intel®

AP-500

APPLICATION NOTE

Optimizations for Intel's 32-Bit Processors

GARY CARLETON
TECHNICAL MARKETING,
INTEL ARCHITECTURE LABS

November 1993

Order Number: 241799-001

Optimizations for Intel's 32-Bit Processors

1.0 INTRODUCTION

The Intel386 CPU Architecture Family represents a series of compatible processors including the Intel386, Intel486, and the Pentium processors. The newer members of the family are capable of executing any binaries created for members of previous generations. For example, any existing 8086/8088, 80286, Intel386 CPU (DX or SX), and Intel486 CPU applications will be able to execute on the Pentium processor without any modification or recompilation. However, there are certain code optimization techniques which will make applications execute faster on a specific member of the family with little or no impact on the performance of other members. Most of these optimizations deal with instruction sequence selection and instruction reordering to complement the processor micro architecture.

The intent of this document is to describe the implementation differences of the processor members and the optimization strategy that gives the best performance for all members of the family.

2.0 OVERVIEW OF Intel386™, Intel486™, AND Pentium™ PROCESSORS

2.1 The Intel386™ Processor

The Intel386 processor is the first implementation of the 32-bit Intel386 architecture. It includes full 32-bit data paths, rich 32-bit addressing modes and on-chip memory management.

2.1.1 INSTRUCTION PREFETCHER

The instruction prefetcher prefetches the instruction stream from external memory and the prefetched instructions are kept in its four-deep, four-byte-wide prefetch buffers. The instruction decoder operates on the code stream fed to it through the prefetch buffers.

2.1.2 INSTRUCTION DECODER

The instruction decoder places the decoded information in a three-deep FIFO. It is decoupled from both the prefetcher and the execution core and has separate protocols with each.

2.1.3 EXECUTION CORE

The core engine executes the incoming instructions one at a time, but for certain cases, it allows overlapping the last execution cycle of the current instruction with the effective address calculation of the next instruction's memory reference.

From the compiler writer's point of view, the decoupled prefetch/decode/execution stages and the sequential nature of the core engine has placed few requirements for instruction scheduling. Avoiding the use of an index in the effective address will save one extra clock. A careful choice of instructions to minimize the execution clock counts is the best optimization approach.

2.2 The Intel486™ Processor

The Intel486 processor has a full blown integer pipeline delivering a peak throughput of one instruction per clock. It has integrated the first level cache and the floating-point unit on chip, and it has the same on-chip memory management capabilities as the Intel386 processor.

2.2.1 INTEGER PIPELINE

The Intel486 CPU has a five-stage integer pipeline capable of processing one instruction per clock. The five pipeline stages are:

1. Prefetch (PF)—where instructions are fetched from the cache and placed in one of two 16-byte buffers.
2. Decode (D1)—where incoming code stream is being decoded. Prefixed instructions stay in D1 for two clocks.
3. Address Generation (D2)—where effective address and linear address are calculated in parallel. The address generation can usually be completed in one cycle except in cases where the indexed addressing mode is used. If the index is used, the instruction stays in D2 for two clocks.
4. Execution (E)—where the machine operations are performed. Simple instructions (those with one machine operation) take one cycle to execute giving a maximum throughput of one instruction per clock. The more complex instructions take multiple execution cycles.
5. Writeback (WB)—where the needed register update occurs.

A taken branch breaks the pipeline stream and causes a two clock penalty whereas the pipeline stream is unaffected by a not-taken branch.

2.2.2 ON-CHIP CACHE

The on-chip cache is a combined instruction and data cache. It is 8 Kbytes in size, four-way set associative with a 16-byte line size and pseudo-LRU replacement algorithm. All data references have priority access to the cache over instruction prefetch cycles.

2.2.3 ON-CHIP FLOATING-POINT UNIT

The on-chip floating-point unit utilizes the integer pipeline for early data access. The bus structure allows 64-bit data to be transferred between the cache and the floating-point hardware in one clock. The floating-point design also allows overlapping the floating-point operations with integer operations.

The execution core of the Intel486 processor has been engineered to maximize the throughput of a class of "frequently used" instructions. Hence, careful selection of an instruction sequence to perform a given task results in faster execution time. Also, code scheduling to avoid pipeline stalls helps to boost application performance. Most of the optimizations targeted to the Intel486 processor do not have negative effects on the Intel386 processor.

2.3 The Pentium™ Processor

The Pentium processor is an advanced *superscalar* 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.

2.3.1 INTEGER PIPELINES

The Pentium processor has two parallel integer pipelines, the main pipe (U) which is an enhanced Intel486 processor pipe and the secondary pipe (V) which 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 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.)

When instructions execute in the two pipes, their behavior is exactly the same as if they were executed sequentially. When a stall occurs successive instructions are not allowed to pass the stalled instruction in either pipe. In the Pentium processor's pipelines, the D2 stage can perform a multiway add, so there is not a one clock index penalty as with the Intel486 CPU pipeline.

2.3.2 CACHES

The on-chip cache subsystem consists of two (instruction and data) 8-Kbyte two-way set associative caches 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 3 clocks.

2.3.3 INSTRUCTION PREFETCHER

The instruction prefetcher has four buffers, each of which is 32 bytes long. It can fetch an instruction which is split among two cache lines with no penalty. Because the instruction and data caches are separate, instruction prefetches no longer conflict with data references for access to the cache (as in the case of the Intel486 processor).

2.3.4 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. There is a 3 cycle penalty if the conditional branch was executed in the U pipe or a 4 cycle penalty if it was executed in the V pipe. Mispredicted calls and unconditional jump instructions have a 3 clock penalty in either pipe. On the Intel486 processor, taken branches have a two clock penalty.

2.3.5 PIPELINED FLOATING-POINT UNIT

The majority of the frequently used instructions are pipelined so that the pipelines can accept a new pair of operands every cycle. Therefore a good code generator can achieve a throughput of almost 1 instruction per cycle (of course this assumes a program with a modest amount of natural parallelism!). The fxch instruction can be executed in parallel with the commonly used FP instructions, which lets the code generator or programmer treat the floating-point stack as a regular register set without any performance degradation.

With the superscalar implementation, it is important to schedule the instruction stream to maximize the usage of the two integer pipelines. Since each of the Pentium processor's integer pipelines is enhanced from the pipeline of the Intel486 processor, the instruction scheduling criteria for the Pentium processor is a superset of the Intel486 processor requirements.

3.0 INTEGER EXAMPLES

With the overview of the Intel386, Intel486 and Pentium processors in the previous section, the examples given in this section further illustrate the execution clock cycles among various instruction sequences.

All examples assume a 100% cache hit rate and non-conflicting memory accesses. A 32-bit flat address model is also assumed.

There is a cycle count next to each instruction. A cycle count appears without an instruction when there is a pipe stall. These examples also assume that the branch prediction is correct.

Going through these examples should give you an intuitive feel for how pairing works on the Pentium processor and additional insight about some of the common delays on both the Intel486 and Pentium processors.

C source:

```
static int a[10], b[10];
int i;
for (i=0; i<10; i++) {
    a[i] = a[i] + 1;
    b[i] = b[i] + 1;
}
```

There are various instruction sequences which will produce a correct program. Their individual performance, however, may vary considerably.

Here are three examples:

```
  Sequence 1               Sequence 2                  Sequence 3

  xor  eax, eax            xor  eax, eax               mov  eax, -40

TopOfLoop:               TopOfLoop:                  TopOfLoop:
  mov  edx, eax            inc  dword ptr [eax*4+a]    mov  edx, [eax+40+a]
  shl  edx, 2              inc  dword ptr [eax*4+b]    mov  ecx, [eax+40+b]
  inc  dword ptr [edx+a]   inc  eax                    inc  edx
  mov  edx, eax            cmp  eax, 10                inc  ecx
  shl  edx, 2              jl   TopOfLoop              mov  [eax+40+a], edx
  inc  dword ptr [edx+b]                               mov  [eax+40+b], ecx
  inc  eax                                             add  eax, 4
  cmp  eax, 10                                         jnz  TopOfLoop
  jl   TopOfLoop
```

Code Sequence 1 could benefit from common subexpression elimination. It is not unoptimized code, it is just not thoroughly optimized. Code that is unoptimized would not keep "i" in a register.

Code Sequence 2 is the most straightforward style code.

Code Sequence 3 uses a load/store model. It also incorporates some induction variable elimination optimizations with test replacement. The loop counter in eax counts up to zero. When it becomes zero, the jnz is not taken. This code avoids the compare instruction.

The performance of each of these code sequences is examined on both the Intel486 and Pentium processors.

3.1 Code Sequence 1, Intel486™ Processor

The `shl` instruction takes two cycles on an Intel486 processor. ALU operations (e.g. `add`) with memory results take 3 clocks: 1 to load, 1 to add, and 1 to store.

```
mov  edx, eax            ; 1
shl  edx, 2              ; 2   2 clock instruction
  (shl)                  ; 3
inc  dword ptr [edx+a]   ; 4   3 clocks with memory operand
  (inc)                  ; 5     plus 1 clock for edx AGI
  (inc)                  ; 6
  (inc)                  ; 7
mov  edx, eax            ; 8
shl  edx, 2              ; 9   2 clock instruction
  (shl)                  ; 10
inc  dword ptr [edx+b]   ; 11  3 clocks with memory operand
  (inc)                  ; 12    plus 1 clock for edx AGI
  (inc)                  ; 13
  (inc)                  ; 14
inc  eax                 ; 15
cmp  eax, 10             ; 16
jl   TopOfLoop           ; 17  2 clocks because jl is prefixed
                         ; 18
                         ; 19  branch taken penalty
                         ; 20
mov  edx, eax            ; 21  next iteration
```

Total: 20 cycles

Cycles 4 and 11 had an Address Generation Interlock (AGI) Delay. Register edx was written in cycle 3 and used as a base register in cycle 5. When a register is used in an effective address calculation in the cycle after the register is written, there is a one clock penalty. This happens because the effective address calculation is performed in the D2 stage of the pipeline.

With the assembler used for this code sequence, the `jl` instruction was a “jump near,” not a “jump short.” “Jump near” is a `0f` prefixed instruction. Prefixed instructions take an extra cycle on the Intel486 processor in the D1 stage.

The Intel486 processor does not have any branch prediction mechanism. Whenever jumps are taken, there is a 2 clock penalty (cycles 19 and 20).

3.2 Code Sequence 1, Pentium™ Processor

```
U pipe              V pipe
mov  edx, eax                           ; 1
shl  edx, 2                             ; 2
inc  dword ptr [edx+a]                  ; 3   3 clocks with mem. op
  (inc)                                 ; 4     plus 1 for edx AGI
  (inc)                                 ; 5
  (inc)             mov  edx, eax       ; 6   Pairs with last U cycle
shl  edx, 2                             ; 7
inc  dword ptr [edx+b]                  ; 8   3 clocks with mem. op
  (inc)                                 ; 9     plus 1 for edx AGI
  (inc)                                 ; 10
  (inc)             inc  eax            ; 11  Pairs with last U cycle
cmp  eax, 10        jl   TopOfLoop      ; 12
mov  edx, eax                           ; 13  Next iteration
```

Total: 12 cycles

Note that the "shift" instruction takes two clocks on the Intel486 processor and only one on the Pentium processor. The Pentium processor has special hardware to avoid the `0f` prefix delay on `jcc` "near" instructions. It also can pair the compare and jump, even though `cmp` writes a condition flag and `jl` reads it. The branch prediction hardware, when it predicts the branch to be taken, can execute the target instruction in the cycle following the jump. When a multiple cycle instruction in the U pipe pairs with another instruction, the last memory operation of the U pipe instruction pairs with the first operation of the V pipe instruction (cycles 6 and 11).

3.3 Code Sequence 2, Intel486™ Processor

```
inc  dword ptr [eax*4+a]          ; 1   3 clocks with memory operand
  (inc)                           ; 2     plus 1 for indexing
  (inc)                           ; 3
  (inc)                           ; 4
inc  dword ptr [eax*4+b]          ; 5   3 clocks with memory operand
  (inc)                           ; 6     plus 1 for indexing
  (inc)                           ; 7
  (inc)                           ; 8
inc  eax                          ; 9
cmp  eax, 10                      ; 10
jl   TopOfLoop                    ; 11  2 clocks because jl is prefixed
                                  ; 12
                                  ; 13  Branch taken penalty
                                  ; 14
inc  dword ptr [eax*4+a]          ; 15  Next iteration
```

Total: 14 cycles

On the Intel486 processor, whenever an index register is used in an effective address calculation, there is a one clock penalty in the D2 stage (cycles 1 and 5). This does not apply to base registers.

3.4 Code Sequence 2, Pentium™ Processor

```
U pipe                  V pipe

inc  dword ptr [eax*4+a]                          ; 1 3 clocks with mem. op.
  (inc)                                           ; 2
  (inc)                 inc  dword ptr [eax*4+b] ; 3 1st V pairs with last U
                          (inc)                   ; 4
                          (inc)                   ; 5
inc  eax                                          ; 6
cmp  eax, 10            jl   TopOfLoop            ; 7
inc  dword ptr [eax*4+a]                          ; 8 Next iteration
```

Total: 7 cycles

The `inc eax` instruction at cycle 6 did not pair with the `cmp` instruction because of a register dependence. Other than in a few special cases (such as `cmp-jmp`), a register cannot be accessed until the cycle after it is written.

3.5 Code Sequence 3, Intel486™ Processor

```
mov    edx, [eax+40+a]        ; 1
                              ; 2  Fill prefetch buffer
mov    ecx, [eax+40+b]        ; 3
inc    edx                    ; 4
inc    ecx                    ; 5
mov    [eax+40+a], edx        ; 6
mov    [eax+40+b], ecx        ; 7
add    eax, 4                 ; 8
jnz    TopOfLoop              ; 9  Prefix on jnz
                              ; 10
                              ; 11 Branch penalty
                              ; 12
mov    edx, [eax+40+a]        ; 13 Next iteration
```

Total: 12 cycles

The delay at clock 2 is caused by a miss in the Intel486 processor's prefetch buffer. The previous two examples had a similar penalty, but it was hidden by the 2 clocks used for the shl instruction in code sequence 1, and by the index penalty in code sequence 2.

3.6 Code Sequence 3, Pentium™ Processor

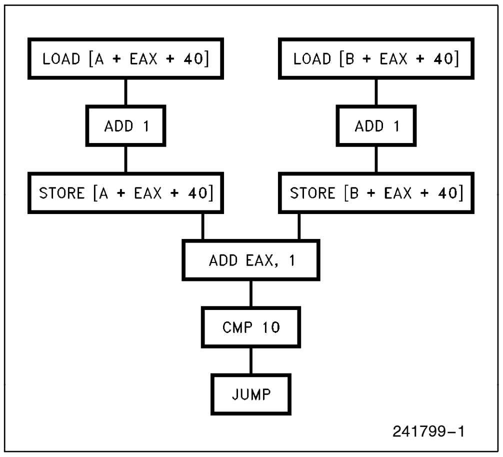
```
U pipe                    V pipe

mov  edx, [eax+40+a]      mov  ecx, [eax+40+b]  ; 1
inc  edx                  inc  ecx              ; 2
mov  [eax+40+a], edx      mov  [eax+40+b], ecx  ; 3
add  eax, 4               jnz  TopOfLoop        ; 4
mov  edx, [eax+40+a]      mov  ecx, [eax+40+b]  ; 5 AGI on eax with
  (mov)                     (mov)               ; 6 next iteration
```

Total: 5 cycles

The prefetch buffer delay on the Intel486 processor is no longer relevant. The Pentium processor has more prefetch buffers and different alignment capabilities. In the example above the loop control is determined by the `add eax, 4` instruction setting the zero condition code as it counts up to zero.

There is an AGI on `eax` because the `add` in cycle 4 writes to `eax` and both `mov`'s in cycle 5 reference it, even though there is a branch in between. On the Intel486 processor, AGI's only happen between adjacent instructions. On the Pentium processor, there can be two instructions in between and still be an AGI; for example, between a U pipe `add` in cycle n and a V pipe `mov` that uses the result of the add as a base in cycle n+1. The general rule is than an AGI will occur when any instruction in cycle n writes to a register that is used in an effective address calculation in any instruction in cycle n+1. This is because the effective address calculation is performed in D2.

In a compiler's intermediate representation of the program before instruction scheduling (reordering), one might expect the order for sequence 3 to be:



Reordering this intermediate code to obtain the assembly code shown earlier involves moving the load from "b" in front of the store into "a." Instruction reordering requires knowing that memory operands are independent. In this case, it can be easily proven that elements of "a" do not overlap in memory with elements of "b."

Comments

Using a load/store paradigm works well on the Pentium processor because it exposes more opportunities for pairing instructions when the instructions are scheduled. It does not, however, increase the number of clocks, even without scheduling, though this ignores possible secondary effects such as larger code size. This can lead to instruction cache misses. Another secondary effect is the use of more registers which are a limited resource on these on these processors. Compiler writers may want to pay more attention to register allocation.

In this document, we will refer to this as "load/store" style code generation.

4.0 CODE GENERATION STRATEGY

Even though each member of the Intel386 processor family has a different micro architecture due to technology versus implementation tradeoffs, the differences induced few conflicts in the overall code optimization strategy. In fact, there is a set of "blended" optimizations that will create an optimal binary across the entire family. The "blended" optimizations include:

1. Optimizations that benefit all members.
2. Optimizations that benefit one or more members but do not hurt the remaining members.
3. Optimizations that benefit one or more members a lot but only hurt the remaining members a little.

For those optimizations that benefit only certain members but cause noticeable degradation to others, it is recommended that they be implemented under switches and left to the user to decide whether maximizing the performance of a specific processor is desirable.

5.0 BLENDED CODE GENERATION CONSIDERATION

5.1 Choice of Index Versus Base Register

The Intel386 and the Intel486 processors need an additional clock cycle to generate an effective address when an index register is used. Therefore, if only one indexing component is used (i.e., not both a base register and an index register) and scaling is not necessary, then it is faster to use the register as a base rather than an index. For example:

```
mov eax, [esi]    ; use esi as base
mov eax, [esi*]   ; use esi as index, 1
                  ; clock penalty
```

It takes the Pentium processor one clock to calculate the effective address even when an index register is used. Hence, Pentium processor is neutral to the choice of index versus base register.

5.2 Addressing Modes and Register Usage

1. For the Intel486 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 a destination register
   mov eax, [esi]  ; esi is a base, 1 clock penalty
   ```

   Since the Pentium processor has two integer pipelines and each pipeline has an organization similar to the Intel486 processor's integer pipeline, 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 cycle (Address Generation Interlock, (AGI)). To avoid the AGI, the instructions should be separated by at least one cycle by placing other instructions between them.

2. 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:

   ```
   sub   esp, 24
           ;  1 cycle stall
   push  ebx

   mov   esp, ebp
           ;  1 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.

   Example:

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

3. On the Intel486 processor there is a 1 clock penalty for decoding an instruction with either an *index* or an *immediate-displacement* combination. On the Pentium processor, the *immediate-displacement* combination is not pairable. When it is necessary to use constants, it would still be more efficient to use immediate data instead of loading the constant into a register first, but if the same immediate data is used more than once, it would be faster to load the constant in a register and then use the register multiple times.

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

4. The Intel486 processor has a 1 clock penalty when using a register immediately after its sub-register was written. The Pentium processor is neutral in this respect.

   Example (Pentium Processor):

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

   Example (Intel486 processor):

```
mov   al, 0             ; 1
                        ; 2
mov   [ebp], eax        ; 3
```

5.3 Prefetch Bandwidth

The Intel486 processor prefetch unit will access the on-chip cache to fill the prefetch queue whenever the cache is idle, and there is enough room in the queue for another cache line (16 bytes). If the prefetch queue becomes empty, it can take up to three additional clocks to start the next instruction. The prefetch queue is 32 bytes in size (2 cache lines).

Because data accesses always have priority over prefetch requests, keeping the cache busy with data accesses can lock out the prefetch unit. As a result, optimized code should avoid four consecutive memory instructions.

It is important to arrange instructions so that the memory bus is not used continuously by a series of memory-reference instructions. The instructions should be rearranged so that there is a non-memory referencing instruction (such as a register instruction) at least two clocks before the prefetch queue becomes exhausted. This will allow the prefetch unit to transfer a cache line into the queue.

Such arrangement of the instructions will not affect the performance of the Intel386 and Pentium processors.

In general, it is difficult for a compiler to model the Intel486 CPU prefetch buffer behavior. A sequence of four consecutive memory instructions without stalls (i.e., index penalty) will probably stall because of the prefetch buffers being exhausted.

5.4 Alignment

5.4.1 CODE

The Intel486 processor has a cache line size of 16 bytes and the Pentium processor has a cache line size of 32 bytes. Since the Intel486 processor has only two prefetch buffers (16 bytes each), code alignment has a direct impact on Intel486 processor performance as a result of the prefetch buffer efficiency. Code alignment has little effect on the Pentium processor performance because of its ability to prefetch across a cache line boundary with no penalty. The Intel386 processor with no on-chip cache and a decoupled prefetch unit is not sensitive to code alignment. For optimal performance across the family, it is recommended that labels be aligned to the next 0MOD16 when it is less than 8 bytes away from that boundary.

5.4.2 DATA

A misaligned access in the data cache costs at least an extra 2 cycles on both the Intel486 and Pentium processor.

5.4.3 2-BYTE DATA

A 2-byte object should be fully contained within an aligned 4-byte word (i.e., its binary address should be xxxx00, xxxx01, xxxx10, but not xxxx11).

5.4.4 4-BYTE DATA

The alignment of a 4-byte object should be on a 4-byte boundary.

5.4.5 8-BYTE DATA

An 8-byte datum (64-bit, e.g., double precision reals) should be aligned on an 8-byte boundary.

5.5 Prefixed Opcodes

On the Intel386 processor and the Intel486 processor, all prefix opcodes require an additional clock to decode. 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). This is a special case of pairing. The prefixes are issued to the U pipe and get decoded in one cycle for each prefix and then the instruction is issued to the U pipe and may be paired.

All these prefixes: lock, segment override, address size, second opcode map (0f), and operand size belong to this group. Note that this includes all the 16-bit instructions when executing in 32-bit mode because an operand size prefix is required (e.g., `mov word ptr [..], add word ptr [..], ...`)

The *near jcc* prefix behaves differently; it does not take an extra cycle to decode and belongs to PV group. Other `0f` opcodes behave as normal prefixed instructions. For optimized code prefixed opcodes should be avoided.

When prefixed opcodes have to be used, there are two cases in which overlap can be achieved between the extra clock it takes to decode a prefix and a cycle used by the previous instruction executing in the same pipe:

1. The one cycle penalty from using the result register of a previous instruction as a base or index (AGI).
2. The last cycle of a preceding multi-cycle instruction.

5.6 Integer Instruction Scheduling

Instruction scheduling is the process of reordering the instructions in a program to avoid stalls and delays while maintaining the semantics of the generated code.

Scheduling of integer instructions has two purposes:

1. Eliminate stalls in the Intel486 CPU pipeline and each pipe of the Pentium processor.

There are some conditions where pipe stalls are encountered. The general guideline is to find instructions that can be inserted between the instructions that cause a stall. Since most of the commonly used integer instructions take only one clock, there is not much need to hide latencies. The most common delays which can be avoided through scheduling are AGI's.

2. Create pairs for maximum throughput from the Pentium processor's dual pipe architecture:

   The Pentium processor can issue two instructions for execution simultaneously. This is called *pairing*. There are limitations on which two instructions can be paired and some pairs, even when issued, will not execute in parallel. Pairing details are described in following sections. More information about instruction pairability can be found in Appendix A.

   Reordering instructions should be done in order to increase the possibility of issuing two instructions simultaneously. Dependent instructions should be separated by at least one other instruction. Scheduling for the Pentium processor's dual pipe is overkill for the Intel486 processor but has otherwise little effect on its performance.

The following subsections are Pentium processor specific optimizations. These optimizations do not adversely impact the Intel386 and Intel486 processors.

5.6.1 PAIRING

The Pentium processor can issue two instructions for execution simultaneously. This is called *pairing*. The limitations on which two instructions can be paired are discussed in this section. Some pairs, even when issued, will not execute in parallel.

Pairing cannot be performed when the following conditions occur:

1. 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.
2. 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.
3. Both instructions are not in the instruction cache. An exception to this which permits pairing is if the first instruction is a one-byte instruction.

5.6.2 INSTRUCTION SET PAIRABILITY

5.6.2.1 Unpairable Instructions (NP)

1. *shift/rotate* with the shift count in cl
2. Long-Arithmetic instructions for example, `mul, div`
3. Extended instructions for example, `ret, enter, pusha, movs, rep stos, loopnz`
4. Some Floating-Point Instructions for example, `fscale, fldcw, fst`
5. Inter-segment instructions for example, `push sreg, call far`

5.6.2.2 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`

5.6.2.3 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`

5.6.2.4 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 "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. `fxch`

5.6.3 UNPAIRABILITY DUE TO REGISTERS

The pairability of an instruction is also affected by its operands. The following are the combinations that are not pairable due to *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*).

   Example:

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

2. Both instructions write to the same register (*output-dependence*).

   Example:

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

This limitation does not apply to a pair of instructions which write to the eflags register (e.g. 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 it (anti-dependence) is pairable.

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. Hence,

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

do not pair due to apparent output dependencies on `eax`.

5.6.4 SPECIAL PAIRS

There are some instructions that can be paired although the general rule prohibits this. These special pairs overcome register dependencies. Most of these exceptions involve implicit reads/writes to the esp register or implicit writes to the condition codes:

Stack Pointer:

1. `push reg/imm; push reg/imm`
2. `push reg/imm; call`
3. `pop reg     ; pop reg`

Condition Codes:

1. `cmp         ; jcc`
2. `add         ; jne`

Note that the special pairs that consist of `push/pop` instructions may have only *immediate* or register operands.

5.6.5 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 2–4 are the same in the two physical addresses. This is because the cache is organized as 8 banks of 32-bit wide data entries. 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 above instructions 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 3 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 may 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. Compiler writers/programmers can also choose to reconstruct the complex form if the pairing opportunity does not materialize. The load/store style 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 the register allocator and instruction scheduler so that extra registers are only used when parallelism increases.

5.7 Integer Instruction Selection

The following highlights some instruction sequences to avoid and some sequences to use when generating optimal assembly code.

The *lea* instruction can be advantageous:

1. Lea may 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 and shift instructions may be used to replace constant multiply instructions.
3. This 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 shift takes 2 clocks on Intel486 processor whereas `lea` only takes one. On the Pentium processor, `lea` can execute in either U or V pipes, but shift can only execute in the U pipe.

Complex Instructions

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

Zero-Extension of Short

The `movzx` instruction has a prefix and takes 3 cycles to execute, totalling 4 cycles. As with the Intel486 processor, it is recommended 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 may be paired with adjacent instructions.

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  mem,  reg
push reg
```

Short Opcodes

Use one byte long 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`.

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.

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

Code for comparing these 16-bit operands might be:

```
xor  eax, eax  ; 1
mov  ax, [a]   ; 2   1 (prefix) + 1
mov  bx, [b]   ; 4   1 (prefix) + 1
cmp  eax, ebx  ; 6
```

The straightforward method may be slower:

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

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.

Compares

Use `test` when comparing a value in a register with 0. Test essentially "ands" the operands together without writing to a destination register. If you "and" a value 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 `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) { }).`

`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 4 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.

When there is a choice to use either a base or index register, always choose the base because there is a 1 clock penalty on the Intel486 processor for using an index.

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 both the Pentium and Intel486 processors, but the copy/shift scheme allows two other instructions to execute at the same time on the Pentium processor. If you know the value is positive, use `xor edx, edx`.

Prolog Sequences

Be careful to avoid AGI's in the prolog 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`.

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`.

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.

Epilog Sequence

If only 4 bytes were allocated in the stack frame for the current function, instead of incrementing the stack pointer by 4, use `pop` instructions. This avoids AGIs and helps both Intel486 and Pentium processor. For Pentium processor use 2 `pops` for eight bytes.

Integer Multiply by Constant

The integer multiply by an immediate can usually be replaced by a faster series of shifts, adds, subs, and leas.

1. Binary Method

   In general, if there are 8 or fewer bits set in the binary representation of the constant, it is better not to do the integer multiply. On an Intel486 processor, the break even point is lower: it is profitable if 6 bits or less are in the constant. Basically, shift and add for each bit set.

2. Factorization Method

   This is done by factoring the constant by powers of two plus or minus one, and the constant plus or minus one by powers of two. If the number can be factored by powers of two, then the multiplication can be performed by a series of shifts. If powers of two plus or minus one are included a shift of the previous result and an add or subtract of the previous result can be generated. If the given number plus or minus one can be factored by a power of two, a shift of the previous result and an add or subtract of the original operand can be generated An iterative for check posers of two from 31 to 1 can be done. The shift amount needed, and an ordinal to specify an add or subtract is saved for each factor. This information can be used in reverse order to generate the needed instructions.

For example:

```
imul  eax, 217 ; 10 clocks, no pairing
```

In checking powers of two in decreasing order it is found that 217 will divide by 31.

217/31 = 7. 31 = $2^5 - 1$

save shift = 5 and ordinal = sub__previous__result

After a check of 217/31 or 7, it is found that 7 + 1 is divisible by 8.

save shift = 3 and ordinal = sub__operand

After factoring the instructions can be generated in reverse.

```
mov  ecx, eax          ; 1
shl  eax, 3            ; 2
sub  eax, ecx          ; 3
mov  ecx, eax          ; 4
shl  eax, 5            ; 5
sub  eax, ecx          ; 6
```

This code sequence allows scheduling of other instructions in the Pentium processor's V pipe.

6.0 PROCESSOR SPECIFIC OPTIMIZATIONS

6.1 Pentium™ Processor Floating-Point Optimizations

The Pentium processor is the first generation of the Intel386 CPU family that implements a pipelined floating-point unit however, in order to achieve maximum throughput from the Pentium processor floating-point unit, specific optimizations must be done.

6.1.1 FLOATING-POINT EXAMPLE

FORTRAN source:

```
     subroutine da(x,y,z,n)
     dimension x(n),y(n)

     do 10 i=1,n
10   x(i) = x(i) + y(i) * z

     return
     end
```

Assembly code:

```
                                            Pentium/Intel486 processors
    TopOfLoop:
        fld     dword ptr  [esp+8]          ; 1    /     1
        fmul    dword ptr  [ebx+eax*4]      ; 2    /     5
        fadd    dword ptr  [ecx+eax*4]      ; 5    /    16
        fstp    dword ptr  [ecx+eax*4]      ; 9    /    26
        inc     eax                         ; 11   /    33
        cmp     eax, ebp                    ; 12   /    34
        jle     TopOfLoop                   ; 12   /    36+2 for branch
```

Total: 12 cycles per iteration

On the Intel486 processor, the time it takes to add and multiply varies depending on the values. In this example, 11 was used for multiply and 10 for add. The load takes 3 clocks; the store requires 7 clocks. The extra cycle before the `fmul` is an index penalty for the `fmul`. The `fadd` and `fstp` do not show an index penalty because the penalty overlapped with the execution of the previous floating-point instruction. These overlaps do not occur with `fld` or `fxch`.

On the Pentium processor, the results of `fadd` and `fmul` can be used three cycles after they start, except when the use is `fst`. When a `fst` instruction uses the result of another floating-point operation, an extra cycle is needed. The `fst` instruction executes for two cycles and nothing can execute in parallel.

There is an enormous improvement due to decreasing the clock counts for the common floating-point instructions; however, this example does not overlap any floating-point instructions. A further improvement can be achieved by overlapping the execution of the floating-point instructions as explained in the next section.

To expose more parallelism, loop unrolling can be used if the iterations are independent. Following is the assembly code after unrolling:

```
                                                    Pentium processor      Intel486 CPU
TopOfLoop:
     fld        dword ptr [esp+8]                     ; 1                    1
     fmul       dword ptr [ebx+eax*4]                 ; 2                    5
     fadd       dword ptr [ecx+eax*4]                 ; 5                    16
     fstp       dword ptr [ecx+eas*4]                 ; 9                    26
     fld        dword ptr [esp+8]                     ; 11                   33
     fmul       dword ptr [ebx+eax*4+4]               ; 12                   37
     fadd       dword ptr [ecx+eax*4+4]               ; 15                   48
     fstp       dword ptr [ecx+eax*4+4]               ; 19                   58
     fld        dword ptr [esp+8]                     ; 21                   65
     fmul       dword ptr [ebx+eax*4+8]               ; 22                   69
     fadd       dword ptr [ecx+eax*4+8]               ; 25                   80
     fstp       dword ptr [ecx+eax*4+8]               ; 29                   90
     add        eax, 3                                ; 31                   97
     cmp        eax, ebp                              ; 32                   98
     jle        TopOfLoop                             ; 32               100+2 (br taken)
```

Total: 32 cycles (10.7/ iteration)

The clock count improvements gained through loop unrolling was due to eliminating some of the loop control overhead. To get more improvement, we need to get the floating-point operations overlapped in order to hide their latencies.

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

One obvious way to get around this is to change the architecture and have floating-point registers, rather than a stack. Unfortunately, upward and downward compatibility would be lost. Instead, the `fxch` instruction was made "fast". This provides us another way to avoid the top of stack dependencies. The `fxch` instructions can be paired with the common floating-point operations, so there is no penalty on the Pentium processor. On the Intel486 processor, each `fxch` takes 4 clocks.

To take advantage of the exposed parallelism from loop unrolling, the instructions should be scheduled.

Assembly code after unrolling and scheduling:

					After Instruction		
			Intel486 CPU	Pentium CPU	ST(0)	ST(1)	ST(2)
TopOfLoop:							
	fld	dword ptr [esp+8]	1	1	z		
	fmul	dword ptr [ebx+eax*4]	5	2	y0*z		
	fld	dword ptr [esp+8]	16	3	z	y0*z	
	fmul	dword ptr [ebx+eax*4+4]	20	4	y1*z	y0*z	
	fxch	st(1)	31	4	y0*z	y1*z	
	fadd	dword ptr [ecx+eax*4]	36	5	x0+y0*z	y1*z	
	fld	dword ptr [esp+8]	46	6	z	x0+y0*z	y1*z
	fmul	dword ptr [ebx+eax*4+8]	50	7	y2*z	x0+y0*z	y1*z
	fxch	st(2)	61	7	y1*z	x0+y0*z	y2*z
	fadd	dword ptr [ecx+eax*4+4]	66	8	x1+y1*z	x0+y0*z	y2*z
	fxch	st(1)	76	8	x0+y0*z	x1+y1*z	y2*z
	fstp	dword ptr [ecx+eax*4]	81	9	x1+y1*z	y2*z	
	fxch	st(1)	88	11	y2*z	x1+y1*z	
	fadd	dword ptr [ecx+eax*4+8]	93	12	x2+y2*z	x1+y1*z	
	fxch	st(1)	103	12	x1+y1*z	x2+y2*z	
	fstp	dword ptr [ecx+eax*4+4]	108	13	x2+y2*z		
	fstp	dword ptr [ecx+eax*4+8]	116	16			
	add	eax, 3	123	18			
	cmp	eax, ebp	124	19			
	jle	TopOfLoop	126+2 (jle taken)	19			

Total: 19 cycles (6.3/iteration)

On the Intel486 processor, the index penalty and the added cost of `fxch` are apparent. The index penalty does not overlap with the `fxch` instruction.

On the Pentium processor, the `fxch` instructions pair with preceding fadd and fmul instructions and execute in parallel with them (cycles 7, 8, 12). The `fxch` instructions move an operand into position for the next floating-point instruction. There is a cycle lost at clock 15 due to the store waiting 3 clocks after the instruction defining its operand. The `fxch` instruction does not pair with `fst` and takes one clock as a separate instruction (cycles 9–11).

6.1.2 FXCH RULES AND REGULATIONS

The `fxch` instruction can be executed for "free" when all of the following conditions occur:

An FP instruction follows the `fxch` instruction.

An FP instruction belonging to the following list immediately precedes the fxch instruction: `fadd, fsub, fmul, fld, fcom, fucom, fchs, ftst, fabs, fdiv.`

This `fxch` instruction has already been executed. This is because the instruction boundaries in the cache are marked the first time the instruction is executed, so pairing only happens the second time this instruction is executed from the cache.

This means that this instruction is almost "free" and can be used to access elements in the deeper levels of the FP stack instead of storing them and then loading them again.

6.1.3 MEMORY OPERANDS

Performing a floating-point operation on a memory operand instead of on a stack register costs no cycles. In the integer part of the Pentium processor, it was better to avoid memory operands. In the floating-point part, you are encouraged to use memory operands.

6.1.4 FLOATING-POINT STALLS

There are cases where a delay occurs between two operations. Instructions should be inserted between the pair that cause the pipe stall. These instructions could be integer instructions or floating-point instructions that will not cause a new stall themselves. The number of instructions that should be inserted depends on the delay length.

One example of this is when a floating-point instruction depends on the result of the immediately preceding instruction which is also a floating-point instruction. In this case, it would be advantageous to move integer instructions between the two fp instructions, even if the integer instructions perform loop control. The following example restructures a loop in this manner:

```
for (i=0; i<Size; i++)
      array1 [i] += array2 [i];
                                    Pentium Processor          Intel486 Processor
                                         Clocks                     Clocks
 TopOfLoop:
   flds  [eax + array2]            2 - AGI                    3
   fadds [eax + array1]            1                          3
   fstps [eax + array1]            5 - Wait for fadds         14 - Wait for fadds
   add   eax, 4                    1                          1
   jnz   TopOfLoop                 0 - Pairs with add         3
                                   ___                        ___
                                   9                          24


                                    Pentium Processor          Intel486 Processor
                                         Clocks                     Clocks
 TopOfLoop:
   fstps  [eax + array1]           4 - Wait for fadds, AGI    10 - Wait for fadds
 LoopEntryPoint:
   flds   [eax + array2]           1                          3
   fadds  [eax + array1]           1                          3
   add    eax, 4                   1                          1
   jnz    TopOfLoop                0 - Pairs with add         3
                                   ___                        ___
                                   7                          20
```

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

1. Floating-Point Stores

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

   NOTE:
   1. This set includes also the `faddp, fsubrp`, ... instructions

```
fld     mem1            ; 1 fld takes 1 clock
                        ; 2 fst waits, schedule something here
fst     mem2            ; 3,4 fst takes 2 clocks
fadd    mem1            ; 1 add takes 3 clocks
                        ; 2 add, schedule something here
                        ; 3 add, schedule something here
                        ; 4 fst waits, schedule something here
fst     mem2            ; 5,2 fst takes 2 clocks
```

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

```
fld     mem1            ; 1
fld     mem2            ; 2
fxch    st(1)           ; 2
fst     mem3            ; 3 stores values loaded from mem1
```

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

```
fld     mem1            ; 1
fadd    mem2            ; 2,3,4
```

3. Use of a register by a floating-point operation immediately after it has been written by another `fadd, fsub`, or `fmul` causes a 2 cycle delay. If instructions are inserted between these two, then latency and a potential stall can be hidden.

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

5. The integer multiply operations, `mul` and `imul`, are executed in the floating-point unit so these instructions cannot be executed in parallel with a floating-point instruction.
6. A floating-point multiply instruction (`fmul`) delays for one cycle if the immediately preceding cycle executed an `fmul` or an `fmul/fxch` pair. The multiplier can only accept a new pair of operands every other cycle.
7. Transcendental operations execute in the U pipe and nothing can be overlapped with them, so an integer instruction following such an instruction will wait until that instruction completes.
8. Floating-point operations that take integer operands (`fiadd` or `fisub` ..) should be avoided. These instructions should be split into two instructions: fild and a floating-point operation. The number of cycles before another instruction can be issued (throughput) for fiadd is 4, while for fild and simple floating-point op it is 1.

   Example:

```
Complex Instructions    Better for Potential Overlap
fiadd   [ebp] ; 4       fild     [ebp]  ; 1
                        faddp    st(1)  ; 2
```

   Using the `fild-faddp` instructions yields 2 free cycles for executing other instructions.
9. The `fstsw` instruction that usually appears after a floating-point comparison instruction (`fcom, fcomp, fcompp`) delays for 3 cycles. Other instructions may be inserted after the comparison instruction in order to hide the latency.
10. Moving a floating-point memory/immediate to memory should be done by integer moves (if precision conversion is not needed) instead of doing `fld-fstp`.

    Examples for floating-point moves:

    double precision: 4 vs. 2 cycles

```
fld    [ebp]          ; 1        mov eax, [ebp]          ; 1
                      ; 2        mov edx, [ebp+4]        ; 1
fstp   [edi]          ; 3, 4     mov [edi], eax          ; 2
                                 mov [edi+4], edx        ; 2
```

    single precision: 4 vs. 2 cycles

```
fld    [ebp]          ; 1        mov eax, [ebp]          ; 1
                      ; 2        mov [edi], eax          ; 2
fstp   [edi]          ; 3, 4
```

    This optimization also applies to the Intel486 processor.
11. Transcendental operations execute on the Pentium processor much faster than on the Intel486 processor. It may be worthwhile in-lining some of these math library calls because of the fact that the `call` and *prologue/epilogue* overhead involved with the library calls is no longer negligible. Emulating these operations in software will not be faster than the hardware unless accuracy is sacrificed.
12. Integer instructions generally overlap with the floating-point operations except when the last floating-point operation was fxch. In this case there is a one cycle delay.

```
U pipe          V pipe
fadd            fxch          ; 1
                              ; 2 fxch delay
mov eax, 1      inc  edx      ; 3
```

7.0 SUMMARY

The following tables summarize the micro architecture differences among Intel386, Intel486 and Pentium processors and the corresponding code generation consideration. It is possible to derive a set of code generation strategies that provide the optimal performance across the various members of the Intel386 processor family except for the use of FXCH to maximize the Pentium processor floating-point throughput which can be implemented under a user-directed option.

	Intel386™ Processor	Intel486™ Processor	Pentium™ Processor
Cache	None	8K Combined	8K Code, 8K Data
Prefetch	4x4b filled by external memory access	2x6b shared bus to cache	4x32b private bus to cache
Decoder	3 deep decoded FIFO	Part of core pipeline	Part of core pipeline
Core	Some instruction overlap	5 stages pipeline	5 stages pipeline and superscalar
Math	Co-Processor	On-Chip	On-Chip and pipelined

Processor Characteristics	Optimizations	Intel386™ Processor	Intel486™ Processor	Pentium™ Processor
Cache	Interleave mem with non-mem	Don't care	Interleave if 4 consecutive	Don't care
Prefetcher	Alignment	0-MOD-4	0-MOD-16	Don't care
Pipelined Execution Core	Base vs index	Don't care	Use base	Don't care
	Avoid AGI	Don't care	Next instr	Next 3 instr
	Instruction selection	1 clk penalty	Short instr	Short instr
Superscalar	Pairing	Don't care	Don't care	Pair
Pipelined FPU with FXCH	More scheduling	18 clk penalty	4 clk penalty	Schedule

Recommendations for Blended:

1. Interleave mem with non-mem: do nothing
2. Code alignment: 0-mod-16 on loop
3. Base vs index: use base
4. Avoid AGI: next 3 instructions
5. Instruction selection: short instructions sequence
6. Pairing: pair
7. FP scheduling: avoid FXCH

APPENDIX A
INSTRUCTION PAIRING SUMMARY

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

NP—	Not pairable, executes in U-pipe
PU—	Pairable if issued to U-pipe
PV—	Pairable if issued to V-pipe
UV—	Pairable in either pipe

In the floating-point table:

FX—	Pairs with FXCH
NP—	No pairing.

The I/O instructions are not pairable.

Integer Instruction Pairing

Instruction	Format	Pairing
AAA—ASCII Adjust after Addition		NP
AAD—ASCII Adjust AX before Division		NP
AAM—ASCII Adjust AX after Multiply		NP
AAS—ASCII Adjust AL after Subtraction		NP
ADC—ADD with Carry		PU
ADD—Add		UV
AND—Logical AND		UV
ARPL—Adjust RPL Field of Selector		NP
BOUND—Check Array Against Bounds		NP
BSF—Bit Scan Forward		NP
BSR—Bit Scan Reverse		NP
BSWAP—Byte Swap		NP
BT—Bit Test		NP
BTC—Bit Test and Complement		NP
BTR—Bit Test and Reset		NP
BTS—Bit Test and Set		NP
CALL—Call Procedure (in same segment)		
direct	1110 1000 : full displacement	PV
register indirect	1111 1111 : 11 010 reg	NP
memory indirect	1111 1111 : mod 010 r/m	NP
CALL—Call Procedure (in other segment)		NP
CBW—Convert Byte to Word		NP
CWDE—Convert Word to Doubleword		
CLC—Clear Carry Flag		NP
CLD—Clear Direction Flag		NP
CLI—Clear Interrupt Flag		NP
CLTS—Clear Task-Switched Flag in CR0		NP
CMC—Complement Carry Flag		NP
CMP—Compare Two Operands		UV
CMPS/CMPSB/CMPSW/CMPSD—Compare String Operands		NP
CMPXCHG—Compare and Exchange		NP
CMPXCHG8B—Compare and Exchange 8 Bytes		NP
CWD—Convert Word to Dword		NP
CDQ—Convert Dword to Qword		
DAA—Decimal Adjust AL after Addition		NP
DAS—Decimal Adjust AL after Subtraction		NP
DEC—Decrement by 1		UV
DIV—Unsigned Divide		NP
ENTER—Make Stack Frame for Procedure Parameters		NP
HLT—Halt		

Integer Instruction Pairing

Instruction	Format	Pairing
IDIV—Signed Divide		NP
IMUL—Signed Multiply		NP
INC—Increment by 1		UV
INT n—Interrupt Type n		NP
INT—Single-Step Interrupt 3		NP
INTO—Interrupt 4 on Overflow		NP
INVD—Invalidate Cache		NP
INVLPG—Invalidate TLB Entry		NP
IRET/IRETD—Interrupt Return		NP
Jcc—Jump if Condition is Met		PV
JCXZ/JECXZ—Jump on CX/ECX Zero		NP
JMP—Unconditional Jump (to same segment)		
short	1110 1011 : 8-bit displacement	PV
direct	1110 1001 : full displacement	PV
register indirect	1111 1111 : 11 100 reg	NP
memory indirect	1111 1111 : mod 100 r/m	NP
JMP—Unconditional Jump (to other segment)		NP
LAHF—Load Flags into AH Register		NP
LAR—Load Access Rights Byte		NP
LDS—Load Pointer to DS		NP
LEA—Load Effective Address		UV
LEAVE—High Level Procedure Exit		NP
LES—Load Pointer to ES		NP
LFS—Load Pointer to FS		NP
LGDT—Load Global Descriptor Table Register		NP
LGS—Load Pointer to GS		NP
LIDT—Load Interrupt Descriptor Table Register		NP
LLDT—Load Local Descriptor Table Register		NP
LMSW—Load Machine Status Word		NP
LOCK—Assert LOCK# Signal Prefix		
LODS/LODSB/LODSW/LODSD—Load String Operand		NP
LOOP—Loop Count		NP
LOOPZ/LOOPE—Loop Count while Zero/Equal		NP
LOOPNZ/LOOPNE—Loop Count while not Zero/Equal		NP
LSL—Load Segment Limit		NP
LSS—Load Pointer to SS	0000 1111 : 1011 0010 : mod reg r/m	NP
LTR—Load Task Register		NP
MOV—Move Data		UV
MOV—Move to/from Control Registers		NP
MOV—Move to/from Debug Registers		NP

Integer Instruction Pairing

Instruction	Format	Pairing
MOV—Move to/from Segment Registers		NP
MOVS/MOVSB/MOVSW/MOVSD—Move Data from String to String		NP
MOVSX—Move with Sign-Extend		NP
MOVZX—Move with Zero-Extend		NP
MUL—Unsigned Multiplication of AL or AX		NP
NEG—Two's Complement Negation		NP
NOP—No Operation	1001 0000	UV
NOT—One's Complement Negation		NP
OR—Logical Inclusive OR		UV
POP—Pop a Word from the Stack		
reg	1000 1111 : 11 000 reg	UV
or	0101 1 reg	UV
memory	1000 1111 : mod 000 r/m	NP
POP—Pop a Segment Register from the Stack		NP
POPA/POPAD—Pop All General Registers		NP
POPF/POPFD—Pop Stack into FLAGS or EFLAGS Register		NP
PUSH—Push Operand onto the Stack		
reg	1111 1111 : 11 110 reg	UV
or	0101 0 reg	UV
memory	1111 1111 : mod 110 r/m	NP
immediate	0110 10s0 : immediate data	UV
PUSH—Push Segment Register onto the Stack		NP
PUSHA/PUSHAD—Push All General Registers		NP
PUSHF/PUSHFD—Push Flags Register onto the Stack		NP
RCL—Rotate thru Carry Left		
reg by 1	1101 000w : 11 010 reg	PU
memory by 1	1101 000w : mod 010 r/m	PU
reg by CL	1101 001w : 11 010 reg	NP
memory by CL	1101 001w : mod 010 r/m	NP
reg by immediate count	1100 000w : 11 010 reg : imm8 data	PU
memory by immediate count	1100 000w : mod 010 r/m : imm8 data	PU
RCR—Rotate thru Carry Right		
reg by 1	1101 000w : 11 011 reg	PU
memory by 1	1101 000w : mod 011 r/m	PU
reg by CL	1101 001w : 11 011 reg	NP
memory by CL	1101 001w : mod 011 r/m	NP
reg by immediate count	1100 000w : 11 011 reg : imm8 data	PU
memory by immediate count	1100 000w : mod 011 r/m : imm8 data	PU
RDMSR—Read from Model-Specific Register		NP
REP LODS—Load String		NP

Integer Instruction Pairing

Instruction	Format	Pairing
REP MOVS—Move String		NP
REP STOS—Store String		NP
REPE CMPS—Compare String (Find Non-Match)		NP
REPE SCAS—Scan String (Find Non-AL/AX/EAX)		NP
REPNE CMPS—Compare String (Find Match)		NP
REPNE SCAS—Scan String (Find AL/AX/EAX)		NP
RET—Return from Procedure (to same segment)		NP
RET—Return from Procedure (to other segment)		NP
ROL—Rotate (not thru Carry) Left		
reg by 1	1101 000w : 11 000 reg	PU
memory by 1	1101 000w : mod 000 r/m	PU
reg by CL	1101 001w : 11 000 reg	NP
memory by CL	1101 001w : mod 000 r/m	NP
reg by immediate count	1100 000w : 11 000 reg : imm8 data	PU
memory by immediate count	1100 000w : mod 000 r/m : imm8 data	PU
ROR—Rotate (not thru Carry) Right		
reg by 1	1101 000w : 11 001 reg	PU
memory by 1	1101 000w : mod 001 r/m	PU
reg by CL	1101 001w : 11 001 reg	NP
memory by CL	1101 001w : mod 001 r/m	NP
reg by immediate count	1100 000w : 11 001 reg : imm8 data	PU
memory by immediate count	1100 000w : mod 001 r/m : imm8 data	PU
RSM—Resume from System Management Mode		NP
SAHF—Store AH into Flags		NP
SAL—Shift Arithmetic Left	same instruction as SHL	
SAR—Shift Arithmetic Right		
reg by 1	1101 000w : 11 111 reg	PU
memory by 1	1101 000w : mod 111 r/m	PU
reg by CL	1101 001w : 11 111 reg	NP
memory by CL	1101 001w : mod 111 r/m	NP
reg by immediate count	1100 000w : 11 111 reg : imm8 data	PU
memory by immediate count	1100 000w : mod 111 r/m : imm8 data	PU
SBB—Integer Subtraction with Borrow		PU
SCAS/SCASB/SCASW/SCASD—Scan String		NP
SETcc—Byte Set on Condition		NP
SGDT—Store Global Descriptor Table Register		NP

Integer Instruction Pairing

Instruction	Format	Pairing
SHL—Shift Left		
reg by 1	1101 000w : 11 100 reg	PU
memory by 1	1101 000w : mod 100 r/m	PU
reg by CL	1101 001w : 11 100 reg	NP
memory by CL	1101 001w : mod 100 r/m	NP
reg by immediate count	1100 000w : 11 100 reg : imm8 data	PU
memory by immediate count	1100 000w : mod 100 r/m : imm8 data	PU
SHLD—Double Precision Shift Left		
register by immediate count	0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8	NP
memory by immediate count	0000 1111 : 1010 0100 : mod reg r/m : imm8	NP
register by CL	0000 1111 : 1010 0101 : 11 reg2 reg1	NP
memory by CL	0000 1111 : 1010 0101 : mod reg r/m	NP
SHR—Shift Right		
reg by 1	1101 000w : 11 101 reg	PU
memory by 1	1101 000w : mod 101 r/m	PU
reg by CL	1101 001w : 11 101 reg	NP
memory by CL	1101 001w : mod 101 r/m	NP
reg by immediate count	1100 000w : 11 101 reg : imm8 data	PU
memory by immediate count	1100 000w : mod 101 r/m : imm8 data	PU
SHRD—Double Precision Shift Right		
register by immediate count	0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8	NP
memory by immediate count	0000 1111 : 1010 1100 : mod reg r/m : imm8	NP
register by CL	0000 1111 : 1010 1101 : 11 reg2 reg1	NP
memory by CL	0000 1111 : 1010 1101 : mod reg r/m	NP
SIDT—Store Interrupt Descriptor Table Register		NP
SLDT—Store Local Descriptor Table Register		NP
SMSW—Store Machine Status Word		NP
STC—Set Carry Flag		NP
STD—Set Direction Flag		NP
STI—Set Interrupt Flag		
STOS/STOSB/STOSW/STOSD—Store String Data		NP
STR—Store Task Register		NP
SUB—Integer Subtraction		UV
TEST—Logical Compare		
reg1 and reg2	1000 010w : 11 reg1 reg2	UV
memory and register	1000 010w : mod reg r/m	UV
immediate and register	1111 011w : 11 000 reg : immediate data	NP
immediate and accumulator	1010 100w : immediate data	UV
immediate and memory	1111 011w : mod 000 r/m : immediate data	NP

Integer Instruction Pairing

Instruction	Format	Pairing
VERR—Verify a Segment for Reading		NP
VERW—Verify a Segment for Writing		NP
WAIT—Wait	1001 1011	NP
WBINVD—Write-Back and Invalidate Data Cache		NP
WRMSR—Write to Model-Specific Register		NP
XADD—Exchange and Add		NP
XCHG—Exchange Register/Memory with Register		NP
XLAT/XLATB—Table Look-up Translation		NP
XOR—Logical Exclusive OR		UV

Floating-Point Instruction Pairing

Instruction	Format	Pairing
F2XM1—Compute $2^{ST(0)} - 1$		NP
FABS—Absolute Value		FX
FADD—Add		FX
FADDP—Add and Pop		FX
FBLD—Load Binary Coded Decimal		NP
FBSTP—Store Binary Coded Decimal and Pop		NP
FCHS—Change Sign		FX
FCLEX—Clear Exceptions		NP
FCOM—Compare Real		FX
FCOMP—Compare Real and Pop		FX
FCOMPP—Compare Real and Pop Twice		
FCOS—Cosine of ST(0)		NP
FDECSTP—Decrement Stack-Top Pointer		NP
FDIV—Divide		FX
FDIVP—Divide and Pop		FX
FDIVR—Reverse Divide		FX
FDIVRP—Reverse Divide and Pop		FX
FFREE—Free ST(i) Register		NP
FIADD—Add Integer		NP
FICOM—Compare Integer		NP
FICOMP—Compare Integer and Pop		NP
FIDIV		NP
FIDIVR		NP
FILD—Load Integer		NP
FIMUL		NP
FINCSTP—Increment Stack Pointer		NP
FINIT—Initialize Floating-Point Unit		NP
FIST—Store Integer		NP
FISTP—Store Integer and Pop		NP
FISUB		NP
FISUBR		NP
FLD—Load Real		
32-bit memory	11011 001 : mod 000 r/m	FX
64-bit memory	11011 101 : mod 000 r/m	FX
80-bit memory	11011 011 : mod 101 r/m	NP
ST(i)	11011 001 : 11 000 ST(i)	FX
FLD1—Load + 1.0 into ST(0)		NP
FLDCW—Load Control Word		NP
FLDENV—Load FPU Environment		NP
FLDL2E—Load $\log_2(e)$ into ST(0)		NP
FLDL2T—Load $\log_2(10)$ into ST(0)		NP

Floating-Point Instruction Pairing (Continued)

Instruction Format	Pairing
FLDLG2—Load $\log_{10}(2)$ into ST(0)	NP
FLDLN2—Load $\log_e(2)$ into ST(0)	NP
FLDPI—Load π into ST(0)	NP
FLDZ—Load +0.0 into ST(0)	NP
FMUL—Multiply	FX
FMULP—Multiply	FX
FNOP—No Operation	NP
FPATAN—Partial Arctangent	NP
FPREM—Partial Remainder	NP
FPREM1—Partial Remainder (IEEE)	NP
FPTAN—Partial Tangent	NP
FRNDINT—Round to Integer	
FRSTOR—Restore FPU State	NP
FSAVE—Store FPU State	NP
FSCALE—Scale	NP
FSIN—Sine	NP
FSINCOS—Sine and Cosine	NP
FSQRT—Square Root	NP
FST—Store Real	NP
FSTCW—Store Control Word	NP
FSTENV—Store FPU Environment	NP
FSTP—Store Real and Pop	NP
FSTSW—Store Status Word into AX	NP
FSTSW—Store Status Word into Memory	NP
FSUB—Subtract	FX
FSUBP—Subtract and Pop	FX
FSUBR—Reverse Subtract	FX
FSUBRP—Reverse Subtract and Pop	FX
FTST—Test	FX
FUCOM—Unordered Compare Real	FX
FUCOMP—Unordered Compare and Pop	FX
FUCOMPP—Unordered Compare and Pop Twice	FX
FXAM—Examine	NP
FXCH—Exchange ST(0) and ST(i)	
FXTRACT—Extract Exponent and Significand	NP
FYL2X—ST(1) $\times$ $\log_2$(ST(0))	NP
FYL2XP1—ST(1) $\times$ $\log_2$(ST(0) + 1.0)	NP
FWAIT—Wait until FPU Ready	