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.

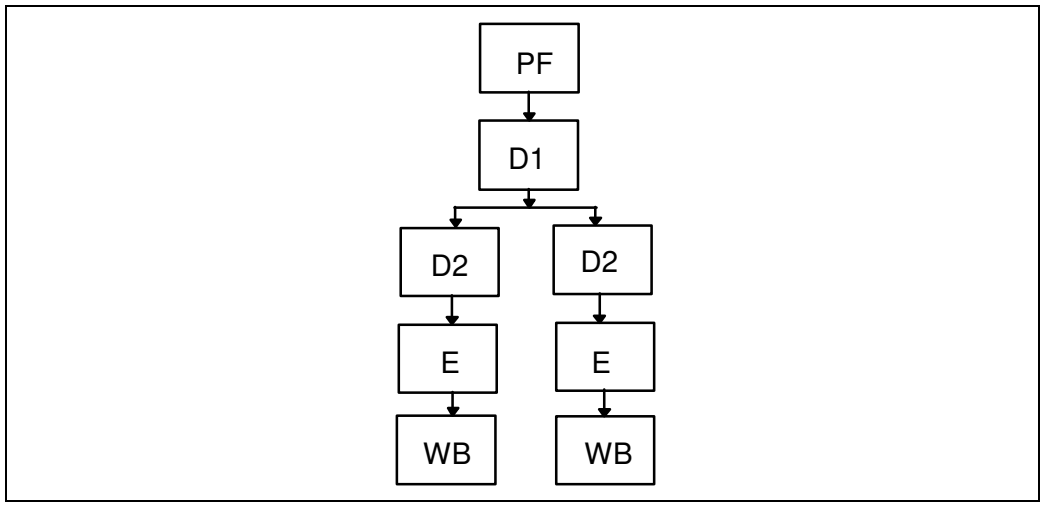


Figure 2-1. Pentium® Processor Integer Pipelines

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.

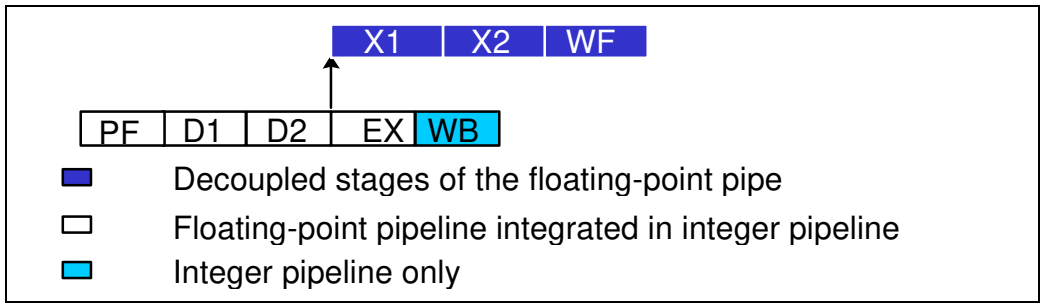


Figure 2-2. Integration of Integer and Floating-Point Pipeline

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.

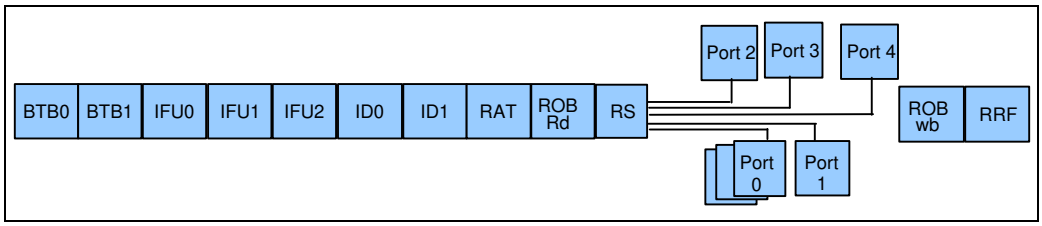


Figure 2-3. Pentium® Pro Processor Pipeline

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

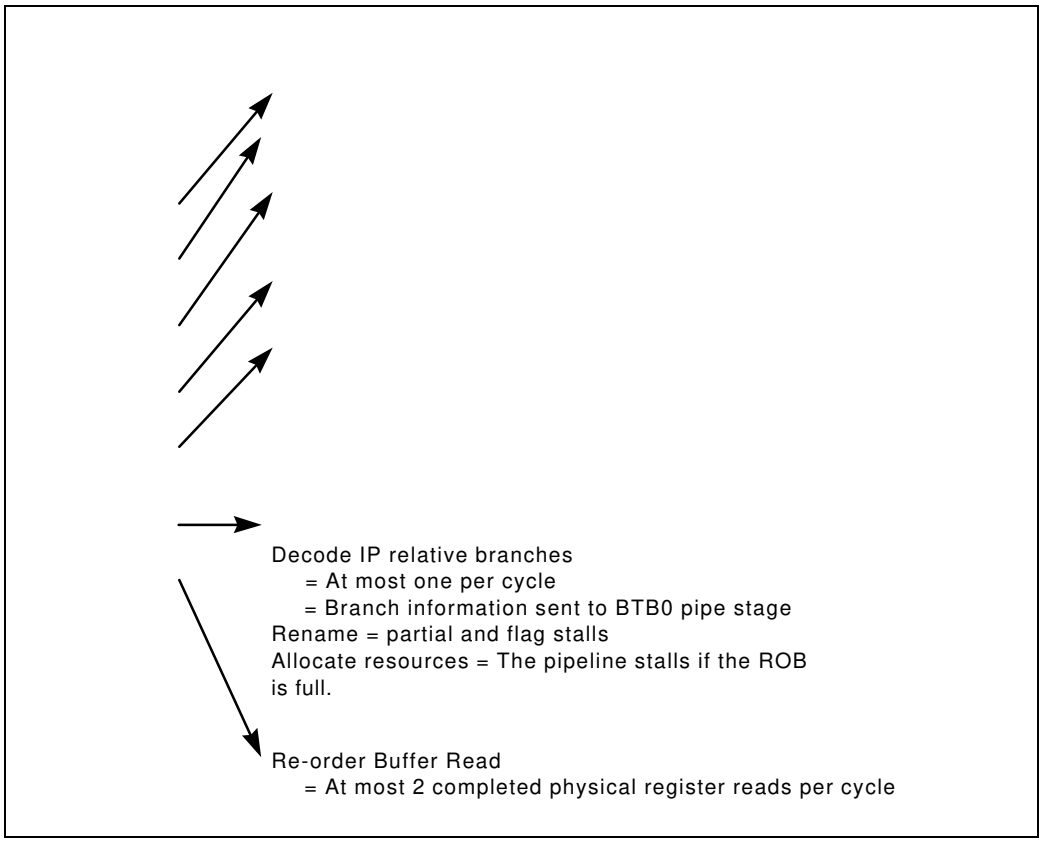


Figure 2-4. In-Order Issue Front-End

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.

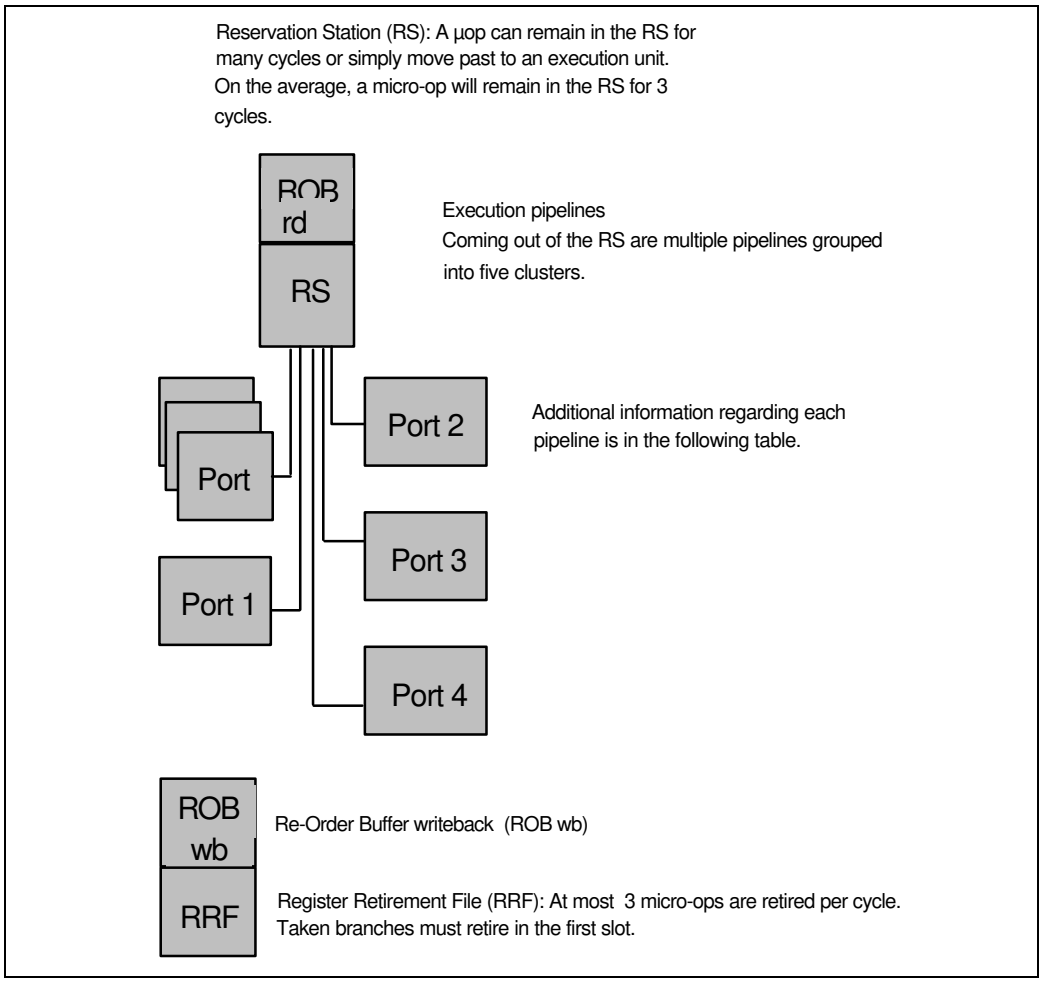


Figure 2-5. Out-Of-Order Core and Retirement Pipeline

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.

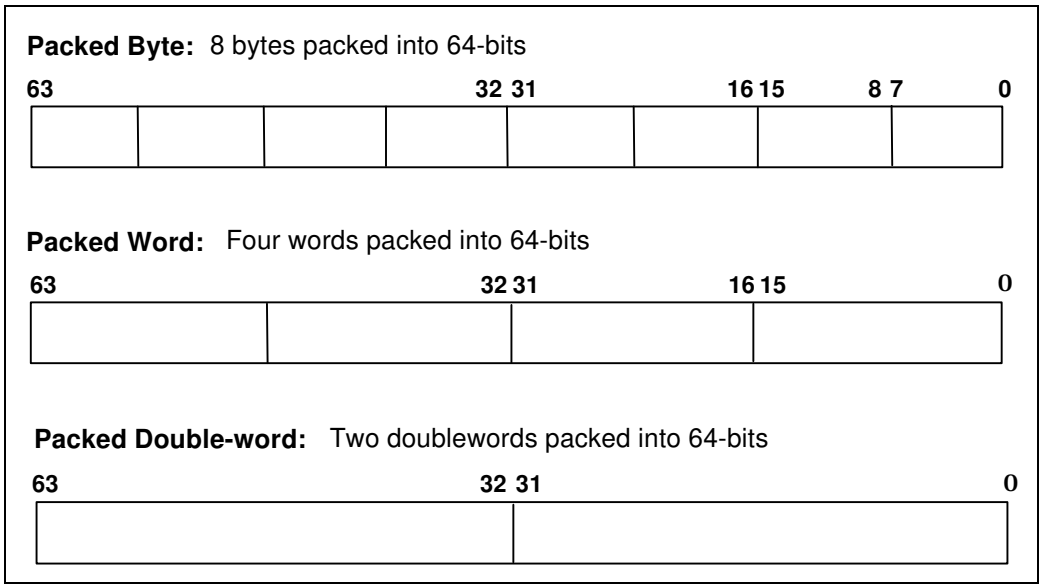


Figure 2-6. New Data Types

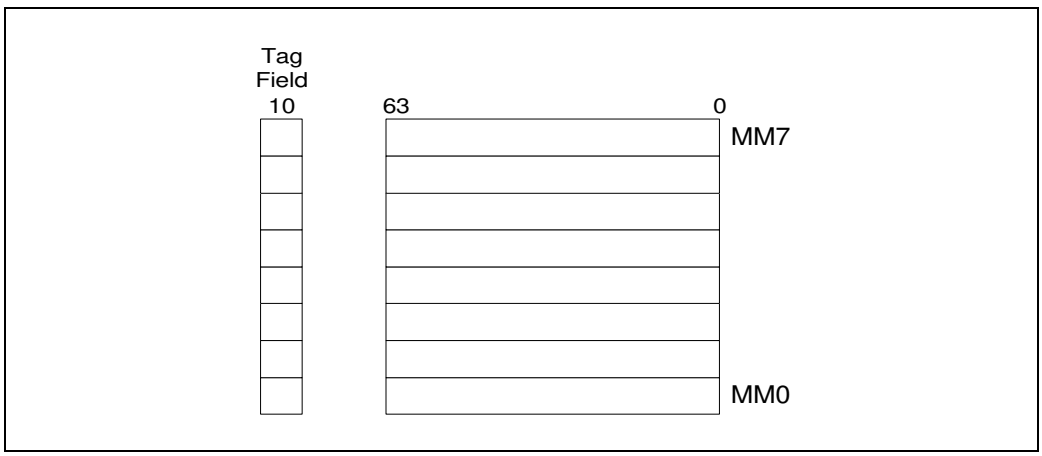


Figure 2-7. MMX™ Register Set

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.

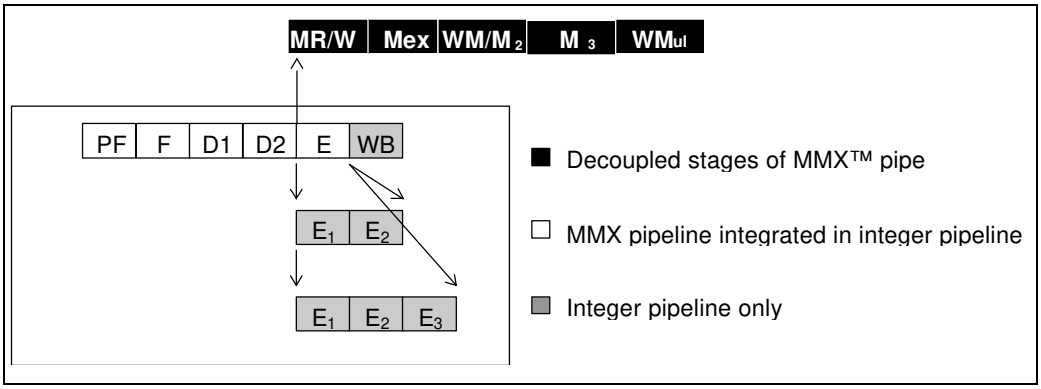


Figure 2-8. MMX™ Pipeline Structure

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.

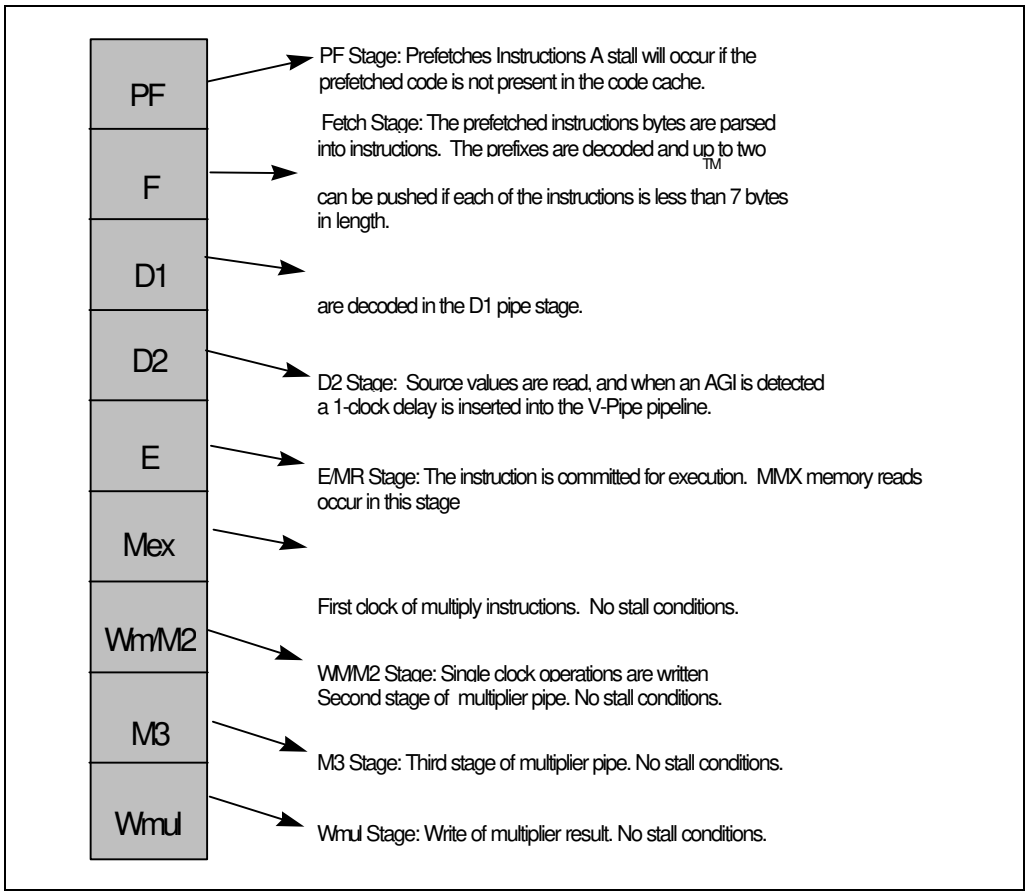


Figure 2-9. MMX™ Instruction Flow in the Pentium® Processor with MMX Technology

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 LEA instructions Shift instructions	Latency 1, Throughput 1/cycle Latency 1, Throughput 1/cycle Latency 1, Throughput 1/cycle
	Integer Multiplication instruction	Latency 4, Throughput 1/cycle[2]
	Floating-Point Unit FADD instruction FMUL FDIV Unit	 Latency 3, Throughput 1/cycle Latency 5, Throughput 1/2 cycle[1,2] 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/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:

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.

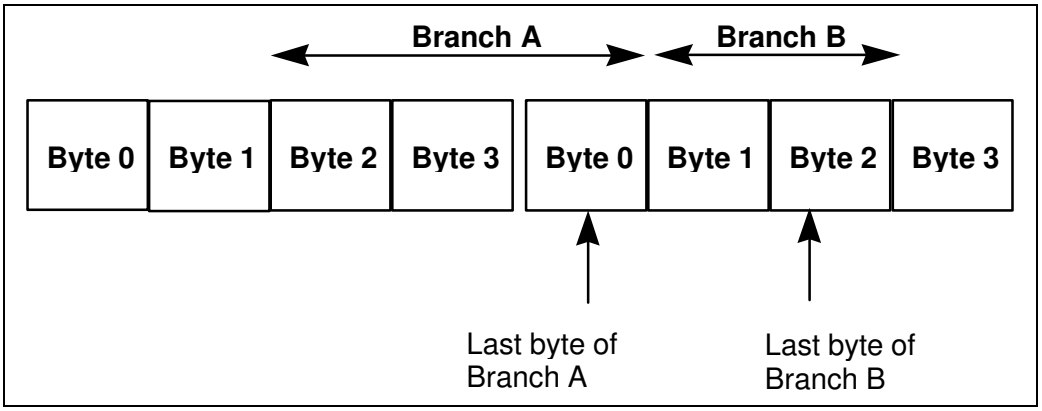


Figure 2-10. Consecutive Branch Example

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.

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:

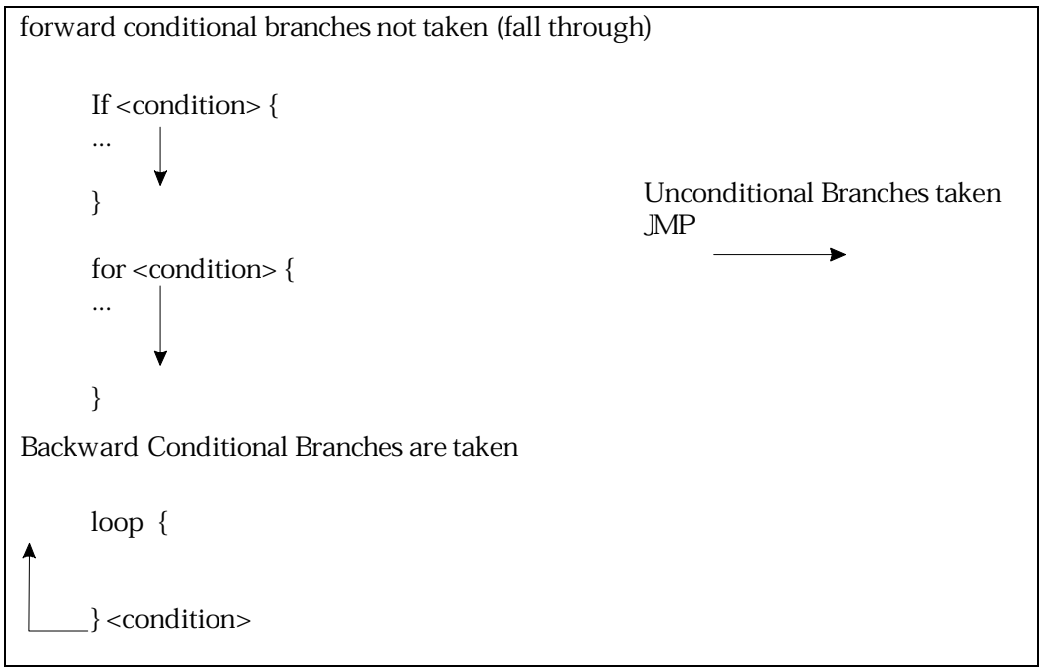


Figure 3-1. Pentium® Pro and Pentium II Processor's Static Branch Prediction Algorithm

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
```

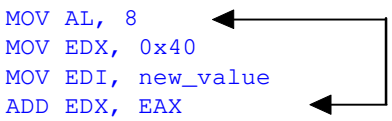
This applies to all of the 8- and 16-bit/32-bit register pairs, listed below:

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
```

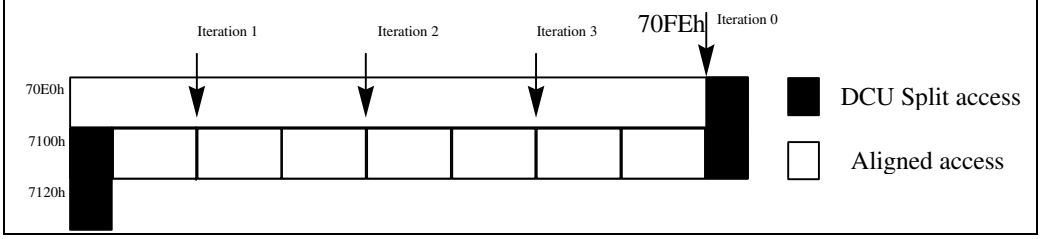


Figure 3-2. DCU Split in the Data Cache

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:

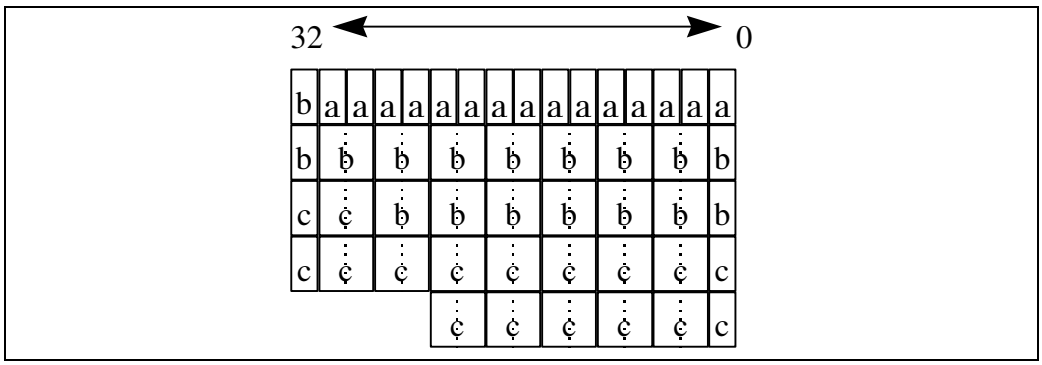


Figure 3-3. Cache Layout of Structures a, b and c

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]
}
```

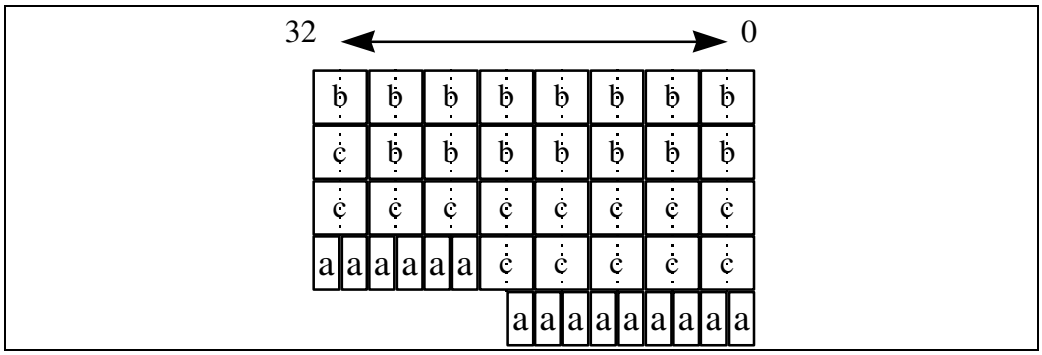


Figure 3-4. Optimized Data Layout of Structures a, b and c

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:

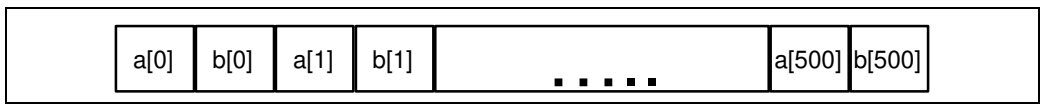


Figure 3-5. Compound Array as Stored in Memory

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.

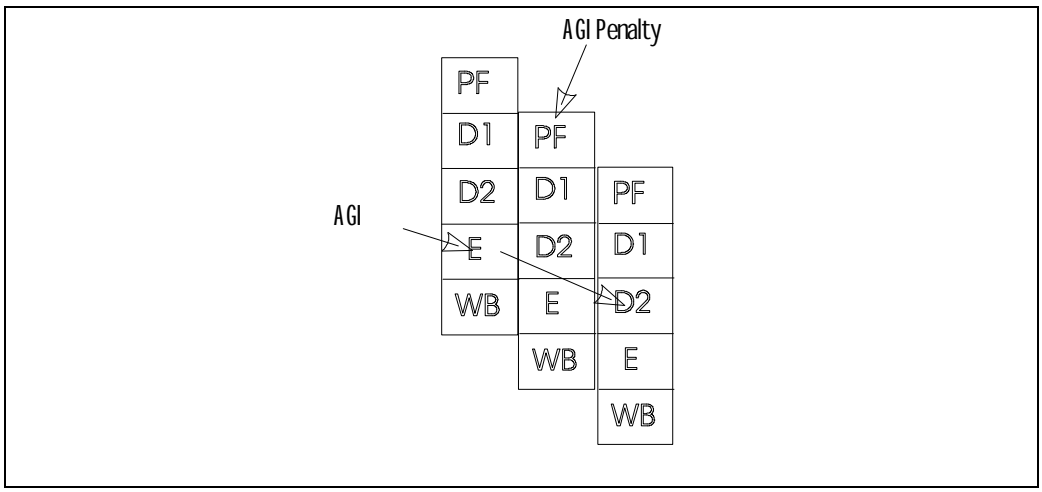


Figure 3-6. Pipeline Example of AGI Stall

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.