R#) distinguishes between write and read operations. When high, this signal indicates that the current bus cycle is a memory write, I/O write or a special bus cycle. When low, this signal indicates that the current cycle is a memory read, I/O read or interrupt acknowledge cycle.

Table 3-7. Bus Cycle Types

BUS CYCLE TYPE	M/IO#	D/C#	W/R#	CACHE#	LOCK#
Interrupt Acknowledge	0	0	0	1	0
Does not occur.	0	0	0	X	1
Does not occur.	0	0	1	X	0
Special Cycles: If BE(7-0)# = FEh: Shutdown If BE(7-0)# = FDh: Flush (INVD, WBINVD) If A4 = 0 and BE(7-0)# = FBh: Halt (HLT) If BE(7-0)# = F7h: Write-Back (WBINVD) If BE(7-0)# = EFh: Flush Acknowledge (FLUSH#) If A4 = 1 and BE(7-0)# = FBh: Stop Grant (SUSP#)	0	0	1	1	1
Does not occur.	0	1	X	X	0
I/O Data Read	0	1	0	1	1
I/O Data Write	0	1	1	1	1
Does not occur.	1	0	X	X	0
Cacheable Memory Code Read (Burst Cycle if KEN# Returned Active)	1	0	0	0	1
Non-cacheable Memory Code Read	1	0	0	1	1
Does not occur.	1	0	1	X	1
Locked Memory Data Read	1	1	0	1	0
Cacheable Memory Data Read (Burst Cycle if KEN# Returned Active)	1	1	0	0	1
Non-cacheable Memory Data Read	1	1	0	1	1
Locked Memory Write	1	1	1	1	0
Burst Memory Write (Writeback or Line Replacement)	1	1	1	0	1*
Single Transfer Memory Write	1	1	1	1	1

Note: X = Don't Care
*Note: LOCK# continues to be asserted during a write-back cycle that occurs following an aborted (BOFF# asserted) locked bus cycle.

3.2.8 Bus Cycle Control

The bus cycle control signals (ADS#, ADSC#, BRDY#, BRDYC#, NA#, and SMIACT#) indicate the beginning of a bus cycle and allow system hardware to control bus cycle termination timing and address pipelining.

Address Strobe (ADS#) is an active low output which indicates that the CPU has driven a valid address and bus cycle definition on the appropriate output pins. ADS# floats during bus hold states.

Cache Address Strobe (ADSC#) performs the same function as ADS#. ADSC# is used to interface directly to a secondary cache controller.

Burst Ready (BRDY#) is an active low input that is driven by the system to indicate that the current transfer within a burst cycle or the current single transfer bus cycle can be terminated. The CPU samples BRDY# in the second and subsequent clocks of a cycle. BRDY# is active during address hold states.

Cache Burst Ready (BRDYC#) performs the same function as BRDY# and is logically ORed with BRDY internally by the CPU. BRDYC# is used to interface directly to a secondary cache controller.

Next Address (NA#) is an active low input that is driven by the system to request the next pending bus cycle address and cycle definition information even though all data transfers for the current bus cycle are not complete. This new bus cycle is referred to as a "pipelined" cycle. If either the current or next bus cycle is a locked cycle, a line replacement, a write-back cycle or there is no pending bus cycle, the 6x86MX CPU does not start a pipelined bus cycle regardless of the state of the NA# input.

System Management Mode Active (SMIACT#) behaves in one of two ways depending on which SMM mode is in effect.

In SL-Compatible Mode, SMIACT# is an active low output which indicates that the CPU is operating in System Management Mode. SMIACT# is asserted in response to the assertion of SMI# or due to execution of SMINT instruction. SMIACT# is also asserted during accesses to define SMM memory if SMAC bit CCR1 is set. The SMAC bit allows access to SMM memory while not in SMM mode and typically used for initialization purposes.

While in SL-compatible mode, when servicing an SMI# interrupt or SMINT instruction, SMIACT# remains asserted until a RSM instruction is executed. The RSM instruction causes the 6x86MX CPUT to exit SMM mode and negate the SMIACT# output. If a cache inquiry cycle occurs while SMIACT# is active, any resulting write-back cycle is issued with SMIACT# asserted. This occurs even thought the write-back cycle is intended for normal memory rather than SMM memory.

In Cyrix Enhanced Mode, SMIACT# does not indicate that the CPU is operating in system management mode. In Cyrix Enhanced Mode, SMIACT# is asserted for every SMM memory bus cycle and negated for every non-SMM memory cycle. In this mode SMIACT# follows the timing of MIO# and W/R#.

During RESET, the USE_SMI bit in CCR1 is cleared. While USE_SMI is zero, SMIACT# is always negated. SMIACT# does not float during bus hold states, except during Cyrix Enhanced SMM Operations.

3.2.9 Interrupt Control

The interrupt control signals (INTR, NMI, SMI#) allow the execution of the current instruction stream to be interrupted and suspended.

Maskable Interrupt Request (INTR) is an active high level-sensitive input which causes the processor to suspend execution of the current instruction stream and begin execution of an interrupt service routine. The INTR input can be masked (ignored) through the IF bit in the Flags Register.

When not masked, the 6x86MX CPU responds to the INTR input by performing two locked interrupt acknowledge bus cycles. During the second interrupt acknowledge cycle, the 6x86MX CPU reads the interrupt vector (an 8-bit value), from the data bus. The 8-bit interrupt vector indicates the interrupt level that caused generation of the INTR and is used by the CPU to determine the beginning address of the interrupt service routine. To assure recognition of the INTR request, INTR must remain active until the start of the first interrupt acknowledge cycle.

Non-Maskable Interrupt Request (NMI) is a rising edge sensitive input which causes the processor to suspend execution of the current instruction stream and begin execution of an NMI interrupt service routine. The NMI interrupt cannot be masked by the IF bit in the Flags Register. Asserting NMI causes an interrupt which internally supplies interrupt vector 2h to the CPU core. Therefore, external interrupt acknowledge cycles are not issued.

Once NMI processing has started, no additional NMIs are processed until an IRET instruction is executed, typically at the end of the NMI service routine. If NMI is re-asserted prior to execution of the IRET, one and only one NMI rising edge is stored and then processed after execution of the next IRET.

System Management Interrupt Request (SMI#) is an interrupt input with higher priority than the NMI input. Asserting SMI# forces the processor to save the CPU state to SMM memory and to begin execution of the SMI service routine.

SMI# behaves one of two ways depending on the 6x86MX's SMM mode.

In SL-compatible mode SMI# is a falling edge sensitive input and is sampled on every rising edge of the processor input clock. Once SMI# servicing has started, no additional SMI# interrupts are processed until a RSM instruction is executed. If SMI# is reasserted prior to execution of a RSM instruction, one and only one SMI# falling edge is stored and then processed after execution of the next RSM.

In Cyrix enhanced SMM mode, SMI# is level sensitive, and nested SMI's are permitted under control of the SMI service routine. As a level sensitive input, software can process all SMI interrupts until all sources in the chipset have cleared. In enhanced mode, SMIACT# is asserted for every SMM memory bus cycle and negated for every non-SMM bus cycle.

In either mode, SMI# is ignored following reset and recognition is enabled by setting the USE_SMI bit in CCR1.

3.2.10 Cache Control

The cache control signals (EWBE#, FLUSH#, KEN#, PCD, PWT, WB/WT#) are used to indicate cache status and control caching activity.

External Write Buffer Empty (EWBE#) is an active low input driven by the system to indicate when there are no pending write cycles in the external system. The 6x86MX CPU samples EWBE# during write cycles (I/O and memory) only. If EWBE# is not asserted, the processor delays all subsequent writes to on-chip cache lines in the "exclusive" or "modified" state until EWBE# is asserted. Regardless of the state of EWBE#, all writes to the on-chip cache are delayed until any previously issued external write cycle is complete. This ensures that external write cycles occur in program order and is referred to as "strong write ordering". To enhance performance, "weak write ordering" may be allowed for specific address regions using the Address Region Registers (ARRs) and Region Control Registers (RCRs).

Cache Flush (FLUSH#) is a falling edge sensitive input that forces the processor to write-back all dirty data in the cache and then invalidate the entire cache contents. FLUSH# need only be asserted for a single clock but must meet specified setup and hold times to guarantee recognition at a particular clock edge.

Once FLUSH# is sampled active, the 6x86MX CPU begins the cache flush sequence after completion of the current instruction. External interrupts and additional FLUSH# requests are ignored while the cache flush is in progress. However, cache inquiry cycles are permitted during the flush sequence. The 6x86MX CPU issues a special flush acknowledge cycle to indicate completion of the flush sequence. If the processor is in a halt or shutdown state, FLUSH# is recognized and the 6x86MX CPU returns to the halt or shutdown state following completion of the flush sequence. If FLUSH# is active at the falling edge of RESET, the processor enters three state test mode.

Cache Enable (KEN#) is an active low input which indicates that the data being returned during the current cycle is cacheable. When the 6x86MX CPU is performing a cacheable code fetch or memory data read cycle and KEN# is sampled asserted, the cycle is transformed into a cache line fill (4 transfer burst cycle) or a "1+4" cache line fill. KEN# is sampled with the first asserted BRDY# or NA# for the cycle. I/O accesses, locked reads, system management memory accesses and interrupt acknowledge cycles are never cached.

Page Cache Disable (PCD) is an active high output that reflects the state of the PCD page attribute bit in the page table entry or the directory table entry. If paging is disabled or for cycles that are not paged, the PCD pin is driven low. PCD is masked by the cache disable (CD) bit in CR0 (driven high if CD=1) and floats during bus hold states.

Page Write Through (PWT) is an active high output that reflects the state of the PWT page attribute bit in the page table entry or the directory table entry. During non-paging cycles, and while paging is disabled the PWT pin is driven low. If PWT is asserted, PWT takes priority over the WB/WT# input. If PWT is asserted for either reads or writes, the cache line is saved in, or remains in, the shared (write-through) state. PWT floats during bus hold states.

The **Write-Back/Write-Through (WB/WT#)** input allows the system to define the write policy of the on-chip cache on a line-by-line basis. If WB/WT# is sampled high during a line fill cycle and PWT is low, the line is defined as write-back and is stored in the exclusive state. If WB/WT# is sampled high during a write to a write-through cache line (shared state) and PWT is low, the line is transitioned to write-back (exclusive state). If WB/WT# is sampled low or PWT is high, the line is defined as write-through and is stored in (line fill), or remains in (write), the shared state. Table 3-8 (Page 3-16) lists the effects of WB/WT# on the state of the cache line for various bus cycles.

Table 3-8. Effects of WB/WT# on Cache Line State

BUS CYCLE TYPE	PWT	WB/ WT#	WRITE POLICY	MESI STATE
Line Fill	0	0	Write-through	Shared
Line Fill	0	1	Write-back	Exclusive
Line Fill	1	x	Write-through	Shared
Memory Write (Note)	0	0	Write-through	Shared
Memory Write (Note)	0	1	Write-back	Exclusive
Memory Write (Note)	1	x	Write-through	Shared

Note: Only applies to memory writes to addresses that are currently valid in the cache.

3.2.11 Bus Arbitration

The bus arbitration signals (BOFF#, BREQ, HOLD, and HLDA) allow the 6x86MX CPU to relinquish control of its local bus when requested by another bus master device. Once the processor has released its bus, the bus master device can then drive the local bus signals.

Back-Off (BOFF#) is an active low input that forces the 6x86MX CPU to abort the current bus cycle and relinquish control of the CPU's local bus in the next clock. The 6x86MX CPU responds to BOFF# by entering the bus hold state as listed in Table 3-9 (Page 3-17). The 6x86MX CPU remains in bus hold until BOFF# is negated. Once BOFF# is negated, the 6x86MX CPU restarts any aborted bus cycle in its entirety. Any data returned to the 6x86MX CPU while BOFF# is asserted is ignored. If BOFF# is asserted in the same clock that ADS# is asserted, the 6x86MX CPU may float ADS# while in the active low state.

Bus Request (BREQ) is an active high output asserted by the 6x86MX CPU whenever a bus cycle is pending internally. The 6x86MX CPU always asserts BREQ in the first clock of a bus cycle with ADS# as well as during bus hold and address hold states if a bus cycle is pending. If no additional bus cycles are pending, BREQ is negated prior to termination of the current cycle.

Bus Hold Request (HOLD) is an active high input used to indicate that another bus master requests control of the CPU's local bus. After recognizing the HOLD request and completing the current bus cycle or sequence of locked bus cycles, the 6x86MX CPU responds by floating the local bus and asserting the hold acknowledge (HLDA) output. The bus remains granted to the requesting bus master until HOLD is negated. Once HOLD is sampled negated, the 6x86MX CPU simultaneously drives the local bus and negates HLDA.

Hold Acknowledge (HLDA) is an active high output used to indicate that the 6x86MX CPU has responded to the HOLD input and has relinquished control of its local bus. Table 3-9 (Page 3-17) lists the state of all the 6x86MX CPU signals during a bus hold state. The 6x86MX CPU continues to operate during bus hold states as long as the on-chip cache can satisfy bus requests. HLDA is asserted until HOLD is negated. Once HOLD is sampled negated, the 6x86MX CPU simultaneously drives the local bus and negates HLDA.

Table 3-9. Signal States During Bus Hold

SIGNAL LINE	STATE
A20M#	Recognized internally
A31-A3	Float
ADS#	Float
ADSC#	Float
AHOLD	Ignored
AP	Float
APCHK#	Driven
BE7#-BE0#	Float
BOFF#	Recognized
BRDY#	Ignored
BRDYC#	Ignored
BREQ	Driven
CACHE#	Float
D/C#	Float
D63-D0	Float
DP7-DP0	Float
EADS#	Recognized
EWBE#	Recognized internally
FERR#	Driven
FLUSH#	Recognized
HIT#	Driven
HITM#	Driven
HLDA	Responds to HOLD
HOLD	Recognized
IGNNE#	Recognized internally

SIGNAL LINE	STATE
INTR	Recognized internally
INV	Recognized
KEN#	Ignored
LOCK#	Float
M/IO#	Float
NA#	Ignored
NMI	Recognized internally
PCD	Float
PCHK#	Driven
PWT	Float
RESET	Recognized
SCYC	Float
SMI#	Recognized
SMIACT#	Driven
SUSP#	Recognized
SUSPA#	Driven
TCK	Recognized
TDI	Recognized
TDO	Responds to TCK, TDI, TMS, TRST#
TMS	Recognized
TRST#	Recognized
W/R#	Float
WB/WT#	Ignored
WM_RST	Recognized

3.2.12 Cache Coherency

The cache coherency signals (AHOLD, EADS#, HIT#, HITM#, and INV) are used to initiate and monitor cache inquiry cycles. These signals are intended to be used to ensure cache coherency in a uni-processor environment only. Contact Cyrix for additional specifications on maintaining coherency in a multi-processor environment.

Address Hold Request (AHOLD) is an active high input which forces the 6x86MX CPU to float A31-A3 and AP in the next clock cycle. While AHOLD is asserted, only the address bus is disabled. The current bus cycle remains active and can be completed in the normal fashion. The 6x86MX CPU does not generate additional bus cycles while AHOLD is asserted except write-back cycles in response to a cache inquiry cycle.

External Address Strobe (EADS#) is an active low input used to indicate to the 6x86MX CPU that a valid cache inquiry address is being driven on the 6x86MX CPU address bus (A31-A5) and AP. The 6x86MX CPU checks the on-chip cache for this address. If the address is present in the cache the HIT# signal is asserted. If the data associated with the inquiry address is "dirty" (modified state), the HITM# signal is also asserted. If dirty data exists, a write-back cycle is issued to update external memory with the dirty data. Additional cache inquiry cycles are ignored while HITM# is asserted.

The state of the INV pin at the time EADS# is sampled active determines the final state of the cache line. If INV is sampled high, the final state of the cache line is "invalid". If INV is sampled low, the final state of the cache line is "shared". A cache inquiry cycle using EADS# may be run while the 6x86MX CPU is in either an address hold or bus hold state. The inquiry address must be driven by an external device.

Hit on Cache Line (HIT#) is an active low output used to indicate that the current cache inquiry address has been found in the cache (modified, exclusive or shared states). HIT# is valid two clocks after EADS# is sampled active, and remains valid until the next cache inquiry cycle.

Hit on Modified Data (HITM#) is an active low output used to indicate that the current cache inquiry address has been found in the cache and dirty data exists in the cache line (modified state). If HITM# is asserted, a write-back cycle is issued to update external memory. HITM# is valid two clocks after EADS# is sampled active, and remains asserted until two clocks after the last BRDY# of the write-back cycle is sampled active. The 6x86MX CPU does not accept additional cache inquiry cycles while HITM# is asserted.

Invalidate Request (INV) is an active high input used to determine the final state of the cache line in the case of a cache inquiry hit. INV is sampled with EADS#. A logic one on INV directs the processor to change the state of the cache line to "invalid". A logic zero on INV

directs the processor to change the state of the cache line to "shared".

3.2.13 FPU Error Interface

The FPU interface signals FERR# and IGNNE# are used to control error reporting for the on-chip floating point unit. These signals are typically used for a PC-compatible system implementation. For other applications, FPU errors are reported to the 6x86MX CPU core through an internal interface.

Floating Point Error Status (FERR#) is an active low output asserted by the 6x86MX CPU when an unmasked floating point error occurs. FERR# is asserted during execution of the FPU instruction that caused the error. FERR# does not float during bus hold states.

Ignore Numeric Error (IGNNE#) is an active low input which forces the 6x86MX CPU to ignore any pending unmasked FPU errors and allows continued execution of floating point instructions. When IGNNE# is not asserted and an unmasked FPU error is pending, the 6x86MX CPU only executes the following floating point instructions: FNCLEX, FNINIT, FNSAVE, FNSTCW, FNSTENV, and FNSTSW#. IGNNE# is ignored when the NE bit in CR0 is set to a 1.

3.2.14 Power Management Interface

The two power management signals (SUSP#, SUSPA#) allow the 6x86MX CPU to enter and exit suspend mode. The 6x86MX CPU also enters suspend mode as the result of executing a HALT instruction if the HALT bit is set in CCR2. Suspend mode circuitry forces the 6x86MX CPU to consume minimal power while maintaining the entire internal CPU state.

Suspend Request (SUSP#) is an active low input which requests that the 6x86MX CPU enter suspend mode. After recognition of an active SUSP# input, the 6x86MX CPU completes execution of the current instruction, any pending decoded instructions and associated bus cycles, issues a stop grant bus cycle, and then asserts the SUSPA# output. SUSP# is ignored following RESET and is enabled by setting the SUSP bit in CCR2.

The **Suspend Acknowledge (SUSPA#)** output indicates that the 6x86MX CPU has entered low-power suspend mode as the result of either assertion of SUSP# or execution of a HALT instruction. SUSPA# remains asserted

until SUSP# is negated, or until an interrupt is serviced if suspend mode was entered via the HALT instruction. If SUSP# is asserted and then negated prior to SUSPA# assertion, SUSPA# may toggle state after SUSP# negates.

The 6x86MX CPU accepts cache flush requests and cache inquiry cycles while SUSPA# is asserted. If FLUSH# is asserted, the CPU exits the low power state and services the flush request. After completion of all required write-back cycles, the CPU returns to the low power state. SUSPA# negates during the write-back cycles. Before issuing the write-back cycle, the CPU may execute several code fetches.

If AHOLD, BOFF# or HOLD is asserted while SUSPA# is asserted, the CPU exits the low power state in preparation for a cache inquiry cycle. After completion of any required write-back cycles resulting from the cache inquiry, the CPU returns to the low power state only if HOLD, BOFF# and AHOLD are negated. SUSPA# negates during the write-back cycle.

Table 3-10 (Page 3-21) lists the 6x86MX CPU signal states for suspend mode when initiated by either SUSP# or the HALT instruction. SUSPA# is disabled (three-state) following RESET and is enabled by setting the SUSP bit in CCR2.

3.2.15 Performance Monitoring

The PM0 and PM1 pins are outputs that are associated with performance monitoring. These pins can be defined in two different ways.

If PM0, bit 9 in the Counter Event Control Register is set, the PM0 pin indicates an overflow has occurred; if reset, the PM0 pin indicates that a performance counter event has occurred. The PM1 pin operates in the same manner, but is controlled by PM1, bit 25.

The PM0 and PM1 pins indicate only that an event or overflow occurred at least once. More than one event or overflow can occur in the same CPU or external clock cycle.

Table 3-10. Signal States During Suspend Mode

SIGNAL LINE	SUSP# INITIATED/ HALT INITIATED
A20M#	Ignored
A31-A3	Driven
ADS#	1
ADSC#	1
AHOLD	Recognized
AP	Driven
APCHK#	1
BE7#-BE0#	Driven
BOFF#	Recognized
BRDY#	Ignored
BRDYC#	Ignored
BREQ	0
CACHE#	Driven
D/C#	Driven
D63-D0	Float
DP7-DP0	Float
EADS#	Recognized
EWBE#	Ignored
FERR#	1
FLUSH#	Recognized
HIT#	Driven
HITM#	1
HLDA	Driven in response to HOLD
HOLD	Recognized
IGNNE#	Ignored

SIGNAL LINE	SUSP# INITIATED/ HALT INITIATED
INTR	Latched/Recognized
INV	Recognized
KEN#	Ignored
LOCK#	1
M/IO#	Driven
NA#	Ignored
NMI	Latched/Recognized
PCD	Driven
PCHK#	1
PWT	Driven
RESET	Recognized
SCYC	Driven
SMI#	Latched/Recognized
SMIACT#	1
SUSP#	0 / Recognized
SUSPA#	0
TCK	Recognized
TDI	Recognized
TDO	Responds to TCK, TDI, TMS, TRST#
TMS	Recognized
TRST#	Recognized
W/R#	Driven
WB/WT#	Ignored
WM_RST	Latched/Recognized

3.2.16 JTAG Interface

The 6x86MX CPU can be tested using JTAG Interface (IEEE Std. 1149.1) boundary scan test logic. The 6x86MX CPU pin state can be set according to serial data supplied to the chip. The 6x86MX CPU pin state can also be recorded and supplied as serial data.

Test Clock (TCK) is the clock input used by the 6x86MX CPU boundary scan (JTAG) test logic. The rising edge of TCK is used to clock control and data information into the 6x86MX processor using the TMS and TDI pins. The falling edge of TCK is used to clock data information out of the 6x86MX processor using the TDO pin.

Test Data Input (TDI) is the serial data input used by the 6x86MX CPU boundary scan (JTAG) test logic. TDI is sampled on the rising edge of TCK.

Test Data Output (TDO) is the serial data output used by the 6x86MX CPU boundary scan (JTAG) test logic. TDO is output on the falling edge of TCK.

Test Mode Select (TMS) is the control input used by the 6x86MX CPU boundary scan (JTAG) test logic. TMS is sampled on the rising edge of TCK.

Test Reset (TRST#) is an active low input used to initialize the 6x86MX CPU boundary scan (JTAG) test logic.

3.3 Functional Timing

3.3.1 Reset Timing

Figure 3-2 illustrates the required RESET timing for both a power-on reset and a reset that occurs during operation. The WM_RST and FLUSH# inputs are sampled at the falling edge of RESET to determine if the 6x86MX CPU should enter built-in self-test, enable tree-state test mode or enable the scatter-gather interface pins, respectively. WM_RST and FLUSH# must be valid at least two clocks prior to the RESET falling edge.

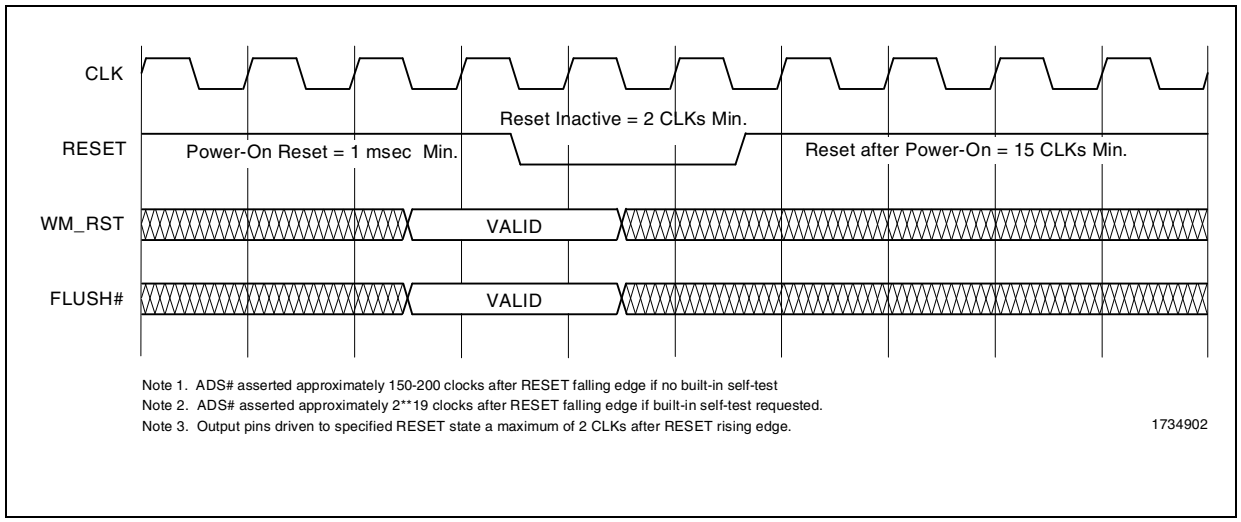


Figure 3-2. RESET Timing

3.3.2 Bus State Definition

The 6x86MX CPU bus controller supports non-pipelined and pipelined operation as well as single transfer and burst bus cycles. During each CLK period, the bus controller exists in one of six states as listed in Table 3-11. Each of bus state and its associated state transitions are illustrated in Figure 3-3, (Page 3-25) and listed in Table 3-12, (Page 3-26).

Table 3-11. 6x86MX CPU Bus States

STATE	NAME	DESCRIPTION
Ti	Idle Clock	During Ti, no bus cycles are in progress. BOFF# and RESET force the bus to the idle state. The bus is always in the idle state while HLDA is active.
T1	First Bus Cycle Clock	During the first clock of a non-pipelined bus cycle, the bus enters the T1 state. ADS# is asserted during T1 along with valid address and bus cycle definition information.
T2	Second and Subsequent Bus Cycle Clock	During the second clock of a non-pipelined bus cycle, the bus enters the T2 state. The bus remains in the T2 state for subsequent clocks of the bus cycle as long as a pipelined cycle is not initiated. During T2, valid data is driven during write cycles and data is sampled during reads. BRDY# is also sampled during T2. The bus also enters the T2 state to complete bus cycles that were initiated as pipelined cycles but complete as the only outstanding bus cycle.
T12	First Pipelined Bus Cycle Clock	During the first clock of a pipelined cycle, the bus enters the T12 state. During T12, data is being transferred and BRDY# is sampled for the current cycle at the same time that ADS# is asserted and address/bus cycle definition information is driven for the next (pipelined) cycle.
T2P	Second and Subsequent Pipelined Bus Cycle Clock	During the second and subsequent clocks of a pipelined bus cycle where two cycles are outstanding, the bus enters the T2P state. During T2P, data is being transferred and BRDY# is sampled for the current cycle. However, valid address and bus cycle definition information continues to be driven for the next pipelined cycle.
Td	Dead Clock	The bus enters the Td state if a pipelined cycle was initiated that requires one idle clock to turn around the direction of the data bus. Td is required for a read followed immediately by a pipelined write, and for a write followed immediately by a pipelined read.

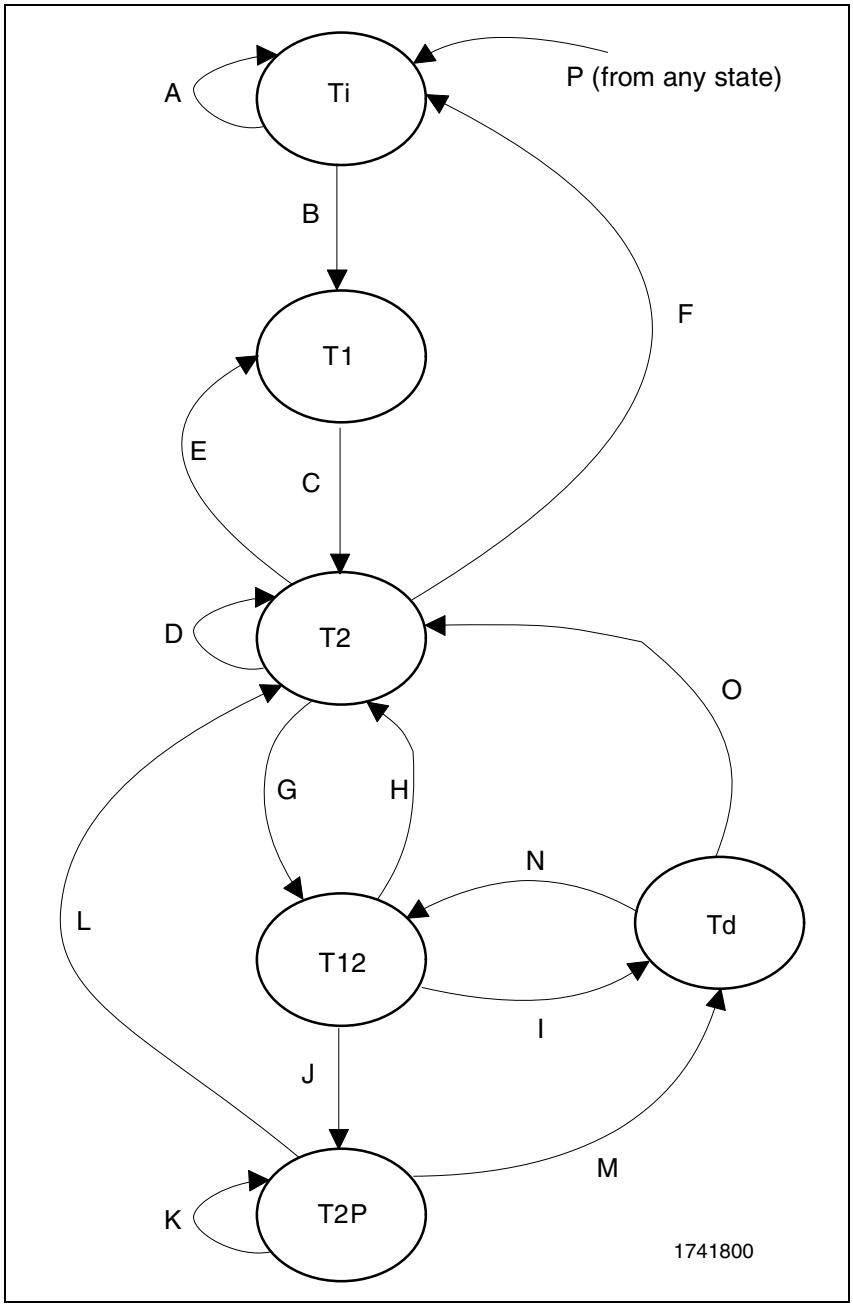


Figure 3-3. 6x86MX CPU Bus State Diagram

Table 3-12. Bus State Transitions

TRANSITION	CURRENT STATE	NEXT STATE	EQUATION
A	Ti	Ti	No Bus Cycle Pending.
B	Ti	T1	New or Aborted Bus Cycle Pending.
C	T1	T2	Always.
D	T2	T2	Not Last BRDY# and No New Bus Cycle Pending, or Not Last BRDY# and New Bus Cycle Pending and NA# Negated.
E	T2	T1	Last BRDY# and New Bus Cycle Pending and HITM# Negated.
F	T2	Ti	Last BRDY# and No New Bus Cycle Pending, or Last BRDY# and HITM# Asserted.
G	T2	T12	Not Last BRDY# and New Bus Cycle Pending and NA# Sampled Asserted.
H	T12	T2	Last BRDY# and No Dead Clock Required.
I	T12	Td	Last BRDY# and Dead Clock Required.
J	T12	T2P	Not Last BRDY#.
K	T2P	T2P	Not Last BRDY#.
L	T2P	T2	Last BRDY# and No Dead Clock Required.
M	T2P	Td	Last BRDY# and Dead Clock Required.
N	Td	T12	New Bus Cycle Pending and NA# Sampled Asserted.
O	Td	T2	No New Bus Cycle Pending, or New Bus Cycle Pending and NA# Negated.
P	Any State	Ti	RESET Asserted, or BOFF# Asserted.

3.3.3 Non-Pipelined Bus Cycles

Non-pipelined bus operation may be used for all bus cycle types. The term "non-pipelined" refers to a mode of operation where the CPU allows only one outstanding bus cycle. In other words, the current bus cycle must complete before a second bus cycle is allowed to start.

3.3.3.1 Non-Pipelined Single Transfer Cycles

Single transfer read cycles occur during non-cacheable memory reads, I/O read cycles, and special cycles. A non-pipelined single transfer read cycle begins with address and bus cycle definition information driven on the bus during the first clock (T1 state) of the bus cycle. The CPU then monitors the BRDY# input at the end of the second clock (T2 state). If BRDY# is asserted, the CPU reads the appropriate data and data parity lines and terminates the bus cycle. If BRDY# is not active, the CPU continues to sample the BRDY# input at the end of each subsequent cycle (T2 states). Each of the additional clocks is referred to as a wait state.

The CPU uses the data parity inputs to check for even parity on the active data lines. If the CPU detects an error, the parity check output (PCHK#) asserts during the second clock following the termination of the read cycle.

Figure 3-4 (Page 3-28) illustrates the functional timing for two non-pipelined single--transfer read cycles. Cycle 2 is a potentially cacheable cycle as indicated by the CACHE# output. Because this cycle is potentially cacheable, the CPU samples the KEN# input at the same clock edge that BRDY# is asserted. If KEN# is negated, the cycle terminates as shown in the diagram. If KEN# is asserted, the CPU converts this cycle into a burst cycle as described in the next section. NA# must be negated for non-pipelined operation. Pipelined bus cycles are described later in this chapter.

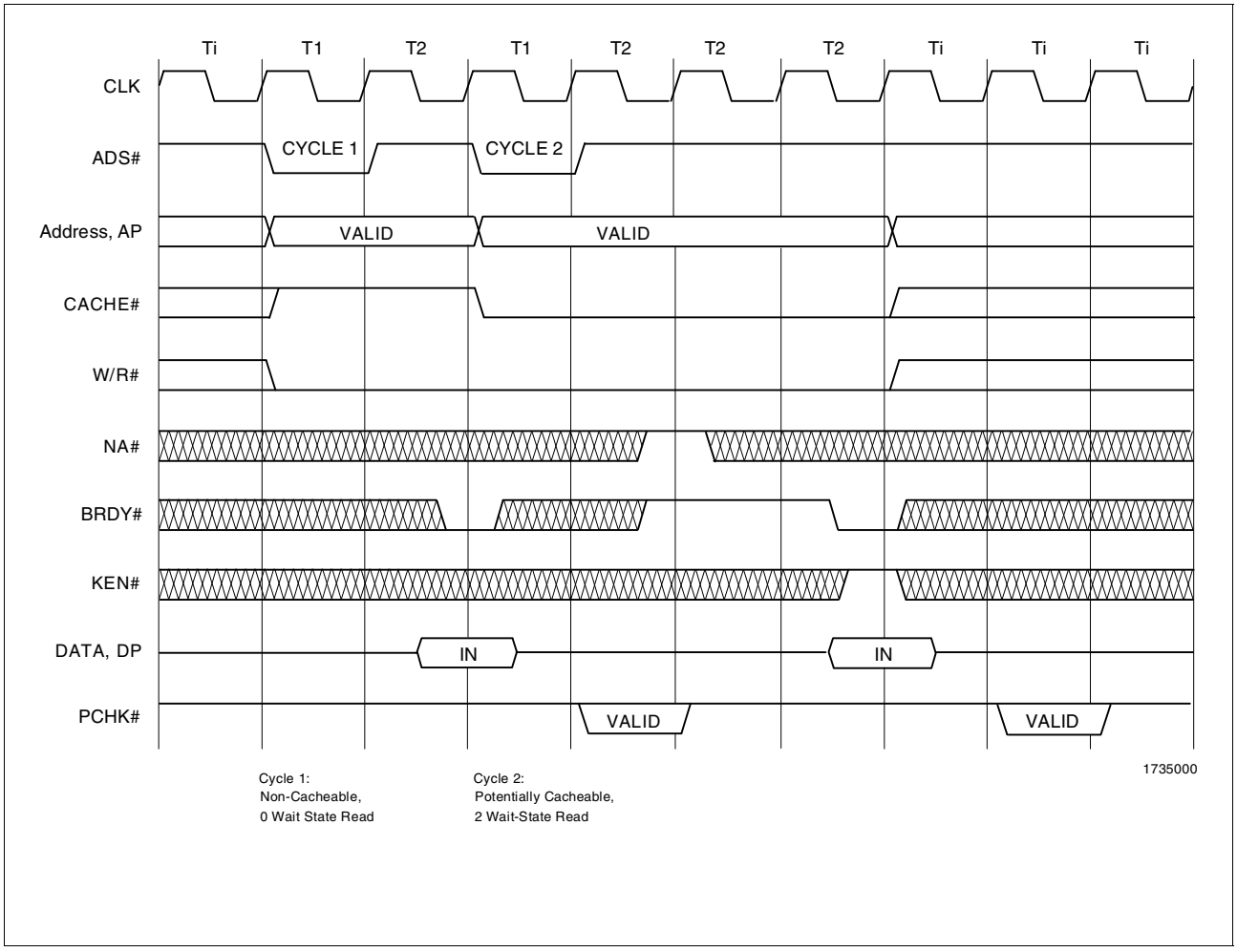


Figure 3-4. Non-Pipelined Single Transfer Read Cycles

Single transfer write cycles occur for writes that are neither line replacement nor write-back cycles. The functional timing of two non-pipelined single transfer write cycles is shown in Figure 3-5. During a write cycle, the data and data parity lines are outputs and are driven valid during the second clock (T2 state) of the bus cycle. Data and data parity remain valid during all wait states. If the write cycle is a write to a valid cache location in the "shared" state, the WB/WT# pin is sampled with BRDY#. If WB/WT# is sampled high, the cache line transitions from the "shared" to the "exclusive" state.

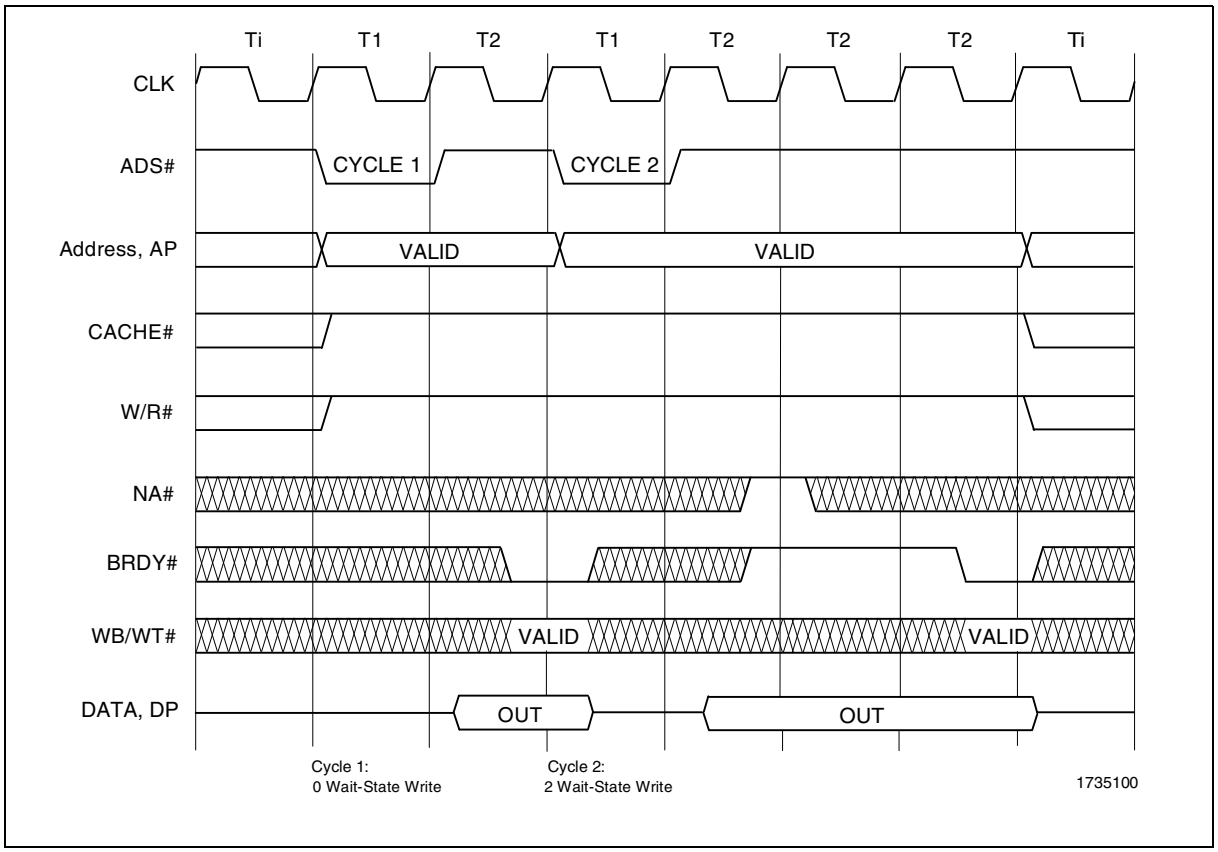


Figure 3-5. Non-Pipelined Single Transfer Write Cycles

3.3.3.2 Non-pipelined Burst Read Cycles

The 6x86MX CPU uses burst read cycles to perform cache line fills. During a burst read cycle, four 64-bit data transfers occur to fill one of the CPU's 32-byte internal cache lines. A non-pipelined burst read cycle begins with address and bus cycle definition information driven on the bus during the first clock (T1 state) of the bus cycle. The CACHE# output is always active during a burst read cycle and is driven during the T1 clock.

The CPU then monitors the BRDY# input at the end of the second clock (T2 state). If BRDY# is asserted, the CPU reads the data and data parity and also checks the KEN# input. If KEN# is negated, the CPU terminates the bus cycle as a single transfer cycle. If KEN# is asserted, the CPU converts the cycle into a burst (cache line fill) by continuing to sample BRDY# at the end of each subsequent clock. BRDY# must be asserted a total of four times to complete the burst cycle.

WB/WT# is sampled at the same clock edge as KEN#. In conjunction with PWT and the on-chip configuration registers, WB/WT# determines the MESI state of the cache line for the current line fill.

Each time BRDY# is sampled asserted during the burst cycle, a data transfer occurs. The CPU reads the data and data parity busses and assigns the data to an internally generated burst address. Although the CPU internally generates the burst address sequence, only the first address of the burst is driven on the external address bus. System logic must predict the burst address sequence based on the first address. Wait states may be added to any transfer within a burst by delaying the assertion of BRDY# by the desired number of clocks.

The CPU checks even data parity for each of the four transfers within the burst. If the CPU detects an error, the parity check output (PCHK#) asserts during the second clock following the BRDY# assertion of the data transfer.

Figure 3-6 (Page 3-31) illustrates two non--pipelined burst read cycles. The cycles shown are the fastest possible burst sequences (2-1-1-1). NA# must be negated for non-pipelined operation as shown in the diagram. Pipelined bus cycles are described later in this chapter.

Figure 3-7 (Page 3-32) depicts a burst read cycle with wait states. A 3-2-2-2 burst read is shown.

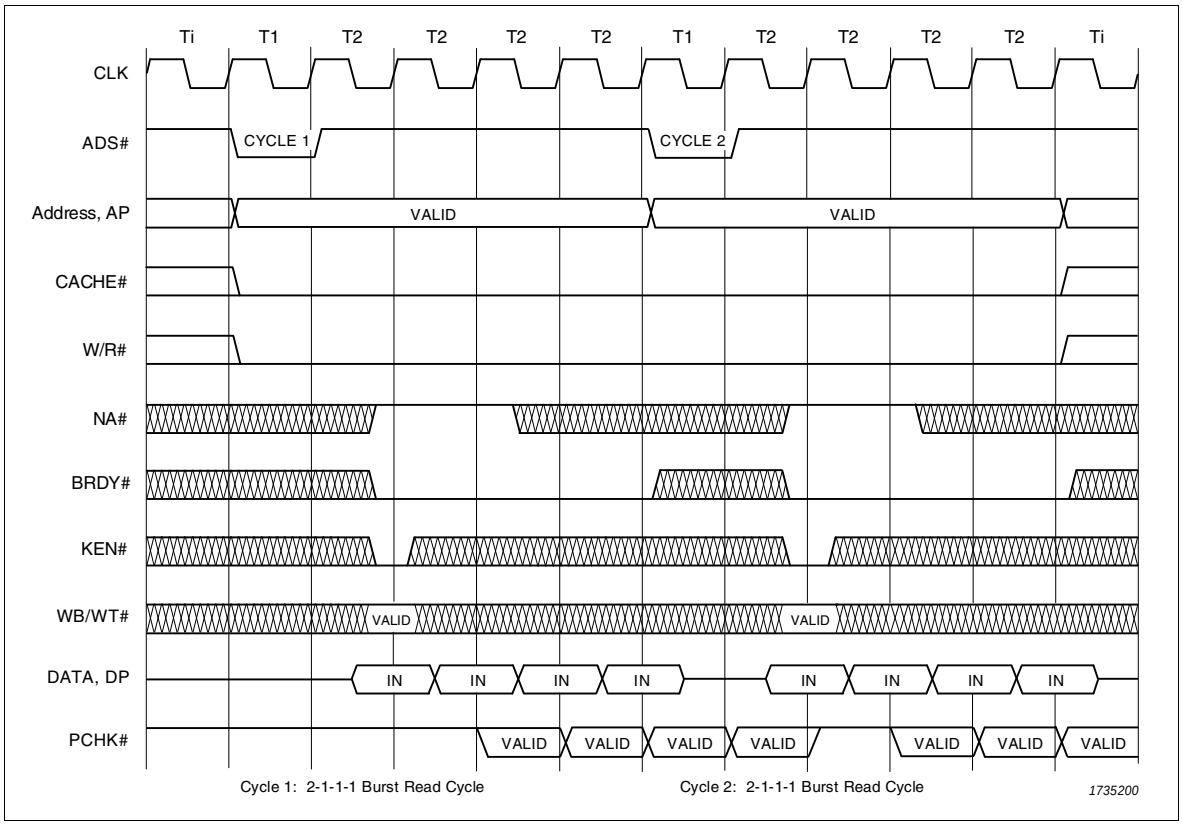


Figure 3-6. Non-Pipelined Burst Read Cycles

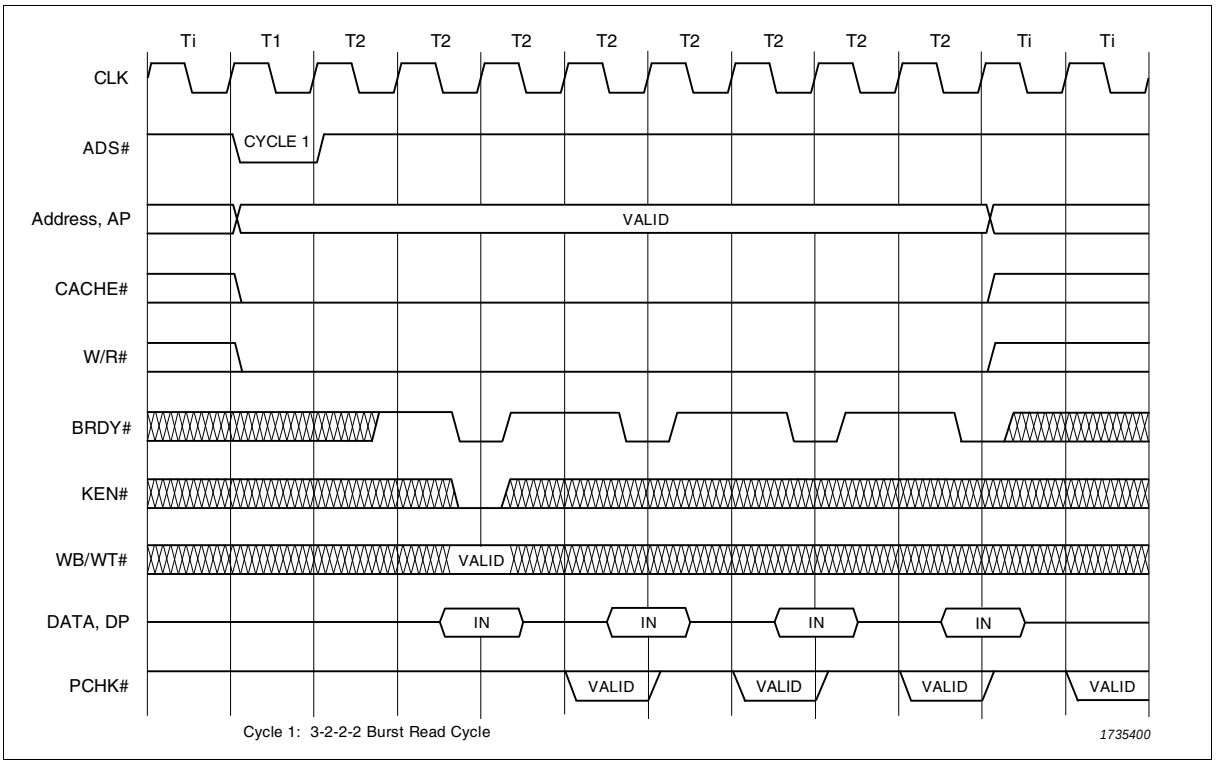


Figure 3-7. Burst Cycle with Wait States

Burst Cycle Address Sequence.

The 6x86MX CPU provides two different address sequences for burst read cycles. The 6x86MX CPU burst cycle address sequence modes are referred to as "1+4" and "linear". After reset, the CPU default mode is "1+4".

In "1+4" mode, the CPU performs a single transfer read cycle prior to the burst cycle, if the desired first address is (...xx8). During this single transfer read cycle, the CPU reads the critical data. In addition, the 6x86MX CPU samples the state of KEN#. If KEN# is active, the CPU then performs the burst cycle with the address sequence shown in Table 3-13 (Page 3-33). The 6x86MX CPU CACHE# output is not asserted during the single read cycle prior to the burst. Therefore, CACHE# must not be used to qualify the KEN# input to the processor. In addition, if KEN# is returned active for the "1" read cycle in the "1+4", all data bytes supplied to the CPU must be valid. The CPU samples WB/WT# during the "1" read cycle, and does not resample WB/WT# during the following burst cycle. Figure 3-8 (Page 3-33) illustrates a "1+4" burst read cycle.

Table 3-13. "1+4" Burst Address Sequences

BURST CYCLE FIRST ADDRESS	SINGLE READ CYCLE PRIOR TO BURST	BURST CYCLE ADDRESS SEQUENCE
0	None	0-8-10-18
8	Address 8	0-8-10-18
10	None	10-18-0-8
18	Address 18	10-18-0-8

.

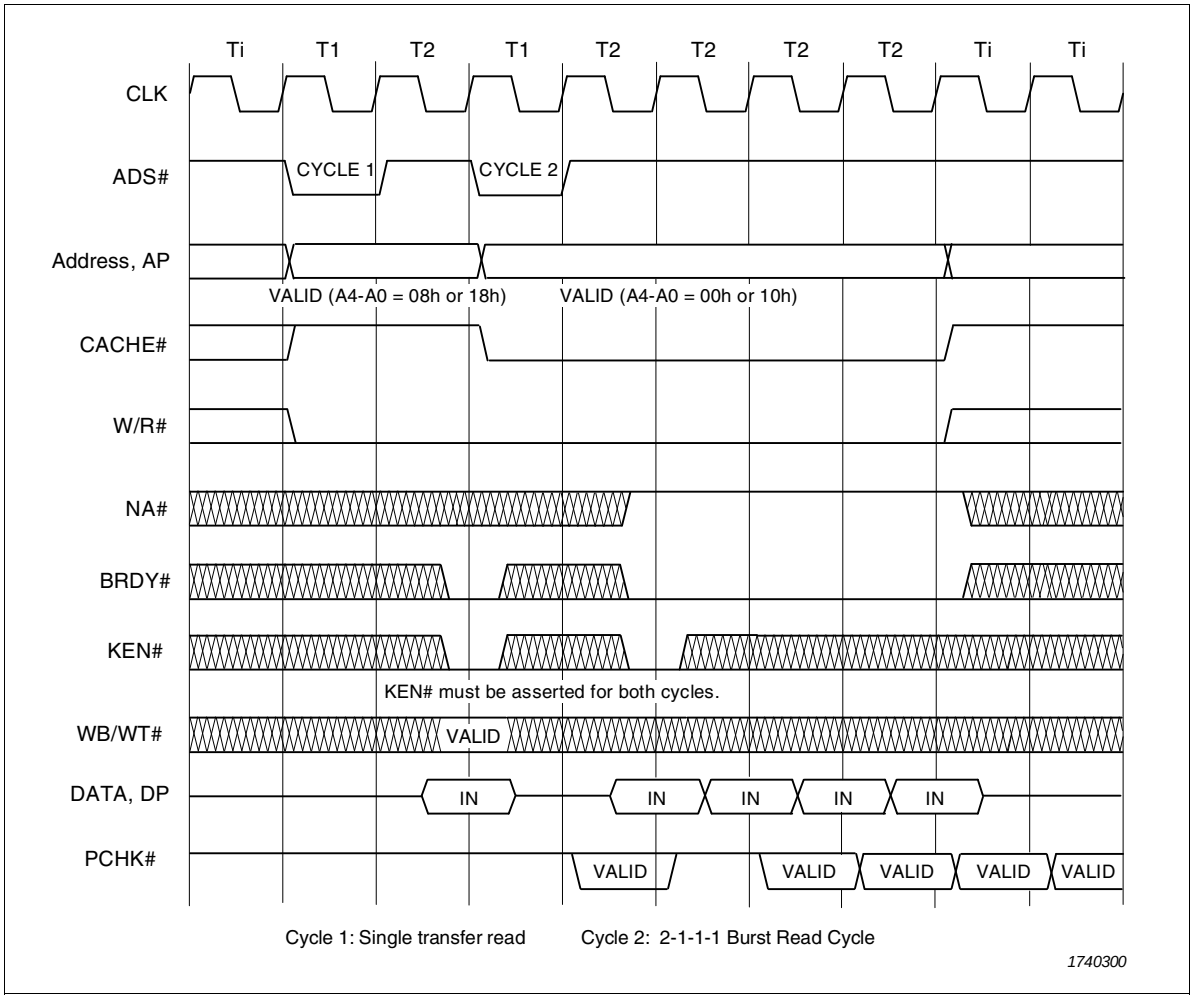


Figure 3-8. "1+4" Burst Read Cycle

The address sequences for the 6x86MX CPU's linear burst mode are shown in Table 3-14. Operating the CPU in linear burst mode minimizes processor bus activity resulting in higher system performance. Linear burst mode can be enabled through the 6x86MX CPU CCR3 configuration register.

Table 3-14. Linear Burst Address Sequences

BURST CYCLE FIRST ADDRESS	BURST CYCLE ADDRESS SEQUENCE
0	0-8-10-18
8	8-10-18-0
10	10-18-0-8
18	18-0-8-10

3.3.3.3 Burst Write Cycles

Burst write cycles occur for line replacement and write-back cycles. Burst writes are similar to burst read cycles in that the CACHE# output is asserted and four 64-bit data transfers occur. Burst writes differ from burst reads in that the data and data parity lines are outputs rather than inputs. Also, KEN# and WB/WT# are not sampled during burst write cycles.

Data and data parity for the first data transfer are driven valid during the second clock (T2 state) of the bus cycle. Once BRDY# is sampled asserted for the first data transfer, valid data and data parity for the second transfer are driven during the next clock cycle. The same timing relationship between BRDY# and data applies for the third and fourth data transfers as well. Wait states may be added to any transfer within a burst by delaying the assertion of BRDY# by the required number of clocks.

As on burst read cycles, only the first address of a burst write cycle is driven on the external address bus. System logic must predict the remaining burst address sequence based on the first address. Burst write cycles always begin with a first address ending in 0 (signals A4-A0=0) and follow an ascending address sequence for the remaining transfers (0-8-10-18).

Figure 3-9 illustrates two non-pipelined burst write cycles. The cycles shown are the fastest possible burst sequences (2-1-1-1). As shown, an idle clock always exists between two back-to-back burst write cycles. Therefore, the second burst write cycle in a pair of back-to-back burst writes is always issued as a non-pipelined cycle regardless of the state of the NA# input.

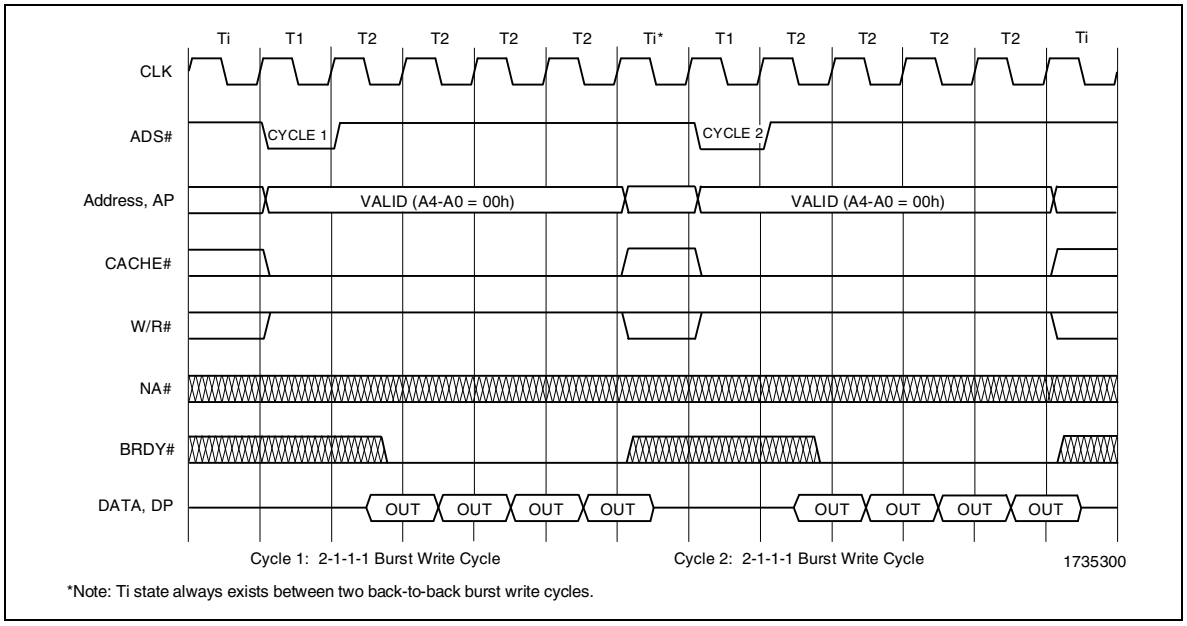


Figure 3-9. Non-Pipelined Burst Write Cycles

3.3.4 Pipelined Bus Cycles

Pipelined addressing is a mode of operation where the CPU allows up to two outstanding bus cycles at any given time. Using pipelined addressing, the address of the first bus cycle is driven on the bus. While the CPU waits for the data for the first cycle, the address for a second bus cycle is issued. Pipelined bus cycles occur for all cycle types except locked cycles and burst write cycles.

Pipelined cycles are initiated by asserting NA#. The CPU samples NA# at the end of each T2, T2P and Td state. KEN# and WB/WT# are sampled at either the same clock as NA# is active, or at the same clock as the first BRDY# for that cycle, whichever occurs first. The CPU issues the next address a minimum of two clocks after NA# is sampled asserted.

The CPU latches the state of the NA# pin internally. Therefore, even if a new bus cycle is not pending internally at the time NA# was sampled asserted, the CPU still issues a pipelined bus cycle if an internal bus request occurs prior to completion of the current bus cycle. Once NA# is sampled asserted, the state of NA# is ignored until the current bus cycle completes. If two cycles are outstanding and the second cycle is a read, the CPU samples KEN# and WB/WT# for the second cycle when NA# is sampled asserted.

Figure 3-10 and Figure 3-11 (Page 3-37) illustrate pipelined single transfer read cycles and pipelined burst read cycles, respectively.

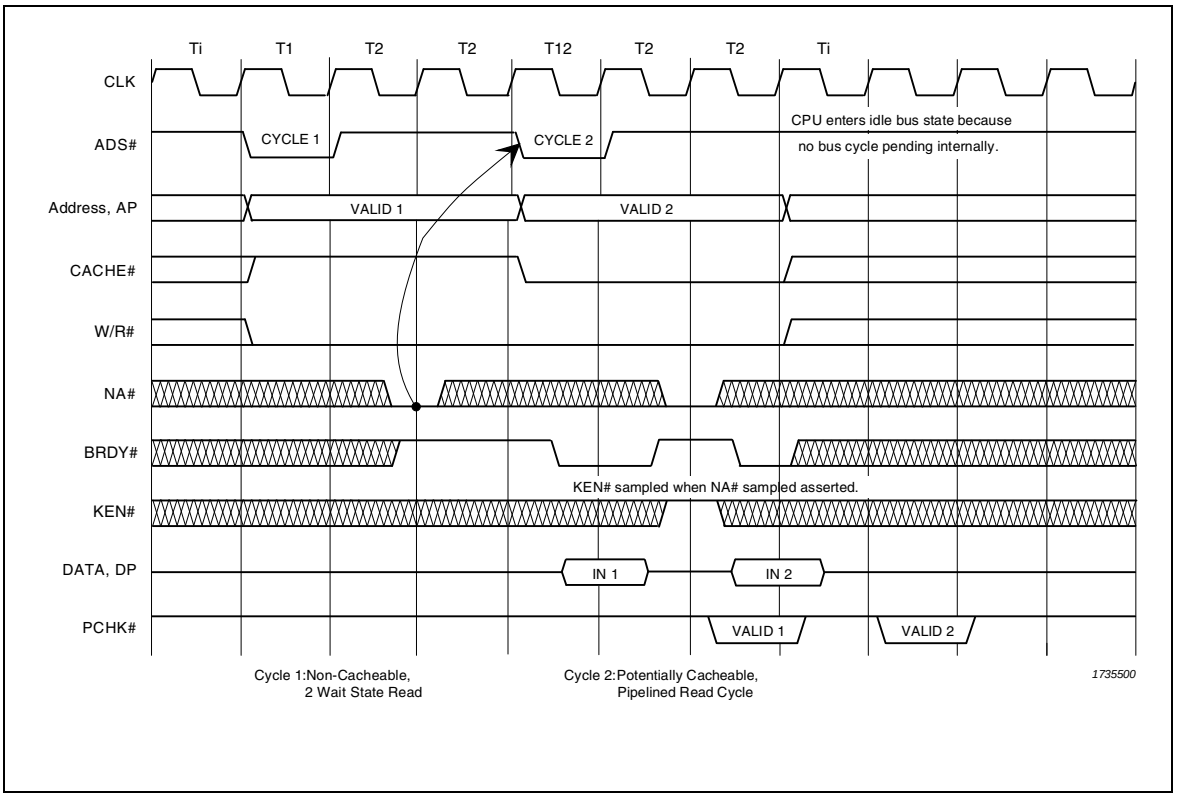


Figure 3-10. Pipelined Single Transfer Read Cycles

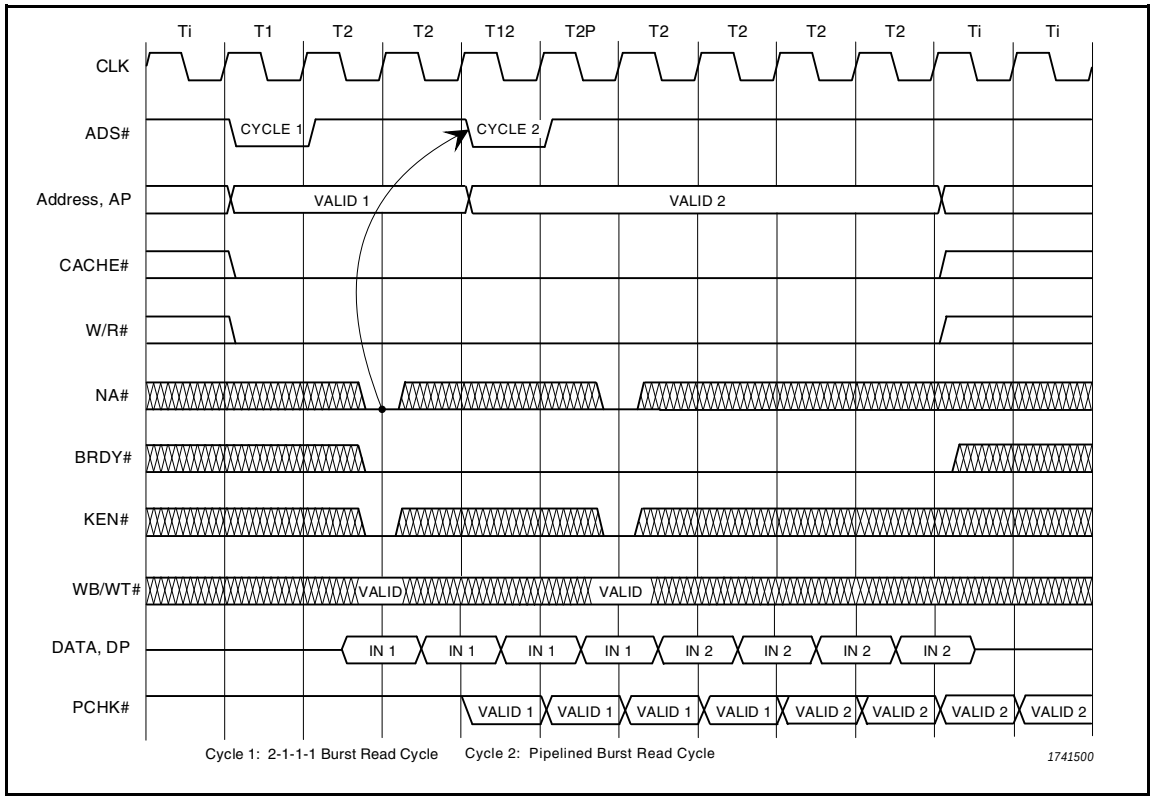


Figure 3-11. Pipelined Burst Read Cycles

3.3.4.1 Pipelined Back-to-Back Read/Write Cycles

Figure 3-12 depicts a read cycle followed by a pipelined write cycle. Under this condition, the data bus must change from an input for the read cycle to an output for the write cycle. In order to accomplish this transition without causing data bus contention, the CPU automatically inserts a "dead" (Td) clock cycle. During the Td state, the data bus floats. The CPU then drives the write data onto the bus in the following clock. The CPU also inserts a Td clock between a write cycle and a pipelined read cycle to allow the data bus to smoothly transition from an output to an input.

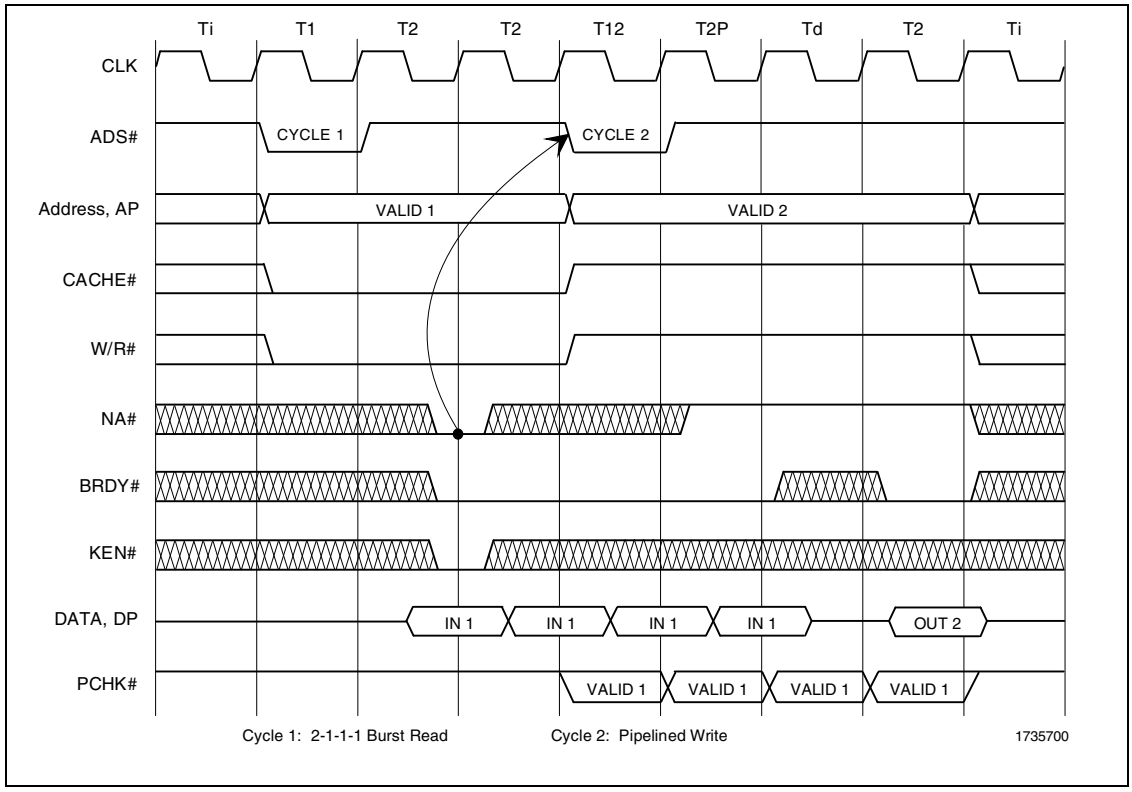


Figure 3-12. Read Cycle Followed by Pipelined Write Cycle

3.3.5 Interrupt Acknowledge Cycles

The CPU issues interrupt acknowledge bus cycles in response to an active INTR input. Interrupt acknowledge cycles are single transfer cycles and always occur in locked pairs as shown in Figure 3-13. The CPU reads the interrupt vector from the lower eight bits of the data bus at the completion of the second interrupt acknowledge cycle. Parity is not checked during the first interrupt acknowledge cycle.

M/IO#, D/C# and W/R# are always logic low during interrupt acknowledge cycles. Additionally, the address bus is driven with a value of 0000 0004h for the first interrupt acknowledge cycle and with a value of 0000 0000h for the second. A minimum of one idle clock always occurs between the two interrupt acknowledge cycles.

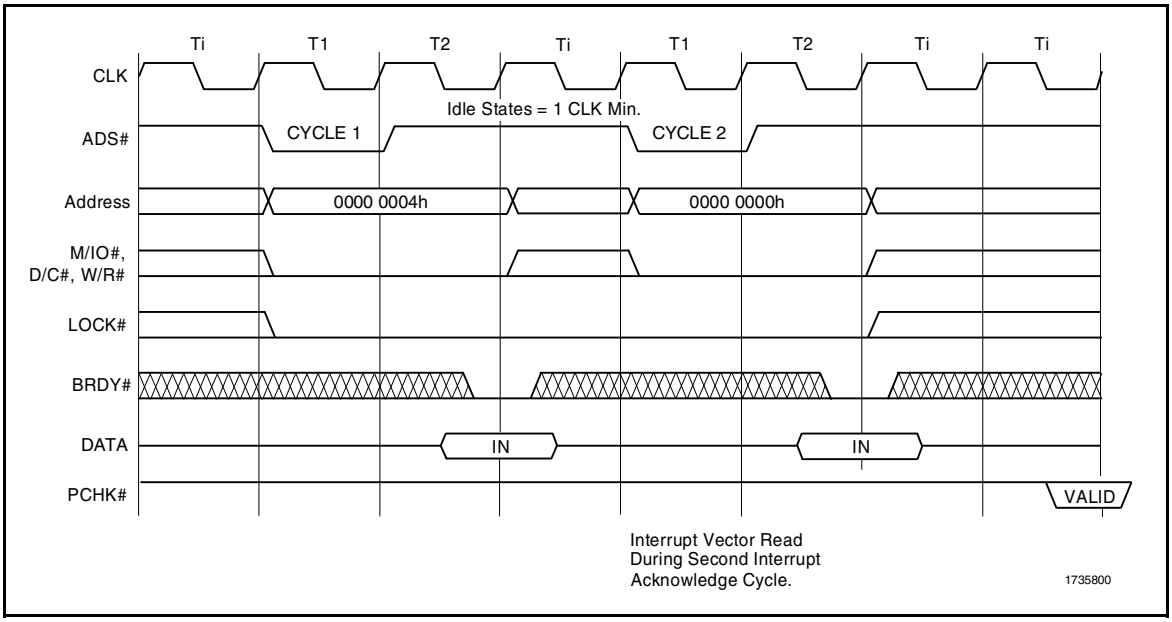


Figure 3-13. Interrupt Acknowledge Cycles

3.3.6 SMI# Interrupt Timing

The CPU samples the System Management Interrupt (SMI#) input at each clock edge. At the next appropriate instruction boundary, the CPU recognizes the SMI# and completes all pending write cycles. The CPU then asserts SMIACT# and begins saving the SMM header information to the SMM address space. SMIACT# remains asserted until after execution of a RSM instruction. Figure 3-14 illustrates the functional timing of the SMIACT# signal.

To facilitate using SMI# to power manage I/O peripherals, the 6x86MX CPU implements a feature called I/O trapping. If the current bus cycle is an I/O cycle and SMI# is asserted a minimum of three clocks prior to BRDY#, the CPU immediately begins execution of the SMI service routine following completion of the I/O instruction. No additional instructions are executed prior to entering the SMI service routine. I/O trap timing requirements are shown in Figure 3-15 (Page 3-41).

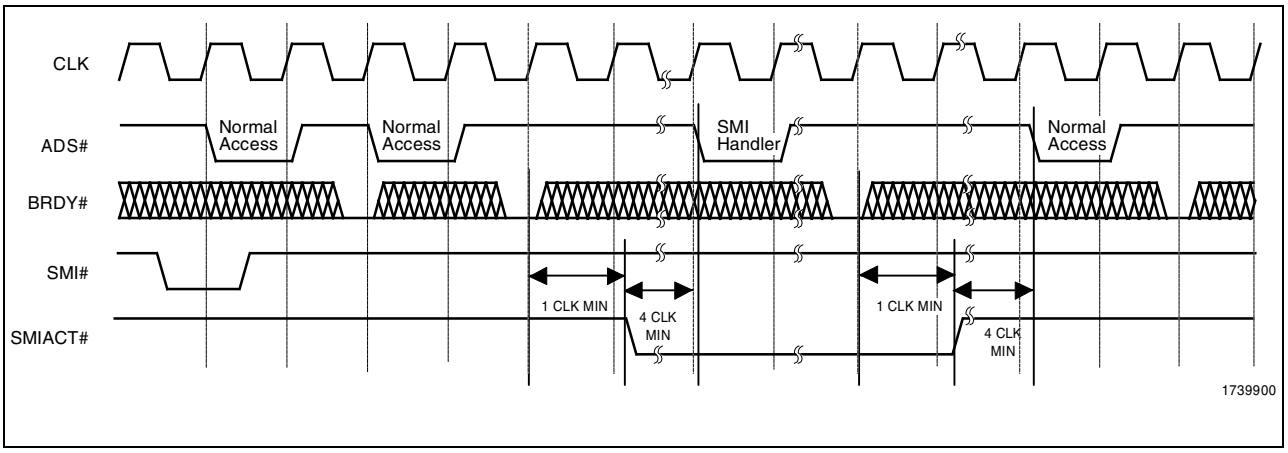


Figure 3-14. SMIACT# Timing in SL Compatible Mode

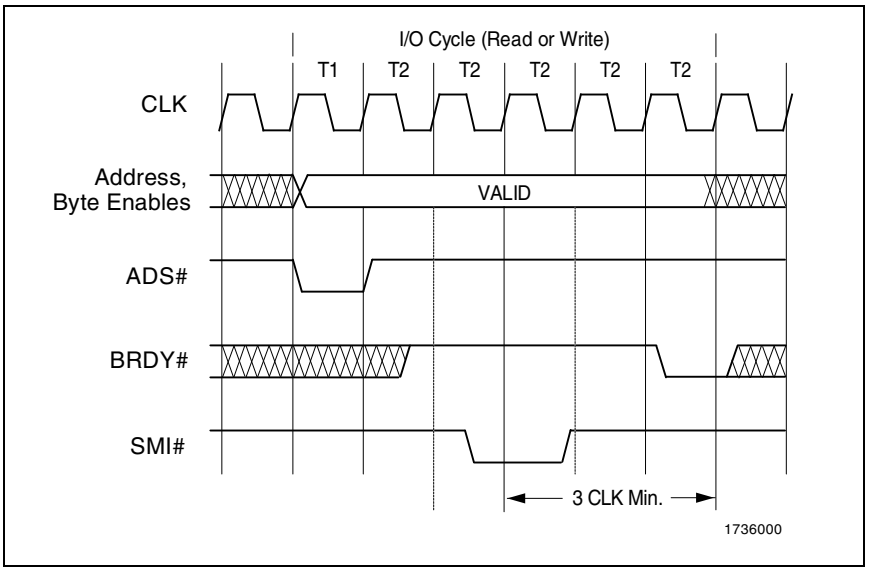


Figure 3-15. SMM I/O Trap Timing

3.3.7 Cache Control Timing

3.3.7.1 Invalidating the Cache Using FLUSH#

The FLUSH# input forces the CPU to write-back and invalidate the entire contents of the on-chip cache. FLUSH# is sampled at each clock edge, latched internally and then recognized internally at the next instruction boundary. Once FLUSH# is recognized, the CPU issues a series of burst write cycles to write-back any "modified" cache lines. The cache lines are invalidated as they are written back. Following completion of the write-back cycles, the CPU issues a flush acknowledge special bus cycle.

The latency between when FLUSH# occurs and when the cache invalidation actually completes varies depending on:

(1) the state of the processor when FLUSH# is asserted,
(2) the number of modified cache lines,
(3) the number of wait states inserted during the write-back cycles.

Figure 3-16 (Page 3-42) illustrates the sequence of events that occur on the bus in response to a FLUSH# request.

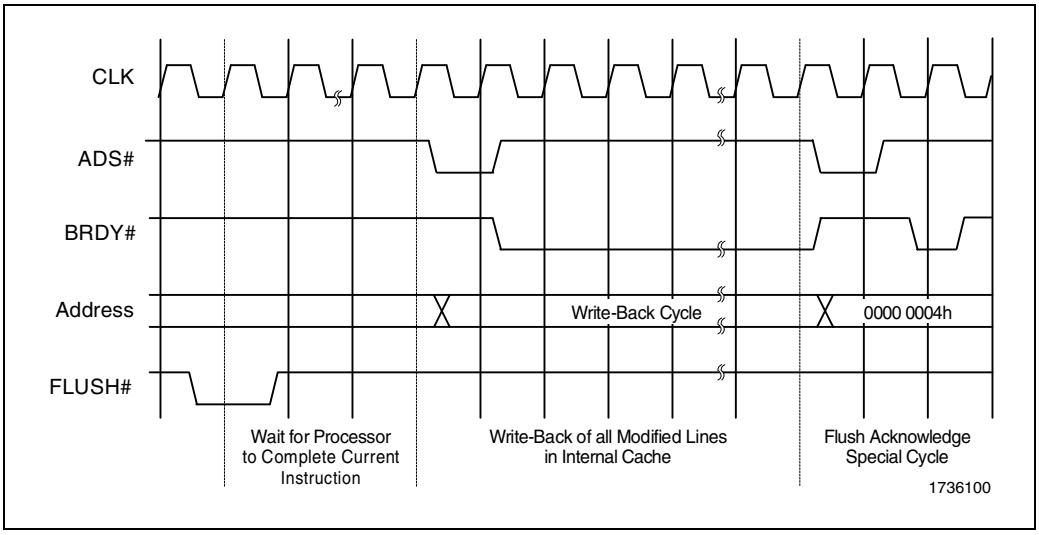


Figure 3-16. Cache Invalidation Using FLUSH#

3.3.7.2 EWBE# Timing

During memory and I/O write cycles, the 6x86MX CPU samples the external write buffer empty (EWBE#) input. If EWBE# is negated, the CPU does not write any data to "exclusive" or "modified" internal cache lines. After sampling EWBE# negated, the CPU continues to sample EWBE# at each clock edge until it asserts. Once EWBE# is asserted, all internal cache writes are allowed. Through use of this signal, the external system may enforce strong write ordering when external write buffers are used. EWBE# functional timing is shown in Figure 3-17.

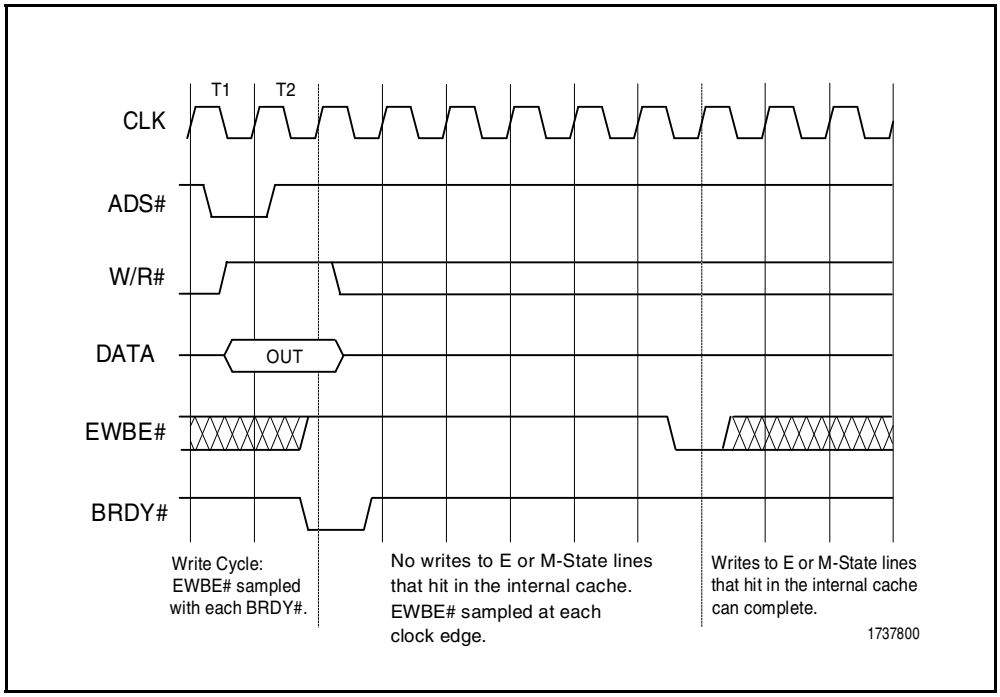


Figure 3-17. External Write Buffer Empty (EWBE#) Timing

3.3.8 Bus Arbitration

An external bus master can take control of the CPU's bus using either the HOLD/HLDA handshake signals or the back-off (BOFF#) input. Both mechanisms force the 6x86MX CPU to enter the bus hold state.

3.3.8.1 HOLD and HLDA

Using the HOLD/HLDA handshake, an external bus master requests control of the CPU's bus by asserting the HOLD signal. In response to an active HOLD signal, the CPU completes all outstanding bus cycles, enters the bus hold state by floating the bus, and asserts the HLDA output. The CPU remains in the bus hold state until HOLD is negated. Figures 3-18 (this page), Figure 3-19 (Page 3-45) and Figure 3-20 (Page 3-46) illustrate the timing associated with requesting HOLD during an idle bus, during a non-pipelined bus cycle and during a pipelined bus cycle, respectively.

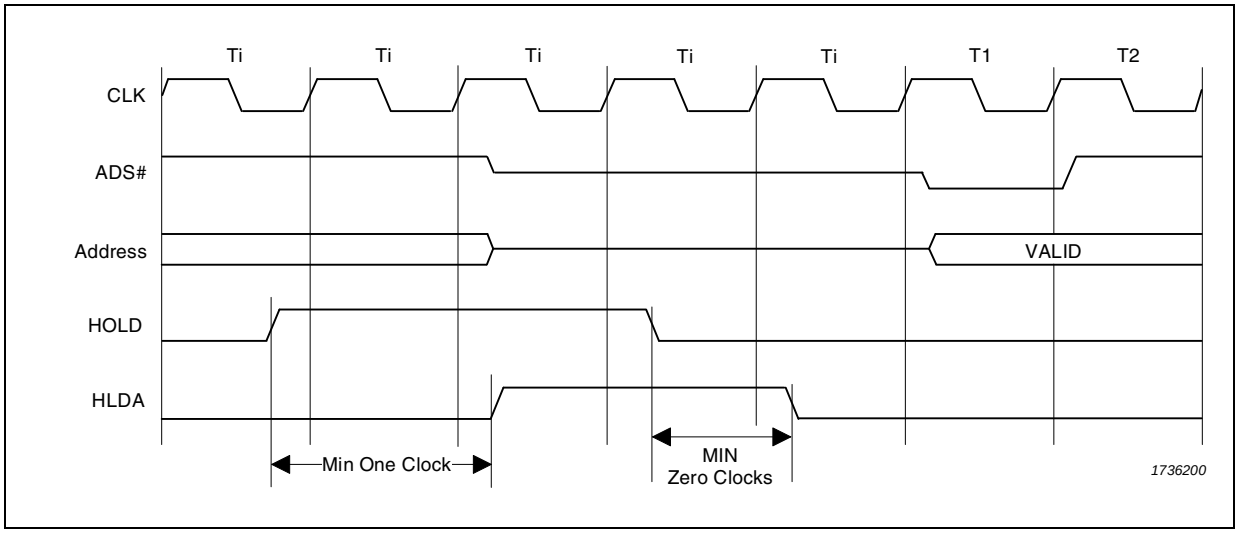


Figure 3-18. Requesting Hold from an Idle Bus

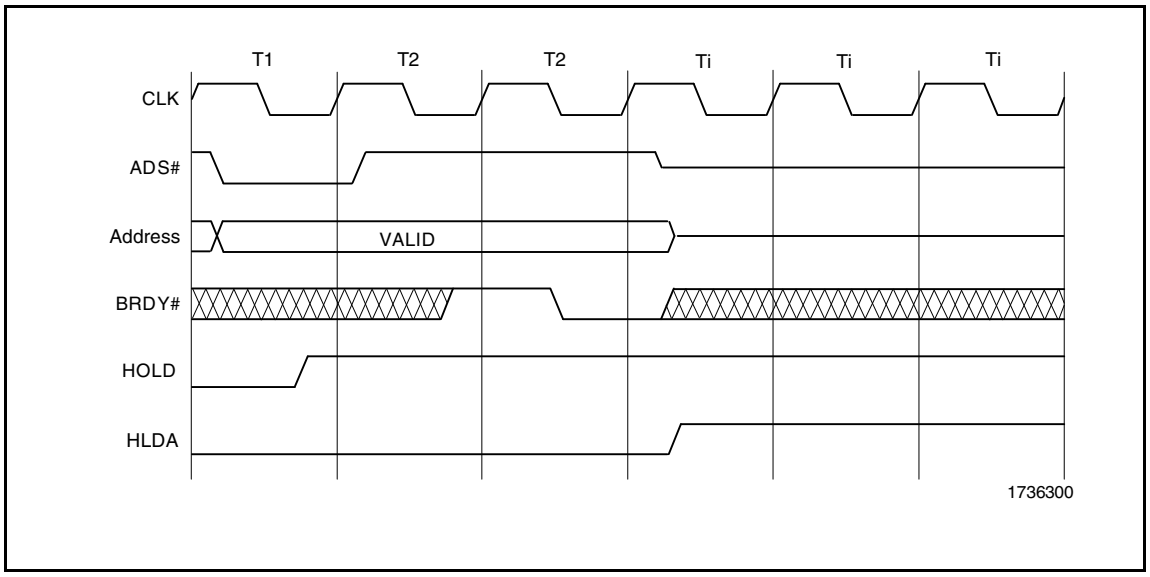


Figure 3-19. Requesting Hold During a Non-Pipelined Bus Cycle

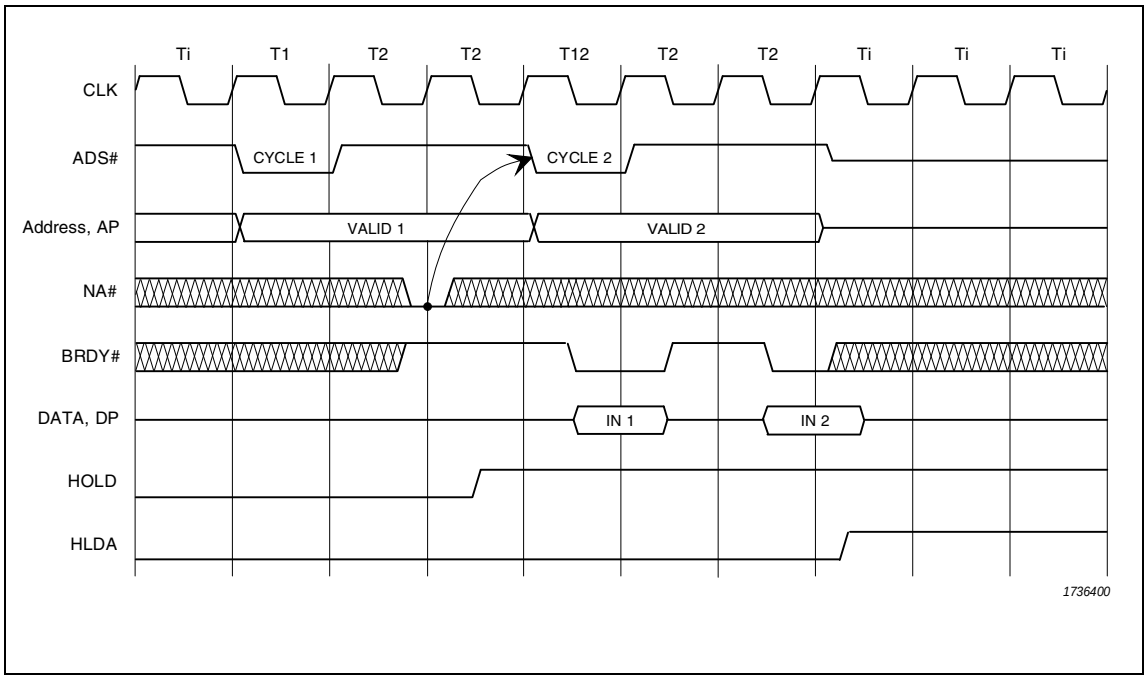


Figure 3-20. Requesting Hold During a Pipelined Bus Cycle

3.3.8.2 Back-Off Timing

An external bus master requests immediate control of the CPU's bus by asserting the back-off (BOFF#) input. The CPU samples BOFF# at each clock edge and responds by floating the bus in the next clock cycle as shown in Figure 3-21. The CPU remains in the bus hold state until BOFF# is negated.

If the assertion of BOFF# interrupts a bus cycle, the bus cycle is restarted in its entirety following the negation of BOFF#. If KEN# was sampled by the processor before the cycle was aborted, it must be returned with the same value during the restarted cycle. The state of WB/WT# may be changed during the restarted cycle.

If BOFF# and BRDY# are active at the same clock edge, the CPU ignores BRDY#. Any data returned to the CPU with the BRDY# is also ignored. If BOFF# interrupts a burst read cycle, the CPU does not cache any data returned prior to BOFF#. However, this data may be used for internal CPU execution.

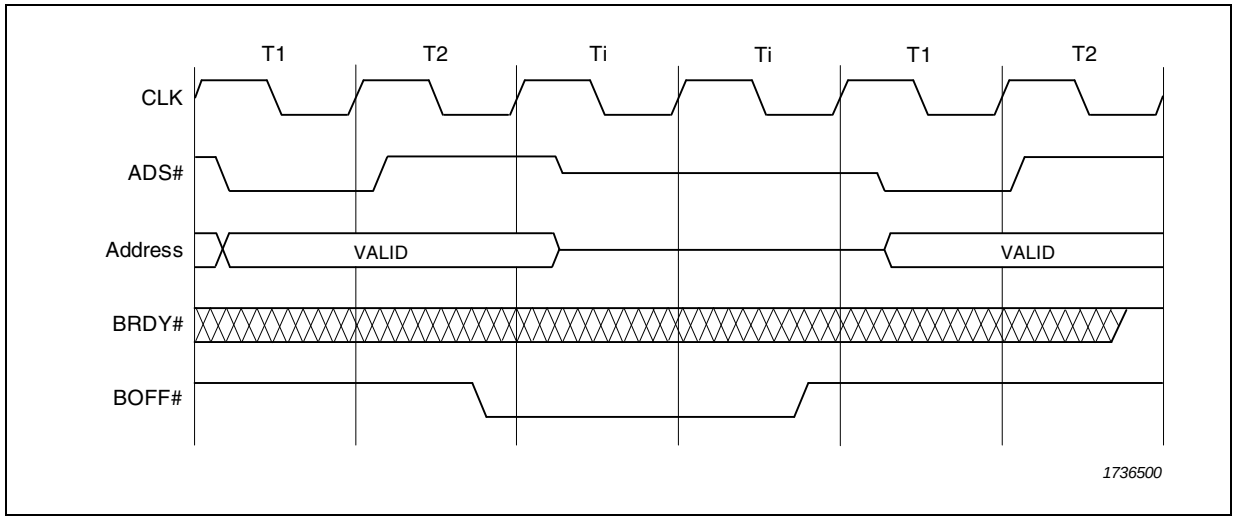


Figure 3-21. Back-Off Timing

3.3.9 Cache Inquiry Cycles

Cache inquiry cycles are issued by the system with the CPU in either a bus hold or address hold state. Bus hold is requested by asserting either HOLD or BOFF#, and address hold is requested by asserting AHOLD. The system initiates the cache inquiry cycle by asserting the EADS# input. The system must also drive the desired inquiry address on the address lines, and a valid state on the INV input.

In response to the cache inquiry cycle, the CPU checks to see if the specified address is present in the internal cache. If the address is present in the cache, the CPU checks the MESI state of the cache line. If the line is in the "exclusive" or "shared" state, the CPU asserts the HIT# output and changes the cache line state to "invalid" if the INV input was sampled logic high with EADS#.

If the line is in the "modified" state, the CPU asserts both HIT# and HITM#. The CPU then issues a bus cycle request to write the modified cache line to external memory. HITM# remains asserted until the write-back bus cycle completes. No additional cache inquiry cycles are accepted while HITM# is asserted. Write-back cycles always start at burst address 0. Once the write-back cycle has completed, the CPU changes the cache line state to "invalid" if the INV input was sampled logic high, or "shared" if the INV input was sampled low.

In addition to checking the cache, the CPU also snoops the internal line fill and cache write-back buffers in response to a cache inquiry cycle. The following sections describe the functional timing for cache inquiry cycles and the corresponding write-back cycles for the various types of inquiry cycles.

3.3.9.1 Inquiry Cycles Using HOLD/HLDA

Figure 3-22 illustrates an inquiry cycle where HOLD is used to force the CPU into a bus hold state. In this case, the system asserts HOLD and must wait for the CPU to respond with HLDA before issuing the cache inquiry cycle. To avoid address bus contention, EADS# should not be asserted until the second clock after HLDA as shown in the diagram. If the inquiry address hits on a modified cache line, HIT# and HITM# are asserted during the second clock following EADS#. Once HITM# asserts, the system must negate HOLD to allow the CPU to run the corresponding write-back cycle. The first cycle issued following negation of HLDA is the write-back bus cycle.

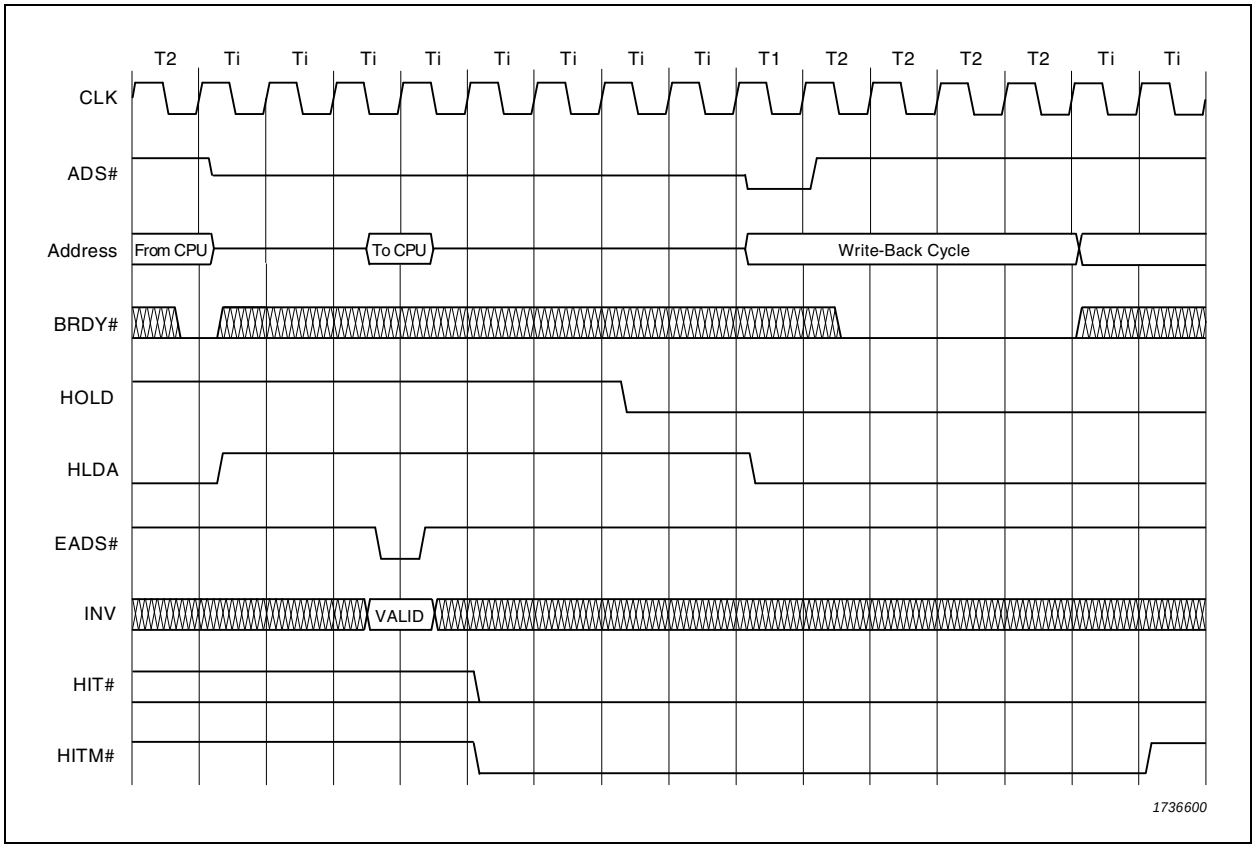


Figure 3-22. HOLD Inquiry Cycle that Hits on a Modified Line

3.3.9.2 Inquiry Cycles Using BOFF#

Figure 3-23 illustrates an inquiry cycle where BOFF# is used to force the CPU into a bus hold state. In this case, the system asserts BOFF# and the CPU immediately relinquishes control of the bus in the next clock. To avoid address bus contention, EADS# should not be asserted until the second clock edge after BOFF# as shown in the diagram. If the inquiry address hits on a modified cache line, HIT# and HITM# are asserted during the second clock following EADS#. Once HITM# asserts, the system must negate BOFF# to allow the CPU to run the corresponding write-back cycle. The first cycle issued following negation of BOFF# is the write-back bus cycle.

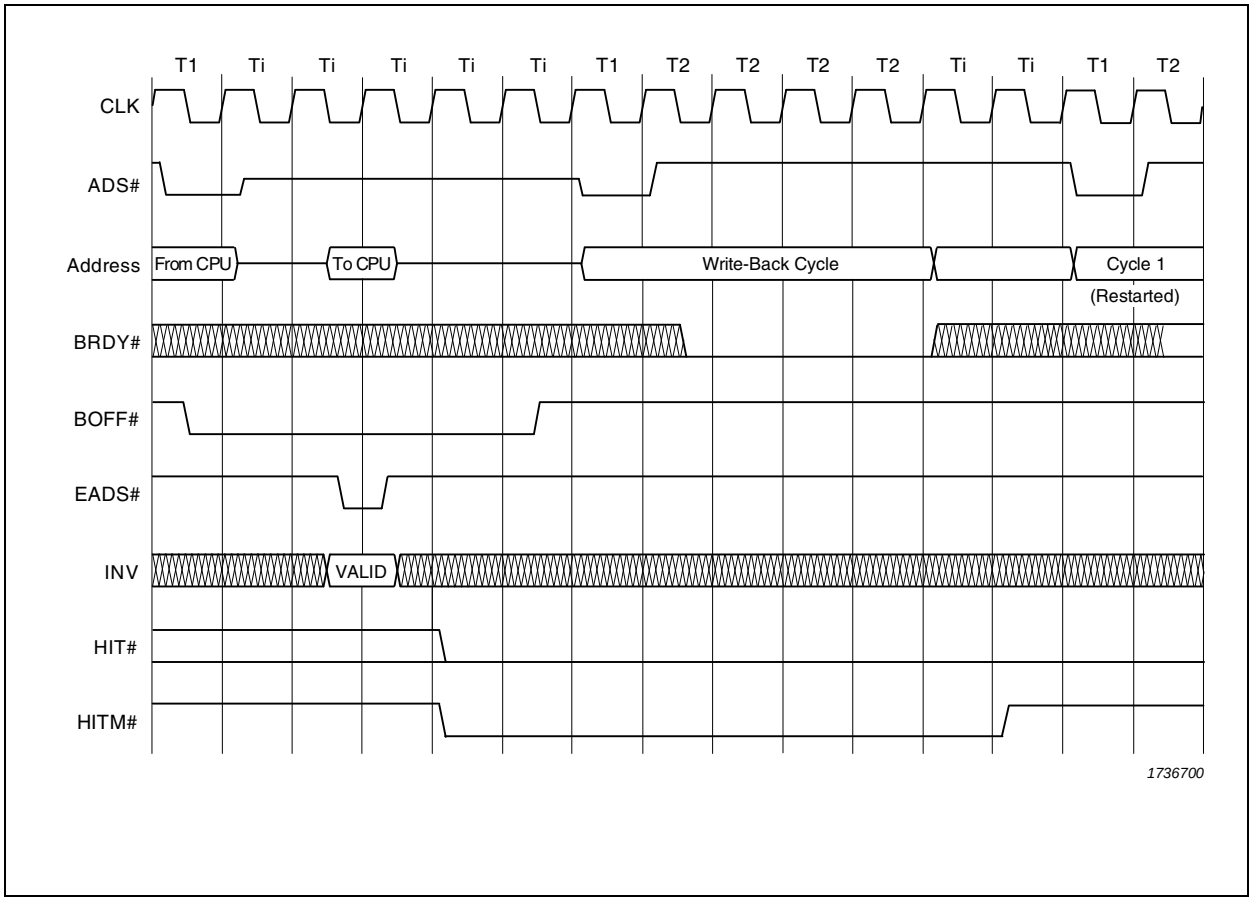


Figure 3-23. BOFF# Inquiry Cycle that Hits on a Modified Line

3.3.9.3 Inquiry Cycles Using AHOLD

Figure 3-24 illustrates an inquiry cycle where AHOLD is used to force the CPU into an address hold state. In this case, the system asserts AHOLD and the CPU immediately floats the address bus in the next clock. To avoid address bus contention, EADS# should not be asserted until the second clock edge after AHOLD as shown in the diagram. If the inquiry address hits on a modified cache line, the CPU asserts HIT# and HITM# during the second clock following EADS#. The CPU then issues the write-back cycle even if AHOLD remains asserted. ADS# for the write-back cycle asserts two clocks after HITM# is asserted. To prevent the address bus and data bus from switching simultaneously, the system must adhere to the restrictions on negation of AHOLD as shown in Figure 3-24.

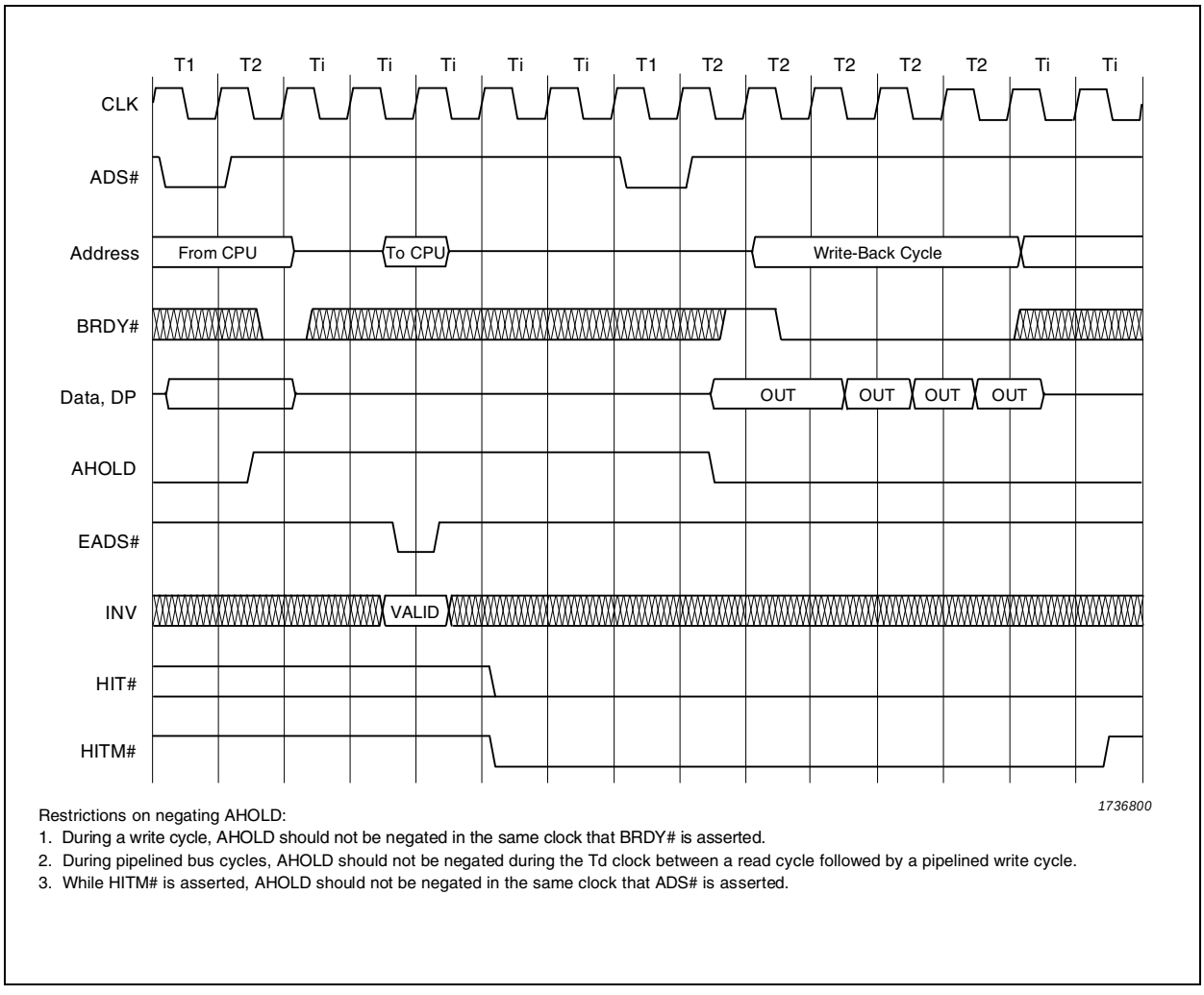


Figure 3-24. AHOLD Inquiry Cycle that Hits on a Modified Line

Figure 3-25 depicts an AHOLD inquiry cycle during a line fill. In this case, the write-back cycle occurs after the line fill is completed. At least one idle clock exists between the final BRDY# of the line fill and the ADS# for the write-back cycle. If the inquiry cycle hits on the address of the line fill that is in progress, the data from the line fill cycle is always used to complete the pending internal operation. However, the data is not placed in the cache if INV is sampled asserted with EADS#. The data is placed in the cache in a "shared" state if INV is sampled negated.

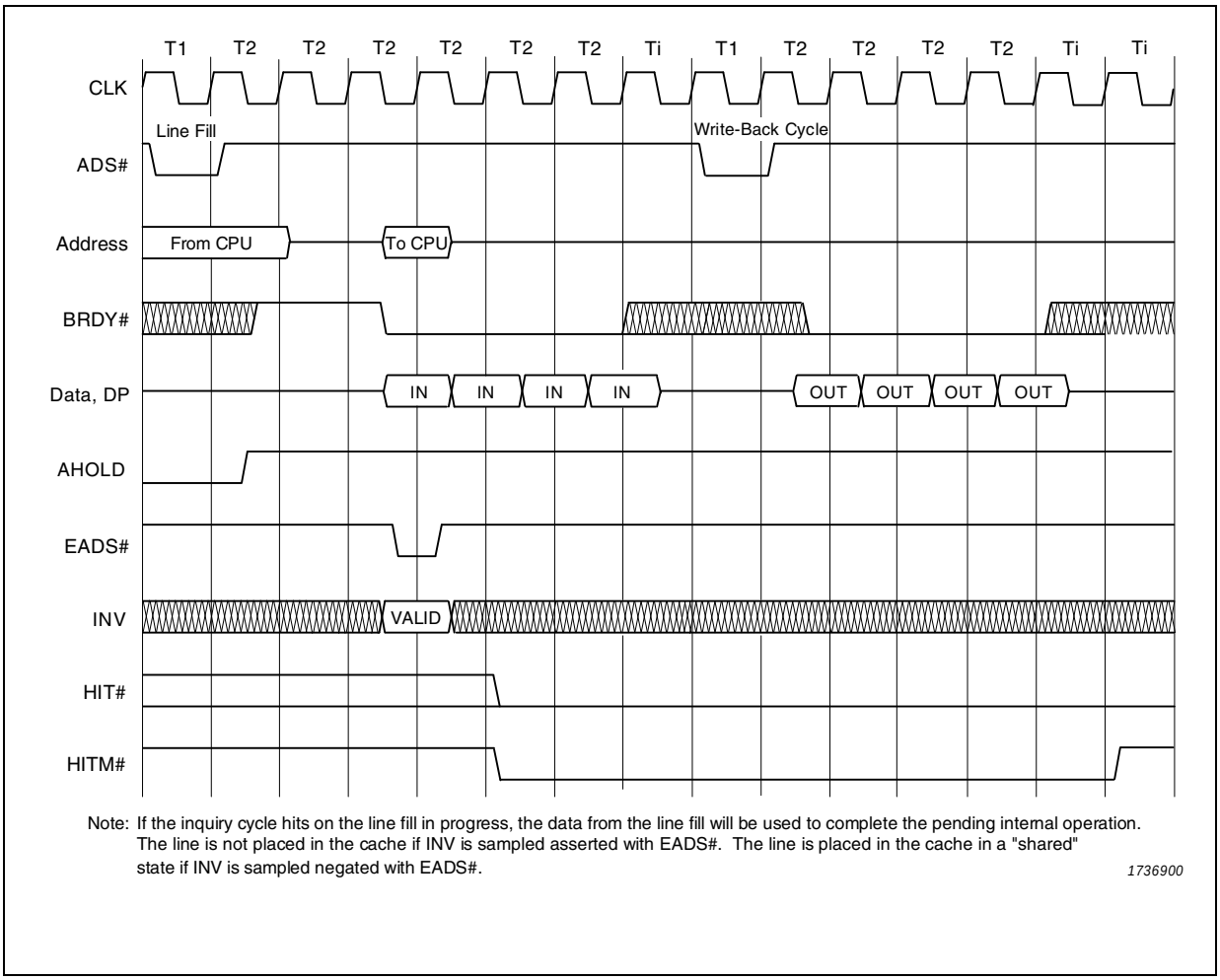


Figure 3-25. AHOLD Inquiry Cycle During a Line Fill

During cache inquiry cycles, the CPU performs address parity checking using A31-A5 and the AP signal. The CPU checks for even parity and asserts the APCHK# output if a parity error is detected. Figure 3-26 illustrates the functional timing of the APCHK# output.

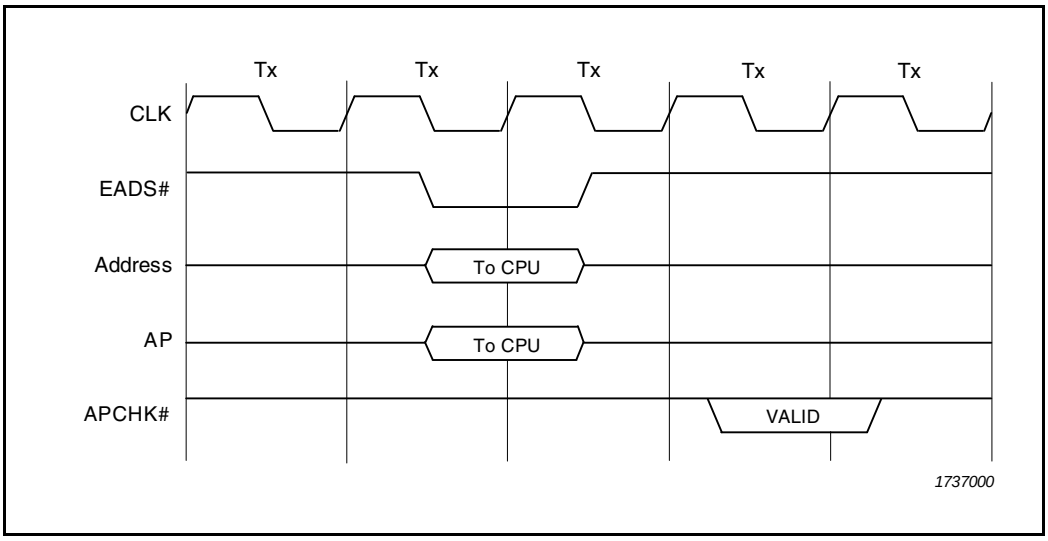


Figure 3-26. APCHK# Timing

3.3.10 Cache Inquiry Cycles During SMM Mode

It is assumed that while operating in SL-compatible mode SMM code and data are non-cacheable thereby precluding any inquiry cycles from hitting on cache lines containing modified SMM data. Therefore this section is only relevant while operating in Cyrix enhanced SMM mode.

Cache inquiry cycles are issued by the system with the CPU in either a bus hold or address hold state. The SMIACT# pin is floated along with the other buses, and bus control signals as defined by the bus hold state. The SMIACT# pin follows the timing protocol shown in Figure 3-27 in

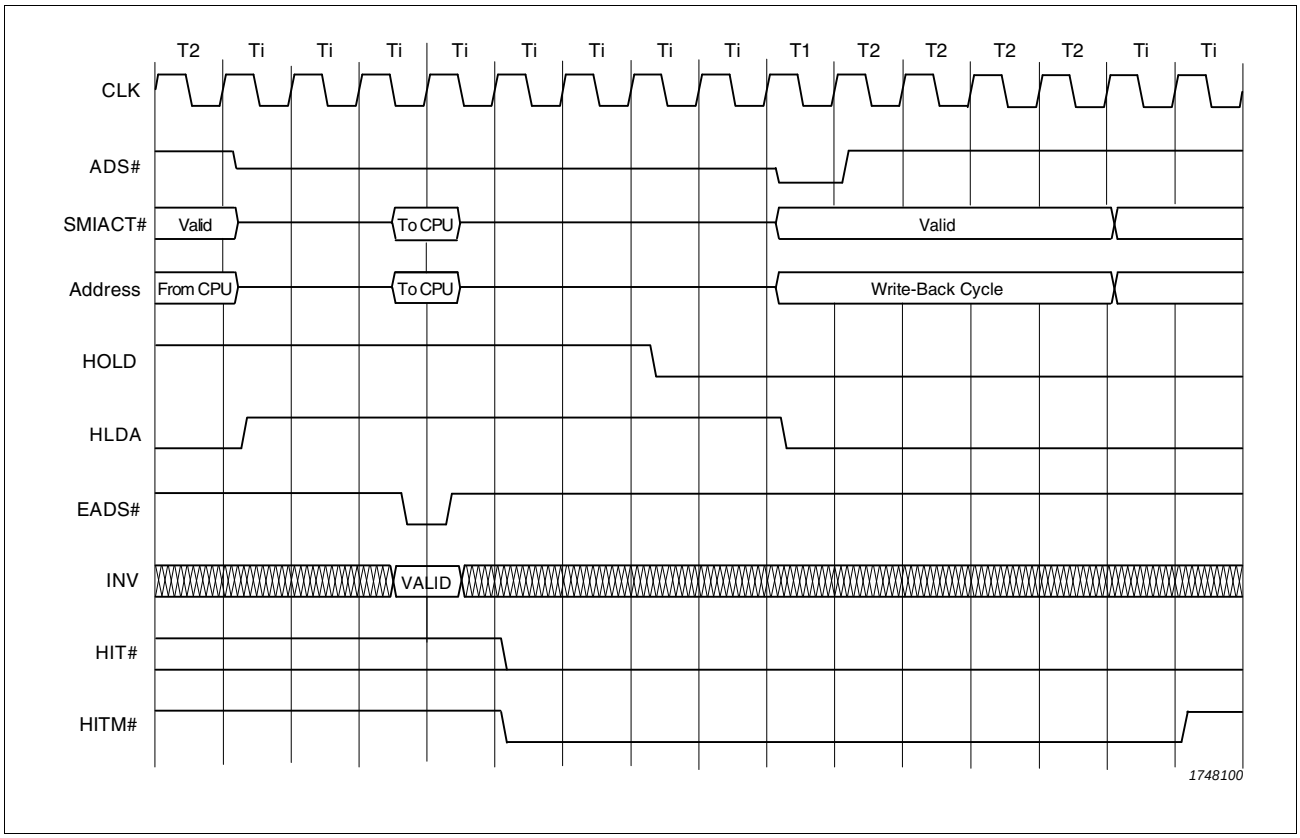


Figure 3-27. Hold Inquiry that Hits on a Modified Data Line

regards to an inquiry during an address hold request. Bus hold is requested by asserting either HOLD or BOFF#, and address hold is requested by asserting AHOLD. The system initiates the cache inquiry cycle by asserting the EADS# input. The system must also drive the desired inquiry address on the address lines, and a valid state on the INV input.

In response to the cache inquiry cycle the CPU checks to see if the specified address is present in the internal cache. If the address is present in the cache, the CPU checks the MESI state of the cache line. If the line is in the "exclusive" or "shared" state, the CPU asserts the HIT# output and changes the cache line state to "invalid" if the INV input was sampled logic high with EADS#. If the line is in the "modified" state, the CPU asserts both HIT# and HITM#. The CPU then issues a bus cycle request to write the modified cache line to external memory. If the data to be written back is SMM data, the CPU asserts SMIACT# 1 cycle before asserting the ADS of the write back cycle. HITM# remains asserted until the write-back bus cycle completes. No additional cache inquiry cycles are accepted while HITM# is asserted. Write-back cycles always start at burst address 0. Once the write-back cycle has completed, the CPU changes the cache line state to "invalid" if the INV input was sampled logic high, or "shared" if the INV input was sampled low.

3.3.10.1 Inquiry Cycles Using BOFF, HOLD/HLDA

The system asserts HOLD or BOFF# to force the CPU into a bus hold state. The system must wait for the CPU to respond with HLDA before issuing the cache inquiry cycle, or in the case of BOFF# the CPU immediately relinquishes control to the bus in the next cycle. To avoid address bus contention, EADS# should not be asserted until the second clock edge after HLDA/BOFF#. If the inquiry address hits on a modified cache line, HIT# and HITM# are asserted during the second clock following EADS#. Once HITM# asserts, the system must negate HOLD/BOFF# to allow the CPU to run the corresponding write-back cycle. The first cycle issued following negation of HLDA/BOFF# is the write-back bus cycle. If this cycle is to SMM memory then SMIACT# is asserted, otherwise this cycle is run with SMIACT# high.

If SMIACT# was low prior to HLDA/BOFF# assertion and write-back cycle is intended for main memory then SMIACT# must be pulled high at least one clock prior to assertion of ADS# for the write-back cycle. See Figure 3-28 and Figure 3-29 (Page 3-57). If there is no write-back bus cycle to run and the next cycle to be run is to SMM memory then SMIACT# must be asserted at least 1 clock prior to assertion of ADS# as defined in Figure 3-30 (Page 3-58).

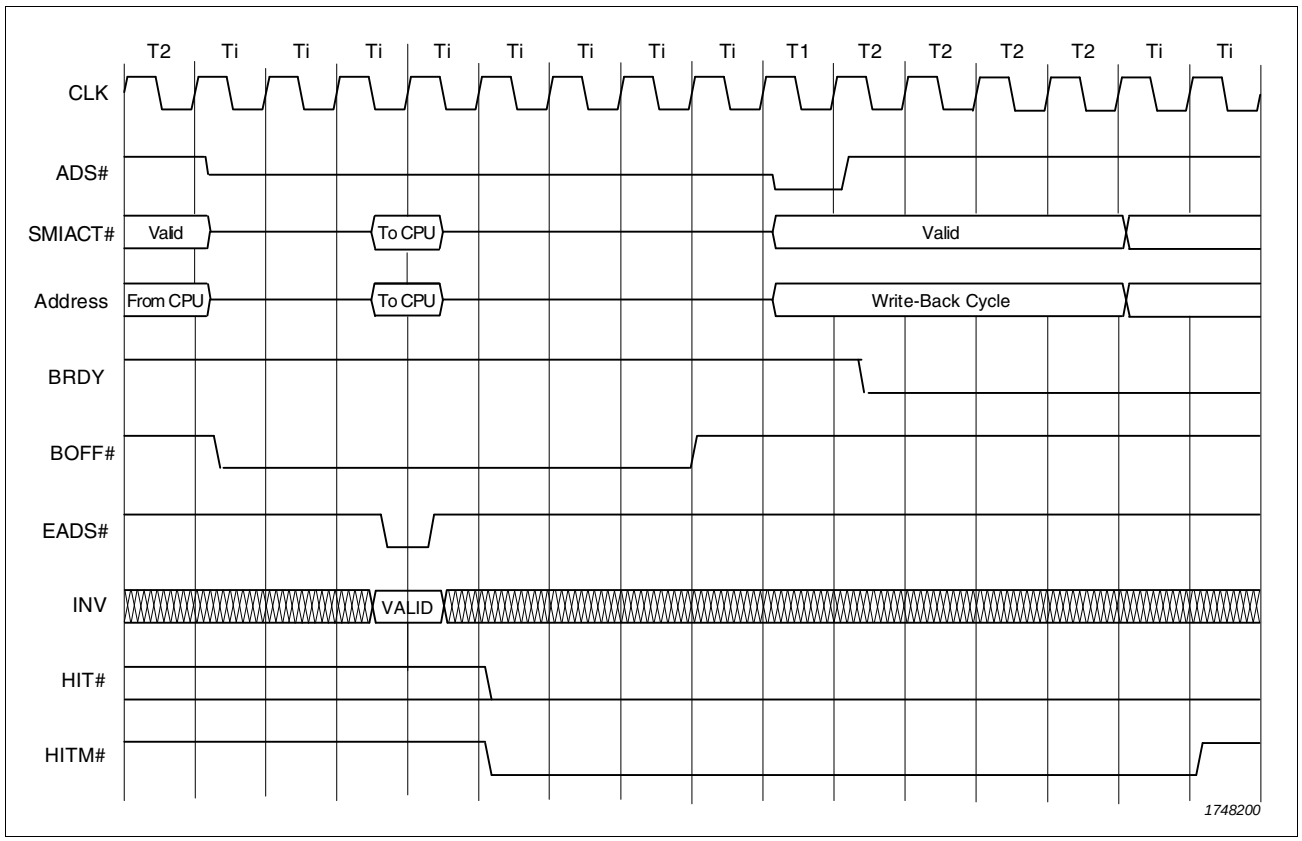


Figure 3-28. BOFF# Inquiry Cycle that Hits on a Modified Data Line

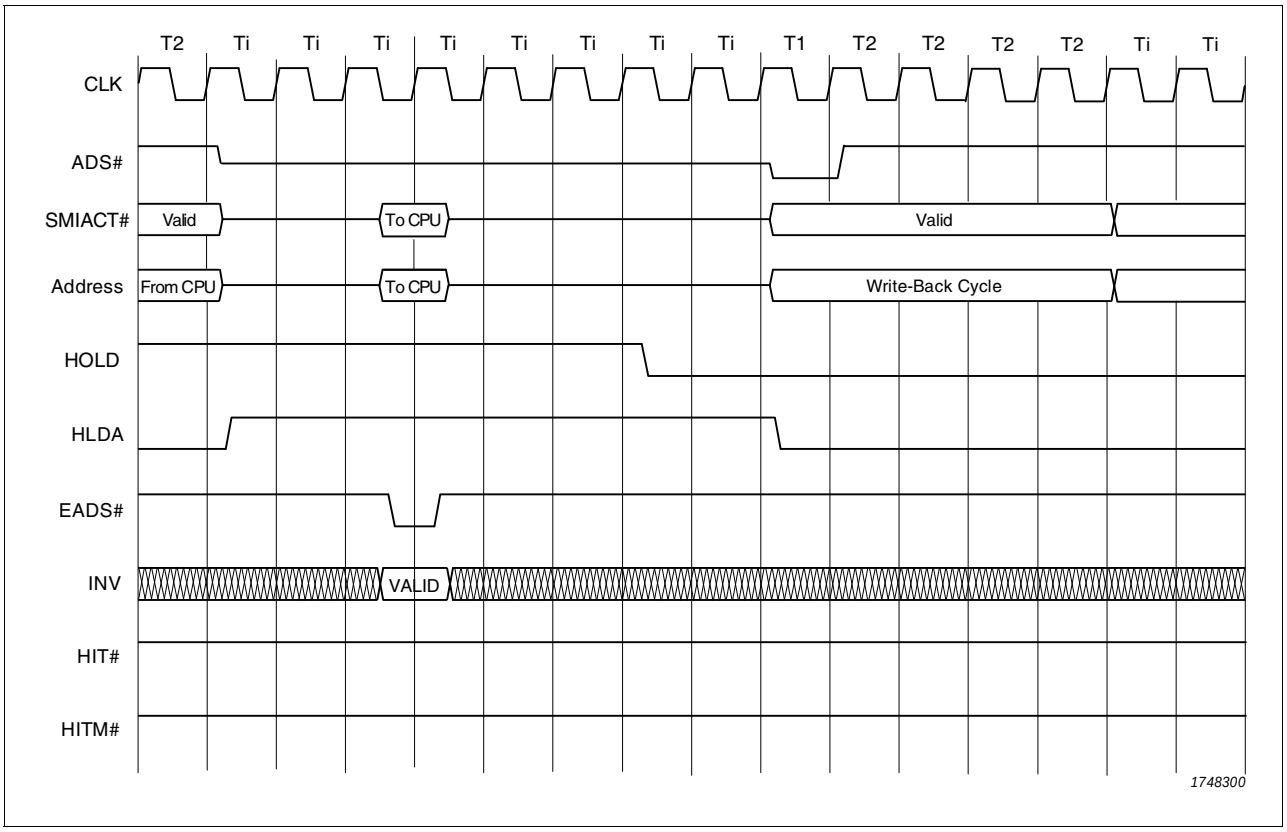


Figure 3-29. HOLD Inquiry Cycle that Misses the Cache While in SMM Mode

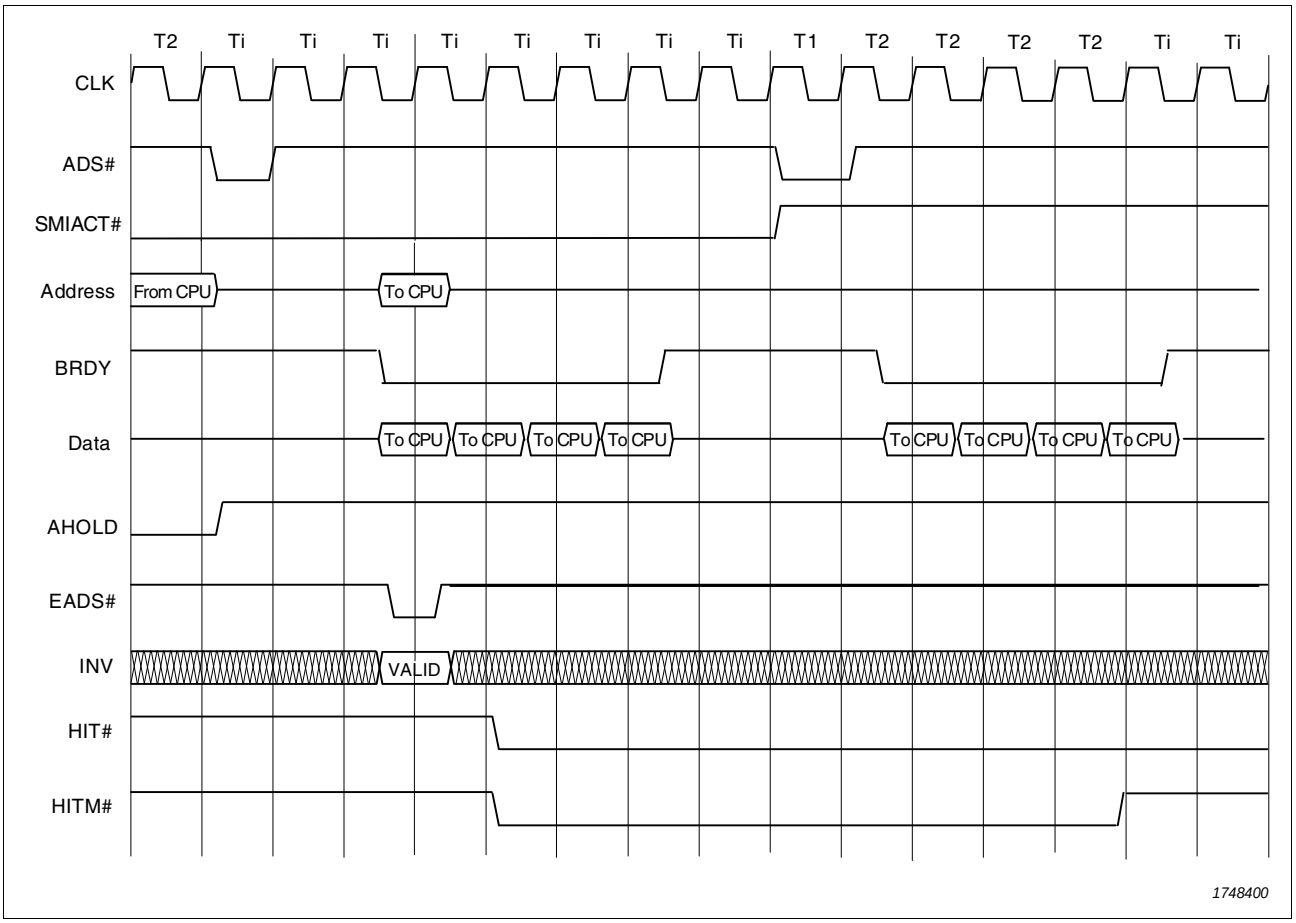


Figure 3-30. AHOLD Inquiry Cycle During a Line Fill from SMM Memory

3.3.10.2 Inquiry Cycles Using AHOLD

In this case, the system asserts AHOLD the CPU immediately floats the address bus in the next clock. To avoid address bus contention, EADS# should not be asserted until the second clock edge after AHOLD. If the inquiry address hits on a modified cache line the CPU asserts HIT# and HITM# during the second clock following EADS#. The CPU then issues the write-back cycle even if AHOLD remains asserted. If this cycle is to SMM memory then SMIACT# is asserted, otherwise this cycle is run with SMIACT# high. If SMIACT# was low prior to AHOLD assertion and write back cycle is intended for main memory then SMIACT# must be pulled high at least one clock prior to assertion of ADS# for the write-back cycle.

Likewise, if SMIACT# was high prior to AHOLD assertion and the write-back cycle is intended for SMM memory then SMIACT# must be pulled low at least one clock prior to assertion of ADS#. If there is no write-back bus cycle to run and the next cycle to be run is to SMM memory then SMIACT# must be asserted at least one clock prior to assertion of ADS#.

The following timing diagram depicts an AHOLD inquiry cycle during a line fill from SMM memory. In this case, the write-back cycle occurs after the line fill is completed, and one clock after SMIACT# is set to a logic high provided the write-back cycle is to main memory. For this case, if the write-back cycle is to SMM memory then the one clock setup time criterion for SMIACT# to ADS# is met and the write-back cycle can start immediately.

3.3.11 Power Management Interface

SUSP# Initiated Suspend Mode

The 6x86MX CPU enters suspend mode when the SUSP# input is asserted and execution of the current instruction, any pending decoded instructions and associated bus cycles are completed. A stop grant bus cycle is then issued and the SUSPA# output is asserted. The CPU responds to SUSP# and asserts SUSPA# only if the SUSP bit is set in the CCR2 configuration register.

SUSP# is sampled (Figure 3-31) on the rising edge of CLK. SUSP# must meet specified setup and hold times to be recognized at a particular CLK edge. The time from assertion of SUSP# to activation of SUSPA# varies depending on which instructions were decoded prior to assertion of SUSP#. The minimum time from SUSP# sampled active to SUSPA# asserted is eight CLKs. As a maximum, the CPU may execute up to two instructions and associated bus cycles prior to asserting SUSPA#. The time required for the CPU to deactivate SUSPA# once SUSP# has been sampled inactive is five CLKs.

If the CPU is in a hold acknowledge state and SUSP# is asserted, the CPU may or may not enter suspend mode depending on the state of the CPU internal execution pipeline. If the CPU is in a SUSP# initiated suspend mode, one occurrence of NMI, INTR and SMI# is stored for execution once suspend mode is exited. The 6x86MX CPU also recognizes and acknowledges the HOLD, AHOLD, BOFF# and FLUSH# signals while in suspend mode.

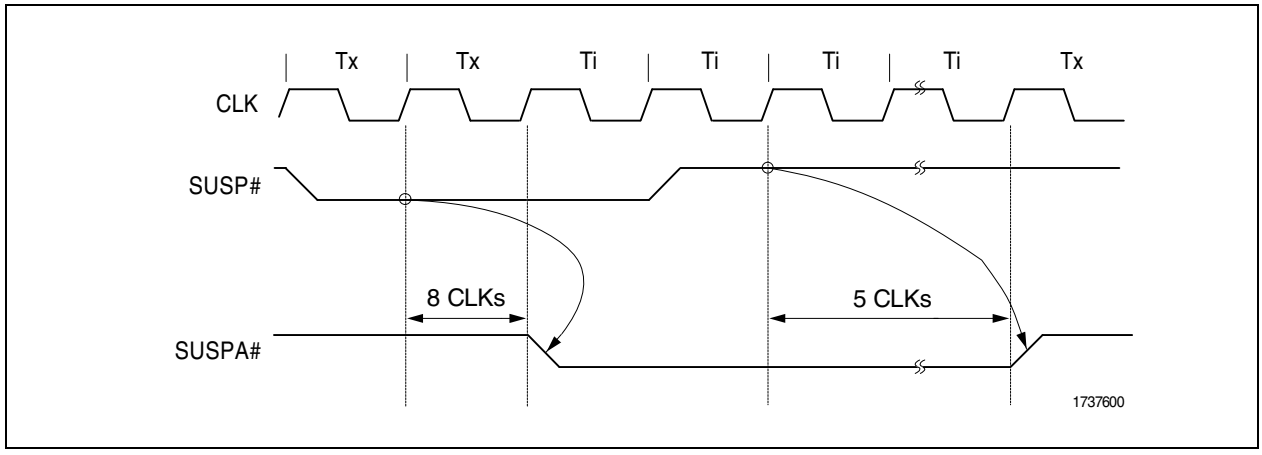


Figure 3-31. SUSP# Initiated Suspend Mode

HALT Initiated Suspend Mode

The CPU also enters suspend mode as a result of executing a HALT instruction if the HALT bit in CCR2 is set. The SUSPA# output is asserted no later than 40 CLKs following BRDY# sampled active for the HALT bus cycle as shown in Figure 3-32. Suspend mode is then exited upon recognition of an NMI, an unmasked INTR or an SMI#. SUSPA# is deactivated 10 CLKs after sampling of an active interrupt.

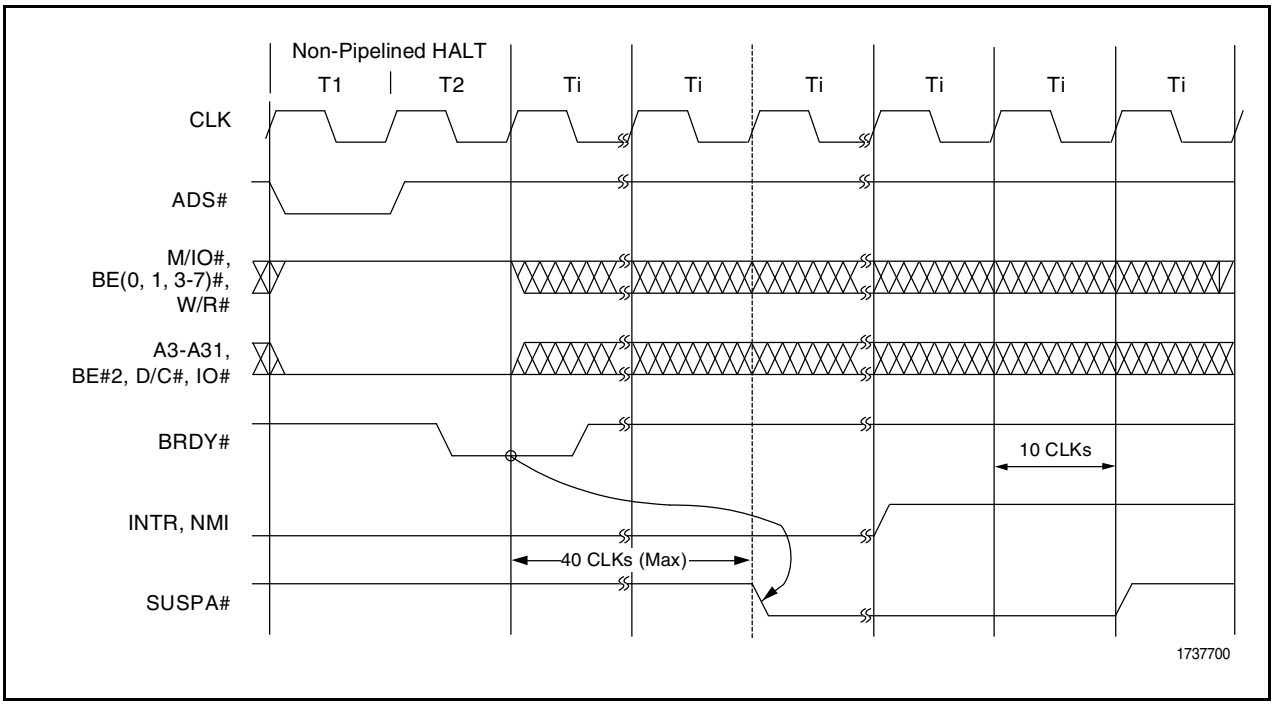


Figure 3-32. HALT Initiated Suspend Mode

Stopping the Input Clock

Once the CPU has entered suspend mode, the input clock (CLK) can be stopped and restarted without loss of any internal CPU data. The CLK input can be stopped at either a logic high or logic low state.

The CPU remains suspended until CLK is restarted and suspend mode is exited as described earlier. While the CLK is stopped, the CPU can no longer sample and respond to any input stimulus.

Figure 3-33 illustrates the recommended sequence for stopping the CLK using SUSP# to initiate suspend mode. CLK may be started prior to or following negation of the SUSP# input. The system must allow sufficient time for the CPU's internal PLL to lock to the desired frequency before exiting suspend mode.

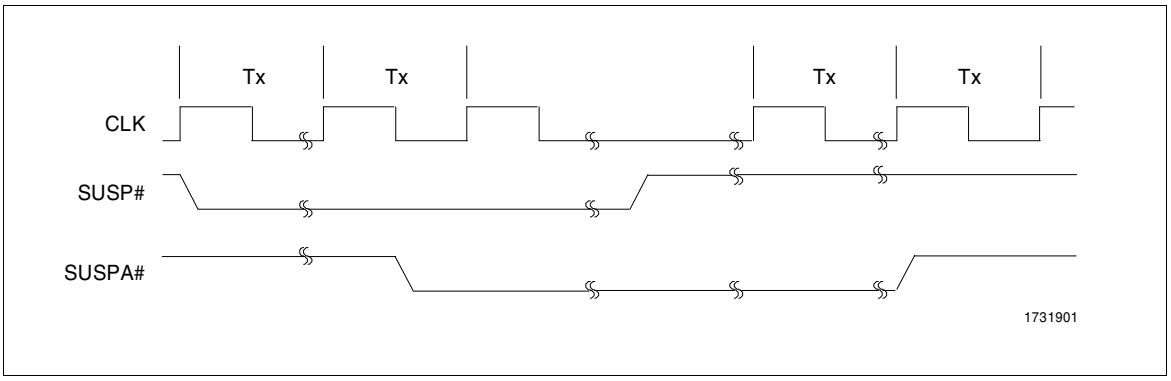


Figure 3-33. Stopping CLK During Suspend Mode

4.0 ELECTRICAL SPECIFICATIONS

4.1 Electrical Connections

This section provides information on electrical connections, absolute maximum ratings, recommended operating conditions, DC characteristics, and AC characteristics. All voltage values in Electrical Specifications are measured with respect to V_{SS} unless otherwise noted.

The 6x86MX CPU operates using two power supply voltages—one for the I/O (3.3 V) and one for the core (2.9 V).

4.1.1 Power and Ground Connections and Decoupling

Testing and operating the 6x86MX CPU requires the use of standard high frequency techniques to reduce parasitic effects. The high clock frequencies used in the 6x86MX CPU and its output buffer circuits can cause transient power surges when several output buffers switch output levels simultaneously. These effects can be minimized by filtering the DC power leads with low-inductance decoupling capacitors, using low impedance wiring, and by utilizing all of the V_{CC} and GND pins. The 6x86MX CPU contains 296 pins with 25 pins connected to V_{CC2} (2.9 volts), 28 pins connected to V_{CC3} (3.3 volts), and 53 pins connected to V_{SS} (ground).

4.1.2 Pull-Up/Pull-Down Resistors

Table 4-1 lists the input pins that are internally connected to pull-up and pull-down resistors. The pull-up resistors are connected to V_{CC} and the pull-down resistors are connected to V_{SS}. When unused, these inputs do not require connection to external pull-up or pull-down resistors. The SUSP# pin is unique in that it is connected to a pull-up resistor only when SUSP# is not asserted.

Table 4-1. Pins Connected to Internal Pull-Up and Pull-Down Resistors

SIGNAL	PIN NO.	RESISTOR
BRDYC#	Y3	20-kΩ pull-up
CKMUL0	Y33	20-kΩ pull-down (see text)
CKMUL1	X34	20-kΩ pull-up (see text)
Reserved	AN35	20-kΩ pull-down
Reserved	W35	20-kΩ pull-up
SMI#	AB34	20-kΩ pull-up
SUSP#	Y34	20-kΩ pull-up (see text)
TCK	M34	20-kΩ pull-up
TDI	N35	20-kΩ pull-up
TMS	P34	20-kΩ pull-up
TRST#	Q33	20-kΩ pull-up

4.1.3 Unused Input Pins

All inputs not used by the system designer and not listed in Table 4-1 should be connected either to ground or to V_{CC}. Connect active--high inputs to ground through a 10 kΩ (± 10%) pull-down resistor and active-low inputs to V_{CC} through a 10 kΩ (± 10%) pull-up resistor to prevent possible spurious operation.

4.1.4 NC and Reserved Pins

Pins designated NC have no internal connections. Pins designated RESV or RESERVED should be left disconnected. Connecting a reserved pin to a pull-up resistor, pull-down resistor, or an active signal could cause unexpected results and possible circuit malfunctions.

4.2 Absolute Maximum Ratings

The following table lists absolute maximum ratings for the 6x86MX CPU processors. Stresses beyond those listed under Table 4-2 limits may cause permanent damage to the device. These are stress ratings only and do not imply that operation under any conditions other than those listed under "Recommended Operating Conditions" Table 4-3 (Page 4-3) is possible. Exposure to conditions beyond Table 4-2 may (1) reduce device reliability and (2) result in premature failure even when there is no immediately apparent sign of failure. Prolonged exposure to conditions at or near the absolute maximum ratings may also result in reduced useful life and reliability.

Table 4-2. Absolute Maximum Ratings

PARAMETER	MIN	MAX	UNITS	NOTES
Operating Case Temperature	-65	110	°C	Power Applied
Storage Temperature	-65	150	°C	
Supply Voltage, V_{CC3}	-0.5	4.0	V	
Supply Voltage, V_{CC2}	-0.5	3.3	V	
Voltage On Any Pin	-0.5	V_{CC3} + 0.5	V	Not to exceed Vcc3 max
Input Clamp Current, I_{IK}		10	mA	Power Applied
Output Clamp Current, I_{OK}		25	mA	Power Applied

4.3 Recommended Operating Conditions

Table 4-3 presents the recommended operating conditions for the 6x86MX CPU device.

Table 4-3. Recommended Operating Conditions

PARAMETER	MIN	MAX	UNITS	NOTES
T_C Operating Case Temperature	0	70	°C	Power Applied
V_{CC3} Supply Voltage (3.3 V)	3.135	3.465	V	
V_{CC2} Supply Voltage (2.9 V)	2.8	3.0	V	
V_{IH} High-Level Input Voltage (except CLK)	2.00	3.55	V	
V_{IH} CLK High-Level Input Voltage	2.0	5.5	V	
V_{IL} Low-Level Input Voltage	-0.3	0.8	V	
I_{OH} High-Level Output Current		-1.0	mA	$V_O=V_{OH(MIN)}$
I_{OL} Low-Level Output Current		5.0	mA	$V_O=V_{OL(MAX)}$

4.4 DC Characteristics

Table 4-4. DC Characteristics (at Recommended Operating Conditions) 1 of 2

PARAMETER	MIN	TYP	MAX	UNITS	NOTES
V_{OL} Low-Level Output Voltage			0.4	V	I_{OL} = 5 mA
V_{OH} High-Level Output Voltage	2.4			V	I_{OH} = -1 mA
I_I Input Leakage Current For all pins (except those listed in Table 4-1).			±15	µA	$0 < V_{IN} < V_{CC3}$ Note 1
I_{IH} Input Leakage Current For all pins with internal pull-downs.			200	µA	V_{IH} = 2.4 V Note 1
I_{IL} Input Leakage Current For all pins with internal pull-ups.			-400	µA	V_{IL} = 0.45 V Note 1
C_{IN} Input Capacitance			15	pF	f = 1 MHz*
C_{OUT} Output Capacitance			20	pF	f = 1 MHz*
C_{IO} I/O Capacitance			25	pF	f = 1 MHz*
C_{CLK} CLK Capacitance			15	pF	f = 1 MHz*

*Note: Not 100% tested.

Table 4-5. DC Characteristics (at Recommended Operating Conditions) 2 of 2

PARAMETER	ICC2 MAX	ICC3 MAX	UNITS	NOTES
I_{CC} Active I_{CC}				Notes 1, 2
133 MHz (PR166)	5770	100	mA	
150 MHz (PR166)	6100	100		
166 MHz (PR200)	6600	100		
188 MHz (PR233)	7480	100		
208 MHz (PR266)	7920	100		
I_{CCSM} Active I_{CC}				Notes 1, 2, 3
133 MHz (PR166)	37	100	mA	
150 MHz (PR166)	39	100		
166 MHz (PR200)	42	100		
188 MHz (PR233)	44	100		
208 MHz (PR266)	49	100		
I_{CCSS} Standby I_{CC}				Notes 1, 2, 4
0 MHz (Suspended/CLK Stopped)	30	50.0	mA	

Notes:
1. These values should be used for power supply design. Maximum I_{CC} is determined using the worst-case instruction sequences and functions at maximum Vcc.
2. Frequency (MHz) ratings refer to the internal clock frequency.
3. All inputs at 0.4 or V_{CC3} - 0.4 (CMOS levels). All inputs held static except clock and all outputs unloaded (static I_{OUT} = 0 mA).
4. All inputs at 0.4 or V_{CC3} - 0.4 (CMOS levels). All inputs held static and all outputs unloaded (static I_{OUT} = 0 mA).

Table 4-6. Power Dissipation

PARAMETER	POWER		UNITS	NOTES
	TYP	MAX		
Active Power Dissipation				Note 1
133 MHz (PR166)	10.1	16.7	W	
150 MHz (PR166)	10.7	17.7		
166 MHz (PR200)	11.5	19.1		
188 MHz (PR233)	13.1	21.7		
208 MHz (PR266)	13.8	23.0		
Suspend Mode Power Dissipation				Notes 1, 2
133 MHz (PR166)		0.100	W	
150 MHz (PR166)		0.112		
166 MHz (PR200)		0.122		
188 MHz (PR233)		0.135		
208 MHz (PR266)		0.147		
Standby Mode Power Dissipation				Notes 1, 3
0 MHz (Suspended/CLK Stopped)		0.070	W	

Notes:
1. Systems must be designed to thermally dissipate the maximum active power dissipation. Maximum power is determined using the worst-case instruction sequences and functions with Vcc2 = 2.9 V and Vcc3 = 3.3 V.
2. All inputs at 0.4 or V_{CC3} - 0.4 (CMOS levels). All inputs held static except clock and all outputs unloaded (static I_{OUT} = 0 mA).
3. All inputs at 0.4 or V_{CC3} - 0.4 (CMOS levels). All inputs held static and all outputs unloaded (static I_{OUT} = 0 mA).

4.5 AC Characteristics

Tables 4-7 through 4-12 (Pages 4-8 through 4-11) list the AC characteristics including output delays, input setup requirements, input hold requirements and output float delays. These measurements are based on the measurement points identified in Figure 4-1 (Page 4-7) and Figure 4-2 (Page 4-8). The rising clock edge reference level V_{REF} and other reference levels are shown in Table 4-7. Input or output signals must cross these levels during testing.

Figure 4-1 shows output delay (A and B) and input setup and hold times (C and D). Input setup and hold times (C and D) are specified minimums, defining the smallest acceptable sampling window a synchronous input signal must be stable for correct operation.

The JTAG AC timing is shown in Table 4-13 (Page 13) supported by Figures 4-6 (Page 4-13) though 4-8 (Page 4-14).

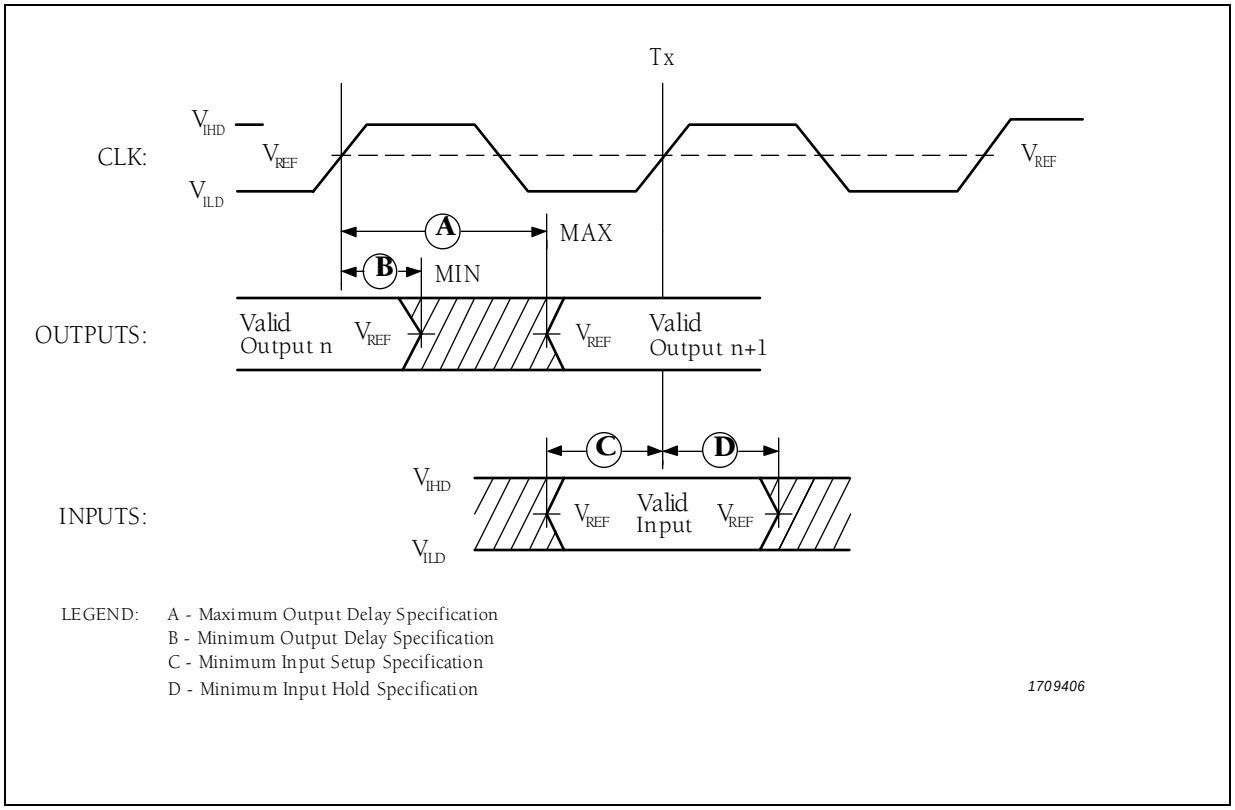


Figure 4-1. Drive Level and Measurement Points for Switching Characteristics

Table 4-7. Drive Level and Measurement Points for Switching Characteristics

SYMBOL	VOLTAGE (Volts)
V_{REF}	1.5
V_{IHD}	2.3
V_{ILD}	0

Note: Refer to Figure 4-1.

Table 4-8. Clock Specifications

T_{CASE} = 0°C to 70°C, See Figure 4-2

	PARAMETER	60-MHz BUS		66-MHz BUS		75-MHz BUS		UNITS
		MIN	MAX	MIN	MAX	MIN	MAX	
f	CLK Frequency		60		66.6		75	MHz
T1	CLK Period	16.67		15.0		13.33		ns
T2	CLK Period Stability		±250		±250		±250	ps
T3	CLK High Time	4.0		4.0		4.0		ns
T4	CLK Low Time	4.0		4.0		4.0		ns
T5	CLK Fall Time	0.15	1.5	0.15	1.5	0.15	1.5	ns
T6	CLK Rise Time	0.15	1.5	0.15	1.5	0.15	1.5	ns

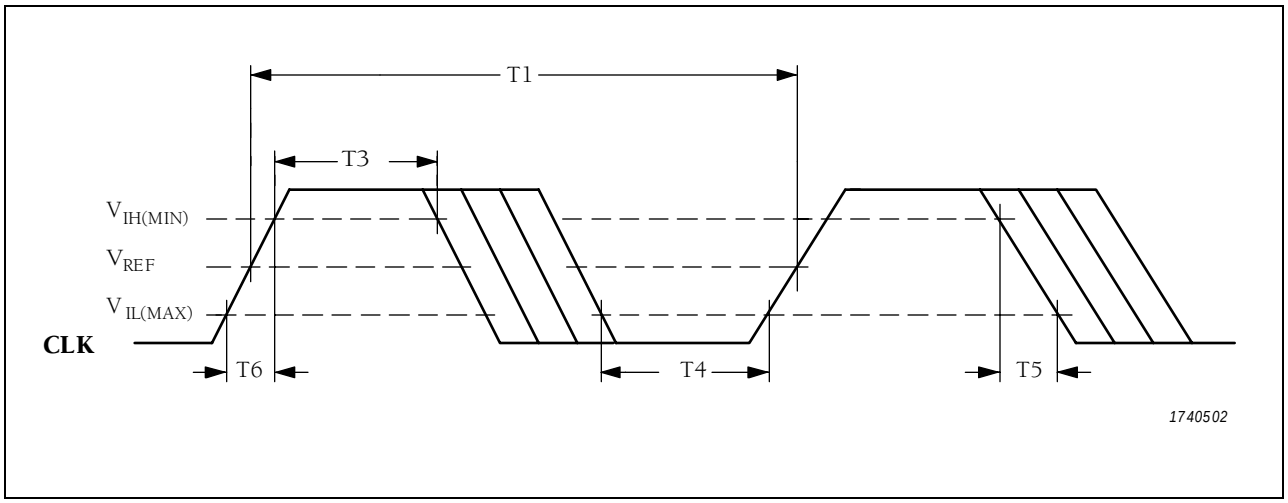


Figure 4-2. CLK Timing and Measurement Points

Table 4-9. Output Valid Delays

C_L = 50 pF, T_{case} = 0°C to 70°C, See Figure 4-3

	PARAMETER	60-MHz BUS		66-MHz BUS		75-MHz BUS		UNITS
		MIN	MAX	MIN	MAX	MIN	MAX	
T7a	A31-A3	1.0	7.0	1.0	6.3	1.0	6.3	ns
T7b	BE7#-BE0#, CACHE#, D/C#, LOCK#, PCD, PWT, SCYC, SMIACT#, W/R#	1.0	7.0	1.0	7.0	1.0	7.0	ns
T7c	ADS#	1.0	7.0	1.0	6.0	1.0	6.0	ns
T7d	M/IO#	1.0	7.0	1.0	5.9	1.0	5.9	ns
T8	ADSC#	1.0	7.0	1.0	7.0	1.0	7.0	ns
T9	AP	1.0	8.5	1.0	8.5	1.0	8.5	ns
T10	APCHK#, PCHK#, FERR#	1.0	8.3	1.0	7.0	1.0	7.0	ns
T11	D63-D0, DP7-DP0 (Write)	1.3	7.5	1.3	7.5	1.3	7.5	ns
T12a	HIT#	1.0	8.0	1.0	6.8	1.0	6.8	ns
T12b	HITM#	1.1	6.0	1.1	6.0	1.1	6.0	ns
T13a	BREQ	1.0	8.0	1.0	8.0	1.0	8.0	ns
T13b	HLDA	1.0	8.0	1.0	6.8	1.0	6.8	ns
T14	SUSPA#	1.0	8.0	1.0	8.0	1.0	8.0	ns

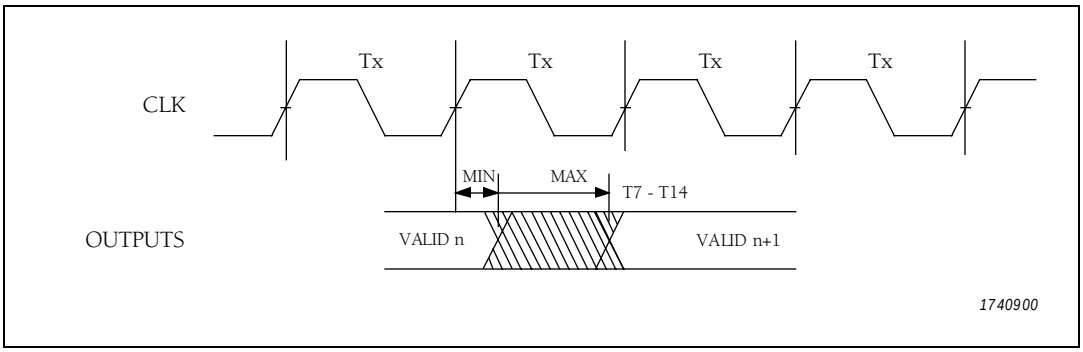


Figure 4-3. Output Valid Delay Timing

Table 4-10. Output Float Delays

C_L = 50 pF, T_{case} = 0°C to 70°C, See Figure 4-5

	PARAMETER	60-MHz BUS		66-MHz BUS		75-MHz BUS		UNITS
		MIN	MAX	MIN	MAX	MIN	MAX	
T15	A31-A3, ADS#, BE7#-BE0#, CACHE#, D/C#, LOCK#, PCD, PWT, SCYC, SMIACT#, W/R#		10.0		10.0		10.0	ns
T16	AP		10.0		10.0		10.0	ns
T17	D63-D0, DP7-DP0 (Write)		10.0		10.0		10.0	ns

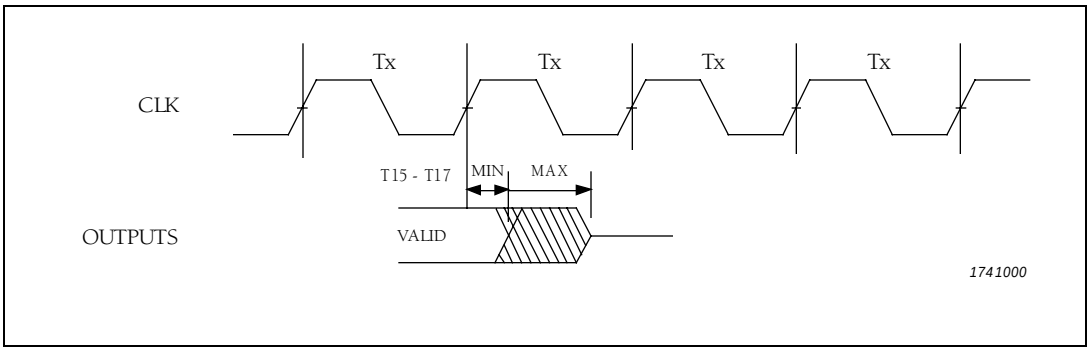


Figure 4-4. Output Float Delay Timing

Table 4-11. Input Setup Times

T_{case} = 0°C to 70°C, See Figure 4-5

	PARAMETER	60-MHz BUS	66-MHz BUS	75-MHz BUS	UNITS
		MIN	MIN	MIN	
T18	A20M#, FLUSH#, IGNNE#, SUSP#	5.0	5.0	3.3	ns
T19	AHOLD, BOFF#, HOLD	5.0	5.0	3.3	ns
T20	BRDY#	5.0	5.0	3.3	ns
T21	BRDYC#	5.0	5.0	3.3	ns
T22a	A31-A3, AP, BE7#-BE0#,	5.0	5.0	3.3	ns
T22b	AP	5.0	5.0	3.3	ns
T22c	D63-D0 (Read), DP7-DP0 (Read)	3.0	3.0	3.0	ns
T23	EADS#, INV	5.0	5.0	5.0	ns
T24	INTR, NMI, RESET, SMI#, WM_RST	5.0	5.0	5.0	ns
T25	EWBE#, KEN#, NA#, WB/WT#	4.5	4.5	3.0	ns

Table 4-12. Input Hold Times

T_{case} = 0°C to 70°C, See Figure 4-5

SYMBOL	PARAMETER	60-MHz BUS	66-MHz BUS	75-MHz BUS	UNITS
		MIN	MIN	MIN	
T27	A20M#, FLUSH#, IGNNE#, SUSP#	1.0	1.0	1.0	ns
T28	AHOLD, BOFF#, HOLD	1.0	1.0	1.0	ns
T29	BRDY#	1.0	1.0	1.0	ns
T30	BRDYC#	1.0	1.0	1.0	ns
T31a	A31-A3, AP, BE7#-BE0#,	1.0	1.0	1.0	ns
T31b	AP	1.0	1.0	1.0	ns
T31c	D63-D0 (Read), DP7-DP0 (Read)	2.0	1.5	1.5	ns
T32	EADS#, INV	1.0	1.0	1.0	ns
T33	INTR, NMI, RESET, SMI#, WM_RST	1.0	1.0	1.0	ns
T34	EWBE#, KEN#, NA#, WB/WT#	1.0	1.0	1.0	ns

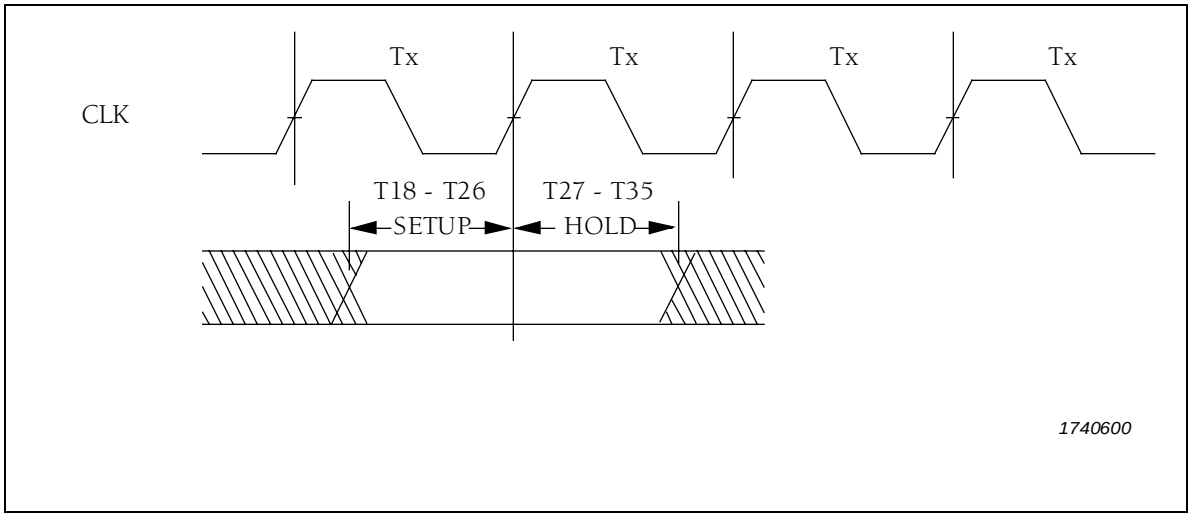


Figure 4-5. Input Setup and Hold Timing

Table 4-13. JTAG AC Specifications

SYMBOL	PARAMETER	ALL BUS FREQUENCIES		UNITS	FIGURE
		MIN	MAX		
	TCK Frequency (MHz)		20	ns	
T36	TCK Period	50		ns	4-6
T37	TCK High Time	25		ns	4-6
T38	TCK Low Time	25		ns	4-6
T39	TCK Rise Time		5	ns	4-6
T40	TCK Fall Time		5	ns	4-6
T41	TDO Valid Delay	3	20	ns	4-7
T42	Non-test Outputs Valid Delay	3	20	ns	4-7
T43	TDO Float Delay		25	ns	4-7
T44	Non-test Outputs Float Delay		25	ns	4-7
T45	TRST# Pulse Width	40		ns	4-8
T46	TDI, TMS Setup Time	20		ns	4-7
T47	Non-test Inputs Setup Time	20		ns	4-7
T48	TDI, TMS Hold Time	13		ns	4-7
T49	Non-test Inputs Hold Time	13		ns	4-7

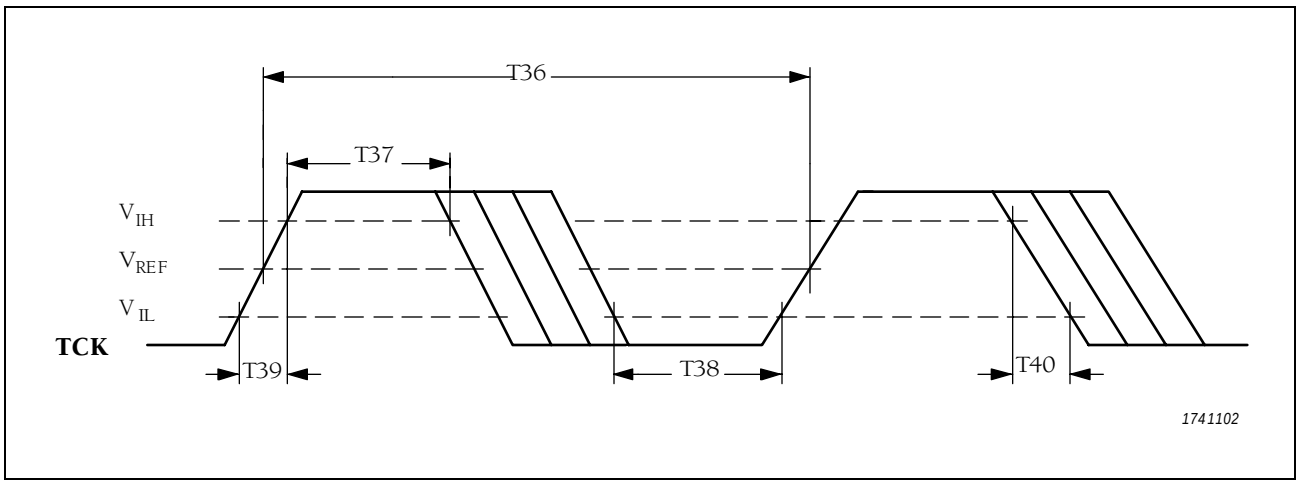


Figure 4-6. TCK Timing and Measurement Points

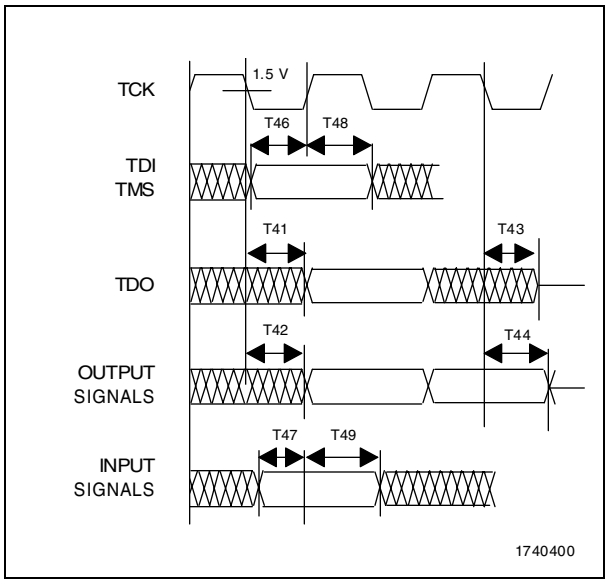


Figure 4-7. JTAG Test Timings

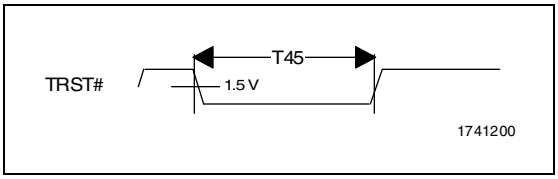


Figure 4-8. Test Reset Timing

5.0 MECHANICAL SPECIFICATIONS

5.1 296-Pin SPGA Package

The pin assignments for the 6x86MX CPU in a 296-pin SPGA package are shown in Figure 5-1. The pins are listed by signal name in Table 5-1(Page 5-3) and by pin number in Table 5-2 (Page 5-4). Dimensions are shown in Figure 5-2 (Page 5-5) and Table 5-3 (Page 5-6).

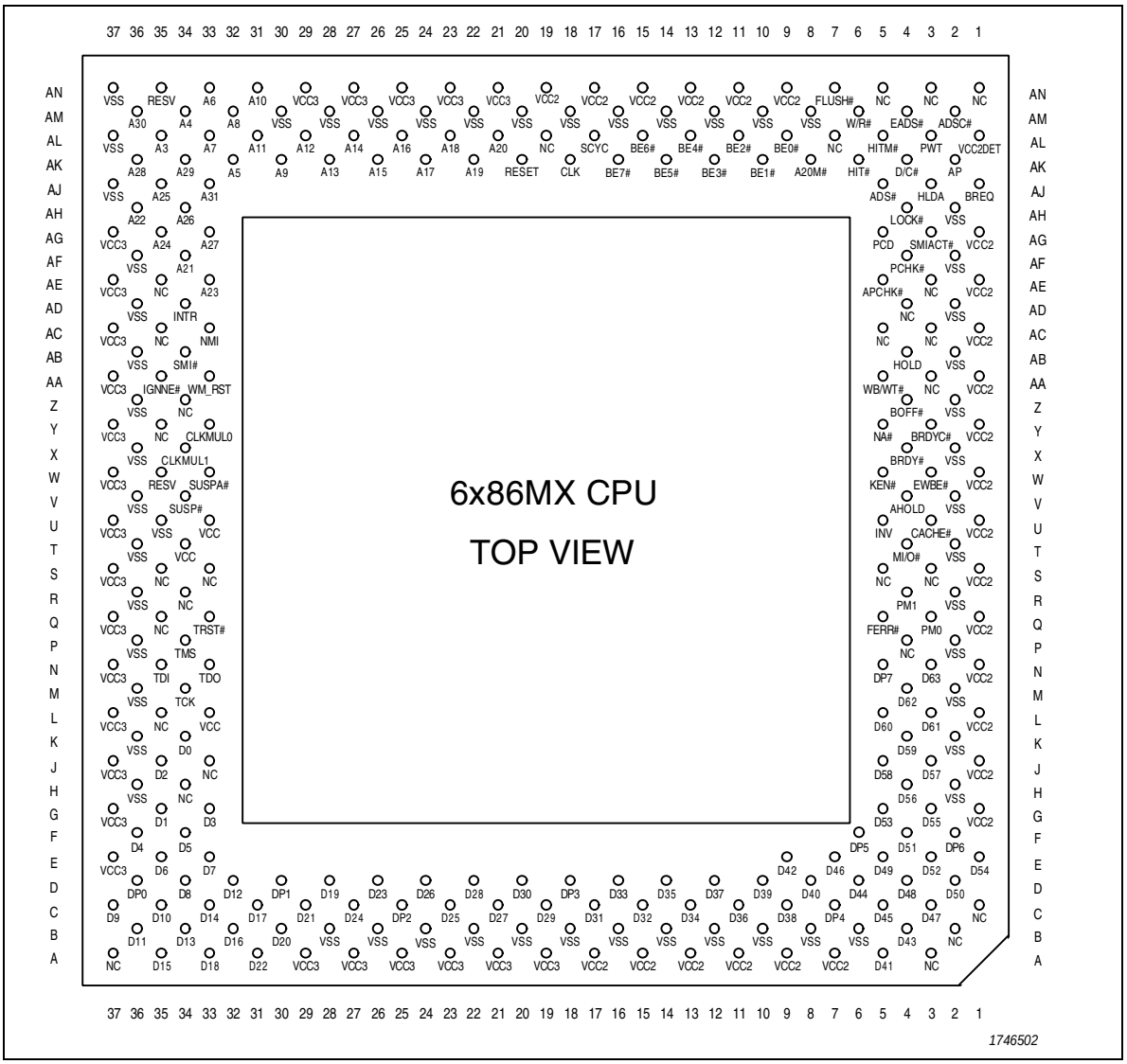


Figure 5-1. 296-Pin SPGA Package Pin Assignments

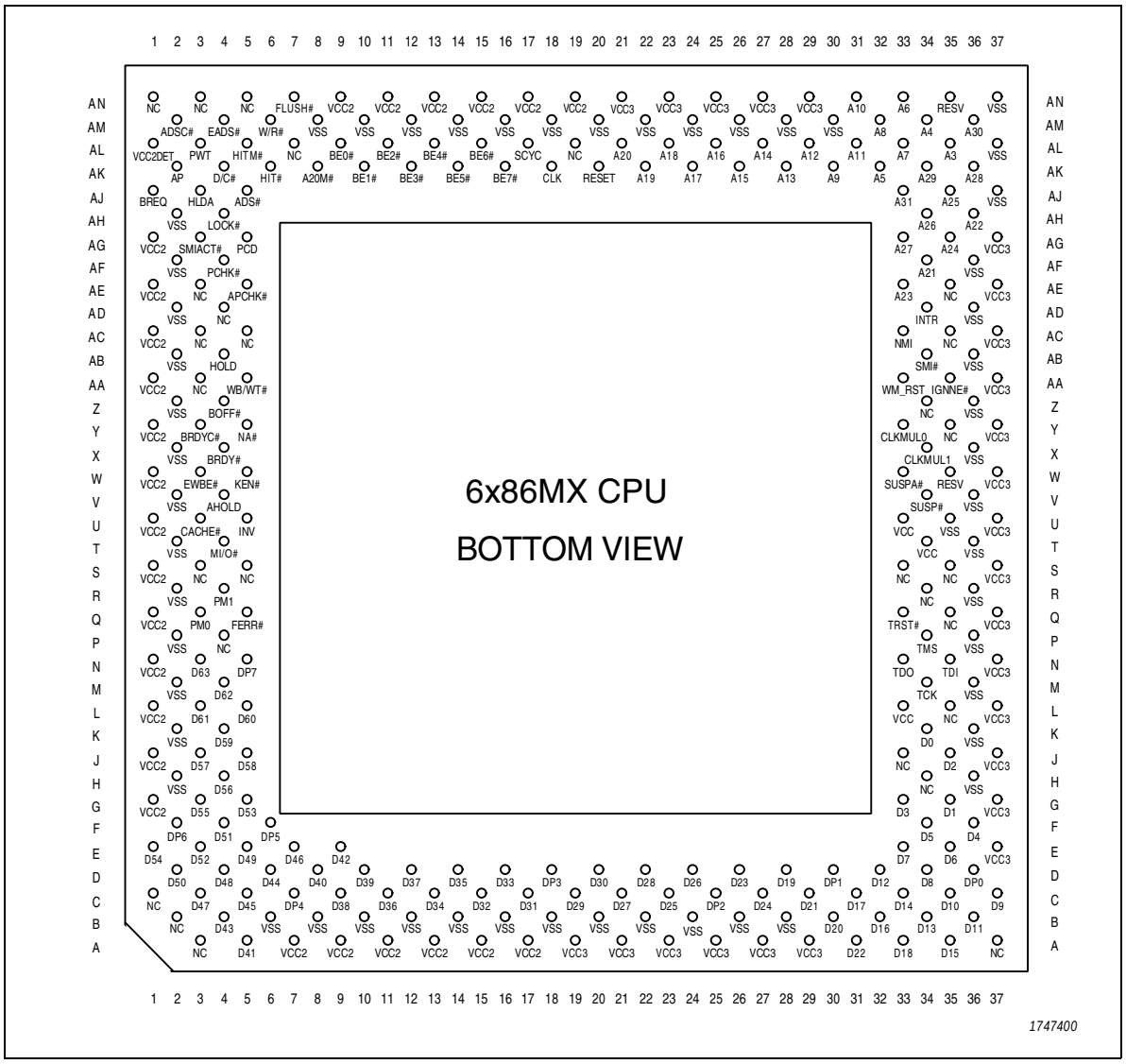


Figure 5-2. 296-Pin SPGA Package Pin Assignments (Bottom View)

Table 5-1. 296-Pin SPGA Package Signal Names Sorted by Pin Number

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
A3	NC	C29	D21	J35	D2	U35	Vss	AE35	NC	AL21	A20
A5	D41	C31	D17	J37	Vcc3	U37	Vcc3	AE37	Vcc3	AL23	A18
A7	Vcc2	C33	D14	K2	Vss	V2	Vss	AF2	Vss	AL25	A16
A9	Vcc2	C35	D10	K4	D59	V4	AHOLD	AF4	PCHK#	AL27	A14
A11	Vcc2	C37	D9	K34	D0	V34	SUSP#	AF34	A21	AL29	A12
A13	Vcc2	D2	D50	K36	Vss	V36	Vss	AF36	Vss	AL31	A11
A15	Vcc2	D4	D48	L1	Vcc2	W1	Vcc2	AG1	Vcc2	AL33	A7
A17	Vcc2	D6	D44	L3	D61	W3	EWBE#	AG3	SMIACT#	AL35	A3
A19	Vcc3	D8	D40	L5	D60	W5	KEN#	AG5	PCD	AL37	Vss
A21	Vcc3	D10	D39	L33	Vcc3	W33	SUSPA#	AG33	A27	AM2	ADSC#
A23	Vcc3	D12	D37	L35	NC	W35	Reserved	AG35	A24	AM4	EADS#
A25	Vcc3	D14	D35	L37	Vcc3	W37	Vcc3	AG37	Vcc3	AM6	W/R#
A27	Vcc3	D16	D33	M2	Vss	X2	Vss	AH2	Vss	AM8	Vss
A29	Vcc3	D18	DP3	M4	D62	X4	BRDY#	AH4	LOCK#	AM10	Vss
A31	D22	D20	D30	M34	TCK	X34	CLKMUL1	AH34	A26	AM12	Vss
A33	D18	D22	D28	M36	Vss	X36	Vss	AH36	A22	AM14	Vss
A35	D15	D24	D26	N1	Vcc2	Y1	Vcc2	AJ1	BREQ	AM16	Vss
A37	NC	D26	D23	N3	D63	Y3	BRDYC#	AJ3	HLDA	AM18	Vss
B2	NC	D28	D19	N5	DP7	Y5	NA#	AJ5	ADS#	AM20	Vss
B4	D43	D30	DP1	N33	TDO	Y33	CLKMUL0	AJ33	A31	AM22	Vss
B6	Vss	D32	D12	N35	TDI	Y35	NC	AJ35	A25	AM24	Vss
B8	Vss	D34	D8	N37	Vcc3	Y37	Vcc3	AJ37	Vss	AM26	Vss
B10	Vss	D36	DP0	P2	Vss	Z2	Vss	AK2	AP	AM28	Vss
B12	Vss	E1	D54	P4	NC	Z4	BOFF#	AK4	D/C#	AM30	Vss
B14	Vss	E3	D52	P34	TMS	Z34	NC	AK6	HIT#	AM32	A8
B16	Vss	E5	D49	P36	Vss	Z36	Vss	AK8	A20M#	AM34	A4
B18	Vss	E7	D46	Q1	Vcc2	AA1	Vcc2	AK10	BE1#	AM36	A30
B20	Vss	E9	D42	Q3	PM0	AA3	NC	AK12	BE3#	AN1	NC
B22	Vss	E33	D7	Q5	FERR#	AA5	WB/WT#	AK14	BE5#	AN3	NC
B24	Vss	E35	D6	Q33	TRST#	AA33	WM_RST	AK16	BE7#	AN5	NC
B26	Vss	E37	Vcc3	Q35	NC	AA35	IGNNE#	AK18	CLK	AN7	FLUSH#
B28	Vss	F2	DP6	Q37	Vcc3	AA37	Vcc3	AK20	RESET	AN9	Vcc2
B30	D20	F4	D51	R2	Vss	AB2	Vss	AK22	A19	AN11	Vcc2
B32	D16	F6	DP5	R4	PM1	AB4	HOLD	AK24	A17	AN13	Vcc2
B34	D13	F34	D5	R34	NC	AB34	SMI#	AK26	A15	AN15	Vcc2
B36	D11	F36	D4	R36	Vss	AB36	Vss	AK28	A13	AN17	Vcc2
C1	NC	G1	Vcc2	S1	Vcc2	AC1	Vcc2	AK30	A9	AN19	Vcc2
C3	D47	G3	D55	S3	NC	AC3	NC	AK32	A5	AN21	Vcc3
C5	D45	G5	D53	S5	NC	AC5	NC	AK34	A29	AN23	Vcc3
C7	DP4	G33	D3	S33	NC	AC33	NMI	AK36	A28	AN25	Vcc3
C9	D38	G35	D1	S35	NC	AC35	NC	AL1	Vcc2DET	AN27	Vcc3
C11	D36	G37	Vcc3	S37	Vcc3	AC37	Vcc3	AL3	PWT	AN29	Vcc3
C13	D34	H2	Vss	T2	Vss	AD2	Vss	AL5	HITM#	AN31	A10
C15	D32	H4	D56	T4	MI/O#	AD4	NC	AL7	NC	AN33	A6
C17	D31	H34	NC	T34	Vcc3	AD34	INTR	AL9	BE0#	AN35	Reserved
C19	D29	H36	Vss	T36	Vss	AD36	Vss	AL11	BE2#	AN37	Vss
C21	D27	J1	Vcc2	U1	Vcc2	AE1	Vcc2	AL13	BE4#		
C23	D25	J3	D57	U3	CACHE#	AE3	NC	AL15	BE6#		
C25	DP2	J5	D58	U5	INV	AE5	APCHK#	AL17	SCYC		
C27	D24	J33	NC	U33	Vcc3	AE33	A23	AL19	NC		

Table 5-2. 296-Pin SPGA Package Signal Names Sorted by Signal Names

Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin
A3	AL35	CLKMUL1	X34	D48	D4	NC	S5	Vcc2	Y1	Vss	B26
A4	AM34	D/C#	AK4	D49	E5	NC	S33	Vcc2	AA1	Vss	B28
A5	AK32	D0	K34	D50	D2	NC	S35	Vcc2	AC1	Vss	H2
A6	AN33	D1	G35	D51	F4	NC	Y35	Vcc2	AE1	Vss	H36
A7	AL33	D2	J35	D52	E3	NC	Z34	Vcc2	AG1	Vss	K2
A8	AM32	D3	G33	D53	G5	NC	AA3	Vcc2	AN9	Vss	K36
A9	AK30	D4	F36	D54	E1	NC	AC3	Vcc2	AN11	Vss	M2
A10	AN31	D5	F34	D55	G3	NC	AC5	Vcc2	AN13	Vss	M36
A11	AL31	D6	E35	D56	H4	NC	AC35	Vcc2	AN15	Vss	P2
A12	AL29	D7	E33	D57	J3	NC	AD4	Vcc2	AN17	Vss	P36
A13	AK28	D8	D34	D58	J5	NC	AE3	Vcc2	AN19	Vss	R2
A14	AL27	D9	C37	D59	K4	NC	AE35	Vcc3	A19	Vss	R36
A15	AK26	D10	C35	D60	L5	NC	AL7	Vcc3	A21	Vss	T2
A16	AL25	D11	B36	D61	L3	NC	AL19	Vcc3	A23	Vss	T36
A17	AK24	D12	D32	D62	M4	NC	AN1	Vcc3	A25	Vss	U35
A18	AL23	D13	B34	D63	N3	NC	AN3	Vcc3	A27	Vss	V2
A19	AK22	D14	C33	DP0	D36	NC	AN5	Vcc3	A29	Vss	V36
A20	AL21	D15	A35	DP1	D30	NMI	AC33	Vcc3	E37	Vss	X2
A20M#	AK8	D16	B32	DP2	C25	PCD	AG5	Vcc3	G37	Vss	X36
A21	AF34	D17	C31	DP3	D18	PCHK#	AF4	Vcc3	J37	Vss	Z2
A22	AH36	D18	A33	DP4	C7	PM0	Q3	Vcc3	L33	Vss	Z36
A23	AE33	D19	D28	DP5	F6	PM1	R4	Vcc3	L37	Vss	AB2
A24	AG35	D20	B30	DP6	F2	PWT	AL3	Vcc3	N37	Vss	AB36
A25	AJ35	D21	C29	DP7	N5	Reserved	W35	Vcc3	Q37	Vss	AD2
A26	AH34	D22	A31	EADS#	AM4	Reserved	AN35	Vcc3	S37	Vss	AD36
A27	AG33	D23	D26	EWBE#	W3	RESET	AK20	Vcc3	T34	Vss	AF2
A28	AK36	D24	C27	FERR#	Q5	SCYC	AL17	Vcc3	U33	Vss	AF36
A29	AK34	D25	C23	FLUSH#	AN7	SMI#	AB34	Vcc3	U37	Vss	AH2
A30	AM36	D26	D24	HIT#	AK6	SMIACT#	AG3	Vcc3	W37	Vss	AJ37
A31	AJ33	D27	C21	HITM#	AL5	SUSP#	V34	Vcc3	Y37	Vss	AL37
ADS#	AJ5	D28	D22	HLDA	AJ3	SUSPA#	W33	Vcc3	AA37	Vss	AM8
ADSC#	AM2	D29	C19	HOLD	AB4	TCK	M34	Vcc3	AC37	Vss	AM10
AHOLD	V4	D30	D20	IGNNE#	AA35	TDI	N35	Vcc3	AE37	Vss	AM12
AP	AK2	D31	C17	INTR	AD34	TDO	N33	Vcc3	AG37	Vss	AM14
APCHK#	AE5	D32	C15	INV	U5	TMS	P34	Vcc3	AN21	Vss	AM16
BE0#	AL9	D33	D16	KEN#	W5	TRST#	Q33	Vcc3	AN23	Vss	AM18
BE1#	AK10	D34	C13	LOCK#	AH4	Vcc2	A7	Vcc3	AN25	Vss	AM20
BE2#	AL11	D35	D14	MI/O#	T4	Vcc2	A9	Vcc3	AN27	Vss	AM22
BE3#	AK12	D36	C11	NA#	Y5	Vcc2	A11	Vcc3	AN29	Vss	AM24
BE4#	AL13	D37	D12	NC	A3	Vcc2	A13	Vcc2DET	AL1	Vss	AM26
BE5#	AK14	D38	C9	NC	A37	Vcc2	A15	Vss	B6	Vss	AM28
BE6#	AL15	D39	D10	NC	B2	Vcc2	A17	Vss	B8	Vss	AM30
BE7#	AK16	D40	D8	NC	C1	Vcc2	G1	Vss	B10	Vss	AN37
BOFF#	Z4	D41	A5	NC	H34	Vcc2	J1	Vss	B12	W/R#	AM6
BRDY#	X4	D42	E9	NC	J33	Vcc2	L1	Vss	B14	WB/WT#	AA5
BRDYC#	Y3	D43	B4	NC	L35	Vcc2	N1	Vss	B16	WM_RST	AA33
BREQ	AJ1	D44	D6	NC	P4	Vcc2	Q1	Vss	B18		
CACHE#	U3	D45	C5	NC	Q35	Vcc2	S1	Vss	B20		
CLK	AK18	D46	E7	NC	R34	Vcc2	U1	Vss	B22		
CLKMUL0	Y33	D47	C3	NC	S3	Vcc2	W1	Vss	B24		

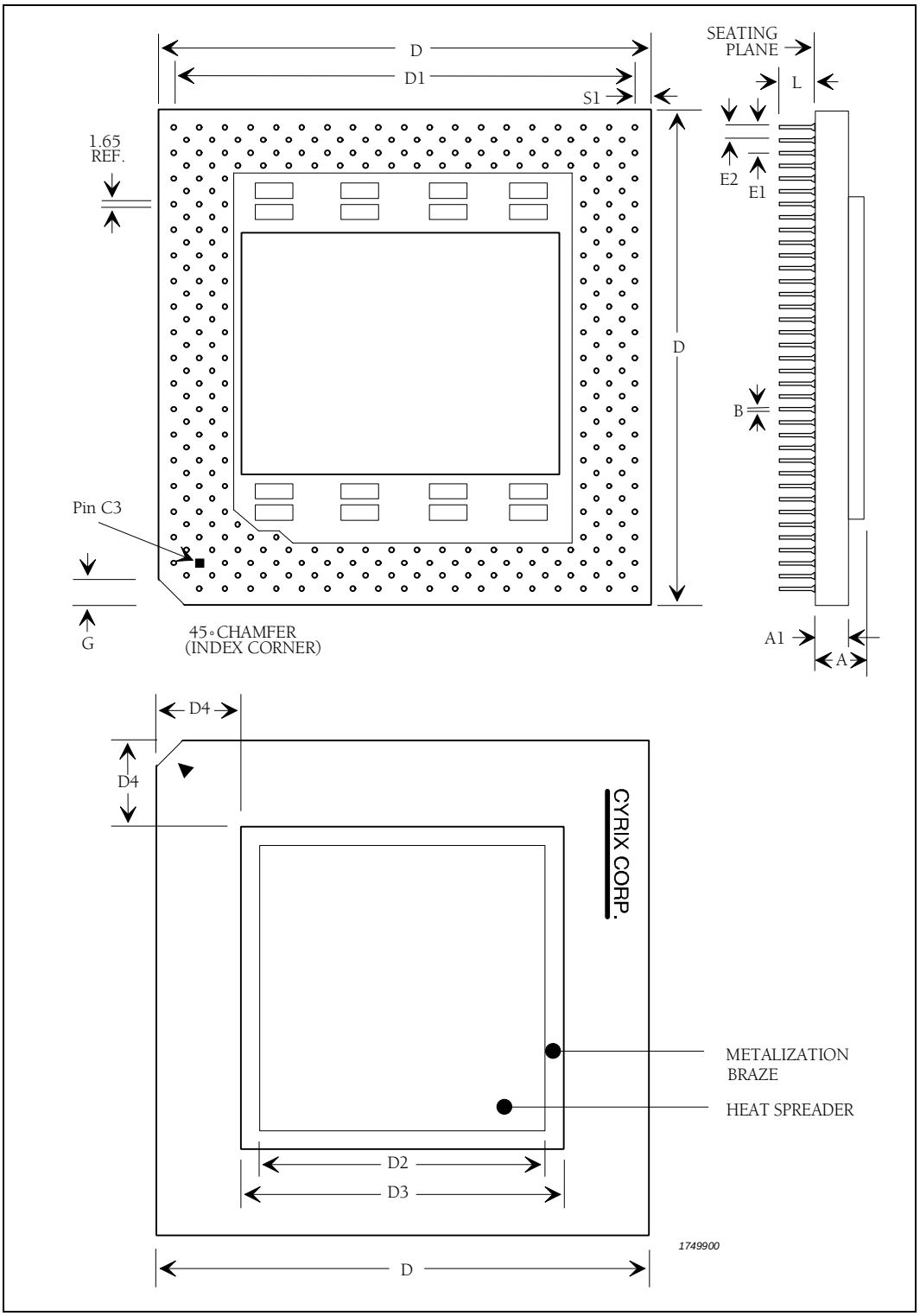


Figure 5-3. 296-Pin SPGA Package

Table 5-3. 296-Pin SPGA Package Dimensions

SYMBOL	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	3.43	4.34	0.135	0.171
A1	2.51	3.07	0.099	0.121
B	0.43	0.51	0.017	0.020
D	49.28	49.91	1.940	1.965
D1	45.47	45.97	1.790	1.810
D2	31.37 Sq.	32.13 Sq.	1.235	1.265
D3	33.43	34.42	1.316	1.355
D4	7.49	6.71	0.295	0.264
E1	2.41	2.67	0.095	0.105
E2	1.14	1.40	0.045	0.055
G	1.52	2.29	0.060	0.090
L	2.97	3.38	0.117	0.133
S1	1.65	2.16	0.065	0.085

5.2 Thermal Resistances

Three thermal resistances can be used to idealize the heat flow from the junction of the 6x86MX CPU to ambient:

θ_{JC} = thermal resistance from junction to case in °C/W

θ_{CS} = thermal resistance from case to heatsink in °C/W,

θ_{SA} = thermal resistance from heatsink to ambient in °C/W,

$\theta_{CA} = \theta_{CS} + \theta_{SA}$, thermal resistance from case to ambient in °C/W.

$T_C = T_A + P * \theta_{CA}$ (where T_A = ambient temperature and P = power applied to the CPU).

To maintain the case temperature under 70°C during operation θ_{CA} can be reduced by a heat-sink/fan combination. (The heatsink/fan decreases θ_{CA} by a factor of three compared to using a heatsink alone.) The required θ_{CA} to maintain 70°C is shown in Table 5-4. The designer should ensure that adequate air flow is maintained to control the ambient temperature (T_A).

Table 5-4. Required θ_{CA} to Maintain 70°C Case Temperature

Frequency (MHz)	Power* (W)	θ_{CA} For Different Ambient Temperatures				
		25°C	30°C	35°C	40°C	45°C
150	16.7	2.68	2.39	2.09	1.79	1.49
166	18.1	2.48	2.20	1.92	1.65	1.37
188	20.6	2.17	1.93	1.69	1.45	1.20
200	22.0	2.04	1.81	1.58	1.35	1.13
225	24.0	1.87	1.66	1.45	1.24	1.03
233	24.7	1.81	1.61	1.41	1.20	1.00

*Note: Power based on Max Active Power values from Table 4-6, Page 4-5.
Refer to the Cyrix Application AP105 titled "Thermal Design Considerations" for more information.

A typical θ_{JC} value for the 6x86MX 296-pin PGA-package value is 0.5 °C/W.

[redacted]

6. INSTRUCTION SET

This section summarizes the 6x86MX CPU instruction set and provides detailed information on the instruction encodings.

All instructions are listed in CPU, FPU and MMX Instruction Set Summary Tables shown on pages 6-14, 6-31 and 6-38. These tables provide information on the instruction encoding, and the instruction clock counts for each instruction. The clock count values for these tables are based on the assumptions described in Section 6.3.

Depending on the instruction, the 6x86MX CPU instructions follow the general instruction format shown in Table 6-1. These instructions vary in length and can start at any byte address.

6.1 Instruction Set Format

An instruction consists of one or more bytes that can include: prefix byte(s), at least one opcode byte(s), mod r/m byte, s-i-b byte, address displacement byte(s) and immediate data byte(s). An instruction can be as short as one byte and as long as 15 bytes. If there are more than 15 bytes in the instruction a general protection fault (error code of 0) is generated.

Table 6-1. Instruction Set Format

PREFIX	OPCODE	REGISTER AND ADDRESS MODE SPECIFIER						ADDRESS DISPLACEMENT	IMMEDIATE DATA
		mod r/m Byte			s-i-b Byte				
		mod	reg	r/m	ss	Index	Base		
0 or More Bytes	1 or 2 Bytes	7 - 6	5 - 3	2 - 0	7 - 6	5 - 3	2 - 0	0, 8, 16, or 32 Bits	0, 8, 16, or 32 Bits

6.2 General Instruction Format

The fields in the general instruction format at the byte level are listed in Table 6-2.

Table 6-2. Instruction Fields

FIELD NAME		DESCRIPTION	REFERENCE
Prefix		Segment register override Address size Operand size Repeat elements in string instructions LOCK# assertion	6.2.1 (Page 6-3)
Opcode		Instruction operation	6.2.2 (Page 6-4)
mod	Address Mode Specifier	Used with r/m field to select address mode	6.2.3 (Page 6-6)
reg	General Register Specifier	Uses reg, sreg2 or sreg3 encoding depending on opcode field	6.2.4 (Page 6-7)
r/m	Address Mode Specifier	Used with mod field to select addressing mode.	6.2.3 (Page 6-6)
ss	Scale Factor	Scaled-index address mode	6.2.5 (Page 6-9)
Index		Determines general register to be selected as index register	6.2.6 (Page 6-9)
Base		Determines general register to be selected as base register	6.2.7 (Page 6-10)
Address Displacement		Determines address displacement	
Immediate data		Immediate data operand used by instruction	

6.2.1 Prefix Field

Prefix bytes can be placed in front of any instruction. The prefix modifies the operation of the next instruction only. When more than one prefix is used, the order is not important. There are five type of prefixes as follows:

1. Segment Override explicitly specifies which segment register an instruction will use for effective address calculation.

2. Address Size switches between 16- and 32-bit addressing. Selects the inverse of the default.

3. Operand Size switches between 16- and 32-bit operand size. Selects the inverse of the default.

4. Repeat is used with a string instruction which causes the instruction to be repeated for each element of the string.

5. Lock is used to assert the hardware LOCK# signal during execution of the instruction.

Table 6-3 lists the encodings for each of the available prefix bytes.

Table 6-3. Instruction Prefix Summary

PREFIX	ENCODING	DESCRIPTION
ES:	26h	Override segment default, use ES for memory operand
CS:	2Eh	Override segment default, use CS for memory operand
SS:	36h	Override segment default, use SS for memory operand
DS:	3Eh	Override segment default, use DS for memory operand
FS:	64h	Override segment default, use FS for memory operand
GS:	65h	Override segment default, use GS for memory operand
Operand Size	66h	Make operand size attribute the inverse of the default
Address Size	67h	Make address size attribute the inverse of the default
LOCK	F0h	Assert LOCK# hardware signal.
REPNE	F2h	Repeat the following string instruction.
REP/REPE	F3h	Repeat the following string instruction.

6.2.2 Opcode Field

The opcode field specifies the operation to be performed by the instruction. The opcode field is either one or two bytes in length and may be further defined by additional bits in the mod r/m byte. Some operations have more than one opcode, each specifying a different form of the operation. Some opcodes name instruction groups. For example, opcode 80h names a group of operations that have an immediate operand and a register or memory operand. The reg field may appear in the second opcode byte or in the mod r/m byte.

6.2.2.1 Opcode Field: w Bit

The 1-bit w bit (Table 6-4) selects the operand size during 16 and 32 bit data operations.

Table 6-4. w Field Encoding

w BIT	OPERAND SIZE	
	16-BIT DATA OPERATIONS	32-BIT DATA OPERATIONS
0	8 Bits	8 Bits
1	16 Bits	32 Bits

6.2.2.2 Opcode Field: d Bit

The d bit (Table 6-11) determines which operand is taken as the source operand and which operand is taken as the destination.

Table 6-5. d Field Encoding

d BIT	DIRECTION OF OPERATON	SOURCE OPERAND	DESTINATION OPERAND
0	Register --> Register or Register --> Memory	reg	mod r/m or mod ss-index-base
1	Register --> Register or Memory --> Register	mod r/m or mod ss-index-base	reg

6.2.2.3 Opcode Field: s Bit

The s bit (Table 6-11) determines the size of the immediate data field. If the S bit is set, the immediate field of the OP code is 8-bits wide and is sign extended to match the operand size of the opcode.

Table 6-6. s Field Encoding

s FIELD	IMMEDIATE FIELD SIZE		
	8-BIT OPERAND SIZE	16-BIT OPERAND SIZE	32-BIT OPERAND SIZE
0 (or not present)	8 bits	16 bits	32 bits
1	8 bits	8 bits (sign extended)	8 bits (sign extended)

6.2.2.4 Opcode Field: eee Bits

The eee field (Table 6-7) is used to select the control, debug and test registers in the MOV instructions. The type of register and base registers selected by the eee bits are listed in Table 6-7. The values shown in Table 6-7 are the only valid encodings for the eee bits.

Table 6-7. eee Field Encoding

eee BITS	REGISTER TYPE	BASE REGISTER
000	Control Register	CR0
010	Control Register	CR2
011	Control Register	CR3
100	Control Register	CR4
000	Debug Register	DR0
001	Debug Register	DR1
010	Debug Register	DR2
011	Debug Register	DR3
110	Debug Register	DR6
111	Debug Register	DR7
011	Test Register	TR3
100	Test Register	TR4
101	Test Register	TR5
110	Test Register	TR6
111	Test Register	TR7

6.2.3 mod and r/m Fields

The mod and r/m fields (Table 6-8), within the mod r/m byte, select the type of memory addressing to be used. Some instructions use a fixed addressing mode (e.g., PUSH or POP) and therefore, these fields are not present. Table 6-8 lists the addressing method when 16-bit addressing is used and a mod r/m byte is present. Some mod r/m field encodings are dependent on the w field and are shown in Table 6-9 (Page 6-7).

Table 6-8. mod r/m Field Encoding

mod and r/m fields	16-BIT ADDRESS MODE with mod r/m Byte	32-BIT ADDRESS MODE with mod r/m Byte and No s-i-b Byte Present
00 000	DS:[BX+SI]	DS:[EAX]
00 001	DS:[BX+DI]	DS:[ECX]
00 010	DS:[BP+SI]	DS:[EDX]
00 011	DS:[BP+DI]	DS:[EBX]
00 100	DS:[SI]	Note 1
00 101	DS:[DI]	DS:[d32]
00 110	DS:[d16]	DS:[ESI]
00 111	DS:[BX]	DS:[EDI]
01 000	DS:[BX+SI+d8]	DS:[EAX+d8]
01 001	DS:[BX+DI+d8]	DS:[ECX+d8]
01 010	DS:[BP+SI+d8]	DS:[EDX+d8]
01 011	DS:[BP+DI+d8]	DS:[EBX+d8]
01 100	DS:[SI+d8]	Note 1
01 101	DS:[DI+d8]	SS:[EBP+d8]
01 110	SS:[BP+d8]	DS:[ESI+d8]
01 111	DS:[BX+d8]	DS:[EDI+d8]
10 000	DS:[BX+SI+d16]	DS:[EAX+d32]
10 001	DS:[BX+DI+d16]	DS:[ECX+d32]
10 010	DS:[BP+SI+d16]	DS:[EDX+d32]
10 011	DS:[BP+DI+d16]	DS:[EBX+d32]
10 100	DS:[SI+d16]	Note 1
10 101	DS:[DI+d16]	SS:[EBP+d32]
10 110	SS:[BP+d16]	DS:[ESI+d32]
10 111	DS:[BX+d16]	DS:[EDI+d32]
11 000 through 11 111	See Table 6-9 (Page 6-7)	

Note 1: An "s-i-d" (ss, Index, Base) field is present. Refer to the ss Table 6-13 (Page 6-9), Index Table 6-14 (Page 6-9) and Base Table 6-15 (Page 6-10).

Table 6-9. mod r/m Field Encoding Dependent on w Field

mod r/m	16-BIT OPERATION w = 0	16-BIT OPERATION w = 1	32-BIT OPERATION w = 0	32-BIT OPERATION w = 1
11 000	AL	AX	AL	EAX
11 001	CL	CX	CL	ECX
11 010	DL	DX	DL	EDX
11 011	BL	BX	BL	EBX
11 100	AH	SP	AH	ESP
11 101	CH	BP	CH	EBP
11 110	DH	SI	DH	ESI
11 111	BH	DI	BH	EDI

6.2.4 reg Field

The reg field (Table 6-10) determines which general registers are to be used. The selected register is dependent on whether a 16 or 32 bit operation is current and the status of the w bit.

Table 6-10. reg Field

reg	16-BIT OPERATION w Field Not Present	32-BIT OPERATION w Field Not Present	16-BIT OPERATION w = 0	16-BIT OPERATION w = 1	32-BIT OPERATION w = 0	32-BIT OPERATION w = 1
000	AX	EAX	AL	AX	AL	EAX
001	CX	ECX	CL	CX	CL	ECX
010	DX	EDX	DL	DX	DL	EDX
011	BX	EBX	BL	BX	BL	EBX
100	SP	ESP	AH	SP	AH	ESP
101	BP	EBP	CH	BP	CH	EBP
110	SI	ESI	DH	SI	DH	ESI
111	DI	EDI	BH	DI	BH	EDI

6.2.4.1 reg Field: sreg3 Encoding

The sreg3 field (Table 6-11) is 3-bit field that is similar to the sreg2 field, but allows use of the FS and GS segment registers.

Table 6-11. sreg3 Field Encoding

sreg3 FIELD	SEGMENT REGISTER SELECTED
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS
110	undefined
111	undefined

6.2.4.2 reg Field: sreg2 Encoding

The sreg2 field (Table 6-4) is a 2-bit field that allows one of the four 286-type segment registers to be specified.

Table 6-12. sreg2 Field Encoding

sreg2 FIELD	SEGMENT REGISTER SELECTED
00	ES
01	CS
10	SS
11	DS

6.2.5 ss Field

The ss field (Table 6-13) specifies the scale factor used in the offset mechanism for address calculation. The scale factor multiplies the index value to provide one of the components used to calculate the offset address.

Table 6-13. *ss* Field Encoding

ss FIELD	SCALE FACTOR
00	x1
01	x2
01	x4
11	x8

6.2.6 Index Field

The index field (Table 6-14) specifies the index register used by the offset mechanism for offset address calculation. When no index register is used (index field = 100), the ss value must be 00 or the effective address is undefined.

Table 6-14. Index Field Encoding

Index FIELD	INDEX REGISTER
000	EAX
001	ECX
010	EDX
011	EBX
100	none
101	EBP
110	ESI
111	EDI

6.2.7 Base Field

In Table 6-8 (Page 6-6), the note "s-i-b present" for certain entries forces the use of the mod and base field as listed in Table 6-15. The first two digits in the first column of Table 6-15 identifies the mod bits in the mod r/m byte. The last three digits in the first column of this table identifies the base fields in the s-i-b byte.

Table 6-15. mod base Field Encoding

mod FIELD WITHIN mode/rm BYTE	base FIELD WITHIN s-i-b BYTE	32-BIT ADDRESS MODE with mod r/m and s-i-b Bytes Present
00	000	DS:[EAX+(scaled index)]
00	001	DS:[ECX+(scaled index)]
00	010	DS:[EDX+(scaled index)]
00	011	DS:[EBX+(scaled index)]
00	100	SS:[ESP+(scaled index)]
00	101	DS:[d32+(scaled index)]
00	110	DS:[ESI+(scaled index)]
00	111	DS:[EDI+(scaled index)]
01	000	DS:[EAX+(scaled index)+d8]
01	001	DS:[ECX+(scaled index)+d8]
01	010	DS:[EDX+(scaled index)+d8]
01	011	DS:[EBX+(scaled index)+d8]
01	100	SS:[ESP+(scaled index)+d8]
01	101	SS:[EBP+(scaled index)+d8]
01	110	DS:[ESI+(scaled index)+d8]
01	111	DS:[EDI+(scaled index)+d8]
10	000	DS:[EAX+(scaled index)+d32]
10	001	DS:[ECX+(scaled index)+d32]
10	010	DS:[EDX+(scaled index)+d32]
10	011	DS:[EBX+(scaled index)+d32]
10	100	SS:[ESP+(scaled index)+d32]
10	101	SS:[EBP+(scaled index)+d32]
10	110	DS:[ESI+(scaled index)+d32]
10	111	DS:[EDI+(scaled index)+d32]

6.3 CPUID Instruction

The 6x86MX CPU executes the CPUID instruction (opcode 0FA2) as documented in this section only if the CPUID bit in the CCR4 configuration register is set. The CPUID instruction may be used by software to determine the vendor and type of CPU.

When the CPUID instruction is executed with EAX = 0, the ASCII characters "CyrixInstead" are placed in the EBX, EDX, and ECX registers as shown in Table 6-16:

Table 6-16. CPUID Data Returned When EAX = 0

REGISTER	CONTENTS (D31 - D0)
EBX	69 72 79 43 i r y C*
EDX	73 6E 49 78 s n I x*
ECX	64 61 65 74 d a e t*

*ASCII equivalent

When the CPUID instruction is executed with EAX = 1, EAX and EDX contain the values shown in Table 6-17.

Table 6-17. CPUID Data Returned When EAX = 1

REGISTER	CONTENTS
EAX[7 - 0]	00h
EAX[15 - 8]	06h
EDX[0]	1 = FPU Built In
EDX[1]	0 = No V86 Enhancements
EDX[2]	1 = I/O Breakpoints
EDX[3]	0 = No Page Size Extensions
EDX[4]	1 = Time Stamp Counter
EDX[5]	1 = RDMSR and WRMSR
EDX[6]	0 = No Physical Address Extensions
EDX[7]	0 = No Machine Check Exception
EDX[8]	1 = CMPXCHG8B Instruction
EDX[9]	0 = No APIC
EDX[11 - 10]	0 = Undefined
EDX[12]	0 = No Memory Type Range Registers
EDX[13]	1 = PTE Global Bit
EDX[14]	0 = No Machine Check Architecture
EDX[15]	1 = CMOV, FCMOV, FCOMI Instructions
EDX[22 - 16]	0 = Undefined
EDX[23]	1 = MMX Instructions
EDX[31 - 24]	0 = Undefined

6.4 Instruction Set Tables

The 6x86MX CPU instruction set is presented in three tables: Table 6-21. "6x86MX CPU Instruction Set Clock Count Summary" on page 6-14, Table 6-23. "6x86MX FPU Instruction Set Summary" on page 6-31 and the Table 6-25. "6x86MX Processor MMX Instruction Set Clock Count Summary" on page 6-38. Additional information concerning the FPU Instruction Set is presented on page 6-30, and the 6x86MX MMX instruction set on page 6-37.

6.4.1 Assumptions Made in Determining Instruction Clock Count

The assumptions made in determining instruction clock counts are listed below:

1. All clock counts refer to the internal CPU internal clock frequency.
2. The instruction has been prefetched, decoded and is ready for execution.
3. Bus cycles do not require wait states.
4. There are no local bus HOLD requests delaying processor access to the bus.
5. No exceptions are detected during instruction execution.
6. If an effective address is calculated, it does not use two general register components. One register, scaling and displacement can be used within the clock count shown. However, if the effective address calculation uses two general register components, add 1 clock to the clock count shown.
7. All clock counts assume aligned 32-bit memory/IO operands.
8. If instructions access a 32-bit operand that crosses a 64-bit boundary, add 1 clock for read or write and add 2 clocks for read and write.
9. For non-cached memory accesses, add two clocks (6x86MX CPU with 2x clock) or four clocks (6x86MX CPU with 3x clock). (Assumes zero wait state memory accesses).
10. Locked cycles are not cacheable. Therefore, using the LOCK prefix with an instruction adds additional clocks as specified in paragraph 9 above.
11. No parallel execution of instructions.

6.4.2 CPU Instruction Set Summary Table Abbreviations

The clock counts listed in the CPU Instruction Set Summary Table are grouped by operating mode and whether there is a register/cache hit or a cache miss. In some cases, more than one clock count is shown in a column for a given instruction, or a variable is used in the clock count. The abbreviations used for these conditions are listed in Table 6-18.

Table 6-18. CPU Clock Count Abbreviations

CLOCK COUNT SYMBOL	EXPLANATION
/	Register operand/memory operand.
n	Number of times operation is repeated.
L	Level of the stack frame.
|	Conditional jump taken | Conditional jump not taken. (e.g. "4|1" = 4 clocks if jump taken, 1 clock if jump not taken)
\	CPL ≤ IOPL \ CPL > IOPL (where CPL = Current Privilege Level, IOPL = I/O Privilege Level)
m	Number of parameters passed on the stack.

6.4.3 CPU Instruction Set Summary Table Flags Table

The CPU Instruction Set Summary Table lists nine flags that are affected by the execution of instructions. The conventions shown in Table 6-19 are used to identify the different flags. Table 6-20 lists the conventions used to indicate what action the instruction has on the particular flag.

Table 6-19. Flag Abbreviations

ABBREVIATION	NAME OF FLAG
OF	Overflow Flag
DF	Direction Flag
IF	Interrupt Enable Flag
TF	Trap Flag
SF	Sign Flag
ZF	Zero Flag
AF	Auxiliary Flag
PF	Parity Flag
CF	Carry Flag

Table 6-20. Action of Instruction on Flag

INSTRUCTION TABLE SYMBOL	ACTION
x	Flag is modified by the instruction.
-	Flag is not changed by the instruction.
0	Flag is reset to "0".
1	Flag is set to "1".
u	Flag is undefined following execution of the instruction.

Table 6-21. 6x86MX CPU Instruction Set Clock Count Summary

INSTRUCTION	OPCODE	FLAGS	REAL MODE CLOCK COUNT	PROTECTED MODE CLOCK COUNT	NOTES	
		OF DF IF TF SF ZF AF PF CF	Reg/ Cache Hit	Reg/ Cache Hit	Real Mode	Protected Mode
AAA *ASCII Adjust AL after Add*	37	u - - - u u x u x	7	7		
AAD *ASCII Adjust AX before Divide*	D5 0A	u - - - x x u x u	7	7		
AAM ASCII *Adjust AX after Multiply*	D4 0A	u - - - x x u x u	13-21	13-21		
AAS ASCII *Adjust AL after Subtract*	3F	u - - - u u x u x	7	7		
ADC *Add with Carry*		x - - - x x x x x			b	h
Register to Register	1 [00dw] [11 reg r/m]		1	1		
Register to Memory	1 [000w] [mod reg r/m]		1	1		
Memory to Register	1 [001w] [mod reg r/m]		1	1		
Immediate to Register/Memory	8 [00sw] [mod 010 r/m]###		1	1		
Immediate to Accumulator	1 [010w] ###		1	1		
ADD *Integer Add*		x - - - x x x x x			b	h
Register to Register	0 [00dw] [11 reg r/m]		1	1		
Register to Memory	0 [000w] [mod reg r/m]		1	1		
Memory to Register	0 [001w] [mod reg r/m]		1	1		
Immediate to Register/Memory	8 [00sw] [mod 000 r/m]###		1	1		
Immediate to Accumulator	0 [010w] ###		1	1		
AND *Boolean AND*		0 - - - x x u x 0			b	h
Register to Register	2 [00dw] [11 reg r/m]		1	1		
Register to Memory	2 [000w] [mod reg r/m]		1	1		
Memory to Register	2 [001w] [mod reg r/m]		1	1		
Immediate to Register/Memory	8 [00sw] [mod 100 r/m]###		1	1		
Immediate to Accumulator	2 [010w] ###		1	1		
ARPL *Adjust Requested Privilege Level*		- - - - - x - - -		9	a	h
From Register/Memory	63 [mod reg r/m]					
BOUND *Check Array Boundaries*	62 [mod reg r/m]	- - - - - - - - -			b, e	g,h,j,k,r
If Out of Range (Int 5)			20	20+INT		
If In Range			11	11		
BSF *Scan Bit Forward*		- - - - - x - - -	3	3	b	h
Register, Register/Memory	0F BC [mod reg r/m]					
BSR *Scan Bit Reverse*		- - - - - x - - -	3	3	b	h
Register, Register/Memory	0F BD [mod reg r/m]					
BSWAP *Byte Swap*	0F C[1 reg]	- - - - - - - - -	4	4		
BT *Test Bit*		- - - - - - - - x			b	h
Register/Memory, Immediate	0F BA [mod 100 r/m]#		2	2		
Register/Memory, Register	0F A3 [mod reg r/m]		5/6	5/6		
BTC *Test Bit and Complement*		- - - - - - - - x			b	h
Register/Memory, Immediate	0F BA [mod 111 r/m]#		3	3		
Register/Memory, Register	0F BB [mod reg r/m]		5/6	5/6		

\# = immediate 8-bit data
= immediate 16-bit data
= full immediate 32-bit data (8, 16, 32 bits)

\+ = 8-bit signed displacement
+++ = full signed displacement (16, 32 bits)

x = modified
\- = unchanged
u = undefined

Table 6-21. 6x86MX CPU Instruction Set Clock Count Summary (Continued)

INSTRUCTION	OPCODE	FLAGS	REAL MODE CLOCK COUNT	PROTECTED MODE CLOCK COUNT	NOTES	
		OF DF IF TF SF ZF AF PF CF	Reg/ Cache Hit	Reg/ Cache Hit	Real Mode	Protected Mode
BTR *Test Bit and Reset*		- - - - - - - - - x			b	h
Register/Memory, Immediate	0F BA [mod 110 r/m]#		3	3		
Register/Memory, Register	0F B3 [mod reg r/m]		5/6	5/6		
BTS *Test Bit and Set*		- - - - - - - - - x			b	h
Register/Memory	0F BA [mod 101 r/m]		3	3		
Register (short form)	0F AB [mod reg r/m]		5/6	5/6		
CALL *Subroutine Call*		- - - - - - - - - -			b	h,j,k,r
Direct Within Segment	E8 +++		1	1		
Register/Memory Indirect Within Segment	FF [mod 010 r/m]		1/3	1/3		
Direct Intersegment	9A [unsigned full offset,		3	4		
Call Gate to Same Privilege	selector]			15		
Call Gate to Different Privilege No Parameters				26		
Call Gate to Different Privilege m Par's				35+2m		
16-bit Task to 16-bit TSS				110		
16-bit Task to 32-bit TSS				118		
16-bit Task to V86 Task				96		
32-bit Task to 16-bit TSS				112		
32-bit Task to 32-bit TSS				120		
32-bit Task to V86 Task				98		
Indirect Intersegment	FF [mod 011 r/m]		5	8		
Call Gate to Same Privilege				20		
Call Gate to Different Privilege No Parameters				31		
Call Gate to Different Privilege Level m Par's				40+2m		
16-bit Task to 16-bit TSS				114		
16-bit Task to 32-bit TSS				122		
16-bit Task to V86 Task				100		
32-bit Task to 16-bit TSS				116		
32-bit Task to 32-bit TSS				124		
32-bit Task to V86 Task				102		
CBW *Convert Byte to Word*	98	- - - - - - - - - -	3	3		
CDQ *Convert Doubleword to Quadword*	99	- - - - - - - - - -	2	2		
CLC *Clear Carry Flag*	F8	- - - - - - - - - 0	1	1		
CLD *Clear Direction Flag*	FC	- 0 - - - - - - - -	7	7		
CLI *Clear Interrupt Flag*	FA	- - 0 - - - - - - -	7	7		m
CLTS *Clear Task Switched Flag*	0F 06	- - - - - - - - - -	10	10	c	l
CMC *Complement the Carry Flag*	F5	- - - - - - - - - x	2	2		

\# = immediate 8-bit data
= immediate 16-bit data
= full immediate 32-bit data (8, 16, 32 bits)

\+ = 8-bit signed displacement
+++ = full signed displacement (16, 32 bits)

x = modified
- = unchanged
u = undefined

Table 6-21. 6x86MX CPU Instruction Set Clock Count Summary (Continued)

INSTRUCTION	OPCODE	FLAGS	REAL MODE CLOCK COUNT	PROTECTED MODE CLOCK COUNT	NOTES	
		OF DF IF TF SF ZF AF PF CF	Reg/ Cache Hit	Reg/ Cache Hit	Real Mode	Protected Mode
CMP *Compare Integers* Register to Register Register to Memory Memory to Register Immediate to Register/Memory Immediate to Accumulator	 3 [10dw] [11 reg r/m] 3 [101w] [mod reg r/m] 3 [100w] [mod reg r/m] 8 [00sw] [mod 111 r/m] ### 3 [110w] ###	x - - - x x x x x	 1 1 1 1 1	 1 1 1 1 1	b	h
CMOVA/CMOVNBE *Move if Above/ Not Below or Equal* Register, Register/Memory	0F 47 [mod reg r/m]	- - - - - - - - - -	1	1		r
CMOVBE/CMOVNA *Move if Below or Equal/ Not Above* Register, Register/Memory	0F 46 [mod reg r/m]	- - - - - - - - - -	1	1		r
CMOVAE/CMOVNB/CMOVNC/ *Move if Above or Equal/Not Below/Not Carry* Register, Register/Memory	0F 43 [mod reg r/m]	- - - - - - - - - -	1	1		r
CMOVB/CMOVC/CMOVNAE *Move if Below/ Carry/Not Above or Equal* Register, Register/Memory	0F 42 [mod reg r/m]	- - - - - - - - - -	1	1		r
CMOVE/CMOVZ *Move if Equal/Zero* Register, Register/Memory	0F 44 [mod reg r/m]	- - - - - - - - - -	1	1		r
CMOVNE/CMOVNZ *Move if Not Equal/ Not Zero* Register, Register/Memory	0F 45 [mod reg r/m]	- - - - - - - - - -	1	1		r
CMOVG/CMOVNLE *Move if Greater/ Not Less or Equal* Register, Register/Memory	0F 4F [mod reg r/m]	- - - - - - - - - -	1	1		r
CMOVLE/CMOVNG *Move if Less or Equal/ Not Greater* Register, Register/Memory	0F 4E [mod reg r/m]	- - - - - - - - - -	1	1		r
CMOVL/CMOVNGE *Move if Less/ Not Greater or Equal* Register, Register/Memory	0F 4C [mod reg r/m]	- - - - - - - - - -	1	1		r
CMOVGE/CMOVNL *Move if Greater or Equal/ Not Less* Register, Register/Memory	0F 4D [mod reg r/m]	- - - - - - - - - -	1	1		r

\# = immediate 8-bit data
= immediate 16-bit data
= full immediate 32-bit data (8, 16, 32 bits)

\+ = 8-bit signed displacement
+++ = full signed displacement (16, 32 bits)

x = modified
\- = unchanged
u = undefined

Table 6-21. 6x86MX CPU Instruction Set Clock Count Summary (Continued)

INSTRUCTION	OPCODE	FLAGS	REAL MODE CLOCK COUNT	PROTECTED MODE CLOCK COUNT	NOTES	
		OF DF IF TF SF ZF AF PF CF	Reg/ Cache Hit	Reg/ Cache Hit	Real Mode	Protected Mode
CMOVO *Move if Overflow* Register, Register/Memory	0F 40 [mod reg r/m]	- - - - - - - - -	1	1		r
CMOVNO *Move if No Overflow* Register, Register/Memory	0F 41 [mod reg r/m]	- - - - - - - - -	1	1		r
CMOVP/CMOVPE *Move if Parity/Parity Even* Register, Register/Memory	0F 4A [mod reg r/m]	- - - - - - - - -	1	1		r
CMONP/CMOVPO *Move if Not Parity/ Parity Odd* Register, Register/Memory	0F 4B [mod reg r/m]	- - - - - - - - -	1	1		r
CMOVS *Move if Sign* Register, Register/Memory	0F 48 [mod reg r/m]	- - - - - - - - -	1	1		r
CMOVNS *Move if Not Sign* Register, Register/Memory	0F 49 [mod reg r/m]	- - - - - - - - -	1	1		r
CMPS *Compare String*	A [011w]	x - - - x x x x x	5	5	b	h
CMPXCHG *Compare and Exchange* Register1, Register2 Memory, Register	0F B [000w] [11 reg2 reg1] 0F B [000w] [mod reg r/m]	x - - - x x x x x	11 11	11 11		
CMPXCHG8B *Compare and Exchange* 8 Bytes	0F C7 [mod 001 r/m]	- - - - - - - - -				
CPUID *CPU Identification*	0F A2	- - - - - - - - -	12	12		
CWD *Convert Word to Doubleword*	99	- - - - - - - - -	2	2		
CWDE *Convert Word to Doubleword Extended*	98	- - - - - - - - -	2	2		
DAA *Decimal Adjust AL after Add*	27	- - - - x x x x x	9	9		
DAS *Decimal Adjust AL after Subtract*	2F	- - - - x x x x x	9	9		
DEC *Decrement by 1* Register/Memory Register (short form)	F [111w] [mod 001 r/m] 4 [1 reg]	x - - - x x x x -	1 1	1 1	b	h
DIV *Unsigned Divide* Accumulator by Register/Memory Divisor: Byte Word Doubleword	F [011w] [mod 110 r/m]	- - - - x x u u -	13-17 13-25 13-41	13-17 13-25 13-41	b,e	e,h

\# = immediate 8-bit data
\## = immediate 16-bit data
= full immediate 32-bit data (8, 16, 32 bits)

\+ = 8-bit signed displacement
+++ = full signed displacement (16, 32 bits)

x = modified
- = unchanged
u = undefined

Table 6-21. 6x86MX CPU Instruction Set Clock Count Summary (Continued)

INSTRUCTION	OPCODE	FLAGS	REAL MODE CLOCK COUNT	PROTECTED MODE CLOCK COUNT	NOTES	
		OF DF IF TF SF ZF AF PF CF	Reg/ Cache Hit	Reg/ Cache Hit	Real Mode	Protected Mode
ENTER *Enter New Stack Frame*	C8 ##,#	- - - - - - - - -			b	h
Level = 0			10	10		
Level = 1			13	13		
Level (L) > 1			10+L*3	10+L*3		
HLT Halt	F4	- - - - - - - - -	5	5		l
IDIV *Integer (Signed) Divide*		- - - - x x u u -			b,e	e,h
Accumulator by Register/Memory	F [011w] [mod 111 r/m]					
Divisor: Byte			16-20	16-20		
Word			16-28	16-28		
Doubleword			17-45	17-45		
IMUL *Integer (Signed) Multiply*		x - - - x x u u x			b	h
Accumulator by Register/Memory	F [011w] [mod 101 r/m]					
Multiplier: Byte			4	4		
Word			4	4		
Doubleword			10	10		
Register with Register/Memory	0F AF [mod reg r/m]					
Multiplier: Word			4	4		
Doubleword			10	10		
Register/Memory with Immediate to Register2						
Multiplier: Word	6 [10s1] [mod reg r/m] ###		5	5		
Doubleword			11	11		
IN *Input from I/O Port*		- - - - - - - - -				m
Fixed Port	E [010w] [#]		14	14/28		
Variable Port	E [110w]		14	14/28		
INC *Increment by 1*		x - - - x x x x -			b	h
Register/Memory	F [111w] [mod 000 r/m]		1	1		
Register (short form)	4 [0 reg]		1	1		
INS *Input String from I/O Port*	6 [110w]	- - - - - - - - -	14	14/28	b	h,m

\# = immediate 8-bit data
= immediate 16-bit data
= full immediate 32-bit data (8, 16, 32 bits)

\+ = 8-bit signed displacement
+++ = full signed displacement (16, 32 bits)

x = modified
\- = unchanged
u = undefined

Table 6-21. 6x86MX CPU Instruction Set Clock Count Summary (Continued)

INSTRUCTION	OPCODE	FLAGS	REAL MODE CLOCK COUNT	PROTECTED MODE CLOCK COUNT	NOTES	
		OF DF IF TF SF ZF AF PF CF	Reg/ Cache Hit	Reg/ Cache Hit	Real Mode	Protected Mode
INT *Software Interrupt*		- - x 0 - - - - -			b,e	g,j,k,r
INT i	CD #		9			
Protected Mode:						
Interrupt or Trap to Same Privilege				21		
Interrupt or Trap to Different Privilege				32		
16-bit Task to 16-bit TSS by Task Gate				114		
16-bit Task to 32-bit TSS by Task Gate				122		
16-bit Task to V86 by Task Gate				100		
16-bit Task to 16-bit TSS by Task Gate				116		
32-bit Task to 32-bit TSS by Task Gate				124		
32-bit Task to V86 by Task Gate				102		
V86 to 16-bit TSS by Task Gate				124		
V86 to 32-bit TSS by Task Gate				102		
V86 to Privilege 0 by Trap Gate/Int Gate				46		
INT 3	CC		INT	INT		
INTO	CE					
If OF==0			6	6		
If OF==1 (INT 4)				15+INT		
INVD *Invalidate Cache*	0F 08	- - - - - - - - -	12	12	t	t
INVLPG *Invalidate TLB Entry*	0F 01 [mod 111 r/m]	- - - - - - - - -	13	13		
IRET *Interrupt Return*	CF	x x x x x x x x x				g,h,j,k,r
Real Mode			7			
Protected Mode:						
Within Task to Same Privilege				10		
Within Task to Different Privilege				26		
16-bit Task to 16-bit Task				117		
16-bit Task to 32-bit TSS				125		
16-bit Task to V86 Task				103		
32-bit Task to 16-bit TSS				119		
32-bit Task to 32-bit TSS				127		
32-bit Task to V86 Task				105		
JB/JNAE/JC *Jump on Below/Not Above or Equal/ Carry*		- - - - - - - - -				r
8-bit Displacement	72 +		1	1		
Full Displacement	0F 82 +++		1	1		
JBE/JNA *Jump on Below or Equal/Not Above*		- - - - - - - - -				r
8-bit Displacement	76 +		1	1		
Full Displacement	0F 86 +++		1	1		

\# = immediate 8-bit data
= immediate 16-bit data
= full immediate 32-bit data (8, 16, 32 bits)

\+ = 8-bit signed displacement
+++ = full signed displacement (16, 32 bits)

x = modified
\- = unchanged
u = undefined

Table 6-21. 6x86MX CPU Instruction Set Clock Count Summary (Continued)

INSTRUCTION	OPCODE	FLAGS	REAL MODE CLOCK COUNT	PROTECTED MODE CLOCK COUNT	NOTES	
		OF DF IF TF SF ZF AF PF CF	Reg/ Cache Hit	Reg/ Cache Hit	Real Mode	Protected Mode
JCXZ/JECXZ *Jump on CX/ECX Zero*	E3 +	- - - - - - - - -	1	1		r
JE/JZ *Jump on Equal/Zero*		- - - - - - - - -				r
8-bit Displacement	74 +		1	1		
Full Displacement	0F 84 +++		1	1		
JL/JNGE *Jump on Less/Not Greater or Equal*		- - - - - - - - -				r
8-bit Displacement	7C +		1	1		
Full Displacement	0F 8C +++		1	1		
JLE/JNG *Jump on Less or Equal/Not Greater*		- - - - - - - - -				r
8-bit Displacement	7E +		1	1		
Full Displacement	0F 8E +++		1	1		
JMP *Unconditional Jump*		- - - - - - - - -			b	h,j,k,r
8-bit Displacement	EB +		1	1		
Full Displacement	E9 +++		1	1		
Register/Memory Indirect Within Segment	FF [mod 100 r/m]		1/3	1/3		
Direct Intersegment	EA [unsigned full offset, selector]		1	4		
Call Gate Same Privilege Level				14		
16-bit Task to 16-bit TSS				110		
16-bit Task to 32-bit TSS				118		
16-bit Task to V86 Task				96		
32-bit Task to 16-bit TSS				112		
32-bit Task to 32-bit TSS				120		
32-bit Task to V86 Task				98		
Indirect Intersegment	FF [mod 101 r/m]		5	7		
Call Gate Same Privilege Level				17		
16-bit Task to 16-bit TSS				113		
16-bit Task to 32-bit TSS				121		
16-bit Task to V86 Task				99		
32-bit Task to 16-bit TSS				115		
32-bit Task to 32-bit TSS				123		
32-bit Task to V86 Task				101		
JNB/JAE/JNC *Jump on Not Below/Above or Equal/Not Carry*		- - - - - - - - -				r
8-bit Displacement	73 +		1	1		
Full Displacement	0F 83 +++		1	1		
JNBE/JA *Jump on Not Below or Equal/Above*		- - - - - - - - -				r
8-bit Displacement	77 +		1	1		
Full Displacement	0F 87 +++		1	1		
JNE/JNZ *Jump on Not Equal/Not Zero*		- - - - - - - - -				r
8-bit Displacement	75 +		1	1		
Full Displacement	0F 85 +++		1	1		

\# = immediate 8-bit data
\## = immediate 16-bit data
\### = full immediate 32-bit data (8, 16, 32 bits)

\+ = 8-bit signed displacement
+++ = full signed displacement (16, 32 bits)

x = modified
\- = unchanged
u = undefined

Table 6-21. 6x86MX CPU Instruction Set Clock Count Summary (Continued)

INSTRUCTION	OPCODE	FLAGS	REAL MODE CLOCK COUNT	PROTECTED MODE CLOCK COUNT	NOTES	
		OF DF IF TF SF ZF AF PF CF	Reg/ Cache Hit	Reg/ Cache Hit	Real Mode	Protected Mode
JNL/JGE *Jump on Not Less/Greater or Equal*		- - - - - - - - -				r
8-bit Displacement	7D +		1	1		
Full Displacement	0F 8D +++		1	1		
JNLE/JG *Jump on Not Less or Equal/Greater*		- - - - - - - - -				r
8-bit Displacement	7F +		1	1		
Full Displacement	0F 8F +++		1	1		
JNO *Jump on Not Overflow*		- - - - - - - - -				r
8-bit Displacement	71 +		1	1		
Full Displacement	0F 81 +++		1	1		
JNP/JPO *Jump on Not Parity/Parity Odd*		- - - - - - - - -				r
8-bit Displacement	7B +		1	1		
Full Displacement	0F 8B +++		1	1		
JNS *Jump on Not Sign*		- - - - - - - - -				r
8-bit Displacement	79 +		1	1		
Full Displacement	0F 89 +++		1	1		
JO *Jump on Overflow*		- - - - - - - - -				r
8-bit Displacement	70 +		1	1		
Full Displacement	0F 80 +++		1	1		
JP/JPE *Jump on Parity/Parity Even*		- - - - - - - - -				r
8-bit Displacement	7A +		1	1		
Full Displacement	0F 8A +++		1	1		
JS *Jump on Sign*		- - - - - - - - -				r
8-bit Displacement	78 +		1	1		
Full Displacement	0F 88 +++		1	1		
LAHF *Load AH with Flags*	9F	- - - - - - - - -	2	2		
LAR *Load Access Rights*		- - - - - x - - -			a	g,h,j,p
From Register/Memory	0F 02 [mod reg r/m]			8		
LDS *Load Pointer to DS*	C5 [mod reg r/m]	- - - - - - - - -	2	4	b	h,i,j
LEA *Load Effective Address*	8D [mod reg r/m]	- - - - - - - - -				
No Index Register			1	1		
With Index Register			1	1		
LEAVE *Leave Current Stack Frame*	C9	- - - - - - - - -	4	4	b	h
LES *Load Pointer to ES*	C4 [mod reg r/m]	- - - - - - - - -	2	4	b	h,i,j

\# = immediate 8-bit data
\## = immediate 16-bit data
= full immediate 32-bit data (8, 16, 32 bits)

\+ = 8-bit signed displacement
+++ = full signed displacement (16, 32 bits)

x = modified
\- = unchanged
u = undefined

Table 6-21. 6x86MX CPU Instruction Set Clock Count Summary (Continued)

INSTRUCTION	OPCODE	FLAGS	REAL MODE CLOCK COUNT	PROTECTED MODE CLOCK COUNT	NOTES	
		OF DF IF TF SF ZF AF PF CF	Reg/ Cache Hit	Reg/ Cache Hit	Real Mode	Protected Mode
LFS *Load Pointer to FS*	0F B4 [mod reg r/m]	- - - - - - - - -	2	4	b	h,i,j
LGDT *Load GDT Register*	0F 01 [mod 010 r/m]	- - - - - - - - -	8	8	b,c	h,l
LGS *Load Pointer to GS*	0F B5 [mod reg r/m]	- - - - - - - - -	2	4	b	h,i,j
LIDT *Load IDT Register*	0F 01 [mod 011 r/m]	- - - - - - - - -	8	8	b,c	h,l
LLDT *Load LDT Register*		- - - - - - - - -			a	g,h,j,l
From Register/Memory	0F 00 [mod 010 r/m]		5	5		
LMSW *Load Machine Status Word*		- - - - - - - - -			b,c	h,l
From Register/Memory	0F 01 [mod 110 r/m]		13	13		
LODS *Load String*	A [110 w]	- - - - - - - - -	3	3	b	h
LOOP *Offset Loop/No Loop*	E2 +	- - - - - - - - -	1	1		r
LOOPNZ/LOOPNE *Offset*	E0 +	- - - - - - - - -	1	1		r
LOOPZ/LOOPE *Offset*	E1 +	- - - - - - - - -	1	1		r
LSL *Load Segment Limit*		- - - - - x - - -			a	g,h,j,p
From Register/Memory	0F 03 [mod reg r/m]			8		
LSS *Load Pointer to SS*	0F B2 [mod reg r/m]	- - - - - - - - -	2	4	a	h,i,j
LTR *Load Task Register*		- - - - - - - - -			a	g,h,j,l
From Register/Memory	0F 00 [mod 011 r/m]			7		
MOV *Move Data*		- - - - - - - - -			b	h,i,j
Register to Register	8 [10dw] [11 reg r/m]		1	1		
Register to Memory	8 [100w] [mod reg r/m]		1	1		
Register/Memory to Register	8 [101w] [mod reg r/m]		1	1		
Immediate to Register/Memory	C [011w] [mod 000 r/m] ###		1	1		
Immediate to Register (short form)	B [w reg] ###		1	1		
Memory to Accumulator (short form)	A [000w] +++		1	1		
Accumulator to Memory (short form)	A [001w] +++		1	1		
Register/Memory to Segment Register	8E [mod sreg3 r/m]		1	1/3		
Segment Register to Register/Memory	8C [mod sreg3 r/m]		1	1		
MOV *Move to/from Control/Debug/Test Regs*		- - - - - - - - -				l
Register to CR0/CR2/CR3/CR4	0F 22 [11 eee reg]		20/5/5	20/5/5		
CR0/CR2/CR3/CR4 to Register	0F 20 [11 eee reg]		6	6		
Register to DR0-DR3	0F 23 [11 eee reg]		16	16		
DR0-DR3 to Register	0F 21 [11 eee reg]		14	14		
Register to DR6-DR7	0F 23 [11 eee reg]		16	16		
DR6-DR7 to Register	0F 21 [11 eee reg]		14	14		
Register to TR3-5	0F 26 [11 eee reg]		10	10		
TR3-5 to Register	0F 24 [11 eee reg]		5	5		
Register to TR6-TR7	0F 26 [11 eee reg]		10	10		
TR6-TR7 to Register	0F 24 [11 eee reg]		6	6		

\# = immediate 8-bit data
= immediate 16-bit data
= full immediate 32-bit data (8, 16, 32 bits)

\+ = 8-bit signed displacement
+++ = full signed displacement (16, 32 bits)

x = modified
- = unchanged
u = undefined

Table 6-21. 6x86MX CPU Instruction Set Clock Count Summary (Continued)

INSTRUCTION	OPCODE	FLAGS	REAL MODE CLOCK COUNT	PROTECTED MODE CLOCK COUNT	NOTES	
		OF DF IF TF SF ZF AF PF CF	Reg/ Cache Hit	Reg/ Cache Hit	Real Mode	Protected Mode
MOVS *Move String*	A [010w]	- - - - - - - - -	4	4	b	h
MOVSX *Move with Sign Extension* Register from Register/Memory	0F B[111w] [mod reg r/m]	- - - - - - - - -	1	1	b	h
MOVZX *Move with Zero Extension* Register from Register/Memory	0F B[011w] [mod reg r/m]	- - - - - - - - -	1	1	b	h
MUL *Unsigned Multiply* Accumulator with Register/Memory Multiplier: Byte Word Doubleword	F [011w] [mod 100 r/m]	x - - - x x u u x	 4 4 10	 4 4 10	b	h
NEG *Negate Integer*	F [011w] [mod 011 r/m]	x - - - x x x x x	1	1	b	h
NOP *No Operation*	90	- - - - - - - - -	1	1		
NOT *Boolean Complement*	F [011w] [mod 010 r/m]	- - - - - - - - -	1	1	b	h
OIO *Official Invalid OpCode*	0F FF	- - x 0 - - - - -	1	8 - 125		
OR *Boolean OR* Register to Register Register to Memory Memory to Register Immediate to Register/Memory Immediate to Accumulator	 0 [10dw] [11 reg r/m] 0 [100w] [mod reg r/m] 0 [101w] [mod reg r/m] 8 [00sw] [mod 001 r/m] ### 0 [110w] ###	0 - - - x x u x 0	 1 1 1 1 1	 1 1 1 1 1	b	h
OUT *Output to Port* Fixed Port Variable Port	 E [011w] # E [111w]	- - - - - - - - -	 14 14	 14/28 14/28		m
OUTS *Output String*	6 [111w]	- - - - - - - - -	14	14/28	b	h,m
POP *Pop Value off Stack* Register/Memory Register (short form) Segment Register (ES, SS, DS) Segment Register (FS, GS)	 8F [mod 000 r/m] 5 [1 reg] [000 sreg2 111] 0F [10 sreg3 001]	- - - - - - - - -	 1 1 1 1	 1 1 3 3	b	h,i,j
POPA *Pop All General Registers*	61	- - - - - - - - -	6	6	b	h
POPF *Pop Stack into FLAGS*	9D	x x x x x x x x x	9	9	b	h,n

\# = immediate 8-bit data
= immediate 16-bit data
= full immediate 32-bit data (8, 16, 32 bits)

\+ = 8-bit signed displacement
+++ = full signed displacement (16, 32 bits)

x = modified
\- = unchanged
u = undefined

Table 6-21. 6x86MX CPU Instruction Set Clock Count Summary (Continued)

INSTRUCTION	OPCODE	FLAGS	REAL MODE CLOCK COUNT	PROTECTED MODE CLOCK COUNT	NOTES	
		OF DF IF TF SF ZF AF PF CF	Reg/ Cache Hit	Reg/ Cache Hit	Real Mode	Protected Mode
PREFIX BYTES		- - - - - - - - -				m
Assert Hardware LOCK Prefix	F0					
Address Size Prefix	67					
Operand Size Prefix	66					
Segment Override Prefix						
CS	2E					
DS	3E					
ES	26					
FS	64					
GS	65					
SS	36					
PUSH *Push Value onto Stack*		- - - - - - - - -			b	h
Register/Memory	FF [mod 110 r/m]		1	1		
Register (short form)	5 [0 reg]		1	1		
Segment Register (ES, CS, SS, DS)	[000 sreg2 110]		1	1		
Segment Register (FS, GS)	0F [10 sreg3 000]		1	1		
Immediate	6 [10s0] ###		1	1		
PUSHA *Push All General Registers*	60	- - - - - - - - -	6	6	b	h
PUSHF *Push FLAGS Register*	9C	- - - - - - - - -	2	2	b	h
RCL *Rotate Through Carry Left*					b	h
Register/Memory by 1	D [000w] [mod 010 r/m]	x - - - - - - - - x	3	3		
Register/Memory by CL	D [001w] [mod 010 r/m]	u - - - - - - - - x	8	8		
Register/Memory by Immediate	C [000w] [mod 010 r/m] #	u - - - - - - - - x	8	8		
RCR *Rotate Through Carry Right*					b	h
Register/Memory by 1	D [000w] [mod 011 r/m]	x - - - - - - - - x	4	4		
Register/Memory by CL	D [001w] [mod 011 r/m]	u - - - - - - - - x	9	9		
Register/Memory by Immediate	C [000w] [mod 011 r/m] #	u - - - - - - - - x	9	9		
RDMSR *Read Model Specific Register*	0F 32	- - - - - - - - -				
RDPMC *Read Performance-Monitoring Counters*	0F 33	- - - - - - - - -				
RDSHR *Read SMM Header Pointer Register*	0F 36	- - - - - - - - -				
RDTSC *Read Time Stamp Counter*	0F 31	- - - - - - - - -				
REP INS *Input String*	F3 6[110w]	- - - - - - - - -	12+5n	12+5n\ 28+5n	b	h,m
REP LODS *Load String*	F3 A[110w]	- - - - - - - - -	10+n	10+n	b	h
REP MOVS *Move String*	F3 A[010w]	- - - - - - - - -	9+n	9+n	b	h
REP OUTS *Output String*	F3 6[111w]	- - - - - - - - -	12+5n	12+5n\ 28+5n	b	h,m
REP STOS *Store String*	F3 A[101w]	- - - - - - - - -	10+n	10+n	b	h
REPE CMPS *Compare String* (*Find non-match*)	F3 A[011w]	x - - - x x x x x	10+2n	10+2n	b	h

\# = immediate 8-bit data
= immediate 16-bit data
= full immediate 32-bit data (8, 16, 32 bits)

\+ = 8-bit signed displacement
+++ = full signed displacement (16, 32 bits)

x = modified
\- = unchanged
u = undefined

Table 6-21. 6x86MX CPU Instruction Set Clock Count Summary (Continued)

INSTRUCTION	OPCODE	FLAGS	REAL MODE CLOCK COUNT	PROTECTED MODE CLOCK COUNT	NOTES	
		OF DF IF TF SF ZF AF PF CF	Reg/ Cache Hit	Reg/ Cache Hit	Real Mode	Protected Mode
REPE SCAS *Scan String (Find non-AL/AX/EAX)*	F3 A[111w]	x - - - x x x x x	10+2n	10+2n	b	h
REPNE CMPS *Compare String (Find match)*	F2 A[011w]	x - - - x x x x x	10+2n	10+2n	b	h
REPNE SCAS *Scan String (Find AL/AX/EAX)*	F2 A[111w]	x - - - x x x x x	10+2n	10+2n	b	h
RET *Return from Subroutine*		- - - - - - - - -			b	g,h,j,k,r
Within Segment	C3		3	3		
Within Segment Adding Immediate to SP	C2 ##		4	4		
Intersegment	CB		4	7		
Intersegment Adding Immediate to SP	CA ##		4	7		
Protected Mode: Different Privilege Level Intersegment				23		
Intersegment Adding Immediate to SP				23		
ROL *Rotate Left*					b	h
Register/Memory by 1	D[000w] [mod 000 r/m]	x - - - - - - - x	1	1		
Register/Memory by CL	D[001w] [mod 000 r/m]	u - - - - - - - x	2	2		
Register/Memory by Immediate	C[000w] [mod 000 r/m] #	u - - - - - - - x	1	1		
ROR *Rotate Right*					b	h
Register/Memory by 1	D[000w] [mod 001 r/m]	x - - - - - - - x	1	1		
Register/Memory by CL	D[001w] [mod 001 r/m]	u - - - - - - - x	2	2		
Register/Memory by Immediate	C[000w] [mod 001 r/m] #	u - - - - - - - x	1	1		
RSDC *Restore Segment Register and Descriptor*	0F 79 [mod sreg3 r/m]	- - - - - - - - -	6	6	s	s
RSLDT *Restore LDTR and Descriptor*	0F 7B [mod 000 r/m]	- - - - - - - - -	6	6	s	s
RSM *Resume from SMM Mode*	0F AA	x x x x x x x x x	40	40	s	s
RSTS *Restore TSR and Descriptor*	0F 7D [mod 000 r/m]	- - - - - - - - -	6	6	s	s
SAHF *Store AH in FLAGS*	9E	- - - - x x x x x	1	1		
SAL *Shift Left Arithmetic*					b	h
Register/Memory by 1	D[000w] [mod 100 r/m]	x - - - x x u x x	1	1		
Register/Memory by CL	D[001w] [mod 100 r/m]	u - - - x x u x x	2	2		
Register/Memory by Immediate	C[000w] [mod 100 r/m] #	u - - - x x u x x	1	1		
SAR *Shift Right Arithmetic*					b	h
Register/Memory by 1	D[000w] [mod 111 r/m]	x - - - x x u x x	1	1		
Register/Memory by CL	D[001w] [mod 111 r/m]	u - - - x x u x x	2	2		
Register/Memory by Immediate	C[000w] [mod 111 r/m] #	u - - - x x u x x	1	1		
SBB *Integer Subtract with Borrow*		x - - - x x x x x			b	h
Register to Register	1[10dw] [11 reg r/m]		1	1		
Register to Memory	1[100w] [mod reg r/m]		1	1		
Memory to Register	1[101w] [mod reg r/m]		1	1		
Immediate to Register/Memory	8[00sw] [mod 011 r/m] ###		1	1		
Immediate to Accumulator (short form)	1[110w] ###		1	1		

\# = immediate 8-bit data
= immediate 16-bit data
= full immediate 32-bit data (8, 16, 32 bits)

\+ = 8-bit signed displacement
+++ = full signed displacement (16, 32 bits)

x = modified
\- = unchanged
u = undefined

Table 6-21. 6x86MX CPU Instruction Set Clock Count Summary (Continued)

INSTRUCTION	OPCODE	FLAGS	REAL MODE CLOCK COUNT	PROTECTED MODE CLOCK COUNT	NOTES	
		OF DF IF TF SF ZF AF PF CF	Reg/ Cache Hit	Reg/ Cache Hit	Real Mode	Protected Mode
SCAS *Scan String*	A [111w]	x - - - x x x x x	2	2	b	h
SETB/SETNAE/SETC *Set Byte on Below/Not Above or Equal/Carry* To Register/Memory	0F 92 [mod 000 r/m]	- - - - - - - - -	1	1		h
SETBE/SETNA *Set Byte on Below or Equal/Not Above* To Register/Memory	0F 96 [mod 000 r/m]	- - - - - - - - -	1	1		h
SETE/SETZ *Set Byte on Equal/Zero* To Register/Memory	0F 94 [mod 000 r/m]	- - - - - - - - -	1	1		h
SETL/SETNGE *Set Byte on Less/Not Greater or Equal* To Register/Memory	0F 9C [mod 000 r/m]	- - - - - - - - -	1	1		h
SETLE/SETNG *Set Byte on Less or Equal/Not Greater* To Register/Memory	0F 9E [mod 000 r/m]	- - - - - - - - -	1	1		h
SETNB/SETAE/SETNC *Set Byte on Not Below/ Above or Equal/Not Carry* To Register/Memory	0F 93 [mod 000 r/m]	- - - - - - - - -	1	1		h
SETNBE/SETA *Set Byte on Not Below or Equal/Above* To Register/Memory	0F 97 [mod 000 r/m]	- - - - - - - - -	1	1		h
SETNE/SETNZ *Set Byte on Not Equal/Not Zero* To Register/Memory	0F 95 [mod 000 r/m]	- - - - - - - - -	1	1		h
SETNL/SETGE *Set Byte on Not Less/Greater or Equal* To Register/Memory	0F 9D [mod 000 r/m]	- - - - - - - - -	1	1		h
SETNLE/SETG *Set Byte on Not Less or Equal/Greater* To Register/Memory	0F 9F [mod 000 r/m]	- - - - - - - - -	1	1		h
SETNO *Set Byte on Not Overflow* To Register/Memory	0F 91 [mod 000 r/m]	- - - - - - - - -	1	1		h
SETNP/SETPO *Set Byte on Not Parity/Parity Odd* To Register/Memory	0F 9B [mod 000 r/m]	- - - - - - - - -	1	1		h
SETNS *Set Byte on Not Sign* To Register/Memory	0F 99 [mod 000 r/m]	- - - - - - - - -	1	1		h
SETO *Set Byte on Overflow* To Register/Memory	0F 90 [mod 000 r/m]	- - - - - - - - -	1	1		h

\# = immediate 8-bit data
= immediate 16-bit data
= full immediate 32-bit data (8, 16, 32 bits)

\+ = 8-bit signed displacement
+++ = full signed displacement (16, 32 bits)

x = modified
- = unchanged
u = undefined

Table 6-21. 6x86MX CPU Instruction Set Clock Count Summary (Continued)

INSTRUCTION	OPCODE	FLAGS	REAL MODE CLOCK COUNT	PROTECTED MODE CLOCK COUNT	NOTES	
		OF DF IF TF SF ZF AF PF CF	Reg/ Cache Hit	Reg/ Cache Hit	Real Mode	Protected Mode
SETP/SETPE *Set Byte on Parity/Parity Even*		- - - - - - - - -				h
To Register/Memory	0F 9A [mod 000 r/m]		1	1		
SETS *Set Byte on Sign*		- - - - - - - - -				h
To Register/Memory	0F 98 [mod 000 r/m]		1	1		
SGDT *Store GDT Register*		- - - - - - - - -	4	4	b,c	h
To Register/Memory	0F 01 [mod 000 r/m]					
SHL *Shift Left Logical*					b	h
Register/Memory by 1	D [000w] [mod 100 r/m]	x - - - x x u x x	1	1		
Register/Memory by CL	D [001w] [mod 100 r/m]	u - - - x x u x x	2	2		
Register/Memory by Immediate	C [000w] [mod 100 r/m] #	u - - - x x u x x	1	1		
SHLD *Shift Left Double*		u - - - x x u x x			b	h
Register/Memory by Immediate	0F A4 [mod reg r/m] #		4	4		
Register/Memory by CL	0F A5 [mod reg r/m]		5	5		
SHR *Shift Right Logical*					b	h
Register/Memory by 1	D [000w] [mod 101 r/m]	x - - - x x u x x	1	1		
Register/Memory by CL	D [001w] [mod 101 r/m]	u - - - x x u x x	2	2		
Register/Memory by Immediate	C [000w] [mod 101 r/m] #	u - - - x x u x x	1	1		
SHRD *Shift Right Double*		u - - - x x u x x			b	h
Register/Memory by Immediate	0F AC [mod reg r/m] #		4	4		
Register/Memory by CL	0F AD [mod reg r/m]		5	5		
SIDT *Store IDT Register*		- - - - - - - - -			b,c	h
To Register/Memory	0F 01 [mod 001 r/m]		4	4		
SLDT *Store LDT Register*		- - - - - - - - -			a	h
To Register/Memory	0F 00 [mod 000 r/m]			1		
SMINT *Software SMM Entry*	0F 38	- - - - - - - - -	55	55	s	s
SMSW *Store Machine Status Word*	0F 01 [mod 100 r/m]	- - - - - - - - -	6	6	b,c	h
STC *Set Carry Flag*	F9	- - - - - - - - 1	1	1		
STD *Set Direction Flag*	FD	- 1 - - - - - - -	7	7		
STI *Set Interrupt Flag*	FB	- - 1 - - - - - -	7	7		m
STOS *Store String*	A [101w]	- - - - - - - - -	2	2	b	h
STR *Store Task Register*		- - - - - - - - -			a	h
To Register/Memory	0F 00 [mod 001 r/m]			4		

\# = immediate 8-bit data
= immediate 16-bit data
= full immediate 32-bit data (8, 16, 32 bits)

\+ = 8-bit signed displacement
+++ = full signed displacement (16, 32 bits)

x = modified
- = unchanged
u = undefined

Table 6-21. 6x86MX CPU Instruction Set Clock Count Summary (Continued)

INSTRUCTION	OPCODE	FLAGS	REAL MODE CLOCK COUNT	PROTECTED MODE CLOCK COUNT	NOTES	
		OF DF IF TF SF ZF AF PF CF	Reg/ Cache Hit	Reg/ Cache Hit	Real Mode	Protected Mode
SUB *Integer Subtract*		x - - - x x x x x			b	h
Register to Register	2 [10dw] [11 reg r/m]		1	1		
Register to Memory	2 [100w] [mod reg r/m]		1	1		
Memory to Register	2 [101w] [mod reg r/m]		1	1		
Immediate to Register/Memory	8 [00sw] [mod 101 r/m] ###		1	1		
Immediate to Accumulator (short form)	2 [110w] ###		1	1		
SVDC *Save Segment Register and Descriptor*	0F 78 [mod sreg3 r/m]	- - - - - - - - - -	12	12	s	s
SVLDT *Save LDTR and Descriptor*	0F 7A [mod 000 r/m]	- - - - - - - - - -	12	12	s	s
SVTS *Save TSR and Descriptor*	0F 7C [mod 000 r/m]	- - - - - - - - - -	14	14	s	s
TEST *Test Bits*		0 - - - x x u x 0			b	h
Register/Memory and Register	8 [010w] [mod reg r/m]		1	1		
Immediate Data and Register/Memory	F [011w] [mod 000 r/m] ###		1	1		
Immediate Data and Accumulator	A [100w] ###		1	1		
VERR *Verify Read Access*		- - - - - x - - -			a	g,h,j,p
To Register/Memory	0F 00 [mod 100 r/m]			7		
VERW *Verify Write Access*		- - - - - x - - -			a	g,h,j,p
To Register/Memory	0F 00 [mod 101 r/m]			7		
WAIT *Wait Until FPU Not Busy*	9B	- - - - - - - - - -	5	5		
WBINVD *Write-Back and Invalidate Cache*	0F 09	- - - - - - - - - -	15	15	t	t
WRMSR *Write to Model Specific Register*	0F 30	- - - - - - - - - -				
WRSHR *Write SMM Header Pointer Register*	0F 37	- - - - - - - - - -				
XADD *Exchange and Add*		x - - - x x x x x				
Register1, Register2	0F C[000w] [11 reg2 reg1]		2	2		
Memory, Register	0F C[000w] [mod reg r/m]		2	2		
XCHG *Exchange*		- - - - - - - - - -			b,f	f,h
Register/Memory with Register	8[011w] [mod reg r/m]		2	2		
Register with Accumulator	9[0 reg]		2	2		
XLAT *Translate Byte*	D7	- - - - - - - - - -	4	4		h
XOR *Boolean Exclusive OR*		0 - - - x x u x 0			b	h
Register to Register	3 [00dw] [11 reg r/m]		1	1		
Register to Memory	3 [000w] [mod reg r/m]		1	1		
Memory to Register	3 [001w] [mod reg r/m]		1	1		
Immediate to Register/Memory	8 [00sw] [mod 110 r/m] ###		1	1		
Immediate to Accumulator (short form)	3 [010w] ###		1	1		

\# = immediate 8-bit data
= immediate 16-bit data
= full immediate 32-bit data (8, 16, 32 bits)

\+ = 8-bit signed displacement
+++ = full signed displacement (16, 32 bits)

x = modified
\- = unchanged
u = undefined

Instruction Notes for Instruction Set Summary

Notes a through c apply to Real Address Mode only:

a. This is a Protected Mode instruction. Attempted execution in Real Mode will result in exception 6 (invalid op-code).
b. Exception 13 fault (general protection) will occur in Real Mode if an operand reference is made that partially or fully extends beyond the maximum CS, DS, ES, FS, or GS segment limit (FFFFH). Exception 12 fault (stack segment limit violation or not present) will occur in Real Mode if an operand reference is made that partially or fully extends beyond the maximum SS limit.
c. This instruction may be executed in Real Mode. In Real Mode, its purpose is primarily to initialize the CPU for Protected Mode.
d. -

Notes e through g apply to Real Address Mode and Protected Virtual Address Mode:

e. An exception may occur, depending on the value of the operand.
f. LOCK# is automatically asserted, regardless of the presence or absence of the LOCK prefix.
g. LOCK# is asserted during descriptor table accesses.

Notes h through r apply to Protected Virtual Address Mode only:

h. Exception 13 fault will occur if the memory operand in CS, DS, ES, FS, or GS cannot be used due to either a segment limit violation or an access rights violation. If a stack limit is violated, an exception 12 occurs.
i. For segment load operations, the CPL, RPL, and DPL must agree with the privilege rules to avoid an exception 13 fault. The segment's descriptor must indicate "present" or exception 11 (CS, DS, ES, FS, GS not present). If the SS register is loaded and a stack segment not present is detected, an exception 12 occurs.
j. All segment descriptor accesses in the GDT or LDT made by this instruction will automatically assert LOCK# to maintain descriptor integrity in multiprocessor systems.
k. JMP, CALL, INT, RET, and IRET instructions referring to another code segment will cause an exception 13, if an applicable privilege rule is violated.
l. An exception 13 fault occurs if CPL is greater than 0 (0 is the most privileged level).
m. An exception 13 fault occurs if CPL is greater than IOPL.
n. The IF bit of the flag register is not updated if CPL is greater than IOPL. The IOPL and VM fields of the flag register are updated only if CPL = 0.
o. The PE bit of the MSW (CR0) cannot be reset by this instruction. Use MOV into CRO if desiring to reset the PE bit.
p. Any violation of privilege rules as apply to the selector operand does not cause a Protection exception, rather, the zero flag is cleared.
q. If the coprocessor's memory operand violates a segment limit or segment access rights, an exception 13 fault will occur before the ESC instruction is executed. An exception 12 fault will occur if the stack limit is violated by the operand's starting address.
r. The destination of a JMP, CALL, INT, RET, or IRET must be in the defined limit of a code segment or an exception 13 fault will occur.

Note s applies to Cyrix specific SMM instructions:

s. All memory accesses to SMM space are non-cacheable. An invalid opcode exception 6 occurs unless SMI is enabled and ARR3 size > 0, and CPL = 0 and [SMAC is set or if in an SMI handler].

Note t applies to cache invalidation instructions with the cache operating in write-back mode:

t. The total clock count is the clock count shown plus the number of clocks required to write all "modified" cache lines to external memory.

6.5 FPU Instruction Clock Counts

The CPU is functionally divided into the FPU unit, and the integer unit. The FPU has been extended to processes MMX instructions as well as floating point instructions in parallel with the integer unit.

For example, when the integer unit detects a floating point instruction the instruction passes to the FPU for execution. The integer unit continues to execute instructions while the FPU executes the floating point instruction.

If another FPU instruction is encountered, the second FPU instruction is placed in the FPU queue. Up to four FPU instructions can be queued. In the event of an FPU exception, while other FPU instructions are queued, the state of the CPU is saved to ensure recovery.

6.5.1 FPU Clock Count Table

The clock counts for the FPU instructions are listed in Table 6-23 (Page 6-31). The abbreviations used in this table are listed in Table 6-22.

Table 6-22. FPU Clock Count Table Abbreviations

ABBREVIATION	MEANING
n	Stack register number
TOS	Top of stack register pointed to by SSS in the status register.
ST(1)	FPU register next to TOS
ST(n)	A specific FPU register, relative to TOS
M.WI	16-bit integer operand from memory
M.SI	32-bit integer operand from memory
M.LI	64-bit integer operand from memory
M.SR	32-bit real operand from memory
M.DR	64-bit real operand from memory
M.XR	80-bit real operand from memory
M.BCD	18-digit BCD integer operand from memory
CC	FPU condition code
Env Regs	Status, Mode Control and Tag Registers, Instruction Pointer and Operand Pointer

Table 6-23. 6x86MX FPU Instruction Set Summary

FPU INSTRUCTION	OP CODE	OPERATION	CLOCK COUNT	NOTES
F2XM1 *Function Evaluation 2^x-1*	D9 F0	TOS <— 2^{TOS}-1	92 - 108	See Note 2
FABS *Floating Absolute Value*	D9 E1	TOS <— | TOS |	2	
FADD *Floating Point Add*				
Top of Stack	DC [1100 0 n]	ST(n) <— ST(n) + TOS	4 - 7	
80-bit Register	D8 [1100 0 n]	TOS <— TOS + ST(n)	4 - 7	
64-bit Real	DC [mod 000 r/m]	TOS <— TOS + M.DR	4 - 7	
32-bit Real	D8 [mod 000 r/m]	TOS <— TOS + M.SR	4 - 7	
FADDP *Floating Point Add, Pop*	DE [1100 0 n]	ST(n) <— ST(n) + TOS; then pop TOS	4 - 7	
FIADD *Floating Point Integer Add*				
32-bit integer	DA [mod 000 r/m]	TOS <— TOS + M.SI	8 - 12	
16-bit integer	DE [mod 000 r/m]	TOS <— TOS + M.WI	8 - 12	
FCHS *Floating Change Sign*	D9 E0	TOS <— - TOS	2	
FCLEX *Clear Exceptions*	(9B)DB E2	Wait then Clear Exceptions	5	
FNCLEX *Clear Exceptions*	DB E2	Clear Exceptions	3	
FCOM *Floating Point Compare*				
80-bit Register	D8 [1101 0 n]	CC set by TOS - ST(n)	4	
64-bit Real	DC [mod 010 r/m]	CC set by TOS - M.DR	4	
32-bit Real	D8 [mod 010 r/m]	CC set by TOS - M.SR	4	
FCOMP *Floating Point Compare, Pop*				
80-bit Register	D8 [1101 1 n]	CC set by TOS - ST(n); then pop TOS	4	
64-bit Real	DC [mod 011 r/m]	CC set by TOS - M.DR; then pop TOS	4	
32-bit Real	D8 [mod 011 r/m]	CC set by TOS - M.SR; then pop TOS	4	
FCOMPP *Floating Point Compare, Pop Two Stack Elements*	DE D9	CC set by TOS - ST(1); then pop TOS and ST(1)	4	
FICOM *Floating Point Compare*				
32-bit integer	DA [mod 010 r/m]	CC set by TOS - M.WI	9 - 10	
16-bit integer	DE [mod 010 r/m]	CC set by TOS - M.SI	9 - 10	
FICOMP *Floating Point Compare*				
32-bit integer	DA [mod 011 r/m]	CC set by TOS - M.WI; then pop TOS	9 - 10	
16-bit integer	DE [mod 011 r/m]	CC set by TOS - M.SI; then pop TOS	9 - 10	
FCOMI *Floating Point Compare Real and Set EFLAGS*				
80-bit Register	DB [1111 0 n]	EFLAG set by TOS - ST(n)	4	
FCOMIP *Floating Point Compare Real and Set EFLAGS, Pop*				
80-bit Register	DF [1111 0 n]	EFLAG set by TOS - ST(n); then pop TOS	4	
FUCOMI *Floating Point Unordered Compare Real and Set EFLAGS*				
80-bit integer	DB [1110 1 n]	EFLAG set by TOS - ST(n)	9 - 10	
FUCOMIP *Floating Point Unordered Compare Real and Set EFLAGS*				
80-bit integer	DF [1110 1 n]	EFLAG set by TOS - ST(n); then pop TOS	9 - 10	
FCMOVB *Floating Point Conditional Move if Below*	DA [1100 0 n]	If (CF=1) ST(0) <—ST(n)	4	
FCMOVE *Floating Point Conditional Move if* Equal	DA [1100 1 n]	If (ZF=1) ST(0) <—ST(n)	4	

Table 6-23. 6x86MX FPU Instruction Set Summary (Continued)

FPU INSTRUCTION	OP CODE	OPERATION	CLOCK COUNT	NOTES
FCMOVBE *Floating Point Conditional Move if* Below or Equal	DA [1101 0 n]	If (CF=1 or ZF=1) ST(0) <—ST(n)	4	
FCMOVU *Floating Point Conditional Move if Unordered*	DA [1101 1 n]	If (PF=1) ST(0) <—ST(n)	4	
FCMOVNB *Floating Point Conditional Move if Not Below*	DB [1100 0 n]	If (CF=0) ST(0) <—ST(n)	4	
FCMOVNE *Floating Point Conditional Move if Not Equal*	DB [1100 1 n]	If (ZF=0) ST(0) <—ST(n)	4	
FCMOVNBE *Floating Point Conditional Move if Not Below or Equal*	DB [1101 0 n]	If (CF=0 and ZF=0) ST(0) <—ST(n)	4	
FCMOVNU *Floating Point Conditional Move if Not Unordered*	DB [1101 1 n]	If (DF=0) ST(0)<—ST(n)	4	
FCOS *Function Evaluation: Cos(x)*	D9 FF	TOS <— COS(TOS)	92 - 141	See Note 1
FDECSTP *Decrement Stack Pointer*	D9 F6	Decrement top of stack pointer	4	
FDIV *Floating Point Divide*				
Top of Stack	DC [1111 1 n]	ST(n) <— ST(n) / TOS	24 - 34	
80-bit Register	D8 [1111 0 n]	TOS <— TOS / ST(n)	24 - 34	
64-bit Real	DC [mod 110 r/m]	TOS <— TOS / M.DR	24 - 34	
32-bit Real	D8 [mod 110 r/m]	TOS <— TOS / M.SR	24 - 34	
FDIVP *Floating Point Divide, Pop*	DE [1111 1 n]	ST(n) <— ST(n) / TOS; then pop TOS	24 - 34	
FDIVR *Floating Point Divide Reversed*				
Top of Stack	DC [1111 0 n]	TOS <— ST(n) / TOS	24 - 34	
80-bit Register	D8 [1111 1 n]	ST(n) <— TOS / ST(n)	24 - 34	
64-bit Real	DC [mod 111 r/m]	TOS <— M.DR / TOS	24 - 34	
32-bit Real	D8 [mod 111 r/m]	TOS <— M.SR / TOS	24 - 34	
FDIVRP *Floating Point Divide Reversed*, Pop	DE [1111 0 n]	ST(n) <— TOS / ST(n); then pop TOS	24 - 34	
FIDIV *Floating Point Integer Divide*				
32-bit Integer	DA [mod 110 r/m]	TOS <— TOS / M.SI	34 - 38	
16-bit Integer	DE [mod 110 r/m]	TOS <— TOS / M.WI	33 - 38	
FIDIVR *Floating Point Integer Divide Reversed*				
32-bit Integer	DA [mod 111 r/m]	TOS <— M.SI / TOS	34 - 38	
16-bit Integer	DE [mod 111 r/m]	TOS <— M.WI / TOS	33 - 38	
FFREE *Free Floating Point Register*	DD [1100 0 n]	TAG(n) <— Empty	3	
FINCSTP *Increment Stack Pointer*	D9 F7	Increment top of stack pointer	2	
FINIT *Initialize FPU*	(9B)DB E3	Wait then initialize	8	
FNINIT *Initialize FPU*	DB E3	Initialize	6	

Table 6-23. 6x86MX FPU Instruction Set Summary (Continued)

FPU INSTRUCTION	OP CODE	OPERATION	CLOCK COUNT	NOTES
FLD *Load Data to FPU Reg.*				
Top of Stack	D9 [1100 0 n]	Push ST(n) onto stack	2	
64-bit Real	DD [mod 000 r/m]	Push M.DR onto stack	2	
32-bit Real	D9 [mod 000 r/m]	Push M.SR onto stack	2	
FBLD *Load Packed BCD Data to FPU Reg.*	DF [mod 100 r/m]	Push M.BCD onto stack	41 - 45	
FILD *Load Integer Data to FPU Reg.*				
64-bit Integer	DF [mod 101 r/m]	Push M.LI onto stack	4 - 8	
32-bit Integer	DB [mod 000 r/m]	Push M.SI onto stack	4 - 6	
16-bit Integer	DF [mod 000 r/m]	Push M.WI onto stack	3 - 6	
FLD1 *Load Floating Const.= 1.0*	D9 E8	Push 1.0 onto stack	4	
FLDCW *Load FPU Mode Control Register*	D9 [mod 101 r/m]	Ctl Word ⟵ Memory	4	
FLDENV *Load FPU Environment*	D9 [mod 100 r/m]	Env Regs ⟵ Memory	30	
FLDL2E *Load Floating Const.= $Log_2(e)$*	D9 EA	Push $Log_2(e)$ onto stack	4	
FLDL2T *Load Floating Const.= $Log_2(10)$*	D9 E9	Push $Log_2(10)$ onto stack	4	
FLDLG2 *Load Floating Const.= $Log_{10}(2)$*	D9 EC	Push $Log_{10}(2)$ onto stack	4	
FLDLN2 *Load Floating Const.= $Ln(2)$*	D9 ED	Push $Log_e(2)$ onto stack	4	
FLDPI *Load Floating Const.= π*	D9 EB	Push π onto stack	4	
FLDZ *Load Floating Const.= 0.0*	D9 EE	Push 0.0 onto stack	4	
FMUL *Floating Point Multiply*				
Top of Stack	DC [1100 1 n]	ST(n) ⟵ ST(n) × TOS	4 - 6	
80-bit Register	D8 [1100 1 n]	TOS ⟵ TOS × ST(n)	4 - 6	
64-bit Real	DC [mod 001 r/m]	TOS ⟵ TOS × M.DR	4 - 6	
32-bit Real	D8 [mod 001 r/m]	TOS ⟵ TOS × M.SR	4 - 5	
FMULP *Floating Point Multiply* & Pop	DE [1100 1 n]	ST(n) ⟵ ST(n) × TOS; then pop TOS	4 - 6	
FIMUL *Floating Point Integer Multiply*				
32-bit Integer	DA [mod 001 r/m]	TOS ⟵ TOS × M.SI	9 - 11	
16-bit Integer	DE [mod 001 r/m]	TOS ⟵ TOS × M.WI	8 - 10	
FNOP *No Operation*	D9 D0	No Operation	2	
FPATAN *Function Eval: $Tan^{-1}(y/x)$*	D9 F3	ST(1) ⟵ ATAN[ST(1) / TOS]; then pop TOS	97 - 161	See Note 3
FPREM *Floating Point Remainder*	D9 F8	TOS ⟵ Rem[TOS / ST(1)]	82 - 91	
FPREM1 *Floating Point Remainder IEEE*	D9 F5	TOS ⟵ Rem[TOS / ST(1)]	82 - 91	
FPTAN *Function Eval: Tan(x)*	D9 F2	TOS ⟵ TAN(TOS); then push 1.0 onto stack	117 - 129	See Note 1
FRNDINT *Round to Integer*	D9 FC	TOS ⟵ Round(TOS)	10 - 20	
FRSTOR *Load FPU Environment and Reg.*	DD [mod 100 r/m]	Restore state.	56 - 72	
FSAVE *Save FPU Environment and Reg*	(9B)DD[mod 110 r/m]	Wait then save state.	57 - 67	
FNSAVE *Save FPU Environment and Reg*	DD [mod 110 r/m]	Save state.	55 - 65	
FSCALE *Floating Multiply by 2^n*	D9 FD	TOS ⟵ TOS × $2^{(ST(1))}$	7 - 14	
FSIN *Function Evaluation: Sin(x)*	D9 FE	TOS ⟵ SIN(TOS)	76 - 140	See Note 1
FSINCOS *Function Eval.: Sin(x)& Cos(x)*	D9 FB	temp ⟵ TOS; TOS ⟵ SIN(temp); then push COS(temp) onto stack	145 - 161	See Note 1
FSQRT *Floating Point Square Root*	D9 FA	TOS ⟵ Square Root of TOS	59 - 60	

Table 6-23. 6x86MX FPU Instruction Set Summary (Continued)

FPU INSTRUCTION	OP CODE	OPERATION	CLOCK COUNT	NOTES
FST *Store FPU Register*				
Top of Stack	DD [1101 0 n]	ST(n) ⟵ TOS	2	
80-bit Real	DB [mod 111 r/m]	M.XR ⟵ TOS	2	
64-bit Real	DD [mod 010 r/m]	M.DR ⟵ TOS	2	
32-bit Real	D9 [mod 010 r/m]	M.SR ⟵ TOS	2	
FSTP *Store FPU Register, Pop*				
Top of Stack	DB [1101 1 n]	ST(n) ⟵ TOS; then pop TOS	2	
80-bit Real	DB [mod 111 r/m]	M.XR ⟵ TOS; then pop TOS	2	
64-bit Real	DD [mod 011 r/m]	M.DR ⟵ TOS; then pop TOS	2	
32-bit Real	D9 [mod 011 r/m]	M.SR ⟵ TOS; then pop TOS	2	
FBSTP *Store BCD Data, Pop*	DF [mod 110 r/m]	M.BCD ⟵ TOS; then pop TOS	57 - 63	
FIST *Store Integer FPU Register*				
32-bit Integer	DB [mod 010 r/m]	M.SI ⟵ TOS	8 - 13	
16-bit Integer	DF [mod 010 r/m]	M.WI ⟵ TOS	7 - 10	
FISTP *Store Integer FPU Register, Pop*				
64-bit Integer	DF [mod 111 r/m]	M.LI ⟵ TOS; then pop TOS	10 - 13	
32-bit Integer	DB [mod 011 r/m]	M.SI ⟵ TOS; then pop TOS	8 - 13	
16-bit Integer	DF [mod 011 r/m]	M.WI ⟵ TOS; then pop TOS	7 - 10	
FSTCW *Store FPU Mode Control Register*	(9B)D9[mod 111 r/m]	Wait Memory ⟵ Control Mode Register	5	
FNSTCW *Store FPU Mode Control Register*	D9 [mod 111 r/m]	Memory ⟵ Control Mode Register	3	
FSTENV *Store FPU Environment*	(9B)D9[mod 110 r/m]	Wait Memory ⟵ Env. Registers	14 - 24	
FNSTENV *Store FPU Environment*	D9 [mod 110 r/m]	Memory ⟵ Env. Registers	12 - 22	
FSTSW *Store FPU Status Register*	(9B)DD[mod 111 r/m]	Wait Memory ⟵ Status Register	6	
FNSTSW *Store FPU Status Register*	DD [mod 111 r/m]	Memory ⟵ Status Register	4	
FSTSW AX *Store FPU Status Register to AX*	(9B)DF E0	Wait AX ⟵ Status Register	4	
FNSTSW AX *Store FPU Status Register to AX*	DF E0	AX ⟵ Status Register	2	
FSUB *Floating Point Subtract*				
Top of Stack	DC [1110 1 n]	ST(n) ⟵ ST(n) - TOS	4 - 7	
80-bit Register	D8 [1110 0 n]	TOS ⟵ TOS - ST(n)	4 - 7	
64-bit Real	DC [mod 100 r/m]	TOS ⟵ TOS - M.DR	4 - 7	
32-bit Real	D8 [mod 100 r/m]	TOS ⟵ TOS - M.SR	4 - 7	
FSUBP *Floating Point Subtract, Pop*	DE [1110 1 n]	ST(n) ⟵ ST(n) - TOS; then pop TOS	4 - 7	

Table 6-23. 6x86MX FPU Instruction Set Summary (Continued)

FPU INSTRUCTION	OP CODE	OPERATION	CLOCK COUNT	NOTES
FSUBR *Floating Point Subtract Reverse*				
Top of Stack	DC [1110 0 n]	TOS ⟵ ST(n) - TOS	4 - 7	
80-bit Register	D8 [1110 1 n]	ST(n) ⟵ TOS - ST(n)	4 - 7	
64-bit Real	DC [mod 101 r/m]	TOS ⟵ M.DR - TOS	4 - 7	
32-bit Real	D8 [mod 101 r/m]	TOS ⟵ M.SR - TOS	4 - 7	
FSUBRP *Floating Point Subtract Reverse, Pop*	DE [1110 0 n]	ST(n) ⟵ TOS - ST(n); then pop TOS	4 - 7	
FISUB *Floating Point Integer Subtract*				
32-bit Integer	DA [mod 100 r/m]	TOS ⟵ TOS - M.SI	14 - 29	
16-bit Integer	DE [mod 100 r/m]	TOS ⟵ TOS - M.WI	14 - 27	
FISUBR *Floating Point Integer Subtract Reverse*				
32-bit Integer Reversed	DA [mod 101 r/m]	TOS ⟵ M.SI - TOS	14 - 29	
16-bit Integer Reversed	DE [mod 101 r/m]	TOS ⟵ M.WI - TOS	14 - 27	
FTST *Test Top of Stack*	D9 E4	CC set by TOS - 0.0	4	
FUCOM *Unordered Compare*	DD [1110 0 n]	CC set by TOS - ST(n)	4	
FUCOMP *Unordered Compare, Pop*	DD [1110 1 n]	CC set by TOS - ST(n); then pop TOS	4	
FUCOMPP *Unordered Compare, Pop two elements*	DA E9	CC set by TOS - ST(I); then pop TOS and ST(1)	4	
FWAIT *Wait*	9B	Wait for FPU not busy	2	
FXAM *Report Class of Operand*	D9 E5	CC ⟵ Class of TOS	4	
FXCH *Exchange Register with TOS*	D9 [1100 1 n]	TOS ⟷ ST(n) Exchange	2	
FXTRACT *Extract Exponent*	D9 F4	temp ⟵ TOS; TOS ⟵ exponent (temp); then push significant (temp) onto stack	11 - 16	
FLY2X *Function Eval. $y \times Log2(x)$*	D9 F1	ST(1) ⟵ ST(1) × Log_2(TOS); then pop TOS	145 - 154	
FLY2XP1 *Function Eval. $y \times Log2(x+1)$*	D9 F9	ST(1) ⟵ ST(1) × Log_2(1+TOS); then pop TOS	131 - 133	See Note 4

FPU Instruction Summary Notes

All references to TOS and ST(n) refer to stack layout prior to execution.

Values popped off the stack are discarded.

A pop from the stack increments the top of stack pointer.

A push to the stack decrements the top of stack pointer.

Note 1:
For FCOS, FSIN, FSINCOS and FPTAN, time shown is for absolute value of TOS < 3π/4.
Add 90 clock counts for argument reduction if outside this range.

For FCOS, clock count is 141 if TOS < π/4 and clock count is 92 if π/4 < TOS > π/2.

For FSIN, clock count is 81 to 82 if absolute value of TOS < π/4.

Note 2:
For F2XM1, clock count is 92 if absolute value of TOS < 0.5.

Note 3:
For FPATAN, clock count is 97 if ST(1)/TOS < π/32.

Note 4:
For FYL2XP1, clock count is 170 if TOS is out of range and regular FYL2X is called.

Note 5:
The following opcodes are reserved by Cyrix:
D9D7, D9E2, D9E7, DDFC, DED8, DEDA, DEDC, DEDD, DEDE, DFFC.
If a reserved opcode is executed, and unpredictable results may occur (exceptions are not generated).

6.6 6x86MX Processor MMX Instruction Clock Counts

The CPU is functionally divided into the FPU unit, and the integer unit. The FPU has been extended to processes both MMX instructions and floating point instructions in parallel with the integer unit.

For example, when the integer unit detects a MMX instruction, the instruction passes to the FPU unit for execution. The integer unit continues to execute instructions while the FPU unit executes the MMX instruction. If another MMX instruction is encountered, the second MMX instruction is placed in the MMX queue. Up to four MMX instructions can be queued.

6.6.1 MMX Clock Count Table

The clock counts for the MMX instructions are listed in Table 6-25 (Page 38). The abbreviations used in this table are listed in Table 6-24.

Table 6-24. MMX Clock Count Table Abbreviations

ABBREVIATION	MEANING
<----	Result written
[11 mm reg]	Binary or binary groups of digits
mm	One of eight 64-bit MMX registers
reg	A general purpose register
<--sat--	If required, the resultant data is saturated to remain in the associated data range
<--move--	Source data is moved to result location
[byte]	Eight 8-bit bytes are processed in parallel
[word]	Four 16-bit word are processed in parallel
[dword]	Two 32-bit double words are processed in parallel
[qword]	One 64-bit quad word is processed
[sign xxx]	The byte, word, double word or quad word most significant bit is a sign bit
mm1, mm2	MMX register 1, MMX register 2
mod r/m	Mod and r/m byte encoding (page 6-6 of this manual)
pack	Source data is truncated or saturated to next smaller data size, then concatenated.
packdw	Pack two double words from source and two double words from destination into four words in destination register.
packwb	Pack four words from source and four words from destination into eight bytes in destination register.

Table 6-25. 6x86MX Processor MMX Instruction Set Clock Count Summary

MMX INSTRUCTIONS	OPCODE	OPERATION	CLOCK COUNT LATENCY/ THROUGHPUT
EMMS *Empty MMX State*	0F77	Tag Word <--- FFFFh (empties the floating point tag word)	1/1
MOVD *Move Doubleword*			
Register to MMX Register	0F6E [11 mm reg]	MMX reg [qword] <--move, zero extend-- reg [dword]	1/1
MMX Register to Register	0F7E [11 mm reg]	reg [qword] <--move-- MMX reg [low dword]	5/1
Memory to MMX Register	0F6E [mod mm r/m]	MMX regr[qword] <--move, zero extend-- memory[dword]	1/1
MMX Register to Memory	0F7E [mod mm r/m]	Memory [dword] <--move-- MMX reg [low dword]	1/1
MOVQ *Move Quardword*			
MMX Register 2 to MMX Register 1	0F6F [11 mm1 mm2]	MMX reg 1 [qword] <--move-- MMX reg 2 [qword]	1/1
MMX Register 1 to MMX Register 2	0F7F [11 mm1 mm2]	MMX reg 2 [qword] <--move-- MMX reg 1 [qword]	1/1
Memory to MMX Register	0F6F [mod mm r/m]	MMX reg [qword] <--move-- memory[qword]	1/1
MMX Register to Memory	0F7F [mod mm r/m]	Memory [qword] <--move-- MMX reg [qword]	1/1
PACKSSDW *Pack Dword with Signed Saturation*			
MMX Register 2 to MMX Register 1	0F6B [11 mm1 mm2]	MMX reg 1 [qword] <--packdw, signed sat-- MMX reg 2, MMX reg 1	1/1
Memory to MMX Register	0F6B [mod mm r/m]	MMX reg [qword] <--packdw, signed sat-- memory, MMX reg	1/1
PACKSSWB *Pack Word with Signed Saturation*			
MMX Register 2 to MMX Register 1	0F63 [11 mm1 mm2]	MMX reg 1 [qword] <--packwb, signed sat-- MMX reg 2, MMX reg 1	1/1
Memory to MMX Register	0F63 [mod mm r/m]	MMX reg [qword] <--packwb, signed sat-- memory, MMX reg	1/1
PACKUSWB *Pack Word with Unsigned Saturation*			
MMX Register 2 to MMX Register 1	0F67 [11 mm1 mm2]	MMX reg 1 [qword] <--packwb, unsigned sat-- MMX reg 2, MMX reg 1	1/1
Memory to MMX Register	0F67 [mod mm r/m]	MMX reg [qword] <--packwb, unsigned sat-- memory, MMX reg	1/1
PADDB *Packed Add Byte with Wrap-Around*			
MMX Register 2 to MMX Register 1	0FFC [11 mm1 mm2]	MMX reg 1 [byte] <---- MMX reg 1 [byte] + MMX reg 2 [byte]	1/1
Memory to MMX Register	0FFC [mod mm r/m]	MMX reg[byte] <---- memory [byte] + MMX reg [byte]	1/1
PADDD *Packed Add Dword with Wrap-Around*			
MMX Register 2 to MMX Register 1	0FFE [11 mm1 mm2]	MMX reg 1 [sign dword] <---- MMX reg 1 [sign dword] + MMX reg 2 [sign dword]	1/1
Memory to MMX Register	0FFE [mod mm r/m]	MMX reg [sign dword] <---- memory [sign dword] + MMX reg [sign dword]	1/1
PADDSB *Packed Add Signed Byte with Saturation*			
MMX Register 2 to MMX Register1	0FEC [11 mm1 mm2]	MMX reg 1 [sign byte] <--sat-- MMX reg 1 [sign byte] + MMX reg 2 [sign byte]	1/1
Memory to Register	0FEC [mod mm r/m]	MMX reg [sign byte] <--sat-- memory [sign byte] + MMX reg [sign byte]	1/1
PADDSW *Packed Add Signed Word with Saturation*			
MMX Register 2 to MMX Register1	0FED [11 mm1 mm2]	MMX reg 1 [sign word] <--sat-- MMX reg 1 [sign word] + MMX reg 2 [sign word]	1/1
Memory to Register	0FED [mod mm r/m]	MMX reg [sign word] <--sat-- memory [sign word] + MMX reg [sign word]	1/1
PADDUSB *Add Unsigned Byte with Saturation*			
MMX Register 2 to MMX Register1	0FDC [11 mm1 mm2]	MMX reg 1 [byte] <--sat-- MMX reg 1 [byte] + MMX reg 2 [byte]	1/1
Memory to Register	0FDC [mod mm r/m]	MMX reg [byte] <--sat-- memory [byte] + MMX reg [byte]	1/1
PADDUSW *Add Unsigned Word with Saturation*			
MMX Register 2 to MMX Register1	0FDD [11 mm1 mm2]	MMX reg 1 [word] <--sat-- MMX reg 1 [word] + MMX reg 2 [word]	1/1
Memory to Register	0FDD [mod mm r/m]	MMX reg [word] <--sat-- memory [word] + MMX reg [word]	1/1

Table 6-25. 6x86MX Processor MMX Instruction Set Clock Count Summary (Continued)

MMX INSTRUCTIONS	OPCODE	OPERATION	CLOCK COUNT LATENCY/ THROUGHPUT
PADDW *Packed Add Word with Wrap-Around*			
MMX Register 2 to MMX Register1	0FFD [11 mm1 mm2]	MMX reg 1 [word] <---- MMX reg 1 [word] + MMX reg 2 [word]	1/1
Memory to MMX Register	0FFD [mod mm r/m]	MMX reg [word] <---- memory [word] + MMX reg [word]	1/1
PAND *Bitwise Logical AND*			
MMX Register 2 to MMX Register1	0FDB [11 mm1 mm2]	MMX Reg 1 [qword] <--logic AND-- MMX Reg 1 [qword], MMX Reg 2 [qword]	1/1
Memory to MMX Register	0FDB [mod mm r/m]	MMX Reg [qword] <--logic AND-- memory[qword], MMX Reg [qword]	1/1
PANDN *Bitwise Logical AND NOT*			
MMX Register 2 to MMX Register1	0FDF [11 mm1 mm2]	MMX Reg 1 [qword] <--logic AND -- NOT MMX Reg 1 [qword], MMX Reg 2 [qword]	1/1
Memory to MMX Register	0FDF [mod mm r/m]	MMX Reg [qword] <--logic AND-- NOT MMX Reg [qword], Memory[qword]	1/1
PCMPEQB *Packed Byte Compare for Equality*			
MMX Register 2 with MMX Register1	0F74 [11 mm1 mm2]	MMX reg 1 [byte] <--FFh-- if MMX reg 1 [byte] = MMX reg 2 [byte] MMX reg 1 [byte]<--00h-- if MMX reg 1 [byte] NOT = MMX reg 2 [byte]	1/1
Memory with MMX Register	0F74 [mod mm r/m]	MMX reg [byte] <--FFh-- if memory[byte] = MMX reg [byte] MMX reg [byte] <--00h-- if memory[byte] NOT = MMX reg [byte]	1/1
PCMPEQD *Packed Dword Compare for Equality*			
MMX Register 2 with MMX Register1	0F76 [11 mm1 mm2]	MMX reg 1 [dword] <--FFFF FFFFh-- if MMX reg 1 [dword] = MMX reg 2 [dword] MMX reg 1 [dword]<--0000 0000h--if MMX reg 1[dword] NOT = MMX reg 2 [dword]	1/1
Memory with MMX Register	0F76 [mod mm r/m]	MMX reg [dword] <--FFFF FFFFh-- if memory[dword] = MMX reg [dword] MMX reg [dword] <--0000 0000h-- if memory[dword] NOT = MMX reg [dword]	1/1
PCMPEQW *Packed Word Compare for Equality*			
MMX Register 2 with MMX Register1	0F75 [11 mm1 mm2]	MMX reg 1 [word] <--FFFFh-- if MMX reg 1 [word] = MMX reg 2 [word] MMX reg 1 [word]<--0000h-- if MMX reg 1 [word] NOT = MMX reg 2 [word]	1/1
Memory with MMX Register	0F75 [mod mm r/m]	MMX reg [word] <--FFFFh-- if memory[word] = MMX reg [word] MMX reg [word] <--0000h-- if memory[word] NOT = MMX reg [word]	1/1
PCMPGTB *Pack Compare Greater Than Byte*			
MMX Register 2 to MMX Register1	0F64 [11 mm1 mm2]	MMX reg 1 [byte] <--FFh-- if MMX reg 1 [byte] > MMX reg 2 [byte] MMX reg 1 [byte]<--00h-- if MMX reg 1 [byte] NOT > MMX reg 2 [byte]	1/1
Memory with MMX Register	0F64 [mod mm r/m]	MMX reg [byte] <--FFh-- if memory[byte] > MMX reg [byte] MMX reg [byte] <--00h-- if memory[byte] NOT > MMX reg [byte]	1/1
PCMPGTD *Pack Compare Greater Than Dword*			
MMX Register 2 to MMX Register1	0F66 [11 mm1 mm2]	MMX reg 1 [dword] <--FFFF FFFFh-- if MMX reg 1 [dword] > MMX reg 2 [dword] MMX reg 1 [dword]<--0000 0000h--if MMX reg 1 [dword]NOT > MMX reg 2 [dword]	1/1
Memory with MMX Register	0F66 [mod mm r/m]	MMX reg [dword] <--FFFF FFFFh-- if memory[dword] > MMX reg [dword] MMX reg [dword] <--0000 0000h-- if memory[dword] NOT > MMX reg [dword]	1/1

Table 6-25. 6x86MX Processor MMX Instruction Set Clock Count Summary (Continued)

MMX INSTRUCTIONS	OPCODE	OPERATION	CLOCK COUNT LATENCY/ THROUGHPUT
PCMPGTW *Pack Compare Greater Than Word*			
MMX Register 2 to MMX Register1	0F65 [11 mm1 mm2]	MMX reg 1 [word] <--FFFFh-- if MMX reg 1 [word] > MMX reg 2 [word] MMX reg 1 [word]<--0000h-- if MMX reg 1 [word] NOT > MMX reg 2 [word]	1/1
Memory with MMX Register	0F65 [mod mm r/m]	MMX reg [word] <--FFFFh-- if memory[word] > MMX reg [word] MMX reg [word] <--0000h-- if memory[word] NOT > MMX reg [word]	1/1
PMADDWD *Packed Multiply and Add*			
MMX Register 2 to MMX Register 1	0FF5 [11 mm1 mm2]	MMX reg 1 [dword] <--add-- [dword]<---- MMX reg 1 [sign word]*MMX reg 2[sign word]	2/1
Memory to MMX Register	0FF5 [mod mm r/m]	MMX reg 1 [dword] <--add-- [dword] <---- memory[sign word] * Memory[sign word]	2/1
PMULHW *Packed Multiply High*			
MMX Register 2 to MMX Register1	0FE5 [11 mm1 mm2]	MMX reg 1 [word] <--upper bits-- MMX reg 1 [sign word] * MMX reg 2 [sign word]	2/1
Memory to MMX Register	0FE5 [mod mm r/m]	MMX reg 1 [word] <--upper bits-- memory [sign word] * Memory [sign word]	2/1
PMULLW *Packed Multiply Low*			
MMX Register 2 to MMX Register1	0FD5 [11 mm1 mm2]	MMX reg 1 [word] <--lower bits-- MMX reg 1 [sign word] * MMX reg 2 [sign word]	2/1
Memory to MMX Register	0FD5 [mod mm r/m]	MMX reg 1 [word] <--lower bits-- memory [sign word] * Memory [sign word]	2/1
POR *Bitwise OR*			
MMX Register 2 to MMX Register1	0FEB [11 mm1 mm2]	MMX Reg 1 [qword] <--logic OR-- MMX Reg 1 [qword], MMX Reg 2 [qword]	1/1
Memory to MMX Register	0FEB [mod mm r/m]	MMX Reg [qword] <--logic OR-- MMX Reg [qword], memory[qword]	1/1
PSLLD *Packed Shift Left Logical Dword*			
MMX Register 1 by MMX Register 2	0FF2 [11 mm1 mm2]	MMX reg 1 [dword] <--shift left, shifting in zeroes by MMX reg 2 [dword]--	1/1
MMX Register by Memory	0FF2 [mod mm r/m]	MMX reg [dword] <--shift left, shifting in zeroes by memory[dword]--	1/1
MMX Register by Immediate	0F72 [11 110 mm] #	MMX reg [dword] <--shift left, shifting in zeroes by [im byte]--	1/1
PSLLQ *Packed Shift Left Logical Qword*			
MMX Register 1 by MMX Register 2	0FF3 [11 mm1 mm2]	MMX reg 1 [qword] <--shift left, shifting in zeroes by MMX reg 2 [qword]--	1/1
MMX Register by Memory	0FF3 [mod mm r/m]	MMX reg [qword] <--shift left, shifting in zeroes by[qword]--	1/1
MMX Register by Immediate	0F73 [11 110 mm] #	MMX reg [qword] <--shift left, shifting in zeroes by[im byte]--	1/1
PSLLW *Packed Shift Left Logical Word*			
MMX Register 1 by MMX Register 2	0FF1 [11 mm1 mm2]	MMX reg 1 [word] <--shift left, shifting in zeroes by MMX reg 2 [word]--	1/1
MMX Register by Memory	0FF1 [mod mm r/m]	MMX reg [word] <--shift left, shifting in zeroes by memory[word]--	1/1
MMX Register by Immediate	0F71 [11 110mm] #	MMX reg [word] <--shift left, shifting in zeroes by[im byte]--	1/1
PSRAD *Packed Shift Right Arithmetic* Dword			
MMX Register 1 by MMX Register 2	0FE2 [11 mm1 mm2]	MMX reg 1 [dword] <--arith shift right, shifting in zeroes by MMX reg 2 [dword--]	1/1
MMX Register by Memory	0FE2 [mod mm r/m]	MMX reg [dword] <--arith shift right, shifting in zeroes by memory[dword]--	1/1
MMX Register by Immediate	0F72 [11 100 mm] #	MMX reg [dword] <--arith shift right, shifting in zeroes by [im byte]--	1/1

Table 6-25. 6x86MX Processor MMX Instruction Set Clock Count Summary (Continued)

MMX INSTRUCTIONS	OPCODE	OPERATION	CLOCK COUNT LATENCY/ THROUGHPUT
PSRAW *Packed Shift Right Arithmetic* Word MMX Register 1 by MMX Register 2 MMX Register by Memory MMX Register by Immediate	 0FE1 [11 mm1 mm2] 0FE1 [mod mm r/m] 0F71 [11 100 mm] #	 MMX reg 1 [word] <--arith shift right, shifting in zeroes by MMX reg 2 [word]-- MMX reg [word] <--arith shift right, shifting in zeroes by memory[word--] MMX reg [word] <--arith shift right, shifting in zeroes by [im byte]--	 1/1 1/1 1/1
PSRLD *Packed Shift Right Logical Dword* MMX Register 1 by MMX Register 2 MMX Register by Memory MMX Register by Immediate	 0FD2 [11 mm1 mm2] 0FD2 [mod mm r/m] 0F72 [11 010 mm] #	 MMX reg 1 [dword] <--shift right, shifting in zeroes by MMX reg 2 [dword]-- MMX reg [dword] <--shift right, shifting in zeroes by memory[dword]-- MMX reg [dword] <--shift right, shifting in zeroes by[im byte]--	 1/1 1/1 1/1
PSRLQ *Packed Shift Right Logical Qword* MMX Register 1 by MMX Register 2 MMX Register by Memory MMX Register by Immediate	 0FD3 [11 mm1 mm2] 0FD3 [mod mm r/m] 0F73 [11 010 mm] #	 MMX reg 1 [qword] <--shift right, shifting in zeroes by MMX reg 2 [qword] MMX reg [qword] <--shift right, shifting in zeroes by memory[qword] MMX reg [qword] <--shift right, shifting in zeroes by [im byte]	 1/1 1/1 1/1
PSRLW *Packed Shift Right Logical Word* MMX Register 1 by MMX Register 2 MMX Register by Memory MMX Register by Immediate	 0FD1 [11 mm1 mm2] 0FD1 [mod mm r/m] 0F71 [11 010 mm] #	 MMX reg 1 [word] <--shift right, shifting in zeroes by MMX reg 2 [word] MMX reg [word] <--shift right, shifting in zeroes by memory[word] MMX reg [word] <--shift right, shifting in zeroes by imm[word]	 1/1 1/1 1/1
PSUBB *Subtract Byte With Wrap-Around* MMX Register 2 to MMX Register1 Memory to MMX Register	 0FF8 [11 mm1 mm2] 0FF8 [mod mm r/m]	 MMX reg 1 [byte] <---- MMX reg 1 [byte] subtract MMX reg 2 [byte] MMX reg [byte] <---- MMX reg [byte] subtract memory [byte]	 1/1 1/1
PSUBD *Subtract Dword With Wrap-Around* MMX Register 2 to MMX Register1 Memory to MMX Register	 0FFA [11 mm1 mm2] 0FFA [mod mm r/m]	 MMX reg 1 [dword] <---- MMX reg 1 [dword] subtract MMX reg 2 [dword] MMX reg [dword] <---- MMX reg [dword] subtract memory [dword]	 1/1 1/1
PSUBSB *Subtract Byte Signed With Saturation* MMX Register 2 to MMX Register1 Memory to MMX Register	 0FE8 [11 mm1 mm2] 0FE8 [mod mm r/m]	 MMX reg 1 [sign byte] <--sat-- MMX reg 1 [sign byte] subtract MMX reg 2 [sign byte] MMX reg [sign byte] <--sat-- MMX reg [sign byte] subtract memory [sign byte]	 1/1 1/1
PSUBSW *Subtract Word Signed With Saturation* MMX Register 2 to MMX Register1 Memory to MMX Register	 0FE9 [11 mm1 mm2] 0FE9 [mod mm r/m]	 MMX reg 1 [sign word] <--sat-- MMX reg 1 [sign word] subtract MMX reg 2 [sign word] MMX reg [sign word] <--sat-- MMX reg [sign word] subtract memory [sign word]	 1/1 1/1
PSUBUSB *Subtract Byte Unsigned With Saturation* MMX Register 2 to MMX Register1 Memory to MMX Register	 0FD8 [11 mm1 mm2] 0FD8 [11 mm reg]	 MMX reg 1 [byte] <--sat-- MMX reg 1 [byte] subtract MMX reg 2 [byte] MMX reg [byte] <--sat-- MMX reg [byte] subtract memory [byte]	 1/1 1/1
PSUBUSW *Subtract Word Unsigned With Saturation* MMX Register 2 to MMX Register1 Memory to MMX Register	 0FD9 [11 mm1 mm2] 0FD9 [11 mm reg]	 MMX reg 1 [word] <--sat-- MMX reg 1 [word] subtract MMX reg 2 [word] MMX reg [word] <--sat-- MMX reg [word] subtract memory [word]	 1/1 1/1
PSUBW *Subtract Word With Wrap-Around* MMX Register 2 to MMX Register1 Memory to MMX Register	 0FF9 [11 mm1 mm2] 0FF9 [mod mm r/m]	 MMX reg 1 [word] <---- MMX reg 1 [word] subtract MMX reg 2 [word] MMX reg [word] <---- MMX reg [word] subtract memory [word]	 1/1 1/1

Table 6-25. 6x86MX Processor MMX Instruction Set Clock Count Summary (Continued)

MMX INSTRUCTIONS	OPCODE	OPERATION	CLOCK COUNT LATENCY/ THROUGHPUT
PUNPCKHBW *Unpack High Packed Byte Data to Packed Words*			
MMX Register 2 to MMX Register1	0F68 [11 mm1 mm2]	MMX reg 1 [byte] <--interleave-- MMX reg 1 [up byte], MMX reg 2 [up byte]	1/1
Memory to MMX Register	0F68 [11 mm reg]	MMX reg [byte] <--interleave-- memory [up byte], MMX reg [up byte]	1/1
PUNPCKHDQ *Unpack High Packed Dword Data to Qword*			
MMX Register 2 to MMX Register1	0F6A [11 mm1 mm2]	MMX reg 1 [dword] <--interleave-- MMX reg 1 [up dword], MMX reg 2 [up dword]	1/1
Memory to MMX Register	0F6A [11 mm reg]	MMX reg [dword] <--interleave-- memory [up dword], MMX reg [up dword]	1/1
PUNPCKHWD *Unpack High Packed Word Data to Packed Dwords*			
MMX Register 2 to MMX Register1	0F69 [11 mm1 mm2]	MMX reg 1 [word] <--interleave-- MMX reg 1 [up word], MMX reg 2 [up word]	1/1
Memory to MMX Register	0F69 [11 mm reg]	MMX reg [word] <--interleave-- memory [up word], MMX reg [up word]	1/1
PUNPCKLBW *Unpack Low Packed Byte Data to Packed Words*			
MMX Register 2 to MMX Register1	0F60 [11 mm1 mm2]	MMX reg 1 [word] <--interleave-- MMX reg 1 [low byte], MMX reg 2 [low byte]	1/1
Memory to MMX Register	0F60 [11 mm reg]	MMX reg [word] <--interleave-- memory [low byte], MMX reg [low byte]	1/1
PUNPCKLDQ *Unpack Low Packed Dword Data to Qword*			
MMX Register 2 to MMX Register1	0F62 [11 mm1 mm2]	MMX reg 1 [word] <--interleave-- MMX reg 1 [low dword], MMX reg 2 [low dword]	1/1
Memory to MMX Register	0F62 [11 mm reg]	MMX reg [word] <--interleave-- memory [low dword], MMX reg [low dword]	1/1
PUNPCKLWD *Unpack Low Packed Word Data to Packed Dwords*			
MMX Register 2 to MMX Register1	0F61 [11 mm1 mm2]	MMX reg 1 [word] <--interleave-- MMX reg 1 [low word], MMX reg 2 [low word]	1/1
Memory to MMX Register	0F61 [11 mm reg]	MMX reg [word] <--interleave-- memory [low word], MMX reg [low word]	1/1
PXOR *Bitwise XOR*			
MMX Register 2 to MMX Register1	0FEF [11 mm1 mm2]	MMX Reg 1 [qword] <--logic exclusive OR-- MMX Reg 1 [qword], MMX Reg 2 [qword]	1/1
Memory to MMX Register	0FEF [11 mm reg]	MMX Reg [qword] <--logic exclusive OR-- memory[qword], MMX Reg [qword]	1/1

Cyrix® Advancing the Standards

Ordering Information

Note: For further information concerning Performance Ratings, visit our website at www.cyrix.com.

The Cyrix 6x86MX CPU part numbers are listed below.

Cyrix 6x86MX™ Part Numbers

PART NUMBER	CLOCK MULTIPLIER	FREQUENCY (MHz)	
		BUS	INTERNAL
6x86MX - PR166GP	2.0	66	133
6x86MX - PR166GP	2.5	60	150
6x86MX - PR200GP	2.5	66	166
6x86MX - PR233GP	2.5	75	188
6x86MX - PR266GP	2.5	83	208

INDEX

R

S

T

U

V

W

Cyrix Worldwide Offices

United States
Corporate Office
Richardson, Texas
Tel: (972) 968-8388
Fax: (972) 699-9857

Tech Support and Sales: (800) 462-9749
Internet: tech_support@cyrix.com
BBS: (972) 968-8610 (up to 28.8K baud)

See us on the Internet Worldwide Web:
www.cyrix.com

Europe
United Kingdom
Cyrix International Ltd.
Tel: +44 (0) 1 793 417777
Fax: +44 (0) 1 793 417770

Japan
Cyrix K.K.
Tel: 81-45-471-1661
Fax: 81-45-471-1666

Taiwan
Cyrix International, Inc.
Tel: 886-2-718-4118
Fax: 886-2-719-5255

Hong Kong
Cyrix International, Inc.
Tel: (852) 2485-2285
Fax: (852) 2485-2920

Cyrix Corporation
P.O. Box 850118
Richardson, TX 75085-0118
Tel: (972) 968-8388
Fax: (972) 699-9857

Oder Number: 94xxx-xx - January 1997
Printed in the USA on recycled paper.