Component Operation 16

The embedded Pentium® processor has an optimized superscalar micro-architecture capable of executing two instructions in a single clock. A 64-bit external bus, separate data and instruction caches, write buffers, branch prediction, and a pipelined floating-point unit combine to sustain the high execution rate. These architectural features and their operation are discussed in this chapter.

16.1 Pipeline and Instruction Flow

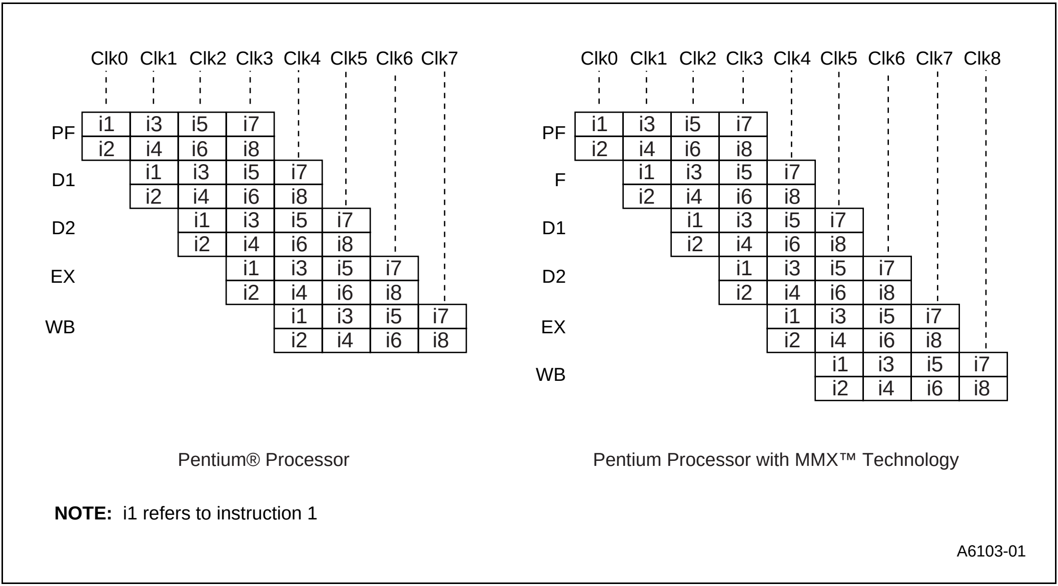
The integer instructions traverse a five stage pipeline in the embedded Pentium processor (the embedded Pentium® processor with MMX™ technology has an additional pipeline stage). The pipeline stages are as follows:

PF	Prefetch
F	Fetch (embedded Pentium processor with MMX technology only)
D1	Instruction Decode
D2	Address Generate
EX	Execute - ALU and Cache Access
WB	Writeback

The embedded Pentium processor is a superscalar machine, built around two general purpose integer pipelines and a pipelined floating-point unit capable of executing two instructions in parallel. Both pipelines operate in parallel, allowing integer instructions to execute in a single clock in each pipeline. Figure 16-1 depicts instruction flow in the embedded Pentium processor.

The pipelines in the embedded Pentium processor are called the “u” and “v” pipes and the process of issuing two instructions in parallel is termed “pairing.” The u-pipe can execute any instruction in the Intel architecture, whereas the v-pipe can execute “simple” instructions as defined in ““Pairing Two MMX™ Instructions” on page 16-194” section of this chapter. When instructions are paired, the instruction issued to the v-pipe is always the next sequential instruction after the one issued to the u-pipe.

Figure 16-1. Embedded Pentium® Processor Pipeline Execution


Clk0 Clk1 Clk2 Clk3 Clk4 Clk5 Clk6 Clk7
PF
D1
D2
EX
WB
i1 i2 i3 i4 i5 i6 i7 i8
Pentium® Processor
Clk0 Clk1 Clk2 Clk3 Clk4 Clk5 Clk6 Clk7 Clk8
PF
F
D1
D2
EX
WB
Pentium Processor with MMX™ Technology
NOTE: i1 refers to instruction 1
A6103-01

16.1.1 Integer Pipeline Description

The embedded Pentium processor pipeline has been optimized to achieve higher throughput compared to previous generations of Intel architecture processors.

The first stage of the pipeline is the Prefetch (PF) stage in which instructions are prefetched from the on-chip instruction cache or memory. Because the processor has separate caches for instructions and data, prefetches do not conflict with data references for access to the cache. If the requested line is not in the code cache, a memory reference is made. In the PF stage, two independent pairs of line-size (32-byte) prefetch buffers operate in conjunction with the branch target buffer. This allows one prefetch buffer to prefetch instructions sequentially while the other prefetches according to the branch target buffer predictions. The prefetch buffers alternate their prefetch paths. In the embedded Pentium processor with MMX technology, four 16-byte prefetch buffers operate in conjunction with the BTB to prefetch up to four independent instruction streams. See the “Instruction Prefetch” on page 16-181 for further details on prefetch buffers.

In the embedded Pentium processor with MMX technology only, the next pipeline stage is Fetch (F), which is used for instruction length decode. It replaces the D1 instruction-length decoder and eliminates the need for end-bits to determine instruction length. Also, any prefixes are decoded in the F stage. The Fetch stage is not supported by the embedded Pentium processor (at 100, 133, 166 MHz) or the embedded Pentium processor with VRT.

The embedded Pentium processor with MMX technology also features an instruction FIFO between the F and D1 stages. This FIFO is transparent; it does not add additional latency when it is empty. During every clock cycle, two instructions can be pushed into the instruction FIFO (depending on availability of the code bytes, and on other factors such as prefixes). Instruction pairs are pulled out of the FIFO into the D1 stage. Since the average rate of instruction execution is less than two per clock, the FIFO is normally full. As long as the FIFO is full, it can buffer any stalls that may occur during instruction fetch and parsing. If such a stall occurs, the FIFO prevents the stall from causing a stall in the execution stage of the pipe. If the FIFO is empty, an execution stall may result from the pipeline being “starved” for instructions to execute. Stalls at the FIFO entrance may be caused by long instructions or prefixes, or “extremely misaligned targets” (i.e., Branch targets that reside at the last bytes of 16-aligned bytes).

The pipeline stage after the PF stage in the embedded Pentium processor is Decode1 (D1), in which two parallel decoders work to decode and issue the next two sequential instructions. The decoders determine whether one or two instructions can be issued contingent upon the instruction pairing rules described in “Pairing Two MMX™ Instructions” on page 16-194.” The embedded Pentium processor requires an extra D1 clock to decode instruction prefixes. Prefixes are issued to the u-pipe at the rate of one per clock without pairing. After all prefixes have been issued, the base instruction is issued and paired according to the pairing rules. The one exception to this is that the embedded Pentium processor decodes near conditional jumps (long displacement) in the second opcode map (0FH prefix) in a single clock in either pipeline. The embedded Pentium processor with MMX technology handles 0FH as part of the opcode and not as a prefix. Consequently, 0FH does not take one extra clock to get into the FIFO. Note that in the embedded Pentium processor with MMX technology, MMX instructions can be paired. This is discussed in “Pairing Two MMX™ Instructions” on page 16-194.

The D1 stage is followed by Decode2 (D2) in which addresses of memory resident operands are calculated. In the Intel486™ processor, instructions containing both a displacement and an immediate or instructions containing a base and index addressing mode require an additional D2 clock to decode. The embedded Pentium processor removes both of these restrictions and is able to issue instructions in these categories in a single clock.

The embedded Pentium processor uses the Execute (EX) stage of the pipeline for both ALU operations and for data cache access; therefore, those instructions specifying both an ALU operation and a data cache access require more than one clock in this stage. In EX, all u-pipe instructions and all v-pipe instructions except conditional branches are verified for correct branch prediction. Microcode is designed to utilize both pipelines; therefore, those instructions requiring microcode execute faster.

The final stage is Writeback (WB), in which instructions are enabled to modify the processor state and complete execution. In this stage, v-pipe conditional branches are verified for correct branch prediction.

During their progression through the pipeline, instructions may be stalled due to certain conditions. Both the u-pipe and v-pipe instructions enter and leave the D1 and D2 stages in unison. When an instruction in one pipe is stalled, the instruction in the other pipe is also stalled at the same pipeline stage. Thus both the u-pipe and the v-pipe instructions enter the EX stage in unison. Once in EX, if the u-pipe instruction is stalled, then the v-pipe instruction (if any) is also stalled. If the v-pipe instruction is stalled, then the instruction paired with it in the u-pipe is not allowed to advance. No successive instructions are allowed to enter the EX stage of either pipeline until the instructions in both pipelines have advanced to WB.

16.1.1.1 Instruction Prefetch

In the embedded Pentium processor PF stage, two independent pairs of line-size (32-byte) prefetch buffers operate in conjunction with the branch target buffer. Only one prefetch buffer actively requests prefetches at any given time. Prefetches are requested sequentially until a branch instruction is fetched. When a branch instruction is fetched, the branch target buffer (BTB) predicts whether the branch will be taken or not. If the branch is predicted not taken, prefetch requests continue linearly. On a predicted taken branch the other prefetch buffer is enabled and begins to prefetch as though the branch were taken. If a branch is discovered mispredicted, the instruction pipelines are flushed and prefetching activity starts over.

The embedded Pentium processor with MMX technology's prefetch stage has four 16-byte buffers that can prefetch up to four independent instruction streams, based on predictions made by the BTB. In this case, the Branch Target Buffer predicts whether the branch will be taken or not in the PF stage. The embedded Pentium processor with MMX technology features an enhanced two-stage Branch prediction algorithm, compared to the embedded Pentium processor.

For more information on branch prediction, see "Component Introduction" on page 15-175.

16.1.2 Integer Instruction Pairing Rules

The embedded Pentium processor can issue one or two instructions every clock. In order for the processor to issue two instructions simultaneously, they must satisfy the following conditions:

- Both instructions in the pair must be "simple" as defined below.
- There must be no read-after-write or write-after-write register dependencies between the instructions.
- Neither instruction may contain both a displacement and an immediate.
- Instructions with prefixes can only occur in the u-pipe (except for JCC instructions with a 0FH prefix on the embedded Pentium processor and instructions with a 0FH, 66H or 67H prefix on the embedded Pentium processor with MMX technology).
- Instruction prefixes are treated as separate 1-byte instructions (except for all 0FH prefixed instructions in the embedded Pentium processor with MMX technology).

Simple instructions are entirely hardwired; they do not require any microcode control and, in general, execute in one clock. The exceptions are the ALU mem,reg and ALU reg,mem instructions which are three and two clock operations, respectively. Sequencing hardware is used to allow them to function as simple instructions. The following integer instructions are considered simple and may be paired:

- mov reg, reg/mem/imm
- mov mem, reg/imm
- alu reg, reg/mem/imm
- alu mem, reg/imm
- inc reg/mem
- dec reg/mem
- push reg/mem
- pop reg
- lea reg,mem
- jmp/call/jcc near
- nop
- test reg, reg/mem
- test acc, imm

In addition, conditional and unconditional branches may be paired only if they occur as the second instruction in the pair. They may not be paired with the next sequential instruction. Also, SHIFT/ROT by 1 and SHIFT by IMM may pair as the first instruction in a pair.

The register dependencies that prohibit instruction pairing include implicit dependencies via registers or flags not explicitly encoded in the instruction. For example, an ALU instruction in the u-pipe (which sets the flags) may not be paired with an ADC or an SBB instruction in the v-pipe. There are two exceptions to this rule. The first is the commonly occurring sequence of compare and branch, which may be paired. The second exception is pairs of pushes or pops. Although these instructions have an implicit dependency on the stack pointer, special hardware is included to allow these common operations to proceed in parallel.

Although two paired instructions generally may proceed in parallel independently, there is an exception for paired “read-modify-write” instructions. Read-modify-write instructions are ALU operations with an operand in memory. When two of these instructions are paired, there is a sequencing delay of two clocks in addition to the three clocks required to execute the individual instructions.

Although instructions may execute in parallel, their behavior as seen by the programmer is exactly the same as if they were executed sequentially.

Information regarding pairing of FPU and MMX instructions is discussed in “Floating-Point Unit” on page 16-185 and “Intel MMX™ Technology Unit” on page 16-189 For additional details on code optimization, refer to *Optimizing for Intel’s 32-Bit Processors* (order number 241799).

16.2 Branch Prediction

The embedded Pentium processor uses a Branch Target Buffer (BTB) to predict the outcome of branch instructions, thereby minimizing pipeline stalls due to prefetch delays.

The processor accesses the BTB with the address of the instruction in the D1 stage. It contains a Branch prediction state machine with four states: (1) strongly not taken, (2) weakly not taken, (3) weakly taken, and (4) strongly taken. In the event of a correct prediction, a branch executes without pipeline stalls or flushes. Branches that miss the BTB are assumed to be not taken. Conditional and unconditional near branches and near calls execute in one clock and may be executed in parallel with other integer instructions. A mispredicted branch (whether a BTB hit or miss) or a correctly predicted branch with the wrong target address causes the pipelines to be flushed and the correct target to be fetched. Incorrectly predicted unconditional branches incur an additional three clock delay, incorrectly predicted conditional branches in the u-pipe incur an additional three clock delay, and incorrectly predicted conditional branches in the v-pipe incur an additional four clock delay.

The benefits of branch prediction are illustrated in the following example. Consider the following loop from a benchmark program for computing prime numbers:

```
for(k=i+prime;k<=SIZE;k+=prime)
    flags[k]=FALSE;
```

A popular compiler generates the following assembly code (prime is allocated to ECX, K is allocated to EDX, and AL contains the value FALSE):

```
inner_loop:
    mov byte ptr flags[edx],al
    add edx,ecx
    cmp edx, SIZE
    jle inner_loop
```

Each iteration of this loop executes in six clocks on the Intel486™ processor. On the embedded Pentium processor, the MOV is paired with the ADD; the CMP with the JLE. With branch prediction, each loop iteration executes in two clocks.

Note: The dynamic branch prediction algorithm speculatively runs code fetch cycles to addresses corresponding to instructions executed some time in the past. Such code fetch cycles are run based on past execution history, regardless of whether the instructions retrieved are relevant to the currently executing instruction sequence.

One effect of the branch prediction mechanism is that the processor may run code fetch bus cycles to retrieve instructions that are never executed. Although the opcodes retrieved are discarded, the system must complete the code fetch bus cycle by returning BRDY#. It is particularly important that the system return BRDY# for all code fetch cycles, regardless of the address.

It should also be noted that upon entering SMM, the branch target buffer (BTB) is not flushed and thus it is possible to get a speculative prefetch to an address outside of SMRAM address space due to branch predictions based on code executed prior to entering SMM. If this occurs, the system must still return BRDY# for each code fetch cycle.

Furthermore, the processor may run speculative code fetch cycles to addresses beyond the end of the current code segment (approximately 100 bytes past end of last executed instruction). Although the processor may prefetch beyond the CS limit, it will not attempt to execute beyond the CS limit. Instead, it will raise a GP fault. Thus, segmentation cannot be used to prevent speculative code fetches to inaccessible areas of memory. On the other hand, the processor never runs code fetch cycles to inaccessible pages (i.e., not present pages or pages with incorrect access rights), so the paging mechanism guards against both the fetch and execution of instructions in inaccessible pages.

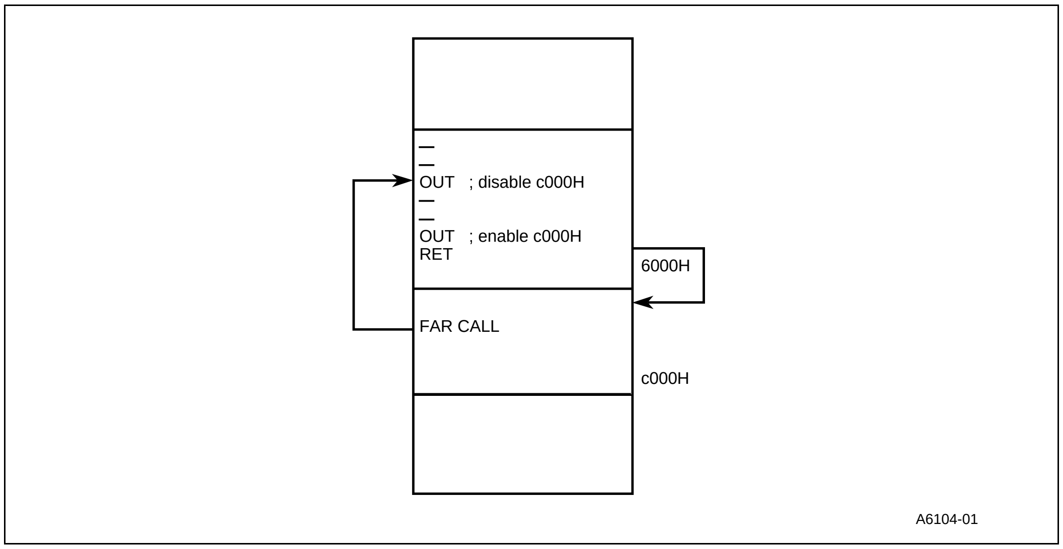
For memory reads and writes, both segmentation and paging prevent the generation of bus cycles to inaccessible regions of memory. If paging is not used, branch prediction can be disabled by setting TR12.NBP (bit 0)[1] and flushing the BTB by loading CR3 before disabling any areas of memory. Branch prediction can be re-enabled after re-enabling memory.

The following is an example of a situation that may occur:

1. Code passes control to segment at address C000H.
2. Code transfers control to code at different address (6000H) by using the FAR CALL instruction.
3. This portion of the code does an I/O write to a port that disables memory at address C000H.
4. At the end of this segment, an I/O write is performed to re-enable memory at address C000H.
5. Following the OUT instruction, there is a RET instruction to C000H segment.

1. Please refer to Chapter 26 of this volume.

Figure 16-2. Branch Prediction Example



The branch prediction mechanism of the embedded Pentium processor, however, predicts that the RET instruction is going to transfer control to the segment at address C000H and performs a prefetch from that address prior to the OUT instruction that re-enables that memory address. The result is that no BRDY is returned for that prefetch cycle and the system hangs.

In this case, branch prediction should be disabled (by setting TR12.NBP and flushing the BTB by loading CR3) prior to disabling memory at address C000H, and re-enabled after the RET instruction by clearing TR12.NBP as indicated above. (See Chapter 26, “Model Specific Registers and Functions” for more information on register operation.)

In the embedded Pentium processor with MMX technology, the branch prediction algorithm changes from the embedded Pentium processor in the following ways:

- BTB Lookup is done when the branch is in the PF stage.
- The BTB Lookup tag is the Prefetch address.
- A Lookup in the BTB performs a search spanning sixteen consecutive bytes.
- BTB can contain four branch instructions for each line of 16 bytes.
- BTB is constructed from four independent Banks. Each Bank contains 64 entries and is 4-way associative.
- Enhanced two-stage branch prediction algorithm.

16.3 Floating-Point Unit

The floating-point unit (FPU) of the embedded Pentium processor is integrated with the integer unit on the same chip. It is heavily pipelined. The FPU is designed to be able to accept one floating-point operation every clock. It can receive up to two floating-point instructions every clock, one of which must be an exchange instruction.

For information on code optimization, please refer to *Optimizing for Intel’s 32-Bit Processors* (order number 241799).

16.3.1 Floating-Point Pipeline Stages

The embedded Pentium processor FPU has eight pipeline stages, the first five of which it shares with the integer unit. Integer instructions pass through only the first five stages. Integer instructions use the fifth (X1) stage as a WB (write-back) stage. The eight FP pipeline stages, and the activities that are performed in them are summarized below:

PF	Prefetch
F	Fetch (applicable to the embedded Pentium processor with MMX technology only)
D1	Instruction decode
D2	Address generation
EX	Memory and register read; conversion of FP data to external memory format and memory write
X1	Floating-Point Execute stage one; conversion of external memory format to internal FP data format and write operand to FP register file; bypass 1 (bypass 1 is described in "FPU Bypasses" on page 16-188)
X2	Floating-Point Execute stage two
WF	Perform rounding and write floating-point result to register file; bypass 2 (bypass 2 is described in "FPU Bypasses" on page 16-188)
ER	Error Reporting/Update Status Word

16.3.2 Instruction Issue

The rules of how floating-point (FP) instructions get issued on the embedded Pentium processor are described as follows:

1. FP instructions do not get paired with integer instructions. However, a limited pairing of two FP instructions can be performed.
2. When a pair of FP instructions is issued to the FPU, only the FXCH instruction can be the second instruction of the pair. The first instruction of the pair must be one of a set F where F = [FLD single/double, FLD ST(i), all forms of FADD, FSUB, FMUL, FDIV, FCOM, FUCOM, FTST, FABS, FCHS].
3. FP instructions other than the FXCH instruction and other than instructions belonging to set F (defined in rule 2) always get issued singly to the FPU.
4. FP instructions that are not directly followed by an FP exchange instruction are issued singly to the FPU.

The embedded Pentium processor stack architecture instruction set requires that all instructions have one source operand on the top of the stack. Since most instructions also have their destination as the top of the stack, most instructions see a "top of stack bottleneck." New source operands must be brought to the top of the stack before we can issue an arithmetic instruction on them. This calls for extra usage of the exchange instruction, which allows the programmer to bring an available operand to the top of the stack. The processor FPU uses pointers to access its registers to allow fast execution of exchanges and the execution of exchanges in parallel with other floating-point instructions. An FP exchange that is paired with other FP instructions takes zero clocks for its execution. Because such exchanges can be executed in parallel, it is recommended that one use them when necessary to overcome the stack bottleneck.

Note that when exchanges are paired with other floating-point instructions, they should not be followed immediately by integer instructions. The processor stalls such integer instructions for a clock if the FP pair is declared safe, or for four clocks if the FP pair is unsafe.

Also note that the FP exchange must always follow another FP instruction to get paired. The pairing mechanism does not allow the FP exchange to be the first instruction of a pair that is issued in parallel. If an FP exchange is not paired, it takes one clock for its execution.

16.3.3 Safe Instruction Recognition

The embedded Pentium processor FPU performs Safe Instruction Recognition or SIR in the X1 stage of the pipeline. SIR is an early inspection of operands and opcodes to determine whether the instruction is guaranteed not to generate an arithmetic overflow, underflow, or unmasked inexact exception. An instruction is declared safe if it cannot raise any other floating-point exception, and if it does not need microcode assist for delivery of special results. If an instruction is declared safe, the next FP instruction is allowed to complete its E stage operation. If an instruction is declared unsafe, the next FP instruction stalls in the E stage until the current one completes (ER stage) with no exception. This means a four clock stall, which is incurred even if the numeric instruction that was declared unsafe does not eventually raise a floating-point exception.

For normal data, the rules used on the embedded Pentium processor for declaring an instruction safe are as follows.

On the embedded Pentium processor, if FOP = FADD/FSUB/FMUL/FDIV, the instruction is safe from arithmetic overflow, underflow, and unmasked inexact exceptions if:

1. Both operands have unbiased exponent ≤1FFEH
   and
2. Both operands have unbiased exponent ≥–1FFEH
   and
3. The inexact exception is masked.

Similarly, on the embedded Pentium processor with MMX technology, if FOP = FADD/FSUB/FMUL/FDIV, the instruction is safe from arithmetic overflow, underflow, and unmasked inexact exceptions if:

1. Both operands have unbiased exponent ≤1000H
   and
2. Both operands have unbiased exponent ≥–0FFFH
   and
3. The inexact exception is masked.

Note that arithmetic overflow of the double precision format occurs when the unbiased exponent of the result is ≥400H, and underflow occurs when the exponent is ≤–3FFH. Hence, the SIR algorithm on the embedded Pentium processor allows improved throughput on a much greater range of numbers than that spanned by the double precision format.

16.3.4 FPU Bypasses

The following section describes the floating-point register file bypasses that exist on the embedded Pentium processor. The register file has two write ports and two read ports. The read ports are used to read data out of the register file in the E stage. One write port is used to write data into the register file in the X1 stage, and the other in the WF stage. A bypass allows data that is about to be written into the register file to be available as an operand that is to be read from the register file by any succeeding floating-point instruction. A bypass is specified by a pair of ports (a write port and a read port) that get circumvented. Using the bypass, data is made available even before actually writing it to the register file.

The following procedures are implemented:

1. Bypass the X1 stage register file write port and the E stage register file read port.
2. Bypass the WF stage register file write port and the E stage register file read port.

With bypass 1, the result of a floating-point load (that writes to the register file in the X1 stage) can bypass the X1 stage write and be sent directly to the operand fetch stage or E stage of the next instruction.

With bypass 2, the result of any arithmetic operation can bypass the WF stage write to the register file, and be sent directly to the desired execution unit as an operand for the next instruction.

Note that the FST instruction reads the register file with a different timing requirement, so that for the FST instruction, which attempts to read an operand in the E stage:

1. There is no bypassing the X1 stage write port and the E stage read port, i.e., no added bypass for FLD followed by FST. Thus FLD (double) followed by FST (double) takes four clocks (two for FLD, and two for FST).
2. There is no bypassing the WF stage write port and the E stage read port. The E stage read for the FST happens only in the clock following the WF write for any preceding arithmetic operation.

Furthermore, there is no memory bypass for an FST followed by an FLD from the same memory location.

16.3.5 Branching Upon Numeric Condition Codes

Branching upon numeric condition codes is accomplished by transferring the floating-point SW to the integer FLAGS register and branching on it. The “test numeric condition codes and branch” construct looks like:

```
FP instruction1; instruction whose effects on the status word are to be examined;
“numeric_test_and_branch_construct”:
FSTSW AX; move the status word to the ax register.
SAHF; transfer the value in ah to the lower half of the eflags register.
JC xyz; jump upon the condition codes in the eflags register.
```

Note that all FP instructions update the status word only in the ER stage. Hence there is a built-in status word interlock between FP instruction1 and the FSTSW AX instruction. The above piece of code takes nine clocks before execution of code begins at the target of the jump. These nine clocks are counted as:

FP instruction1:	X1, X2, WF, ER (4 E stage stalls for the FSTSWAX);
FSTSW AX:	Two E clocks;
SAHF:	Two E clocks;
JC xyz:	One clock if no mispredict on branch.

Note that if there is a branch mispredict, there is a minimum of three clocks added to the clock count of nine.

It is recommended that such attempts to branch upon numeric condition codes be preceded by integer instructions; i.e., you should insert integer instructions in between FP instruction1 and the FSTSW AX instruction that is the first instruction of the “numeric test and branch” construct. This allows the elimination of up to four clocks (the 4 E-stage stalls on FSTSW AX) from the cost attributed to this construct, so that numeric branching can be accomplished in five clocks.

16.4 Intel MMX™ Technology Unit

Intel’s MMX technology, supported on the embedded Pentium processor with MMX technology, is a set of extensions to the Intel architecture that are designed to greatly enhance the performance of advanced media and communications applications. These extensions (which include new registers, data types, and instructions) are combined with a single-instruction, multiple-data (SIMD) execution model to accelerate the performance of applications such as motion video, combined graphics with video, image processing, audio synthesis, speech synthesis and compression, telephony, video conferencing, and 2D and 3D graphics, which typically use compute-intensive algorithms to perform repetitive operations on large arrays of simple, native data elements.

MMX technology defines a simple and flexible software model, with no new mode or operating-system visible state. All existing software runs correctly, without modification, on Intel architecture processors that incorporate MMX technology, even in the presence of existing and new applications that incorporate this technology.

The following sections of this chapter describe the basic programming environment for the technology, the MMX technology register set, data types and instruction set. Detailed descriptions of the MMX instructions are provided in Chapter 3 of the *Intel Architecture Software Developer’s Manual*, Volume 2. The manner in which the MMX technology extensions fit into the Intel architecture system programming model is described in Chapter 10 of the *Intel Architecture Software Developer’s Manual*, Volume 3.

16.4.1 MMX™ Technology Programming Environment

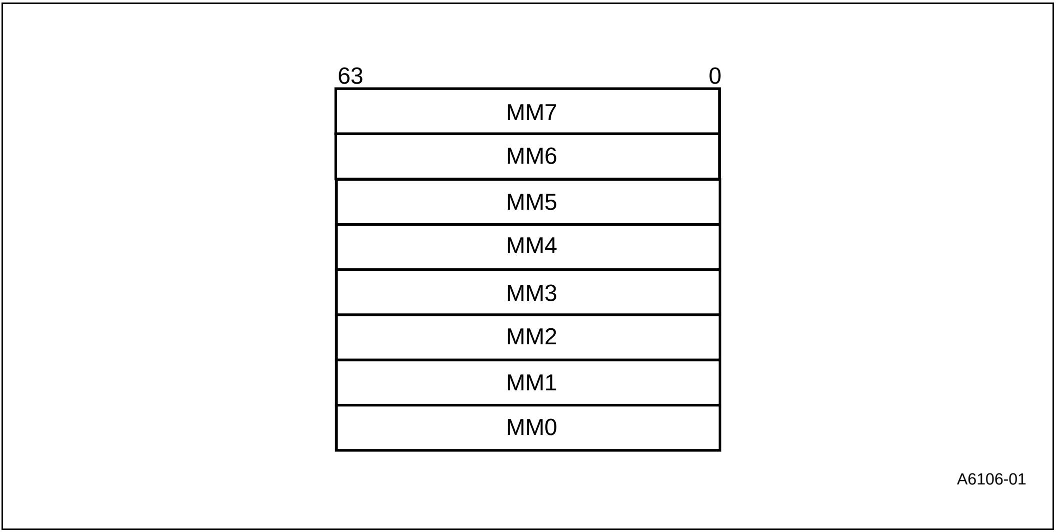
MMX technology provides the following new extensions to the Intel architecture programming environment:

- Eight MMX technology registers (MM0 through MM7)
- Four MMX technology data types (packed bytes, packed words, packed doublewords and quadword)
- The MMX technology instruction set

16.4.1.1 MMX™ Technology Registers

The MMX technology register set consists of eight 64-bit registers (Figure 16-3). The MMX instructions access the registers directly using the register names MM0 through MM7. These registers can only be used to perform calculations on MMX technology data types; they cannot be used to address memory. Addressing of MMX instruction operands in memory is handled by using the standard Intel architecture addressing modes and general-purpose registers (EAX, EBX, ECX, EDX, EBP, ESI, EDI and ESP).

Figure 16-3. MMX™ Technology Register Set



Although the MMX registers are defined in the Intel architecture as separate registers, they are aliased to the registers in the FPU data register stack (R0 through R7). (See Chapter 10 in the *Intel Architecture Software Developer's Manual*, Volume 3, for a more detailed discussion of MMX technology register aliasing.)

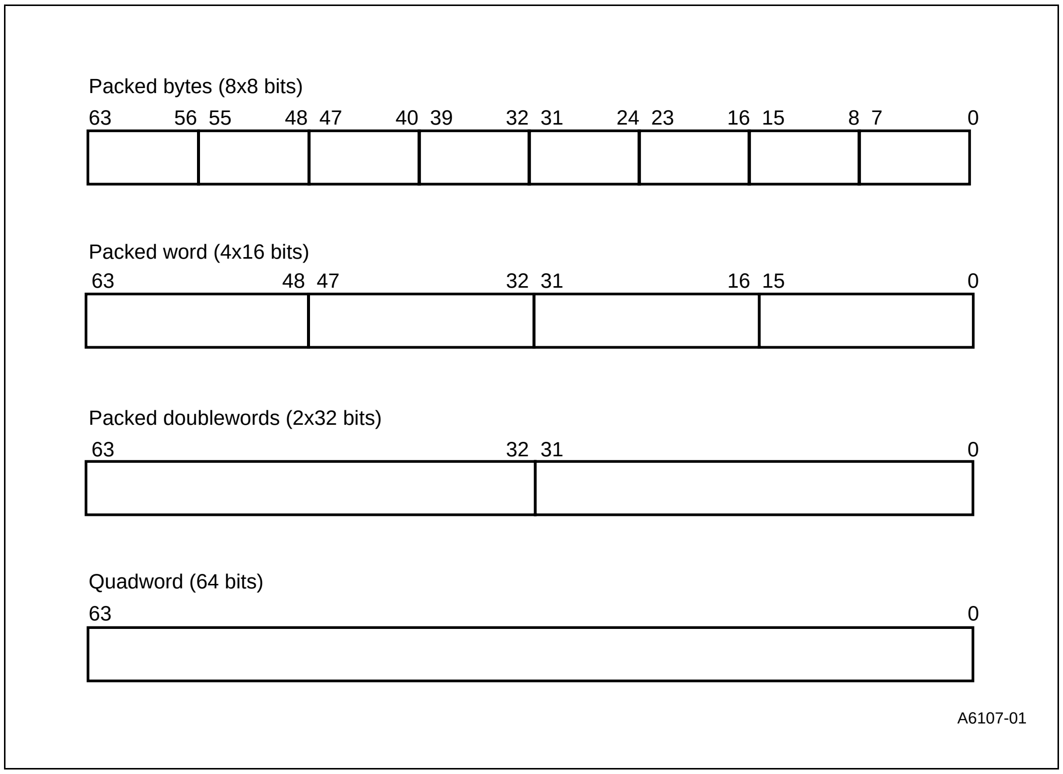
16.4.1.2 MMX™ Technology Data Types

The MMX technology defines the following new 64-bit data types (Figure 16-4):

Packed bytes	Eight bytes packed into one 64-bit quantity.
Packed words	Four (16-bit) words packed into one 64-bit quantity.
Packed doublewords	Two (32-bit) doublewords packed into one 64-bit quantity.
Quadword	One 64-bit quantity.

The bytes in the packed bytes data type are numbered 0 through 7. Byte 0 is contained in the least significant bits of the data type (bits 0 through 7) and byte 7 is contained in the most significant bits (bits 56 through 63). The words in the packed words data type are numbered 0 through 4. Word 0 is contained in the bits 0 through 15 of the data type and word 4 is contained in bits 48 through 63. The doublewords in a packed doublewords data type are numbered 0 through 1. Doubleword 0 is contained in bits 0 through 31 and doubleword 1 is contained in bits 32 through 63.

Figure 16-4. Packed Data Types



The MMX instructions move the packed data types (packed bytes, packed words or packed doublewords) and the quadword data type to-and-from memory or to-and-from the Intel architecture general-purpose registers in 64-bit blocks. However, when performing arithmetic or logical operations on the packed data types, the MMX instructions operate in parallel on the individual bytes, words or doublewords contained in a 64-bit MMX register.

When operating on the bytes, words and doublewords within packed data types, the MMX instructions recognize and operate on both signed and unsigned byte integers, word integers and doubleword integers.

16.4.1.3 Single Instruction, Multiple Data (SIMD) Execution Model

The MMX technology uses the single instruction, multiple data (SIMD) technique for performing arithmetic and logical operations on the bytes, words or doublewords packed in an MMX packed data type. For example, the PADDSB instruction adds eight signed bytes from the source operand to eight signed bytes in the destination operand and stores eight byte-results in the destination operand. This SIMD technique speeds up software performance by allowing the same operation to be carried out on multiple data elements in parallel. The MMX technology supports parallel operations on byte, word and doubleword data elements when contained in MMX packed data types.

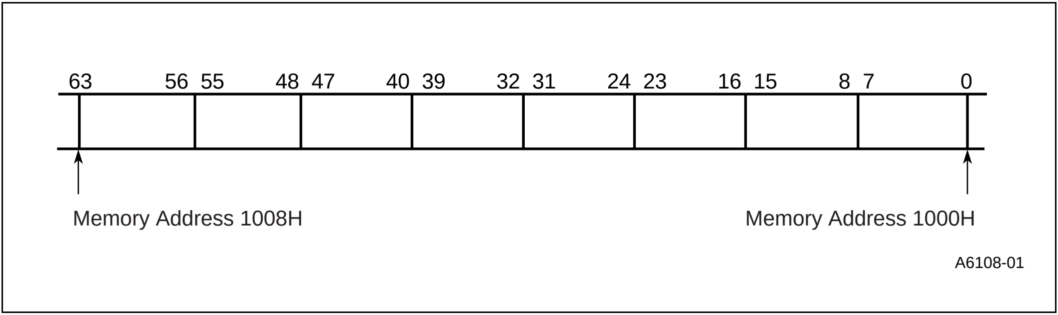
The SIMD execution model supported in the MMX technology directly addresses the needs of modern media, communications and graphics applications, which often use sophisticated algorithms that perform the same operations on a large number of small data types (bytes, words and doublewords). For example, most audio data is represented in 16-bit (word) quantities. The

MMX instructions can operate on four of these words simultaneously with one instruction. Video and graphics information is commonly represented as palletized 8-bit (byte) quantities. Here, one MMX instruction can operate on eight of these bytes simultaneously.

16.4.1.4 Memory Data Formats

When stored in memory the bytes, words and doublewords in the packed data types are stored in consecutive addresses, with the least significant byte, word or doubleword being stored at the lowest address and the more significant bytes, words or doublewords being stored at consecutively higher addresses (see Figure 16-5). The ordering of bytes, words or doublewords in memory is always little endian. That is, the bytes with the lower addresses are less significant than the bytes with the higher addresses.

Figure 16-5. Eight Packed Bytes in Memory (at Address 1000H)



16.4.1.5 MMX™ Technology Register Data Formats

Values in MMX registers have the same format as a 64-bit quantity in memory. MMX registers have two data access modes: 64-bit access mode and 32-bit access mode.

The 64-bit access mode is used for 64-bit memory access, 64-bit transfer between registers, all pack, logical and arithmetic instructions, and some unpack instructions.

The 32-bit access mode is used for 32-bit memory access, 32-bit transfer between integer registers and MMX technology registers, and some unpack instructions.

16.4.2 MMX™ Instruction Set

The MMX instruction set consists of 57 instructions, grouped into the following categories:

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

These instructions provide a rich set of operations that can be performed in parallel on the bytes, words or doublewords of an MMX packed data type.

When operating on the MMX packed data types, the data within a data type is cast by the type specified by the instruction. For example, the PADDB (add packed bytes) instruction adds two groups of eight packed bytes. The PADDW (add packed words) instruction, which adds packed words, can operate on the same 64 bits as the PADDB instruction treating 64 bits as four 16-bit words.

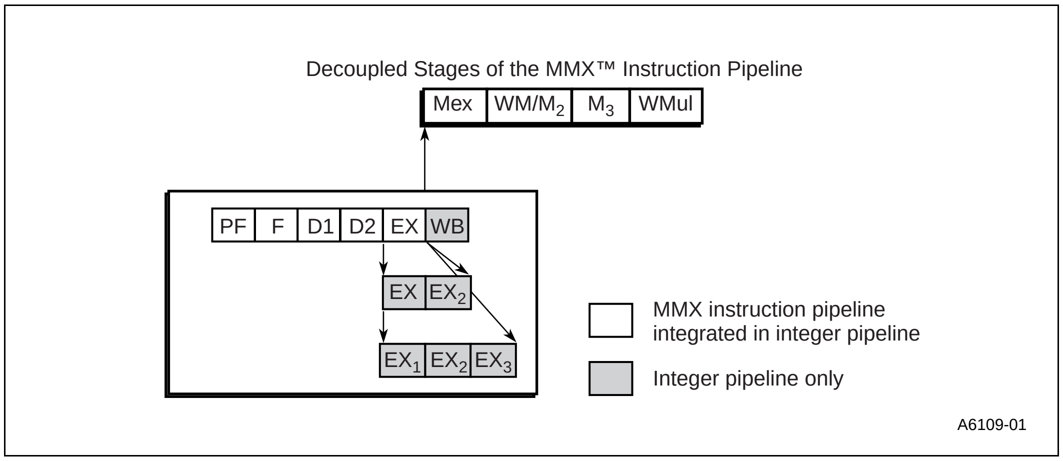
16.4.3 Intel MMX™ Technology Pipeline Stages

The MMX technology unit of the embedded Pentium processor with MMX technology has six pipeline stages. The integration of the MMX technology pipeline with the integer pipeline is very similar to that of the floating-point pipe.

The embedded Pentium processor with MMX technology adds an additional fetch stage to the pipeline. The instruction bytes are prefetched from the code cache in the prefetch (PF) stage, and they are parsed into instructions (and prefixes) in the fetch (F) stage. Additionally, any prefixes are decoded in the F stage.

When instructions execute in the two pipes, their behavior is exactly the same as if they were executed sequentially. When a stall occurs, successive instructions are not allowed to pass the stalled instruction in either pipe. Figure 16-6 shows the pipelining structure for this scheme.

Figure 16-6. MMX™ Technology Pipeline Structure


Decoupled Stages of the MMX™ Instruction Pipeline
Mex
WM/M2
M3
WMul
PF
F
D1
D2
EX
WB
EX
EX2
EX1
EX2
EX3
MMX instruction pipeline integrated in integer pipeline
Integer pipeline only
A6109-01

Instruction parsing is decoupled from the instruction decoding by means of an instruction FIFO, which is situated between the F and D1 (Decode 1) stages. The FIFO has slots for up to four instructions. This FIFO is transparent, it does not add additional latency when it is empty.

Every clock cycle, two instructions can be pushed into the instruction FIFO (depending on the availability of the code bytes, and on other factors such as prefixes). Instruction pairs are pulled out of the FIFO into the D1 stage. Since the average rate of instruction execution is less than two per clock, the FIFO is normally full. If the FIFO is full, then the FIFO can buffer a stall that may have occurred during instruction fetch and parsing. If this occurs, then that stall will not cause a stall in the execution stage of the pipe. If the FIFO is empty, then an execution stall may result from the pipeline being “starved” for instructions to execute. Also, if the FIFO contains only one instruction, then the instruction will not pair. Additionally, if an instruction is longer than 7 bytes, then only one instruction will be pushed into the FIFO. Figure 16-6 details the MMX pipeline on superscalar processors and the conditions where a stall may occur in the pipeline.

Table 16-1. Pipeline Stage Summary

Pipeline Stage	Abbreviation	Description
Prefetch	PF	Prefetches instructions
Fetch	F	The prefetched instruction bytes are passed into instructions. The prefixes are decoded and up to two instructions are pushed into the FIFO. Two MMX instructions can be pushed if each of the instructions are less than seven in bytes length.
Decode1	D1	Integer, floating-point and MMX instructions are decoded in the D1 pipe stage.
Decode2	D2	Source values are read.
Execution	E	The instruction is committed for execution.
MMX Execution	Mex	Execution clock for MMX instructions. ALU, shift, pack, and unpack instructions are executed and completed in this clock. First clock of multiply instructions. No stall conditions.
Write/Multiply2	WM/M_2	Single clock operations are written. Second stage of multiplier pipe. No stall conditions.
Multiply3	M_3	Third stage of multiplier pipe. No stall conditions.
Write of multiply	Wmul	Write of multiplier result. No stall conditions.

16.4.4 Instruction Issue

The rules of how MMX instructions get issued on the embedded Pentium processor with MMX technology are summarized as follows:

- Pairing of two MMX instructions can be performed.
- Pairing of one MMX instruction with an integer instruction can be performed.
- MMX instructions do not get paired with floating-point instructions.

16.4.4.1 Pairing Two MMX™ Instructions

The rules of how two MMX instructions can be paired are listed below:

- Two MMX instructions that both use the MMX shifter unit (pack, unpack and shift instructions) cannot pair since there is only one MMX shifter unit. Shift operations may be issued in either the u-pipe or the v-pipe but not in both in the same clock cycle.
- Two MMX instructions that both use the MMX multiplier unit (PMULL, PMULH, PMADD type instructions) cannot pair since there is only one MMX multiplier unit. Multiply operations may be issued in either the u-pipe or the v-pipe but not in both in the same clock cycle.
- MMX instructions that access either memory or the integer register file can be issued in the u-pipe only. Do not schedule these instructions to the v-pipe as they will wait and be issued in the next pair of instructions (and to the u-pipe).
- The MMX destination register of the u-pipe instruction should not match the source or destination register of the v-pipe instruction (dependency check).
- The EMMS instruction is not pairable.
- If either the CR0.TS or the CR0.EM bits are set, MMX instructions cannot go into the v-pipe.

16.4.4.2 Pairing an Integer Instruction in the U-pipe with an MMX Instruction in the V-pipe

The rules of how an integer instruction in the u-pipe is paired with an MMX instruction in the v-pipe are listed below:

- The MMX instruction cannot be the first MMX instruction following a floating-point instruction.
- The v-pipe MMX instruction does not access either memory or the integer register file.
- The u-pipe integer instruction is a pairable u-pipe integer instruction.

16.4.4.3 Pairing an MMX Instruction in the U-pipe with an Integer Instruction in the V-pipe

The rules of how an MMX instruction in the u-pipe is paired with an integer instruction in the v-pipe are listed below:

- The v-pipe instruction is a pairable integer v-pipe instruction.
- The u-pipe MMX instruction does not access either memory or the integer register file.

16.5 On-Chip Caches

The embedded Pentium processor (at 100/133/166 MHz) implements two internal caches for a total integrated cache size of 16 Kbytes: an 8-Kbyte data cache and a separate 8-Kbyte code cache. These caches are transparent to application software to maintain compatibility with previous Intel architecture generations. The embedded Pentium processor with MMX technology doubles the internal cache size to 32 Kbytes: a 16-Kbyte data cache and a separate 16-Kbyte code cache.

The data cache fully supports the MESI (modified/exclusive/shared/invalid) cache consistency protocol. The code cache is inherently write protected to prevent code from being inadvertently corrupted, and as a consequence supports a subset of the MESI protocol, the S (shared) and I (invalid) states.

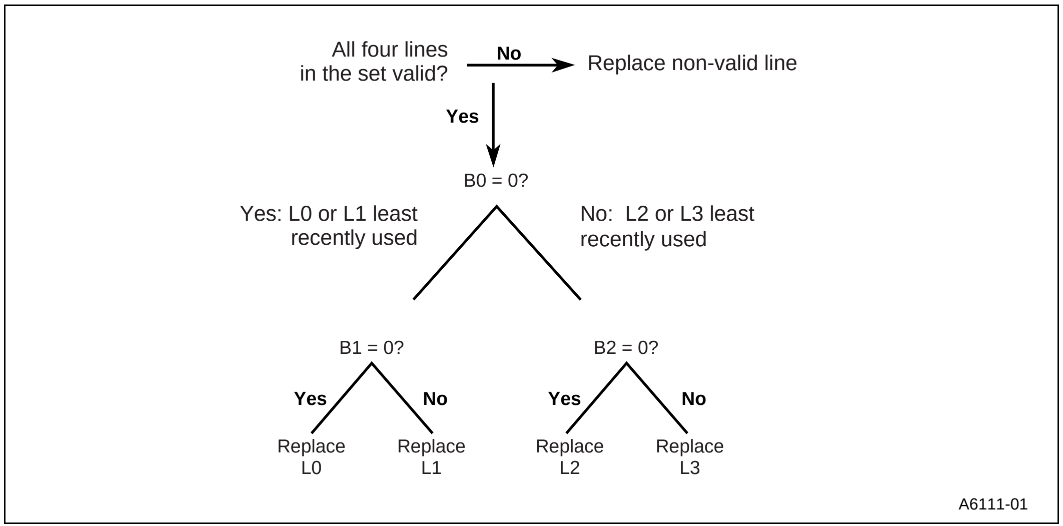
The caches have been designed for maximum flexibility and performance. The data cache is configurable as writeback or writethrough on a line-by-line basis. Memory areas can be defined as non-cacheable by software and external hardware. Cache writeback and invalidations can be initiated by hardware or software. Protocols for cache consistency and line replacement are implemented in hardware, easing system design.

16.5.1 Cache Organization

On the embedded Pentium processor, each cache is 8 Kbytes and is organized as a 2-way set associative cache. There are 128 sets in each cache; each set contains 2 lines (each line has its own tag address). Each cache line is 32 bytes wide. The embedded Pentium processor with MMX technology has two 16-Kbyte, 4-way set-associative caches the with a line length of 32 bytes.

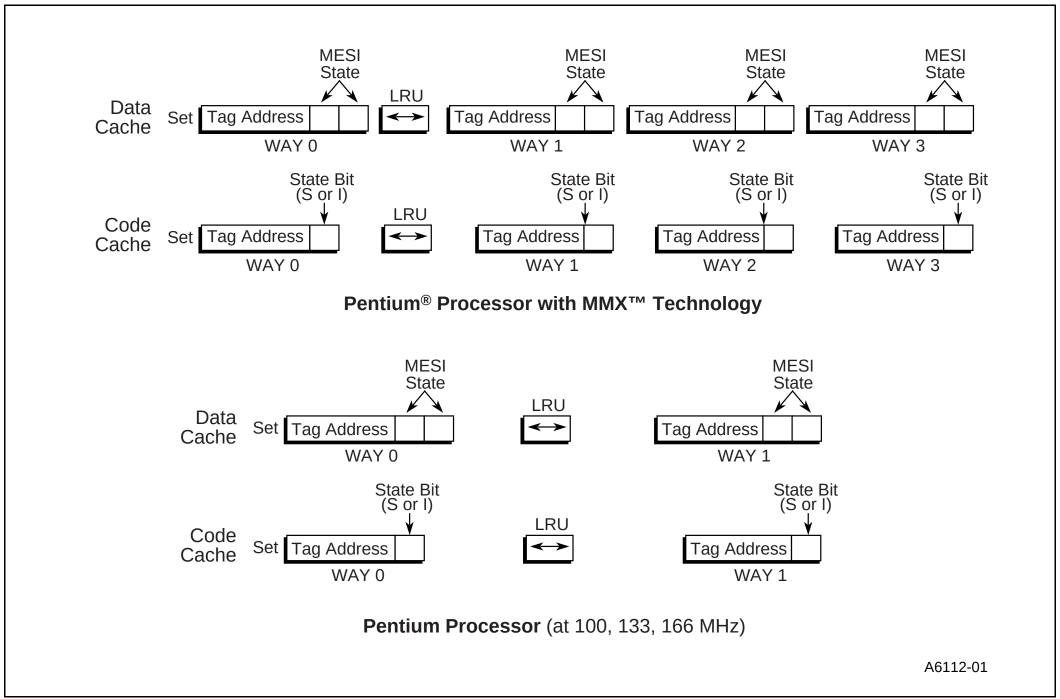
On the embedded Pentium processor, replacement in both the data and instruction caches is handled by the LRU mechanism, which requires one bit per set in each of the caches. The embedded Pentium processor with MMX technology uses a pseudo-LRU replacement algorithm that requires three bits per set in each of the caches. When a line must be replaced, the cache selects which of L0:L1 and L2:L3 was least recently used. Then the cache determines which of the two lines was least recently used and marks it for replacement. This decision tree is shown in Figure 16-7.

Figure 16-7. Pseudo-LRU Cache Replacement Strategy



The data cache consists of eight banks interleaved on 4-byte boundaries. The data cache can be accessed simultaneously from both pipes, as long as the references are to different cache banks. A conceptual diagram of the organization of the data and code caches is shown in Figure 16-8. The data cache supports the MESI writeback cache consistency protocol, which requires two state bits, while the code cache supports the S and I state only and therefore requires only one state bit.

Figure 16-8. Conceptual Organization of Code and Data Caches



16.5.2 Cache Structure

The instruction and data caches can be accessed simultaneously. The instruction cache can provide up to 32 bytes of raw opcodes and the data cache can provide data for two data references all in the same clock. This capability is implemented partially through the tag structure. The tags in the data cache are triple-ported. One of the ports is dedicated to snooping while the other two are used to lookup two independent addresses corresponding to data references from each of the pipelines. The instruction cache tags of the embedded Pentium processor (at 100/133/166 MHz) are also triple-ported. Again, one port is dedicated to support snooping and the other two ports facilitate split line accesses (simultaneously accessing upper half of one line and lower half of the next line). Note that the embedded Pentium processor with MMX technology does not support split line accesses to the code cache; its code cache tags are dual ported.

The storage array in the data cache is single ported but interleaved on 4-byte boundaries to be able to provide data for two simultaneous accesses to the same cache line.

Each of the caches are parity protected. In the instruction cache, there are parity bits on a quarter line basis and there is one parity bit for each tag. The data cache contains one parity bit for each tag and a parity bit per byte of data.

Each of the caches are accessed with physical addresses and each cache has its own TLB (translation lookaside buffer) to translate linear addresses to physical addresses. The TLBs associated with the instruction cache are single-ported whereas the data cache TLBs are fully dual-ported to be able to translate two independent linear addresses for two data references simultaneously. The tag and data arrays of the TLBs are parity protected with a parity bit associated with each of the tag and data entries in the TLBs.

The data cache of the embedded Pentium processor has a 4-way set associative, 64-entry TLB for 4-Kbyte pages and a separate 4-way set associative, 8-entry TLB to support 4-Mbyte pages. The code cache has one 4-way set associative, 32-entry TLB for 4-Kbyte pages and 4-Mbyte pages, which are cached in 4-Kbyte increments. Replacement in the TLBs is handled by a pseudo-LRU mechanism (similar to the Intel486 processor) that requires 3 bits per set. The embedded Pentium processor with MMX technology has a 64-entry fully associative data TLB and a 32-entry fully associative code TLB. Both TLBs can support 4-Kbyte pages as well as 4-Mbyte pages.

16.5.3 Cache Operating Modes

The operating modes of the caches are controlled by the CD (cache disable) and NW (not writethrough) bits in CR0. See Table 16-2 for a description of the modes. For normal operation and highest performance, these bits should both be cleared to “0.” The bits come out of RESET as CD = NW = 1.

When the L1 cache is disabled (CR0.NW and CR0.CD bits are both set to ‘1’) external snoops are accepted in a DP system and inhibited in a UP system. Note that when snoops are inhibited, address parity is not checked, and APCHK# will not be asserted for a corrupt address. When snoops are accepted, address parity is checked (and APCHK# will be asserted for corrupt addresses).

Table 16-2. Cache Operating Modes

CD	NW	Description
1	1	Read hits access the cache.
		Read misses do not cause linefills.
		Write hits update the cache, but do not access memory.
		Write hits cause Exclusive State lines to change to Modified State.
		Shared lines remain in the Shared state after write hits.
		Write misses access memory.
		Inquire and invalidation cycles do not affect the cache state or contents.
		This is the state after reset.
1	0	Read hits access the cache.
		Read misses do not cause linefills.
		Write hits update the cache.
		Writes to Shared lines and write misses update external memory.
		Writes to Shared lines can be changed to the Exclusive State under the control of the WB/WT# pin.
		Inquire cycles (and invalidations) are allowed.
0	1	GP(0)
0	0	Read hits access the cache.
		Read misses may cause linefills.
		These lines will enter the Exclusive or Shared state under the control of the WB/WT# pin.
		Write hits update the cache.
		Only writes to shared lines and write misses appear externally.
		Writes to Shared lines can be changed to the Exclusive State under the control of the WB/WT# pin.
		Inquire cycles (and invalidations) are allowed.

To completely disable the cache, the following two steps must be performed:

1. CD and NW must be set to 1.
2. The caches must be flushed.

If the cache is not flushed, cache hits on reads will still occur and data will be read from the cache. In addition, the cache must be flushed after being disabled to prevent any inconsistencies with memory.

16.5.4 Page Cacheability

Two bits for cache control, PWT and PCD are defined in the page table and page directory entries. The state of these bits are driven out on the PWT and PCD pins during memory access cycles. The PWT bit controls write policy for the second-level caches used with the embedded Pentium processor. Setting PWT to 1 defines a writethrough policy for the current page, while clearing PWT to 0 defines a writeback policy for the current page.

The PCD bit controls cacheability on a page-by-page basis. The PCD bit is internally ANDed with the KEN# signal to control cacheability on a cycle-by-cycle basis. PCD = 0 enables cacheing, while PCD = 1 disables it. Cache linefills are enabled when PCD = 0 and KEN# = 0.

16.5.4.1 PCD and PWT Generation

The value driven on PCD is a function of the PWT bits in CR3, the page directory pointer, the page directory entry and the page table entry, and the CD and PG bits in CR0.

The value driven on PWT is a function of the PCD bits in CR3, the page directory pointer, the page directory entry and the page table entry, and the PG bit in CR0 (CR0.CD does not affect PWT).

CR0.CD = 1

If cacheing is disabled, the PCD pin is always driven high. CR0.CD does not affect the PWT pin.

CR0.PG = 0

If paging is disabled, the PWT pin is forced low and the PCD pin reflects the CR0.CD. The PCD and PWT bits in CR3 are assumed 0 during the caching process.

CR0.CD = 0, PG = 1, normal operation

The PCD and PWT bits from the last entry (can be either PDE or PTE, depends on 4 Mbyte or 4 Kbyte mode) are cached in the TLB and are driven anytime the page mapped by the TLB entry is referenced.

CR0.CD = 0, PG = 1, during TLB Refresh

During TLB refresh cycles when the PDE and PTE entries are read, the PWT and PCD bits are obtained as shown in Table 16-3 and Table 16-4.

Table 16-3. 32-Bits/4-Kbyte Pages

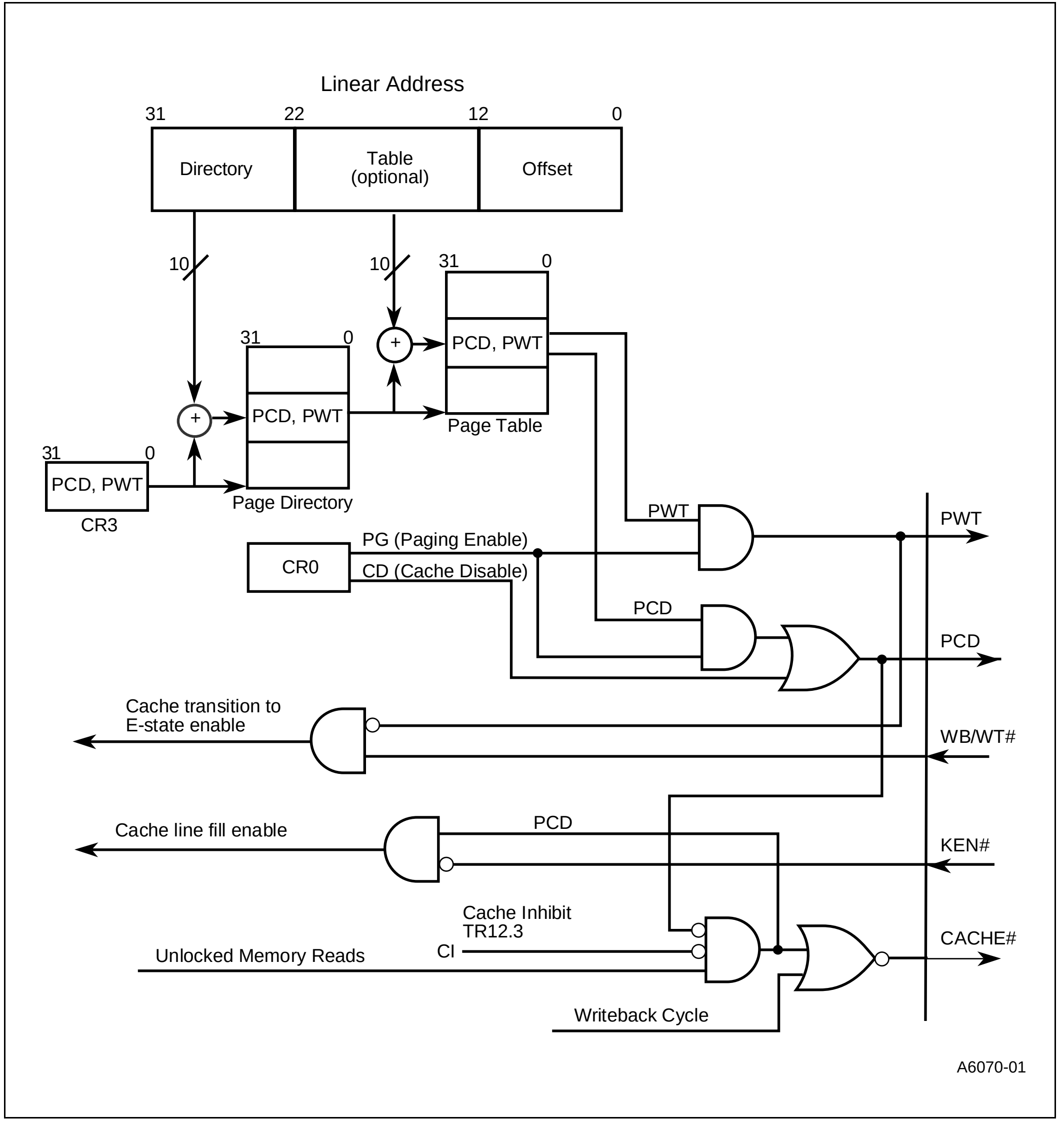
PCD/PWT Taken From	During Accesses To
CR3	PDE
PDE	PTE
PTE	All other paged mem references

Table 16-4. 32-Bits/4-Mbyte Pages

PCD/PWT Taken From	During Accesses To
CR3	PDE
PDE	All other paged mem references

Figure 16-9 shows how PCD and PWT are generated.

Figure 16-9. PCD and PWT Generation



16.5.5 Inquire Cycles

Inquire cycles are initiated by the system to determine if a line is present in the code or data cache, and what state the line is in. This manual refers to inquire cycles and snoop cycles interchangeably.

Inquire cycles are driven to the processor when a bus master other than the processor initiates a read or write bus cycle. Inquire cycles are driven to the processor when the bus master initiates a read to determine if the processor data cache contains the latest information. If the snooped line is in the processor data cache in the modified state, the processor has the most recent information and must schedule a writeback of the data. Inquire cycles are driven to the processor when the other bus master initiates a write to determine if the processor code or data cache contains the snooped line and to invalidate the line if it is present. Inquire cycles are described in detail in Chapter 19, “Bus Functional Description.”

16.5.6 Cache Flushing

The on-chip cache can be flushed by external hardware or by software instructions.

Flushing the cache through hardware is accomplished by driving the FLUSH# pin low. This causes the cache to write back all modified lines in the data cache and mark the state bits for both caches invalid. The Flush Acknowledge special cycle is driven by the processor when all writebacks and invalidations are complete.

The INVD and WBINVD instructions cause the on-chip caches to be invalidated also. WBINVD causes the modified lines in the internal data cache to be written back, and all lines in both caches to be marked invalid. After execution of the WBINVD instruction, the Writeback and Flush special cycles are driven to indicate to any external cache that it should write back and invalidate its contents.

INVD causes all lines in both caches to be invalidated. Modified lines in the data cache are not written back. The Flush special cycle is driven after the INVD instruction is executed to indicate to any external cache that it should invalidate its contents. Care should be taken when using the INVD instruction that cache consistency problems are not created.

Note that the implementation of the INVD and WBINVD instructions are processor dependent. Future processor generations may implement these instructions differently.

16.5.7 Data Cache Consistency Protocol (MESI Protocol)

The embedded Pentium processor Cache Consistency Protocol is a set of rules by which states are assigned to cached entries (lines). The rules apply for memory read/write cycles only. I/O and special cycles are not run through the data cache.

Every line in the data cache is assigned a state dependent on both processor generated activities and activities generated by other bus masters (snooping). The embedded Pentium processor Data Cache Protocol consists of four states that define whether a line is valid (HIT/MISS), if it is available in other caches, and if it has been MODIFIED. The four states are the M (Modified), E (Exclusive), S (Shared) and the I (Invalid) states and the protocol is referred to as the MESI protocol. A definition of the states is given below:

M - Modified:	An M-state line is available in only one cache and it is also MODIFIED (different from main memory). An M-state line can be accessed (read/written to) without sending a cycle out on the bus.
E - Exclusive:	An E-state line is also available in only one cache in the system, but the line is not MODIFIED (i.e., it is the same as main memory). An E-state line can be accessed (read/written to) without generating a bus cycle. A write to an E-state line causes the line to become MODIFIED.
S - Shared:	This state indicates that the line is potentially shared with other caches (i.e., the same line may exist in more than one cache). A read to an S-state line does not generate bus activity, but a write to a SHARED line generates a write-through cycle on the bus. The write-through cycle may invalidate this line in other caches. A write to an S-state line updates the cache.
I - Invalid:	This state indicates that the line is not available in the cache. A read to this line will be a MISS and may cause the processor to execute a LINE FILL (fetch the whole line into the cache from main memory). A write to an INVALID line causes the processor to execute a write-through cycle on the bus.

16.5.7.1 State Transition Tables

Lines cached in the processor can change state because of processor-generated activity or as a result of activity on the processor bus generated by other bus masters (snooping). State transitions happen because of processor-generated transactions (memory reads/writes) and by a set of external input signals and internally generated variables. The processor also drives certain pins as a consequence of the Cache Consistency Protocol.

16.5.7.2 Read Cycle

Table 16-5 shows the state transitions for lines in the data cache during unlocked read cycles.

Table 16-5. Data Cache State Transitions for UNLOCKED Processor Initiated Read Cycles†

Present State	Pin Activity	Next State	Description
M	n/a	M	Read hit; data is provided to processor core by cache. No bus cycle is generated.
E	n/a	E	Read hit; data is provided to processor core by cache. No bus cycle is generated.
S	n/a	S	Read hit; data is provided to the processor by the cache. No bus cycle is generated.
I	CACHE# low AND KEN# low AND WB/WT# high AND PWT low	E	Data item does not exist in cache (MISS). A bus cycle (read) will be generated. This state transition will happen if WB/WT# is sampled high with first BRDY# or NA#.
I	CACHE# low AND KEN# low AND (WB/WT# low OR PWT high)	S	Same as previous read miss case except that WB/WT# is sampled low with first BRDY# or NA#.
I	CACHE# high OR KEN# high	I	KEN# pin inactive; the line is not intended to be cached in the embedded Pentium processor.

† Locked accesses to the data cache cause the accessed line to transition to the Invalid state.

Note the transition from I to E or S states (based on WB/WT#) happens only if KEN# is sampled low with the first of BRDY# or NA#, and the cycle is transformed into a LINE FILL cycle. If KEN# is sampled high, the line is not cached and remains in the I state.

16.5.7.3 Write Cycle

The state transitions of data cache lines during processor-generated write cycles are illustrated in Table 16-6. Writes to SHARED lines in the data cache are always sent out on the bus along with updating the cache with the write item. The status of the PWT and WB/WT# pins during these write cycles on the bus determines the state transitions in the data cache during writes to S-state lines.

A write to a SHARED line in the data cache generates a write cycle on the processor bus to update memory and/or invalidate the contents of other caches. If the PWT pin is driven high when the write cycle is run on the bus, the line is be updated and will stay in the S-state regardless of the status of the WB/WT# pin that is sampled with the first BRDY# or NA#. If PWT is driven low, the status of the WB/WT# pin sampled along with the first BRDY# or NA# for the write cycle determines which state (E:S) the line transitions to.

The state transition from S to E is the only transition in which the data and the status bits are not updated at the same time. The data is updated when the write is written to the processor write buffers. The state transition does not occur until the write has completed on the bus (BRDY# has been returned). Writes to the line after the transition to the E-state do not generate bus cycles. However, it is possible that writes to the same line that were buffered or in the pipeline before the transition to the E-state generate bus cycles after the transition to E-state.

An inactive EWBE# input stalls subsequent writes to an E- or an M-state line. All subsequent writes to E- or M-state lines are held off until EWBE# is returned active.

Table 16-6. Data Cache State Transitions for Processor Initiated Write Cycles

Present State	Pin Activity	Next State	Description
M	n/a	M	Write hit; update data cache. No bus cycle generated to update memory.
E	n/a	M	Write hit; update cache only. No bus cycle generated; line is now MODIFIED.
S	PWT low AND WB/WT# high	E	Write hit; data cache updated with write data item. A write-through cycle is generated on bus to update memory and/or invalidate contents of other caches. The state transition occurs after the writethrough cycle completes on the bus (with the last BRDY#).
S	PWT low AND WB/WT# low	S	Same as above case of write to S-state line except that WB/WT# is sampled low.
S	PWT high	S	Same as above cases of writes to S state lines except that this is a write hit to a line in a writethrough page; status of WB/WT# pin is ignored.
I	n/a	I	Write MISS; a writethrough cycle is generated on the bus to update external memory. No allocation done.

NOTE: Memory writes are buffered while I/O writes are not. There is no guarantee of synchronization between completion of memory writes on the bus and instruction execution after the write. A serializing instruction needs to be executed to synchronize writes with the next instruction if necessary.

16.5.7.4 Inquire Cycles (Snooping)

The purpose of inquire cycles is to check whether the address being presented is contained within the caches in the embedded Pentium processor. Inquire cycles may be initiated with or without an INVALIDATION request (INV = 1 or 0). An inquire cycle is run through the data and code caches through a dedicated snoop port to determine if the address is in one of the processor caches. If the address is in a processor cache, the HIT# pin is asserted. If the address hits a modified line in the data cache, the HITM# pin is also asserted and the modified line is then written back onto the bus.

The state transition tables for inquire cycles are given below:

Table 16-7. Cache State Transitions During Inquiry Cycles

Present State	Next State INV=1	Next State INV=0	Description
M	I	S	Snoop hit to a MODIFIED line indicated by HIT# and HITM# pins low. embedded Pentium® processor schedules the writing back of the modified line to memory.
E	I	S	Snoop hit indicated by HIT# pin low; no bus cycle generated.
S	I	S	Snoop hit indicated by HIT# pin low; no bus cycle generated.
I	I	I	Address not in cache; HIT# pin high.

16.5.7.5 Code Cache Consistency Protocol

The processor code cache follows a subset of the MESI protocol. Accesses to the code cache are either a Hit (Shared) or a Miss (Invalid).

In the case of a read hit, the cycle is serviced internally to the processor and no bus activity is generated. In the case of a read miss, the read is sent to the external bus and may be converted to a linefill.

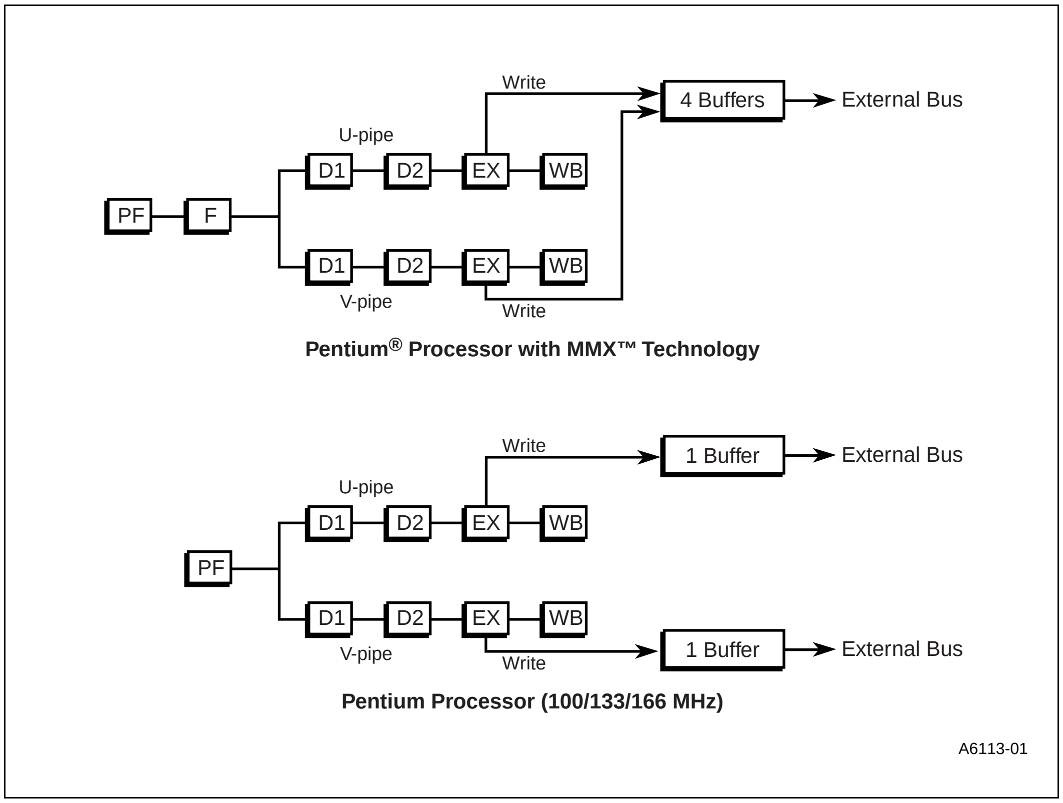
Lines are never overwritten in the code cache. Writes generated by the processor are snooped by the code cache. If the snoop is a hit in the code cache, the line is invalidated. If there is a miss, the code cache is not affected.

16.6 Write Buffers and Memory Ordering

The embedded Pentium processor has two write buffers, one corresponding to each of the pipelines, to enhance the performance of consecutive writes to memory. These write buffers are one quadword wide (64-bits) and can be filled simultaneously in one clock e.g., by two simultaneous write misses in the two instruction pipelines. Writes in these buffers are driven out on the external bus in the order they were generated by the processor core. No reads (as a result of cache miss) are reordered around previously generated writes sitting in the write buffers. The implication of this is that the write buffers will be flushed or emptied before a subsequent bus cycle is run on the external bus (unless BOFF# is asserted and a writeback cycle becomes pending, see "Linefill and Writeback Buffers" on page 16-207).

The embedded Pentium processor with MMX technology has four write buffers that can be used by either the u-pipe or v-pipe. Posting writes to these buffers enables the pipe to continue advancing when consecutive writes to memory occur. The writes will be executed on the bus as soon as it is free, in FIFO order. Reads cannot bypass writes posted in these buffers.

Figure 16-10. Embedded Pentium® Processor Write Buffer Implementation



The embedded Pentium processor supports strong write ordering only. That is, writes generated by the embedded Pentium processor are driven to the bus or updated in the cache in the order in which they occur. The embedded Pentium processor does not write to E or M-state lines in the data cache if there is a write in either write buffer, if a write cycle is running on the bus, or if EWBE# is inactive.

Note that only memory writes are buffered and I/O writes are not. There is no guarantee of synchronization between completion of memory writes on the bus and instruction execution after the write. The OUT instruction or a serializing instruction needs to be executed to synchronize writes with the next instruction. Refer to “Serializing Operations” on page 16-206 for more information.

No re-ordering of read cycles occurs on the embedded Pentium processor. Specifically, the write buffers are flushed before the IN instruction is executed.

16.6.1 External Event Synchronization

When the system changes the value of NMI, INTR, FLUSH#, SMI# or INIT as the result of executing an OUT instruction, these inputs must be at a valid state three clocks before BRDY# is returned to ensure that the new value will be recognized before the next instruction is executed.

Note that if an OUT instruction is used to modify A20M#, this will not affect previously prefetched instructions. A serializing instruction must be executed to guarantee recognition of A20M# before a specific instruction.

16.6.2 Serializing Operations

After executing certain instructions, the embedded Pentium processor serializes instruction execution. This means that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed. The prefetch queue is flushed as a result of serializing operations.

The embedded Pentium processor serializes instruction execution after executing one of the following instructions: MOV to Debug Register, MOV to Control Register, INVD, INVLPG, IRET, IRETD, LGDT, LLDT, LIDT, LTR, WBINVD, CPUID, RSM and WRMSR.

The CPUID instruction can be executed at any privilege level to serialize instruction execution.

When the processor serializes instruction execution, it ensures that it has completed any modifications to memory, including flushing any internally buffered stores; it then waits for the EWBE# pin to go active before fetching and executing the next instruction. Systems may use the EWBE# pin to indicate that a store is pending externally. In this manner, a system designer may ensure that all externally pending stores complete before the processor begins to fetch and execute the next instruction.

The processor does not generally writeback the contents of modified data in its data cache to external memory when it serializes instruction execution. Software can force modified data to be written back by executing the WBINVD instruction.

Whenever an instruction is executed to enable/disable paging (that is, change the PG bit of CR0), this instruction must be followed with a jump. The instruction at the target of the branch is fetched with the new value of PG (i.e., paging enabled/disabled); however, the jump instruction itself is fetched with the previous value of PG. Intel386™, Intel486 and embedded Pentium processors have slightly different requirements to enable and disable paging. In all other respects, an MOV to CR0 that changes PG is serializing. Any MOV to CR0 that does not change PG is completely serializing.

Whenever an instruction is executed to change the contents of CR3 while paging is enabled, the next instruction is fetched using the translation tables that correspond to the new value of CR3. Therefore the next instruction and the sequentially following instructions should have a mapping based upon the new value of CR3.

The embedded Pentium processor implements branch-prediction techniques to improve performance by prefetching the destination of a branch instruction before the branch instruction is executed. Consequently, instruction execution is not generally serialized when a branch instruction is executed.

Although the I/O instructions are not “serializing” because the processor does not wait for these instructions to complete before it prefetches the next instruction, they do have the following properties that cause them to function in a manner that is identical to previous generations. I/O reads are not re-ordered within the processor; they wait for all internally pending stores to complete. Note that the embedded Pentium processor does not sample the EWBE# pin during reads. If necessary, external hardware must ensure that externally pending stores are complete before returning BRDY#. This is the same requirement that exists on Intel386 and Intel486 processor systems. The OUT and OUTS instructions are also not “serializing,” as they do not stop the prefetcher. They do, however, ensure that all internally buffered stores have completed, that EWBE# has been sampled active indicating that all externally pending stores have completed and that the I/O write has completed before they begin to execute the next instruction. Note that unlike the Intel486 processor, it is not necessary for external hardware to ensure that externally pending stores are complete before returning BRDY#.

On the embedded Pentium processor with MMX technology, serializing instructions require an additional clock to complete compared to the embedded Pentium processor due to the additional pipeline stage.

16.6.3 Linefill and Writeback Buffers

In addition to the write buffers corresponding to each of the internal pipelines, the embedded Pentium processor has three writeback buffers. Each of the writeback buffers are 1 deep and 32-bytes (1 line) wide.

A dedicated replacement writeback buffer stores writebacks caused by linefills that replace modified lines in the data cache. In addition, an external snoop writeback buffer stores writebacks caused by a inquire cycles that hit modified lines in the data cache. Finally, an internal snoop writeback buffer stores writebacks caused by internal snoop cycles that hit modified lines in the data cache. Internal and external snoops are discussed in detail in “Cache Consistency Cycles (Inquire Cycles)” on page 19-353. Write cycles are driven to the bus with the following priority:

1. Contents of external snoop writeback buffer
2. Contents of internal snoop writeback buffer
3. Contents of replacement writeback buffer
4. Contents of write buffers.

Note that the contents of the write buffer that was written into first are driven to the bus first. If both write buffers were written to in the same clock, the contents of the u-pipe buffer is written out first. In the embedded Pentium processor with MMX technology, the write buffers are written in order, even though there is no u-pipe buffer and no v-pipe buffer.

The embedded Pentium processor implements two linefill buffers, one for the data cache and one for the code cache. As information (data or code) is returned to the processor for a cache linefill, it is written into the linefill buffer. After the entire line has been returned to the processor it is transferred to the cache. Note that the processor requests the needed information first and uses that information as soon as it is returned. The processor does not wait for the linefill to complete before using the requested information.

If a line fill causes a modified line in the data cache to be replaced, the replaced line remains in the cache until the linefill is complete. After the linefill is complete, the line being replaced is moved into the replacement writeback buffer and the new linefill is moved into the cache.

16.7 External Interrupt Considerations

The embedded Pentium processor recognizes the following external interrupts: BUSCHK#, R/S#, FLUSH#, SMI#, INIT, NMI, INTR and STPCLK#. These interrupts are recognized at instruction boundaries. The instruction boundary is the first clock in the execution stage of the instruction pipeline. This means that before an instruction is executed, the processor checks to see if any interrupts are pending. If an interrupt is pending, the processor flushes the instruction pipeline and then services the interrupt.

The embedded Pentium processor interrupt priority scheme is shown in Table 16-8.

Table 16-8. Embedded Pentium® Processor Interrupt Priority Scheme

Priority Level	ITR = 0 (default)	ITR = 1
1	Breakpoint (INT 3)	Breakpoint (INT 3)
2	BUSCHK#	BUSCHK#
3	Debug Traps (INT 1)	FLUSH#
4	R/S#	SMI#
5	FLUSH#	Debug Traps (INT 1)
6	SMI#	R/S#
7	INIT	INIT
8	NMI	NMI
9	INTR	INTR
10	Floating-Point Error	Floating-Point Error
11	STPCLK#	STPCLK#
12	Faults on Next Instruction	Faults on Next Instruction

NOTE: ITR is bit 9 of the TR12 register.

16.8 Introduction to Dual Processor Mode

Symmetric dual processing in a system is supported with two embedded Pentium processors sharing a single second-level cache. The processors must be of the same type, either two embedded Pentium processors or two embedded Pentium processor with MMX technology. The two processors appear to the system as a single processor. Multiprocessor operating systems properly schedule computing tasks between the two processors. This scheduling of tasks is transparent to software applications and the end-user. Logic built into the processors support a “glueless” interface for easy system design. Through a private bus, the two processors arbitrate for the external bus and maintain cache coherency.

In this manual, in order to distinguish between two processors in dual processing mode, one processor is designated as the Primary processor and the other as the Dual processor. Note that this is a different concept than that of “master” and “checker” processors.

The Dual processor is a configuration option of the embedded Pentium processor. The Dual processor must operate at the same bus and core frequency and bus/core ratio as the Primary processor.

The Primary and Dual processors include logic to maintain cache consistency between the processors and to arbitrate for the common bus. The cache consistency and bus arbitration activity causes the dual processor pair to issue extra bus cycles that does not appear in a embedded Pentium processor uniprocessor system.

Chapter 17, “Microprocessor Initialization and Configuration,” describes in detail how the DP bootup, cache consistency, and bus arbitration mechanisms operate. In order to operate properly in dual processing mode, the Primary and Dual processors require a private APIC, cache consistency, and bus arbitration interfaces, as well as a multiprocessing-ready operating system.

The dual processor interface allows the Dual processor to be added for a substantial increase in system performance. The interface allows the Primary and Dual processor to operate in a coherent manner that is transparent to the system.

The memory subsystem transparency was the primary goal of the cache coherency and bus arbitration mechanisms.

16.8.1 Dual Processing Terminology

This section defines some terms used in the following discussions.

Symmetric Multi-Processing:	Two or more processors operating with equal priorities in a system. No individual processor is a master, and none is a slave.
DP or Dual Processing:	The Primary and Dual processor operating symmetrically in a system sharing a second-level cache.
MRM or Most Recent Master:	The processor (either the Primary or Dual) that currently owns the processor address bus. When interprocessor pipelining, this is the processor which last issued an ADS#.
LRM or Least Recent Master:	The processor (either the Primary or Dual) that does not own the address bus. The LRM automatically snoops every ADS# from the MRM processor in order to maintain level-one cache coherency.
Primary Processor:	The embedded Pentium processor when CPUTYP = V_{SS} (or left floating).
Dual Processor:	The embedded Pentium processor when CPUTYP = V_{CC}.

16.8.2 Dual Processing Overview

The Primary and Dual processor both have logic built-in to support “glueless” dual-processing behind a shared L2 cache. Through a set of private handshake signals, the Primary and Dual processors arbitrate for the external bus and maintain cache coherency between themselves. The bus arbitration and cache coherency mechanisms allow the Primary and Dual processors to look like a single embedded Pentium processor to the external bus.

The Primary and Dual processors implement a fair arbitration scheme. If the Least Recent Master (LRM) requests the bus from the Most Recent Master (MRM), the bus is granted. The embedded Pentium processor arbitration scheme provides no penalty to switch from one master to the next. If pipelining is used, the two processors pipeline into and out of each other’s cycles according to the embedded Pentium processor specification.

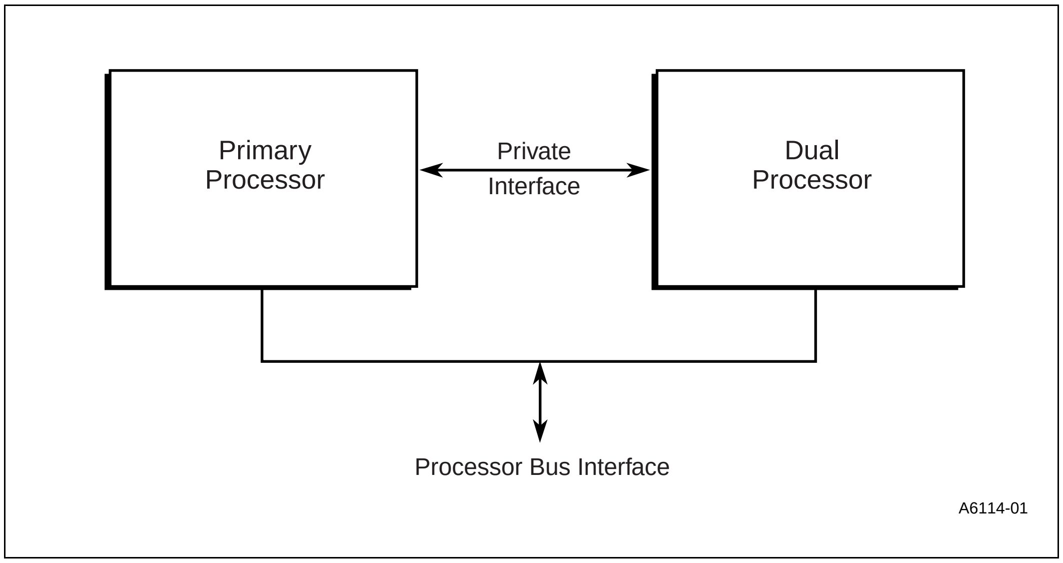
Cache coherency is maintained between the two processors by snooping on every bus access. The LRM must snoop with every ADS# assertion of the MRM. Internal cache states are maintained accordingly. If an access hits a modified line, a writeback is scheduled as the next cycle, in accordance with the embedded Pentium processor specification.

Using the Dual processor may require special design considerations. Refer to Chapter 18, “Hardware Interface” for more details.

16.8.2.1 Conceptual Overview

Figure 16-11 is a block diagram of a two processor system.

Figure 16-11. Dual Processors



The dual processor pair appears to the system bus as a single, unified processor. The operation is identical to a uni-processor embedded Pentium processor, except as noted in "Summary of Dual Processing Bus Cycles" on page 19-363. The interface shields the system designer from the cache consistency and arbitration mechanisms that are necessary for dual processor operation.

Both the Primary and Dual processors contain local APIC modules. The system designer is recommended to supply an I/O APIC or other multiprocessing interrupt controller in the chip set that interfaces to the local APIC blocks over a three-wire bus. The APIC allows directed interrupts as well as inter-processor interrupts.

The Primary and Dual processors, when operating in dual processing mode, require the local APIC modules to be hardware enabled in order to complete the bootup handshake protocol. This method is used to "wake up" the Dual processor at an address other than the normal Intel architecture high memory execution address. On bootup, if the Primary processor detects that a Dual processor is present, the dual processor cache consistency and arbitration mechanisms are automatically enabled. The bootup handshake process is supported in a protocol that is included in the embedded Pentium processor. See Chapter 17, "Microprocessor Initialization and Configuration," for more details on the APIC.

16.8.2.2 Arbitration Overview

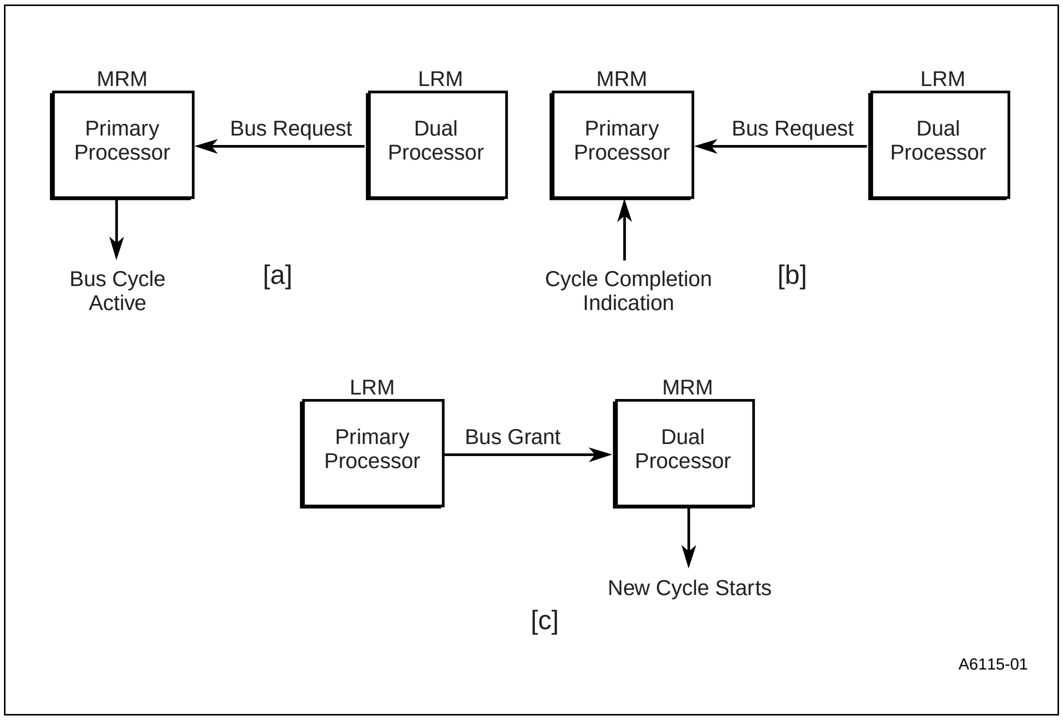
In the dual processor configuration, a single-system bus provides the processors access to the external system. This bus is a single, shared resource.

The dual processor pair must arbitrate for use of the system bus as requests are generated. The processors implement a fair arbitration mechanism.

When the LRM processor needs to run a cycle on the bus it submits a request for bus ownership to the MRM. The MRM processor grants the LRM processor bus ownership as soon as all outstanding bus requests have finished on the processor bus. The LRM processor assumes the MRM state, and the processor that was just the MRM, becomes the LRM. Figure 16-12 further illustrates this point:

Diagram (a) of Figure 16-12 shows a configuration where the Primary processor is in the MRM state and the Dual processor is in the LRM state. The Primary processor is running a cycle on the system bus when it receives a bus request from the Dual processor. In diagram (b) of Figure 16-12 the MRM (still the Primary processor) has received an indication that the bus request has finished. The bus ownership has transferred in diagram (c) of Figure 16-12, where the Dual processor is now the MRM. At this point, the Dual processor starts a bus transaction and continues to own the bus until the LRM requests the bus.

Figure 16-12. Dual Processor Arbitration Mechanism



16.8.2.3 Cache Coherency Overview

The Primary and Dual processors both contain separate code and data caches. The data cache uses the MESI protocol to enforce cache consistency. A line in the data cache can be in the Modified, Exclusive, Shared or Invalid state, whereas a line in the instruction cache can be either in the valid or invalid state.

A situation can arise where the Primary and Dual processors are operating in dual processor mode with shared code or data. The first-level caches attempt to cache this code and data whenever possible (as indicated by the page cacheability bits and the cacheability pins). The private cache coherency mechanism guarantees data consistency across the processors. If any data is cached in one of the processors, and the other processor attempts to access the data, the processor containing the data notifies the requesting processor that it has cached the data. The state of the cache line in the processor containing the data changes depending on the current state and the type of request the other processor has made.

In some cases, the data returned by the system is ignored. This constraint is placed on the dual processor cache consistency mechanism so that the dual processor pair looks like a single processor to the system bus. However, in general, bus accesses are minimized to efficiently use the available bus bandwidth.

The basic coherency mechanism requires the processor that is in the LRM state to snoop all MRM bus activity. The MRM processor running a bus cycle watches the LRM processor for an indication that the data is contained in the LRM cache. The following diagrams illustrate the basic coherency mechanism. These figures show an example in which the Primary processor (the MRM) is performing a cache line fill of data. The data requested by the Primary processor is cached by the Dual processor (the LRM), and is in the modified state.

In diagram (a) of Figure 16-13, the Primary processor has already negotiated with the Dual processor for use of the system bus and has started a cycle. As the Primary processor starts running the cycle on the system bus, the Dual processor snoops the transaction. The key for the start of the snoop sequence for the LRM processor is an assertion of ADS# by the MRM processor.

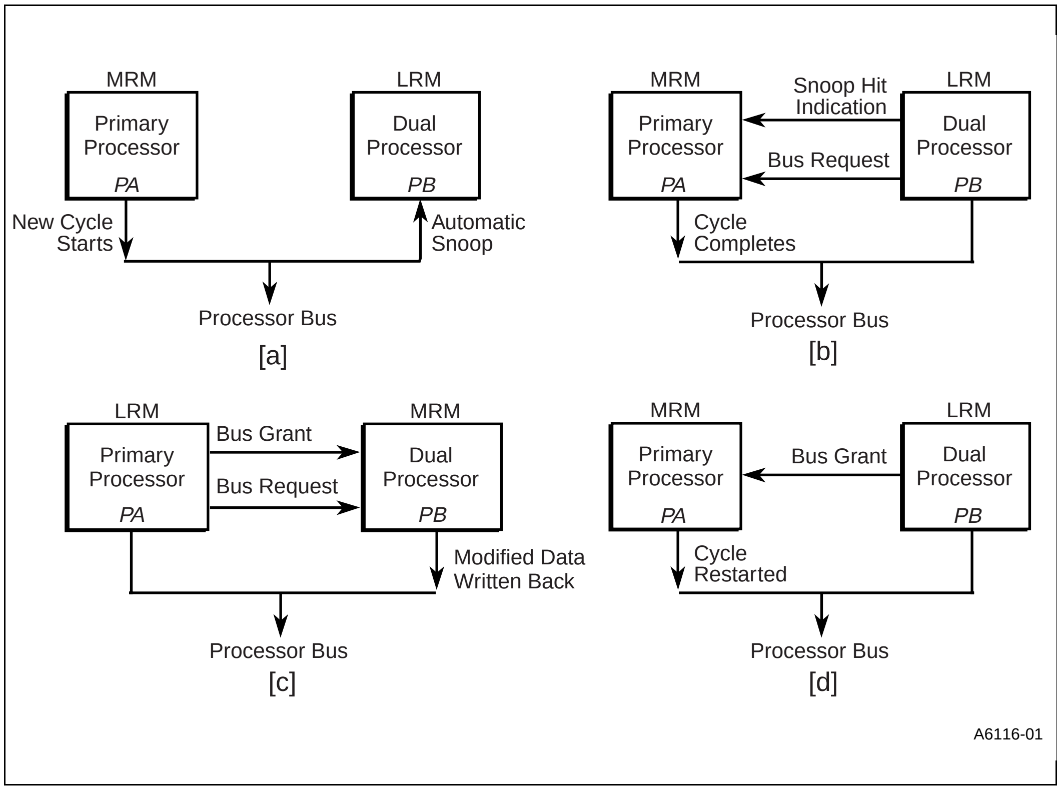
Diagram (b) of Figure 16-13 shows the Dual processor indicating to the Primary processor that the requested data is cached and modified in the Dual processor cache. The snoop notification mechanism uses a dedicated, two-signal interface that is private to the dual processor pair. At the same time that the Dual processor indicates that the transaction is contained as Modified in the its cache, the Dual processor requests the bus from the Primary processor (still the MRM). The MRM processor continues with the transaction that is outstanding on the bus, but ignores the data returned by the system bus.

After the Dual processor notifies the Primary processor that the requested data is modified in the Dual processor cache, the Dual processor waits for the bus transaction to complete. At this point, the LRM/MRM state will toggle, with the Primary processor becoming the LRM processor and the Dual processor becoming the MRM processor. This sequence of events is shown in diagram (c) of Figure 16-13.

Diagram (c) of Figure 16-13 also shows the Dual processor writing the data back on the system bus. The write back cycle looks like a normal cache line replacement to the system bus. The final state of the line in the Dual processor is determined by the value of the W/R# pin as sampled during the ADS# assertion by the Primary processor.

Finally, diagram (d) of Figure 16-13 shows the Primary processor re-running the bus transaction that started the entire sequence. The requested data is returned by the system as a normal line fill request without intervention from the LRM processor.

Figure 16-13. Dual Processor L1 Cache Consistency



16.9 APIC Interrupt Controller

The embedded Pentium processor contains implementations of the Advanced Programmable Interrupt Controller architecture. These implementations are capable of supporting a multiprocessing interrupt scheme with an external APIC-compatible controller.

The Advanced Programmable Interrupt Controller (APIC) is an on-chip interrupt controller that supports multiprocessing. In a uniprocessor system, the APIC may be used as the sole system interrupt controller, or may be disabled and bypassed completely.

In a multiprocessor system, the APIC operates with an additional and external I/O APIC system interrupt controller. The dual-processor configuration requires that the APIC be hardware enabled. The APICs of the Primary and Dual processors are used in the bootup procedure to communicate start-up information.

Note: The APIC is not hardware compatible with the 82489DX.

On the embedded Pentium processor, the APIC uses 3 pins: PICCLK, PICD0, and PICD1. PICCLK is the APIC bus clock while PICD0-PICD1 form the two-wire communication bus.

To use the 8259A interrupt controller, or to completely bypass it, the APIC may be disabled using the APICEN pin. You must use the local APICs when using the dual-processor component.

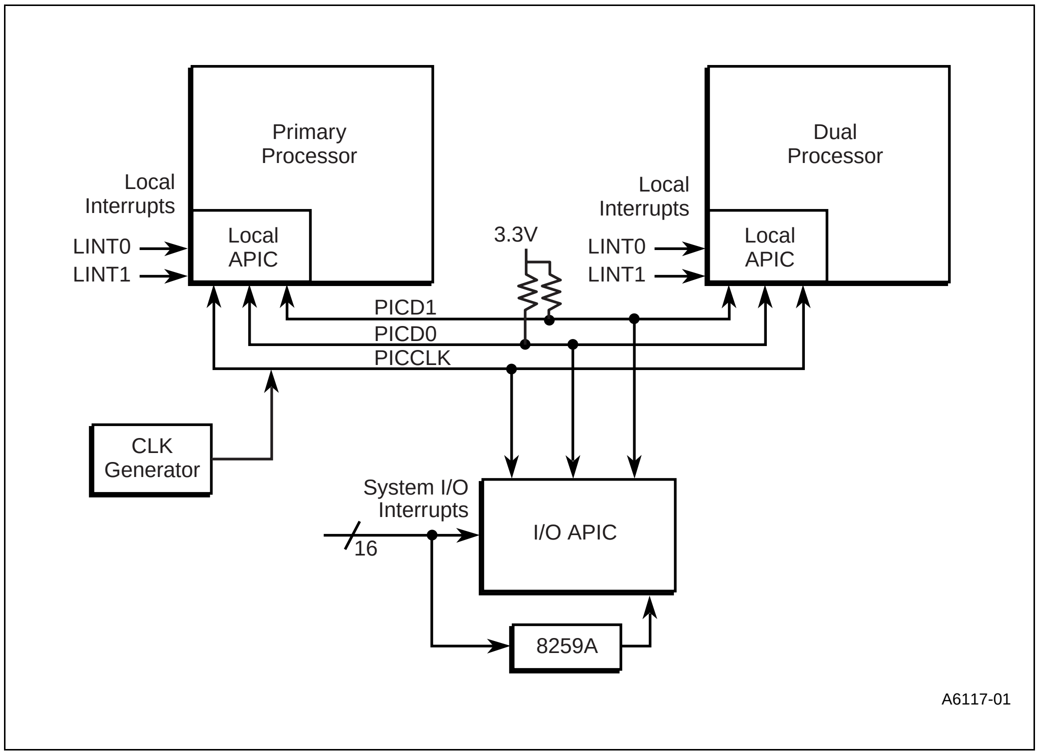
The main features of the APIC architecture include:

- Multiprocessor interrupt management (static and dynamic symmetric interrupt distribution across all processors)
- Dynamic interrupt distribution that includes routing interrupts to the lowest-priority processor
- Inter-processor interrupt support
- Edge or level triggered interrupt programmability
- Various naming/addressing schemes
- System-wide processor control functions related to NMI, INIT, and SMI (see Chapter 24 for APIC handling of SMI)
- 8259A compatibility by becoming virtually transparent with regard to an externally connected 8259A style controller, making the 8259A visible to software
- A 32-bit wide counter used as a timer to generate time slice interrupts local to that processor.

The AC timings of the embedded Pentium processor APIC are described in Chapter 7. Note that although there are minor software differences from the 82489DX, programming to the integrated APIC model ensures compatibility with the external 82489DX. For additional APIC programming information, refer to the *MultiProcessor Specification* (order number 242016).

In a dual-processor configuration, the local APIC may be used with an additional device similar to the I/O APIC. The I/O APIC is a device that captures all system interrupts and directs them to the appropriate processors via various programmable distribution schemes. An external device provides the APIC system clock. Interrupts that are local to each processor go through the APIC on each chip. A system example is shown in Figure 16-14.

Figure 16-14. APIC System Configuration

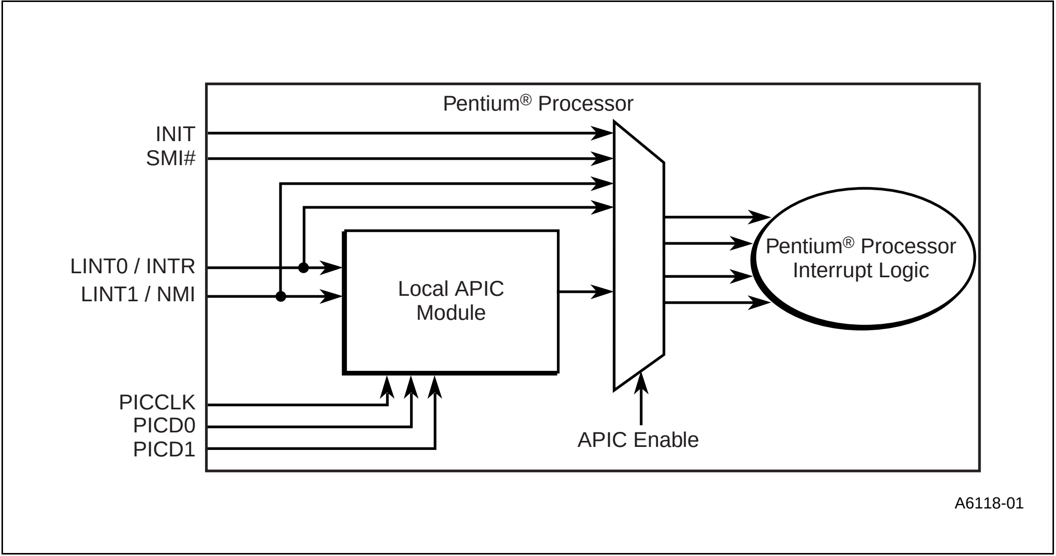


The APIC devices in the Primary and Dual processors may receive interrupts from the I/O APIC via the three-wire APIC bus, locally via the local interrupt pins (LINT0, LINT1), or from the other processor via the APIC bus. The local interrupt pins, LINT0 and LINT1, are shared with the INTR and NMI pins, respectively. When the APIC is bypassed (hardware disabled) or programmed in "through local" mode, the 8259A interrupt (INTR) and NMI are connected to the INTR/LINT0 and NMI/LINT1 pins of the processor. Figure 16-15 shows the APIC implementation in the embedded Pentium processor. Note that the PICCLK has a maximum frequency of 16.67 MHz.

When the local APIC is hardware enabled, *data* memory accesses to its 4 Kbyte address space are executed internally and do not generate an ADS# on the processor bus. However, a *code* memory access in the 4 KByte APIC address space will not be recognized by the APIC and will generate a cycle on the processor bus.

Note: Internally executed data memory accesses may cause the address bus to toggle even though no ADS# is issued on the processor bus.

Figure 16-15. Local APIC Interface



16.9.1 APIC Configuration Modes

There are four possible APIC Modes:

- Normal mode
- Bypass mode (hardware disable)
- Through local mode
- Masked mode (software disable)

16.9.1.1 Normal Mode

This is the normal operating mode of the local APIC. When in this mode, the local APIC is both hardware and software enabled.

16.9.1.2 Bypass Mode

Bypass mode effectively removes (bypasses) the APIC from the embedded Pentium processor, causing it to operate as if there were no APIC present. Any accesses to the APIC address space go to memory. APICEN is sampled at the falling edge of RESET, and later becomes the PICD1 (part of the APIC 3-wire bus) signal. Bypass mode is entered by driving APICEN low at the falling edge of RESET. Since the APIC must be used to enable the Dual processor after RESET, PICD1 must be driven high at reset to ensure that APIC is hardware enabled if a second processor is present.

For hardware disabling operations, the following implications must be considered:

- The INTR and NMI pins become functionally equivalent to the corresponding interrupt pins in the embedded Pentium processor, and the APIC is bypassed.
- The APIC PICCLK must be tied high.
- The system will not operate with the Dual Processor if the local APIC is hardware disabled.

16.9.1.3 Through Local Mode

Configuring in Through Local Mode allows the APICs to be used for the dual-processor bootup handshake protocol and then pass interrupts through the local APIC to the core to support an external interrupt controller.

To use the Through Local Mode of the local APIC, the APIC must be enabled in both hardware and software. This is done by programming two local vector table entries, LVT1 and LVT2, at addresses 0FEE00350H and 0FEE00360H, as external interrupts (ExtInt) and NMI, respectively. The 8259A responds to the INTA cycles and returns the interrupt vector to the processor.

The local APIC should not be sent any interrupts prior to it’s being programmed. Once the APIC is programmed it can receive interrupts.

Note that although external interrupts and NMI are passed through the local APIC to the core, the APIC can still receive messages on the APIC bus.

16.9.1.4 Masked Mode

The local APIC is initialized to masked mode once hardware enabled via the APICEN pin. In order to be programmed in normal or Through Local Modes, the APIC must be “software enabled.” Once operating in normal or Through Local Modes, the APIC may be disabled by software by clearing bit 8 of the APIC’s spurious vector interrupt register (Note: this register is normally cleared at RESET and INIT). This register is at address 0FEE000F0H. Disabling APIC in software returns it to Masked mode. With the exception of NMI, SMI, INIT, remote reads, and the startup IPI, all interrupts are masked on the APIC bus. The local APIC does not accept interrupts on LINT0 or LINT1.

16.9.1.5 Software Disabling Implications

For the software disabling operations, the following implications must be considered:

- The 4-Kbyte address space for the APIC is always blocked for data accesses (i.e., external memory in this region must not be accessed).
- The interrupt control register (ICR) can be read and written (e.g., interprocessor interrupts are sent by writing to this register).

- The APIC can continue to receive SMI, NMI, INIT, “startup,” and remote read messages.
- Local interrupts are masked.
- Software can enable/disable the APIC at any time. After software disabling the local APICs, pending interrupts must be handled or masked by software.
- The APIC PICCLK must be driven at all times.

16.9.1.6 Dual Processing with the Local APIC

The Dual processor bootup protocol may be used in the normal, through local, or masked modes.

16.9.2 Loading the APIC ID

Loading the APIC ID may be done with external logic that would drive the proper address at reset. If the BE3#–BE0# signals are not driven and do not have external resistors to V_{CC} or V_{SS}, the APIC ID value defaults to 0000 for the Primary processor and 0001 for the Dual processor.

Table 16-9. APIC ID

APIC ID Register Bit	Pin Latched at RESET
bit 24	BE0#
bit 25	BE1#
bit 26	BE2#
bit 27	BE3#

Warning: An APIC ID of all 1s is an APIC special case (i.e., a broadcast) and must not be used. Since the Dual processor inverts the lowest order bit of the APIC ID placed on the BE pins, the value “1110” should also be avoided when operating in Dual Processing mode.

In a dual processor configuration, the OEM and Socket 5 should have the four BE pairs tied together. The OEM processor loads the value seen on these four pins at RESET. The dual processor loads the value seen on these pins and automatically inverts bit 24 of the APIC ID Register. Thus, the two processors have unique APIC ID values.

In a general multi-processing system consisting of multiple embedded Pentium processors, these pins must not be tied together, so each local APIC can have unique ID values.

These four pins must be valid and stable two clocks before and after the falling edge of RESET.

16.9.3 Response to HOLD

While the embedded Pentium processor is accessing the APIC, the processor will respond to a HOLD request with a maximum delay of six clocks. To external agents that are not aware of the APIC bus, this looks like the processor is not responding to HOLD even though ADS# has not been driven and the processor bus seems idle.

16.10 Fractional Speed Bus

The embedded Pentium processor is offered in various bus-to-core frequency ratios. The BF2–BF0 configuration pins determine the bus-to-core frequency ratio. The processor multiplies the input CLK by the bus-to-core ratio to achieve higher internal core frequencies.

Note: Only the Low-power Embedded Pentium Processor with MMX technology has a BF2 pin.

The external bus frequency is set on power-up RESET through the CLK pin. The processor samples the BF*n* pins on the falling edge of RESET to determine which bus-to-core ratio to use. When the BF*n* pins are left unconnected, the embedded Pentium processor defaults to the 2/3 ratio and the embedded Pentium processor with MMX technology defaults to the 1/2 ratio. BF*n* settings must not change its value while RESET is active. Changing the external bus speed or bus-to-core ratio requires a “power-on” RESET pulse initialization. Once a frequency is selected, it may not be changed with a warm-reset (15 clocks). The BF pin must meet a 1 ms setup time to the falling edge of RESET.

Each embedded Pentium processor is specified to operate within a single bus-to-core ratio and a specific minimum to maximum bus frequency range (corresponding to a minimum to maximum core frequency range). Operation in other bus-to-core ratios or outside the specified operating frequency range is not supported. Tables 16-10 through 16-12 summarize these specifications.

Table 16-10. Bus-to-Core Frequency Ratios for the Embedded Pentium® Processor (at 100/133/166 MHz)

BF1	BF0	Embedded Pentium® Processor Bus/Core Ratio	Max Bus/Core Frequency (MHz)	Min Bus/Core Frequency (MHz)
0	0	2/5	66/166	33/83
1	0	1/2	66/133	33/66
1	1	2/3[1]	66/100	33/50

NOTES:
1. This is the default bus fraction for the embedded Pentium processor (at 100/133/166 MHz). If the BF pins are left floating, the processor will be configured for the 2/3 bus to core frequency ratio.
2. All other BF1–BF0 settings are Reserved for the embedded Pentium processor (at 100/133/166 MHz).

Table 16-11. Bus-to-Core Frequency Ratios for the Embedded Pentium® Processor with MMX™ Technology

BF1	BF0	Embedded Pentium Processor with MMX™ Technology Bus/Core Ratio	Max Bus/Core Frequency (MHz)	Min Bus/Core Frequency (MHz)
1	1	2/7	66/233	33/117
0	1	1/3	66/200	33/100
1	0	1/2[†]	N/A	N/A

† This is the default bus-to-core ratio for the Pentium processor with MMX technology. If the BF pins are left floating, the processor will be configured for the 1/2 bus-to-core frequency ratio, which is unsupported. *Do not float the BF pins at RESET.*

Table 16-12. Bus-to-Core Frequency Ratios for the Low-Power Embedded Pentium® Processor with MMX™ Technology

BF2	BF1	BF0	Low-Power Embedded Pentium Processor with MMX™ Technology Bus/Core Ratio4	Max Bus/Core Frequency (MHz)	Min Bus/Core Frequency (MHz)
0	0	0	2/5	66/166	
1	0	0	1/4	66/266	

16.10.1 Fractional Bus Operation Examples

The following examples illustrate the embedded Pentium processor synchronization mechanism.

Figure 16-16. Processor 1/2 Bus Internal/External Data Movement

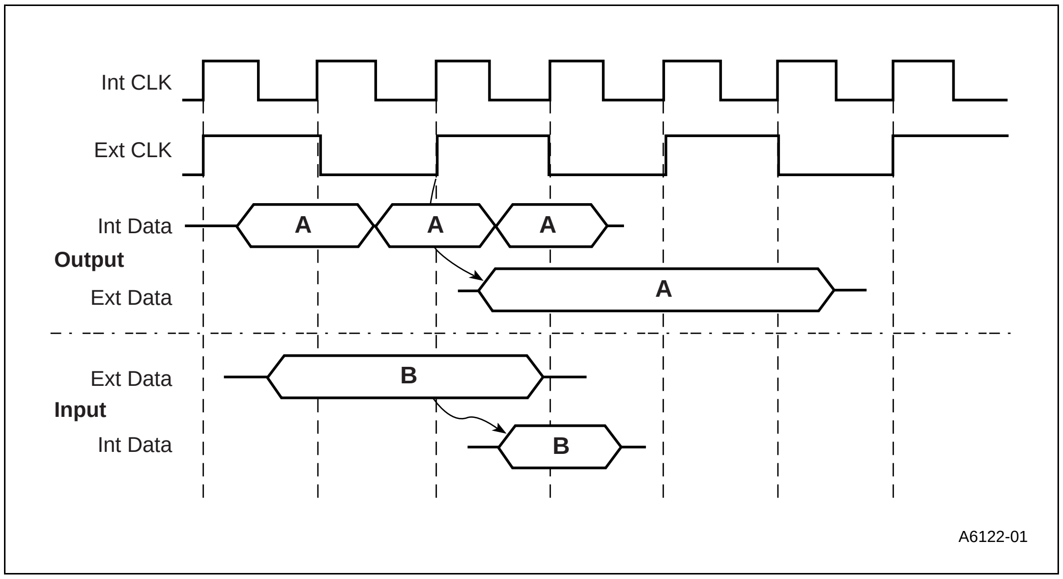


Figure 16-17. Processor 2/3 Bus Internal/External Data Movement

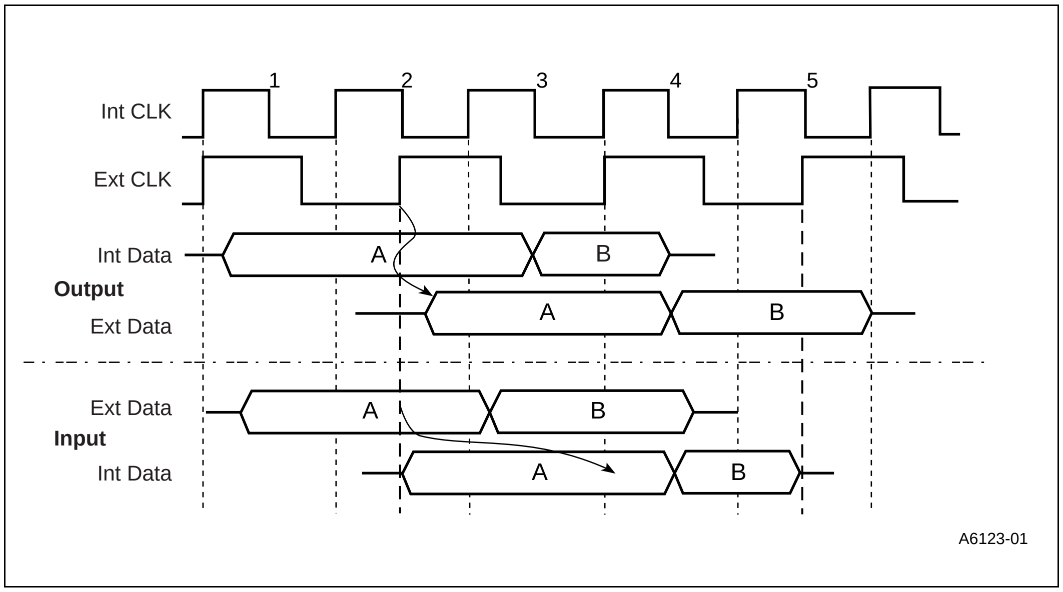


Figure 16-18 shows how the embedded Pentium processor prevents data from changing in clock 2, where the 2/3 external clock rising edge occurs in the middle of the internal clock phase, so it can be properly synchronized and driven.

Figure 16-18. Processor 2/5 Bus Internal/External Data Movement

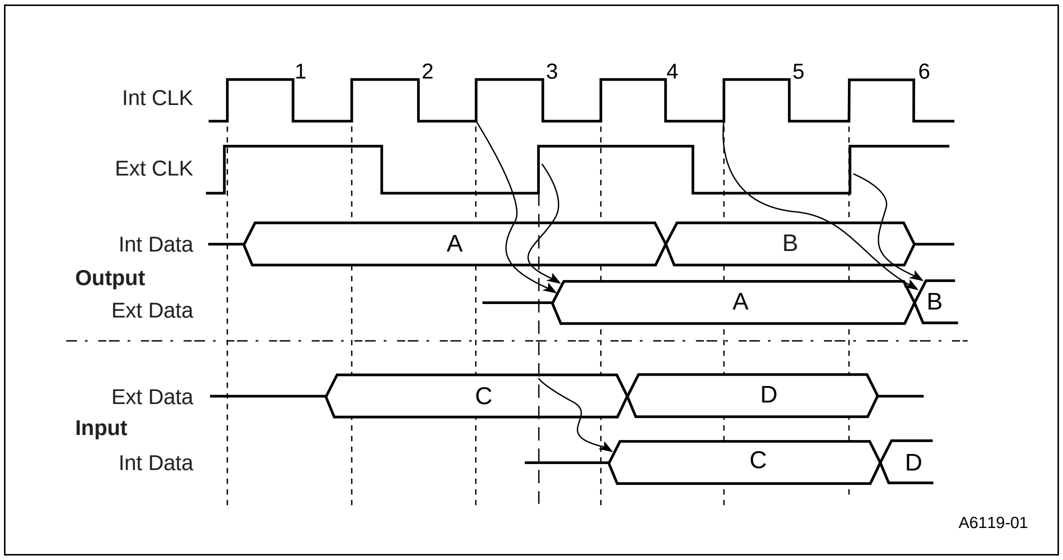
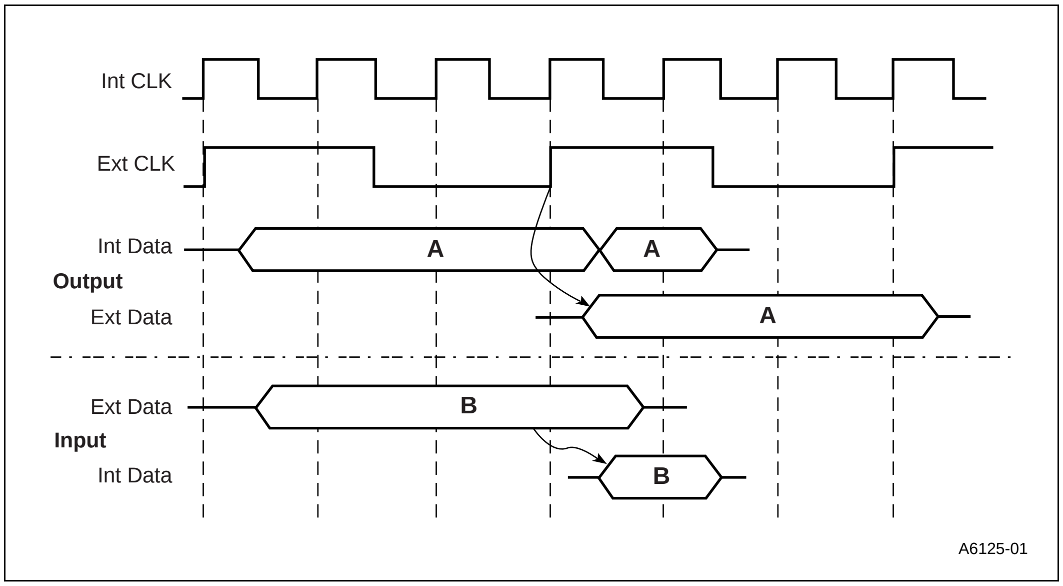


Figure 16-19. **Processor 1/3 Bus Internal/External Data Movement**



16.11 Power Management

16.11.1 I/O Instruction Restart

I/O Instruction restart is a power management feature of the embedded Pentium processor that allows the processor to re-execute an I/O instruction. In this way, an I/O instruction can alert a sleeping device in a system and SMI# can be recognized before the I/O instruction is re-executed. SMI# assertion causes a wake-up routine to be executed, so the restarted I/O instruction can be executed by the system.

16.11.2 Stop Clock and Auto Halt Powerdown

The embedded Pentium processor uses Stop Clock and Auto Halt Powerdown to immediately reduce the power of each device. These features cause the clock to be stopped to most of the processor's internal units and thus significantly reduce power consumption by the processor as a whole.

Stop clock is enabled by asserting the STPCLK# pin of the embedded Pentium processor. While asserted, the embedded Pentium processor stops execution and does service interrupts, but allows external and interprocessor (Primary and Dual processor) snooping.

AutoHalt Powerdown is entered once the embedded Pentium processor executes a HLT instruction. In this state, most internal units are powered-down, but the embedded Pentium processor recognizes all interrupts and snoops.

Embedded Pentium processor pin functions (D/P#, etc.) are not affected by STPCLK# or AutoHalt.

For additional details on power management, refer to Chapter 24, "Power Management."

16.12 CPUID Instruction

The CPUID instruction provides information to software about the vendor, family, model and stepping of the microprocessor on which it is executing. In addition, it indicates the features supported by the processor.

When executing CPUID:

- If the value in EAX is “0,” then the 12-byte ASCII string “GenuineIntel” (little endian) is returned in EBX, EDX, and ECX. Also, EAX contains a value of “1” to indicate the largest value of EAX which should be used when executing CPUID.
- If the value in EAX is “1,” then the processor version is returned in EAX and the processor capabilities (feature flags) are returned in EDX.
- If the value in EAX is neither “0” nor “1”, the embedded Pentium processor writes “0” to EAX, EBX, ECX, and EDX.

The following EAX value is defined for the CPUID instruction executed with EAX = 1. The processor version EAX bit assignments are given in Figure 16-20. Table 16-13 lists the feature flag bits assignment definitions.

Figure 16-20. EAX Bit Assignments for CPUID

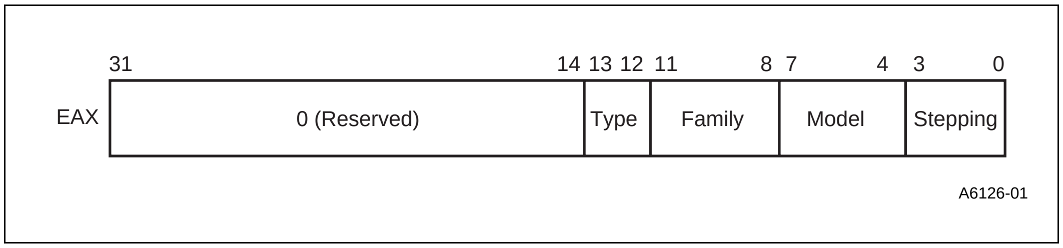


Table 16-13. EDX Bit Assignment Definitions (Feature Flags)

Bit	Name	Value	Description When Flag=1	Comments
0	FPU	1	Floating-point unit on-chip	The processor contains an FPU that supports the Intel 387 floating-point instruction set.
1	VME	1	Virtual Mode Enhancements	The processor supports extensions to virtual-8086 mode.
2	DE	1	Debugging Extension	The processor supports I/O breakpoints, including the CR4.DE bit for enabling debug extensions and optional trapping of access to the DR4 and DR5 registers.
3	PSE	1	Page Size Extension	The processor supports 4-Mbyte pages.
4	TSC	1	Time Stamp Counter	The RDTSC instruction is supported including the CR4.TSD bit for access/privilege control.
5	MSR	1	Embedded Pentium® Processor MSR	Model SpecificRegisters are implemented with the RDMSR, WRMSR instructions.
6	PAE	0	Physical Address Extension	Physical addresses greater than 32 bits are supported.
7	MCE	1	Machine Check Exception	Machine Check Exception, Exception 18, and the CR4.MCE enable bit are supported.
8	CX8	1	CMPXCHG8B Instruction Supported	The compare and exchange 8 bytes instruction is supported.
9	APIC	1	On-chip PIC Hardware Enabled†	The processor contains a local APIC.
10-11		R	Reserved	Do not rely on its value.
12	MTRR	0	Memory Type Range Registers	The processor supports the Memory Type Range Registers specifically the MTRR_CAP register.
13	PGE	0	Page Global Enable	The global bit in the PDE's and PTE's and the CR4.PGE enable bit are supported.
14	MCA	0	Machine Check Architecture	The Machine Check Architecture is supported, specifically the MCG_CAP register.
15-22		R	Reserved	Do not rely on its value.
23	MMX technology	1	Intel Architecture MMX™ technology supported	The processor supports the MMX technology instruction set extensions to the Intel Architecture.
24-31		R	Reserved	Do not rely on its value.

† Indicates that the APIC is present and hardware is enabled (software disabling does not affect this bit).

The family field for the embedded Pentium processor family is 0101B (5H). The model value for the embedded Pentium processor is 0010B (2H) or 0111B (7H), and the model value for the embedded Pentium processor with MMX technology is 0100B (4H). The model value for the low-power embedded Pentium processor with MMX technology is 1000B (8H)

Note: Use the MMX technology feature bit (bit23) in the EFLAGS register, not the model value, to detect the presence of the MMX technology feature set.

For specific information on the stepping field, consult the embedded Pentium processor family Specification Update. The type field is defined in Table 16-14.

Table 16-14. EAX Type Field Values

Bit 13	Bit 12	Processor Type
0	0	Embedded Pentium® processor, embedded Pentium processor with MMX™ technology or low-power embedded Pentium processor with MMX technology
0	1	Reserved
1	0	Dual embedded Pentium processor
1	1	Reserved

16.13 Model Specific Registers

Each embedded Pentium processor contains certain Model Specific Registers that are used in execution tracing, performance monitoring, testing, and machine check errors. They are unique to that embedded Pentium processor and may not be implemented in the same way in future processors.

Two instructions, RDMSR and WRMSR (read/write model specific registers) are used to access these registers. When these instructions are executed, the value in ECX specifies which model specific register is being accessed.

Software must not depend on the value of reserved bits in the model specific registers. Any writes to the model specific registers should write "0" into any reserved bits.

For more information, refer to Chapter 26, "Model Specific Registers and Functions."