Intel Architecture Optimization Manual

Order Number 242816-003

1997

1

Introduction to the Intel Architecture Optimization Manual

CHAPTER 1
INTRODUCTION TO THE INTEL ARCHITECTURE OPTIMIZATION MANUAL

In general, developing fast applications for Intel Architecture (IA) processors is not difficult. An understanding of the architecture and good development practices make the difference between a fast application and one that runs significantly slower than its full potential. Of course, applications developed for the 8086/8088, 80286, Intel386™ (DX or SX), and Intel486™ processors will execute on the Pentium®, Pentium Pro and Pentium II processors without any modification or recompilation. However, the following code optimization techniques and architectural information will help you tune your application to its greatest potential.

1.1 TUNING YOUR APPLICATION

Tuning an application to execute fast across the Intel Architecture (IA) is relatively simple when the programmer has the appropriate tools. To begin the tuning process, you need the following:

- Knowledge of the Intel Architecture. See Chapter 2.
- Knowledge of critical stall situations that may impact the performance of your application. See Chapters 3, 4 and 5.
- Knowledge of how good your compiler is at optimization and an understanding of how to help the compiler produce good code.
- Knowledge of the performance bottlenecks within your application. Use the VTune performance monitoring tool described in this document.
- Ability to monitor the performance of the application. Use VTune.

VTune, Intel's Visual Tuning Environment Release 2.0 is a useful tool to help you understand your application and where to begin tuning. The Pentium and Pentium Pro processors provide the ability to monitor your code with performance event counters. These performance event counters can be accessed using VTune. Within each section of this document the appropriate performance counter for measurement will be noted with additional tuning information. Additional information on the performance counter events and programming the counters can be found in Chapter 7. Section 1.4 contains order information for VTune.

1.2 ABOUT THIS MANUAL

It is assumed that the reader is familiar with the Intel Architecture software model and assembly language programming.

This manual describes the software programming optimizations and considerations for IA processors with and without MMX technology. Additionally, this document describes the implementation differences of the processor members and the optimization strategy that gives the best performance for all members of the family.

This manual is organized into seven chapters, including this chapter (Chapter 1), and four appendices.

Chapter 1 — Introduction to the Intel Architecture Optimization Manual

Chapter 2 — Overview of Processor Architecture and Pipelines: This chapter provides an overview of IA processor architectures and an overview of IA MMX technology.

Chapter 3 — Optimization Techniques for Integer Blended Code: This chapter lists the integer optimization rules and provides explanations of the optimization techniques for developing fast integer applications.

Chapter 4 — Guidelines for Developing MMX™ Technology Code: This chapter lists the MMX technology optimization rules, with an explanation of the optimization techniques and coding examples specific to MMX technology.

Chapter 5 — Optimization Techniques for Floating-Point Applications: This chapter contains a list of rules, optimization techniques, and code examples specific to floating-point code.

Chapter 6 — Suggestions for Choosing a Compiler: This section includes an overview of the architectural differences and a recommendation for blended code.

Chapter 7 — Intel Architecture Performance Monitoring Extensions: This chapter details the performance monitoring counters and their functions.

Appendix A — Integer Pairing Tables: This appendix lists the IA integer instructions with pairing information for the Pentium processor.

Appendix B — Floating-Point Pairing Tables: This appendix lists the IA floating-point instructions with pairing information for the Pentium processor.

Appendix C — Instruction to Micro-op Breakdown

Appendix D — Pentium® Pro Processor Instruction to Decoder Specification: This appendix summarizes the IA macro instructions with Pentium Pro processor decoding information to enable scheduling for the decoder.

1.3 RELATED DOCUMENTATION

Refer to the following documentation for more information on the Intel Architecture and specific techniques referred to in this manual:

- *Intel Architecture MMX™ Technology Programmer's Reference Manual*, Order Number 243007.
- *Pentium® Processor Family Developer's Manual*, Volumes 1, 2 and 3, Order Numbers 241428, 241429 and 241430.
- *Pentium® Pro Processor Family Developer's Manual*, Volumes 1, 2 and 3, Order Numbers 242690, 242691 *and* 242692.

1.4 VTune ORDER INFORMATION

Refer to the VTune home page on the World Wide Web for current order information:

http://www.intel.com/ial/vtune

To place an order in the USA and Canada call 1-800-253-3696 or call Programmer's Paradise at 1-800-445-7899.

International Orders can be placed by calling 503-264-2203.

2

Overview of Processor Architecture and Pipelines

CHAPTER 2
OVERVIEW OF PROCESSOR ARCHITECTURE AND PIPELINES

This section provides an overview of the pipelines and architectural features of Pentium and P6-family processors with and without MMX technology. By understanding how the code flows through the pipeline of the processor, you can better understand why a specific optimization will improve the speed of your code. This information will help you best utilize the suggested optimizations.

2.1 THE PENTIUM® PROCESSOR

The Pentium processor is an advanced superscala*r* processor. It is built around two general purpose integer pipelines and a pipelined floating-point unit. The Pentium processor can execute two integer instructions simultaneously. A software-transparent dynamic branch-prediction mechanism minimizes pipeline stalls due to branches.

2.1.1 Integer Pipelines

The Pentium processor has two parallel integer pipelines as shown in Figure 2-1. The main pipe (U) has five stages: prefetch (PF), Decode stage 1(D1), Decode stage 2 (D2), Execute (E), and Writeback (WB). The secondary pipe (V) is similar to the main one but has some limitations on the instructions it can execute. The limitations will be described in more detail in later sections.

The Pentium processor can issue up to two instructions every cycle. During execution, the next two instructions are checked and, if possible, they are issued such that the first one executes in the U-pipe, and the second in the V-pipe. If it is not possible to issue two instructions, then the next instruction is issued to the U-pipe and no instruction is issued to the V-pipe.

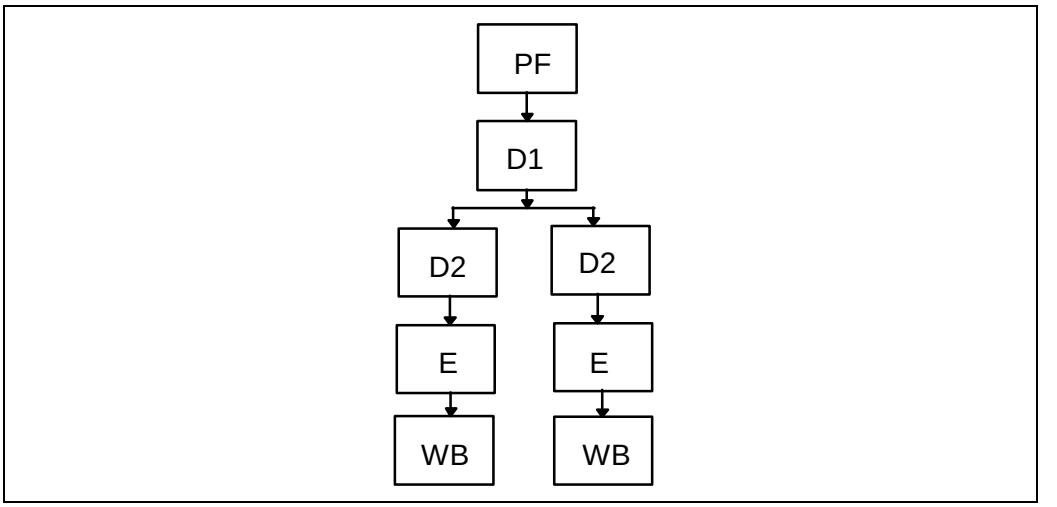


Figure 2-1. Pentium® Processor Integer Pipelines

When instructions execute in the two pipes, the functional behavior of the instructions 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. In the Pentium processor's pipelines, the D2 stage, in which addresses of memory operands are calculated, can perform a multiway add, so there is not a one-clock index penalty as with the Intel486 processor pipeline.

With the superscalar implementation, it is important to schedule the instruction stream to maximize the usage of the two integer pipelines.

2.1.2 Caches

The on-chip cache subsystem consists of two 8-Kbyte two-way set associative caches (one instruction and one data) with a cache line length of 32 bytes. There is a 64-bit wide external data bus interface. The caches employ a write back mechanism and an LRU replacement algorithm. The data cache consists of eight banks interleaved on four byte boundaries. The data cache can be accessed simultaneously from both pipes, as long as the references are to different banks. The minimum delay for a cache miss is four clocks.

2.1.3 Instruction Prefetcher

The instruction prefetcher has four 32-byte buffers. In the prefetch (PF) stage, the two independent pairs of line-size 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 to be taken, prefetch requests continue linearly. On a branch that

is predicted to be taken, the other prefetch buffer is enabled and begins to prefetch as though the branch were taken. If a branch is discovered to be mispredicted, the instruction pipelines are flushed and prefetching activity starts over. The prefetcher can fetch an instruction which is split among two cache lines with no penalty. Because the instruction and data caches are separate, instruction prefetches do not conflict with data references for access to the cache.

2.1.4 Branch Target Buffer

The Pentium processor employs a dynamic branch prediction scheme with a 256-entry BTB. If the prediction is correct, there is no penalty when executing a branch instruction. If the branch is mispredicted, there is a three-cycle penalty if the conditional branch was executed in the U-pipe or a four-cycle penalty if it was executed in the V-pipe. Mispredicted calls and unconditional jump instructions have a three-clock penalty in either pipe.

NOTE

Branches that are not taken are not inserted in the BTB until they are mispredicted.

2.1.5 Write Buffers

The Pentium processor has two write buffers, one corresponding to each of the integer pipelines, to enhance the performance of consecutive writes to memory. These write buffers are one quad-word wide (64-bits) and can be filled simultaneously in one clock, for example by two simultaneous write misses in the two instruction pipelines. Writes in these buffers are sent out to 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 Pentium processor supports strong write ordering, which means that writes happen in the order that they occur.

2.1.6 Pipelined Floating-Point Unit

The Pentium processor provides a high performance floating-point unit that appends a three-stage floating-point pipe to the integer pipeline. floating-point instructions proceed through the pipeline until the E stage. Instructions then spend at least one clock at each of the floating-point stages: X1 stage, X2 stage *and* WF stage. Most floating-point instructions have execution latencies of more than one clock, however most are pipelined which allows the latency to be hidden by the execution of other instructions in different stages of the pipeline. Additionally, integer instructions can be issued during long latency floating-point instructions, such as `FDIV`. Figure 2-2 illustrates the integer and floating-point pipelines.

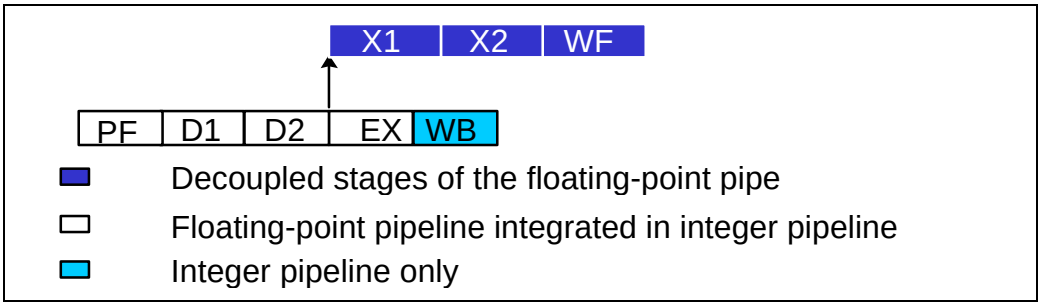


Figure 2-2. Integration of Integer and Floating-Point Pipeline

The majority of the frequently used instructions are pipelined so that the pipelines can accept a new pair of instructions every cycle. Therefore a good code generator can achieve a throughput of almost two instruction per cycle (this assumes a program with a modest amount of natural parallelism). The FXCH instruction can be executed in parallel with the commonly used floating-point instructions, which lets the code generator or programmer treat the floating-point stack as a regular register set with a minimum of performance degradation.

2.2 THE PENTIUM® PRO PROCESSOR

The Pentium Pro processor family uses a dynamic execution architecture that blends out-of-order and speculative execution with hardware register renaming and branch prediction. These processors feature an in-order issue pipeline, which breaks IA processor macroinstructions into simple, micro-operations called micro-ops or µops, and an out-of-order, superscalar processor core, which executes the micro-ops. The out-of-order core of the processor contains several pipelines to which integer, branch, floating-point and memory execution units are attached. Several different execution units may be clustered on the same pipeline. For example, an integer arithmetic logic unit and the floating-point execution units (adder, multiplier and divider) share a pipeline. The data cache is pseudo-dual ported via interleaving, with one port dedicated to loads and the other to stores. Most simple operations (such as integer ALU, floating-point add and floating-point multiply) can be pipelined with a throughput of one or two operations per clock cycle. The floating-point divider is not pipelined. Long latency operations can proceed in parallel with short latency operations.

The Pentium Pro processor pipeline contains three parts: (1) the in-order issue front-end, (2) the out-of-order core, and (3) the in-order retirement unit. Figure 2-3 details the entire Pentium Pro processor pipeline.

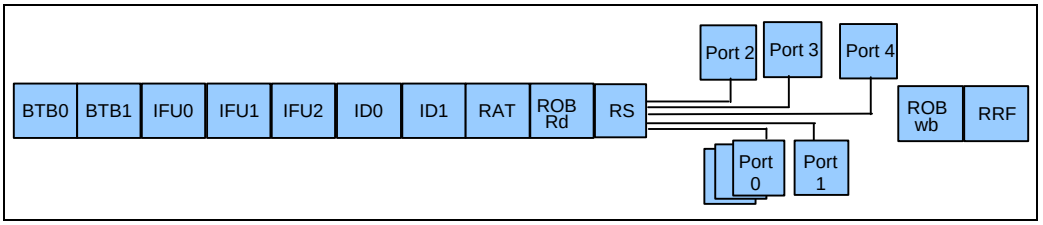


Figure 2-3. Pentium® Pro Processor Pipeline

Details about the in-order issue front-end are illustrated in Figure 2-4.

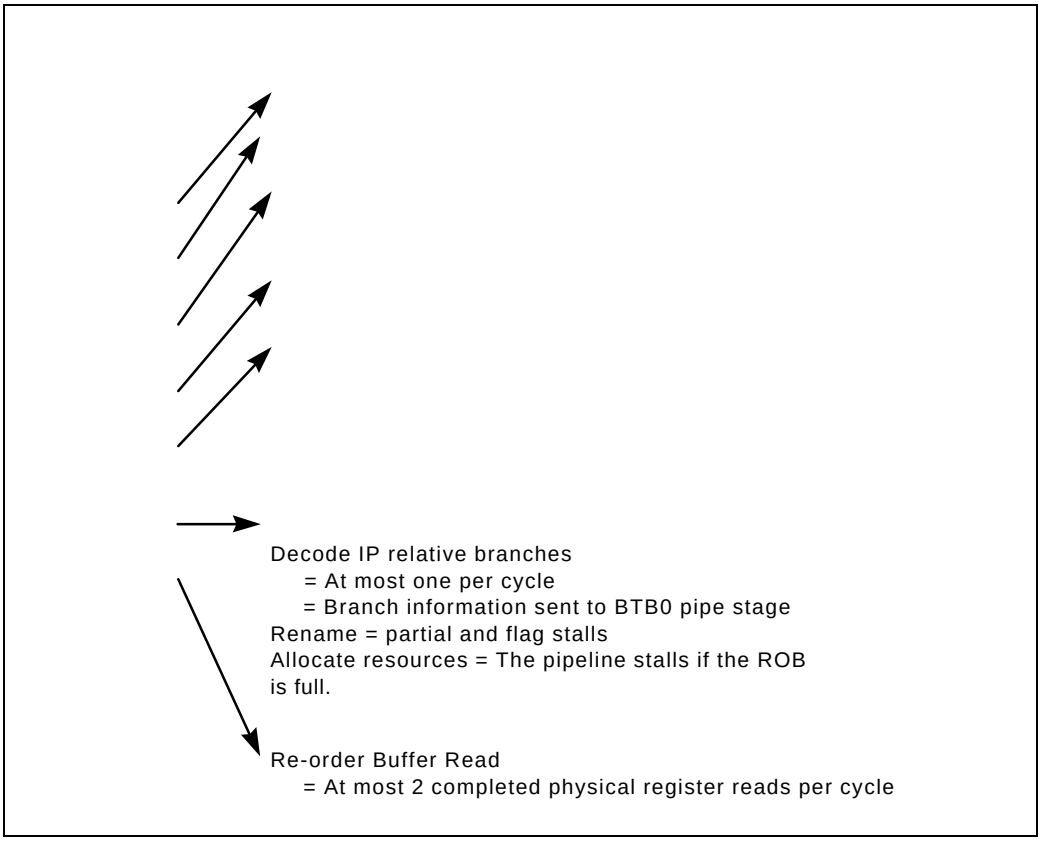


Figure 2-4. In-Order Issue Front-End

Since the Pentium Pro processor executes instructions out of program order, the most important consideration in performance tuning is making sure enough micro-ops are ready for execution. Correct branch prediction and fast decoding are essential to getting the most performance out of the in-order front-end. Branch prediction and the branch target buffer are detailed in Section 3.2. Decoding is discussed below.

During every clock cycle, up to three macro-instructions can be decoded in the ID1 pipestage. However, if the instructions are complex or are over seven bytes long, the decoder is limited to decoding fewer instructions.

The decoders can decode:

- Up to three macro-instructions per clock cycle.
- Up to six micro-ops per clock cycle.
- Macro-instructions up to seven bytes in length.

Pentium Pro processors have three decoders in the D1 pipestage. The first decoder is capable of decoding one macro-instruction of four or fewer micro-ops in each clock cycle. The other two decoders can each decode an instruction of one micro-op in each clock cycle. Instructions composed of more than four micro-ops take multiple cycles to decode. When programming in assembly language, scheduling the instructions in a 4-1-1 micro-op sequence increases the number of instructions that can be decoded each clock cycle. In general:

- Simple instructions of the register-register form are only one micro-op.
- Load instructions are only one micro-op.
- Store instructions have two micro-ops.
- Simple read-modify instructions are two micro-ops.
- Simple instructions of the register-memory form have two to three micro-ops.
- Simple read-modify write instructions are four micro-ops.
- Complex instructions generally have more than four micro-ops, therefore they take multiple cycles to decode.

See Appendix C for a table that specifies the number of micro-ops for each instruction in the Intel Architecture instruction set.

Once the micro-ops are decoded, they are issued from the in-order front-end into the Reservation Station (RS), which is the beginning pipestage of the out-of-order core. In the RS, the micro-ops wait until their data operands are available. Once a micro-op has all data operands available, it is dispatched from the RS to an execution unit. If a micro-op enters the RS in a data-ready state (that is, all data is available) and an appropriate execution unit is available, then the micro-op is immediately dispatched to the execution unit. In this case, the micro-op will spend no extra clock cycles in the RS. All of the execution units are clustered on ports coming out of the RS.

Once the micro-op has been executed it is stored in the Re-Order Buffer (ROB) and waits for retirement. In this pipestage, all data values are written back to memory and all micro-ops are retired in order, three at a time. Figure 2-5 provides details about the Out-of-Order core and the In-Order retirement pipestages.

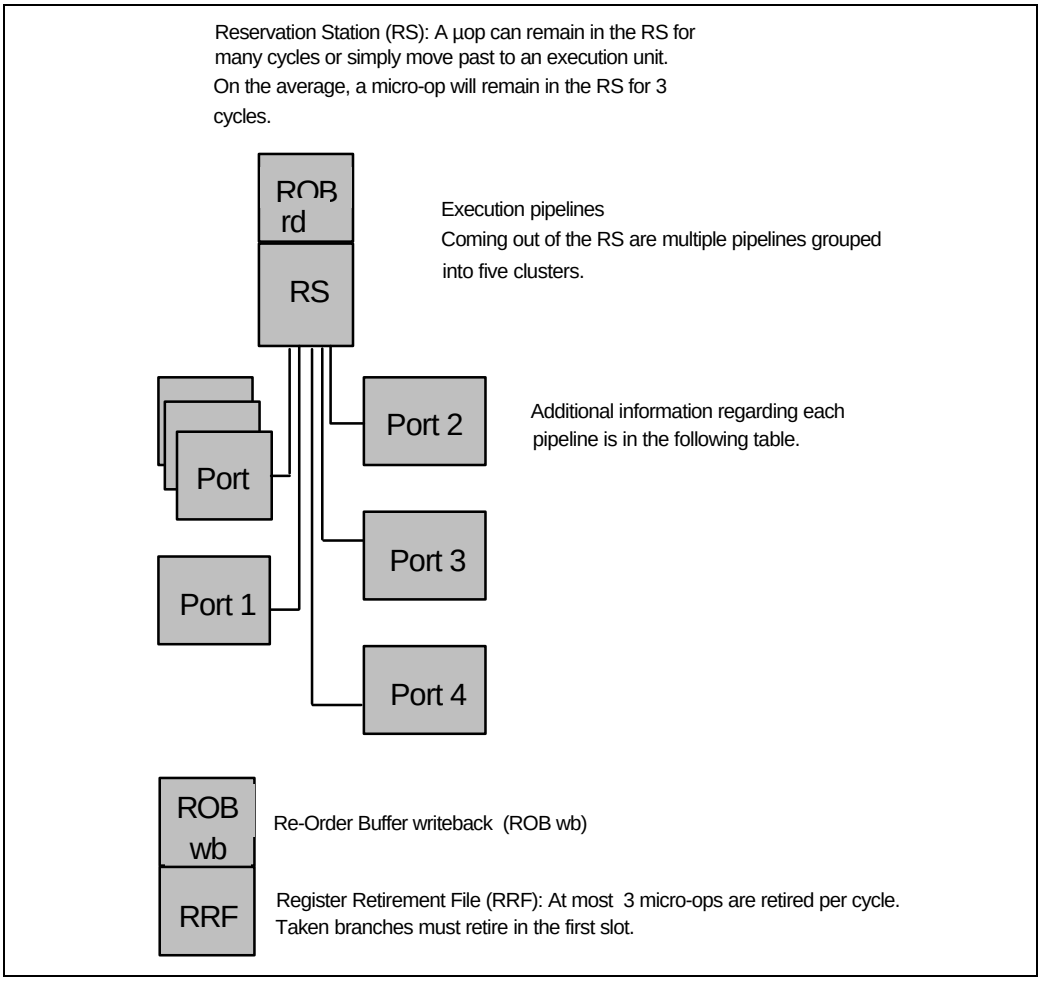


Figure 2-5. Out-Of-Order Core and Retirement Pipeline

Table 2-1. Pentium® Pro Processor Execution Units

Port	Execution Units	Latency/Thruput
0	Integer ALU Unit: LEA instructions Shift instructions Integer Multiplication instruction Floating-Point Unit: FADD instruction FMUL instruction FDIV instruction	Latency 1, Throughput 1/cycle Latency 1, Throughput 1/cycle Latency 1, Throughput 1/cycle Latency 4, Throughput 1/cycle Latency 3, Throughput 1/cycle Latency 5, Throughput 1/2cycle[1,2] Latency: single precision 17 cycles, double precision 36 cycles, extended precision 56 cycles, Throughput non-pipelined
1	Integer ALU Unit	Latency 1, Throughput 1/cycle
2	Load Unit	Latency 3 on a cache hit, Throughput 1/cycle[3]
3	Store Address Unit	Latency 3 (not applicable), Throughput 1/cycle[3]
4	Store Data Unit	Latency 1 (not applicable), Throughput 1/cycle

NOTES:

1. The FMUL unit cannot accept a second FMUL in the cycle after it has accepted the first. This is NOT the same as only being able to do FMULs on even clock cycles. FMUL is pipelined one every two clock cycles.
2. Store latency is not all that important from a dataflow perspective. The latency that matters is with respect to determining when a specific uop can retire and be completed. Store µops also have a different latency with respect to load forwarding. For example, if the store address and store data of a particular address, for example 100, dispatch in clock cycle 10, a load (of the same size and shape) to the same address 100 can dispatch in the same clock cycle 10 and not be stalled.
3. A load and store to the same address can dispatch in the same clock cycle.

2.2.1 Caches

The on-chip level one (L1) caches consist of one 8-Kbyte four-way set associative instruction cache unit with a cache line length of 32 bytes and one 8-Kbyte two-way set associative data cache unit. Not all misses in the L1 cache expose the full memory latency. The level two (L2) cache masks the full latency caused by an L1 cache miss. The minimum delay for a L1 and L2 cache miss is between 11 and 14 cycles based on DRAM page hit or miss. The data cache can be accessed simultaneously by a load instruction and a store instruction, as long as the references are to different cache banks.

2.2.2 Instruction Prefetcher

The Instruction Prefetcher performs aggressive prefetch of straight line code. Arrange code so that non-loop branches that tend to fall through take advantage of this prefetch. Additionally, arrange code so that infrequently executed code is segregated to the bottom of the procedure or end of the program where it is not prefetched unnecessarily.

Note that instruction fetch is always for an aligned 16-byte block. The Pentium Pro processor reads in instructions from 16-byte aligned boundaries. Therefore for example, if a branch

target address (the address of a label) is equal to 14 modulo 16, only two useful instruction bytes are fetched in the first cycle. The rest of the instruction bytes are fetched in subsequent cycles.

2.2.3 Branch Target Buffer

The 512-entry BTB stores the history of the previously seen branches and their targets. When a branch is prefetched, the BTB feeds the target address directly into the Instruction Fetch Unit (IFU). Once the branch is executed, the BTB is updated with the target address. Using the branch target buffer, branches that have been seen previously are dynamically predicted. The branch target buffer prediction algorithm includes pattern matching and up to four prediction history bits per target address. For example, a loop which is four iterations long should have close to 100% correct prediction. Adhering to the following guideline will improve branch prediction performance:

Program conditional branches (except for loops) so that the most executed branch immediately follows the branch instruction (that is, fall through).

Additionally, Pentium Pro processors have a Return Stack Buffer (RSB), which can correctly predict return addresses for procedures that are called from different locations in succession. This increases the benefit of unrolling loops which contain function calls and removes the need to put certain procedures in-line.

Pentium Pro processors have three levels of branch support which can be quantified in the number of cycles lost:

1. Branches that are not taken suffer no penalty. This applies to those branches that are correctly predicted as not taken by the BTB, and to forward branches that are not in the BTB, which are predicted as not taken by default.
2. Branches which are correctly predicted as taken by the BTB suffer a minor penalty (approximately 1 cycle). Instruction fetch is suspended for one cycle. The processor decodes no further instructions in that period, possibly resulting in the issue of less than four µops. This minor penalty applies to unconditional branches which have been seen before (i.e., are in the BTB). The minor penalty for correctly predicted taken branches is one lost cycle of instruction fetch, plus the issue of no instructions after the branch.
3. Mispredicted branches suffer a significant penalty. The penalty for mispredicted branches is at least nine cycles (the length of the In-order Issue Pipeline) of lost instruction fetch, plus additional time spent waiting for the mispredicted branch instruction to retire. This penalty is dependent upon execution circumstances. Typically, the average number of cycles lost because of a mispredicted branch is between 10 and 15 cycles and possibly as many as 26 cycles.

2.2.3.1 STATIC PREDICTION

Branches that are not in the BTB, which are correctly predicted by the static prediction mechanism, suffer a small penalty of about five or six cycles (the length of the pipeline to

this point). This penalty applies to unconditional direct branches which have never been seen before.

Conditional branches with negative displacement, such as loop-closing branches, are predicted taken by the static prediction mechanism. They suffer only a small penalty (approximately six cycles) the first time the branch is encountered and a minor penalty (approximately one cycle) on subsequent iterations when the negative branch is correctly predicted by the BTB.

The small penalty for branches that are not in the BTB but which are correctly predicted by the decoder is approximately five cycles of lost instruction fetch as opposed to 10 – 15 cycles for a branch that is incorrectly predicted or that has no prediction.

2.2.4 Write Buffers

Pentium Pro processors temporarily stores each write (store) to memory in a write buffer. The write buffer improves processor performance by allowing the processor to continue executing instructions without having to wait until a write to memory and/or to a cache is complete. It also allows writes to be delayed for more efficient use of memory-access bus cycles. Writes stored in the write buffer are always written to memory in program order. Pentium Pro processors use processor ordering to maintain consistency in the order that data is read (loaded) and written (stored) in a program and the order in which the processor actually carries out the reads and writes. With this type of ordering, reads can be carried out speculatively and in any order, reads can pass buffered writes, and writes to memory are always carried out in program order.

2.3 IA PROCESSORS WITH MMX™ TECHNOLOGY

Intel's MMX technology is an extension to the Intel Architecture (IA) instruction set. The technology uses a Single Instruction, Multiple Data (SIMD) technique to speed up multimedia and communications software by processing data elements in parallel. The MMX instruction set adds 57 new opcodes and a new 64-bit quadword data type. The new 64-bit data type, illustrated in Figure 2-6, holds packed integer values upon which MMX instructions operate.

In addition, there are eight new 64-bit MMX registers, each of which can be directly addressed using the register names MM0 to MM7. Figure 2-7 shows the layout of the eight new MMX registers.

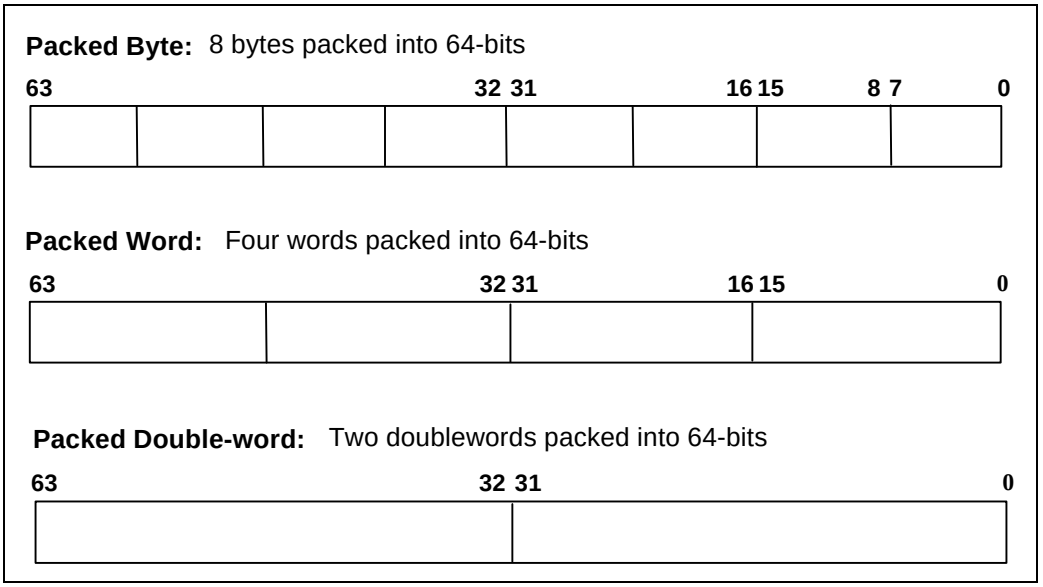


Figure 2-6. New Data Types

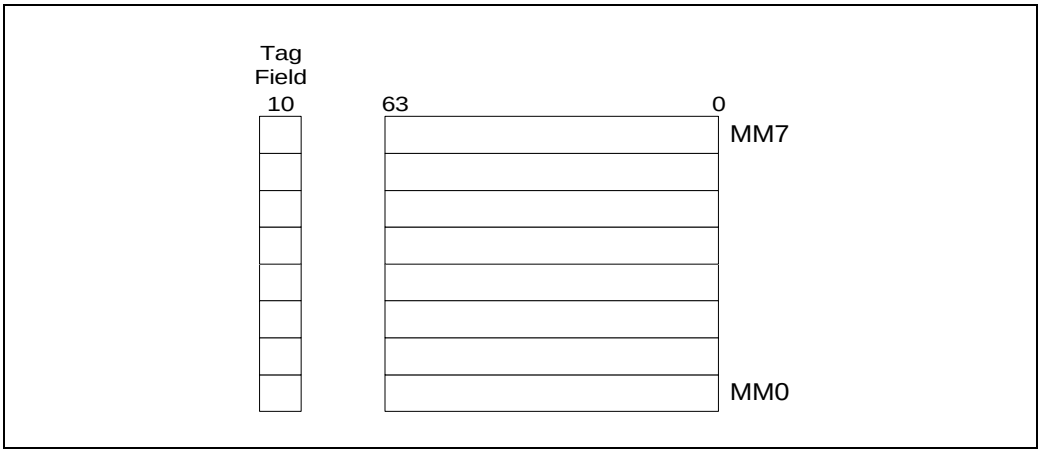


Figure 2-7. MMX™ Register Set

The MMX technology is operating-system transparent and 100% compatible with all existing Intel Architecture software; all applications will continue to run on processors with MMX technology. Additional information and details about the MMX instructions, data types and registers can be found in the *Intel Architecture MMX™ Technology Programmers Reference Manual* (Order Number 243007).

2.3.1 Superscalar (Pentium® Processor Family) Pipeline

Pentium processors with MMX technology add additional stages to the pipeline. The integration of the MMX pipeline with the integer pipeline is very similar to that of the floating-point pipe.

Figure 2-8 shows the pipelining structure for this scheme.

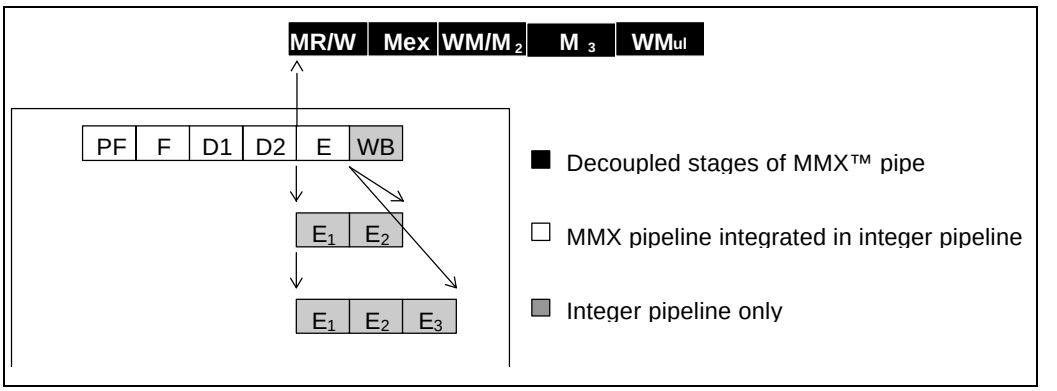


Figure 2-8. MMX™ Pipeline Structure

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

Instruction parsing is decoupled from the instruction decoding by means of an instruction First In, First Out (FIFO) buffer, which is situated between the F and Decode 1 (D1) stages. The FIFO has slots for up to four instructions. 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, then an execution stall may result from the pipeline being "starved" for instructions to execute. Stalls at the FIFO entrance may result from long instructions or prefixes (see Sections 3.7 and 3.4.2).

Figure 2-9 details the MMX pipeline on superscalar processors and the conditions in which a stall may occur in the pipeline.

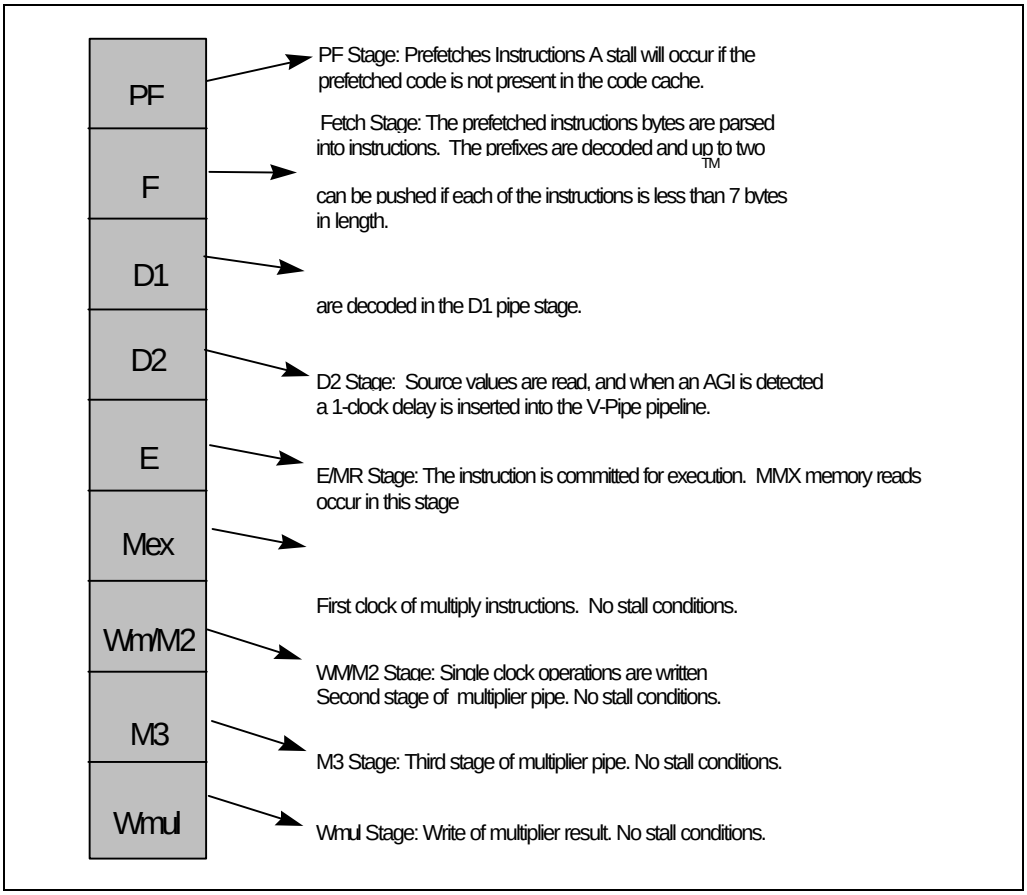


Figure 2-9. MMX™ Instruction Flow in the Pentium® Processor with MMX Technology

Table 2-2 details the functional units, latency, throughput and execution pipes for each type of MMX instruction.

Table 2-2. MMX™ Instructions and Execution Units

Operation	Number of Functional Units	Latency	Throughput	Execution Pipes
ALU	2	1	1	U and V
Multiplier	1	3	1	U or V
Shift/pack/unpack	1	1	1	U or V
Memory access	1	1	1	U only
Integer register access	1	1	1	U only

- The Arithmetic Logic Unit (ALU) executes arithmetic and logic operations (that is, add, subtract, XOR, AND).
- The Multiplier unit performs all multiplication operations. Multiplication requires three cycles but can be pipelined, resulting in one multiplication operation every clock cycle. The processor has only one multiplier unit which means that multiplication instructions cannot pair with other multiplication instructions. However, the multiplication instructions can pair with other types of instructions. They can execute in either the U- or V-pipes.
- The Shift unit performs all shift, pack and unpack operations. Only one shifter is available so shift, pack and unpack instructions cannot pair with other shift unit instructions. However, the shift unit instructions can pair with other types of instructions. They can execute in either the U- or V-pipes.
- MMX instructions that access memory or integer registers can only execute in the U-pipe and cannot be paired with any instructions that are not MMX instructions.
- After updating an MMX register, one additional clock cycle must pass before that MMX register can be moved to either memory or to an integer register.

Information on pairing requirements can be found in Section 3.3.

Additional information on instruction format can be found in the *Intel Architecture MMX™ Technology Programmer's Reference Manual* (Order Number 243007).

2.3.2 Pentium® II Processors

The Pentium II processor uses the same pipeline as discussed in Section 2.3. The addition of MMX technology is the major functional difference. Table 2-3 details the addition of MMX technology to the Pentium Pro processor execution units.

Table 2-3. Pentium® II Processor Execution Units

Port	Execution Units	Latency/Throughput
0	Integer ALU Unit	Latency 1, Throughput 1/cycle
	LEA instructions	Latency 1, Throughput 1/cycle
	Shift instructions	Latency 1, Throughput 1/cycle
	Integer Multiplication instruction	Latency 4, Throughput 1/cycle[2]
	Floating-Point Unit	
	FADD instruction	Latency 3, Throughput 1/cycle
	FMUL	Latency 5, Throughput 1/2 cycle1,2
	FDIV Unit	Latency: single precision 17 cycles, double precision 36 cycles, extended precision 56 cycles, Throughput non-pipelined
	MMX ALU Unit	Latency 1, Throughput 1/cycle
	MMX Multiplier Unit	Latency 3, Throughput 1/cycle
1	Integer ALU Unit	Latency 1, Throughput 1/cycle
	MMX ALU Unit	Latency 1, Throughput 1/cycle
	MMX Shift Unit	Latency 1, Throughput 1/cycle
2	Load Unit	Latency 3 on a cache hit, Throughput 1/cycle3
3	Store Address Unit	Latency 3 (not applicable), Throughput 1/cycle3
4	Store Data Unit	Latency 1 (not applicable), Throughput 1/cycle

NOTES:

See notes following Table 2-1.

2.3.3 Caches

The on-chip cache subsystem of Pentium processors with MMX technology and Pentium II processors consists of two 16 Kbyte four-way set associative caches with a cache line length of 32 bytes. The caches employ a write-back mechanism and a pseudo-LRU replacement algorithm. The data cache consists of eight banks interleaved on four-byte boundaries.

On Pentium processors with MMX technology, the data cache can be accessed simultaneously from both pipes, as long as the references are to different cache banks. On the P6-family processors, the data cache can be accessed simultaneously by a load instruction and a store instruction, as long as the references are to different cache banks. If the references are to the same address they bypass the cache and are executed in the same cycle. The delay for a cache miss on the Pentium processor with MMX technology is eight internal clock cycles. On Pentium II processors the minimum delay is ten internal clock cycles.

2.3.4 Branch Target Buffer

Branch prediction for Pentium processor with MMX technology and the Pentium II processor is functionally identical to the Pentium Pro processor except for one minor exception which is discussed in Section 2.3.4.1.

2.3.4.1 CONSECUTIVE BRANCHES

On the Pentium processor with MMX technology, branches may be mispredicted when the last byte of two branch instructions occurs in the same aligned four-byte section of memory, as shown in the figure below.

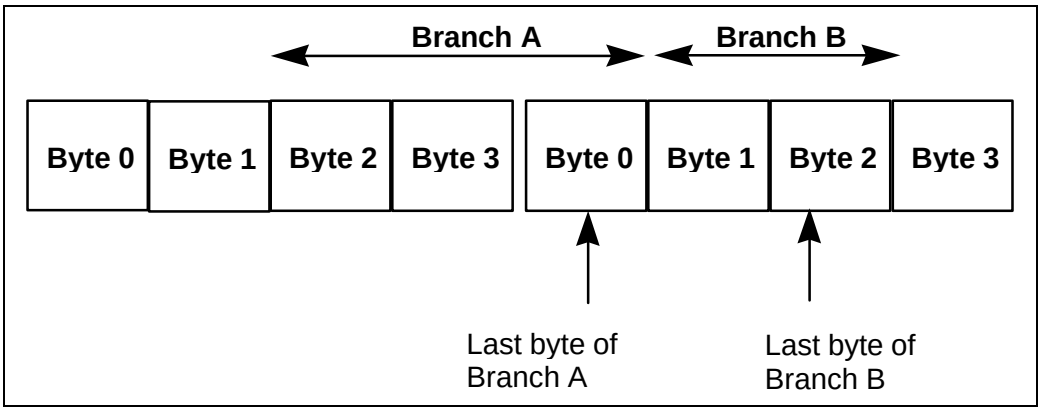


Figure 2-10. Consecutive Branch Example

This may occur when there are two consecutive branches with no intervening instructions and the second instruction is only two bytes long (such as a jump relative ±128).

To avoid a misprediction in these cases, make the second branch longer by using a 16-bit relative displacement on the branch instruction instead of an 8-bit relative displacement.

2.3.5 Write Buffers

Pentium Processors with MMX technology have four write buffers (versus two in Pentium processors without MMX technology). Additionally, the write buffers can be used by either the U-pipe or the V-pipe (versus one corresponding to each pipe in Pentium processors without MMX technology). Write hits cannot pass write misses, therefore performance of critical loops can be improved by scheduling the writes to memory. When you expect to see write misses, you should schedule the write instructions in groups no larger than four, then schedule other instructions before scheduling further write instructions.

3

Optimization Techniques for Integer-Blended Code

CHAPTER 3
OPTIMIZATION TECHNIQUES FOR INTEGER-BLENDED CODE

The following section discusses the optimization techniques which can improve the performance of applications across the Intel Architecture. The first section discusses general guidelines; the second section presents a deeper discussion about each guideline and examples of how to improve your code.

3.1 INTEGER BLENDED CODING GUIDELINES

The following guidelines will help you optimize your code to run well on Intel Architecture.

- Use a current generation compiler that will produce an optimized application. This will help you generate good code from the start. See Chapter 6.
- Work with your compiler by writing code that can be optimized. Minimize use of global variables, pointers and complex control flow. Don't use the 'register' modifier, do use the 'const' modifier. Don't defeat the type system and don't make indirect calls.
- Pay attention to the branch prediction algorithm (See Section 3.2). This is the most important optimization for Pentium Pro and Pentium II processors. By improving branch predictability, your code will spend fewer cycles fetching instructions.
- Avoid partial register stalls. See Section 3.3.
- Make sure all data are aligned. See Section 3.4.
- Arrange code to minimize instruction cache misses and optimize prefetch. See Section 3.5.
- Schedule your code to maximize pairing on Pentium processors. See Section 3.6.
- Avoid prefixed opcodes other than 0F. See Section 3.7.
- Avoid small loads after large stores to the same area of memory. Avoid large loads after small stores to the same area of memory. Load and store data to the same area of memory using the same data sizes and address alignments. See Section 3.8.
- Use software pipelining.
- Always pair CALL and RET (return) instructions.
- Avoid self-modifying code.
- Do not place data in the code segment.
- Calculate store addresses as soon as possible.

- Avoid instructions that contain four or more micro-ops or instructions that are more than seven bytes long. If possible, use instructions that require one micro-op.
- Cleanse partial registers before calling callee-save procedures.

3.2 BRANCH PREDICTION

Branch optimizations are the most important optimizations for Pentium Pro and Pentium II processors. These optimizations also benefit the Pentium processor family. Understanding the flow of branches and improving the predictability of branches can increase the speed of your code significantly.

3.2.1 Dynamic Branch Prediction

Three elements of dynamic branch prediction are important:

1. If the instruction address is not in the BTB, execution is predicted to continue without branching (fall through).
2. Predicted taken branches have a one clock delay.
3. The BTB stores a 4-bit history of branch predictions on Pentium Pro processors, Pentium II processors and Pentium processors with MMX technology. The Pentium Processor stores a two-bit history of branch prediction.

During the process of instruction prefetch the instruction address of a conditional instruction is checked with the entries in the BTB. When the address is not in the BTB, execution is predicted to fall through to the next instruction. This suggests that branches should be followed by code that will be executed. The code following the branch will be fetched and, in the case of Pentium Pro and Pentium II processors, the fetched instructions will be speculatively executed. Therefore, never follow a branch instruction with data.

Additionally, when an instruction address for a branch instruction is in the BTB and it is predicted to be taken, it suffers a one-clock delay on Pentium Pro and Pentium II processors. To avoid the delay of one clock for taken branches, simply insert additional work between branches that are expected to be taken. This delay restricts the minimum size of loops to two clock cycles. If you have a very small loop that takes less than two clock cycles, unroll it to remove the one-clock overhead of the branch instruction.

The branch predictor on Pentium Pro processors, Pentium II processors and Pentium processors with MMX technology correctly predicts regular patterns of branches (up to a length of four). For example, it correctly predicts a branch within a loop that is taken on every odd iteration, and not taken on every even iteration.

3.2.2 Static Prediction on Pentium® Pro and Pentium II Processors

On Pentium Pro and Pentium II processors, branches that do not have a history in the BTB are predicted using a static prediction algorithm, as follows:

- Predict unconditional branches to be taken.
- Predict backward conditional branches to be taken. This rule is suitable for loops.
- Predict forward conditional branches to be NOT taken.

A branch that is statically predicted can lose, at most, the six cycles of prefetch. An incorrect prediction suffers a penalty of greater than twelve clocks. The following chart illustrates the static branch prediction algorithm:

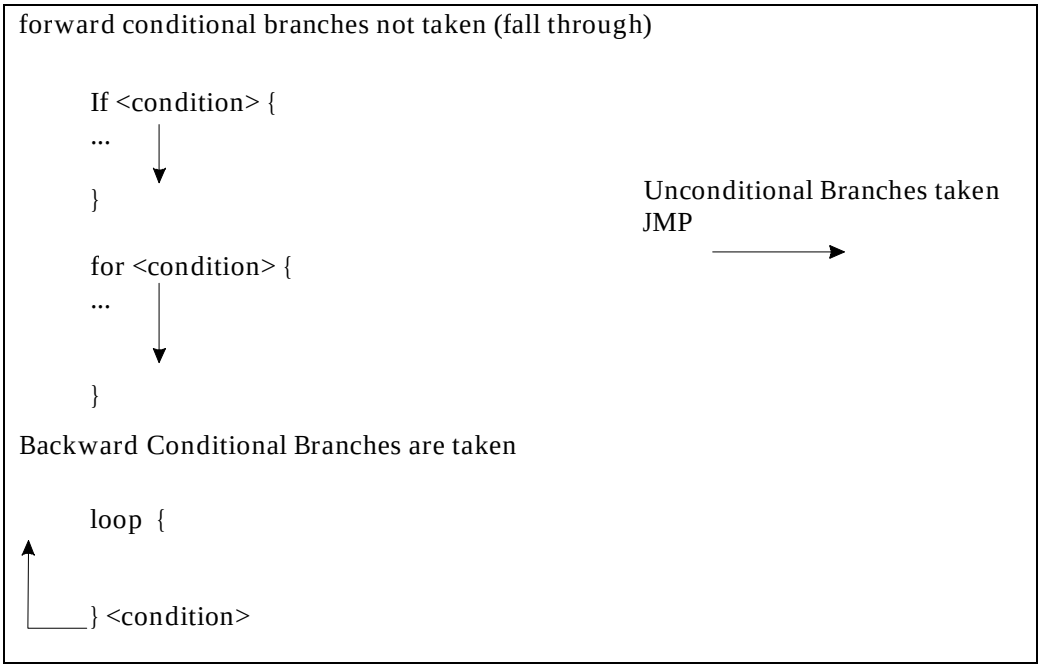


Figure 3-1. Pentium® Pro and Pentium II Processor's Static Branch Prediction Algorithm

The following examples illustrate the basic rules for the static prediction algorithm.

```
Begin:    MOV  EAX, mem32
          AND  EAX, EBX
          IMUL EAX, EDX
          SHLD EAX, 7
          JC   Begin
```

In this example, the backwards branch (`JC Begin`) is not in the BTB the first time through, therefore, the BTB will not issue a prediction. The static predictor, however, will predict the branch to be taken, so a misprediction will not occur.

```
          MOV  EAX, mem32
          AND  EAX, EBX
          IMUL EAX, EDX
          SHLD EAX, 7
          JC   Begin
          MOV  EAX, 0
Begin:    CALL Convert
```

The first branch instruction (`JC Begin`) in this code segment is a conditional forward branch. It is not in the BTB the first time through, but the static predictor will predict the branch to fall through.

The `CALL Convert` instruction will not be predicted in the BTB the first time it is seen by the BTB, but the call will be predicted as taken by the static prediction algorithm. This is correct for an unconditional branch.

In these examples, the conditional branch has only two alternatives: taken and not taken. Indirect branches, such as switch statements, computed GOTOs or calls through pointers, can jump to an arbitrary number of locations. If the branch has a skewed target destination (that is, 90% of the time it branches to the same address), then the BTB will predict accurately most of the time. If, however, the target destination is not predictable, performance can degrade quickly. Performance can be improved by changing the indirect branches to conditional branches that can be predicted.

3.2.3 Eliminating and Reducing the Number of Branches

Eliminating branches improves performance by:

- Removing the possibility of mispredictions.
- Reducing the number of BTB entries required.

Branches can be eliminated by using the `setcc` instruction, or by using the Pentium Pro processor conditional move (`CMOV` or `FCMOVE`) instructions.

Following is an example of C code with a condition that is dependent upon on of the constants:

```
ebx = (A<B) ? C1 : C2;
```

This code conditionally compares two values, A and B. If the condition is true, EBX is set to C1; otherwise it is set to C2. The assembly equivalent is shown in the example below:

```
     cmp A, B                        ; condition
     jge   L30                       ; conditional branch
     mov   ebx, CONST1
     jmp   L31                       ; unconditional branch
L30:
     mov   ebx, CONST2
L31:
```

If you replace the `jge` instruction in the previous example with a `setcc` instruction, the EBX register is set to either C1 or C2. This code can be optimized to eliminate the branches as shown in this example:

```
xor  ebx, ebx                        ;clear ebx
cmp   A, B
setge bl                             ;When ebx = 0 or 1
                                     ;OR the complement condition
dec   ebx                            ;ebx=00...00 or 11...11
and   ebx, (CONST2-CONST1)                ;ebx=0 or(CONST2-CONST1)
add   ebx, min(CONST1,CONST2)        ;ebx=CONST1 or CONST2
```

The optimized code sets EBX to zero, then compares A and B. If A is greater than or equal to B, EBX is set to one. EBX is then decremented and ANDed with the difference of the constant values. This sets EBX to either zero or the difference of the values. By adding the minimum of the two constants the correct value is written to EBX. When `CONST1` or `CONST2` is equal to zero, the last instruction can be deleted, since the correct value already has been written to EBX.

When `abs(CONST1-CONST2)` is one of {2,3,5,9}, the following example applies:

```
xor   ebx, ebx
cmp     A, B
setge bl       ; or the complement condition
lea   ebx, [ebx*D+ebx+CONST1-CONST2]
```

where D stands for `abs(CONST1-CONST2)-1`.

A second way to remove branches on Pentium Pro or Pentium II processors is to use the new CMOV and FCMOV instructions. Following is an example of changing a test and branch instruction sequence using CMOV and eliminating a branch. If the test sets the equal flag, the value in EBX will be moved to EAX. This branch is data dependent, and is representative of a unpredictable branch.

```
test ecx, ecx
jne  1h
mov  eax, ebx
1h:
```

To change the code, the jne and the mov instructions are combined into one CMOVcc instruction which checks the equal flag. The optimized code is shown below:

```
test      ecx, ecx        ; test the flags
cmoveq    eax, ebx        ; if the equal flag is set, move ebx to eax
1h:
```

The label 1h: is no longer needed unless it is the target of another branch instruction. These instructions will generate invalid opcodes when used on previous generation processors. Therefore, be sure to use the CPUID instruction to determine that the application is running on a Pentium Pro or Pentium II processor.

Additional information on branch elimination can be found on the Pentium Pro Processor Computer Based Training (CBT) which is available with VTune.

In addition to eliminating branches, the following guidelines improve branch predictability:

- Ensure that each call has a matching return.
- Don’t intermingle data and instructions.
- Unroll very short loops.
- Follow static prediction algorithm.

3.2.4 Performance Tuning Tip for Branch Prediction

3.2.4.1 PENTIUM® PROCESSOR FAMILY

On Pentium processors with and without MMX technology, the most common reason for pipeline flushes are BTB misses on taken branches or BTB mispredictions. If pipeline flushes are high, behavior of the branches in the application should be examined. Using VTune you can evaluate your program using the performance counters set to the following events.

1. Check total overhead because of pipeline flushes.

   Total overhead of pipeline flushes because of BTB misses is found by:

   Pipeline flushed due to wrong branch prediction * 4 / *Pipeline flushes due to wrong branch prediction in the WB stage*

NOTE

Because of the additional stage in the pipeline, the branch misprediction penalty for Pentium processors with MMX technology is one cycle more than the Pentium processor.

2. Check the BTB prediction rate.

   BTB hit rate is found by:

BTB Predictions / Branches

If the BTB hit rate is low, the number of active branches is greater than the number of BTB entries. Chapter 7 details monitoring of the above events.

3.2.4.2 PENTIUM® PRO AND PENTIUM II PROCESSORS

When a misprediction occurs the entire pipeline is flushed up to the branch instruction and the processor waits for the mispredicted branch to retire.

Branch Misprediction Ratio = BR_Miss_Pred_Ret / Br_Inst_Ret

If the branch misprediction ratio is less than about 5% then branch prediction is within normal range. Otherwise, identify which branches are causing significant mispredictions and try to remedy the situation using the techniques in Section 3.2.3.

3.3 PARTIAL REGISTER STALLS ON PENTIUM® PRO AND PENTIUM II PROCESSORS

On Pentium Pro and Pentium II processors, when a 32-bit register (for example, EAX) is read immediately after a 16- or 8-bit register (for example, AL, AH, AX) is written, the read is stalled until the write retires (a minimum of seven clock cycles). Consider the example below. The first instruction moves the value 8 into the AX register. The following instruction accesses the register EAX. This code sequence results in a partial register stall:

```
MOV AX, 8
ADD  ECX, EAX                    Partial stall occurs on access of
the EAX                          register
```

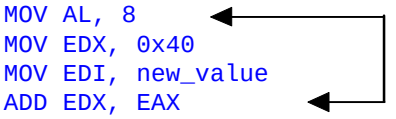
This applies to all of the 8- and 16-bit/32-bit register pairs, listed below:

Small Registers:			Large Registers:
AL	AH	AX	EAX
BL	BH	BX	EBX
CL	CH	CX	ECX
DL	DH	DX	EDX

SP	ESP
BP	EBP
DI	EDI
SI	ESI

Pentium processors do not exhibit this penalty.

Because Pentium Pro and Pentium II processors can execute code out of order, the instructions need not be immediately adjacent for the stall to occur. The following example also contains a partial stall:



```
MOV AL, 8
MOV EDX, 0x40
MOV EDI, new_value
ADD EDX, EAX                    Partial stall occurs on access of
the EAX                         register
```

In addition, any micro-ops that follow the stalled micro-op also wait until the clock cycle after the stalled micro-op continues through the pipe. In general, to avoid stalls, do not read a large (32-bit) register (EAX) after writing a small (16- or 18-bit) register (AL) which is contained in the large register.

Special cases of reading and writing small and large register pairs are implemented in Pentium Pro and Pentium II processors in order to simplify the blending of code across processor generations. The special cases are implemented for XOR and SUB when using EAX, EBX, ECX, EDX, EBP, ESP, EDI and ESI as shown in the following examples:

```
xor  eax, eax
movb  al, mem8
add  eax, mem32                 no partial stall

xor  eax, eax
movw  ax, mem16
add  eax, mem32                 no partial stall

sub  ax,  ax
movb al,  mem8
add  ax, mem16                  no partial stall

sub  eax, eax
movb  al, mem8
or   ax, mem16                  no partial stall

xor   ah, ah
movb  al, mem8
sub   ax, mem16                 no partial stall
```

In general, when implementing this sequence, always zero the large register and then write to the lower half of the register.

3.3.1 Performance Tuning Tip for Partial Stalls

3.3.1.1 PENTIUM® PROCESSORS

Partial stalls do not occur on the Pentium processor.

3.3.1.2 PENTIUM® PRO AND PENTIUM II PROCESSORS

Partial stalls are measured by the Renaming Stalls event in VTune. This event can be programmed as a duration event or a count event. Duration events count the total cycles the processor stalls for each event, where Count events count the total number of events. On VTune, you can set the `cmsk` for the Renaming Stalls event to be either count or duration in the Custom Events Window. The default is duration. By using the duration you can determine the percentage of time stalled by partial stalls with the following formula:

$$\frac{\textit{Renaming Stalls}}{\textit{Total Cycles}}$$

If a particular stall occurs more than about 3% of the execution time, then this stall should be re-coded to eliminate the stall.

3.4 ALIGNMENT RULES AND GUIDELINES

- The following section discusses guidelines for alignment of both code and data.

A misaligned access costs three cycles on the Pentium processor family. On Pentium Pro and Pentium II processors a misaligned access that crosses a cache line boundary costs six to nine cycles. A Data Cache Unit (DCU) split is a memory access which crosses a 32-byte line boundary. Unaligned accesses which cause a DCU split stall Pentium Pro and Pentium II processors. For best performance, make sure that in data structures and arrays greater than 32 bytes that the structure or array elements are 32-byte aligned, and that access patterns to data structure and array elements do not break the alignment rules.

3.4.1 Code

Pentium, Pentium Pro and Pentium II processors have a cache line size of 32 bytes. Since the prefetch buffers fetch on 16-byte boundaries, code alignment has a direct impact on prefetch buffer efficiency.

For optimal performance across the Intel Architecture family, it is recommended that:

- Loop entry labels should be 16-byte aligned when less than eight bytes away from a 16-byte boundary.
- Labels that follow a conditional branch should not be aligned.
- Labels that follow an unconditional branch or function call should be 16-byte aligned when less than eight bytes away from a 16-byte boundary.

On the Pentium processor with MMX technology, the Pentium Pro and Pentium II processors, avoid loops which execute in less than two cycles. Very tight loops have a high probability that one of the instructions will be split across a 16-byte boundary which causes extra cycles in the decoding of the instructions. On the Pentium processor this causes an extra cycle every other iteration. On the Pentium Pro and Pentium II processors it can limit the number of instructions available for execution which limits the number of instructions retired every cycle. It is recommended that critical loop entries be located on a cache line boundary. Additionally, loops that execute in less than two cycles should be unrolled. See Section 2.2 for more information about decoding on the Pentium Pro and Pentium II processors.

3.4.2 Data

A misaligned access in the data cache or on the bus costs at least three extra clock cycles on the Pentium processor. A misaligned access in the data cache, which crosses a cache line boundary, costs nine to twelve clock cycles on Pentium Pro and Pentium II processors. Intel recommends that data be aligned on the following boundaries for the best execution performance on all processors:

- Align 8-bit data on any boundary.
- Align 16-bit data to be contained within an aligned 4-byte word.
- Align 32-bit data on any boundary which is a multiple of four.
- Align 64-bit data on any boundary which is a multiple of eight.
- Align 80-bit data on a 128-bit boundary (that is, any boundary which is a multiple of 16 bytes).

3.4.2.1 DATA STRUCTURES AND ARRAYS GREATER THAN 32 BYTES

A 32-byte or greater data structure or array should be aligned so that the beginning of each structure or array element is aligned on a 32-byte boundary and so that each structure or array element does not cross a 32-byte cache line boundary.

3.4.3 Data Cache Unit (DCU) Split

The following example shows the type of code that can cause a DCU split. The code loads the addresses of two dword arrays. In this example, every four iterations of the first two dword loads causes a DCU split. The data declared at address 029e70feh is not 32-byte aligned, therefore each load to this address and every load that occurs 32 bytes (every four iterations) from this address will cross the cache line boundary, as illustrated in Figure 3-2 below.

```
                    mov     esi, 029e70feh
                    mov     edi, 05be5260h

BlockMove:
                    mov     eax, DWORD PTR [esi]
                    mov     ebx, DWORD PTR [esi+4]
                    mov     DWORD PTR [edi], eax
                    mov     DWORD PTR [edi+4], ebx
                    add     esi, 8
                    add     edi, 8
                    dec     edx
                    jnz     BlockMove
```

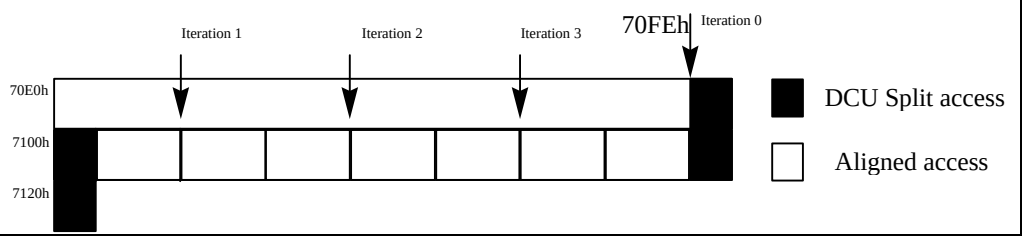


Figure 3-2. DCU Split in the Data Cache

3.4.4 Performance Tuning Tip for Misaligned Accesses

3.4.4.1 PENTIUM® PROCESSORS

Misaligned data causes a three-cycle stall on the Pentium processor. Use VTune dynamic execution functionality to determine the exact location of a misaligned access.

3.4.4.2 PENTIUM® PRO AND PENTIUM II PROCESSORS

Misaligned data can be detected by using the Misaligned Accesses event counter on Pentium Pro processors. When the misaligned data crosses a cache line boundary it causes a six to twelve-cycle stall.

3.5 DATA ARRANGEMENT FOR IMPROVED CACHE PERFORMANCE

Cache behavior can dramatically affect the performance of your application. By having a good understanding of how the cache works, you can structure your code and data to take best advantage of cache capabilities. Cache structure information for each of the processors is discussed in Chapter 2.

3.5.1 C-Language Level Optimizations

The following sections discuss how you can improve the arrangement of data at the C-language level. These optimizations can benefit all processors.

3.5.1.1 DECLARATION OF DATA TO IMPROVE CACHE PERFORMANCE

Compilers generally control allocation of variables, and the developer cannot control how variables are arranged in memory after optimization. Specifically, compilers allocate structure and array values in memory in the order the values are declared as required by language standards. However, when in-line assembly is inserted in a function, many compilers turn off optimization, and the way you declare data in this function becomes important. Additionally, order of data declaration is important when declaring your data at the assembly level. Sometimes a DCU split or unaligned data can be avoided by changing the data layout in the high level or assembly code, consider the following example:

Unoptimized data layout:

```
short a[15];          /* 2 bytes data */
int   b[15], c[15];   /* 4 bytes data */

for (i=0; i<15, i++) {
     a[i] = b[i] + c[i]
}
```

In some compilers the memory is allocated as the variables are declared, therefore the cache layout of the variables in the example above is as follows:

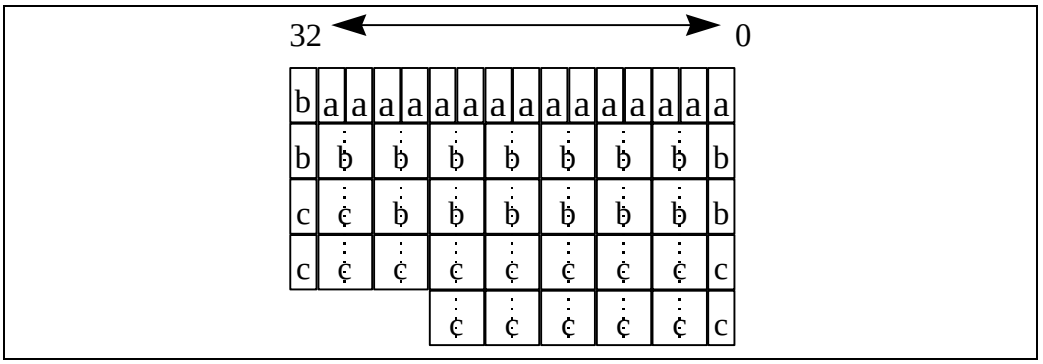


Figure 3-3. Cache Layout of Structures a, b and c

This example assumes that `a[0]` is aligned at a cache line boundary. Each box represents two bytes. Accessing elements `b[0]`, `b[8]`, `c[1]`, `c[9]` will cause DCU splits on the Pentium Pro processor.

Rearrange the data so that the larger elements are declared first, thereby avoiding the misalignment.

Optimized data layout:

```
int   b[15], c[15]; /* 4 bytes data */
short a[15];        /* 2 bytes data */

for (i=0; i<15, i++) {
     a[i] = b[i] + c[i]
}
```

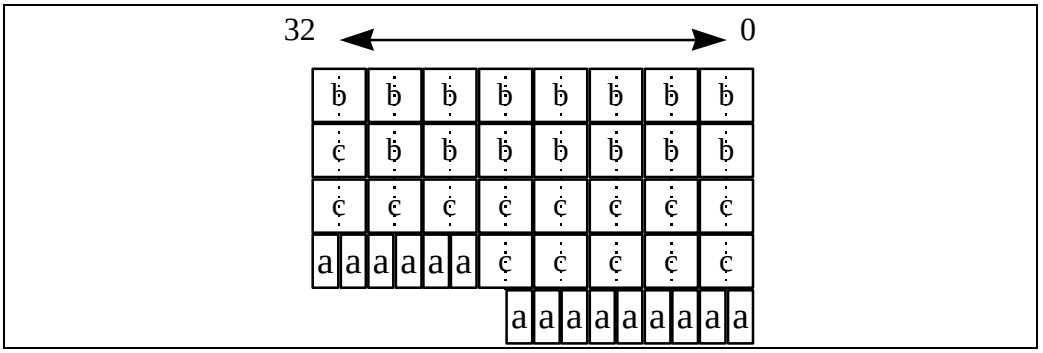


Figure 3-4. Optimized Data Layout of Structures a, b and c

Accessing the above data will not cause a DCU split on Pentium Pro and Pentium II processors.

3.5.1.2 DATA STRUCTURE DECLARATION

Data structure declaration can be very important to the speed of accessing data in structures. The following section discusses easy ways to improve access in your C code.

It is best to have your data structure use as little space as possible. This can be accomplished by always using the following guidelines when declaring arrays and structures:

- Make sure the data structure begins 32-byte aligned.
- Arrange data so an individual structure element does not cross a cache line boundary.
- Declare elements largest to smallest.
- Place together data elements that are accessed at the same time.
- Place together data elements that are used frequently.

How you declare large arrays within a structure is dependent upon how the arrays are accessed in the code. The array could be declared as a structure of two separate arrays, or as a compound array of structures, as shown in the following code segments:

Separate Array:

```
struct    {
     int a[500];
     int b[500];
     } s;
```

Compound Array:

```
struct    {
     int a;
     int b;
     } s[500];
```

Using separate arrays the elements of array `a` are located sequentially in memory followed by the elements of array `b`. In the compound array, the elements of each array are alternated so that for every iteration, `b[i]` is located after `a[i]`, as shown in Figure 3-5 below:

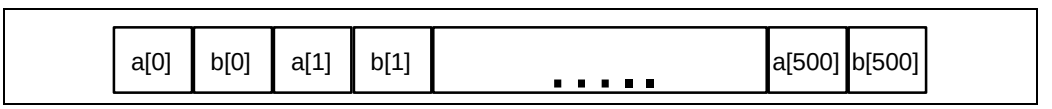


Figure 3-5. Compound Array as Stored in Memory

If your code accesses arrays `a` and `b` sequentially, declare the arrays separately. This way, a cache line fill that brings in element `a[i]` into the cache also brings in the adjacent elements of the array into the cache. If your code accesses arrays `a` and `b` in parallel, use the compound array structure declaration. Then a cache line fill that brings in element `a[i]` into the cache also brings in element `b[i]`.

3.5.1.4 PADDING AND ALIGNMENT OF ARRAYS AND STRUCTURES

Padding and aligning of arrays and structures can help avoid cache misses when structures or arrays are randomly accessed. Structures or arrays that are sequentially accessed should be sequential in memory.

Use the following guidelines to reduce cache misses:

- Pad each structure to make its size equal to an integer multiple of the cache line size.
- Align each structure so it starts at the beginning of a cache line (a multiple of 32 for the Pentium and Pentium Pro processors.
- Make array dimensions be powers of two.

For more information and examples of these techniques see the Pentium processor computer based training.

3.5.1.5 LOOP TRANSFORMATIONS FOR MEMORY PERFORMANCE

In addition to the way data is structured in memory, you can also improve cache performance by improving the way the code accesses the data. Following are a few principal transformations for improving memory access patterns. The goal is to make the references in the inner loop have unit strides and to keep as much as possible of the computation executing from within the caches.

"Loop fusion" is a transformation that combines two loops that access the same data so that more work can be completed on the data while it is in the cache.

Before:

```
for (i = 1; i < n){
  ... A(i) ...
}
for ( i = 1; i < n){
  ... A(i) ...
}
```

After:

```
for (i = 1; i < n{
     ... A(i) ...
     ... A(i) ...
}
```

"Loop fission" is a transformation that splits a loop into two loops so that the data brought into the cache is not flushed from the cache before the work is completed.

Before:

```
for (i = 1; i < n){
      ... A(i) ...
      ... B(i) ...
}
```

After:

```
for (i = 1; i < n){
     ... A(i) ...
}
for (i = 1; i < n){
     ... B(i) ...
}
```

Loop interchanging changes the way the data is accessed. C compilers store matrices in memory, in row order, where FORTRAN compilers store matrices in column order. By accessing the data as it is stored in memory, you can avoid many cache misses and improve performance.

Before:

```
for (i = 1; i < n){
     for (j = 1; j < n){
          ... A(i,j) ...
     }
}
```

After:

```
for (j = 1; j < n){
     for (i = 1; i < n){
          ... A(i,j) ...
     }
}
```

Blocking is the process of restructuring your program so data is accessed with a few cache misses as possible. Blocking is useful for multiplying very large matrixes.

Before:

```
for (j = 1; j < n){
     for (i = 1; i < n){

       ... A(i,j) ...

     }
     for ( i = ii; i < ii+k) {
}
          ... A(i,j) ...
```

After:

```
     for (jj = 1; i < n jj+= k){
          for (ii = 1; ii < n; ii+=k){

               for (j = jj; i < jj+k){

               }
          }
     }
}
```

NOTE

Some of these transformations may not be legal for some programs. Some algorithms may produce different results when these transformations are applied. Additional information on these optimization techniques can be found in *High Performance Computing, Kevin Dowd, O'Reilly and Associates, Inc., 1993* and *High Performance Compilers for Parallel Computing, Michael Wolfe, Addison Wesley Publishing Company, Inc., 1996, ISBN 0-8053-2730-4*.

3.5.1.6 ALIGNING DATA IN MEMORY AND ON THE STACK

Accessing 64-bit variables that are not 8-byte aligned costs an extra three cycles on the Pentium processor. When such a variable crosses a 32-byte cache line boundary it can cause a DCU split in Pentium Pro and Pentium II processors. Some commercial compilers do not align doubles on 8-byte boundaries. If, by using the Misaligned Accesses performance counter, you discover your data is not aligned, the following methods may be used to align your data:

- Use static variables.
- Use assembly code that explicitly aligns data.
- In C code, use malloc to explicitly allocate variables.

Static Variables

When variables are allocated on the stack they may not be aligned. Compilers do not allocate static variables on the stack, but in memory. In most cases when the compiler allocates static variables, they are aligned.

```
static float a;                    float b;
static float c;
```

Alignment using Assembly Language

Use assembly code to explicitly align variables. The following example aligns the stack to 64-bits:

Procedure Prologue:

```
push  ebp
mov   esp, ebp
and   ebp, -8
sub   esp, 12
```

Procedure Epilogue:

```
add  esp, 12
pop  ebp
ret
```

Dynamic Allocation Using Malloc

If you use dynamic allocation, check if your compiler aligns double or quadword values on 8-byte boundaries. If the compiler does not align doubles to 8 bytes, then

- Allocate memory equal to the size of the array or structure plus an extra 4 bytes.
- Use bitwise AND to make sure that the array is aligned.

Example:

```
double a[5];
double *p, *newp;

p = (double*)malloc ((sizeof(double)*5)+4)
newp = (p+4) & (-7)
```

3.5.2 Moving Large Blocks of Memory

When copying large blocks of data on the Pentium Pro and Pentium II processors, you can improve the speed of the copy by enabling the advanced features of the processor. In order to use the special mode your data copy must meet the following criteria:

- The source and destination must be 8 byte aligned
- The copy direction must be ascending
- The length of the data must require greater than 64 repetitions

When all three of these criteria are true, programming a function using the rep movs and rep stos instructions instead of a library function will allow the processor to perform a fast string copy. Additionally, when your application spends a large amount of time copying you can improve overall speed of your application by setting up your data to match these criteria.

Following is an example for copying a page:

```
MOV ECX, 4096       ; instruction sequence for copying a page
MOV EDI, destpageptr ; 8-byte aligned page pointer
MOV ESI, srcpageptr  ; 8-byte aligned page pointer
REP MOVSB
```

3.5.3 Line Fill Order

When a data access to a cacheable address misses the data cache, the entire cache line is brought into the cache from external memory. This is called a line fill. On Pentium, Pentium Pro and Pentium II processors, these data arrive in a burst composed of four 8-byte sections in the following burst order:

1st Address	2nd Address	3rd Address	4th Address
0h	8h	10h	18h
8h	0h	18h	10h
10h	18h	0h	8h
18h	10h	8h	0h

For Pentium processors with MMX technology, Pentium Pro and Pentium II processors, data is available for use in the order that it arrives from memory. If an array of data is being read serially, it is preferable to access it in sequential order so that each data item will be used as it arrives from memory. On Pentium processors the first 8-byte section is available immediately, but the rest of the cache line is not available until the entire line is read from memory.

Arrays with a size that is a multiple of 32 bytes should start at the beginning of a cache line. By aligning on a 32-byte boundary, you take advantage of the line fill ordering and match the cache line size. Arrays with sizes that are not multiples of 32 bytes should begin at 32- or 16-byte boundaries (the beginning or middle of a cache line). In order to align on a 16- or 32-byte boundary, you may need to pad the data. If this is necessary, try to locate data (variables or constants) in the padded space.

3.5.4 Increasing Bandwidth of Memory Fills

It is beneficial to understand how memory is accessed and filled. A memory-to-memory fill (for example a memory-to-video fill) is defined as a 32-byte (cache line) load from memory which is immediately stored back to memory (such as a video frame buffer). The following are guidelines for obtaining higher bandwidth and shorter latencies for sequential memory fills (video fills). These recommendations are relevant for all Intel Architecture processors and refer to cases in which the loads and stores do not hit in the second level cache. See Chapter 4 for more information on memory bandwidth.

3.5.5 Write Allocation Effects

Pentium Pro and Pentium II processors have a “write allocate by read-for-ownership” cache, whereas the Pentium processor has a “no-write-allocate; write through on write miss” cache.

On Pentium Pro and Pentium II processors, when a write occurs and the write misses the cache, the entire 32-byte cache line is fetched. On the Pentium processor, when the same write miss occurs, the write is simply sent out to memory.

Write allocate is generally advantageous, since sequential stores are merged into burst writes and the data remains in the cache for use by later loads. This is why P6- family processors adopted this write strategy, and why some Pentium processor system designs implement it for the L2 cache, even though the Pentium processor uses write-through on a write miss.

Write allocate can be a disadvantage in code where:

- Just one piece of a cache line is written.
- The entire cache line is not read.
- Strides are larger than the 32-byte cache line.
- Writes are made to a large number of addresses (>8000).

When a large number of writes occur within an application, as in the example program below, and both the stride is longer than the 32-byte cache line and the array is large, every store on a Pentium Pro or Pentium II processor will cause an entire cache line to be fetched. In addition, this fetch will probably replace one (sometimes two) dirty cache line.

The result is that every store causes an additional cache line fetch and slows down the execution of the program. When many writes occur in a program, the performance decrease can be significant. The Sieve of Erastothenes program demonstrates these cache effects. In this example, a large array is stepped through in increasing strides while writing a single value of the array with zero.

NOTE

This is a very simplistic example used only to demonstrate cache effects; many other optimizations are possible in this code.

Sieve of Erastothenes example:

```
boolean array[max];
for(i=2;i<max;i++) {
     array = 1;
    }

for(i=2;i<max;i++) {
     if( array[i] ) {
         for(j=2;j<max;j+=i) {
                    array[j] = 0;  /*here we assign memory to 0
causing                                                    the
cache line fetch within the j
                    loop */
         }
     }
}
```

Two optimizations are available for this specific example. One is to pack the array into bits, thereby reducing the size of the array, which in turn reduces the number of cache line fetches. The second is to check the value prior to writing, thereby reducing the number of writes to memory (dirty cache lines).

3.5.5.1 OPTIMIZATION 1: BOOLEAN

In the program above, `boolean` is a char array. It may well be better, in some programs, to make the `boolean` array into an array of bits, packed so that read-modify-writes are done (since the cache protocol makes every read into a read-modify-write). But in this example, the vast majority of strides are greater than 256 bits (one cache line of bits), so the performance increase is not significant.

3.5.5.2 OPTIMIZATION 2: CHECK BEFORE WRITING

Another optimization is to check if the value is already zero before writing.

```
boolean array[max];
for(i=2;i<max;i++) {
     array = 1;
    }

    for(i=2;i<max;i++) {
     if( array[i] ) {
         for(j=2;j<max;j+=i) {
          if( array[j] != 0 ) {  /* check to see if value is

       already 0 */
              array[j] = 0;
          }
         }
     }
    }
```

The external bus activity is reduced by half because most of the time in the Sieve program the data is already zero. By checking first, you need only one burst bus cycle for the read and you save the burst bus cycle for every line you do not write. The actual write back of the modified line is no longer needed, therefore saving the extra cycles.

NOTE

This operation benefits Pentium Pro and Pentium II processors but may not enhance the performance of Pentium processors. As such, it should not be considered generic. Write allocate is generally a performance advantage in most systems, since sequential stores are merged into burst writes and the data remain in the cache for use by later loads. This is why Pentium Pro and Pentium II processors use this strategy, and why some Pentium processor-based systems implement it for the L2 cache.

3.6 INTEGER INSTRUCTION SCHEDULING

Scheduling or pipelining should be done in a way that optimizes performance across all processor generations. The following is a list of pairing and scheduling rules that can improve the speed of your code on Pentium, Pentium Pro and Pentium II processors. In some cases, there are tradeoffs involved in reaching optimal performance on a specific processor; these tradeoffs vary based on the specific characteristics of the application.

On superscalar Pentium processors, the order of instructions is very important to achieving maximum performance. Reordering instructions increases the possibility of issuing two instructions simultaneously. Instructions that have data dependencies should be separated by at least one other instruction.

3.6.1 Pairing

This section describes the rules you need to follow to pair integer instructions. Pairing rules for MMX instructions and floating-point instructions are in Chapters 4 and 5 respectively.

- Several types of rules must be observed to allow pairing:
- Integer pairing rules: Rules for pairing integer instructions.
- General pairing rules: Rules which depend on the machine status and do not depend on the specific opcodes. They are also valid for integer and FP. For example, single-step should be disabled to allow instruction pairing
- MMX instruction pairing rules for a pair of MMX instructions: Rules that allow two MMX instructions to pair. Example: the processor cannot issue two MMX instructions simultaneously because only one multiplier unit exists. See Section 4.3.
- MMX and integer instruction pairing rules: Rules that allow pairing of one integer and one MMX instruction. See Section 4.3.
- Floating-point and integer pairing rules: See Section 5.3.

NOTE

Floating-point instructions are not pairable with MMX instructions.

3.6.2 Integer Pairing Rules

Pairing cannot be performed when the following conditions occur:

- The next two instructions are not pairable instructions (see Appendix A for pairing characteristics of individual instructions). In general, most simple ALU instructions are pairable.
- The next two instructions have some type of register contention (implicit or explicit). There are some special exceptions to this rule where register contention can occur with pairing. These are described later.

- The instructions are not both in the instruction cache. An exception to this which permits pairing is if the first instruction is a one-byte instruction.

Table 3-1. Integer Instruction Pairing

Integer Instruction Pairable in U-Pipe			Integer Instruction Pairable in V-Pipe		
mov r, r	alu r, i	push r	mov r, r	alu r, i	push r
mov r, m	alu m, i	push i	mov r, m	alu m, i	push I
mov m, r	alu eax, i	pop r	mov m, r	alu eax, i	pop r
mov r, i	alu m, r	nop	mov r, i	alu m, r	jmp near
mov m, i	alu r, m	shift/rot by 1	mov m, i	alu r, m	jcc near
mov eax, m	inc/dec r	shift by imm	mov eax, m	inc/dec r	0F jcc
mov m, eax	inc/dec m	test reg, r/m	mov m, eax	inc/dec m	call near
alu r, r	lea r, m	test acc, imm	alu r, r	lea r, m	nop
				test reg, r/m	test acc, imm

3.6.2.1 INSTRUCTION SET PAIRABILITY

Unpairable Instructions (NP)

1. Shift or rotate instructions with the shift count in the CL register.
2. Long arithmetic instructions, for example: `MUL`, `DIV`.
3. Extended instructions, for example: `RET`, `ENTER`, `PUSHA`, `MOVS`, `STOS`, `LOOPNZ`.
4. Some floating-point instructions, for example: `FSCALE`, `FLDCW`, `FST`.
5. Inter-segment instructions, for example: `PUSH`, `sreg`, `CALL far`.

Pairable Instructions Issued to U or V Pipes (UV)

1. Most 8/32 bit ALU operations, for example: `ADD`, `INC`, `XOR`.
2. All 8/32 bit compare instructions, for example: `CMP`, `TEST`.
3. All 8/32 bit stack operations using registers, for example: `PUSH reg`, `POP reg`.

Pairable Instructions Issued to U Pipe (PU)

These instructions must be issued to the U-pipe and can pair with a suitable instruction in the V-Pipe. These instructions never execute in the V-pipe.

1. Carry and borrow instructions, for example: `ADC`, `SBB`.
2. Prefixed instructions (see next section).

3. Shift with immediate.
4. Some floating-point operations, for example: FADD, FMUL, FLD.

Pairable Instructions Issued to V Pipe (PV)

These instructions can execute in either the U-pipe or the V-pipe but they are only paired when they are in the V-pipe. Since these instructions change the instruction pointer (eip), they cannot pair in the U-pipe since the next instruction may not be adjacent. Even when a branch in the U-pipe is predicted to be not taken, it will not pair with the following instruction.

1. Simple control transfer instructions, for example: call near, jmp near, jcc. This includes both the jcc short and the jcc near (which has a 0f prefix) versions of the conditional jump instructions.
2. The fxch instruction.

3.6.2.2 UNPAIRABILITY DUE TO REGISTER DEPENDENCIES

Instruction pairing is also affected by instruction operands. The following combinations cannot be paired because of register contention. Exceptions to these rules are given in the next section.

1. The first instruction writes to a register that the second one reads from (flow dependence). An example follows:

```
mov  eax, 8
mov  [ebp], eax
```

2. Both instructions write to the same register (output dependence), as shown below:

```
mov  eax, 8
mov  eax, [ebp]
```

This limitation does not apply to a pair of instructions that write to the EFLAGS register (for example, two ALU operations that change the condition codes). The condition code after the paired instructions execute will have the condition from the V-pipe instruction.

Note that a pair of instructions in which the first reads a register and the second writes to the same register (anti-dependence), may be paired. See the following example:

```
mov  eax, ebx
mov  ebx, [ebp]
```

For purposes of determining register contention, a reference to a byte or word register is treated as a reference to the containing 32-bit register. Therefore:

```
mov  al, 1
mov  ah, 0
```

Do not pair because of output dependencies on the contents of the EAX register.

3.6.2.3 SPECIAL PAIRS

There are some instructions that can be paired in spite of our general rules above. These special pairs overcome register dependencies, and most involve implicit reads/writes to the esp register or implicit writes to the condition codes:

Stack Pointer:

- `push reg/imm; push reg/imm`
- `push reg/imm; call`
- `pop reg ; pop reg`

Condition Codes:

- `cmp ; jcc`
- `add ; jne`

Note that the special pairs that consist of PUSH/POP instructions can have only immediate or register operands, not memory operands.

3.6.2.4 RESTRICTIONS ON PAIR EXECUTION

There are some pairs that may be issued simultaneously but will not execute in parallel:

1. If both instructions access the same data-cache memory bank then the second request (V-pipe) must wait for the first request to complete. A bank conflict occurs when bits 2 through 4 are the same in the two physical addresses. A bank conflict incurs a one clock penalty on the V-pipe instruction.
2. Inter-pipe concurrency in execution preserves memory-access ordering. A multi-cycle instruction in the U-pipe will execute alone until its last memory access.

```
add  eax, meml
add  ebx, mem2            ; 1
(add)   (add)             ; 2  2-cycle
```

The instructions above add the contents of the register and the value at the memory location, then put the result in the register. An add with a memory operand takes two clocks to execute. The first clock loads the value from cache, and the second clock performs the addition. Since there is only one memory access in the U-pipe instruction, the add in the V-pipe can start in the same cycle.

```
add  meml, eax                ; 1
(add)                              ; 2
(add)add  mem2, ebx  ; 3
(add)                              ; 4
(add)                              ; 5
```

The instructions above add the contents of the register to the memory location and store the result at the memory location. An add with a memory result takes three clocks to execute. The first clock loads the value, the second performs the addition, and the third stores the

result. When paired, the last cycle of the U-pipe instruction overlaps with the first cycle of the V-pipe instruction execution.

No other instructions can begin execution until the instructions already executing have completed.

To expose the opportunities for scheduling and pairing, it is better to issue a sequence of simple instructions rather than a complex instruction that takes the same number of cycles. The simple instruction sequence can take advantage of more issue slots. The load/store style of code generation requires more registers and increases code size. This impacts Intel486 processor performance, although only as a second-order effect. To compensate for the extra registers needed, extra effort should be put into register allocation and instruction scheduling so that extra registers are used only when parallelism increases.

3.6.3 General Pairing Rules

Pentium processors with MMX technology have relaxed some of the general pairing rules:

- Pentium processors do not pair two instructions if either of them is longer than seven bytes. Pentium processors with MMX technology do not pair two instructions if the first instruction is longer than eleven bytes or the second instruction is longer than seven bytes. Prefixes are not counted.
- On Pentium processors, prefixed instructions are pairable only in the U-pipe. On Pentium processors with MMX technology, instructions with 0Fh, 66H or 67H prefixes are also pairable in the V-pipe.

In both of the above cases, stalls at the entrance to the FIFO, on Pentium processors with MMX technology, will prevent pairing.

3.6.4 Scheduling Rules for Pentium® Pro and Pentium II Processors

Pentium Pro and Pentium II processors have three decoders that translate Intel Architecture (IA) macro-instructions into micro-operations (µops) as discussed in Section 2.2. The decoder limitations are as follows:

- The first decoder (0) can decode instructions with
  - — Up to 4 micro-ops.
  - — Up to seven bytes in length.
- The other two decoders decode instructions that are 1 µop.

Appendix C contains a table of all Intel macro-instructions with a the number of µops into which they are decoded. Use this information to determine the decoder on which they can be decoded.

The macro-instructions entering the decoder travel through the pipe in order, therefore if a macro-instruction will not fit in the next available decoder, the instruction must wait until the next cycle to be decoded. It is possible to schedule instructions for the decoder so that the instructions in the in-order pipeline are less likely to be stalled.

Consider the following examples:

- If the next available decoder for a multi-µop instruction is not decoder 0, the multi-op instruction will wait for decoder 0 to be available, usually in the next clock, leaving the other decoders empty during the current clock. Hence, the following two instructions will take two cycles to decode.

```
add  eax, ecx        ; 1 uop instruction  (decoder 0)
add  edx, [ebx]      ; 2 uop instruction (stall 1 cycle wait
                     ;   till decoder 0 is available)
```

- During the beginning of the decoding cycle, if two consecutive instructions are more than 1 µop, decoder 0 will decode one instruction and the next instruction will not be decoded until the next cycle.

```
add  eax, [ebx]      ; 2 uop instruction (decoder 0)
mov  ecx, [eax]      ; 2 uop instruction  (stall 1 cycle to wait
                     ;   until decoder 0 is available)
add  ebx, 8          ; 1 uop instruction  (decoder 1)
```

Instructions of the `op reg, mem` form require two µops: the load from memory and the operation µop. Scheduling for the decoder template (4-1-1) can improve the decoding throughput of your application.

In general, `op reg, mem` forms of instructions are used to reduce register pressure in code that is not memory bound, and when the data is in the cache. Use simple instructions for improved speed on both Pentium and Pentium Pro and Pentium II processors.

The following rules should be observed while using the `op reg, mem` instruction on Pentium processors with MMX technology:

- Schedule for minimal stalls in the Pentium processor pipe. Use as many simple instructions as possible. Generally, 32-bit assembly code that is well optimized for the Pentium processor pipeline will execute well on Pentium Pro and Pentium II processors.
- When scheduling for Pentium processors, keep in mind the primary stall conditions and decoder template (4-1-1) on Pentium Pro and Pentium II processors, as shown in the example below:

```
pmaddwd  mm6, [ebx]       ; 2 uops instruction (decoder 0)
paddd    mm7, mm6              ; 1 uop instruction  (decoder 1)
add      ebx, 8                ; 1 uop instruction  (decoder 2)
```

3.7 PREFIXED OPCODES

On the Pentium processor, an instruction with a prefix is pairable in the U-pipe (PU) if the instruction without the prefix is pairable in both pipes (UV) or in the U-pipe (PU). The prefixes are issued to the U-pipe and get decoded in one cycle for each prefix, then the instruction is issued to the U-pipe and may be paired.

For Pentium, Pentium Pro and Pentium II processors, prefixes that should be avoided are:

- lock ument is pro
- segment override ith Intel pro
- address size e, express o
- operand size e,6express o
- two-byte opcode map (0f) prefix

On Pentium processors with MMX technology, a prefix on an instruction can delay the parsing and inhibit pairing of instructions.

The following list highlights the effects of instruction prefixes on the FIFO:

- There is no penalty on 0F-prefix instructions.
- An instruction with a 66h or 67h prefix takes one clock for prefix detection, another clock for length calculation, and another clock to enter the FIFO (three clock cycles total). It must be the first instruction to enter the FIFO, and a second instruction can be pushed with it.
- Instructions with other prefixes (not 0Fh, 66h or 67h) take one additional clock cycle to detect each prefix. These instructions are pushed into the FIFO only as the first instruction. An instruction with two prefixes takes three clock cycles to be pushed into the FIFO (two clock cycles for the prefixes and one clock cycle for the instruction). A second instruction can be pushed with the first into the FIFO in the same clock cycle.

Performance is impacted only when the FIFO does not hold at least two entries. As long as the decoder (D1 stage) has two instructions to decode there is no penalty. The FIFO will quickly become empty if the instructions are pulled from the FIFO at the rate of two per clock cycle. So if the instructions just before the prefixed instruction suffer from a performance loss (for example, no pairing, stalls due to cache misses, misalignments, etc.), the performance penalty of the prefixed instruction may be masked.

On Pentium Pro and Pentium II processors, instructions longer than seven bytes limit the number of instructions decoded in each cycle (see Section 2.2). Prefixes add one to two bytes to the length of an instruction, possibly limiting the decoder.

It is recommended that, whenever possible, prefixed instructions not be used or that they be scheduled behind instructions which themselves stall the pipe for some other reason.

3.8 ADDRESSING MODES

On the Pentium processor, when a register is used as the base component, an additional clock cycle is used if that register is the destination of the immediately preceding instruction (assuming all instructions are already in the prefetch queue). For example:

```
add  esi, eax    ; esi is destination register
mov  eax, [esi]  ; esi is base, 1 clock penalty
```

1. Since the Pentium processor has two integer pipelines, a register used as the base or index component of an effective address calculation (in either pipe) causes an additional clock cycle if that register is the destination of either instruction from the immediately preceding clock cycle. This effect is known as Address Generation Interlock (AGI). To avoid AGI, such instructions should be separated by at least one cycle by placing other instructions between them. The MMX registers cannot be used as base or index registers, so the AGI does not apply for MMX register destinations.

2. Pentium Pro and Pentium II processors incur no penalty for the AGI condition.

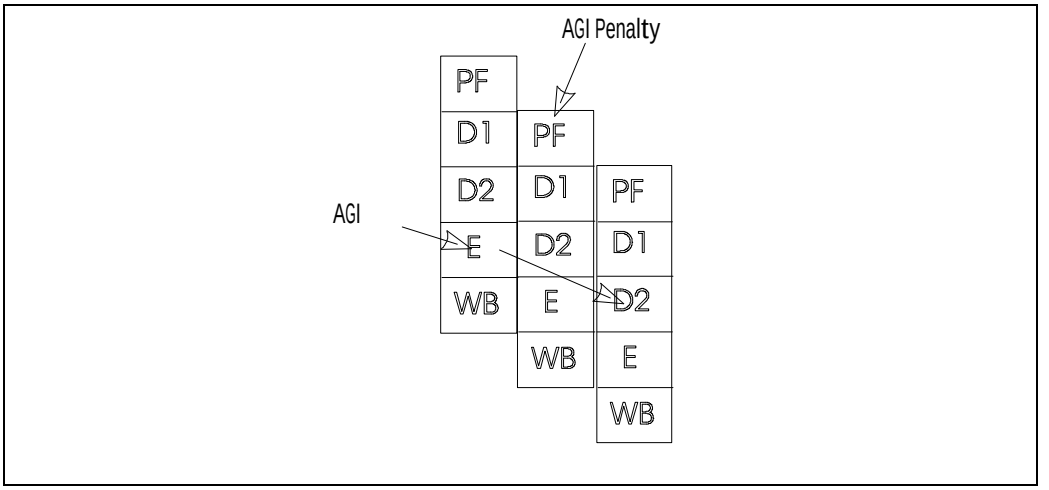


Figure 3-6. Pipeline Example of AGI Stall

Note that some instructions have implicit reads/writes to registers. Instructions that generate addresses implicitly through ESP (PUSH, POP, RET, CALL) also suffer from the AGI penalty. Examples follow:

```
sub  esp, 24
                          ;  1 clock cycle stall
push ebx
mov  esp, ebp
                          ;1  clock cycle stall
pop  ebp
```

PUSH and POP also implicitly write to ESP. This, however, does not cause an AGI when the next instruction addresses through ESP. Pentium processors "rename" ESP from PUSH and POP instructions to avoid the AGI penalty. An example follows:

```
push edi        ; no stall
mov  ebx, [esp]
```

On Pentium processors, instructions which include both an immediate *and* displacement fields are pairable in the U-pipe. When it is necessary to use constants, it is usually more efficient to use immediate data instead of loading the constant into a register first. If the same immediate data is used more than once, however, it is faster to load the constant in a register and then use the register multiple times. Following is an example:

```
mov result, 555             ; 555 is immediate, result is
displacement
mov word ptr [esp+4], 1        ; 1 is immediate,4 is
displacement
```

Since MMX instructions have two-byte opcodes (0x0F opcode map), any MMX instruction that uses base or index addressing with a 4-byte displacement to access memory will have a length of eight bytes. Instructions over seven bytes can limit decoding and should be avoided where possible (see Section 3.6.4). It is often possible to reduce the size of such instructions by adding the immediate value to the value in the base or index register, thus removing the immediate field.

Pentium Pro and Pentium II processors incur a stall when using a full register immediately after a partial register was written. This is called a partial stall condition. The Pentium processor is neutral in this respect. The following example relates to the Pentium processor:

```
mov  al, 0          ; 1
mov  [ebp], eax     ; 2 - No delay on the Pentium processor
```

The following example relates to the Pentium Pro and Pentium II processor:

```
mov  al, 0          ; 1
mov  [ebp], eax     ; 3  PARTIAL REGISTER STALL
```

The read is stalled until the partial write retires, which is estimated to be a minimum of seven clock cycles.

For best performance, avoid using a large register (for example, EAX) after writing a partial register (for example, AL, AH, AX) which is contained in the large register. This guideline prevents partial stall conditions on Pentium Pro and Pentium II processors and applies to all of the small and large register pairs, as shown below:

```
AL   AH   AX   EAX
BL   BH   BX   EBX
CL   CH   CX   ECX
DL   DH   DX   EDX
          SP   ESP
          EP   EBP
          SI   ESI
          DI   EDI
```

Additional information on partial register stalls is in Section 3.3.

3.8.1 Performance Tuning Tip for AGI

3.8.1.1 PENTIUM® PROCESSOR

Monitor for the event Pipeline stalled because of Address Generation Interlock. This is the number of pipe stalls because of an address being not yet available, which can be decreased by keeping at least a clock between the computation of an address and the use of the address.

3.8.1.2 PENTIUM® PRO AND PENTIUM II PROCESSORS

These processors incur no penalty for the AGI condition.

3.9 INSTRUCTION LENGTH

On Pentium processors, instructions greater than seven bytes in length cannot be executed in the V-pipe. In addition, two instructions cannot be pushed into the instruction FIFO on Pentium Processors with MMX technology (see Section 2.3.1) unless both are seven bytes or less in length. If only one instruction is pushed into the FIFO, pairing does not occur unless the FIFO already contains at least one instruction. In code where pairing is very high (this often happens in MMX code) or after a mispredicted branch, the FIFO may be empty, leading to a loss of pairing whenever the instruction length is over seven bytes.

In addition, Pentium Pro and Pentium II processors can only decode one instruction at a time when an instruction is longer than seven bytes.

So for best performance on all Intel processors, use simple instructions that are less than eight bytes in length.

3.10 INTEGER INSTRUCTION SELECTION

The following list highlights some instruction sequences to avoid and some sequences to use when generating optimal assembly code. These apply to Pentium, Pentium Pro and Pentium II processors.

1. The `lea` instruction can be used sometimes as a three/four operand addition instruction `(e.g., lea ecx, [eax+ebx+4+a])`
2. In many cases an `lea` instruction or a sequence of `lea`, `add`, `sub` and `shift` instructions can be used to replace constant multiply instructions. For Pentium Pro and Pentium II processors the constant multiply is faster relative to other instructions than on the Pentium processor, therefore the tradeoff between the two options occurs sooner. It is recommended that the integer multiply instruction be used in code designed for Pentium Pro and Pentium II processors..

3. This technique can also be used to avoid copying a register when both operands to an add are still needed after the add, since `lea` need not overwrite its operands.

The disadvantage of the `lea` instruction is that it increases the possibility of an AGI stall with previous instructions. `Lea` is useful for shifts of 2,4,8 because on the Pentium processor, `lea` can execute in either U or V-pipes, but `shift` can only execute in the U-pipe. On the Pentium Pro processor, both `lea` and `shift` instructions are single µop instructions that execute in one cycle.

Complex Instructions

Avoid using complex instructions (for example, `enter`, `leave`, `loop`). Use sequences of simple instructions instead.

Zero-Extension of Short

On the Pentium processor, the `movzx` instruction has a prefix and takes three cycles to execute, totaling 4 cycles. It is recommended that the following sequence be used instead:

```
xor  eax, eax
mov  al, mem
```

If this occurs within a loop, it may be possible to pull the `xor` out of the loop if the only assignment to `eax` is the `mov al, mem`. This has greater importance for the Pentium processor since the `movzx` is not pairable and the new sequence can be paired with adjacent instructions.

In order to avoid a partial register stall on Pentium Pro and Pentium II processors, special hardware has been implemented that allows this code sequence to execute without a stall. Even so, the `movzx` instructions is better on Pentium Pro and Pentium II processors than the alternative sequences. (See Section 3.3 for additional partial stall information.)

Push mem

The `push mem` instruction takes four cycles for the Intel486 processor. It is recommended to use the following sequence because it takes only two cycles for the Intel486 processor and increases pairing opportunity for the Pentium processor.

```
mov  reg, mem
push reg
```

Short Opcodes

Use one-byte instructions as much as possible. This will reduce code size and help increase instruction density in the instruction cache. The most common example is using `inc` and `dec` rather than adding or subtracting the constant 1 with `add` or `sub`. Another common example is using the `push` and `pop` instructions instead of the equivalent sequence.

8/16 bit Operands

With 8-bit operands, try to use the byte opcodes, rather than using 32-bit operations on sign and zero extended bytes. Prefixes for operand size override apply to 16-bit operands, not to 8-bit operands.

Sign extension is usually quite expensive. Often, the semantics can be maintained by zero extending 16-bit operands. Specifically, the C code in the following example does not need sign extension nor does it need prefixes for operand size overrides.

```
static short int a, b;
if (a==b) {
   . . .
}
```

Code for comparing these 16-bit operands might be:

```
U-pipe:                 V-pipe:
xor  eax, eax           xor  ebx, ebx        ; 1
movw ax, [a]                                                ; 2
(prefix) + 1
movw bx, [b]                                                ; 4
(prefix) + 1
                                      cmp  eax, ebx         ; 6
```

Of course, this can only be done under certain circumstances, but the circumstances tend to be quite common. This would not work if the compare was for greater than, less than, greater than or equal, and so on, or if the values in `eax` or `ebx` were to be used in another operation where sign extension was required.

Pentium Pro and Pentium II processors provide special support to XOR a register with itself, recognizing that clearing a register does not depend on the old value of the register. Additionally, special support is provided for the above specific code sequence to avoid the partial stall. (See Section 3.9 for more information.)

The straightforward method may be slower on Pentium processors.

```
movsw     eax, a     ; 1  prefix + 3
movsw     ebx, b     ; 5
cmp       ebx, eax   ; 9
```

The performance of the `movzx` instructions has been improved in order to reduce the prevalence of partial stalls on Pentium Pro and Pentium II processors. When coding for Pentium Pro and Pentium II processors use the `movzx` instructions.

Compares

Use `test` when comparing a value in a register with zero. `Test` essentially ANDS the operands together without writing to a destination register. If a value is ANDed with itself and the result sets the zero condition flag, the value was zero. `Test` is preferred over `and` because the `and` writes the result register, which may subsequently cause an AGI or an

artificial output dependence on the Pentium Pro or Pentium II processor. `Test` is better than `cmp .., 0` because the instruction size is smaller.

Use `test` when comparing the result of a Boolean AND with an immediate constant for equality or inequality if the register is EAX `(if (avar & 8) { }).`

On the Pentium processor, `test` is a one-cycle pairable instruction when the form is `eax, imm` or `reg, reg`. Other forms of `test` take two cycles and do not pair.

Address Calculations

Pull address calculations into load and store instructions. Internally, memory reference instructions can have four operands: a relocatable load-time constant, an immediate constant, a base register, and a scaled index register. (In the segmented model, a segment register may constitute an additional operand in the linear address calculation.) In many cases, several integer instructions can be eliminated by fully using the operands of memory references.

Clearing a Register

The preferred sequence to move zero to a register is `xor reg, reg`. This saves code space but sets the condition codes. In contexts where the condition codes must be preserved, use `mov reg, 0`.

Integer Divide

Typically, an integer divide is preceded by a `cdq` instruction (divide instructions use EDX:EAX as the dividend and `cdq` sets up EDX). It is better to copy EAX into EDX, then right shift EDX 31 places to sign-extend. The copy/shift takes the same number of clocks as `cdq` on Pentium processors, but the copy/shift scheme allows two other instructions to execute at the same time on the Pentium processor. If you know that the value is positive, use `xor edx, edx`.

On Pentium Pro and Pentium II processors the `cdq` instruction is faster since `cdq` is a single µop instruction as opposed to two instructions for the copy/shift sequence.

Prolog Sequences

Be careful to avoid AGIs in the procedure and function prolog sequences due to register `esp`. Since `push` can pair with other `push` instructions, saving callee-saved registers on entry to functions should use these instructions. If possible, load parameters before decrementing ESP.

In routines that do not call other routines (leaf routines), use ESP as the base register to free up EBP. If you are not using the 32-bit flat model, remember that EBP cannot be used as a general purpose base register because it references the stack segment.

Avoid Compares with Immediate Zero

Often when a value is compared with zero, the operation producing the value sets condition codes which can be tested directly by a `jcc` instruction. The most notable exceptions are `mov` and `lea`. In these cases, use `test`.

Epilog Sequence

If only four bytes were allocated in the stack frame for the current function, instead of incrementing the stack pointer by four, use `pop` instructions. This avoids AGIs. For the Pentium processor use two `pops` for eight bytes.

4

Guidelines for Developing MMX™ Technology Code

CHAPTER 4
GUIDELINES FOR DEVELOPING MMX™ TECHNOLOGY CODE

The following guidelines should be observed in addition to the guidelines in Chapter 3. These and the previous guidelines will help you develop fast and efficient MMX technology code that scales well across all processors with MMX technology.

4.1 LIST OF RULES AND SUGGESTIONS

The following section provides a list of rules and suggestions.

4.1.1 Rules

- Do not intermix MMX instructions and floating-point instructions. See Section 4.2.3.
- Avoid small loads after large stores to the same area of memory. Avoid large loads after small stores to the same area of memory. Load and store data to the same area of memory using the same data sizes and address alignments. See Section 4.5.1.
- Use the `OP reg,mem` format to reduce instruction count or reduce register pressure, but be careful not to hurt performance by introducing excessive loads. See Section 4.4.1.
- Put an EMMS at the end of all sections of MMX instructions that you know will transition to floating-point code. See Section 4.2.4.
- Optimize cache data bandwidth to MMX technology registers. See Section 4.5.2.

4.2 PLANNING CONSIDERATIONS

Whether adapting an existing application or creating a new one, using MMX instructions to optimal advantage requires consideration of several issues. Generally, you should look for code segments that are computationally intensive, that are adaptable to integer implementations, and that support efficient use of the cache architecture. Several tools are provided in the Intel Performance Tool Set to aid in this evaluation and tuning.

Several questions should be answered before beginning your implementation:

- Which part of the code will benefit from MMX technology?
- Is the current algorithm the best for MMX technology?
- Is this code integer or floating-point?

- How should I arrange my data?
- Is my data 8-, 16- or 32-bit?
- Does the application need to run on processors both with and without MMX technology? Can I use `CPUID` to create a scaleable implementation?

4.2.1 Which Part of the Code will Benefit from MMX™ Technology?

Determine which code to convert.

Many applications have sections of code that are highly compute-intensive. Examples include speech compression algorithms and filters, video display routines and rendering routines. These routines are generally small, repetitive loops, operating on 8- or 16-bit integers and take a sizable portion of the application processing time. It is these routines that will yield the greatest performance increase when converted to MMX technology-optimized code. Encapsulating these loops into MMX technology-optimized libraries will allow greater flexibility in supporting platforms with and without MMX technology.

A performance optimization tool such as Intel's VTune visual tuning tool can be used to isolate the compute-intensive sections of code. Once these sections of code are identified, an evaluation should be done to determine whether the current algorithm or a modified one will give the best performance. In some cases, it is possible to improve performance by changing the types of operations in the algorithm. Matching the algorithms to MMX instruction capabilities is key to extracting the best performance.

4.2.2 Floating-Point or Integer?

Determine whether the algorithm contains floating-point or integer data.

If the current algorithm is implemented with integer data, then simply identify the portions of the algorithm that use the most microprocessor clock cycles. Once identified, reimplement these sections of code using MMX instructions.

If the algorithm contains floating-point data, then determine why floating-point was used. Several reasons exist for using floating-point operations: performance, range and precision. If performance was the reason for implementing the algorithm in floating-point, then the algorithm is a candidate for conversion to MMX integer code to increase performance.

If range or precision was an issue when implementing the algorithm in floating-point then further investigation needs to be made. Can the data values be converted to integer with the required range and precision? If not, this code is best left as floating-point code.

4.2.3 Applications with Both Floating-Point and MMX™ Technology Code

When generating MMX technology code, it is important to keep in mind that the eight MMX registers are aliased on the floating-point registers. Switching from MMX instructions to floating-point instructions can take up to fifty clock cycles, so it is best to minimize switching between these instruction types. Do not intermix MMX code and floating-point

code at the instruction level. If an application does perform frequent switches between floating-point and MMX instructions, then consider extending the period that the application stays in the MMX instruction stream or floating-point instruction stream to minimize the penalty of the switch.

When writing an application that uses both floating-point and MMX instructions, use the following guidelines for isolating instruction execution:

- Partition the MMX instruction stream and the floating-point instruction stream into separate instruction streams that contain instructions of one type.
- Do not rely on register contents across transitions.
- Leave an MMX code section with the floating-point tag word empty by using the EMMS instruction when you are sure the code transitions to floating-point code.
- Leave the floating-point code section with an empty stack.

For example:

```
FP_code:
     ...
     ...        /* leave the floating-point stack empty */

MMX_code:
     ...
     EMMS       /* empty the MMX registers */

FP_code1:
     ...
     ...        /* leave the floating-point stack empty */
```

Additional information on the floating-point programming model can be found in the *Pentium® Processor Family Developer's Manual*, Volume 3, Architecture and Programming (Order Number 241430).

4.2.4 EMMS Guidelines

Always call the EMMS instruction at the end of your MMX code when you are sure the code transitions to floating-point code.

Since the MMX registers are aliased on the floating-point registers, it is very important to clear the MMX registers before issuing a floating-point instruction. Use the EMMS instruction to clear the MMX registers and set the value of the floating-point tag word to empty (that is, all ones). This instruction should be inserted at the end of all MMX code segments to avoid an overflow exception in the floating-point stack when a floating-point instruction is executed.

4.2.5 CPUID Usage for Detection of MMX™ Technology

Determine if MMX technology is available.

MMX technology can be included in your application in two ways: Using the first method, have the application check for MMX technology during installation. If MMX technology is available, the appropriate DLLs can be installed. The second method is to check during program execution and install the proper DLLs at runtime. This is effective for programs that may be executed on different machines.

To determine whether you are executing on a processor with MMX technology, your application should check the Intel Architecture feature flags. The `CPUID` instruction returns the feature flags in the EDX register. Based on the results, the program can decide which version of code is appropriate for the system.

Existence of MMX technology support is denoted by bit 23 of the feature flags. When this bit is set to 1 the processor has MMX technology support. The following code segment loads the feature flags in EDX and tests the result for MMX technology. Additional information on `CPUID` usage can be found in application note AP-485, *Intel Processor Identification with CPUID Instruction* (Order Number 241618).

```
…                    identify existence of CPUID instruction
…                    ;
…                    ; identify Intel Processor
…                    ;
mov EAX, 1           ; request for feature flags
CPUID                ; 0Fh, 0A2h   CPUID Instruction
test EDX, 00800000h  ; is MMX technology bit(bit 23)in feature
                     ; flags equal to 1
jnz  Found
```

4.2.6 Alignment of Data

Make sure your data is aligned.

Many compilers allow you to specify the alignment of your variables using controls. In general this guarantees that your variables will be on the appropriate boundaries. However, if you discover that some of the variables are not appropriately aligned as specified, then align the variable using the following C algorithm. This aligns a 64-bit variable on a 64-bit boundary. Once aligned, every access to this variable will save three clock cycles on the Pentium Processor and six to nine cycles on Pentium Pro and Pentium II processors when the misaligned data crosses a cache line boundary.

```
double a[5];
double *p, *newp;

p = (double*)malloc ((sizeof(double)*5)+4)
newp = (p+4) & (-7)
```

Another way to improve data alignment is to copy the data into locations that are aligned on 64-bit boundaries. When the data is accessed frequently this can provide a significant performance improvement.

4.2.6.1 STACK ALIGNMENT

As a matter of convention, compilers allocate anything that is not static on the stack and it may be convenient to make use of the 64-bit data quantities that are stored on the stack. When this is necessary, it is important to make sure the stack is aligned. The following code in the function prologue and epilogue will make sure the stack is aligned.

```
Prologue:
     push      ebp                  ; save old frame ptr
     mov       ebp, esp             ; make new frame ptr
     sub       ebp, 4               ; make room of stack ptr
     and       ebp, 0FFFFFFF8       ; align to 64 bits
     mov       [ebp],esp            ; save old stack ptr
     mov       esp, ebp             ; copy aligned ptr
     sub       esp, FRAMESIZE       ; allocate space
     … callee saves, etc

epilogue:
     … callee restores, etc
     mov       esp, [ebp]
     pop       ebp
     ret
```

In cases where misalignment is unavoidable for some frequently accessed data, it may be useful to copy the data to an aligned temporary storage location.

4.2.7 Data Arrangement

MMX technology uses an SIMD technique to exploit the inherent parallelism of many multimedia algorithms. To get the most performance out of MMX code, data should be formatted in memory according to the guidelines below.

Consider a simple example of adding a 16-bit bias to all the 16-bit elements of a vector. In regular scalar code, you would load the bias into a register at the beginning of the loop, access the vector elements in another register, and do the addition one element at a time.

Converting this routine to MMX code, you would expect a four times speedup since MMX instructions can process four elements of the vector at a time using the `MOVQ` instruction, and can perform four additions at a time using the `PADDW` instruction. However, to achieve the expected speedup, you would need four contiguous copies of the bias in the MMX register when doing the addition.

In the original scalar code, only one copy of the bias is in memory. To use MMX instructions, you could use various manipulations to get four copies of the bias in an MMX register. Or you could format your memory in advance to hold four contiguous copies of the

bias. Then, you need only load these copies using one MOVQ instruction before the loop, and the four times speedup is achieved. For another interesting example of this type of data arrangement see Section 4.6.

Additionally, when accessing SIMD data with SIMD operations access to data can be improved simply by a change in the declaration. For example, consider a declaration of a structure which represents a point in space. The structure consists of three 16-bit values plus one 16-bit value for padding. The sample declaration follows:

```
typedef struct { short x,y,z; short junk; } Point;
Point pt[N];
```

In the following code the second dimension `y` needs to be multiplied by a scaling value. Here the `for` loop accesses each `y` dimension in the array `pt`:

```
for (i=0; i<N; i++) pt[i].y *= scale;
```

The access is not to contiguous data, which can cause a serious number of cache misses, degrading the performance of the application.

However, if the data is declared as follows, the scaling operation can be vectorized:

```
short ptx[N], pty[N], ptz[N];
for (i=0; i<N; i++) pty *= scale;
```

With the advent of MMX technology, choice of data organization becomes more important and should be made carefully based on the operations that will be performed on the data. In some applications, traditional data arrangements may not lead to the maximum performance.

The new 64-bit packed data types defined by MMX technology create more potential for misaligned data accesses. The data access patterns of many algorithms are inherently misaligned when using MMX instructions and other packed data types. A simple example of this is an FIR filter. An FIR filter is effectively a vector dot product in the length of the number of coefficient taps. If the filter operation of data element `i` is the vector dot product that begins at data element `j (data [ j ] *coeff [0] + data [j+1]*coeff [1]+...+data [j+num of taps-1]*coeff [num of taps-1])`, then the filter operation of data element `i+1` begins at data element `j+1`.

Section 4.2.6 covers aligning 64-bit data in memory. Assuming you have a 64-bit aligned data vector and a 64-bit aligned coefficients vector, the filter operation on the first data element will be fully aligned. For the filter operation on the second data element, however, each access to the data vector will be misaligned. Duplication and padding of data structures can be used to avoid the problem of data accesses in algorithms which are inherently misaligned. The application note AP-559, *MMX Instructions to Compute a 16-Bit Real FIR Filter* (Order Number 243044) shows an example of how to avoid the misalignment problem in the FIR filter.

NOTE

The duplication and padding technique overcomes the misalignment problem, thus avoiding the expensive penalty for misaligned data access, at the price of increasing the data size. When developing your code, you should consider this tradeoff and use the option which gives the best performance.

4.2.8 Tuning the Final Application

The best way to tune your application once it is functioning correctly is to use a profiler that measures the application while it is running on a system. Intel's VTune visual tuning tool is such a tool and can help you to determine where to make changes in your application to improve performance. Additionally, Intel's processors provide performance counters on-chip. Section 7.1 documents these counters and provides an explanation of how to use them.

4.3 SCHEDULING

The following section discusses instruction scheduling.

4.3.1 MMX™ Instruction Pairing Guidelines

This section specifies guidelines for pairing MMX instructions with each other and with integer instructions. Pairing of instructions improves Pentium processor performance significantly, it does not harm and sometimes help Pentium Pro and Pentium II processor performance.

4.3.1.1 PAIRING TWO MMX™ INSTRUCTIONS

Following are rules for pairing two MMX instructions:

- Two MMX instructions which 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 which 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 which 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.

4.3.1.2 PAIRING AN INTEGER INSTRUCTION IN THE U-PIPE WITH AN MMX™ INSTRUCTION IN THE V-PIPE

Following are rules for pairing an integer instruction in the U-pipe and an MMX instruction in the V-pipe:

- The MMX instruction is not 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 (see Table 3-1).

4.3.1.3 PAIRING AN MMX™ INSTRUCTION IN THE U-PIPE WITH AN INTEGER INSTRUCTION IN THE V-PIPE

The following rules apply to pairing an MMX instruction in the U-pipe and an integer instruction in the V-pipe:

- The V-pipe instruction is a pairable integer V-pipe instruction (see Table 3-1).
- The U-pipe MMX instruction does not access either memory or the integer register file.

4.3.1.4 SCHEDULING RULES

All MMX instructions can be pipelined, including the multiply instructions on Pentium, Pentium Pro and Pentium II processors. All instructions take a single clock to execute except MMX multiply instructions which take three clocks.

Since multiply instructions take three clocks to execute, the result of a multiply instruction can be used only by other instructions issued three clocks later. For this reason, avoid scheduling a dependent instruction in the two instruction pairs following the multiply.

As mentioned in Section 2.3.1, the store of a register after writing the register must wait for two clocks after the update of the register. Scheduling the store two clock cycles after the update avoids a pipeline stall.

4.4 INSTRUCTION SELECTION

The following section describes instruction selection optimizations.

4.4.1 Using Instructions That Access Memory

An MMX instruction may have two register operands (`OP reg, reg`) or one register and one memory operand (`OP reg, mem`), where `OP` represents the instruction operand, `reg` represents the register, and `mem` represents memory. `OP reg, mem` instructions are

useful in some cases to reduce register pressure, increase the number of operations per cycle, and reduce code size.

The following discussion assumes that the memory operand is present in the data cache. If it is not, then the resulting penalty is usually large enough to obviate the scheduling effects discussed in this section.

In Pentium processors, `OP reg, mem` MMX instructions do not have longer latency than `OP reg, reg` instructions (assuming a cache hit). They do have more limited pairing opportunities, however (see Section 4.3.1). In Pentium Pro and Pentium II processors, `OP reg, mem` MMX instructions translate into two micro-ops, as opposed to one µop for the `OP reg, reg` instructions. Thus, they tend to limit decoding bandwidth (see Section 2.2) and occupy more resources than `OP reg, reg` instructions.

Recommended usage of "`OP reg, mem`" instructions depends on whether the MMX code is memory-bound (that is, execution speed is limited by memory accesses). As a rule of thumb, an MMX code section is considered to be memory-bound if the following inequality holds:

$$\frac{\text{Instructions}}{2} < \text{Memory Accesses} + \frac{\text{Non-MMX Instructions}}{2}$$

For memory-bound MMX code, Intel recommends merging loads whenever the same memory address is used more than once. This reduces the number of memory accesses.

Example:

```
OP              MM0, [address A]
OP              MM1, [address A]
```

becomes:

```
MOVQ       MM2, [address A]
OP              MM0, MM2
OP              MM1, MM2
```

For MMX code that is not memory-bound, load merging is recommended only if the same memory address is used more than twice. Where load merging is not possible, usage of "`OP reg, mem`" instructions is recommended to minimize instruction count and code size.

Example:

```
MOVQ       MM0, [address A]
OP              MM1, MM0
```

becomes:

```
OP              MM1, [address A]
```

In many cases, a `MOVQ reg, reg` and `OP reg, mem` can be replaced by a `MOVQ reg, mem` and `OP reg, reg`. This should be done where possible, since it saves one µop on Pentium Pro and Pentium II processors.

Example (where OP is a symmetric operation):

```
MOVQ      MM1, MM0                (1 micro-op)
OP             MM1, [address A]   (2 micro-ops)
```

becomes:

```
MOVQ      MM1, [address A]    (1 micro-op)
OP             MM1, MM0                (1 micro-op)
```

4.5 MEMORY OPTIMIZATION

This section provides information on improving memory accesses.

4.5.1 Partial Memory Accesses

The MMX registers allow you to move large quantities of data without stalling the processor. Instead of loading single array values that are 8-, 16- or 32-bits long, consider loading the values in a single quadword, then incrementing the structure or array pointer accordingly.

Any data that will be manipulated by MMX instructions should be loaded using either:

- The MMX instruction that loads a 64-bit operand (for example, `MOVQ MM0, m64`), or
- The register-memory form of any MMX instruction that operates on a quadword memory operand (for example, `PMADDW MM0, m64`).

All SIMD data should be stored using the MMX instruction that stores a 64-bit operand (for example, `MOVQ m64, MM0`).

The goal of these recommendations is twofold. First, the loading and storing of SIMD data is more efficient using the larger quadword data block sizes. Second, this helps to avoid the mixing of 8-, 16- or 32-bit load and store operations with 64-bit MMX load and store operations to the same SIMD data. This, in turn, prevents situations in which small loads follow large stores to the same area of memory, or large loads follow small stores to the same area of memory. Pentium Pro and Pentium II processors will stall in these situations.

Consider the following examples. In the first case, there is a large load after a series of small stores to the same area of memory (beginning at memory address `mem`). The large load will stall in this case:

```
MOV       mem, eax        ; store dword to address “mem"
MOV       mem + 4, ebx    ; store dword to address “mem + 4"
      :
      :
MOVQ    mm0, mem          ; load qword at address “mem", stalls
```

The MOVQ must wait for the stores to write memory before it can access all the data it requires. This stall can also occur with other data types (for example, when bytes or words

are stored and then words or doublewords are read from the same area of memory). When you change the code sequence as follows, the processor can access the data without delay:

```
MOVD       mm1, ebx        ; build data into a qword first before
                           ; storing it to memory
MOVD       mm2, eax
PSLLQmm1, 32
POR        mm1, mm2
MOVQ       mem, mm1        ; store SIMD variable to “mem" as a qword
         :
         :
MOVQ       mm0, mem        ; load qword SIMD variable “mem", no stall
```

In the second case, there is a series of small loads after a large store to the same area of memory (beginning at memory address `mem`). The small loads will stall in this case:

```
MOVQ       mem, mm0        ; store qword to address “mem"
         :
         :
MOV        bx,  mem + 2    ; load word at address “mem + 2" stalls
MOV        cx,  mem + 4    ; load word at address “mem + 4" stalls
```

The word loads must wait for the quadword store to write to memory before they can access the data they require. This stall can also occur with other data types (for example, when doublewords or words are stored and then words or bytes are read from the same area of memory). When you change the code sequence as follows, the processor can access the data without delay:

```
MOVQ       mem, mm0        ; store qword to address “mem"
         :
         :
MOVQ       mm1, mem        ; load qword at address “mem"
MOVD       eax, mm1        ; transfer “mem + 2" to ax from
                           ; MMX register not memory
PSRLQmm1, 32
SHR        eax, 16
MOVD       ebx, mm1        ; transfer “mem + 4" to bx from
                           ; MMX register, not memory
AND  ebx, 0ffffh
```

These transformations, in general, increase the number the instructions required to perform the desired operation. For Pentium Pro and Pentium II processors, the performance penalty due to the increased number of instructions is more than offset by the benefit. For Pentium processors, however, the increased number of instructions can negatively impact performance, since these processors do not benefit from the code transformations above. For this reason, careful and efficient coding of these transformations is necessary to minimize any potential negative impact to Pentium processor performance.

4.5.2 Increasing Bandwidth of Memory Fills and Video Fills

It is beneficial to understand how memory is accessed and filled. A memory-to-memory fill (for example a memory-to-video fill) is defined as a 32-byte (cache line) load from memory which is immediately stored back to memory (such as a video frame buffer). The following are guidelines for obtaining higher bandwidth and shorter latencies for sequential memory fills (video fills). These recommendations are relevant for all Intel Architecture processors with MMX technology and refer to cases in which the loads and stores do not hit in the second level cache.

4.5.2.1 INCREASING MEMORY BANDWIDTH USING THE MOVQ INSTRUCTION

Loading any value will cause an entire cache line to be loaded into the on-chip cache. But using `MOVQ` to store the data back to memory instead of using 32-bit stores (for example, `MOVD`) will reduce by half the number of stores per memory fill cycle. As a result, the bandwidth of the memory fill cycle increases significantly. On some Pentium processor-based systems, 30% higher bandwidth was measured when 64-bit stores were used instead of 32-bit stores. Additionally, on Pentium Pro and Pentium II processors, this avoids a partial memory access when both the loads and stores are done with the `MOVQ` instruction.

Also, intermixing reads and writes is slower than doing a series of reads then writing out the data. For example if moving memory, it is faster to read several lines into the cache from memory then write them out again to the new memory location, instead of issuing one read and one write.

4.5.2.2 INCREASING MEMORY BANDWIDTH BY LOADING AND STORING TO AND FROM THE SAME DRAM PAGE

DRAM is divided into pages, which are not the same as operating system (OS) pages. The size of a DRAM page is a function of the total size of the DRAM and the organization of the DRAM. Page sizes of several Kbytes are common. Like OS pages, DRAM pages are constructed of sequential addresses. Sequential memory accesses to the same DRAM page have shorter latencies than sequential accesses to different DRAM pages. In many systems the latency for a page miss (that is, an access to a different page instead of the page previously accessed) can be twice as large as the latency of a memory page hit (access to the same page as the previous access). Therefore, if the loads and stores of the memory fill cycle are to the same DRAM page, a significant increase in the bandwidth of the memory fill cycles can be achieved.

4.5.2.3 INCREASING THE MEMORY FILL BANDWIDTH BY USING ALIGNED STORES

Unaligned stores will double the number of stores to memory. Intel strongly recommends that quadword stores be 8-byte aligned. Four aligned quadword stores are required to write a cache line to memory. If the quadword store is not 8-byte aligned, then two 32-bit writes result from each `MOVQ` store instruction. On some systems, a 20% lower bandwidth was measured when 64-bit misaligned stores were used instead of aligned stores.

4.5.2.4 USE 64-BIT STORES TO INCREASE THE BANDWIDTH TO VIDEO

Although the PCI bus between the processor and the frame buffer is 32 bits wide, using MOVQ to store to video is faster on most Pentium processor-based systems than using twice as many 32-bit stores to video. This occurs because the bandwidth to PCI write buffers (which are located between the processor and the PCI bus) is higher when quadword stores are used.

4.5.2.5 INCREASE THE BANDWIDTH TO VIDEO USING ALIGNED STORES

When a nonaligned store is encountered, there is a dramatic decrease in the bandwidth to video. Misalignment causes twice as many stores and the latency of stores on the PCI bus (to the frame buffer) is much longer. On the PCI bus, it is not possible to burst sequential misaligned stores. On Pentium processor-based systems, a decrease of 80% in the video fill bandwidth is typical when misaligned stores are used instead of aligned stores.

4.6 Coding Techniques

This section contains several simple examples that will help you to get started in coding your application. The goal is to provide simple, low-level operations that are frequently used. Each example uses the minimum number of instructions necessary to achieve best performance on Pentium, Pentium Pro and Pentium II processors.

Each example includes:

- A short description.
- Sample code.
- Any necessary notes.

These examples do not address scheduling as it is assumed the examples will be incorporated in longer code sequences.

4.6.1 Unsigned Unpack

The MMX technology provides several instructions that are used to pack and unpack data in the MMX registers. The unpack instructions can be used to zero-extend an unsigned number. The following example assumes the source is a packed-word (16-bit) data type.

Input: MM0: Source value
MM7: 0 (A local variable can be used instead of the register MM7, if desired.)

Output: MM0: two zero-extended 32-bit doublewords from 2 LOW end words
MM1: two zero-extended 32-bit doublewords from 2 HIGH end words

```
MOVQ       MM1, MM0   ; copy source
PUNPCKLWD  MM0, MM7   ; unpack the 2 low end words
                      ; into two 32-bit double word
PUNPCKHWD  MM1, MM7   ; unpack the 2 high end words into two
                      ; 32-bit double word
```

4.6.2 Signed Unpack

Signed numbers should be sign-extended when unpacking the values. This is done differently than the zero-extend shown above. The following example assumes the source is a packed-word (16-bit) data type.

Input: MM0: source value

Output: MM0: two sign-extended 32-bit doublewords from the two LOW end words
MM1: two sign-extended 32-bit doublewords from the two HIGH end words

```
PUNPCKHWD      MM1, MM0       ; unpack the 2 high end words of the
                    ; source into the second and fourth
                    ; words of the destination
PUNPCKLWD      MM0, MM0       ; unpack the 2 low end words of the
                    ; source into the second and fourth
                    ; words of the destination
PSRAD     MM0, 16   ; Sign-extend the 2 low end words of
                    ; the source into two 32-bit signed
                    ; doublewords
PSRAD     MM1, 16   ; Sign-extend the 2 high end words of
                    ; the source into two 32-bit signed
                    ; doublewords
```

4.6.3 Interleaved Pack with Saturation

The pack instructions pack two values into the destination register in a predetermined order. Specifically, the PACKSSDW instruction packs two signed doublewords from the source operand and two signed doublewords from the destination operand into four signed words in the destination register as shown in the figure below.

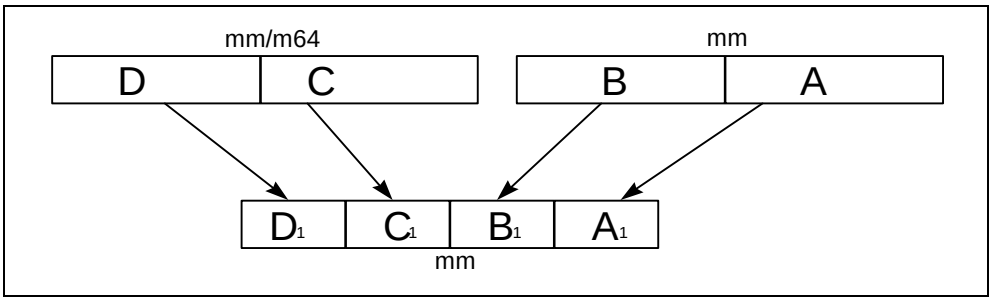


Figure 4-1. PACKSSDW mm, mm/mm64 Instruction Example

The following example interleaves the two values in the destination register, as shown in the figure below.

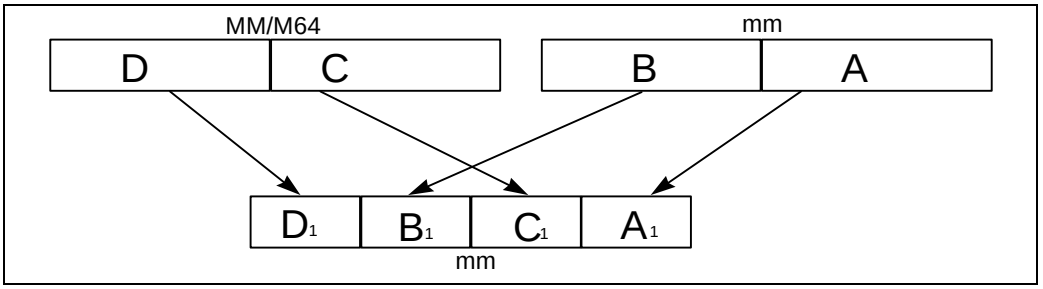


Figure 4-2. Interleaved Pack with Saturation Example

This example uses signed doublewords as source operands and the result is interleaved signed words. The pack instructions can be performed with or without saturation as needed.

Input: MM0: Signed source1 value
MM1: Signed source2 value

Output: MM0: The first and third words contain the signed-saturated doublewords from MM0
MM0: The second and fourth words contain signed-saturated doublewords from MM1

```
PACKSSDW   MM0, MM0   ; pack and sign saturate
PACKSSDW   MM1, MM1   ; pack and sign saturate
PUNPKLWD   MM0, MM1   ; interleave the low end 16-bit values of the
                      ; operands
```

The pack instructions always assume the source operands are signed numbers. The result in the destination register is always defined by the pack instruction that performs the operation. For example, the PACKSSDW instruction, packs each of the two signed 32-bit values of the two sources into four saturated 16-bit signed values in the destination register. The PACKUSWB instruction, on the other hand, packs each of the four signed 16-bit values of the two sources into four saturated 8-bit unsigned values in the destination. A complete specification of the MMX instruction set can be found in the *Intel Architecture MMX™ Technology Programmer's Reference Manual* (Order Number 243007).

4.6.4 Interleaved Pack without Saturation

This example is similar to the last except that the resulting words are not saturated. In addition, in order to protect against overflow, only the low order 16-bits of each doubleword are used in this operation.

Input: MM0: signed source value
MM1: signed source value

Output: MM0: The first and third words contain the low 16-bits of the doublewords in MM0
MM0: The second and fourth words contain the low 16-bits of the doublewords in MM1

```
PSLLD     MM1, 16    ; shift the 16 LSB from each of the doubleword
                     ; values to the 16 MSB position
PAND      MM0, {0,ffff,0,ffff}
                     ; mask to zero the 16 MSB of each
                     ; doubleword value
POR       MM0, MM1   ; merge the two operands
```

4.6.5 Non-Interleaved Unpack

The unpack instructions perform an interleave merge of the data elements of the destination and source operands into the destination register. The following example merges the two operands into the destination registers without interleaving. For example, take two adjacent elements of a packed-word data type in source1 and place this value in the low 32-bits of the results. Then take two adjacent elements of a packed-word data type in source2 and place this value in the high 32-bits of the results. One of the destination registers will have the combination shown in Figure 4-3.

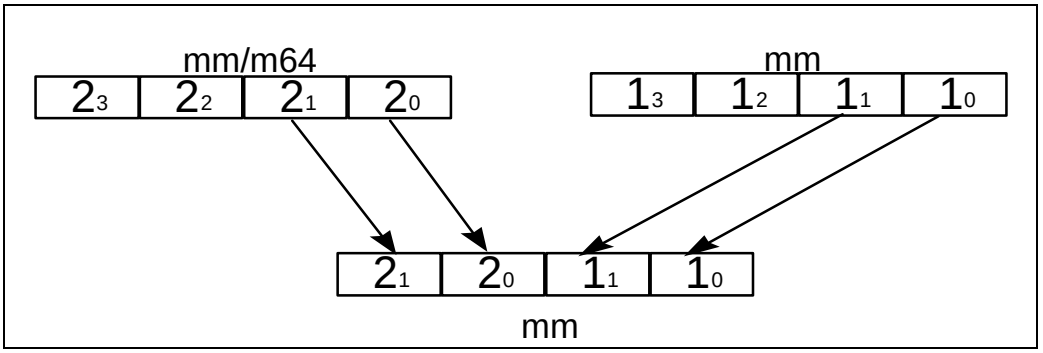


Figure 4-3. Result of Non-Interleaved Unpack in MM0

The other destination register will contain the opposite combination as in Figure 4-4.

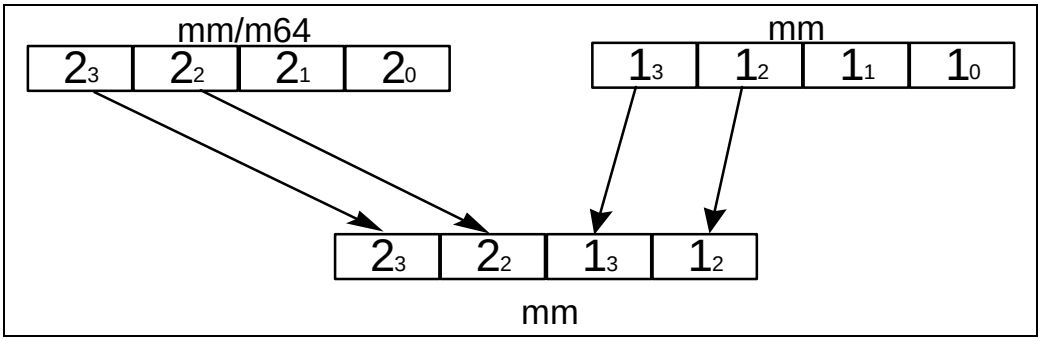


Figure 4-4. Result of Non-Interleaved Unpack in MM1

The following example unpacks two packed-word sources in a non-interleaved way. The trick is to use the instruction which unpacks doublewords to a quadword, instead of using the instruction which unpacks words to doublewords.

Input: MM0: packed-word source value
MM1: packed-word source value

Output: MM0: contains the two low end words of the original sources, non-interleaved
MM2: contains the two high end words of the original sources, non-interleaved.

```
MOVQ            MM2, MM0   ; copy source1
PUNPCKLDQ  MM0, MM1        ; replace the two high end words of MM0
                           ; with the two low end words of MM1; leave
                           ; the two low end words of MM0 in place
PUNPCKHDQ  MM2, MM1        ; move the two high end words of MM2 to the
                           ; two low end words of MM2; place the two
                           ; high end words of MM1 in the two high end
                           ; words of MM2
```

4.6.6 Complex Multiply by a Constant

Complex multiplication is an operation which requires four multiplications and two additions. This is exactly how the PMADDWD instruction operates. In order to use this instruction you need only to format the data into four 16-bit values. The real and imaginary components should be 16-bits each.

Let the input data be Dr and Di

Where:

Dr = real component of the data

Di = imaginary component of the data

Format the constant complex coefficients in memory as four 16-bit values [Cr -Ci Ci Cr]. Remember to load the values into the MMX register using a MOVQ instruction.

Input: MM0: a complex number Dr, Di
MM1: constant complex coefficient in the form[Cr -Ci Ci Cr]

Output: MM0: two 32-bit dwords containing [Pr Pi]

The real component of the complex product is Pr = Dr*Cr - Di*Ci, and the imaginary component of the complex product is Pi = Dr*Ci + Di*Cr

```
PUNPCKLDQ       MM0, MM0        ; This makes [Dr Di Dr Di]
PMADDWD         MM0, MM1        ; and you're done, the result is
                                ; [(Dr*Cr-Di*Ci)(Dr*Ci+Di*Cr)]
```

Note that the output is a packed word. If needed, a pack instruction can be used to convert the result to 16-bit (thereby matching the format of the input).

4.6.7 Absolute Difference of Unsigned Numbers

This example computes the absolute difference of two unsigned numbers. It assumes an unsigned packed-byte data type. Here, we make use of the subtract instruction with unsigned saturation. This instruction receives UNSIGNED operands and subtracts them with UNSIGNED saturation. This support exists only for packed bytes and packed words, NOT for packed dwords.

Input: MM0: source operand
MM1: source operand

Output: MM0: The absolute difference of the unsigned operands

```
MOVQ            MM2, MM0        ; make a copy of MM0
PSUBUSB         MM0, MM1        ; compute difference one way
PSUBUSB         MM1, MM2        ; compute difference the other way
POR             MM0, MM1        ; OR them together
```

This example will not work if the operands are signed. See the next example for signed absolute differences.

4.6.8 Absolute Difference of Signed Numbers

This example computes the absolute difference of two signed numbers. There is no MMX subtract instruction which receives SIGNED operands and subtracts them with UNSIGNED saturation. The technique used here is to first sort the corresponding elements of the input operands into packed-words of the maxima values, and packed-words of the minima values. Then the minima values are subtracted from the maxima values to generate the required absolute difference. The key is a fast sorting technique which uses the fact that B = XOR(A, XOR(A,B)) and A = XOR(A,0). Thus in a packed data type, having some elements being XOR(A,B) and some being 0, you could XOR such an operand with A and receive in some places values of A and in some values of B. The following examples assume a packed-word data type, each element being a signed value.

Input: MM0: signed source operand
MM1: signed source operand

Output: MM0: The absolute difference of the signed operands

```
MOVQ            MM2, MM0   ; make a copy of source1 (A)
PCMPGTW         MM0, MM1   ; create mask of source1>source2 (A>B)
MOVQ            MM4, MM2   ; make another copy of A
PXOR            MM2, MM1   ; Create the intermediate value of the swap
                                        ; operation - XOR(A,B)
PAND            MM2, MM0   ; create a mask of  0s and XOR(A,B)
                                        ; elements. Where A>B there
will be a value
                                        ; XOR(A,B) and where A<=B
there will be 0.
MOVQ            MM3, MM2   ; make a copy of the swap mask
PXOR            MM4, MM2   ; This is the minima - XOR(A, swap mask)
PXOR            MM1, MM3   ; This is the maxima - XOR(B, swap mask)
PSUBW           MM1, MM4   ; absolute difference = maxima-minima
```

4.6.9 Absolute Value

Use the following example to compute $|x|$, where x is signed. This example assumes signed words to be the operands.

Input: MM0: signed source operand

Output: MM1: ABS(MM0)

```
MOVQ      MM1, MM0   ; make a copy of x
PSRAW     MM0,15     ; replicate sign bit (use 31 if doing DWORDS)
PXOR      MM0, MM1   ; take 1's complement of just the
                     ; negative fields
PSUBS     MM1, MM0   ; add 1 to just the negative fields
```

Note that the absolute value of the most negative number (that is, 8000 hex for 16-bit) does not fit, but this code does something reasonable for this case; it gives 7fff which is off by one.

4.6.10 Clipping Signed Numbers to an Arbitrary Signed Range [HIGH, LOW]

This example shows how to clip a signed value to the signed range [HIGH, LOW]. Specifically, if the value is less than LOW or greater than HIGH then clip to LOW or HIGH, respectively. This technique uses the packed-add and packed-subtract instructions with unsigned saturation, which means that this technique can only be used on packed-bytes and packed-words data types.

The following examples use the constants `packed_max` and `packed_min`. The examples show operations on word values. For simplicity we use the following constants (corresponding constants are used in case the operation is done on byte values):

- PACKED_MAX equals 0x7FFF7FFF7FFF7FFF
- PACKED_MIN equals 0x8000800080008000
- PACKED_LOW contains the value LOW in all 4 words of the packed-words data type
- PACKED_HIGH contains the value HIGH in all 4 words of the packed-words data type
- PACKED_USMAX is all 1's
- HIGH_US adds the HIGH value to all data elements (4 words) of PACKED_MIN
- LOW_US adds the LOW value to all data elements (4 words) of PACKED_MIN

Input: MM0: Signed source operands

Output: MM0: Signed operands clipped to the unsigned range [HIGH, LOW]

```
PADD      MM0, PACKED_MIN                ; add with no saturation
                                         ; 0x8000 to convert to
                                         ; unsigned
PADDUSW   MM0, (PACKED_USMAX - HIGH_US)  ; in effect this clips
                                         ; to HIGH
PSUBUSW       MM0, (PACKED_USMAX - HIGH_US + LOW_US) ;
                                         ; in effect this clips
                                         ; to LOW
PADDW     MM0, PACKED_LOW                ; undo the previous
                                         ; two offsets
```

The code above converts values to unsigned numbers first and then clips them to an unsigned range. The last instruction converts the data back to signed data and places the data within the signed range. Conversion to unsigned data is required for correct results when the quantity (HIGH - LOW) < 0x8000.

IF (HIGH - LOW) >= 0x8000, the algorithm can be simplified to the following:

Input: MM0: Signed source operands

Output: MM0: Signed operands clipped to the unsigned range [HIGH, LOW]

```
PADDSSW         MM0, (PACKED_MAX - PACKED_HIGH)     ; in effect this
clips
                                                    ; to HIGH
PSUBSSW         MM0, (PACKED_USMAX - PACKED_HIGH + PACKED_LOW)
                                                    ;clips to LOW
PADDW      MM0, LOW                                 ; undo the previous
                                                    ; two offsets
```

This algorithm saves a cycle when it is known that (HIGH - LOW) >= 0x8000. To see why the three-instruction algorithm does not work when (HIGH - LOW) < 0x8000, realize that 0xffff minus any number less than 0x8000 will yield a number greater in magnitude than 0x8000, which is a negative number. When:

```
PSUBSSW         MM0, (0xFFFF - HIGH + LOW)
```

(the second instruction in the three-step algorithm) is executed, a negative number is subtracted causing the values in MM0 to be increased instead of decreased**,** as should be the case, and causing an incorrect answer to be generated.

4.6.11 Clipping Unsigned Numbers to an Arbitrary Unsigned Range [HIGH, LOW]

This example clips an unsigned value to the unsigned range [HIGH, LOW]. If the value is less than LOW or greater than HIGH, then clip to LOW or HIGH, respectively. This technique uses the packed-add and packed-subtract instructions with unsigned saturation, thus this technique can only be used on packed-bytes and packed-words data types.

The example illustrates the operation on word values.

Input: MM0: Unsigned source operands

Output: MM0: Unsigned operands clipped to the unsigned range [HIGH, LOW]

```
PADDUSW         MM0, 0xFFFF - HIGH        ; in effect this clips to
HIGH
PSUBUSW         MM0, (0xFFFF - HIGH + LOW)     ; in effect this clips
to LOW
PADDW      MM0, LOW                       ; undo the previous two
offsets
```

4.6.12 Generating Constants

The MMX instruction set does not have an instruction that will load immediate constants to MMX registers. The following code segments will generate frequently used constants in an MMX register. Of course, you can also put constants as local variables in memory, but when

doing so be sure to duplicate the values in memory and load the values with a MOVQ instruction.

Generate a zero register in MM0:

```
PXOR       MM0, MM0
```

Generate all 1's in register MM1, which is -1 in each of the packed data type fields:

```
PCMPEQ     MM1, MM1
```

Generate the constant 1 in every packed-byte [or packed-word] (or packed-dword) field:

```
PXOR       MM0, MM0
PCMPEQ     MM1, MM1
PSUBBMM0, MM1        [PSUBW     MM0, MM1] (PSUBD     MM0, MM1)
```

Generate the signed constant 2^{n-1} in every packed-word (or packed-dword) field:

```
PCMPEQ     MM1, MM1
PSRLWMM1, 16-n                            (PSRLD     MM1, 32-n)
```

Generate the signed constant -2^n in every packed-word (or packed-dword) field:

```
PCMPEQ     MM1, MM1
PSLLWMM1, n                               (PSLLD     MM1, n)
```

Because the MMX instruction set does not support shift instructions for bytes, 2^{n-1} and -2^n are relevant only for packed-words and packed-dwords.

5

Optimization Techniques for Floating-Point Applications

CHAPTER 5
OPTIMIZATION TECHNIQUES FOR FLOATING-POINT APPLICATIONS

This chapter details the optimizations for floating-point applications. This chapter contains:

- General rules for optimizing floating-point code.
- Examples that illustrate the optimization techniques.

5.1 IMPROVING THE PERFORMANCE OF FLOATING-POINT APPLICATIONS

When programming floating-point applications it is best to start at the C or FORTRAN language level. Many compilers perform floating-point scheduling and optimization when it is possible. However in order to produce optimal code the compiler may need some assistance.

5.1.1 Guidelines for Optimizing Floating-Point Code

Follow these rules to improve the speed of your floating-point applications:

- Understand how the compiler handles floating-point code. Look at the assembly dump and see what transforms are already performed on the program. Study the loop nests in the application that dominate the execution time. Determine why the compiler is not creating the fastest code.
- Is there a dependence that can be resolved?
  - — large memory bandwidth requirements.
  - — poor cache locality.
  - — long-latency floating-point arithmetic operations.
- Do not use too much precision when it is not necessary. Single precision (32-bits) is faster on some operations and consumes only half the memory space as double precision (64-bits) or double extended (80-bits).
- Make sure you have fast floating-point to integer routines. Many libraries do more work than is necessary; make sure your float-to-int is a fast routine. See Section 5.4.
- Make sure your application stays in range. Out of range numbers cause very high overhead.

- Schedule your code in assembly language using FXCH. Unroll loops and pipeline your code. See Section 5.1.2.
- Perform transformations to improve memory access patterns. Use loop fusion or compression to keep as much of the computation in the cache as possible. See Section 5.5
- Break dependency chains.

5.1.2 Improving Parallelism

Pentium, Pentium Pro and Pentium II processors have a pipelined floating-point unit. By scheduling the floating-point instructions maximum throughput from the Pentium processor floating-point unit can be achieved. Additionally, these optimizations can also help Pentium Pro and Pentium II processors when it improves the pipelining of the floating-point unit. Consider the example in Figure 5-1 below:

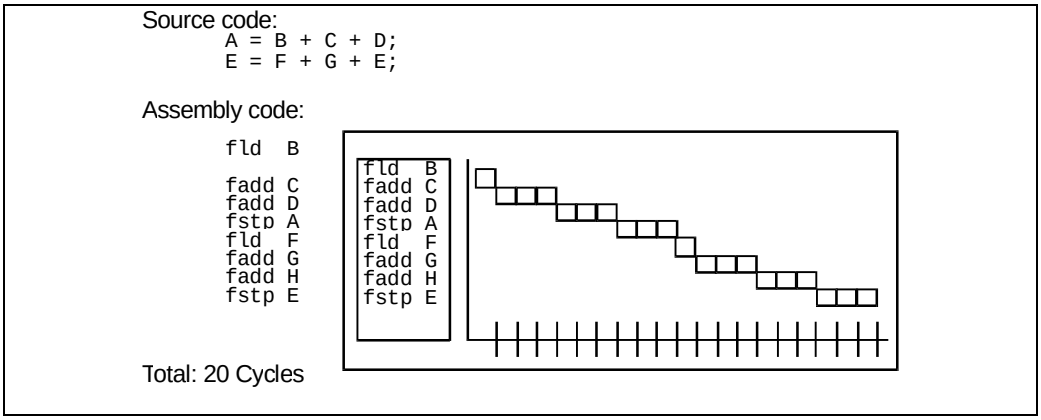


Figure 5-1. Floating-Point Example

To exploit the parallel capability of the Pentium, Pentium Pro and Pentium II processors, determine which instructions can be executed in parallel. The two high level code statements in the example are independent, therefore their assembly instructions can be scheduled to execute in parallel, thereby improving the execution speed.

Source code:

```
A = B + C + D;
E = F + G + E;
```

```
fld   B          fld   F
fadd  C          fadd  G
fadd  D          fadd  H
fstp  A          fstp  E
```

Most floating-point operations require that one operand and the result use the top of stack. This makes each instruction dependent on the previous instruction and inhibits overlapping the instructions.

One obvious way to get around this is to imagine that we have a flat floating-point register file available, rather than a stack. The code would look like this:

```
fld   B         →F1
fadd  F1, C     →F1
fld   F         →F2
fadd  F2,G      →F2
fadd  F1,D      →F1
fadd  F2,H      →F2
fstp  F1        →A
fstp  F2        →E
```

In order to implement these imaginary registers we need to use the `fxch` instruction to change the value on the top of stack. This provides a way to avoid the top of stack dependency. The `fxch` instructions can be paired with the common floating-point operations, so there is no penalty on the Pentium processor. Additionally, the `fxch` uses no extra execution cycles on Pentium Pro and Pentium II processors.

```
                                        STO       ST1
fld   B         →F1  fld B              B
fadd  F1, C     →F1  fadd C             B+C
fld   F         →F2  fld F              F         B+C
fadd  F2,G      →F2  fadd G             F+G       B+C
                     fxch ST(1)         B+C       F+G
fadd  F1,D      →F1  fadd D             B+C+D     F+G
                     fxch ST(1)         F+G       B+C+D
fadd  F2,H      →F2  fadd H             F+G+H     B+C+D
                     fxch ST(1)         B+C+D     F+G+H
fstp  F1        →A   fstp A             F+G+H
fstp  F2        →E   fstp E
```

On the Pentium processor, the `fxch` instructions pair with preceding `fadd` instructions and execute in parallel with them. The `fxch` instructions move an operand into position for the next floating-point instruction. The result is an improvement in execution speed on the Pentium processor as shown in Figure 5-2.

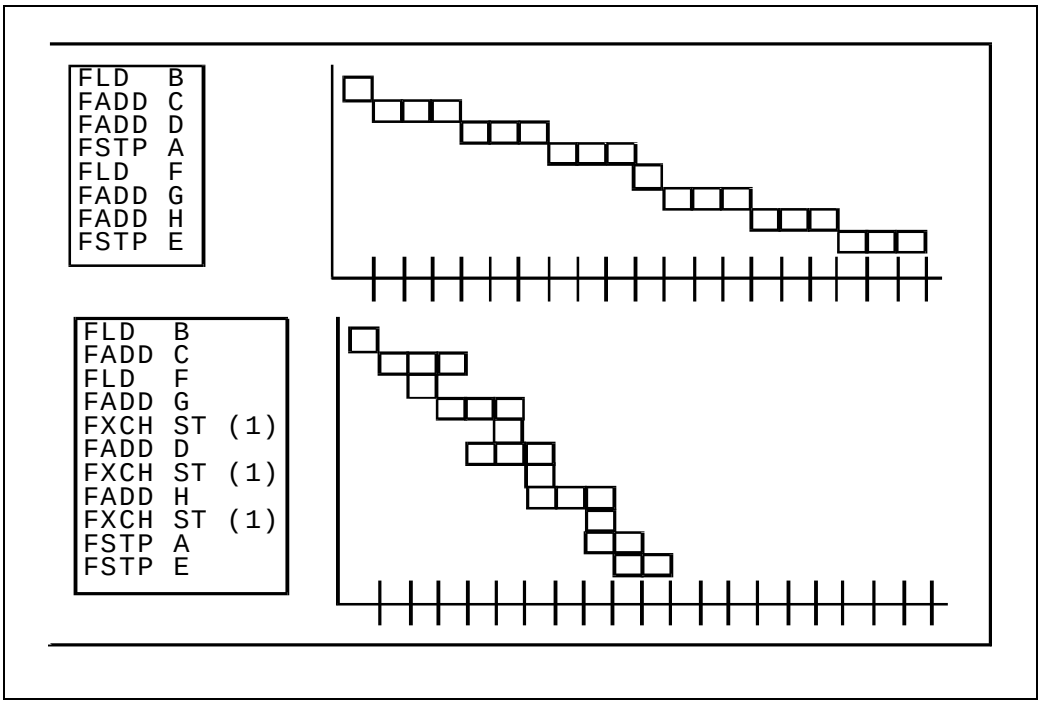


Figure 5-2. Floating-Point Example Before and After Optimization

5.1.2.1 FXCH RULES AND REGULATIONS

The `fxch` instruction costs no extra cycles on the Pentium processor, since it executes in the V-pipe along with other floating-point instructions when all of the following conditions occur:

- An FP instruction follows the `fxch` instruction.
- An FP instruction from the following list immediately precedes the `fxch` instruction: `fadd`, `fsub`, `fmul`, `fld`, `fcom`, `fucom`, `fchs`, `ftst`, `fabs`, `fdiv`.
- The `fxch` instruction has already been executed. This is because the instruction boundaries in the cache are marked the first time the instruction is executed, so pairing only happens the second time this instruction is executed from the cache.

When the above conditions are true, the instruction is almost “free” and can be used to access elements in the deeper levels of the FP stack instead of storing them and then loading them again.

5.2 MEMORY OPERANDS

Performing a floating-point operation on a memory operand instead of on a stack register costs no cycles on the Pentium processor when the memory operand is in the cache. On Pentium Pro and Pentium II processors, instructions with memory operands produce two micro-ops, which can limit decoding. Additionally, memory operands may cause a data cache miss, causing a penalty. Floating-point operands that are 64-bit operands need to be 8-byte aligned. For more information on decoding see Section 3.6.4.

5.3 MEMORY ACCESS STALL INFORMATION

Floating-point registers allow loading of 64-bit values as doubles. Instead of loading single array values that are 8-, 16- or 32-bits long, consider loading the values in a single quadword, then incrementing the structure or array pointer accordingly.

First, the loading and storing of quadword data is more efficient using the larger quadword data block sizes. Second, this helps to avoid the mixing of 8-, 16- or 32-bit load and store operations with a 64-bit load and store operation to the memory address. This avoids the possibility of a memory access stall on Pentium Pro or Pentium II processors. Memory access stalls occur when:

- Small loads follow large stores to the same area of memory.
- Large loads follow small stores to the same area of memory. Pentium Pro and Pentium II processors will stall in these situations.

Consider the following examples. In the first case, there is a large load after a series of small stores to the same area of memory (beginning at memory address `mem`). The large load will stall in this case:

```
mov       mem, eax        ; store dword to address “mem"
mov       mem + 4, ebx    ; store dword to address “mem + 4"
          :
          :
fld       mem             ; load qword at address “mem", stalls
```

The `fld` must wait for the stores to write memory before it can access all the data it requires. This stall can also occur with other data types (for example, when bytes or words are stored and then words or doublewords are read from the same area of memory).

In the second case, there is a series of small loads after a large store to the same area of memory (beginning at memory address `mem`). The small loads will stall in this case:

```
fstp      mem             ; store qword to address “mem"
          :
          :
mov       bx,  mem + 2    ; load word at address “mem + 2", stalls
mov       cx,  mem + 4    ; load word at address “mem + 4", stalls
```

The word loads must wait for the quadword store to write to memory before they can access the data they require. This stall can also occur with other data types (for example, when

doublewords or words are stored and then words or bytes are read from the same area of memory). This can be avoided by moving the store as far from the loads as possible. In general, the loads and stores should be separated by at least 10 instructions to avoid the stall condition.

5.4 FLOATING-POINT TO INTEGER CONVERSION

Many libraries provide the float to integer library routines that convert floating-point values to integer. Many of these libraries conform to ANSI C coding standards which state that the rounding mode should be truncation. The default of the `FIST` instruction is round to nearest, therefore many compiler writers implement a change in the rounding mode in the processor in order to conform to the C and FORTRAN standards. This implementation requires changing the control word on the processor using the `fldcw` instruction. This instruction is a synchronizing instruction and will cause a significant slowdown in the performance of your application on Pentium, Pentium Pro and Pentium II processors.

When implementing an application, consider if the rounding mode is important to the results. If not, use the following function to avoid the synchronization and overhead of the `fldcw` instruction.

To avoid changing the rounding mode use the following algorithm:

```
_ftol32proc
     lea      ecx,[esp-8]
     sub      esp,16       ; allocate frame
     and      ecx,-8       ; align pointer on boundary of 8
     fld      st(0)        ; duplicate FPU stack top
     fistp    qword ptr[ecx]
     fild     qword ptr[ecx]
     mov      edx,[ecx+4]  ; high dword of integer
     mov      eax,[ecx]    ; low dword of integer
     test     eax,eax
     je       integer_QNaN_or_zero

arg_is_not_integer_QNaN:
     fsubp    st(1),st     ; TOS=d-round(d),
                           { st(1)=st(1)-st & pop ST }
     test     edx,edx      ; what's sign of integer
     jns      positive
; number is negative
                           ; dead cycle
                           ; dead cycle
     fstp     dword ptr[ecx] result of subtraction
     mov      ecx,[ecx]    ; dword of difference(single precision)
     add      esp,16
     xor      ecx,80000000h
     add      ecx,7fffffffh; if difference>0 then increment integer

     adc      eax,0        ; inc eax (add CARRY flag)
     ret
positive:
     fstp     dword ptr[ecx]17-18  ; result of subtraction
     mov      ecx,[ecx]    ; dword of difference (single precision)

     add      esp,16
     add      ecx,7fffffffh; if difference<0 then decrement integer
     sbb      eax,0        ; dec eax (subtract CARRY flag)
     ret
integer_QNaN_or_zero:
     test     edx,7fffffffh
     jnz      arg_is_not_integer_QNaN
     add      esp,16
     ret
```

5.5 LOOP UNROLLING

There are many benefits to unrolling loops; however, these benefits need to be balanced with I-Cache constraints and other machine resources. The benefits are:

- Unrolling amortizes the branch overhead. The BTB is good at predicting loops on Pentium, Pentium Pro and Pentium II processors and the instructions to increment the loop index and jump are inexpensive.
- Unrolling allows you to aggressively schedule (or pipeline) the loop to hide latencies. This is useful if you have enough free registers to keep variables live as you stretch out the dependency chain to expose the critical path
- You can aggressively schedule the loop to better set up I-fetch and decode constraints.
- The backwards branch (predicted taken) has only a 1 clock penalty on Pentium Pro and Pentium II processors, so you can unroll very tiny loop bodies for free
- Unrolling can expose other optimizations, as shown in the examples below.

This loop executes 100 times assigning `x` to every even-numbered element and `y` to every odd-numbered element.

```
do i=1,100
  if (i mod 2 == 0) then a(i) = x
  else a(i) = y
enddo
```

By unrolling the loop you can make both assignments each iteration, removing one branch in the loop body.

```
do i=1,100,2
  a(i) = y
  a(i+1) = x
enddo
```

5.6 FLOATING-POINT STALLS

Many of the floating-point instructions have a latency greater than one cycle, therefore on the Pentium processor family the next floating-point instruction cannot access the result until the first operation has finished execution. To hide this latency, instructions should be inserted between the pair that cause the pipe stall. These instructions can be integer instructions or floating-point instructions that will not cause a new stall themselves. The number of instructions that should be inserted depends on the length of the latency. Because of the out-of-order nature of Pentium Pro and Pentium II processors, stalls will not necessarily occur on an instruction or µop basis. However, if an instruction has a very long latency such as an `FDIV`, then scheduling can improve the throughput of the overall application. The following sections list considerations for floating-point pipelining on the Pentium processor family.

5.6.1 Using Integer Instructions to Hide Latencies of Floating-Point Instructions

When a floating-point instruction depends on the result of the immediately preceding instruction, and it is also a floating-point instruction, it is advantageous to move integer instructions between the two FP instructions, even if the integer instructions perform loop control. The following example restructures a loop in this manner:

```
for (i=0; i<Size; i++)
     array1 [i] += array2 [i];
; assume eax=Size-1, esi=array1, edi=array2
                                Pentium Processor
                                     CLOCKS
     LoopEntryPoint:
     fld  real4 ptr [esi+eax*4]          ; 2 - AGI
     fadd real4 ptr [edi+eax*4]          ; 1
     fstp real4 ptr [esi+eax*4]          ; 5 - waits for fadd
     dec  eax                            ; 1
     jnz  LoopEntryPoint

; assume eax=Size-1, esi=array1, edi=array2
     jmp  LoopEntryPoint
     Align     16
TopOfLoop:
     fstp real4 ptr [esi+eax*4+4]        ; 4 - waits for fadd + AGI
LoopEntryPoint:
     fld  real4 ptr [esi+eax*4]          ;1
     fadd real4 ptr [edi+eax*4]          ;1
     dec  eax                            ;1
     jnz  TopOfLoop
     ;
fstp real4 ptr [esi+eax*4+4]
```

By moving the integer instructions between the `fadds` and `fstps`, the integer instructions can be executed while the `fadds` is completing in the floating-point unit and before the `fstps` begins execution. Note that this new loop structure requires a separate entry point for the first iteration because the loop needs to begin with the `flds`. Also, there needs to be an additional `fstps` after the conditional jump to finish the final loop iteration.

5.6.2 Hiding the One-Clock Latency of a Floating-Point Store

A floating-point store must wait an extra cycle for its floating-point operand. After an `fld`, an `fst` must wait one clock. After the common arithmetic operations, `fmul` and `fadd`, which normally have a latency of three, `fst` waits an extra cycle for a total of four[1].

```
fld       meml            ; 1 fld takes 1 clock
                          ; 2 fst waits, schedule something here
fst       mem2            ; 3,4 fst takes 2 clocks

fadd      meml            ; 1 add takes 3 clocks
                          ; 2 add, schedule something here
                          ; 3 add, schedule something here
                          ; 4 fst waits, schedule something here
fst       mem2            ; 5,2 fst takes 2 clocks
```

In the next example, the store is not dependent on the previous load:

```
fld       meml            ; 1
fld       mem2            ; 2
fxch      st(l)           ; 2
fst       mem3            ; 3 stores values loaded from meml
```

A register may be used immediately after it has been loaded (with `fld`):

```
fld       mem1                    ; l
fadd      mem2                    ; 2,3,4
```

Use of a register by a floating-point operation immediately after it has been written by another `fadd`, `fsub` or `fmul` causes a two-cycle delay. If instructions are inserted between these two, then latency and a potential stall can be hidden.

Additionally, there are multi-cycle floating-point instructions (`fdiv` and `fsqrt`) that execute in the floating-point unit pipe. While executing these instructions in the floating-point unit pipe, integer instructions can be executed in parallel. Emitting a number of integer instructions after such an instruction will keep the integer execution units busy (the exact number of instructions depends on the floating-point instruction's cycle count).

Integer instructions generally overlap with the floating-point operations except when the last floating-point operation was `fxch`. In this case there is a one-cycle delay:

```
U-pipe:                   V-pipe:
  fadd                    fxch      ; 1
                                    ; 2 fxch delay
  mov eax, 1              inc edx   ;
```

---

[1] This set also includes the `faddp, fsubrp, ...` instructions.

5.6.3 Integer and Floating-Point Multiply

The integer multiply operations, `mul` and `imul`, are executed in the floating-point unit so these instructions cannot be executed in parallel with a floating-point instruction.

A floating-point multiply instruction (`fmul`) delays for one cycle if the immediately preceding cycle executed an `fmul` or an `fmul` / `fxch` pair. The multiplier can only accept a new pair of operands every other cycle.

5.6.4 Floating-Point Operations with Integer Operands

Floating-point operations that take integer operands (`fiadd` or `fisub` ..) should be avoided. These instructions should be split into two instructions: `fild` and a floating-point operation. The number of cycles before another instruction can be issued (throughput) for `fiadd` is four, while for `fild` and simple floating-point operations it is one, as shown in the example below.

Complex Instructions:

```
fiadd  [ebp] ; 4
```

Better for Potential Overlap:

```
fild   [ebp]  ; 1
faddp  st(l)  ; 2
```

Using the `fild` - `faddp` instructions yields two free cycles for executing other instructions.

5.6.5 FSTSW Instructions

The `fstsw` instruction that usually appears after a floating-point comparison instruction (`fcom`, `fcomp`, `fcompp`) delays for three cycles. Other instructions may be inserted after the comparison instruction in order to hide the latency. On Pentium Pro and Pentium II processors the `fcmov` instruction can be used instead.

5.6.6 Transcendental Functions

Transcendental operations execute in the U-pipe and nothing can be overlapped with them, so an integer instruction following such an instruction will wait until that instruction completes.

Transcendental operations execute on Pentium Pro and Pentium II processors much faster. It may be worthwhile in-lining some of these math library calls because of the fact that the `call` and prologue/epilogue overhead involved with the library calls is no longer negligible. Emulating these operations in software will not be faster than the hardware unless accuracy is sacrificed.

5.6.7 Back-to-Back Floating-Point Instructions

The Pentium processor with MMX technology will stall when computing back to back floating-point multiply or division instructions when the operand exponents are in the ranges [-1FFE, -0FFF] and [1000,1FFE], in other words, for very big and very small extended precision numbers. The Pentium processor does not exhibit this stall.

For example executing:

```
FMUL ST0,ST1
FLD fld1
```

If the exponents added together produce a value in the above range, the FLD operation will wait to see if the FMUL operation produces an exception as a result of overflow (overflow is generated if ST0*ST1 > MAX).

intel®

6

Suggestions for Choosing a Compiler

CHAPTER 6
SUGGESTIONS FOR CHOOSING A COMPILER

Many compilers are available on the market today. The difficult question is which is the right compiler to use for the most optimized code. This chapter gives a list of suggestions on what to look for in a compiler; it also gives an overview the different optimization switches for compilation and summarizes the differences on the Intel Architecture. Finally, Section 6.2.4 recommends a blended strategy for code optimization.

6.1 IMPORTANT FEATURES FOR A COMPILER

Following is a list of features for consideration when choosing a compiler for application development. These are primarily performance-oriented features, and the order is not prioritized; an ISV/developer should weigh each element equally.

- The compiler should have switches that target specific processors (as described in Section 6.2) as well as a switch to generate “blended code”.
- The compiler should align all data sizes appropriately. It should also have the ability to align target branches to 16 bytes.
- The compiler should be able to perform interprocedural (whole program) analysis and optimization.
- The compiler should be able to perform profile-guided optimizations.
- The compiler should be able to provide a listing of the generated assembly code with line numbers and other annotations.
- The compiler should have good in-line assembly support. An added benefit exists when the compiler can optimize high level language code in the presence of in-line assembly.
- The compiler should perform “advanced optimizations” that target memory hierarchy, such as loop transforms as described in Section 3.5.1.5.
- The tools should provide the ability to debug optimized code. Generation of debug information is very important with respect to the VTune tuning environment.
- The compiler should provide support for MMX technology. Minimum support is with in-line assembly and a 64-bit data type. Best support is with intrinsic functions.
- The compiler should be reliable. It should produce correct code under all levels of optimization.

There are many other important issues to consider when purchasing a compiler that are related to usability. At a minimum, order an evaluation copy of the compilers that you are

considering, then benchmark the compiler on your application. This is the best information for your decision on which compiler to purchase.

6.2 COMPILER SWITCHES RECOMMENDATION

The following section summarizes the compiler switch recommendations for Intel Architecture compilers. The default for compilers should be a blended switch that optimizes for the family of processors. Switches specific to each processor should be offered as an alternative for application programmers.

6.2.1 Default (Blended Code)

Generates blended code. Code compiled with this switch will execute on all Intel Architecture processors (Intel386, Intel486, Pentium, Pentium Pro and Pentium II). This switch is intended for code which will possibly run on more than one processor. There should be no partial register stalls generated by the code generator when this switch is set.

6.2.2 Processor-Specific Switches

6.2.2.1 TARGET PROCESSOR — PENTIUM® PROCESSOR

Generates the best Pentium processor code. Code will run on all 32-bit Intel Architecture processors. This is intended for code which will run only on the Pentium processor.

6.2.2.2 TARGET PROCESSOR — PENTIUM® PRO PROCESSOR

Generates the best Pentium Pro processor code. Code will run on all 32-bit Intel Architecture processors. This is intended for code which will run only on Pentium Pro and Pentium II processors. There should be no partial stalls generated.

6.2.3 Other Switches

6.2.3.1 PENTIUM® PRO PROCESSOR NEW INSTRUCTIONS

This will use the new Pentium Pro processor specific instructions: `cmov`, `fcmov` and `fcomi`. This is independent of the Pentium Pro processor specific switch. If a target processor switch is also specified, the '`if to cmov`' optimization will be done depending on Pentium Pro processor style cost analysis.

6.2.3.2 OPTIMIZE FOR SMALL CODE SIZE

This switch optimizes for small code size. Execution speed will be sacrificed when necessary. An example is to use pushes rather than stores. This is intended for programs with high instruction cache miss rates. This switch also turns off code alignment, regardless of target processor.

6.2.4 Summary

The following tables summarize the micro architecture differences among the Pentium and Pentium Pro processors. The table lists the corresponding code generation considerations.

Table 6.1. Intel Microprocessor Architecture Differences

	Pentium® Processor	Pentium® Pro Processor	Pentium Processor with MMX™ Technology	Pentium II Processor
Cache	8K Code, 8K Data	8K Code, 8K Data	16K Code, 16K Data	16K Code, 16K Data
Prefetch	4x32b private bus to cache	4x32b private bus to cache	4x32b private bus to cache	4x32b private bus to cache
Decoder	2 decoders	3 decoders	2 decoders	3 decoders
Core	5 stages pipeline & superscalar	12 stages pipeline & Dynamic Execution	6 stages pipeline & superscalar	12 stages pipeline & Dynamic Execution
Math	On-Chip & pipelined	On-Chip and pipelined	On-Chip & pipelined	On-Chip and pipelined

Following are the recommendations for blended code across the Intel Architecture family:

- Important code entry points, such as a mispredicted label or an interrupt function, should be aligned on 16-byte boundaries.
- Avoid partial stalls.
- Schedule to remove address generation interlock and other pipeline stalls.
- Use simple instructions.
- Follow the branch prediction algorithm.

Schedule floating-point code to improve throughput.

7

Intel Architecture Performance Monitoring Extensions

CHAPTER 7
INTEL ARCHITECTURE PERFORMANCE MONITORING EXTENSIONS

The most effective way to improve the performance of application code is to find the performance bottlenecks in the code and remedy the stall conditions. In order to identify stall conditions, Intel Architecture processors include two counters on the processors that allow you to gather information about the performance of applications. The counters keep track of events that occur while your code is executing. The counters can be read during program execution. Using the counters, it is easier to determine if and where an application has stalls. The counters can be accessed by using Intel's VTune or by using the performance counter instructions within the application code.

The section describes the performance monitoring features on Pentium, Pentium Pro and Pentium II processors.

The RDPMC instruction is described in Section 7.3.

7.1 SUPERSCALAR (PENTIUM® PROCESSOR FAMILY) PERFORMANCE MONITORING EVENTS

All Pentium processors feature performance counters and several new events have been added to support MMX technology. All new events are assigned to one of the two event counters (CTR0, CTR1), with the exception of "twin events" (such as "D1 starvation" and "FIFO is empty") which are assigned to different counters to allow their concurrent measurement. The events must be assigned to their specified counter. Table 7-1 lists the performance monitoring events. New events are shaded.

Table 7-1. Performance Monitoring Events

Serial	Encoding	Counter 0	Counter 1	Performance Monitoring Event	Occurrence or Duration
0	000000	Yes	Yes	Data Read	OCCURRENCE
1	000001	Yes	Yes	Data Write	OCCURRENCE
2	000010	Yes	Yes	Data TLB Miss	OCCURRENCE
3	000011	Yes	Yes	Data Read Miss	OCCURRENCE
4	000100	Yes	Yes	Data Write Miss	OCCURRENCE
5	000101	Yes	Yes	Write (hit) to M or E state lines	OCCURRENCE
6	000110	Yes	Yes	Data Cache Lines Written Back	OCCURRENCE
7	000111	Yes	Yes	External Data Cache Snoops	OCCURRENCE
8	001000	Yes	Yes	External Data Cache Snoop Hits	OCCURRENCE
9	001001	Yes	Yes	Memory Accesses in Both Pipes	OCCURRENCE
10	001010	Yes	Yes	Bank Conflicts	OCCURRENCE
11	001011	Yes	Yes	Misaligned Data Memory or I/O References	OCCURRENCE
12	001100	Yes	Yes	Code Read	OCCURRENCE
13	001101	Yes	Yes	Code TLB Miss	OCCURRENCE
14	001110	Yes	Yes	Code Cache Miss	OCCURRENCE
15	001111	Yes	Yes	Any Segment Register Loaded	OCCURRENCE
16	010000	Yes	Yes	Reserved	
17	010001	Yes	Yes	Reserved	
18	010010	Yes	Yes	Branches	OCCURRENCE
19	010011	Yes	Yes	BTB Predictions	OCCURRENCE
20	010100	Yes	Yes	Taken Branch or BTB hit.	OCCURRENCE
21	010101	Yes	Yes	Pipeline Flushes	OCCURRENCE
22	010110	Yes	Yes	Instructions Executed	OCCURRENCE

Table 7-1. Performance Monitoring Events (Cont'd)

Serial	Encoding	Counter 0	Counter 1	Performance Monitoring Event	Occurrence or Duration
23	010111	Yes	Yes	Instructions Executed in the V-pipe e.g. parallelism/pairing	OCCURRENCE
24	**011000**	**Yes**	**Yes**	**Clocks while a bus cycle is in progress (bus utilization)**	**DURATION**
25	011001	Yes	Yes	Number of clocks stalled due to full write buffers	DURATION
26	011010	Yes	Yes	Pipeline stalled waiting for data memory read	DURATION
27	011011	Yes	Yes	Stall on write to an E or M state line	DURATION
29	011101	Yes	Yes	I/O Read or Write Cycle	OCCURRENCE
30	011110	Yes	Yes	Non-cacheable memory reads	OCCURRENCE
31	011111	Yes	Yes	Pipeline stalled because of an address generation interlock	DURATION
32	100000	Yes	Yes	Reserved	
33	100001	Yes	Yes	Reserved	
34	100010	Yes	Yes	FLOPs	OCCURRENCE
35	100011	Yes	Yes	Breakpoint match on DR0 Register	OCCURRENCE
36	100100	Yes	Yes	Breakpoint match on DR1 Register	OCCURRENCE
37	100101	Yes	Yes	Breakpoint match on DR2 Register	OCCURRENCE
38	100110	Yes	Yes	Breakpoint match on DR3 Register	OCCURRENCE
39	100111	Yes	Yes	Hardware Interrupts	OCCURRENCE
40	101000	Yes	Yes	Data Read or Data Write	OCCURRENCE
41	101001	Yes	Yes	Data Read Miss or Data Write Miss	OCCURRENCE

Table 7-1. Performance Monitoring Events (Cont'd)

43	101011	Yes	No	MMX™ instructions executed in U-pipe	OCCURRENCE
43	101011	No	Yes	MMX instructions executed in V-pipe	OCCURRENCE
45	101101	Yes	No	EMMS instructions executed	OCCURRENCE
45	101101	No	Yes	Transition between MMX instructions and FP instructions	OCCURRENCE
46	101110	No	Yes	Writes to Non-Cacheable Memory	OCCURRENCE
47	101111	Yes	No	Saturating MMX instructions executed	OCCURRENCE
47	101111	No	Yes	Saturations performed	OCCURRENCE
48	110000	Yes	No	Number of Cycles Not in HLT State	DURATION
49	110001	Yes	No	MMX instruction data reads	OCCURRENCE
50	110010	Yes	No	Floating-Point Stalls	DURATION
50	110010	No	Yes	Taken Branches	OCCURRENCE
51	110011	No	Yes	D1 Starvation and one instruction in FIFO	OCCURRENCE
52	110100	Yes	No	MMX instruction data writes	OCCURRENCE
52	110100	No	Yes	MMX instruction data write misses	OCCURRENCE
53	110101	Yes	No	Pipeline flushes due to wrong branch prediction	OCCURRENCE
53	110101	No	Yes	Pipeline flushes due to wrong branch predictions resolved in WB-stage	OCCURRENCE
54	110110	Yes	No	Misaligned data memory reference on MMX instruction	OCCURRENCE

Table 7-1. Performance Monitoring Events (Cont'd)

54	110110	No	Yes	Pipeline stalled waiting for MMX instruction data memory read	DURATION
55	110111	Yes	No	Returns Predicted Incorrectly	OCCURRENCE
55	110111	No	Yes	Returns Predicted (Correctly and Incorrectly)	OCCURRENCE
56	111000	Yes	No	MMX instruction multiply unit interlock	DURATION
56	111000	No	Yes	MOVD/MOVQ store stall due to previous operation	DURATION
57	111001	Yes	No	Returns	OCCURRENCE
57	111001	No	Yes	RSB Overflows	OCCURRENCE
58	111010	Yes	No	BTB false entries	OCCURRENCE
58	111010	No	Yes	BTB miss prediction on a Not-Taken Branch	OCCURRENCE
59	111011	Yes	No	Number of clocks stalled due to full write buffers while executing MMX instructions	DURATION
59	111011	No	Yes	Stall on MMX instruction write to E or M line	DURATION

7.1.1 Description of MMX™ Instruction Events

The event codes/counter are provided in parentheses.

- **MMX instructions executed in U-pipe (101011/0):**
  Total number of MMX instructions executed in the U-pipe.
- **MMX instructions executed in V-pipe (101011/1):**
  Total number of MMX instructions executed in the V-pipe.
- **EMMS instructions executed (101101/0):**
  Counts number of EMMS instructions executed.

- **Transition between MMX instructions and FP instructions (101101/1):**

  Counts first floating-point instruction following any MMX instruction or first MMX instruction following a floating-point instruction. This count can be used to estimate the penalty in transitions between FP state and MMX state. An even count indicates the processor is in the MMX state. An odd count indicates it is in the FP state.

- **Writes to non-cacheable memory (101110/1):**

  Counts the number of write accesses to non-cacheable memory. It includes write cycles caused by TLB misses and I/O write cycles. Cycles restarted due to BOFF# are not re-counted.

- **Saturating MMX instructions executed (101111/0):**

  Counts saturating MMX instructions executed, independently of whether or not they actually saturated. Saturating MMX instructions may perform add, subtract or pack operations .

- **Saturations performed (101111/1):**

  Counts the number of MMX instructions that used saturating arithmetic where at least one of the results actually saturated (that is, if an MMX instruction operating on four dwords saturated in three out of the four results, the counter will be incremented by only one).

- **Number of cycles not in HALT (HLT) state (110000/0):**

  Counts the number of cycles the processor is not idle due to a HALT (HLT) instruction. Use this event to calculate “net CPI.” Note that during the time the processor is executing the HLT instruction, the Time Stamp Counter (TSC) is not disabled. Since this event is controlled by the Counter Controls CC0, CC1 it can be used to calculate the CPI at CPL=3, which the TSC cannot provide.

- **MMX instruction data reads (110001/0):**

  Analogous to “Data reads”, counting only MMX instruction accesses.

- **MMX instruction data read misses (110001/1):**

  Analogous to “Data read misses”, counting only MMX instruction accesses.

- **Floating-Point stalls (110010/0):**

  Counts the number of clocks while pipe is stalled due to a floating-point freeze.

- **Number of Taken Branches (110010/1):**

  Counts the number of Taken Branches.

- **D1 starvation and FIFO is empty (110011/0), D1 starvation and only one instruction in FIFO (110011/1):**

  The D1 stage can issue 0, 1 or 2 instructions per clock if instructions are available in the FIFO buffer. The first event counts how many times D1 cannot issue ANY instructions because the FIFO buffer is empty. The second event counts how many times the D1 stage issues just a single instruction because the FIFO buffer had just one instruction ready. Combined with two other events, Instruction Executed (010110) and Instruction

Executed in the V-pipe (010110), the second event lets you calculate the number of times pairing rules prevented issue of two instructions.

- **MMX instruction data writes (110001/1):**

  Analogous to “Data writes”, counting only MMX instruction accesses.

- **MMX instruction data write misses (110100/1):**

  Analogous to “Data write misses”, counting only MMX instruction accesses.

- **Pipeline flushes due to wrong branch prediction (110101/0); Pipeline flushes due to wrong branch prediction resolved in WB-stage(110101/1):**

  Counts any pipeline flush due to a branch which the pipeline did not follow correctly. It includes cases where a branch was not in the BTB, cases where a branch was in the BTB but was mispredicted, and cases where a branch was correctly predicted but to the wrong address. Branches are resolved in either the Execute (E) stage or the Writeback (WB) stage. In the latter case, the misprediction penalty is larger by one clock. The first event listed above counts the number of incorrectly predicted branches resolved in either the E stage or the WB stage. The second event counts the number of incorrectly predicted branches resolved in the WB stage. The difference between these two counts is the number of E stage-resolved branches.

- **Misaligned data memory reference on MMX instruction (110110/0):**

  Analogous to “Misaligned data memory reference,” counting only MMX instruction accesses.

- **Pipeline stalled waiting for data memory read (110110/1):**

  Analogous to “Pipeline stalled waiting for data memory read,” counting only MMX accesses.

- **Returns predicted incorrectly or not predicted at all (110111/0):**

  The actual number of Returns that were either incorrectly predicted or were not predicted at all. It is the difference between the total number of executed returns and the number of returns that were correctly predicted. Only RET instructions are counted (that is, IRET instructions are not counted).

- **Returns predicted (correctly and incorrectly) (110111/1):**

  The actual number of Returns for which a prediction was made. Only RET instructions are counted (that is, IRET instructions are not counted).

- **MMX multiply unit interlock (111000/0):**

  Counts the number of clocks the pipe is stalled because the destination of a previous MMX multiply instruction is not yet ready. The counter will not be incremented if there is another cause for a stall. For each occurrence of a multiply interlock, this event may be counted twice (if the stalled instruction comes on the next clock after the multiply) or only once (if the stalled instruction comes two clocks after the multiply).

- **MOVD/MOVQ store stall due to previous operation (111000/1):**

  Number of clocks a MOVD/MOVQ store is stalled in the D2 stage due to a previous MMX operation with a destination to be used in the store instruction.

- **Returns (111001/0):**

  The actual number of Returns executed. Only RET instructions are counted (that is, IRET instructions are not counted). Any exception taken on a RET instruction also updates this counter.

- **RSB overflows (111001/1):**

  Counts the number of times the Return Stack Buffer (RSB) cannot accommodate a call address.

- **BTB false entries (111010/0):**

  Counts the number of false entries in the Branch Target Buffer. False entries are causes for misprediction other than a wrong prediction.

- **BTB miss-prediction on a Not-Taken Branch (111010/1):**

  Counts the number of times the BTB predicted a Not-Taken **Branch** as Taken.

- **Number of clocks stalled due to full write buffers while executing MMX instructions (111011/0):**

  Analogous to "Number of clocks stalled due to full write buffers," counting only MMX instruction accesses.

- **Stall on MMX instruction write to an E or M state line (111011/1):**

  Analogous to "Stall on write to an E or M state line," counting only MMX instruction accesses.

7.2 PENTIUM® PRO AND PENTIUM II PERFORMANCE MONITORING EVENTS

This section describes the counters on Pentium Pro and Pentium II processors. Table 7-2 lists the events that can be counted with the performance-monitoring counters and read with the RDPMC instruction.

In the table:

- The Unit column gives the microarchitecture or bus unit that produces the event.
- The Event Number column gives the hexadecimal number identifying the event.
- The Mnemonic Event Name column gives the name of the event.
- The Unit Mask column gives the unit mask required (if any).
- The Description column describes the event.
- The Comments column gives additional information about the event.

These performance monitoring events are intended to be used as guides for performance tuning. The counter values reported are not guaranteed to be absolutely accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable. All performance events are model-specific to the Pentium Pro processor family and are not architecturally guaranteed in future versions of the processor. All performance

event encodings not listed in the table are reserved and their use will result in undefined counter results.

See the end of the table for notes related to certain entries in the table.

Table 7-2. Performance Monitoring Counters

Unit	Event No.	Mnemonic Event Name	Unit Mask	Description	Comments
Data Cache Unit (DCU)	43H	DATA_MEM_ REFS	00H	All loads from any memory type. All stores to any memory type. Each part of a split is counted separately. **NOTE**: 80-bit floating-point accesses are double counted, since they are decomposed into a 16 bit exponent load and a 64 bit mantissa load. Memory accesses are only counted when they are actually performed, e.g., a load that gets squashed because a previous cache miss is outstanding to the same address, and which finally gets performed, is only counted once. Does not include I/O accesses, or other non-memory accesses.	
	45H	DCU_LINES_IN	00H	Total number of lines that have been allocated in the DCU.	
	46H	DCU_M_LINES_ IN	00H	Number of Modified state lines that have been allocated in the DCU.	
	47H	DCU_M_LINES_ OUT	00H	Number of Modified state lines that have been evicted from the DCU. This includes evictions as a result of external snoops, internal intervention or the natural replacement algorithm.	

Table 7-2. Performance Monitoring Counters (Cont'd)

Unit	Event No.	Mnemonic Event Name	Unit Mask	Description	Comments
Data Cache Unit (DCU) (Cont'd)	48H	DCU_MISS_OUTSTANDING	00H	Weighted number of cycles while a DCU miss is outstanding. Incremented by the number of outstanding cache misses at any particular time. Cacheable read requests only are considered. Uncacheable requests are excluded. Read-for-ownerships are counted as well as line fills, invalidates and stores.	An access that also misses the L2 is short-changed by two cycles. (i.e. if count is N cycles, should be N+2 cycles.) Subsequent loads to the same cache line will not result in any additional counts. Count value not precise, but still useful.
Instruction Fetch Unit (IFU)	80H	IFU_IFETCH	00H	Number of instruction fetches, both cacheable and non-cacheable. Including UC fetches.	Will be incremented by 1 for each cacheable line fetched and by 1 for each uncached instruction fetched.
	81H	IFU_IFETCH_MISS	00H	Number of instruction fetch misses. All instruction fetches that do not hit the IFU i.e. that produce memory requests. Includes UC accesses.	
	85H	ITLB_MISS	00H	Number of ITLB misses.	
	86H	IFU_MEM_STALL	00H	Number of cycles instruction fetch is stalled, for any reason. Includes IFU cache misses, ITLB misses, ITLB faults and other minor stalls.	
	87H	ILD_STALL	00H	Number of cycles that the instruction length decoder stage of the processors pipeline is stalled.	
L2 Cache	28H	L2_IFETCH	MESI 0FH	Number of L2 instruction fetches. This event indicates that a normal instruction fetch was received by the L2. The count includes only L2 cacheable instruction fetches; it does not include UC instruction fetches. It does not include ITLB miss accesses.	

Table 7-2. Performance Monitoring Counters (Cont'd)

Unit	Event No.	Mnemonic Event Name	Unit Mask	Description	Comments
L2 Cache (Cont'd)	29H	L2_LD	MESI 0FH	Number of L2 data loads. This event indicates that a normal, unlocked, load memory access was received by the L2. It includes only L2 cacheable memory accesses; it does not include I/O accesses, other non-memory accesses, or memory accesses such as UC/WT memory accesses. It does include L2 cacheable TLB miss memory accesses.	
	2AH	L2_ST	MESI 0FH	Number of L2 data stores. This event indicates that a normal, unlocked, store memory access was received by the L2. Specifically, it indicates that the DCU sent a read-for-ownership request to the L2. It also includes Invalid to Modified requests sent by the DCU to the L2. It includes only L2 cacheable store memory accesses; it does not include I/O accesses, other non-memory accesses, or memory accesses like UC/WT stores. It includes TLB miss memory accesses.	
	24H	L2_LINES_IN	00H	Number of lines allocated in the L2.	
	26H	L2_LINES_OUT	00H	Number of lines removed from the L2 for any reason.	
	25H	L2_M_LINES_IN M	00H	Number of Modified state lines allocated in the L2.	
	27H	L2_M_LINES_ OUTM	00H	Number of Modified state lines removed from the L2 for any reason.	
	2EH	L2_RQSTS	MESI 0FH	Total number of all L2 requests.	
	21H	L2_ADS	00H	Number of L2 address strobes.	
	22H	L2_DBUS_BUSY	00H	Number of cycles during which the L2 cache data bus was busy.	

Table 7-2. Performance Monitoring Counters (Cont'd)

Unit	Event No.	Mnemonic Event Name	Unit Mask	Description	Comments
L2 Cache (Cont'd)	23H	L2_DBUS_BUSY_RD	00H	Number of cycles during which the data bus was busy transferring read data from L2 to the processor.	
External Bus Logic (EBL) (2)	62H	BUS_DRDY_CLOCKS	00H (Self) 20H (Any)	Number of clocks during which DRDY# is asserted. Essentially, utilization of the external system data bus.	Unit Mask = 00H counts bus clocks when the processor is driving DRDY Unit Mask = 20H counts in processor clocks when any agent is driving DRDY.
	63H	BUS_LOCK_CLOCKS	00H (Self) 20H (Any)	Number of clocks during which LOCK# is asserted on the external system bus.	Always counts in processor clocks
	60H	BUS_REQ_OUTSTANDING	00H (Self)	Number of bus requests outstanding. This counter is incremented by the number of cacheable read bus requests outstanding in any given cycle.	Counts only DCU full-line cacheable reads, not Reads for ownership, writes, instruction fetches, or anything else. Counts "waiting for bus to complete" (last data chunk received).
	65H	BUS_TRAN_BRD	00H (Self) 20H (Any)	Number of bus burst read transactions.	
	66H	BUS_TRAN_RFO	00H (Self) 20H (Any)	Number of completed bus read for ownership transactions.	
	67H	BUS_TRANS_WB	00H (Self) 20H (Any)	Number of completed bus write back transactions.	
	68H	BUS_TRAN_IFETCH	00H (Self) 20H (Any)	Number of completed bus instruction fetch transactions.	
	69H	BUS_TRAN_INVAL	00H (Self) 20H (Any)	Number of completed bus invalidate transactions.	
	6AH	BUS_TRAN_PWR	00H (Self) 20H (Any)	Number of completed bus partial write transactions.	

Table 7-2. Performance Monitoring Counters (Cont'd)

Unit	Event No.	Mnemonic Event Name	Unit Mask	Description	Comments
External Bus Logic (EBL) (2) (Cont'd)	6BH	BUS_TRANS_P	00H (Self) 20H (Any)	Number of completed bus partial transactions.	
	6CH	BUS_TRANS_IO	00H (Self) 20H (Any)	Number of completed bus I/O transactions.	
	6DH	BUS_TRAN_DEF	00H (Self) 20H (Any)	Number of completed bus deferred transactions.	
	6EH	BUS_TRAN_ BURST	00H (Self) 20H (Any)	Number of completed bus burst transactions.	
	70H	BUS_TRAN_ANY	00H (Self) 20H (Any)	Number of all completed bus transactions. Address bus utilization can be calculated knowing the minimum address bus occupancy. Includes special cycles etc.	
	6FH	BUS_TRAN_ME M	00H (Self) 20H (Any)	Number of completed memory transactions.	
	64H	BUS_DATA_RCV	00H (Self)	Number of bus clock cycles during which this processor is receiving data.	
	61H	BUS_BNR_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the BNR pin.	
	7AH	BUS_HIT_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the HIT pin.	Includes cycles due to snoop stalls.
	7BH	BUS_HITM_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the HITM pin.	Includes cycles due to snoop stalls.
	7EH	BUS_SNOOP_ STALL	00H (Self)	Number of clock cycles during which the bus is snoop stalled.	

Table 7-2. Performance Monitoring Counters (Cont'd)

Unit	Event No.	Mnemonic Event Name	Unit Mask	Description	Comments
Floating-Point Unit	C1H	FLOPS	00H	Number of computational floating-point operations retired. Excludes floating-point computational operations that cause traps or assists. Includes floating-point computational operations executed by the assist handler. Includes internal sub-operations of complex floating-point instructions such as a transcendental instruction. Excludes floating-point loads and stores.	Counter 0 only.
	10H	FP_COMP_OPS_EXE	00H	Number of computational floating-point operations executed. The number of FADD, FSUB, FCOM, FMULs, integer MULs and IMULs, FDIVs, FPREMs, FSQRTS, integer DIVs and IDIVs. Note counts the number of operations not number of cycles. This event does not distinguish an FADD used in the middle of a transcendental flow from a separate FADD instruction.	Counter 0 only.
	11H	FP_ASSIST	00H	Number of floating-point exception cases handled by microcode.	Counter 1 only. This event includes counts due to speculative execution.
	12H	MUL	00H	Number of multiplies. **NOTE**: includes integer and FP multiplies.	Counter 1 only. This event includes counts due to speculative execution.
	13H	DIV	00H	Number of divides. **NOTE**: includes integer and FP multiplies.	Counter 1 only. This event includes counts due to speculative execution.
	14H	CYCLES_DIV_BUSY	00H	Number of cycles that the divider is busy, and cannot accept new divides. **NOTE**: includes integer and FP divides, FPREM, FPSQRT, etc.	Counter 0 only. This event includes counts due to speculative execution.

Table 7-2. Performance Monitoring Counters (Cont'd)

Unit	Event No.	Mnemonic Event Name	Unit Mask	Description	Comments
Memory Ordering	03H	LD_BLOCKS	00H	Number of store buffer blocks. Includes counts caused by preceding stores whose addresses are unknown, preceding stores whose addresses are known to conflict, but whose data is unknown and preceding stores that conflicts with the load, but which incompletely overlap the load.	
	04H	SB_DRAINS	00H	Number of store buffer drain cycles. Incremented during every cycle the store buffer is draining. Draining is caused by serializing operations like CPUID, synchronizing operations like XCHG, Interrupt acknowledgment, as well as other conditions such as cache flushing.	
	05H	MISALIGN_MEM _REF	00H	Number of misaligned data memory references. Incremented by 1 every cycle during which either the Pentium® Pro load or store pipeline dispatches a misaligned micro-op. Counting is performed if its the first half or second half, or if it is blocked, squashed or misses. Note in this context misaligned means crossing a 64-bit boundary.	It should be noted that MISALIGN_MEM_REF is only an approximation, to the true number of misaligned memory references. The value returned is roughly proportional to the number of misaligned memory accesses, i.e., the size of the problem.
Instruction Decoding and Retirement	C0H	INST_RETIRED	OOH	Total number of instructions retired.	
	C2H	UOPS_RETIRED	00H	Total umber of micro-ops retired.	
	D0H	INST_DECODER	00H	Total number of instructions decoded.	
Interrupts	C8H	HW_INT_RX	00H	Total number of hardware interrupts received.	
	C6H	CYCLES_INT_ MASKED	00H	Total number of processor cycles for which interrupts are disabled.	

Table 7-2. Performance Monitoring Counters (Cont'd)

Unit	Event No.	Mnemonic Event Name	Unit Mask	Description	Comments
Interrupts (Cont'd)	C7H	CYCLES_INT_PENDING_AND_MASKED	00H	Total number of processor cycles for which interrupts are disabled and interrupts are pending.	
Branches	C4H	BR_INST_RETIRED	00H	Total number of branch instructions retired.	
	C5H	BR_MISS_PRED_RETIRED	00H	Total number of branch mispredictions that get to the point of retirement. Includes not taken conditional branches.	
	C9H	BR_TAKEN_RETIRED	00H	Total number of taken branches retired.	
	CAH	BR_MISS_PRED_TAKEN_RET	00H	Total number of taken but mispredicted branches that get to the point of retirement. Includes conditional branches only when taken.	
	E0H	BR_INST_DECODED	00H	Total number of branch instructions decoded.	
	E2H	BTB_MISSES	00H	Total number of branches that the BTB did not produce a prediction	
	E4H	BR_BOGUS	00H	Total number of branch predictions that are generated but are not actually branches.	
	E6H	BACLEARS	00H	Total number of time BACLEAR is asserted. This is the number of times that a static branch prediction was made by the decoder.	

Table 7-2. Performance Monitoring Counters (Cont'd)

Unit	Event No.	Mnemonic Event Name	Unit Mask	Description	Comments
Stalls	A2H	RESOURCE_ STALLS	00H	Incremented by one during every cycle that there is a resource related stall. Includes register renaming buffer entries, memory buffer entries. Does not include stalls due to bus queue full, too many cache misses, etc. In addition to resource related stalls, this event counts some other events. Includes stalls arising during branch misprediction recovery e.g. if retirement of the mispredicted branch is delayed and stalls arising while store buffer is draining from synchronizing operations.	
	D2H	PARTIAL_RAT_ STALLS	00H	Number of cycles or events for partial stalls. **NOTE**: Includes flag partial stalls.	
Segment Register Loads	06H	SEGMENT_REG _LOADS	00H	Number of segment register loads.	
Clocks	79H	CPU_CLK_ UNHALTED	00H	Number of cycles during which the processor is not halted.	
MMX™ Technology Instruction Events					
MMX Instructions Executed	B0H	MMX_INSTR_ EXEC	00H	Number of MMX instructions executed.	
MMX Saturating Instructions Executed	B1H	MMX_SAT_ INSTR_EXEC	00H	Number of MMX saturating instructions executed.	
MMX µops Executed	B2H	MMX_UOPS_ EXEC	0FH	Number of MMX µops executed.	

Table 7-2. Performance Monitoring Counters (Cont'd)

Unit	Event No.	Mnemonic Event Name	Unit Mask	Description	Comments
MMX Instructions Executed	B3H	MMX_INSTR_ TYPE_EXEC	01H	MMX Packed multiply instructions executed	
			02H	MMX Packed shift instructions executed	
			04H	MMX Pack operations instructions executed	
			08H	MMX Unpack operations instructions executed	
			10H	MMX Packed logical instructions executed	
			20H	MMX Packed arithmetic instructions executed	
MMX Transitions	CCH	FP_MMX_ TRANS	00H	Transitions from MMX instruction to FP instructions.	
			01H	Transitions from FP instructions to MMX instructions.	
MMX Assists	CDH	MMX_ASSIST	00H	Number of MMX Assists.	MMX Assists is the number of EMMS instructions executed.
MMX Instructions Retired	CEH	MMX_INSTR_ RET	00H	Number of MMX instructions retired.	
Segment Register Renaming Stalls	D4H	SEG_RENAME_ STALLS	01H	Segment register ES	
			02H	Segment register DS	
			04H	Segment register FS	
			08H	Segment register FS	
			0FH	Segment registers ES + DS + FS + GS	
Segment Registers Renamed	D5H	SEG_REG_ RENAMES	01H	Segment register ES	
			02H	Segment register DS	
			04H	Segment register FS	
			08H	Segment register FS	
			0FH	Segment registers ES + DS + FS + GS	
Segment Registers Renamed & Retired	D6H	RET_SEG_ RENAMES	00H	Number of segment register rename events retired.	

NOTES:

1. Several L2 cache events, where noted, can be further qualified using the Unit Mask (UMSK) field in the PerfEvtSel0 and PerfEvtSel1 registers. The lower four bits of the Unit Mask field are used in conjunction with L2 events to indicate the cache state or cache states involved. The Pentium® Pro processor family identifies cache states using the "MESI" protocol, and consequently each bit in the Unit Mask field represents one of the four states: UMSK[3] = M (8h) state, UMSK[2] = E (4h) state, UMSK[1] = S (2h) state, and UMSK[0] = I (1h) state. UMSK[3:0] = MESI (Fh) should be used to collect data for all states; UMSK = 0h, for the applicable events, will result in nothing being counted.
2. All of the external bus logic (EBL) events, except where noted, can be further qualified using the Unit Mask (UMSK) field in the PerfEvtSel0 and PerfEvtSel1 registers. Bit 5 of the UMSK field is used in conjunction with the EBL events to indicate whether the processor should count transactions that are self generated (UMSK[5] = 0) or transactions that result from any processor on the bus (UMSK[5] = 1).

7.3 RDPMC INSTRUCTION

The RDPMC (Read Processor Monitor Counter) instruction lets you read the performance monitoring counters in CPL=3 if bit #8 is set in the CR4 register (CR4.PCE). This is similar to the RDTSC (Read Time Stamp Counter) instruction, which is enabled in CPL=3 if the Time Stamp Disable bit in CR4 (CR4.TSD) is not disabled. Note that access to the performance monitoring Control and Event Select Register (CESR) is not possible in CPL=3.

7.3.1 Instruction Specification

Opcode: 0F 33

Description: Read event monitor counters indicated by ECX into EDX:EAX

Operation: EDX:EAX ← Event Counter [ECX]

The value in ECX (either 0 or 1) specifies one of the two 40-bit event counters of the processor. EDX is loaded with the high-order 32 bits, and EAX with the low-order 32 bits.

```
IF CR4.PCE = 0 AND CPL <> 0 THEN # GP(0)
IF ECX = 0 THEN EDX:EAX := PerfCntr0
IF ECX = 1 THEN EDX:EAX := PerfCntr1
ELSE #GP(0)
END IF
```

Protected & Real Address Mode Exceptions:

#GP(0) if ECX does not specify a valid counter (either 0 or 1).

#GP(0) if RDPMC is used in CPL<> 0 and CR4.PCE = 0

Remarks:

16-bit code: RDPMC will execute in 16-bit code and VM mode but will give a 32-bit result. It will use the full ECX index.

A

Integer Pairing Tables

APPENDIX A
INTEGER PAIRING TABLES

The following abbreviations are used in the Pairing column of the integer table in this appendix:

NP — Not pairable, executes in U-pipe

PU — Pairable if issued to U-pipe

PV — Pairable if issued to V-pipe

UV — Pairable in either pipe

The I/O instructions are not pairable.

A.1 INTEGER INSTRUCTION PAIRING TABLES

Table A-1. Integer Instruction Pairing

Instruction	Format	Pairing
AAA — ASCII Adjust after Addition		NP
AAD — ASCII Adjust AX before Division		NP
AAM — ASCII Adjust AX after Multiply		NP
AAS — ASCII Adjust AL after Subtraction		NP
ADC — ADD with Carry		PU
ADD — Add		UV
AND — Logical AND		UV
ARPL — Adjust RPL Field of Selector		NP
BOUND — Check Array Against Bounds		NP
BSF — Bit Scan Forward		NP
BSR — Bit Scan Reverse		NP
BSWAP — Byte Swap		NP
BT — Bit Test		NP
BTC — Bit Test and Complement		NP
BTR — Bit Test and Reset		NP
BTS — Bit Test and Set		NP

Table A-1. Integer Instruction Pairing (Cont'd)

Instruction	Format	Pairing
CALL — Call Procedure (in same segment)		
direct	1110 1000 : full displacement	PV
register indirect	1111 1111 : 11 010 reg	NP
memory indirect	1111 1111 : mod 010 r/m	NP
CALL — Call Procedure (in other segment)		NP
CBW — Convert Byte to Word CWDE — Convert Word to Doubleword		NP
CLC — Clear Carry Flag		NP
CLD — Clear Direction Flag		NP
CLI — Clear Interrupt Flag		NP
CLTS — Clear Task-Switched Flag in CR0		NP
CMC — Complement Carry Flag		NP
CMP — Compare Two Operands		UV
CMPS/CMPSB/CMPSW/CMPSD — Compare String Operands		NP
CMPXCHG — Compare and Exchange		NP
CMPXCHG8B — Compare and Exchange 8 Bytes		NP
CWD — Convert Word to Dword CDQ — Convert Dword to Qword		NP
DAA — Decimal Adjust AL after Addition		NP
DAS — Decimal Adjust AL after Subtraction		NP
DEC — Decrement by 1		UV
DIV — Unsigned Divide		NP
ENTER — Make Stack Frame for Procedure Parameters		NP
HLT — Halt		
IDIV — Signed Divide		NP
IMUL — Signed Multiply		NP
INC — Increment by 1		UV
INT n — Interrupt Type n		NP
INT — Single-Step Interrupt 3		NP
INTO — Interrupt 4 on Overflow		NP

Table A-1. Integer Instruction Pairing (Cont'd)

Instruction	Format	Pairing
INVD — Invalidate Cache		NP
INVLPG — Invalidate TLB Entry		NP
IRET/IRETD — Interrupt Return		NP
Jcc — Jump if Condition is Met		PV
JCXZ/JECXZ — Jump on CX/ECX Zero		NP
JMP — Unconditional Jump (to same segment)		
short	1110 1011 : 8-bit displacement	PV
direct	1110 1001 : full displacement	PV
register indirect	1111 1111 : 11 100 reg	NP
memory indirect	1111 1111 : mod 100 r/m	NP
JMP — Unconditional Jump (to other segment)		NP
LAHF — Load Flags into AH Register		NP
LAR — Load Access Rights Byte		NP
LDS — Load Pointer to DS		NP
LEA — Load Effective Address		UV
LEAVE — High Level Procedure Exit		NP
LES — Load Pointer to ES		NP
LFS — Load Pointer to FS		NP
LGDT — Load Global Descriptor Table Register		NP
LGS — Load Pointer to GS		NP
LIDT — Load Interrupt Descriptor Table Register		NP
LLDT — Load Local Descriptor Table Register		NP
LMSW — Load Machine Status Word		NP
LOCK — Assert LOCK# Signal Prefix		
LODS/LODSB/LODSW/LODSD — Load String Operand		NP
LOOP — Loop Count		NP
LOOPZ/LOOPE — Loop Count while Zero/Equal		NP
LOOPNZ/LOOPNE — Loop Count while not Zero/Equal		NP
LSL — Load Segment Limit		NP
LSS — Load Pointer to SS	0000 1111 : 1011 0010 : mod reg r/m	NP

Table A-1. Integer Instruction Pairing (Cont'd)

Instruction	Format	Pairing
LTR — Load Task Register		NP
MOV — Move Data		UV
MOV — Move to/from Control Registers		NP
MOV — Move to/from Debug Registers		NP
MOV — Move to/from Segment Registers		NP
MOVS/MOVSB/MOVSW/MOVSD — Move Data from String to String		NP
MOVSX — Move with Sign-Extend		NP
MOVZX — Move with Zero-Extend		NP
MUL — Unsigned Multiplication of AL, AX or EAX		NP
NEG — Two's Complement Negation		NP
NOP — No Operation	1001 0000	UV
NOT — One's Complement Negation		NP
OR — Logical Inclusive OR		UV
POP — Pop a Word from the Stack		
reg	1000 1111 : 11 000 reg	UV
or	0101 1 reg	UV
memory	1000 1111 : mod 000 r/m	NP
POP — Pop a Segment Register from the Stack		NP
POPA/POPAD — Pop All General Registers		NP
POPF/POPFD — Pop Stack into FLAGS or EFLAGS Register		NP
PUSH — Push Operand onto the Stack		
reg	1111 1111 : 11 110 reg	UV
or	0101 0 reg	UV
memory	1111 1111 : mod 110 r/m	NP
immediate	0110 10s0 : immediate data	UV
PUSH — Push Segment Register onto the Stack		NP
PUSHA/PUSHAD — Push All General Registers		NP
PUSHF/PUSHFD — Push Flags Register onto the Stack		NP
RCL — Rotate thru Carry Left		

Table A-1. Integer Instruction Pairing (Cont'd)

Instruction	Format	Pairing
reg by 1	1101 000w : 11 010 reg	PU
memory by 1	1101 000w : mod 010 r/m	PU
reg by CL	1101 001w : 11 010 reg	NP
memory by CL	1101 001w : mod 010 r/m	NP
reg by immediate count	1100 000w : 11 010 reg : imm8 data	PU
memory by immediate count	1100 000w : mod 010 r/m : imm8 data	PU
RCR — Rotate thru Carry Right		
reg by 1	1101 000w : 11 011 reg	PU
memory by 1	1101 000w : mod 011 r/m	PU
reg by CL	1101 001w : 11 011 reg	NP
memory by CL	1101 001w : mod 011 r/m	NP
reg by immediate count	1100 000w : 11 011 reg : imm8 data	PU
memory by immediate count	1100 000w : mod 011 r/m : imm8 data	PU
RDMSR — Read from Model-Specific Register		NP
REP LODS — Load String		NP
REP MOVS — Move String		NP
REP STOS — Store String		NP
REPE CMPS — Compare String (Find Non-Match)		NP
REPE SCAS — Scan String (Find Non-AL/AX/EAX)		NP
REPNE CMPS — Compare String (Find Match)		NP
REPNE SCAS — Scan String (Find AL/AX/EAX)		NP
RET — Return from Procedure (to same segment)		NP
RET — Return from Procedure (to other segment)		NP
ROL — Rotate (not thru Carry) Left		
reg by 1	1101 000w : 11 000 reg	PU
memory by 1	1101 000w : mod 000 r/m	PU
reg by CL	1101 001w : 11 000 reg	NP
memory by CL	1101 001w : mod 000 r/m	NP
reg by immediate count	1100 000w : 11 000 reg : imm8 data	PU

Table A-1. Integer Instruction Pairing (Cont'd)

Instruction	Format	Pairing
memory by immediate count	1100 000w : mod 000 r/m : imm8 data	PU
ROR — Rotate (not thru Carry) Right		
reg by 1	1101 000w : 11 001 reg	PU
memory by 1	1101 000w : mod 001 r/m	PU
reg by CL	1101 001w : 11 001 reg	NP
memory by CL	1101 001w : mod 001 r/m	NP
reg by immediate count	1100 000w : 11 001 reg : imm8 data	PU
memory by immediate count	1100 000w : mod 001 r/m : imm8 data	PU
RSM — Resume from System Management Mode		NP
SAHF — Store AH into Flags		NP
SAL — Shift Arithmetic Left same instruction as SHL		
SAR — Shift Arithmetic Right		
reg by 1	1101 000w : 11 111 reg	PU
memory by 1	1101 000w : mod 111 r/m	PU
reg by CL	1101 001w : 11 111 reg	NP
memory by CL	1101 001w : mod 111 r/m	NP
reg by immediate count	1100 000w : 11 111 reg : imm8 data	PU
memory by immediate count	1100 000w : mod 111 r/m : imm8 data	PU
SBB — Integer Subtraction with Borrow		PU
SCAS/SCASB/SCASW/SCASD — Scan String		NP
SETcc — Byte Set on Condition		NP
SGDT — Store Global Descriptor Table Register		NP
SHL — Shift Left		
reg by 1	1101 000w : 11 100 reg	PU
memory by 1	1101 000w : mod 100 r/m	PU
reg by CL	1101 001w : 11 100 reg	NP
memory by CL	1101 001w : mod 100 r/m	NP
reg by immediate count	1100 000w : 11 100 reg : imm8 data	PU

Table A-1. Integer Instruction Pairing (Cont'd)

Instruction	Format	Pairing
memory by immediate count	1100 000w : mod 100 r/m : imm8 data	PU
SHLD — Double Precision Shift Left		
register by immediate count	0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8	NP
memory by immediate count	0000 1111 : 1010 0100 : mod reg r/m : imm8	NP
register by CL	0000 1111 : 1010 0101 : 11 reg2 reg1	NP
memory by CL	0000 1111 : 1010 0101 : mod reg r/m	NP
SHR — Shift Right		
reg by 1	1101 000w : 11 101 reg	PU
memory by 1	1101 000w : mod 101 r/m	PU
reg by CL	1101 001w : 11 101 reg	NP
memory by CL	1101 001w : mod 101 r/m	NP
reg by immediate count	1100 000w : 11 101 reg : imm8 data	PU
memory by immediate count	1100 000w : mod 101 r/m : imm8 data	PU
SHRD — Double Precision Shift Right		
register by immediate count	0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8	NP
memory by immediate count	0000 1111 : 1010 1100 : mod reg r/m : imm8	NP
register by CL	0000 1111 : 1010 1101 : 11 reg2 reg1	NP
memory by CL	0000 1111 : 1010 1101 : mod reg r/m	NP
SIDT — Store Interrupt Descriptor Table Register		NP
SLDT — Store Local Descriptor Table Register		NP
SMSW — Store Machine Status Word		NP
STC — Set Carry Flag		NP
STD — Set Direction Flag		NP
STI — Set Interrupt Flag		
STOS/STOSB/STOSW/STOSD — Store String Data		NP

Table A-1. Integer Instruction Pairing (Cont'd)

Instruction	Format	Pairing
STR — Store Task Register		NP
SUB — Integer Subtraction		UV
TEST — Logical Compare		
reg1 and reg2	1000 010w : 11 reg1 reg2	UV
memory and register	1000 010w : mod reg r/m	UV
immediate and register	1111 011w : 11 000 reg : immediate data	NP
immediate and accumulator	1010 100w : immediate data	UV
immediate and memory	1111 011w : mod 000 r/m : immediate data	NP
VERR — Verify a Segment for Reading		NP
VERW — Verify a Segment for Writing		NP
WAIT — Wait	1001 1011	NP
WBINVD — Write-Back and Invalidate Data Cache		NP
WRMSR — Write to Model-Specific Register		NP
XADD — Exchange and Add		NP
XCHG — Exchange Register/Memory with Register		NP
XLAT/XLATB — Table Look-up Translation		NP
XOR — Logical Exclusive OR		UV

B

Floating-Point Pairing Tables

APPENDIX B
FLOATING-POINT PAIRING TABLES

In the floating-point table in this appendix, the following abbreviations are used:

FX — Pairs with FXCH

NP — No pairing.

Table B-1. Floating-Point Instruction Pairing

Instruction	Format	Pairing
F2XM1 — Compute $2^{ST(0)} - 1$		NP
FABS — Absolute Value		FX
FADD — Add		FX
FADDP — Add and Pop		FX
FBLD — Load Binary Coded Decimal		NP
FBSTP — Store Binary Coded Decimal and Pop		NP
FCHS — Change Sign		FX
FCLEX — Clear Exceptions		NP
FCOM — Compare Real		FX
FCOMP — Compare Real and Pop		FX
FCOMPP — Compare Real and Pop Twice		
FCOS — Cosine of ST(0)		NP
FDECSTP — Decrement Stack-Top Pointer		NP
FDIV — Divide		FX
FDIVP — Divide and Pop		FX
FDIVR — Reverse Divide		FX
FDIVRP — Reverse Divide and Pop		FX
FFREE — Free ST(i) Register		NP
FIADD — Add Integer		NP
FICOM — Compare Integer		NP
FICOMP — Compare Integer and Pop		NP

Table B-1. Floating-Point Instruction Pairing (Cont'd)

Instruction	Format	Pairing
FIDIV		NP
FIDIVR		NP
FILD — Load Integer		NP
FIMUL		NP
FINCSTP — Increment Stack Pointer		NP
FINIT — Initialize Floating-Point Unit		NP
FIST — Store Integer		NP
FISTP — Store Integer and Pop		NP
FISUB		NP
FISUBR		NP
FLD — Load Real		
32-bit memory	11011 001 : mod 000 r/m	FX
64-bit memory	11011 101 : mod 000 r/m	FX
80-bit memory	11011 011 : mod 101 r/m	NP
ST(i)	11011 001 : 11 000 ST(i)	FX
FLD1 — Load +1.0 into ST(0)		NP
FLDCW — Load Control Word		NP
FLDENV — Load FPU Environment		NP
FLDL2E — Load $\log_2(e)$ into ST(0)		NP
FLDL2T — Load $\log_2(10)$ into ST(0)		NP
FLDLG2 — Load $\log_{10}(2)$ into ST(0)		NP
FLDLN2 — Load $\log_e(2)$ into ST(0)		NP
FLDPI — Load p into ST(0)		NP
FLDZ — Load +0.0 into ST(0)		NP
FMUL — Multiply		FX
FMULP — Multiply		FX
FNOP — No Operation		NP
FPATAN — Partial Arctangent		NP
FPREM — Partial Remainder		NP
FPREM1 — Partial Remainder (IEEE)		NP

Table B-1. Floating-Point Instruction Pairing (Cont'd)

Instruction	Format	Pairing
FPTAN — Partial Tangent		NP
FRNDINT — Round to Integer		
FRSTOR — Restore FPU State		NP
FSAVE — Store FPU State		NP
FSCALE — Scale		NP
FSIN — Sine		NP
FSINCOS — Sine and Cosine		NP
FSQRT — Square Root		NP
FST — Store Real		NP
FSTCW — Store Control Word		NP
FSTENV — Store FPU Environment		NP
FSTP — Store Real and Pop		NP
FSTSW — Store Status Word into AX		NP
FSTSW — Store Status Word into Memory		NP
FSUB — Subtract		FX
FSUBP — Subtract and Pop		FX
FSUBR — Reverse Subtract		FX
FSUBRP — Reverse Subtract and Pop		FX
FTST — Test		FX
FUCOM — Unordered Compare Real)		FX
FUCOMP — Unordered Compare and Pop		FX
FUCOMPP — Unordered Compare and Pop Twice		FX
FXAM — Examine		NP
FXCH — Exchange ST(0) and ST(i)		
FXTRACT — Extract Exponent and Significant		NP
FYL2X — ST(1) ´ $\log_2$(ST(0))		NP
FYL2XP1 — ST(1) ´ $\log_2$(ST(0) + 1.0)		NP
FWAIT — Wait until FPU Ready		

C

Pentium® Pro Processor Instruction to Decoder Specification

APPENDIX C
PENTIUM® PRO PROCESSOR INSTRUCTION TO DECODER SPECIFICATION

Following is the table of macro-instructions and the number of µops decoded from each instruction.

AAA	1	ADD r16/32,imm16/32	1
AAD	3	ADD r16/32,imm8	1
AAM	4	ADD r16/32,m16/32	2
AAS	1	ADD r16/32,rm16/32	1
ADC AL,imm8	2	ADD r8,imm8	1
ADC eAX,imm16/32	2	ADD r8,m8	2
ADC m16/32,imm16/32	4	ADD r8,rm8	1
ADC m16/32,r16/32	4	ADD rm16/32,r16/32	1
ADC m8,imm8	4	ADD rm8,r8	1
ADC m8,r8	4	AND AL,imm8	1
ADC r16/32,imm16/32	2	AND eAX,imm16/32	1
ADC r16/32,m16/32	3	AND m16/32,imm16/32	4
ADC r16/32,rm16/32	2	AND m16/32,r16/32	4
ADC r8,imm8	2	AND m8,imm8	4
ADC r8,m8	3	AND m8,r8	4
ADC r8,rm8	2	AND r16/32,imm16/32	1
ADC rm16/32,r16/32	2	AND r16/32,imm8	1
ADC rm8,r8	2	AND r16/32,m16/32	2
ADD AL,imm8	1	AND r16/32,rm16/32	1
ADD eAX,imm16/32	1	AND r8,imm8	1
ADD m16/32,imm16/32	4	AND r8,m8	2
ADD m16/32,r16/32	4	AND r8,rm8	1
ADD m8,imm8	4	AND rm16/32,r16/32	1
ADD m8,r8	4	AND rm8,r8	1

ARPL m16	complex
ARPL rm16, r16	complex
BOUND r16,m16/32&16/32	complex
BSF r16/32,m16/32	3
BSF r16/32,rm16/32	2
BSR r16/32,m16/32	3
BSR r16/32,rm16/32	2
BSWAP r32	2
BT m16/32, imm8	2
BT m16/32, r16/32	complex
BT rm16/32, imm8	1
BT rm16/32, r16/32	1
BTC m16/32, imm8	4
BTC m16/32, r16/32	complex
BTC rm16/32, imm8	1
BTC rm16/32, r16/32	1
BTR m16/32, imm8	4
BTR m16/32, r16/32	complex
BTR rm16/32, imm8	1
BTR rm16/32, r16/32	1
BTS m16/32, imm8	4
BTS m16/32, r16/32	complex
BTS rm16/32, imm8	1
BTS rm16/32, r16/32	1
CALL m16/32 near	complex
CALL m16	complex
CALL ptr16	complex
CALL r16/32 near	complex
CALL rel16/32 near	4
CBW	1
CLC	1

CLD	4
CLI	complex
CLTS	complex
CMC	1
CMOVB/NAE/C r16/32,m16/32	3
CMOVB/NAE/C r16/32,r16/32	2
CMOVBE/NA r16/32,m16/32	3
CMOVBE/NA r16/32,r16/32	2
CMOVE/Z r16/32,m16/32	3
CMOVE/Z r16/32,r16/32	2
CMOVL/NGE r16/32,m16/32	3
CMOVL/NGE r16/32,r16/32	2
CMOVLE/NG r16/32,m16/32	3
CMOVLE/NG r16/32,r16/32	2
CMOVNB/AE/NC r16/32,m16/32	3
CMOVNB/AE/NC r16/32,r16/32	2
CMOVNBE/A r16/32,m16/32	3
CMOVNBE/A r16/32,r16/32	2
CMOVNE/NZ r16/32,m16/32	3
CMOVNE/NZ r16/32,r16/32	2
CMOVNL/GE r16/32,m16/32	3
CMOVNL/GE r16/32,r16/32	2
CMOVNLE/G r16/32,m16/32	3
CMOVNLE/G r16/32,r16/32	2
CMOVNO r16/32,m16/32	3
CMOVNO r16/32,r16/32	2
CMOVNP/PO r16/32,m16/32	3
CMOVNP/PO r16/32,r16/32	2
CMOVNS r16/32,m16/32	3
CMOVNS r16/32,r16/32	2
CMOVOr16/32,m16/32	3

SPECIFICATION

CMOVOr16/32,r16/32	2	CWDE	1
CMOVP/PE r16/32,m16/32	3	DAA	1
CMOVP/PE r16/32,r16/32	2	DAS	1
CMOVS r16/32,m16/32	3	DECm16/32	4
CMOVS r16/32,r16/32	2	DECm8	4
CMP AL, imm8	1	DECr16/32	1
CMP eAX,imm16/32	1	DECrm16/32	1
CMP m16/32, imm16/32	2	DECrm8	1
CMP m16/32, imm8	2	DIV AL,rm8	3
CMP m16/32,r16/32	2	DIV AX,m16/32	4
CMP m8, imm8	2	DIV AX,m8	4
CMP m8, imm8	2	DIV AX,rm16/32	4
CMP m8,r8	2	ENTER	complex
CMP r16/32,m16/32	2	F2XM1	complex
CMP r16/32,rm16/32	1	FABS	1
CMP r8,m8	2	FADD ST(i),ST	1
CMP r8,rm8	1	FADD ST,ST(i)	1
CMP rm16/32,imm16/32	1	FADD m32real	2
CMP rm16/32,imm8	1	FADD m64real	2
CMP rm16/32,r16/32	1	FADDP ST(i),ST	1
CMP rm8,imm8	1	FBLD m80dec	complex
CMP rm8,imm8	1	FBSTP m80dec	complex
CMP rm8,r8	1	FCHS	3
CMPSB/W/D m8/16/32,m8/16/32	complex	FCMOVB STi	2
CMPXCHG m16/32,r16/32	complex	FCMOVBE STi	2
CMPXCHG m8,r8	complex	FCMOVE STi	2
CMPXCHG rm16/32,r16/32	complex	FCMOVNB STi	2
CMPXCHG rm8,r8	complex	FCMOVNBE STi	2
CMPXCHG8B rm64	complex	FCMOVNE STi	2
CPUID	complex	FCMOVNU STi	2
CWD/CDQ	1	FCMOVU STi	2

FCOM STi	1	FICOM m32int	complex
FCOM m32real	2	FICOMP m16int	complex
FCOM m64real	2	FICOMP m32int	complex
FCOM2 STi	1	FIDIV m16int	complex
FCOMI STi	1	FIDIV m32int	complex
FCOMIP STi	1	FIDIVR m16int	complex
FCOMP STi	1	FIDIVR m32int	complex
FCOMP m32real	2	FILD m16int	4
FCOMP m64real	2	FILD m32int	4
FCOMP3 STi	1	FILD m64int	4
FCOMP5 STi	1	FIMUL m16int	complex
FCOMPP	2	FIMUL m32int	complex
FCOS	complex	FINCSTP	1
FDECSTP	1	FIST m16int	4
FDISI	1	FIST m32int	4
FDIV ST(i),ST	1	FISTP m16int	4
FDIV ST,ST(i)	1	FISTP m32int	4
FDIV m32real	2	FISTP m64int	4
FDIV m64real	2	FISUB m16int	complex
FDIVP ST(i),ST	1	FISUB m32int	complex
FDIVR ST(i),ST	1	FISUBR m16int	complex
FDIVR ST,ST(i)	1	FISUBR m32int	complex
FDIVR m32real	2	FLD STi	1
FDIVR m64real	2	FLD m32real	1
FDIVRP ST(i),ST	1	FLD m64real	1
FENI	1	FLD m80real	4
FFREE ST(i)	1	FLD1	2
FFREEP ST(i)	2	FLDCW m2byte	3
FIADD m16int	complex	FLDENV m14/28byte	complex
FIADD m32int	complex	FLDL2E	2
FICOM m16int	complex	FLDL2T	2

SPECIFICATION

FLDLG2	2	FSTP STi	1
FLDLN2	2	FSTP m32real	2
FLDPI	2	FSTP m64real	2
FLDZ	1	FSTP m80real	complex
FMUL ST(i),ST	1	FSTP1 STi	1
FMUL ST,ST(i)	1	FSTP8 STi	1
FMUL m32real	2	FSTP9 STi	1
FMUL m64real	2	FSUB ST(i),ST	1
FMULP ST(i),ST	1	FSUB ST,ST(i)	1
FNCLEX	3	FSUB m32real	2
FNINIT	complex	FSUB m64real	2
FNOP	1	FSUBP ST(i),ST	1
FNSAVE m94/108byte	complex	FSUBR ST(i),ST	1
FNSTCW m2byte	3	FSUBR ST,ST(i)	1
FNSTENV m14/28byte	complex	FSUBR m32real	2
FNSTSW AX	3	FSUBR m64real	2
FNSTSW m2byte	3	FSUBRP ST(i),ST	1
FPATAN	complex	FTST	1
FPREM	complex	FUCOM STi	1
FPREM1	complex	FUCOMI STi	1
FPTAN	complex	FUCOMIP STi	1
FRNDINT	complex	FUCOMP STi	1
FRSTOR m94/108byte	complex	FUCOMPP	2
FSCALE	complex	FWAIT	2
FSETPM	1	FXAM	1
FSIN	complex	FXCH STi	1
FSINCOS	complex	FXCH4 STi	1
FSQRT	1	FXCH7 STi	1
FST STi	1	FXTRACT	complex
FST m32real	2	FYL2X	complex
FST m64real	2	FYL2XP1	complex

HALT	complex	JB/NAE/C rel8	1
IDIV AL,rm8	3	JBE/NA rel16/32	1
IDIV AX,m16/32	4	JBE/NA rel8	1
IDIV AX,m8	4	JCXZ/JECXZ rel8	2
IDIV eAX,rm16/32	4	JE/Z rel16/32	1
IMUL m16	4	JE/Z rel8	1
IMUL m32	4	JL/NGE rel16/32	1
IMUL m8	2	JL/NGE rel8	1
IMUL r16/32,m16/32	2	JLE/NG rel16/32	1
IMUL r16/32,rm16/32	1	JLE/NG rel8	1
IMUL r16/32,rm16/32,imm8/16/32	2	JMP m16	complex
IMUL r16/32,rm16/32,imm8/16/32	1	JMP near m16/32	2
IMUL rm16	3	JMP near reg16/32	1
IMUL rm32	3	JMP ptr16	complex
IMUL rm8	1	JMP rel16/32	1
IN eAX, DX	complex	JMP rel8	1
IN eAX, imm8	complex	JNB/AE/NC rel16/32	1
INCm16/32	4	JNB/AE/NC rel8	1
INCm8	4	JNBE/A rel16/32	1
INCr16/32	1	JNBE/A rel8	1
INCrm16/32	1	JNE/NZ rel16/32	1
INCrm8	1	JNE/NZ rel8	1
INSB/W/D m8/16/32,DX	complex	JNL/GE rel16/32	1
INT1	complex	JNL/GE rel8	1
INT3	complex	JNLE/G rel16/32	1
INTN	3	JNLE/G rel8	1
INTO	complex	JNO rel16/32	1
INVD	complex	JNO rel8	1
INVLPG m	complex	JNP/PO rel16/32	1
IRET	complex	JNP/PO rel8	1
JB/NAE/C rel16/32	1	JNS rel16/32	1

SPECIFICATION

JNS rel8	1	LOCK AND m16/32,r16/32	complex
JOrel16/32	1	LOCK AND m8,imm8	complex
JOrel8	1	LOCK AND m8,r8	complex
JP/PE rel16/32	1	LOCK BTC m16/32, imm8	complex
JP/PE rel8	1	LOCK BTC m16/32, r16/32	complex
JS rel16/32	1	LOCK BTR m16/32, imm8	complex
JS rel8	1	LOCK BTR m16/32, r16/32	complex
LAHF	1	LOCK BTS m16/32, imm8	complex
LAR m16	complex	LOCK BTS m16/32, r16/32	complex
LAR rm16	complex	LOCK CMPXCHG m16/32,r16/32	complex
LDS r16/32,m16	complex	LOCK CMPXCHG m8,r8	complex
LEA r16/32,m	1	LOCK CMPXCHG8B rm64	complex
LEAVE	3	LOCK DECm16/32	complex
LES r16/32,m16	complex	LOCK DECm8	complex
LFS r16/32,m16	complex	LOCK INCm16/32	complex
LGDT m16&32	complex	LOCK INCm8	complex
LGS r16/32,m16	complex	LOCK NEGm16/32	complex
LIDT m16&32	complex	LOCK NEGm8	complex
LLDT m16	complex	LOCK NOTm16/32	complex
LLDT rm16	complex	LOCK NOTm8	complex
LMSW m16	complex	LOCK ORm16/32,imm16/32	complex
LMSW r16	complex	LOCK ORm16/32,r16/32	complex
LOCK ADC m16/32,imm16/32	complex	LOCK ORm8,imm8	complex
LOCK ADC m16/32,r16/32	complex	LOCK ORm8,r8	complex
LOCK ADC m8,imm8	complex	LOCK SBB m16/32,imm16/32	complex
LOCK ADC m8,r8	complex	LOCK SBB m16/32,r16/32	complex
LOCK ADD m16/32,imm16/32	complex	LOCK SBB m8,imm8	complex
LOCK ADD m16/32,r16/32	complex	LOCK SBB m8,r8	complex
LOCK ADD m8,imm8	complex	LOCK SUB m16/32,imm16/32	complex
LOCK ADD m8,r8	complex	LOCK SUB m16/32,r16/32	complex
LOCK AND m16/32,imm16/32	complex	LOCK SUB m8,imm8	complex

LOCK SUB m8,r8	complex	MOV GS,rm16	4
LOCK XADD m16/32,r16/32	complex	MOV SS,m16	4
LOCK XADD m8,r8	complex	MOV SS,rm16	4
LOCK XCHG m16/32,r16/32	complex	MOV eAX,moffs16/32	1
LOCK XCHG m8,r8	complex	MOV m16,CS	3
LOCK XOR m16/32,imm16/32	complex	MOV m16,DS	3
LOCK XOR m16/32,r16/32	complex	MOV m16,ES	3
LOCK XOR m8,imm8	complex	MOV m16,FS	3
LOCK XOR m8,r8	complex	MOV m16,GS	3
LODSB/W/D m8/16/32,m8/16/32	2	MOV m16,SS	3
LOOP rel8	4	MOV m16/32,imm16/32	2
LOOPE rel8	4	MOV m16/32,r16/32	2
LOOPNE rel8	4	MOV m8,imm8	2
LSL m16	complex	MOV m8,r8	2
LSL rm16	complex	MOV moffs16/32,eAX	2
LSS r16/32,m16	complex	MOV moffs8,AL	2
LTR m16	complex	MOV r16/32,imm16/32	1
LTR rm16	complex	MOV r16/32,m16/32	1
MOV AL,moffs8	1	MOV r16/32,rm16/32	1
MOV CR0, r32	complex	MOV r32, CR0	complex
MOV CR2, r32	complex	MOV r32, CR2	complex
MOV CR3, r32	complex	MOV r32, CR3	complex
MOV CR4, r32	complex	MOV r32, CR4	complex
MOV DRx, r32	complex	MOV r32, DRx	complex
MOV DS,m16	4	MOV r8,imm8	1
MOV DS,rm16	4	MOV r8,m8	1
MOV ES,m16	4	MOV r8,rm8	1
MOV ES,rm16	4	MOV rm16,CS	1
MOV FS,m16	4	MOV rm16,DS	1
MOV FS,rm16	4	MOV rm16,ES	1
MOV GS,m16	4	MOV rm16,FS	1

SPECIFICATION

MOV rm16,GS	1	NOTm8	4
MOV rm16,SS	1	NOTrm16/32	1
MOV rm16/32,imm16/32	1	NOTrm8	1
MOV rm16/32,r16/32	1	ORAL,imm8	1
MOV rm8,imm8	1	OReAX,imm16/32	1
MOV rm8,r8	1	ORm16/32,imm16/32	4
MOVSB/W/D m8/16/32,m8/16/32	complex	ORm16/32,r16/32	4
MOVSX r16,m8	1	ORm8,imm8	4
MOVSX r16,rm8	1	ORm8,r8	4
MOVSX r16/32,m16	1	ORr16/32,imm16/32	1
MOVSX r32,m8	1	ORr16/32,imm8	1
MOVSX r32,rm16	1	ORr16/32,m16/32	2
MOVSX r32,rm8	1	ORr16/32,rm16/32	1
MOVZX r16,m8	1	ORr8,imm8	1
MOVZX r16,rm8	1	ORr8,m8	2
MOVZX r32,m16	1	ORr8,rm8	1
MOVZX r32,m8	1	ORrm16/32,r16/32	1
MOVZX r32,rm16	1	ORrm8,r8	1
MOVZX r32,rm8	1	OUT DX, eAX	complex
MUL AL,m8	2	OUT imm8, eAX	complex
MUL AL,rm8	1	OUTSB/W/D DX,m8/16/32	complex
MUL AX,m16	4	POP DS	complex
MUL AX,rm16	3	POP ES	complex
MUL EAX,m32	4	POP FS	complex
MUL EAX,rm32	3	POP GS	complex
NEGm16/32	4	POP SS	complex
NEGm8	4	POP eSP	3
NEGrm16/32	1	POP m16/32	complex
NEGrm8	1	POP r16/32	2
NOP	1	POP r16/32	2
NOTm16/32	4	POPA/POPAD	complex

POPF	complex
POPFD	complex
PUSH CS	4
PUSH DS	4
PUSH ES	4
PUSH FS	4
PUSH GS	4
PUSH SS	4
PUSH imm16/32	3
PUSH imm8	3
PUSH m16/32	4
PUSH r16/32	3
PUSH r16/32	3
PUSHA/PUSHAD	complex
PUSHF/PUSHFD	complex
RCL m16/32,1	4
RCL m16/32,CL	complex
RCL m16/32,imm8	complex
RCL m8,1	4
RCL m8,CL	complex
RCL m8,imm8	complex
RCL rm16/32,1	2
RCL rm16/32,CL	complex
RCL rm16/32,imm8	complex
RCL rm8,1	2
RCL rm8,CL	complex
RCL rm8,imm8	complex
RCR m16/32,1	4
RCR m16/32,CL	complex
RCR m16/32,imm8	complex
RCR m8,1	4

RCR m8,CL	complex
RCR m8,imm8	complex
RCR rm16/32,1	2
RCR rm16/32,CL	complex
RCR rm16/32,imm8	complex
RCR rm8,1	2
RCR rm8,CL	complex
RCR rm8,imm8	complex
RDMSR	complex
RDPMC	complex
RDTSC	complex
REP CMPSB/W/D m8/16/32,m8/16/32	complex
REP INSB/W/D m8/16/32,DX	complex
REP LODSB/W/D m8/16/32,m8/16/32	complex
REP MOVSB/W/D m8/16/32,m8/16/32	complex
REP OUTSB/W/D DX,m8/16/32	complex
REP SCASB/W/D m8/16/32,m8/16/32	complex
REP STOSB/W/D m8/16/32,m8/16/32	complex
RET	4
RET	complex
RET near	4
RET near iw	complex
ROL m16/32,1	4
ROL m16/32,CL	4
ROL m16/32,imm8	4
ROL m8,1	4
ROL m8,CL	4
ROL m8,imm8	4
ROL rm16/32,1	1
ROL rm16/32,CL	1
ROL rm16/32,imm8	1

SPECIFICATION

ROL rm8,1	1	SBB m16/32,imm16/32	4
ROL rm8,CL	1	SBB m16/32,r16/32	4
ROL rm8,imm8	1	SBB m8,imm8	4
ROR m16/32,1	4	SBB m8,r8	4
ROR m16/32,CL	4	SBB r16/32,imm16/32	2
ROR m16/32,imm8	4	SBB r16/32,m16/32	3
ROR m8,1	4	SBB r16/32,rm16/32	2
ROR m8,CL	4	SBB r8,imm8	2
ROR m8,imm8	4	SBB r8,m8	3
ROR rm16/32,1	1	SBB r8,rm8	2
ROR rm16/32,CL	1	SBB rm16/32,r16/32	2
ROR rm16/32,imm8	1	SBB rm8,r8	2
ROR rm8,1	1	SCASB/W/D m8/16/32,m8/16/32	3
ROR rm8,CL	1	SETB/NAE/C m8	3
ROR rm8,imm8	1	SETB/NAE/C rm8	1
RSM	complex	SETBE/NA m8	3
SAHF	1	SETBE/NA rm8	1
SAR m16/32,1	4	SETE/Z m8	3
SAR m16/32,CL	4	SETE/Z rm8	1
SAR m16/32,imm8	4	SETL/NGE m8	3
SAR m8,1	4	SETL/NGE rm8	1
SAR m8,CL	4	SETLE/NG m8	3
SAR m8,imm8	4	SETLE/NG rm8	1
SAR rm16/32,1	1	SETNB/AE/NC m8	3
SAR rm16/32,CL	1	SETNB/AE/NC rm8	1
SAR rm16/32,imm8	1	SETNBE/A m8	3
SAR rm8,1	1	SETNBE/A rm8	1
SAR rm8,CL	1	SETNE/NZ m8	3
SAR rm8,imm8	1	SETNE/NZ rm8	1
SBB AL,imm8	2	SETNL/GE m8	3
SBB eAX,imm16/32	2	SETNL/GE rm8	1

SETNLE/G m8	3	SHL/SAL rm16/32,imm8	1
SETNLE/G rm8	1	SHL/SAL rm16/32,imm8	1
SETNO m8	3	SHL/SAL rm8,1	1
SETNO rm8	1	SHL/SAL rm8,1	1
SETNP/PO m8	3	SHL/SAL rm8,CL	1
SETNP/PO rm8	1	SHL/SAL rm8,CL	1
SETNS m8	3	SHL/SAL rm8,imm8	1
SETNS rm8	1	SHL/SAL rm8,imm8	1
SETOm8	3	SHLD m16/32,r16/32,CL	4
SETOrm8	1	SHLD m16/32,r16/32,imm8	4
SETP/PE m8	3	SHLD rm16/32,r16/32,CL	2
SETP/PE rm8	1	SHLD rm16/32,r16/32,imm8	2
SETS m8	3	SHR m16/32,1	4
SETS rm8	1	SHR m16/32,CL	4
SGDT m16&32	4	SHR m16/32,imm8	4
SHL/SAL m16/32,1	4	SHR m8,1	4
SHL/SAL m16/32,1	4	SHR m8,CL	4
SHL/SAL m16/32,CL	4	SHR m8,imm8	4
SHL/SAL m16/32,CL	4	SHR rm16/32,1	1
SHL/SAL m16/32,imm8	4	SHR rm16/32,CL	1
SHL/SAL m16/32,imm8	4	SHR rm16/32,imm8	1
SHL/SAL m8,1	4	SHR rm8,1	1
SHL/SAL m8,1	4	SHR rm8,CL	1
SHL/SAL m8,CL	4	SHR rm8,imm8	1
SHL/SAL m8,CL	4	SHRD m16/32,r16/32,CL	4
SHL/SAL m8,imm8	4	SHRD m16/32,r16/32,imm8	4
SHL/SAL m8,imm8	4	SHRD rm16/32,r16/32,CL	2
SHL/SAL rm16/32,1	1	SHRD rm16/32,r16/32,imm8	2
SHL/SAL rm16/32,1	1	SIDT m16&32	complex
SHL/SAL rm16/32,CL	1	SLDT m16	complex
SHL/SAL rm16/32,CL	1	SLDT rm16	4

SPECIFICATION

SMSW m16	complex	TEST rm16/32,imm16/32	1
SMSW rm16	4	TEST rm16/32,r16/32	1
STC	1	TEST rm8,imm8	1
STD	4	TEST rm8,r8	1
STI	complex	VERR m16	complex
STOSB/W/D m8/16/32,m8/16/32	3	VERR rm16	complex
STR m16	complex	VERW m16	complex
STR rm16	4	VERW rm16	complex
SUB AL,imm8	1	WBINVD	complex
SUB eAX,imm16/32	1	WRMSR	complex
SUB m16/32,imm16/32	4	XADD m16/32,r16/32	complex
SUB m16/32,r16/32	4	XADD m8,r8	complex
SUB m8,imm8	4	XADD rm16/32,r16/32	4
SUB m8,r8	4	XADD rm8,r8	4
SUB r16/32,imm16/32	1	XCHG eAX,r16/32	3
SUB r16/32,imm8	1	XCHG m16/32,r16/32	complex
SUB r16/32,m16/32	2	XCHG m8,r8	complex
SUB r16/32,rm16/32	1	XCHG rm16/32,r16/32	3
SUB r8,imm8	1	XCHG rm8,r8	3
SUB r8,m8	2	XLAT/B	2
SUB r8,rm8	1	XOR AL,imm8	1
SUB rm16/32,r16/32	1	XOR eAX,imm16/32	1
SUB rm8,r8	1	XOR m16/32,imm16/32	4
TEST AL,imm8	1	XOR m16/32,r16/32	4
TEST eAX,imm16/32	1	XOR m8,imm8	4
TEST m16/32,imm16/32	2	XOR m8,r8	4
TEST m16/32,imm16/32	2	XOR r16/32,imm16/32	1
TEST m16/32,r16/32	2	XOR r16/32,imm8	1
TEST m8,imm8	2	XOR r16/32,m16/32	2
TEST m8,imm8	2	XOR r16/32,rm16/32	1
TEST m8,r8	2	XOR r8,imm8	1

XOR r8,m8	2	XOR rm16/32,r16/32	1
XOR r8,rm8	1	XOR rm8,r8	1

D

Pentium® Pro Processor MMX™ Instructions to Decoder Specification

APPENDIX D
PENTIUM® PRO PROCESSOR MMX™ INSTRUCTIONS TO DECODER SPECIFICATION

EMMS	complex
MOVD m32,mm	2
MOVD mm,ireg	1
MOVD mm,m32	1
MOVQ mm,m64	1
MOVQ mm,mm	1
MOVQ m64,mm	2
MOVQ mm,mm	1
PACKSSDW mm,m64	2
PACKSSDW mm,mm	1
PACKSSWB mm,m64	2
PACKSSWB mm,mm	1
PACKUSWB mm,m64	2
PACKUSWB mm,mm	1
PADDB mm,m64	2
PADDB mm,mm	1
PADDD mm,m64	2
PADDD mm,mm	1
PADDSB mm,m64	2
PADDSB mm,mm	1
PADDSW mm,m64	2
PADDSW mm,mm	1
PADDUSB mm,m64	2
PADDUSB mm,mm	1
PADDUSW mm,m64	2
PADDUSW mm,mm	1
PADDW mm,m64	2

PADDW mm,mm	1
PAND mm,m64	2
PAND mm,mm	1
PANDN mm,m64	2
PANDN mm,mm	1
PCMPEQB mm,m64	2
PCMPEQB mm,mm	1
PCMPEQD mm,m64	2
PCMPEQD mm,mm	1
PCMPEQW mm,m64	2
PCMPEQW mm,mm	1
PCMPGTB mm,m64	2
PCMPGTB mm,mm	1
PCMPGTD mm,m64	2
PCMPGTD mm,mm	1
PCMPGTW mm,m64	2
PCMPGTW mm,mm	1
PMADDWD mm,m64	2
PMADDWD mm,mm	1
PMULHW mm,m64	2
PMULHW mm,mm	1
PMULLW mm,m64	2
PMULLW mm,mm	1
POR mm,m64	2
POR mm,mm	1
PSLLD mm,m64	2
PSLLD mm,mm	1

PSLLimmD mm,imm8	1
PSLLimmQ mm,imm8	1
PSLLimmW mm,imm8	1
PSLLQ mm,m64	2
PSLLQ mm,mm	1
PSLLW mm,m64	2
PSLLW mm,mm	1
PSRAD mm,m64	2
PSRAD mm,mm	1
PSRAimmD mm,imm8	1
PSRAimmW mm,imm8	1
PSRAW mm,m64	2
PSRAW mm,mm	1
PSRLD mm,m64	2
PSRLD mm,mm	1
PSRLimmD mm,imm8	1
PSRLimmQ mm,imm8	1
PSRLimmW mm,imm8	1
PSRLQ mm,m64	2
PSRLQ mm,mm	1
PSRLW mm,m64	2
PSRLW mm,mm	1
PSUBB mm,m64	2
PSUBB mm,mm	1
PSUBD mm,m64	2
PSUBD mm,mm	1

PSUBSB mm,m64	2
PSUBSB mm,mm	1
PSUBSW mm,m64	2
PSUBSW mm,mm	1
PSUBUSB mm,m64	2
PSUBUSB mm,mm	1
PSUBUSW mm,m64	2
PSUBUSW mm,mm	1
PSUBW mm,m64	2
PSUBW mm,mm	1
PUNPCKHBW mm,m64	2
PUNPCKHBW mm,mm	1
PUNPCKHDQ mm,m64	2
PUNPCKHDQ mm,mm	1
PUNPCKHWD mm,m64	2
PUNPCKHWD mm,mm	1
PUNPCKLBW mm,m32	2
PUNPCKLBW mm,mm	1
PUNPCKLDQ mm,m32	2
PUNPCKLDQ mm,mm	1
PUNPCKLWD mm,m32	2
PUNPCKLWD mm,mm	1
PXOR mm,m64	2
PXOR mm,mm	1