How to optimize for the Pentium family of microprocessors

cse.chalmers.se/~abela/lehre/WS07-08/Compiler/Lec10/Pentium_optimise.html

Contents

1. Introduction
2. Literature
3. Calling assembly functions from high level language
4. Debugging and verifying
5. Memory model
6. Alignment
7. Cache
8. First time versus repeated execution
9. Address generation interlock (PPlain and PMMX)
10. Pairing integer instructions (PPlain and PMMX)
    1. Perfect pairing
    2. Imperfect pairing
11. Splitting complex instructions into simpler ones (PPlain and PMMX)
12. Prefixes (PPlain and PMMX)
13. Overview of PPro, PII and PIII pipeline
14. Instruction decoding (PPro, PII and PIII)
15. Instruction fetch (PPro, PII and PIII)
16. Register renaming (PPro, PII and PIII)
    1. Eliminating dependencies
    2. Register read stalls
17. Out of order execution (PPro, PII and PIII)
18. Retirement (PPro, PII and PIII)
19. Partial stalls (PPro, PII and PIII)
20. Dependency chains (PPro, PII and PIII)
21. Searching for bottlenecks (PPro, PII and PIII)
22. Jumps and branches (all processors)
23. Reducing code size (all processors)
24. Scheduling floating point code (PPlain and PMMX)
25. Loop optimization (all processors)
    1. Loops in PPlain and PMMX
    2. Loops in PPro, PII and PIII
26. Problematic Instructions
27. Special topics
28. List of instruction timings for PPlain and PMMX
29. List of instruction timings and micro-op breakdown for PPro, PII and PIII
30. Testing speed

31. Comparison of the different microprocessors

1. Introduction

This manual describes in detail how to write optimized assembly language code, with particular focus on the Pentium® family of microprocessors.

Most of the information herein is based on my own research. Many people have sent me useful information and corrections for this manual, and I keep updating it whenever I have new important information. This manual is therefore more accurate, detailed, comprehensive and exact than any other source of information, and it contains many details not found anywhere else. This information will enable you in many cases to calculate exactly how many clock cycles a piece of code will take. I do not claim, though, that all information in this manual is exact: Some timings etc. can be difficult or impossible to measure exactly, and I do not have access to the inside information on technical implementations that the writers of Intel manuals have.

The following versions of Pentium processors are discussed in this manual:

abbreviation	name
PPlain	plain old Pentium (without MMX)
PMMX	Pentium with MMX
PPro	Pentium Pro
PII	Pentium II (including Celeron and Xeon)
PIII	Pentium III (including variants)

The assembly language syntax used in this manual is MASM 5.10 syntax. There is no official standard for X86 assembly language, but this is the closest you can get to a de facto standard since most assemblers have a MASM 5.10 compatible mode. (I do not recommend using MASM version 5.10 though, because it has a serious bug in 32 bit mode. Use TASM or a later version of MASM).

Some of the remarks in this manual may seem like a criticism of Intel. This should not be taken to mean that other brands are better. The Pentium family of microprocessors compare well with competing brands, they are better documented, and have better testability features. For these reasons, no competing brand has been subjected to the same level of independent research by me or by anybody else.

Programming in assembly language is much more difficult than high level language. Making bugs is very easy, and finding them is very difficult. Now you have been warned! It is assumed that the reader is already experienced in assembly programming. If not, then please read some books on the subject and get some programming experience before you begin to do complicated optimizations.

The hardware design of the PPlain and PMMX chips has many features which are optimized specifically for some commonly used instructions or instruction combinations, rather than using general optimization methods. Consequently, the rules for optimizing software for this design are complicated and have many exceptions, but the possible gain in performance may be substantial. The PPro, PII and PIII processors have a very different design where the processor takes care of much of the optimization work by executing instructions out of order, but the more complicated design of these processors generate many potential bottlenecks, so there may be a lot to gain by optimizing manually for these processors. The Pentium 4 processor has yet another design, and the optimization guidelines for Pentium 4 are quite different from previous versions. This manual does not cover the Pentium 4 - the reader is referred to manuals from Intel.

Before you start to convert your code to assembly, make sure that your algorithm is optimal. Often you can improve a piece of code much more by improving the algorithm than by converting it to assembly code.

Next, you have to identify the critical parts of your program. Often more than 99% of the CPU time is spent in the innermost loop of a program. In this case you should optimize only this loop and leave everything else in high level language. Some assembly programmers waste a lot of energy optimizing the wrong parts of their programs, the only significant effect of their effort being that the programs become more difficult to debug and maintain!

If it is not obvious where the critical parts of your program are then you may use a profiler to find them. If it turns out that the bottleneck is disk access, then you may modify your program to make disk access sequential in order to improve disk caching, rather than turning to assembly programming. If the bottleneck is graphics output then you may look for a way of reducing the number of calls to graphic procedures.

Some high level language compilers offer relatively good optimization for specific processors, but further optimization by hand can usually make it much better.

Please don't send your programming questions to me. I am not gonna do your homework for you!

Good luck with your hunt for nanoseconds!

2. Literature

A lot of useful literature and tutorials can be downloaded for free from Intel's www site or acquired in print or on CD-ROM. It is recommended that you study this literature in order to get acquainted with the microprocessor architecture. However, the documents from Intel are not always accurate - especially the tutorials have many errors (evidently, they haven't tested their own examples).

I will not give the URL's here because the file locations change very often. You can find the documents you need by using the search facilities at: developer.intel.com or follow the links from www.agner.org/assem

Some documents are in .PDF format. If you don't have software for viewing or printing .PDF files, then you may download the Acrobat file reader from www.adobe.com

The use of MMX and XMM (SIMD) instructions for optimizing specific applications are described in several application notes. The instruction set is described in various manuals and tutorials.

VTUNE is a software tool from Intel for optimizing code. I have not tested it and can therefore not give any evalutation of it here.

A lot of other sources than Intel also have useful information. These sources are listed in the FAQ for the newsgroup comp.lang.asm.x86. For other internet ressources follow the links from www.agner.org/assem

3. Calling assembly functions from high level language

You can either use inline assembly or code a subroutine entirely in assembly language and link it into your project. If you choose the latter option, then it is recommended that you use a compiler which is capable of translating high level code directly to assembly. This assures that you get the function calling method right. Most C++ compilers can do this.

The methods for function calling and name mangling can be quite complicated. There are many different calling conventions, and the different brands of compilers are not compatible in this respect. If you are calling assembly language subroutines from C++, then the best method in terms of consistency and compatibility is to declare your functions `extern "C"` and `_cdecl`. The assembly code must then have the function name prefixed by an underscore (`_`) and be assembled with case sensitivity on externals (option -mx).

If you need to make overloaded functions, overloaded operators, member functions, and other C++ specialties then you have to code it in C++ first and make your compiler translate it to assembly in order to get the right linking information and calling method. These details are different for different brands of compilers. If you want an assembly function with any other calling method than `extern "C"` and `_cdecl` to be callable from code compiled with different compilers then you need to give it one public name for each compiler. For example an overloaded square function:

```
; int square (int x);
SQUARE_I PROC NEAR             ; integer square function
@square$qi LABEL NEAR          ; link name for Borland compiler
?square@@YAHH@Z LABEL NEAR     ; link name for Microsoft compiler
_square__Fi LABEL NEAR         ; link name for Gnu compiler
PUBLIC @square$qi, ?square@@YAHH@Z, _square__Fi
        MOV     EAX, [ESP+4]
        IMUL    EAX
        RET
SQUARE_I ENDP

; double square (double x);
SQUARE_D PROC NEAR             ; double precision float square function
@square$qd LABEL NEAR          ; link name for Borland compiler
?square@@YANN@Z LABEL NEAR     ; link name for Microsoft compiler
_square__Fd LABEL NEAR         ; link name for Gnu compiler
PUBLIC @square$qd, ?square@@YANN@Z, _square__Fd
        FLD     QWORD PTR [ESP+4]
        FMUL    ST(0), ST(0)
        RET
SQUARE_D ENDP
```

The way of transferring parameters depends on the calling convention:

calling convention	parameter order on stack	parameters removed by
_cdecl	first par. at low address	caller
_stdcall	first par. at low address	subroutine
_fastcall	compiler specific	subroutine
_pascal	first par. at high address	subroutine

Register usage in 16 bit mode DOS or Windows, C or C++:
16-bit return value in `AX`, 32-bit return value in `DX:AX`, floating point return value in `ST(0)`. Registers `AX, BX, CX, DX, ES` and arithmetic flags may be changed by the procedure; all other registers must be saved and restored. A procedure can rely on `SI, DI, BP, DS` and `SS` being unchanged across a call to another procedure.

Register usage in 32 bit Windows, C++ and other programming languages:
Integer return value in `EAX`, floating point return value in `ST(0)`. Registers `EAX, ECX, EDX` (not `EBX`) may be changed by the procedure; all other registers must be saved and restored. Segment registers cannot be changed, not even temporarily. `CS, DS, ES,` and `SS` all point to the flat segment group. `FS` is used by the operating system. `GS` is unused, but reserved. Flags may be changed by the procedure with the following restrictions: The direction flag is 0 by default. The direction flag may be set temporarily, but must be cleared before any call or return. The interrupt flag cannot be cleared. The floating point register stack is empty at the entry of a procedure and must be empty at return, except for `ST(0)` if it is used for return value. MMX registers may be changed by the procedure and if so cleared by `EMMS` before returning and before calling any other procedure that may use floating point registers. All XMM registers may be modified by procedures. Rules for passing parameters and return values in XMM registers are

described in Intel's application note AP 589. A procedure can rely on `EBX, ESI, EDI, EBP` and all segment registers being unchanged across a call to another procedure.

4. Debugging and verifying

Debugging assembly code can be quite hard and frustrating, as you probably already have discovered. I would recommend that you start with writing the piece of code you want to optimize as a subroutine in a high level language. Next, write a test program that will test your subroutine thoroughly. Make sure the test program goes into all branches and boundary cases.

When your high level language subroutine works with your test program then you are ready to translate the code to assembly language.

Now you can start to optimize. Each time you have made a modification you should run it on the test program to see if it works correctly. Number all your versions and save them so that you can go back and test them again in case you discover an error that the test program didn't catch (such as writing to a wrong address).

Test the speed of the most critical part of your program with the method described in chapter 30 or with a test program. If the code is significantly slower than expected, then the most probable causes are: cache misses (chapter 7), misaligned operands (chapter 6), first time penalty (chapter 8), branch mispredictions (chapter 22), instruction fetch problems (chapter 15), register read stalls (16), or long dependency chains (chapter 20).

Highly optimized code tends to be very difficult to read and understand for others, and even for yourself when you get back to it after some time. In order to make it possible to maintain the code it is important that you organize it into small logical units (procedures or macros) with a well-defined interface and appropriate comments. The more complicated the code is to read, the more important is a good documentation.

5. Memory model

The Pentiums are designed primarily for 32 bit code, and the performance is inferior on 16 bit code. Segmenting your code and data also degrades performance significantly, so you should generally prefer 32 bit flat mode, and an operating system which supports this mode. The code examples shown in this manual assume a 32 bit flat memory model, unless otherwise specified.

6. Alignment

All data in RAM should be aligned to addresses divisible by 2, 4, 8, or 16 according to this scheme:

	alignment	
operand size	PPlain and PMMX	PPro, PII and PIII

1 (byte)	1	1
2 (word)	2	2
4 (dword)	4	4
6 (fword)	4	8
8 (qword)	8	8
10 (tbyte)	8	16
16 (oword)	n.a.	16

On PPlain and PMMX, misaligned data will take at least 3 clock cycles extra to access if a 4 byte boundary is crossed. The penalty is higher when a cache line boundary is crossed.

On PPro, PII and PIII, misaligned data will cost you 6-12 clocks extra when a cache line boundary is crossed. Misaligned operands smaller than 16 bytes that do not cross a 32 byte boundary give no penalty.

Aligning data by 8 or 16 on a dword size stack may be a problem. A common method is to set up an aligned frame pointer. A function with aligned local data may look like this:

```
_FuncWithAlign PROC NEAR
        PUSH    EBP                       ; prolog code
        MOV     EBP, ESP
        AND     EBP, -8                   ; align frame pointer by 8
        FLD     DWORD PTR [ESP+8]         ; function parameter
        SUB     ESP, LocalSpace + 4       ; allocate local space
        FSTP    QWORD PTR [EBP-LocalSpace] ; store something in aligned space
        ...
        ADD     ESP, LocalSpace + 4       ; epilog code. restore ESP
        POP     EBP                       ; (AGI stall on PPlain/PMMX)
        RET
_FuncWithAlign ENDP
```

While aligning data is always important, aligning code is not necessary on the PPlain and PMMX. Principles for aligning code on PPro, PII and PIII are explained in chapter 15.

7. Cache

The PPlain and PPro have 8 kb of on-chip cache (level one cache) for code, and 8 kb for data. The PMMX, PII and PIII have 16 kb for code and 16 kb for data. Data in the level 1 cache can be read or written to in just one clock cycle, whereas a cache miss may cost many clock cycles. It is therefore important that you understand how the cache works in order to use it most efficiently.

The data cache consists of 256 or 512 lines of 32 bytes each. Each time you read a data item which is not cached, the processor will read an entire cache line from memory. The cache lines are always aligned to a physical address divisible by 32. When you have read a byte at an address divisible by 32, then the next 31 bytes can be read or written to at almost no extra cost. You can take advantage of this by arranging data items which are used near each other together into aligned blocks of 32 bytes of memory. If, for example, you have a loop which accesses two arrays, then you may interleave the two arrays into one array of structures, so that data which are used together are also stored together.

If the size of an array or other data structure is a multiple of 32 bytes, then you should preferably align it by 32.

The cache is set-associative. This means that a cache line can not be assigned to an arbitrary memory address. Each cache line has a 7-bit set-value which must match bits 5 through 11 of the physical RAM address (bit 0-4 define the 32 bytes within a cache line). The PPlain and PPro have two cache lines for each of the 128 set-values, so there are two possible cache lines to assign to any RAM address. The PMMX, PII and PIII have four.

The consequence of this is that the cache can hold no more than two or four different data blocks which have the same value in bits 5-11 of the address. You can determine if two addresses have the same set-value by the following method: Strip off the lower 5 bits of each address to get a value divisible by 32. If the difference between the two truncated addresses is a multiple of 4096 (=1000H), then the addresses have the same set-value.

Let me illustrate this by the following piece of code, where ESI holds an address divisible by 32:

```
AGAIN:  MOV  EAX, [ESI]
        MOV  EBX, [ESI + 13*4096 +  4]
        MOV  ECX, [ESI + 20*4096 + 28]
        DEC  EDX
        JNZ  AGAIN
```

The three addresses used here all have the same set-value because the differences between the truncated addresses are multipla of 4096. This loop will perform very poorly on the PPlain and PPro. At the time you read `ECX` there is no free cache line with the proper set-value so the processor takes the least recently used of the two possible cache lines, that is the one which was used for `EAX`, and fills it with the data from `[ESI+20*4096]` to `[ESI+20*4096+31]` and reads `ECX`. Next, when reading `EAX`, you find that the cache line that held the value for `EAX` has now been discarded, so you take the least recently used line, which is the one holding the `EBX` value, and so on.. You have nothing but cache misses and the loop takes something like 60 clock cycles. If the third line is changed to:

```
        MOV  ECX, [ESI + 20*4096 + 32]
```

then we have crossed a 32 byte boundary, so that we do not have the same set-value as in the first two lines, and there will be no problem assigning a cache line to each of the three addresses. The loop now takes only 3 clock cycles (except for the first time) - a very considerable improvement! As already mentioned, the PMMX, PII and PIII have 4-way caches so that you have four cache lines with the same set-value. (Some Intel documents erroneously say that the PII cache is 2-way).

It may be very difficult to determine if your data addresses have the same set-values, especially if they are scattered around in different segments. The best thing you can do to avoid problems of this kind is to keep all data used in the critical part or your program within one contiguous block not bigger than the cache, or two contiguous blocks no bigger than half that size (for example one block for static data and another block for data on the stack). This will make sure that your cache lines are used optimally.

If the critical part of your code accesses big data structures or random data addresses, then you may want to keep all frequently used variables (counters, pointers, control variables, etc.) within a single contiguous block of max 4 kbytes so that you have a complete set of cache lines free for accessing random data. Since you probably need stack space anyway for subroutine parameters and return addresses, the best thing is to copy all frequently used static data to dynamic variables on the stack, and copy them back again outside the critical loop if they have been changed.

Reading a data item which is not in the level one cache causes an entire cache line to be filled from the level two cache, which takes approximately 200 ns (that is 20 clocks on a 100 MHz system or 40 clocks on a 200 MHz system), but the bytes you ask for first are available already after 50-100 ns. If the data item is not in the level two cache either, then you will get a delay of something like 200-300 ns. This delay will be somewhat longer if you cross a DRAM page boundary. (The size of a DRAM page is 1 kb for 4 and 8 MB 72 pin RAM modules, and 2 kb for 16 and 32 MB modules).

When reading big blocks of data from memory, the speed is limited by the time it takes to fill cache lines. You can sometimes improve speed by reading data in a non-sequential order: before you finish reading data from one cache line start reading the first item from the next cache line. This method can increase reading speed by 20 - 40% when reading from main memory or level 2 cache on PPlain and PMMX, and from level 2 cache on PPro, PII and PIII. A disadvantage of this method is of course that the program code becomes extremely clumsy and complicated. For further information on this trick see www.intelligentfirm.com.

When you write to an address which is not in the level 1 cache, then the value will go right through to the level 2 cache or to the RAM (depending on how the level 2 cache is set up) on the PPlain and PMMX. This takes approximately 100 ns. If you write eight or more times to the same 32 byte block of memory without also reading from it, and the block is not in the level one cache, then it may be advantageous to make a dummy read from the block first to load it into a cache line. All subsequent writes to the same block will then go to the cache instead, which takes only one clock cycle. On PPlain and PMMX, there is sometimes a small penalty for writing repeatedly to the same address without reading in between.

On PPro, PII and PIII, a write miss will normally load a cache line, but it is possible to setup an area of memory to perform differently, for example video RAM (See Pentium Pro Family Developer's Manual, vol. 3: Operating System Writer's Guide").

Other ways of speeding up memory reads and writes are discussed in chapter 27.8 below.

The PPlain and PPro have two write buffers, PMMX, PII and PIII have four. On the PMMX, PII and PIII you may have up to four unfinished writes to uncached memory without delaying the subsequent instructions. Each write buffer can handle operands up to 64 bits wide.

Temporary data may conveniently be stored on the stack because the stack area is very likely to be in the cache. However, you should be aware of the alignment problems if your data elements are bigger than the stack word size.

If the life ranges of two data structures do not overlap, then they may share the same RAM area to increase cache efficiency. This is consistent with the common practice of allocating space for temporary variables on the stack.

Storing temporary data in registers is of course even more efficient. Since registers is a scarce ressource you may want to use `[ESP]` rather than `[EBP]` for addressing data on the stack, in order to free `EBP` for other purposes. Just don't forget that the value of `ESP` changes every time you do a `PUSH` or `POP`. (You cannot use `ESP` under 16-bit Windows because the timer interrupt will modify the high word of `ESP` at unpredictable places in your code.)

There is a separate cache for code, which is similar to the data cache. The size of the code cache is 8 kb on PPlain and PPro and 16 kb on the PMMX, PII and PIII. It is important that the critical part of your code (the innermost loops) fit in the code cache. Frequently used pieces of code or routines which are used together should preferable be stored near each other. Seldom used branches or procedures should be put away in the bottom of your code or somewhere else.

8. First time versus repeated execution

A piece of code usually takes much more time the first time it is executed than when it is repeated. The reasons are the following:

1. Loading the code from RAM into the cache takes longer time than executing it.
2. Any data accessed by the code has to be loaded into the cache, which may take much more time than executing the instructions. When the code is repeated then the data are more likely to be in the cache.
3. Jump instructions will not be in the branch target buffer the first time they execute, and therefore are less likely to be predicted correctly. See chapter 22.
4. In the PPlain, decoding the code is a bottleneck. If it takes one clock cycle to determine the length of an instruction, then it is not possible to decode two instructions per clock cycle, because the processor doesn't know where the second instruction begins. The PPlain solves this problem by remembering the length of any instruction which has remained in the cache since last time it was executed. As a consequence of this, a set of instructions will not pair in the PPlain the first time they are executed, unless the first of the two instructions is only one byte long. The PMMX, PPro, PII and PIII have no penalty on first time decoding.

For these four reasons, a piece of code inside a loop will generally take much more time the first time it executes than the subsequent times.

If you have a big loop which doesn't fit into the code cache then you will get penalties all the time because it doesn't run from the cache. You should therefore try to reorganize the loop to make it fit into the cache.

If you have very many jumps, calls, and branches inside a loop, then you may get the penalty of branch target buffer misses repeatedly.

Likewise, if a loop repeatedly accesses a data structure too big for the data cache, then you will get the penalty of data cache misses all the time.

9. Address generation interlock (PPlain and PMMX)

It takes one clock cycle to calculate the address needed by an instruction which accesses memory. Normally, this calculation is done at a separate stage in the pipeline while the preceding instruction or instruction pair is executing. But if the address depends on the result of an instruction executing in the preceding clock cycle, then you have to wait an extra clock cycle for the address to be calculated. This is called an AGI stall. Example: `ADD EBX,4 / MOV EAX,[EBX] ; AGI stall`
The stall in this example can be removed by putting some other instructions in between `ADD EBX,4` and `MOV EAX,[EBX]` or by rewriting the code to: `MOV EAX,[EBX+4] / ADD EBX,4`

You can also get an AGI stall with instructions which use `ESP` implicitly for addressing, such as `PUSH, POP, CALL,` and `RET`, if `ESP` has been changed in the preceding clock cycle by instructions such as `MOV, ADD,` or `SUB`. The PPlain and PMMX have special circuitry to predict the value of `ESP` after a stack operation so that you do not get an AGI delay after changing `ESP` with `PUSH, POP,` or `CALL`. You can get an AGI stall after `RET` only if it has an immediate operand to add to `ESP`.

Examples:

```
ADD ESP,4 / POP ESI            ; AGI stall
POP EAX   / POP ESI            ; no stall, pair
MOV ESP,EBP / RET              ; AGI stall
CALL L1 / L1: MOV EAX,[ESP+8]  ; no stall
RET / POP EAX                  ; no stall
RET 8 / POP EAX                ; AGI stall
```

The `LEA` instruction is also subject to an AGI stall if it uses a base or index register which has been changed in the preceding clock cycle. Example:

```
INC ESI / LEA EAX,[EBX+4*ESI]  ; AGI stall
```

PPro, PII and PIII have no AGI stalls for memory reads and `LEA`, but they do have AGI stalls for memory writes. This is not very significant unless the subsequent code has to wait for the write to finish.

10. Pairing integer instructions (PPlain and PMMX)

10.1 Perfect pairing

The PPlain and PMMX have two pipelines for executing instructions, called the U-pipe and the V-pipe. Under certain conditions it is possible to execute two instructions simultaneously, one in the U-pipe and one in the V-pipe. This can almost double the speed. It is therefore advantageous to reorder your instructions to make them pair.

The following instructions are pairable in either pipe:

- `MOV` register, memory, or immediate into register or memory
- `PUSH` register or immediate, `POP` register
- `LEA, NOP`
- `INC, DEC, ADD, SUB, CMP, AND, OR, XOR,`
- and some forms of `TEST` (see chapter 26.14)

The following instructions are pairable in the U-pipe only:

- `ADC, SBB`
- `SHR, SAR, SHL, SAL` with immediate count
- `ROR, ROL, RCR, RCL` with an immediate count of 1

The following instructions can execute in either pipe but are only pairable when in the V-pipe:

- near call
- short and near jump
- short and near conditional jump.

All other integer instructions can execute in the U-pipe only, and are not pairable.

Two consecutive instructions will pair when the following conditions are met:

1. The first instruction is pairable in the U-pipe and the second instruction is pairable in the V-pipe.

2. The second instruction does not read or write a register which the first instruction writes to.
Examples:

```
MOV EAX, EBX / MOV ECX, EAX     ; read after write, do not pair
MOV EAX, 1   / MOV EAX, 2       ; write after write, do not pair
MOV EBX, EAX / MOV EAX, 2       ; write after read, pair OK
MOV EBX, EAX / MOV ECX, EAX     ; read after read, pair OK
MOV EBX, EAX / INC EAX          ; read and write after read, pair OK
```

3. In rule 2 partial registers are treated as full registers. Example:

```
MOV AL, BL  /  MOV AH, 0
```

writes to different parts of the same register, do not pair

4. Two instructions which both write to parts of the flags register can pair despite rule 2 and 3. Example:

```
SHR EAX, 4 / INC EBX            ; pair OK
```

5. An instruction which writes to the flags can pair with a conditional jump despite rule 2.
Example:

```
CMP EAX, 2 / JA LabelBigger     ; pair OK
```

6. The following instruction combinations can pair despite the fact that they both modify the stack pointer:

```
PUSH + PUSH,  PUSH + CALL,  POP + POP
```

7. There are restrictions on the pairing of instructions with prefix. There are several types of prefixes:

- instructions addressing a non-default segment have a segment prefix.
- instructions using 16 bit data in 32 bit mode, or 32 bit data in 16 bit mode have an operand size prefix.
- instructions using 32 bit base or index registers in 16 bit mode have an address size prefix.
- repeated string instructions have a repeat prefix.
- locked instructions have a `LOCK` prefix.
- many instructions which were not implemented on the 8086 processor have a two byte opcode where the first byte is `0FH`. The `0FH` byte behaves as a prefix on the PPlain, but not on the other versions. The most common instructions with `0FH` prefix are: `MOVZX, MOVSX, PUSH FS, POP FS, PUSH GS, POP GS, LFS, LGS, LSS, SETcc, BT, BTC, BTR, BTS, BSF, BSR, SHLD, SHRD,` and `IMUL` with two operands and no immediate operand.

On the PPlain, a prefixed instruction can only execute in the U-pipe, except for conditional near jumps.

On the PMMX, instructions with operand size, address size, or `0FH` prefix can execute in either pipe, whereas instructions with segment, repeat, or lock prefix can only execute in the U-pipe.

8. An instruction which has both a displacement and immediate data is not pairable on the PPlain and only pairable in the U-pipe on the PMMX:

```
MOV DWORD PTR DS:[1000], 0    ; not pairable or only in U-pipe
CMP BYTE PTR [EBX+8], 1       ; not pairable or only in U-pipe
CMP BYTE PTR [EBX], 1         ; pairable
CMP BYTE PTR [EBX+8], AL      ; pairable
```

(Another problem with instructions which have both a displacement and immediate data on the PMMX is that such instructions may be longer than 7 bytes, which means that only one instruction can be decoded per clock cycle, as explained in chapter 12.)

9. Both instructions must be preloaded and decoded. This is explained in chapter 8.

10. There are special pairing rules for MMX instructions on the PMMX:

- MMX shift, pack or unpack instructions can execute in either pipe but cannot pair with other MMX shift, pack or unpack instructions.
- MMX multiply instructions can execute in either pipe but cannot pair with other MMX multiply instructions. They take 3 clock cycles and the last 2 clock cycles can overlap

with subsequent instructions in the same way as floating point instructions can (see chapter 24).
- an MMX instruction which accesses memory or integer registers can execute only in the U-pipe and cannot pair with a non-MMX instruction.

10.2 Imperfect pairing

There are situations where the two instructions in a pair will not execute simultaneously, or only partially overlap in time. They should still be considered a pair, though, because the first instruction executes in the U-pipe, and the second in the V-pipe. No subsequent instruction can start to execute before both instructions in the imperfect pair have finished.

Imperfect pairing will happen in the following cases:

1. If the second instructions suffers an AGI stall (see chapter 9).

2. Two instructions cannot access the same DWORD of memory simultaneously. The following examples assume that `ESI` is divisible by 4:
`MOV AL, [ESI] / MOV BL, [ESI+1]`
The two operands are within the same DWORD, so they cannot execute simultaneously. The pair takes 2 clock cycles.
`MOV AL, [ESI+3] / MOV BL, [ESI+4]`
Here the two operands are on each side of a DWORD boundary, so they pair perfectly, and take only one clock cycle.

3. Rule 2 is extended to the case where bit 2-4 is the same in the two addresses (cache bank conflict). For DWORD addresses this means that the difference between the two addresses should not be divisible by 32. Examples:

```
MOV [ESI], EAX / MOV [ESI+32000], EBX ;  imperfect pairing
MOV [ESI], EAX / MOV [ESI+32004], EBX ;  perfect pairing
```

Pairable integer instructions which do not access memory take one clock cycle to execute, except for mispredicted jumps. `MOV` instructions to or from memory also take only one clock cycle if the data area is in the cache and properly aligned. There is no speed penalty for using complex addressing modes such as scaled index registers.

A pairable integer instruction which reads from memory, does some calculation, and stores the result in a register or flags, takes 2 clock cycles. (read/modify instructions).

A pairable integer instruction which reads from memory, does some calculation, and writes the result back to the memory, takes 3 clock cycles. (read/modify/write instructions).

4. If a read/modify/write instruction is paired with a read/modify or read/modify/write instruction, then they will pair imperfectly.

The number of clock cycles used is given in the following table:

First instruction	Second instruction		
	MOV or register only	read/modify	read/modify/write

MOV or register only	1	2	3
read/modify	2	2	3
read/modify/write	3	4	5

Example:
`ADD [mem1], EAX / ADD EBX, [mem2] ; 4 clock cycles`
`ADD EBX, [mem2] / ADD [mem1], EAX ; 3 clock cycles`

5. When two paired instructions both take extra time due to cache misses, misalignment, or jump misprediction, then the pair will take more time than each instruction, but less than the sum of the two.

6. A pairable floating point instruction followed by `FXCH` will make imperfect pairing if the next instruction is not a floating point instruction.

In order to avoid imperfect pairing you have to know which instructions go into the U-pipe, and which to the V-pipe. You can find out this by looking backwards in your code and search for instructions which are unpairable, pairable only in one of the pipes, or cannot pair due to one of the rules above.

Imperfect pairing can often be avoided by reordering your instructions. Example:

```
L1:     MOV     EAX,[ESI]
        MOV     EBX,[ESI]
        INC     ECX
```

Here the two `MOV` instructions form an imperfect pair because they both access the same memory location, and the sequence takes 3 clock cycles. You can improve the code by reordering the instructions so that `INC ECX` pairs with one of the `MOV` instructions.

```
L2:     MOV     EAX,OFFSET A
        XOR     EBX,EBX
        INC     EBX
        MOV     ECX,[EAX]
        JMP     L1
```

The pair `INC EBX / MOV ECX,[EAX]` is imperfect because the latter instruction has an AGI stall. The sequence takes 4 clocks. If you insert a `NOP` or any other instruction so that `MOV ECX,[EAX]` pairs with `JMP L1` instead, then the sequence takes only 3 clocks.

The next example is in 16 bit mode, assuming that `SP` is divisible by 4:

```
L3:     PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        CALL    FUNC
```

Here the `PUSH` instructions form two imperfect pairs, because both operands in each pair go into the same dword of memory. `PUSH BX` could possibly pair perfectly with `PUSH CX` (because they go on each side of a DWORD boundary) but it doesn't because it has already been paired with `PUSH AX`. The sequence therefore takes 5 clocks. If you insert a `NOP` or any other instruction so that `PUSH BX` pairs with `PUSH CX`, and `PUSH DX` with `CALL FUNC`, then the sequence will take only 3 clocks. Another way to solve the problem is to make sure that `SP` is not divisible by 4. Knowing whether `SP` is divisible by 4 or not in 16 bit mode can be difficult, so the best way to avoid this problem is to use 32 bit mode.

11. Splitting complex instructions into simpler ones (PPlain and PMMX)

You may split up read/modify and read/modify/write instructions to improve pairing.
Example:
`ADD [mem1],EAX / ADD [mem2],EBX ; 5 clock cycles`
This code may be split up into a sequence which takes only 3 clock cycles:

```
MOV ECX,[mem1] / MOV EDX,[mem2] / ADD ECX,EAX / ADD EDX,EBX
MOV [mem1],ECX / MOV [mem2],EDX
```

Likewise you may split up non-pairable instructions into pairable instructions:

```
PUSH [mem1]
PUSH [mem2]  ; non-pairable
```

Split up into:

```
MOV EAX,[mem1]
MOV EBX,[mem2]
PUSH EAX
PUSH EBX     ; everything pairs
```

Other examples of non-pairable instructions which may be split up into simpler pairable instructions:
`CDQ` split into: `MOV EDX,EAX / SAR EDX,31`
`NOT EAX` change to `XOR EAX,-1`
`NEG EAX` split into `XOR EAX,-1 / INC EAX`
`MOVZX EAX,BYTE PTR [mem]` split into `XOR EAX,EAX / MOV AL,BYTE PTR [mem]`
`JECXZ` split into `TEST ECX,ECX / JZ`
`LOOP` split into `DEC ECX / JNZ`
`XLAT` change to `MOV AL,[EBX+EAX]`

If splitting instructions doesn't improve speed, then you may keep the complex or nonpairable instructions in order to reduce code size.

Splitting instructions is not needed on the PPro, PII and PIII, except when the split instructions generate fewer uops.

12. Prefixes (PPlain and PMMX)

An instruction with one or more prefixes may not be able to execute in the V-pipe (se chapter 10, sect. 7), and it may take more than one clock cycle to decode.

On the PPlain, the decoding delay is one clock cycle for each prefix except for the `0FH` prefix of conditional near jumps.

The PMMX has no decoding delay for `0FH` prefix. Segment and repeat prefixes take one clock extra to decode. Address and operand size prefixes take two clocks extra to decode. The PMMX can decode two instructions per clock cycle if the first instruction has a segment or repeat prefix or no prefix, and the second instruction has no prefix. Instructions with address or operand size prefixes can only decode alone on the PMMX. Instructions with more than one prefix take one clock extra for each prefix.

Address size prefixes can be avoided by using 32 bit mode. Segment prefixes can be avoided in 32 bit mode by using a flat memory model. Operand size prefixes can be avoided in 32 bit mode by using only 8 bit and 32 bit integers.

Where prefixes are unavoidable, the decoding delay may be masked if a preceding instruction takes more than one clock cycle to execute. The rule for the PPlain is that any instruction which takes N clock cycles to execute (not to decode) can 'overshadow' the decoding delay of N-1 prefixes in the next two (sometimes three) instructions or instruction pairs. In other words, each extra clock cycle that an instruction takes to execute can be used to decode one prefix in a later instruction. This shadowing effect even extends across a predicted branch. Any instruction which takes more than one clock cycle to execute, and any instruction which is delayed because of an AGI stall, cache miss, misalignment, or any other reason except decoding delay and branch misprediction, has a shadowing effect.

The PMMX has a similar shadowing effect, but the mechanism is different. Decoded instructions are stored in a transparent first-in-first-out (FIFO) buffer, which can hold up to four instructions. As long as there are instructions in the FIFO buffer you get no delay. When the buffer is empty then instructions are executed as soon as they are decoded. The buffer is filled when instructions are decoded faster than they are executed, i.e. when you have unpaired or multi-cycle instructions. The FIFO buffer is emptied when instructions execute faster than they are decoded, i.e. when you have decoding delays due to prefixes. The FIFO buffer is empty after a mispredicted branch. The FIFO buffer can receive two instructions per clock cycle provided that the second instruction is without prefixes and none of the instructions are longer than 7 bytes. The two execution pipelines (U and V) can each receive one instruction per clock cycle from the FIFO buffer.

Examples:
`CLD / REP MOVSD`
The `CLD` instruction takes two clock cycles and can therefore overshadow the decoding delay of the `REP` prefix. The code would take one clock cycle more if the `CLD` instruction was placed far from the `REP MOVSD.`
`CMP DWORD PTR [EBX],0 / MOV EAX,0 / SETNZ AL`
The `CMP` instruction takes two clock cycles here because it is a read/modify instruction. The `0FH` prefix of the `SETNZ` instruction is decoded during the second clock cycle of the `CMP` instruction, so that the decoding delay is hidden on the PPlain (The PMMX has no decoding delay for the `0FH`).

Prefix penalties in PPro, PII and PIII are described in chapter 14.

13. Overview of PPro, PII and PIII pipeline

The architecture of the PPro, PII and PIII microprocessors is well explained and illustrated in various manuals and tutorials from Intel. It is recommended that you study this material in order to get an understanding of how these microprocessors work. I will describe the structure briefly here with particular focus on those elements that are important for optimizing code.

Instruction codes are fetched from the code cache in aligned 16-byte chunks into a double buffer that can hold two 16-byte chunks. The code is passed on from the double buffer to the decoders in blocks which I will call ifetch blocks (instruction fetch blocks). The ifetch blocks are usually 16 bytes long, but not aligned. The purpose of the double-buffer is to make it possible to decode an instruction that crosses a 16-byte boundary (i.e. an address divisible by 16).

The ifetch block goes to the instruction length decoder, which determines where each instruction begins and ends, and next to the instruction decoders. There are three decoders so that you can decode up to three instructions in each clock cycle. A group of up to three instructions that are decoded in the same clock cycle is called a decode group.

The decoders translate instructions into micro-operations, abbreviated uops. Simple instructions generate only one uop, while more complex instructions may generate several uops. For example, the instruction `ADD EAX,[MEM]` is decoded into two uops: one for reading the source operand from memory, and one for doing the addition. The purpose of splitting instructions into uops is to make the handling later in the system more effective.

The three decoders are called D0, D1, and D2. D0 can handle all instructions, while D1 and D2 can handle only simple instructions that generate one uop.

The uops from the decoders go via a short queue to the register allocation table (RAT). The execution of uops work on temporary registers which are later written to the permanent registers `EAX, EBX`, etc. The purpose of the RAT is to tell the uops which temporary registers to use, and to allow register renaming (see later).

After the RAT, the uops to go the reorder buffer (ROB). The purpose of the ROB is to enable out-of-order execution. A uop stays in the reservation station until the operands it needs are available. If an operand for one uop is delayed because a previous uop that generates the operand is not finished yet, then the ROB may find another uop later in the queue that can be executed in the meantime in order to save time.

The uops that are ready for execution are sent to the execution units, which are clustered around five ports: Port 0 and 1 can handle arithmetic operations, jumps, etc. Port 2 takes care of all reads from memory, port 3 calculates addresses for memory writes, and port 4 does memory writes.

When an instruction has been executed then it is marked in the ROB as ready to retire. It then goes to the retirement station. Here the contents of the temporary registers used by the uops are written to the permanent registers. While uops can be executed out of order, they must be retired in order.

In the following chapters, I will describe in detail how to optimize the throughput of each step in the pipeline.

14. Instruction decoding (PPro, PII and PIII)

I am describing instruction decoding before instruction fetching here because you need to know how the decoders work in order to understand the possible delays in instruction fetching.

The decoders can handle three instructions per clock cycle, but only when certain conditions are met. Decoder D0 can handle any instruction that generates up to 4 uops in a single clock cycle. Decoders D1 and D2 can handle only instructions that generate 1 uop and these instructions can be no more than 8 bytes long.

To summarize the rules for decoding two or three instructions in the same clock cycle:

- The first instruction (D0) generates no more than 4 uops,
- The second and third instructions generate no more than 1 uop each,
- The second and third instructions are no more than 8 bytes long each,
- The instructions must be contained within the same 16 bytes ifetch block (see next chapter).

There is no limit to the length of the instruction in D0 (despite Intel manuals saying something else), as long as the three instructions fit into one 16 bytes ifetch block.

An instruction that generates more than 4 uops takes two or more clock cycles to decode, and no other instructions can decode in parallel.

It follows from the rules above that the decoders can produce a maximum of 6 uops per clock cycle if the first instruction in each decode group generates 4 uops and the next two generate 1 uop each. The minimum production is 2 uops per clock cycle, which you get when all instructions generate 2 uops each, so that D1 and D2 are never used.

For maximum throughput, it is recommended that you order your instructions according to the 4-1-1 pattern: instructions that generate 2 to 4 uops can be interspearsed with two simple 1-uop instructions for free, in the sense that they do not add to the decoding time. Example:

```
MOV     EBX, [MEM1]     ; 1 uop  (D0)
INC     EBX             ; 1 uop  (D1)
ADD     EAX, [MEM2]     ; 2 uops (D0)
ADD     [MEM3], EAX     ; 4 uops (D0)
```

This takes 3 clock cycles to decode. You can save one clock cycle by reordering the instructions into two decode groups:

```
ADD     EAX, [MEM2]     ; 2 uops (D0)
MOV     EBX, [MEM1]     ; 1 uop  (D1)
INC     EBX             ; 1 uop  (D2)
ADD     [MEM3], EAX     ; 4 uops (D0)
```

The decoders now generate 8 uops in two clock cycles, which is probably satisfactory.

Later stages in the pipeline can handle only 3 uops per clock cycle so with a decoding rate higher than this you can assume that decoding is not a bottleneck. However, complications in the fetch mechanism can delay decoding as described in the next chapter, so to be safe you may want to aim at a decoding rate higher than 3 uops per clock cycle.

You can see how many uops each instruction generates in the tables in chapter 29.

Instruction prefixes can also incur penalties in the decoders. Instructions can have several kinds of prefixes:

- An operand size prefix is needed when you have a 16-bit operand in a 32-bit environment or vice versa. (Except for instructions that can only have one operand size, such as `FNSTSW AX`). An operand size prefix gives a penalty of a few clocks if the instruction has an immediate operand of 16 or 32 bits because the length of the operand is changed by the prefix. Examples:

  ```
  ADD BX, 9      ; no penalty because immediate operand is 8 bits
  MOV WORD PTR [MEM16], 9  ; penalty because operand is 16 bits
  ```

  The last instruction should be changed to:

  ```
  MOV EAX, 9
  MOV WORD PTR [MEM16], AX  ; no penalty because no immediate
  ```

- An address size prefix is used when you use 32-bit addressing in 16 bit mode or vice versa. This is seldom needed and should generally be avoided. The address size prefix gives a penalty whenever you have an explicit memory operand (even when there is no displacement) because the interpretation of the r/m bits in the instruction code is changed by the prefix. Instructions with only implicit memory operands, such as string instructions, have no penalty with address size prefix.
- Segment prefixes are used when you address data in a non-default data segment. Segment prefixes give no penalty on the PPro, PII and PIII.
- Repeat prefixes and lock prefixes give no penalty in the decoders.
- There is always a penalty if you have more than one prefix. This penalty is usually one clock per prefix.

15. Instruction fetch (PPro, PII and PIII)

The code is fetched in aligned 16-bytes chunks from the code cache and placed in the double buffer, which is called so because it can contain two such chunks. The code is then taken from the double buffer and fed to the decoders in blocks which are usually 16 bytes long, but not necessarily aligned by 16. I will call these blocks ifetch blocks (instruction fetch blocks). If an ifetch block crosses a 16 byte boundary in the code then it needs to take from both chunks in the double buffer. So the purpose of the double buffer is to allow instruction fetching across 16 byte boundaries.

The double buffer can fetch one 16-bytes chunk per clock cycle and can generate one ifetch block per clock cycle. The ifetch blocks are usually 16 bytes long, but can be shorter if there is a predicted jump in the block. (See chapter 22 about jump prediction).

Unfortunately, the double buffer is not big enough for handling fetches around jumps without delay. If the ifetch block that contains the jump instruction crosses a 16-byte boundary then the double buffer needs to keep two consecutive aligned 16-bytes chunks of code in order to generate it. If the first instruction after the jump crosses a 16-byte boundary, then the double buffer needs to load two new 16-bytes chunks of code before a valid ifetch block can be generated. This means that, in the worst case, the decoding of the first instruction after a jump can be delayed for two clock cycles. You get one penalty for a 16-byte boundary in the ifetch block containing the jump instruction, and one penalty for a 16-byte boundary in the first instruction after the jump. You can get bonus if you have more than one decode group in the ifetch block that contains the jump because this gives the double buffer extra time to fetch one or two 16-byte chunks of code in advance for the instructions after the jump. The bonuses can compensate for the penalties according to the table below. If the double buffer has fetched only one 16-byte chunk of code after the jump, then the first ifetch block after the jump will be identical to this chunk, that is, aligned to a 16-byte boundary. In other words, the first ifetch block after the jump will not begin at the first instruction, but at the nearest preceding address divisible by 16. If the double buffer has had time to load two 16-byte chunks, then the new ifetch block can cross a 16-byte boundary and begin at the first instruction after the jump. These rules are summarized in the following table:

Number of decode groups in ifetch-block containing jump	16-byte boundary in this ifetch-block	16-byte boundary in first instruction after jump	decoder delay	alignment of first ifetch after jump
1	0	0	0	by 16
1	0	1	1	to instruction
1	1	0	1	by 16
1	1	1	2	to instruction
2	0	0	0	to instruction
2	0	1	0	to instruction
2	1	0	0	by 16
2	1	1	1	to instruction
3 or more	0	0	0	to instruction
3 or more	0	1	0	to instruction
3 or more	1	0	0	to instruction
3 or more	1	1	0	to instruction

Jumps delay the fetching so that a loop always takes at least two clock cycles more per iteration than the number of 16 byte boundaries in the loop.

A further problem with the instruction fetch mechanism is that a new ifetch block is not generated until the previous one is exhausted. Each ifetch block can contain several decode groups. If a 16 bytes long ifetch block ends with an unfinished instruction, then the next ifetch block will begin at the beginning of that instruction. The first instruction in an ifetch block always goes to decoder D0, and the next two instructions go to D1 and D2, if possible. The consequence of this is that D1 and D2 are used less than optimally. If the code is structured according to the recommended 4-1-1 pattern, and an instruction intended to go into D1 or D2 happens to be the first instruction in an ifetch block, then that instruction has to go into D0 with the result that one clock cycle is wasted. This is probably a hardware design flaw. At least it is suboptimal design. The consequence of this problem is that the time it takes to decode a piece of code can vary considerably depending on where the first ifetch block begins.

If decoding speed is critical and you want to avoid these problems then you have to know where each ifetch block begins. This is quite a tedious job. First you need to make your code segment paragraph-aligned in order to know where the 16-byte boundaries are. Then you have to look at the output listing from your assembler to see how long each instruction is. (It is recommended that you study how instructions are coded so that you can predict the lengths of the instructions.) If you know where one ifetch block begins then you can find where the next ifetch block begins in the following way: Make the block 16 bytes long. If it ends at an instruction boundary then the next block will begin there. If it ends with an unfinished instruction then the next block will begin at the beginning of this instruction. (Only the lengths of the instructions counts here, it doesn't matter how many uops they generate or what they do). This way you can work your way all through the code and mark where each ifetch block begins. The only problem is knowing where to start. If you know where one ifetch block is then you can find all the subsequent ones, but you have to know where the first one begins. Here are some guidelines:

- The first ifetch block after a jump, call, or return can begin either at the first instruction or at the nearest preceding 16-bytes boundary, according to the table above. If you align the first instruction to begin at a 16-byte boundary then you can be sure that the first ifetch block begins here. You may want to align important subroutine entries and loop entries by 16 for this purpose.
- If the combined length of two consecutive instructions is more than 16 bytes then you can be certain that the second one doesn't fit into the same ifetch block as the first one, and consequently you will always have an ifetch block beginning at the second instruction. You can use this as a starting point for finding where subsequent ifetch blocks begin.
- The first ifetch block after a branch misprediction begins at a 16-byte boundary. As explained in chapter 22.2, a loop that repeats more than 5 times will always have a misprediction when it exits. The first ifetch block after such a loop will therefore begin at the nearest preceding 16-byte boundary.
- Other serializing events also cause the next ifetch block to start at a 16-byte boundary. Such events include interrupts, exceptions, self-modifying code, and

serializing instructions such as `CPUID, IN,` and `OUT`.

I am sure you want an example now:

```
address      instruction            length    uops    expected decoder
---------------------------------------------------------------------
1000h        MOV ECX, 1000            5         1       D0
1005h   LL:  MOV [ESI], EAX           2         2       D0
1007h        MOV [MEM], 0            10         2       D0
1011h        LEA EBX, [EAX+200]       6         1       D1
1017h        MOV BYTE PTR [ESI], 0    3         2       D0
101Ah        BSR EDX, EAX             3         2       D0
101Dh        MOV BYTE PTR [ESI+1],0   4         2       D0
1021h        DEC ECX                  1         1       D1
1022h        JNZ LL                   2         1       D2
```

Let's assume that the first ifetch block begins at address 1000h and ends at 1010h. This is before the end of the `MOV [MEM],0` instruction so the next ifetch block will begin at 1007h and end at 1017h. This is at an instruction boundary so the third ifetch block will begin at 1017h and cover the rest of the loop. The number of clock cycles it takes to decode this is the number of D0 instructions, which is 5 per iteration of the LL loop. The last ifetch block contained three decode blocks covering the last five instructions, and it has one 16-byte boundary (1020h). Looking at the table above we find that the first ifetch block after the jump will begin at the first instruction after the jump, that is the `LL` label at 1005h, and end at 1015h. This is before the end of the `LEA` instruction, so the next ifetch block will go from 1011h to 1021h, and the last one from 1021h covering the rest. Now the `LEA` instruction and the `DEC` instruction both fall at the beginning of an ifetch block which forces them to go into D0. We now have 7 instructions in D0 and the loop takes 7 clocks to decode in the second iteration. The last ifetch block contains only one decode group (`DEC ECX / JNZ LL`) and has no 16-byte boundary. According to the table, the next ifetch block after the jump will begin at a 16-byte boundary, which is 1000h. This will give us the same situation as in the first iteration, and you will see that the loop takes alternatingly 5 and 7 clock cycles to decode. Since there are no other bottlenecks, the complete loop will take 6000 clocks to run 1000 iterations. If the starting address had been different so that you had a 16-byte boundary in the first or the last instruction of the loop then it would take 8000 clocks. If you reorder the loop so that no D1 or D2 instructions fall at the beginning of an ifetch block then you can make it take only 5000 clocks.

The example above was deliberately constructed so that fetch and decoding is the only bottleneck. The easiest way to avoid this problem is to structure your code to generate much more than 3 uops per clock cycle so that decoding will not be a bottleneck despite the penalties described here. In small loops this may not be possible and then you have to find out how to optimize the instruction fetch and decoding.

One thing you can do is to change the starting address of your procedure in order to avoid 16-byte boundaries where you don't want them. Remember to make your code segment paragraph aligned so that you know where the boundaries are.

If you insert an `ALIGN 16` directive before the loop entry then the assembler will put in `NOP`'s and other filler instructions up to the nearest 16 byte boundary. Most assemblers use the instruction `XCHG EBX,EBX` as a 2-byte filler (the so called 2-byte `NOP`). Whoever got this idea, it's a bad one because this instruction takes more time than two `NOP`'s on most processors! If the loop executes many times then whatever is outside the loop is unimportant in terms of speed and you don't have to care about the suboptimal filler instructions. But if the time taken by the fillers is important then you may select the filler instructions manually. You may as well use filler instructions that do something useful, such as refreshing a register in order to avoid register read stalls (see chapter 16.2) For example, if you are using register `EBP` for addressing but seldom write to it, then you may use `MOV EBP,EBP` or `ADD EBP, 0` as filler in order to reduce the possibilities of register read stalls. If you have nothing useful to do, you may use `FXCH ST(0)` as a good filler because it doesn't put any load on the execution ports, provided that `ST(0)` contains a valid floating point value.

Another possible remedy is to reorder your instructions in order to get the ifetch boundaries where they don't hurt. This can be quite a difficult puzzle and it is not always possible to find a satisfactory solution.

Yet another possibility is to manipulate instruction lengths. Sometimes you can substitute one instruction with another one with a different length. Many instructions can be coded in different versions with different lengths. The assembler always chooses the shortest possible version of an instruction, but it is often possible to hard-code a longer version. For example, `DEC ECX` is one byte long, `SUB ECX,1` is 3 bytes, and you can code a 6 bytes version with a long immediate operand using this trick:

```
SUB ECX, 9999
ORG $-4
DD 1
```

Instructions with a memory operand can be made one byte longer with a SIB byte, but the easiest way of making an instruction one byte longer is to add a `DS:` segment prefix (`DB 3Eh`). The microprocessors generally accept redundant and meaningless prefixes (except `LOCK`) as long as the instruction length does not exceed 15 bytes. Even instructions without a memory operand can have a segment prefix. So if you want the `DEC ECX` instruction to be 2 bytes long, write:

```
DB  3Eh
DEC ECX
```

Remember that you get a penalty in the decoder if an instruction has more than one prefix. It is possible that instructions with meaningless prefixes - especially repeat and lock prefixes - will be used in future processors for new instructions when there are no more vacant instruction codes, but I would consider it safe to use a segment prefix with any instruction.

With these methods it will usually be possible to put the ifetch boundaries where you want them, although it can be a tedious puzzle.

16. Register renaming (PPro, PII and PIII)

16.1 Eliminating dependencies

Register renaming is an advanced technique used by these microprocessors to remove dependencies between different parts of the code. Example:

```
MOV EAX, [MEM1]
IMUL EAX, 6
MOV [MEM2], EAX
MOV EAX, [MEM3]
INC EAX
MOV [MEM4], EAX
```

Here the last three instructions are independent of the first three in the sense that they don't need any result from the first three instructions. To optimize this on earlier processors you would have to use a different register instead of `EAX` in the last three instructions and reorder the instructions so that the last three instructions could execute in parallel with the first three instructions. The PPro, PII and PIII processors do this for you automatically. They assign a new temporary register for `EAX` every time you write to it. Thereby the `MOV EAX,[MEM3]` instruction becomes independent of the preceding instructions. With out-of-order execution it is likely to finish the move to `[MEM4]` before the slow `IMUL` instruction is finished.

Register renaming goes fully automatically. A new temporary register is assigned as an alias for the permanent register every time an instruction writes to this register. An instruction that both reads and writes a register also causes renaming. For example the `INC EAX` instruction above uses one temporary register for input and another temporary register for output. This does not remove any dependency, of course, but it has some significance for subsequent register reads as I will explain later.

All general purpose registers, stack pointer, flags, floating point registers, MMX registers, XMM registers and segment registers can be renamed. Control words, and the floating point status word cannot be renamed and this is the reason why the use of these registers is slow. There are 40 universal temporary registers so it is unlikely that you will run out of temporary registers.

A common way of setting a register to zero is `XOR EAX,EAX` or `SUB EAX,EAX`. These instructions are not recognized as independent of the previous value of the register. If you want to remove the dependency on slow preceding instructions then use `MOV EAX,0.`

Register renaming is controlled by the register alias table (RAT) and the reorder buffer (ROB). The uops from the decoders go to the RAT via a queue, and then to the ROB and the reservation station. The RAT can handle only 3 uops per clock cycle. This means that the overall throughput of the microprocessor can never exceed 3 uops per clock cycle on average.

There is no practical limit to the number of renamings. The RAT can rename three registers per clock cycle, and it can even rename the same register three times in one clock cycle.

16.2 Register read stalls

But there is another limitation which may be quite serious, and that is that you can only read two different permanent register names per clock cycle. This limitation applies to all registers used by an instruction except those registers that the instruction writes to only. Example:

```
MOV [EDI + ESI], EAX
MOV EBX, [ESP + EBP]
```

The first instruction generates two uops: one that reads `EAX` and one that reads `EDI` and `ESI`. The second instruction generates one uop that reads `ESP` and `EBP`. `EBX` does not count as a read because it is only written to by the instruction. Let's assume that these three uops go through the RAT together. I will use the word triplet for a group of three consecutive uops that go through the RAT together. Since the ROB can handle only two permanent register reads per clock cycle and we need five register reads, our triplet will be delayed for two extra clock cycles before it comes to the reservation station. With 3 or 4 register reads in the triplet it would be delayed by one clock cycle.

The same register can be read more than once in the same triplet without adding to the count. If the instructions above are changed to:

```
MOV [EDI + ESI], EDI
MOV EBX, [EDI + EDI]
```

then you will need only two register reads (`EDI` and `ESI`) and the triplet will not be delayed.

A register that is going to be written to by a pending uop is stored in the ROB so that it can be read for free until it is written back, which takes at least 3 clock cycles, and usually more. Write-back is the end of the execution stage where the value becomes available. In other words, you can read any number of registers in the RAT without stall if their values are not yet available from the execution units. The reason for this is that when a value becomes available it is immediately written directly to any subsequent ROB entries that need it. But if the value has already been written back to a temporary or permanent register when a subsequent uop that needs it goes into the RAT, then the value has to be read from the register file, which has only two read ports. There are three pipeline stages from the RAT to the execution unit so you can be certain that a register written to in one uop-triplet can be read for free in at least the next three triplets. If the writeback is delayed by reordering, slow instructions, dependency chains, cache misses, or by any other kind of stall, then the register can be read for free further down the instruction stream.

Example:

```
MOV EAX, EBX
SUB ECX, EAX
INC EBX
MOV EDX, [EAX]
ADD ESI, EBX
ADD EDI, ECX
```

These 6 instructions generate 1 uop each. Let's assume that the first 3 uops go through the

RAT together. These 3 uops read register `EBX`, `ECX`, and `EAX`. But since we are writing to `EAX` before reading it, the read is free and we get no stall. The next three uops read `EAX`, `ESI`, `EBX`, `EDI`, and `ECX`. Since both `EAX`, `EBX` and `ECX` have been modified in the preceding triplet and not yet written back then they can be read for free, so that only `ESI` and `EDI` count, and we get no stall in the second triplet either. If the `SUB ECX,EAX` instruction in the first triplet is changed to `CMP ECX,EAX` then `ECX` is not written to and we will get a stall in the second triplet for reading `ESI`, `EDI` and `ECX`. Similarly, if the `INC EBX` instruction in the first triplet is changed to `NOP` or something else then we will get a stall in the second triplet for reading `ESI`, `EBX` and `EDI`.

No uop can read more than two registers. Therefore, all instructions that need to read more than two registers are split up into two or more uops.

To count the number of register reads, you have to include all registers which are read by the instruction. This includes integer registers, the flags register, the stack pointer, floating point registers and MMX registers. An XMM register counts as two registers, except when only part of it is used, as e.g. in `ADDSS` and `MOVHLPS`. Segment registers and the instruction pointer do not count. For example, in `SETZ AL` you count the flags register but not `AL`. `ADD EBX,ECX` counts both `EBX` and `ECX`, but not the flags because they are written to only. `PUSH EAX` reads `EAX` and the stack pointer and then writes to the stack pointer.

The `FXCH` instruction is a special case. It works by renaming, but doesn't read any values so that it doesn't count in the rules for register read stalls. An `FXCH` instruction behaves like 1 uop that neither reads nor writes any registers with regard to the rules for register read stalls.

Don't confuse uop triplets with decode groups. A decode group can generate from 1 to 6 uops, and even if the decode group has three instructions and generates three uops there is no guarantee that the three uops will go into the RAT together.

The queue between the decoders and the RAT is so short (10 uops) that you cannot assume that register read stalls do not stall the decoders or that fluctuations in decoder throughput do not stall the RAT.

It is very difficult to predict which uops go through the RAT together unless the queue is empty, and for optimized code the queue should be empty only after mispredicted branches. Several uops generated by the same instruction do not necessarily go through the RAT together; the uops are simply taken consecutively from the queue, three at at time. The sequence is not broken by a predicted jump: uops before and after the jump can go through the RAT together. Only a mispredicted jump will discard the queue and start over again so that the next three uops are sure to go into the RAT together.

If three consecutive uops read more than two different registers then you would of course prefer that they do not go through the RAT together. The probability that they do is one third. The penalty of reading three or four written-back registers in one triplet of uops is one clock cycle. You can think of the one clock delay as equivalent to the load of three more uops through the RAT. With the probability of 1/3 of the three uops going into the RAT together, the average penalty will be the equivalent of 3/3 = 1 uop. To calculate the average time it will take for a piece of code to go through the RAT, add the number of potential register read stalls to the number of uops and divide by three. You can see that It doesn't pay to remove the stall by putting in an extra instruction unless you know for sure which uops go into the RAT together or you can prevent more than one potential register read stall by one extra instruction.

In situations where you aim at a throughput of 3 uops per clock, the limit of two permanent register reads per clock cycle may be a problematic bottleneck to handle. Possible ways to remove register read stalls are:

- keep uops that read the same register close together so that they are likely to go into the same triplet.
- keep uops that read different registers spaced so that they cannot go into the same triplet.
- place uops that read a register no more than 3 - 4 triplets after an instruction that writes to or modifies this register to make sure it hasn't been written back before it is read (it doesn't matter if you have a jump between as long as it is predicted). If you have reason to expect the register write to be delayed for whatever reason then you can safely read the register somewhat further down the instruction stream.
- use absolute addresses instead of pointers in order to reduce the number of register reads.
- you may rename a register in a triplet where it doesn't cause a stall in order to prevent a read stall for this register in one or more later triplets. Example: `MOV ESP,ESP / ... / MOV EAX,[ESP+8]`. This method costs an extra uop and therefore doesn't pay unless the expected average number of read stalls prevented is more than 1/3.

For instructions that generate more than one uop you may want to know the order of the uops generated by the instruction in order to make a precise analysis of the possibility of register read stalls. I have therefore listed the most common cases below.

Writes to memory
A memory write generates two uops. The first one (to port 4) is a store operation, reading the register to store. The second uop (port 3) calculates the memory address, reading any pointer registers. Examples:
`MOV [EDI], EAX`
First uop reads `EAX`, second uop reads `EDI`.
`FSTP QWORD PTR [EBX+8*ECX]`
First uop reads `ST(0)`, second uop reads `EBX` and `ECX.`

Read and modify
An instruction that reads a memory operand and modifies a register by some arithmetic or logical operation generates two uops. The first one (port 2) is a memory load instruction reading any pointer registers, the second uop is an arithmetic instruction (port 0 or 1) reading and writing to the destination register and possibly writing to the flags. Example:
`ADD EAX, [ESI+20]`
First uop reads `ESI,` second uop reads `EAX` and writes `EAX` and flags.

Read/modify/write

A read/modify/write instruction generates four uops. The first uop (port 2) reads any pointer registers, the second uop (port 0 or 1) reads and writes to any source register and possibly writes to the flags, the third uop (port 4) reads only the temporary result which doesn't count here, the fourth uop (port 3) reads any pointer registers again. Since the first and the fourth uop cannot go into the RAT together you cannot take advantage of the fact that they read the same pointer registers. Example:
`OR [ESI+EDI], EAX`
The first uop reads `ESI` and `EDI`, the second uop reads `EAX` and writes `EAX` and the flags, the third uop reads only the temporary result, the fourth uop reads `ESI` and `EDI` again. No matter how these uops go into the RAT you can be sure that the uop that reads `EAX` goes together with one of the uops that read `ESI` and `EDI`. A register read stall is therefore inevitable for this instruction unless one of the registers has been modified recently.

Push register
A push register instruction generates 3 uops. The first one (port 4) is a store instruction, reading the register. The second uop (port 3) generates the address, reading the stack pointer. The third uop (port 0 or 1) subtracts the word size from the stack pointer, reading and modifying the stack pointer.

Pop register
A pop register instruction generates 2 uops. The first uop (port 2) loads the value, reading the stack pointer and writing to the register. The second uop (port 0 or 1) adjusts the stack pointer, reading and modifying the stack pointer.

Call
A near call generates 4 uops (port 1, 4, 3, 01). The first two uops read only the instruction pointer which doesn't count because it cannot be renamed. The third uop reads the stack pointer. The last uop reads and modifies the stack pointer.

Return
A near return generates 4 uops (port 2, 01, 01, 1). The first uop reads the stack pointer. The third uop reads and modifies the stack pointer.

An example of how to avoid a register read stall is given in example 2.6.

17. Out of order execution (PPro, PII and PIII)

The reorder buffer (ROB) can hold 40 uops. Each uop waits in the ROB until all its operands are ready and there is a vacant execution unit for it. This makes out-of-order execution possible. If one part of the code is delayed because of a cache miss then it won't delay later parts of the code if they are independent of the delayed operations.

Writes to memory cannot execute out of order relative to other writes. There are four write buffers, so if you expect many cache misses on writes or you are writing to uncached memory then it is recommended that you schedule four writes at at time and make sure the processor has something else to do before you give it the next four writes. Memory reads and other instructions can execute out of order, except `IN, OUT` and serializing instructions.

If your code writes to a memory address and soon after reads from the same address, then the read may by mistake be executed before the write because the ROB doesn't know the memory addresses at the time of reordering. This error is detected when the write address is calculated, and then the read operation (which was executed speculatively) has to be re-done. The penalty for this is approximately 3 clocks. The only way to avoid this penalty is to make sure the execution unit has other things to do between a write and a subsequent read from the same memory address.

There are several execution units clustered around five ports. Port 0 and 1 are for arithmetic operations etc. Simple move, arithmetic and logic operations can go to either port 0 or 1, whichever is vacant first. Port 0 also handles multiplication, division, integer shifts and rotates, and floating point operations. Port 1 also handles jumps and some MMX and XMM operations. Port 2 handles all reads from memory and a few string and XMM operations, port 3 calculates addresses for memory write, and port 4 executes all memory write operations. In chapter 29 you'll find a complete list of the uops generated by code instructions with an indication of which ports they go to. Note that all memory write operations require two uops, one for port 3 and one for port 4, while memory read operations use only one uop (port 2).

In most cases each port can receive one new uop per clock cycle. This means that you can execute up to 5 uops in the same clock cycle if they go to five different ports, but since there is a limit of 3 uops per clock earlier in the pipeline you will never execute more than 3 uops per clock on average.

You must make sure that no execution port receives more than one third of the uops if you want to maintain a throughput of 3 uops per clock. Use the table of uops in chapter 29 and count how many uops go to each port. If port 0 and 1 are saturated while port 2 is free then you can improve your code by replacing some `MOV register,register` or `MOV register,immediate` instructions with `MOV register,memory` in order to move some of the load from port 0 and 1 to port 2.

Most uops take only one clock cycle to execute, but multiplications, divisions, and many floating point operations take more:

Floating point addition and subtraction takes 3 clocks, but the execution unit is fully pipelined so that it can receive a new `FADD` or `FSUB` in every clock cycle before the preceding ones are finished (provided, of course, that they are independent).

Integer multiplication takes 4 clocks, floating point multiplication 5, and MMX multiplication 3 clocks. Integer and MMX multiplication is pipelined so that it can receive a new instruction every clock cycle. Floating point multiplication is partially pipelined: The execution unit can receive a new `FMUL` instruction two clocks after the preceding one, so that the maximum throughput is one `FMUL` per two clock cycles. The holes between the `FMUL`'s cannot be filled by integer multiplications because they use the same circuitry. XMM additions and multiplications take 3 and 4 clocks respectively, and are fully pipelined. But since each logical XMM register is implemented as two physical 64-bit registers, you need two uops for a packed XMM operation, and the throughput will then be one arithmetic XMM instruction every two clock cycles. XMM add and multiply instructions can execute in parallel because they don't use the same execution port.

Integer and floating point division takes up to 39 clocks and is not pipelined. This means that the execution unit cannot begin a new division until the previous division is finished. The same applies to squareroot and transcendental functions.

Also jump instructions, calls, and returns are not fully pipelined. You cannot execute a new jump in the first clock cycle after a preceding jump. So the maximum throughput for jumps, calls, and returns is one for every two clocks.

You should, of course, avoid instructions that generate many uops. The `LOOP XX` instruction, for example, should be replaced by `DEC ECX / JNZ XX`.

If you have consecutive `POP` instructions then you may break them up to reduce the number of uops:

```
POP ECX / POP EBX / POP EAX     ; can be changed to:
MOV ECX,[ESP] / MOV EBX,[ESP+4] / MOV EAX,[ESP] / ADD ESP,12
```

The former code generates 6 uops, the latter generates only 4 and decodes faster. Doing the same with `PUSH` instructions is less advantageous because the split-up code is likely to generate register read stalls unless you have other instructions to put in between or the registers have been renamed recently. Doing it with `CALL` and `RET` instructions will interfere with prediction in the return stack buffer. Note also that the `ADD ESP` instruction can cause an AGI stall in earlier processors.

18. Retirement (PPro, PII and PIII)

Retirement is a process where the temporary registers used by the uops are copied into the permanent registers `EAX, EBX`, etc. When a uop has been executed it is marked in the ROB as ready to retire.

The retirement station can handle three uops per clock cycle. This may not seem like a problem because the throughput is already limited to 3 uops per clock in the RAT. But retirement may still be a bottleneck for two reasons. Firstly, instructions must retire in order. If a uop is executed out of order then it cannot retire before all preceding uops in the order have retired. And the second limitation is that taken jumps must retire in the first of the three slots in the retirement station. Just like decoder D1 and D2 can be idle if the next instruction only fits into D0, the last two slots in the retirement station can be idle if the next uop to retire is a taken jump. This is significant if you have a small loop where the number of uops in the loop is not divisible by three.

All uops stay in the reorder buffer (ROB) until they retire. The ROB can hold 40 uops. This sets a limit to the number of instructions that can execute during the long delay of a division or other slow operation. Before the division is finished the ROB will be filled up with executed uops waiting to retire. Only when the division is finished and retired can the subsequent uops begin to retire, because retirement takes place in order.

In case of speculative execution of predicted branches (see chapter 22) the speculatively executed uops cannot retire until it is certain that the prediction was correct. If the prediction turns out to be wrong then the speculatively executed uops are discarded without retirement.

The following instructions cannot execute speculatively: memory writes, `IN, OUT`, and serializing instructions.

19. Partial stalls (PPro, PII and PIII)

19.1 Partial register stalls

Partial register stall is a problem that occurs when you write to part of a 32 bit register and later read from the whole register or a bigger part of it. Example:

```
MOV AL, BYTE PTR [M8]
MOV EBX, EAX            ; partial register stall
```

This gives a delay of 5-6 clocks. The reason is that a temporary register has been assigned to `AL` (to make it independent of `AH`). The execution unit has to wait until the write to `AL` has retired before it is possible to combine the value from `AL` with the value of the rest of `EAX`. The stall can be avoided by changing to code to:

```
MOVZX EBX, BYTE PTR [MEM8]
AND EAX, 0FFFFFF00h
OR EBX, EAX
```

Of course you can also avoid the partial stalls by putting in other instructions after the write to the partial register so that it has time to retire before you read from the full register.

You should be aware of partial stalls whenever you mix different data sizes (8, 16, and 32 bits):

```
MOV BH, 0
ADD BX, AX              ; stall
INC EBX                 ; stall
```

You don't get a stall when reading a partial register after writing to the full register, or a bigger part of it:

```
MOV EAX, [MEM32]
ADD BL, AL              ; no stall
ADD BH, AH              ; no stall
MOV CX, AX              ; no stall
MOV DX, BX              ; stall
```

The easiest way to avoid partial register stalls is to always use full registers and use `MOVZX` or `MOVSX` when reading from smaller memory operands. These instructions are fast on the PPro, PII and PIII, but slow on earlier processors. Therefore, a compromise is offered when you want your code to perform reasonably well on all processors. The replacement for `MOVZX EAX,BYTE PTR [M8]` looks like this:

```
XOR EAX, EAX
MOV AL, BYTE PTR [M8]
```

The PPro, PII and PIII processors make a special case out of this combination to avoid a partial register stall when later reading from `EAX`. The trick is that a register is tagged as empty when it is `XOR`'ed with itself. The processor remembers that the upper 24 bits of `EAX` are zero, so that a partial stall can be avoided. This mechanism works only on certain combinations:

```
        XOR EAX, EAX
        MOV AL, 3
        MOV EBX, EAX            ; no stall

        XOR AH, AH
        MOV AL, 3
        MOV BX, AX              ; no stall

        XOR EAX, EAX
        MOV AH, 3
        MOV EBX, EAX            ; stall

        SUB EBX, EBX
        MOV BL, DL
        MOV ECX, EBX            ; no stall

        MOV EBX, 0
        MOV BL, DL
        MOV ECX, EBX            ; stall

        MOV BL, DL
        XOR EBX, EBX            ; no stall
```

Setting a register to zero by subtracting it from itself works the same as the `XOR`, but setting it to zero with the `MOV` instruction doesn't prevent the stall.

You can set the `XOR` outside a loop:

```
        XOR EAX, EAX
        MOV ECX, 100
LL:     MOV AL, [ESI]
        MOV [EDI], EAX          ; no stall
        INC ESI
        ADD EDI, 4
        DEC ECX
        JNZ LL
```

The processor remembers that the upper 24 bits of `EAX` are zero as long as you don't get an interrupt, misprediction, or other serializing event.

You should remember to neutralize any partial register you have used before calling a subroutine that might push the full register:

```
        ADD BL, AL
        MOV [MEM8], BL
        XOR EBX, EBX            ; neutralize BL
        CALL _HighLevelFunction
```

Most high level language procedures push `EBX` at the start of the procedure which would generate a partial register stall in the example above if you hadn't neutralized `BL`.

Setting a register to zero with the `XOR` method doesn't break its dependency on earlier instructions:

```
        DIV EBX
        MOV [MEM], EAX
        MOV EAX, 0              ; break dependency
        XOR EAX, EAX            ; prevent partial register stall
        MOV AL, CL
        ADD EBX, EAX
```

Setting `EAX` to zero twice here seems redundant, but without the `MOV EAX,0` the last instructions would have to wait for the slow `DIV` to finish, and without `XOR EAX,EAX` you would have a partial register stall.

The `FNSTSW AX` instruction is special: in 32 bit mode it behaves as if writing to the entire `EAX`. In fact, it does something like this in 32 bit mode:
`AND EAX,0FFFF0000h / FNSTSW TEMP / OR EAX,TEMP`
hence, you don't get a partial register stall when reading `EAX` after this instruction in 32 bit mode:

```
    FNSTSW AX / MOV EBX,EAX         ; stall only if 16 bit mode
    MOV AX,0  / FNSTSW AX           ; stall only if 32 bit mode
```

19.2 Partial flags stalls

The flags register can also cause partial register stalls:

```
        CMP EAX, EBX
        INC ECX
        JBE XX          ; partial flags stall
```

The `JBE` instruction reads both the carry flag and the zero flag. Since the `INC` instruction changes the zero flag, but not the carry flag, the `JBE` instruction has to wait for the two preceding instructions to retire before it can combine the carry flag from the `CMP` instruction and the zero flag from the `INC` instruction. This situation is likely to be a bug rather than an intended combination of flags. To correct it change `INC ECX` to `ADD ECX,1`. A similar bug that causes a partial flags stall is `SAHF / JL XX`. The `JL` instruction tests the sign flag and the overflow flag, but `SAHF` doesn't change the overflow flag. To correct it, change `JL XX` to `JS XX`.

Unexpectedly (and contrary to what Intel manuals say) you also get a partial flags stall after an instruction that modifies some of the flag bits when reading only unmodified flag bits:

```
        CMP EAX, EBX
        INC ECX
        JC  XX          ; partial flags stall
```

but not when reading only modified bits:

```
        CMP EAX, EBX
        INC ECX
        JE  XX          ; no stall
```

Partial flags stalls are likely to occur on instructions that read many or all flags bits, i.e. `LAHF, PUSHF, PUSHFD`. The following instructions cause partial flags stalls when followed by `LAHF` or `PUSHF(D)`: `INC, DEC, TEST`, bit tests, bit scan, `CLC, STC, CMC, CLD, STD, CLI, STI, MUL, IMUL`, and all shifts and rotates. The following instructions do not cause partial flags stalls: `AND, OR, XOR, ADD, ADC, SUB, SBB, CMP, NEG`. It is strange that `TEST` and `AND` behave differently while, by definition, they do exactly the same thing to the flags. You may use a `SETcc` instruction instead of `LAHF` or `PUSHF(D)` for storing the value of a flag in order to avoid a stall.

Examples:

```
    INC EAX   / PUSHFD      ; stall
    ADD EAX,1 / PUSHFD      ; no stall

    SHR EAX,1 / PUSHFD      ; stall
    SHR EAX,1 / OR EAX,EAX / PUSHFD   ; no stall

    TEST EBX,EBX / LAHF     ; stall
    AND  EBX,EBX / LAHF     ; no stall
    TEST EBX,EBX / SETZ AL  ; no stall

    CLC / SETZ AL           ; stall
    CLD / SETZ AL           ; no stall
```

The penalty for partial flags stalls is approximately 4 clocks.

19.3 Flags stalls after shifts and rotates

You can get a stall resembling the partial flags stall when reading any flag bit after a shift or rotate, except for shifts and rotates by one (short form):

```
    SHR EAX,1 / JZ XX                ; no stall
    SHR EAX,2 / JZ XX                ; stall
    SHR EAX,2 / OR EAX,EAX / JZ XX   ; no stall

    SHR EAX,5 / JC XX                ; stall
    SHR EAX,4 / SHR EAX,1 / JC XX    ; no stall

    SHR EAX,CL / JZ XX               ; stall, even if CL = 1
    SHRD EAX,EBX,1 / JZ XX           ; stall
    ROL EBX,8 / JC XX                ; stall
```

The penalty for these stalls is approximately 4 clocks.

19.4 Partial memory stalls

A partial memory stall is somewhat analogous to a partial register stall. It occurs when you mix data sizes for the same memory address:

```
        MOV BYTE PTR [ESI], AL
        MOV EBX, DWORD PTR [ESI]        ; partial memory stall
```

Here you get a stall because the processor has to combine the byte written from AL with the next three bytes, which were in memory before, to get the four bytes needed for reading into `EBX`. The penalty is approximately 7-8 clocks.

Unlike the partial register stalls, you also get a partial memory stall when you write a bigger operand to memory and then read part of it, if the smaller part doesn't start at the same address:

```
MOV DWORD PTR [ESI], EAX
MOV BL, BYTE PTR [ESI]          ; no stall
MOV BH, BYTE PTR [ESI+1]        ; stall
```

You can avoid this stall by changing the last line to `MOV BH,AH`, but such a solution is not possible in a situation like this:

```
FISTP QWORD PTR [EDI]
MOV EAX, DWORD PTR [EDI]
MOV EDX, DWORD PTR [EDI+4]      ; stall
```

Interestingly, you can also get a partial memory stall when writing and reading completely different addresses if they happen to have the same set-value in different cache banks:

```
MOV BYTE PTR [ESI], AL
MOV EBX, DWORD PTR [ESI+4092]   ; no stall
MOV ECX, DWORD PTR [ESI+4096]   ; stall
```

20. Dependency chains (PPro, PII and PIII)

A series of instructions where each instruction depends on the result of the preceding one is called a dependency chain. Long dependency chains should be avoided, if possible, because they prevent out-of-order and parallel execution.

Example:

```
MOV EAX, [MEM1]
ADD EAX, [MEM2]
ADD EAX, [MEM3]
ADD EAX, [MEM4]
MOV [MEM5], EAX
```

In this eaxmple, the `ADD` instructions generate 2 uops each, one for reading from memory (port 2), and one for adding (port 0 or 1). The read uops can execute out or order, while the add uops must wait for the previous uops to finish. This dependency chain does not take very long to execute, because each addition adds only 1 clock to the execution time. But if you have slow instructions like multiplications, or even worse: divisions, then you should definitely do something to break the dependency chain. The way to do this is to use multiple accumulators:

```
MOV EAX, [MEM1]         ; start first chain
MOV EBX, [MEM2]         ; start other chain in different accumulator
IMUL EAX, [MEM3]
IMUL EBX, [MEM4]
IMUL EAX, EBX           ; join chains in the end
MOV [MEM5], EAX
```

Here, the second `IMUL` instruction can start before the first one is finished. Since the `IMUL` instruction has a delay of 4 clocks and is fully pipelined, you may have up to 4 accumulators.

Division is not pipelined so you cannot do the same with chained divisions, but you can of course multiply all the divisors and do only one division in the end.

Floating point instructions have a longer delay than integer instructions, so you should definitely break up long dependency chains with floating point instructions:

```
FLD [MEM1]         ; start first chain
FLD [MEM2]         ; start second chain in different accumulator
FADD [MEM3]
FXCH
FADD [MEM4]
FXCH
FADD [MEM5]
FADD               ; join chains in the end
FSTP [MEM6]
```

You need a lot of `FXCH` instructions for this, but don't worry: they are cheap. `FXCH` instructions are resolved in the RAT by register renaming so they don't put any load on the execution ports. An `FXCH` does count as 1 uop in the RAT, ROB, and retirement station, though.

If the dependency chain is long you may need three accumulators:

```
        FLD [MEM1]              ; start first chain
        FLD [MEM2]              ; start second chain
        FLD [MEM3]              ; start third chain
        FADD [MEM4]             ; third chain
        FXCH ST(1)
        FADD [MEM5]             ; second chain
        FXCH ST(2)
        FADD [MEM6]             ; first chain
        FXCH ST(1)
        FADD [MEM7]             ; third chain
        FXCH ST(2)
        FADD [MEM8]             ; second chain
        FXCH ST(1)
        FADD                    ; join first and third chain
        FADD                    ; join with second chain
        FSTP [MEM9]
```

Avoid storing intermediate data in memory and read them immediately afterwards:

```
        MOV [TEMP], EAX
        MOV EBX, [TEMP]
```

There is a penalty for attempting to read from a memory address before a previous write to that address is finished. In the example above, change the last instruction to `MOV EBX,EAX` or put some other instructions in between.

There is one situation where you cannot avoid storing intermediate data in memory, and that is when transferring data from an integer register to a floating point register, or vice versa. For example:

```
MOV EAX, [MEM1]
ADD EAX, [MEM2]
MOV [TEMP], EAX
FILD [TEMP]
```

If you don't have anything to put in between the write to `TEMP` and the read from `TEMP`, then you may consider using a floating point register instead of `EAX`:

```
FILD [MEM1]
FIADD [MEM2]
```

Consecutive jumps, calls, or returns may also be considered dependency chains. The throughput for these instructions is one jump per two clock cycles. It is therefore recommended that you give the microprocessor something else to do between the jumps.

21. Searching for bottlenecks (PPro, PII and PIII)

When optimizing code for these processors, it is important to analyze where the bottlenecks are. Spending time on optimizing away one bottleneck doesn't make sense if there is another bottleneck which is narrower.

If you expect code cache misses then you should restructure your code to keep the most used parts of code together.

If you expect many data cache misses then forget about everything else and concentrate on how to restructure your data to reduce the number of cache misses (chapter 7), and avoid long dependency chains after a data read cache miss (chapter 20).

If you have many divisions then try to reduce them (chapter 27.2) and make sure the processor has something else to do during the divisions.

Dependency chains tend to hamper out-of-order execution (chapter 20). Try to break long dependency chains, especially if they contain slow instructions such as multiplication, division, and floating point instructions.

If you have many jumps, calls, or returns, and especially if the jumps are poorly predictable, then try if some of them can be avoided. Replace conditional jumps with conditional moves if possible, and replace small procedures with macros (chapter 22.3).

If you are mixing different data sizes (8, 16, and 32 bit integers) then look out for partial stalls. If you use `PUSHF` or `LAHF` instructions then look out for partial flags stalls. Avoid testing flags after shifts or rotates by more than 1 (chapter 19).

If you aim at a throughput of 3 uops per clock cycle then be aware of possible delays in instruction fetch and decoding (chapter and 14 and 15), especially in small loops.

The limit of two permanent register reads per clock cycle may reduce your throughput to less than 3 uops per clock cycle (chapter 16.2). This is likely to happen if you often read registers more than 4 clock cycles after they last were modified. This may, for example, happen if you often use pointers for addressing your data but seldom modify the pointers.

A throughput of 3 uops per clock requires that no execution port gets more than one third of the uops (chapter 17).

The retirement station can handle 3 uops per clock, but may be slightly less effective for taken jumps (chapter 18).

22. Jumps and branches (all processors)

The Pentium family of processors attempt to predict where a jump will go to, and whether a conditional jump will be taken or fall through. If the prediction is correct, then it can save a considerable amount of time by loading the subsequent instructions into the pipeline and start decoding them before the jump is executed. If the prediction turns out to be wrong, then the pipeline has to be flushed, which will cost a penalty depending on the length of the pipeline.

The predictions are based on a Branch Target Buffer (BTB) which stores the history for each branch or jump instruction and makes predictions based on the prior history of executions of each instruction. The BTB is organized like a set-associative cache where new entries are allocated according to a pseudo-random replacement method.

When optimizing code, it is important to minimize the number of misprediction penalties. This requires a good understanding of how the jump prediction works.

The branch prediction mechanisms are not described adequately in Intel manuals or anywhere else. I am therefore giving a very detailed description here. This information is based on my own research (with the help of Karki Jitendra Bahadur for the PPlain).

In the following, I will use the term 'control transfer instruction' for any instruction which can change the instruction pointer, including conditional and unconditional, direct and indirect, near and far, jumps, calls, and returns. All these instructions use prediction.

22.1 Branch prediction in PPlain

The branch prediction mechanism for the PPlain is very different from the other three processors. Information found in Intel documents and elsewhere on this subject is directly misleading, and following the advises given is such documents is likely to lead to sub-optimal code.

The PPlain has a branch target buffer (BTB), which can hold information for up to 256 jump instructions. The BTB is organized like a 4-way set-associative cache with 64 entries per way. This means that the BTB can hold no more than 4 entries with the same set value. Unlike the data cache, the BTB uses a pseudo random replacement algorithm, which means that a new entry will not necessarily displace the least recently used entry of the same set-value. How the set-value is calculated will be explained later. Each BTB entry stores the address of the jump target and a prediction state, which can have four different values:

state 0: "strongly not taken"
state 1: "weakly not taken"
state 2: "weakly taken"
state 3: "strongly taken"

A branch instruction is predicted to jump when in state 2 or 3, and to fall through when in state 0 or 1. The state transition works like a two-bit counter, so that the state is incremented when the branch is taken, and decremented when it falls through. The counter saturates, rather than wrap around, so that it does not decrement beyond 0 or increment beyond 3. Ideally, this would provide a reasonably good prediction, because a branch instruction would have to deviate twice from what it does most of the time, before the prediction changes.

However, this mechanism has been compromised by the fact that state 0 also means 'unused BTB entry'. So a BTB entry in state 0 is the same as no BTB entry. This makes sense, because a branch instruction is predicted to fall through if it has no BTB entry. This improves the utilization of the BTB, because a branch instruction which is seldom taken will most of the time not take up any BTB entry.

Now, if a jumping instruction has no BTB entry, then a new BTB entry will be generated, and this new entry will always be set to state 3. This means that it is impossible to go from state 0 to state 1 (except for a very special case discussed later). From state 0 you can only go to state 3, if the branch is taken. If the branch falls through, then it will stay out of the BTB.

This is a serious design flaw. By throwing state 0 entries out of the BTB and always setting new entries to state 3, the designers apparently have given priority to minimizing the first time penalty for unconditional jumps and branches often taken, and ignored that this seriously compromises the basic idea behind the mechanism and reduces the performance in small innermost loops. The consequence of this flaw is, that a branch instruction which falls through most of the time will have up to three times as many mispredictions as a branch instruction which is taken most of the time. (Apparently, Intel engineers have been unaware of this flaw until I published my findings).

You may take this asymmetry into account by organizing your branches so that they are taken more often than not. Consider for example this if-then-else construction:

```
        TEST EAX,EAX
        JZ   A
        <branch 1>
        JMP  E
A:      <branch 2>
E:
```

If branch 1 is executed more often than branch 2, and branch 2 is seldom executed twice in succession, then you can reduce the number of branch mispredictions by up to a factor 3 by swapping the two branches so that the branch instruction will jump more often than fall through:

```
        TEST EAX,EAX
        JNZ  A
        <branch 2>
        JMP  E
A:      <branch 1>
E:
```

(This is contrary to the recommendations in Intel's manuals and tutorials).

There may be reasons to put the most often executed branch first, however:

1. Putting seldom executed branches away in the bottom of your code can improve code cache utilization.
2. A branch instruction seldom taken will stay out of the BTB most of the time, possibly improving BTB utilization.
3. The branch instruction will be predicted as not taken if it has been flushed out of the BTB by other branch instructions.
4. The asymmetry in branch prediction only exists on the PPlain.

These considerations have little weight, however, for small critical loops, so I would still recommend organizing branches with a skewed distribution so that the branch instruction is taken more often than not, unless branch 2 is executed so seldom, that misprediction doesn't matter.

Likewise, you should preferably organize loops with the testing branch instruction at the bottom, as in this example:

```
        MOV ECX, [N]
L:      MOV [EDI],EAX
        ADD EDI,4
        DEC ECX
        JNZ L
```

If N is high, then the JNZ instruction here will be taken more often than not, and never fall through twice in succession.

Consider the situation where a branch is taken every second time. The first time it jumps the BTB entry will go into state 3, and will then alternate between state 2 and 3. It is predicted to jump all the time, which gives 50% mispredictions. Assume now that it deviates from this regular pattern and falls through an extra time. The jump pattern is:

```
01010100101010101010101, where 0 means nojump, and 1 means jump.
       ^
```

The extra nojump is indicated with a `^` above. After this incident, the BTB entry will alternate between state 1 and 2, which gives 100% mispredictions. It will continue in this unfortunate mode until there is another deviation from the 0101 pattern. This is the worst case for this branch prediction mechanism.

22.1.2 BTB is looking ahead (PPlain)

The BTB mechanism is counting instruction pairs, rather than single instructions, so you have to know how instructions are pairing in order to analyze where a BTB entry is stored. The BTB entry for any control instruction is attached to the address of the U-pipe instruction in the preceding instruction pair. (An unpaired instruction counts as one pair). Example:

```
SHR EAX,1
MOV EBX,[ESI]
CMP EAX,EBX
JB  L
```

Here `SHR` pairs with `MOV`, and `CMP` pairs with `JB`. The BTB entry for `JB L` is thus attached to the address of the `SHR EAX,1` instruction. When this BTB entry is met, and if it is in state 2 or 3, then the Pentium will read the target address from the BTB entry, and load the instructions following L into the pipeline. This happens before the branch instruction has been decoded, so the Pentium relies solely on the information in the BTB when doing this.

You may remember, that instructions are seldom pairing the first time they are executed (see chapter 8). If the instructions above are not pairing, then the BTB entry should be attached to the address of the `CMP` instruction, and this entry would be wrong on the next execution, when instructions are pairing. However, in most cases the PPlain is smart enough to not make a BTB entry when there is an unused pairing opportunity, so you don't get a BTB entry until the second execution, and hence you won't get a prediction until the third execution. (In the rare case, where every second instruction is a single-byte instruction, you may get a BTB entry on the first execution which becomes invalid in the second execution, but since the instruction it is attached to will then go to the V-pipe, it is ignored and gives no penalty. A BTB entry is only read if it is attached to the address of a U-pipe instruction).

A BTB entry is identified by its set-value which is equal to bits 0-5 of the address it is attached to. Bits 6-31 are then stored in the BTB as a tag. Addresses which are spaced a multiple of 64 bytes apart will have the same set-value. You can have no more than four BTB entries with the same set-value. If you want to check whether your jump instructions contend for the same BTB entries, then you have to compare bits 0-5 of the addresses of the U-pipe instructions in the preceding instruction pairs. This is very tedious, and I have never heard of anybody doing so. There are no tools available to do this job for you.

22.1.3 Consecutive branches (PPlain)

When a jump is mispredicted, then the pipeline gets flushed. If the next instruction pair executed also contains a control transfer instruction, then the PPlain won't load its target because it cannot load a new target while the pipeline is being flushed. The result is that the second jump instruction is predicted to fall through regardless of the state of its BTB entry. Therefore, if the second jump is also taken, then you will get another penalty. The state of the BTB entry for the second jump instruction does get correctly updated, though.

If you have a long chain of control transfer instructions, and the first jump in the chain is mispredicted, then the pipeline will get flushed all the time, and you will get nothing but mispredictions until you meet an instruction pair which does not jump. The most extreme case of this is a loop which jumps to itself: It will get a misprediction penalty for each iteration.

This is not the only problem with consecutive control transfer instructions. Another problem is that you can have another branch instruction between a BTB entry and the control transfer instruction it belongs to. If the first branch instruction jumps to somewhere else, then strange things may happen. Consider this example:

```
        SHR EAX,1
        MOV EBX,[ESI]
        CMP EAX,EBX
        JB  L1
        JMP L2

L1:     MOV EAX,EBX
        INC EBX
```

When `JB L1` falls through, then you will get a BTB entry for `JMP L2` attached to the address of `CMP EAX,EBX`. But what will happen when `JB L1` later is taken? At the time when the BTB entry for `JMP L2` is read, the processor doesn't know that the next instruction pair does not contain a jump instruction, so it will actually predict the instruction pair `MOV EAX,EBX / INC EBX` to jump to `L2`. The penalty for predicting non-jump instructions to jump is 3 clock cycles. The BTB entry for `JMP L2` will get its state decremented, because it is applied to something which doesn't jump. If we keep going to `L1`, then the BTB entry for `JMP L2` will be decremented to state 1 and 0, so that the problem will disappear until next time `JMP L2` is executed.

The penalty for predicting the non-jumping instructions to jump only occurs when the jump to `L1` is predicted. In the case that `JB L1` is mispredictedly jumping, then the pipeline gets flushed and we won't get the false `L2` target loaded, so in this case we will not see the penalty of predicting the non-jumping instructions to jump, but we do get the BTB entry for `JMP L2` decremented.

Suppose, now, that we replace the `INC EBX` instruction above with another jump instruction. This third jump instruction will then use the same BTB entry as `JMP L2` with the possible penalty of predicting a wrong target, (unless it happens to also have `L2` as target).

To summarize, consecutive jumps can lead to the following problems:

- failure to load a jump target when the pipeline is being flushed by a preceding mispredicted jump.
- a BTB entry being mis-applied to non-jumping instructions and predicting them to jump.
- a second consequence of the above is that a mis-applied BTB entry will get its state decremented, possibly leading to a later misprediction of the jump it belongs to. Even unconditional jumps can be predicted to fall through for this reason.

- two jump instructions may share the same BTB entry, leading to the prediction of a wrong target.

All this mess may give you a lot of penalties, so you should definitely avoid having an instruction pair containing a jump immediately after another poorly predictable control transfer instruction or its target.

It is time for another illustrative example:

```
        CALL P
        TEST EAX,EAX
        JZ   L2
L1:     MOV  [EDI],EBX
        ADD  EDI,4
        DEC  EAX
        JNZ  L1
L2:     CALL P
```

This looks like a quite nice and normal piece of code: A function call, a loop which is bypassed when the count is zero, and another function call. How many problems can you spot in this program?

First, we may note that the function `P` is called alternatingly from two different locations. This means that the target for the return from `P` will be changing all the time. Consequently, the return from `P` will always be mispredicted.

Assume, now, that `EAX` is zero. The jump to `L2` will not have its target loaded because the mispredicted return caused a pipeline flush. Next, the second `CALL P` will also fail to have its target loaded because `JZ L2` caused a pipeline flush. Here we have the situation where a chain of consecutive jumps makes the pipeline flush repeatedly because the first jump was mispredicted. The BTB entry for `JZ L2` is stored at the address of `P`'s return instruction. This BTB entry will now be mis-applied to whatever comes after the second `CALL P`, but that doesn't give a penalty because the pipeline is flushed by the mispredicted second return.

Now, let's see what happens if `EAX` has a nonzero value the next time: `JZ L2` is always predicted to fall through because of the flush. The second `CALL P` has a BTB entry at the address of `TEST EAX,EAX`. This entry will be mis-applied to the `MOV/ADD` pair, predicting it to jump to `P`. This causes a flush which prevents `JNZ L1` from loading its target. If we have been here before, then the second `CALL P` will have another BTB entry at the address of `DEC EAX`. On the second and third iteration of the loop, this entry will also be mis-applied to the `MOV/ADD` pair, until it has had its state decremented to 1 or 0. This will not cause a penalty on the second iteration because the flush from `JNZ L1` prevents it from loading its false target, but on the third iteration it will. The subsequent iterations of the loop have no penalties, but when it exits, `JNZ L1` is mispredicted. The flush would now prevent `CALL P` from loading its target, were it not for the fact that the BTB entry for `CALL P` has already been destroyed by being mis-applied several times.

We can improve this code by putting in some `NOP`'s to separate all consecutive jumps:

```
        CALL P
        TEST EAX,EAX
        NOP
        JZ   L2
L1:     MOV  [EDI],EBX
        ADD  EDI,4
        DEC  EAX
        JNZ  L1
L2:     NOP
        NOP
        CALL P
```

The extra `NOP`'s cost 2 clock cycles, but they save much more. Furthermore, `JZ L2` is now moved to the U-pipe which reduces its penalty from 4 to 3 when mispredicted. The only problem that remains is that the returns from `P` are always mispredicted. This problem can only be solved by replacing the call to `P` by an inline macro (if you have enough code cache).

The lesson to learn from this example is that you should always look carefully for consecutive jumps and see if you can save time by inserting some `NOP`'s. You should be particularly aware of those situations where misprediction is unavoidable, such as loop exits and returns from procedures which are called from varying locations. If you have something useful to put in, instead of the `NOP`'s, then you should of course do so.

Multiway branches (case statements) may be implemented either as a tree of branch instructions or as a list of jump addresses. If you choose to use a tree of branch instructions, then you have to include some `NOP`'s or other instructions to separate the consecutive branches. A list of jump addresses may therefore be a better solution on the PPlain. The list of jump addresses should be placed in the data segment. Never put data in the code segment!

22.1.4 Tight loops (PPlain)

In a small loop you will often access the same BTB entry repeatedly with small intervals. This never causes a stall. Rather than waiting for a BTB entry to be updated, the PPlain somehow bypasses the pipeline and gets the resulting state from the last jump before it has been written to the BTB. This mechanism is almost transparent to the user, but it does in some cases have funny effects: You can see a branch prediction going from state 0 to state 1, rather than to state 3, if the zero has not yet been written to the BTB. This happens if the loop has no more than four instruction pairs. In loops with only two instruction pairs you may sometimes have state 0 for two consecutive iterations without going out of the BTB. In such small loops it also happens in rare cases that the prediction uses the state resulting from two iterations ago, rather than from the last iteration. These funny effects will usually not have any negative effects on performance.

22.2 Branch prediction in PMMX, PPro, PII and PIII

22.2.1 BTB organization (PMMX, PPro, PII and PIII)

The branch target buffer (BTB) of the PMMX has 256 entries organized as 16 ways * 16 sets. Each entry is identified by bits 2-31 of the address of the last byte of the control transfer instruction it belongs to. Bits 2-5 define the set, and bits 6-31 are stored in the BTB as a tag. Control transfer instructions which are spaced 64 bytes apart have the same set-value and may therefore occasionally push each other out of the BTB. Since there are 16 ways per set, this won't happen too often.

The branch target buffer (BTB) of the PPro, PII and PIII has 512 entries organized as 16 ways * 32 sets. Each entry is identified by bits 4-31 of the address of the last byte of the control transfer instruction it belongs to. Bits 4-8 define the set, and all bits are stored in the BTB as a tag. Control transfer instructions which are spaced 512 bytes apart have the same set-value and may therefore occasionally push each other out of the BTB. Since there are 16 ways per set, this won't happen too often.

The PPro, PII and PIII allocate a BTB entry to any control transfer instruction the first time it is executed. The PMMX allocates it the first time it jumps. A branch instruction which never jumps will stay out of the BTB on the PMMX. As soon as it has jumped once, it will stay in the BTB, even if it never jumps again.

An entry may be pushed out of the BTB when another control transfer instruction with the same set-value needs a BTB entry.

22.2.2 Misprediction penalty (PMMX, PPro, PII and PIII)

In the PMMX, the penalty for misprediction of a conditional jump is 4 clocks in the U-pipe, and 5 clocks if it is executed in the V-pipe. For all other control transfer instructions it is 4 clocks.

In the PPro, PII and PIII, the misprediction penalty is very high due to the long pipeline. A misprediction usually costs between 10 and 20 clock cycles. It is therefore very important to be aware of poorly predictable branches when running on PPro, PII and PIII.

22.2.3 Pattern recognition for conditional jumps (PMMX, PPro, PII and PIII)

These processors have an advanced pattern recognition mechanism which will correctly predict a branch instruction which, for example, is taken every fourth time and falls through the other three times. In fact, they can predict any repetitive pattern of jumps and nojumps with a period of up to five, and many patterns with higher periods.

The mechanism is a so-called "two-level adaptive branch prediction scheme", invented by T.-Y. Yeh and Y. N. Patt. It is based on the same kind of two-bit counters as described above for the PPlain (but without the assymmetry flaw). The counter is incremented when the jump is taken and decremented when not taken. There is no wrap-around when counting up from 3 or down from 0. A branch instruction is predicted to be taken when the corresponding counter is in state 2 or 3, and to fall through when in state 0 or 1. An impressive improvement is now obtained by having sixteen such counters for each BTB entry. It selects one of these sixteen counters based on the history of the branch instruction for the last four executions. If, for example, the branch instruction jumps once and then falls through three times, then you have the history bits 1000 (1=jump, 0=nojump). This will make it use counter number 8 (1000 binary = 8) for predicting the next time and update counter 8 afterwards.

If the sequence 1000 is always followed by a 1, then counter number 8 will soon end up in its highest state (state 3) so that it will always predict a 1000 sequence to be followed by a 1. It will take two deviations from this pattern to change the prediction. The repetitive pattern 100010001000 will have counter 8 in state 3, and counter 1, 2 and 4 in state 0. The other twelve counters will be unused.

22.2.4 Perfectly predicted patterns (PMMX, PPro, PII and PIII)

A repetitive branch pattern is predicted perfectly by this mechanism if every 4-bit sub-sequence in the period is unique. Below is a list of repetitive branch patterns which are predicted perfectly:

period	perfectly predicted patterns
1-5	all
6	000011, 000101, 000111, 001011
7	0000101, 0000111, 0001011
8	00001011, 00001111, 00010011, 00010111, 00101101
9	000010011, 000010111, 000100111, 000101101
10	0000100111, 0000101101, 0000101111, 0000110111, 0001010011, 0001011101
11	00001001111, 00001010011, 00001011101, 00010100111
12	000010100111, 000010111101, 000011010111, 000100110111, 000100111011
13	0000100110111, 0000100111011, 0000101001111
14	00001001101111, 00001001111011, 00010011010111, 00010011101011, 00010110011101, 00010110100111
15	000010011010111, 000010011101011, 000010100110111, 000010100111011, 000010110011101, 000010110100111, 000010111010011, 000011010010111
16	0000100110101111, 0000100111101011, 0000101100111101, 0000101101001111

When reading this table, you should be aware that if a pattern is predicted correctly than the same pattern reversed (read backwards) is also predicted correctly, as well as the same pattern with all bits inverted. Example: In the table we find the pattern: 0001011. Reversing this pattern gives: 1101000. Inverting all bits gives: 1110100. Both reversing and inverting: 0010111. These four patterns are all recognizable. Rotating the pattern one place to the left gives: 0010110. This is of course not a new pattern, only a phase shifted version of the same pattern. All patterns which can be derived from one of the patterns in the table by reversing, inverting and rotating are also recognizable. For reasons of brevity, these are not listed.

It takes two periods for the pattern recognition mechanism to learn a regular repetitive pattern after the BTB entry has been allocated. The pattern of mispredictions in the learning period is not reproducible. This is probably because the BTB entry contained something prior to allocation. Since BTB entries are allocated according to a random scheme, there is little chance of predicting what happens during the initial learning period.

22.2.5 Handling deviations from a regular pattern (PMMX, PPro, PII and PIII)

The branch prediction mechanism is also extremely good at handling 'almost regular' patterns, or deviations from the regular pattern. Not only does it learn what the regular pattern looks like. It also learns what deviations from the regular pattern look like. If deviations are always of the same type, then it will remember what comes after the irregular event, and the deviation will cost only one misprediction.

Example:

```
0001110001110001110001011100011100011100010111000
                      ^                   ^
```

In this sequence, a 0 means nojump, a 1 means jump. The mechanism learns that the repeated sequence is 000111. The first irregularity is an unexpected 0, which I have marked with a `^`. After this 0 the next three jumps may be mispredicted, because it hasn't learned what comes after 0010, 0101, and 1011. After one or two irregularities of the same kind it has learned that after 0010 comes a 1, after 0101 comes 1, and after 1011 comes 1. This means that after at most two irregularities of the same kind, it has learned to handle this kind of irregularity with only one misprediction.

The prediction mechanism is also very effective when alternating between two different regular patterns. If, for example, we have the pattern 000111 (with period 6) repeated many times, then the pattern 01 (period 2) many times, and then return to the 000111 pattern, then the mechanism doesn't have to relearn the 000111 pattern, because the counters used in the 000111 sequence have been left un-touched during the 01 sequence. After a few alternations between the two patterns, it has also learned to handle the changes of pattern with only one misprediction for each time the pattern is switched.

22.2.6 Patterns which are not predicted perfectly (PMMX, PPro, PII and PIII)

The simplest branch pattern which cannot be predicted perfectly is a branch which is taken on every 6'th execution. The pattern is:

```
000001000001000001
    ^^    ^^    ^^
    ab    ab    ab
```

The sequence 0000 is alternatingly followed by a 0, in the positions marked a above, and by a 1, in the positions marked b. This affects counter number 0 which will count up and down all the time. If counter 0 happens to start in state 0 or 1, then it will alternate between state

0 and 1. This will lead to a misprediction in position b. If counter 0 happens to start in state 3, then it will alternate between state 2 and 3 which will cause a misprediction in position a. The worst case is when it starts in state 2. It will alternate between state 1 and 2 with the unfortunate consequence that we get a misprediction both in position a and b. (This is analogous to the worst case for the PPlain explained above). Which of these four situations we will get depends on the history of the BTB entry prior to allocation to this branch. This is beyond our control because of the random allocation method.

In principle, it is possible to avoid the worst case situation where we have two mispredictions per cycle by giving it an initial branch sequence which is specially designed for putting the counter in the desired state. Such an approach cannot be recommended, however, because of the considerable extra code complexity required, and because whatever information we have put into the counter is likely to be lost during the next timer interrupt or task switch.

22.2.7 Completely random patterns (PMMX, PPro, PII and PIII)

The powerful capability of pattern recognition has a minor drawback in the case of completely random sequences with no regularities.

The following table lists the experimental fraction of mispredictions for a completely random sequence of jumps and nojumps:

fraction of jumps/nojumps	fraction of mispredictions
0.001/0.999	0.001001
0.01/0.99	0.0101
0.05/0.95	0.0525
0.10/0.90	0.110
0.15/0.85	0.171
0.20/0.80	0.235
0.25/0.75	0.300
0.30/0.70	0.362
0.35/0.65	0.418
0.40/0.60	0.462
0.45/0.55	0.490
0.50/0.50	0.500

The fraction of mispredictions is slightly higher than it would be without pattern recognition because the processor keeps trying to find repeated patterns in a sequence which has no regularities.

22.2.8 Tight loops (PMMX)

The branch prediction is not reliable in tiny loops where the pattern recognition mechanism doesn't have time to update its data before the next branch is met. This means that simple patterns, which would normally be predicted perfectly, are not recognized. Incidentally, some patterns which normally would not be recognized, are predicted perfectly in tight loops. For example, a loop which always repeats 6 times would have the branch pattern 111110 for the branch instruction at the bottom of the loop. This pattern would normally have one or two mispredictions per iteration, but in a tight loop it has none. The same applies to a loop which repeats 7 times. Most other repeat counts are predicted poorer in tight loops than normally. This means that a loop which iterates 6 or 7 times should preferably be tight, whereas other loops should preferably not be tight. You may unroll a loop if necessary to make it less tight.

To find out whether a loop will behave as 'tight' on the PMMX you may follow the following rule of thumb: Count the number of instructions in the loop. If the number is 6 or less, then the loop will behave as tight. If you have more than 7 instructions, then you can be reasonably sure that the pattern recognition functions normally. Strangely enough, it doesn't matter how many clock cycles each instruction takes, whether it has stalls, or whether it is paired or not. Complex integer instructions do not count. A loop can have lots of complex integer instructions and still behave as a tight loop. A complex integer instruction is a non-pairable integer instruction which always takes more than one clock cycle. Complex floating point instructions and MMX instructions still count as one. Note, that this rule of thumb is heuristic and not completely reliable. In important cases you may want to do your own testing. You can use performance monitor counter number 35H for the PMMX to count branch mispredictions. Test results may not be completely deterministic, because branch predictions may depend on the history of the BTB entry prior to allocation.

Tight loops on PPro, PII and PIII are predicted normally, and take minimum two clock cycles per iteration.

22.2.9 Indirect jumps and calls (PMMX, PPro, PII and PIII)

There is no pattern recognition for indirect jumps and calls, and the BTB can remember no more than one target for an indirect jump. It is simply predicted to go to the same target as it did last time.

22.2.10 JECXZ and LOOP (PMMX)

There is no pattern recognition for these two instructions in the PMMX. They are simply predicted to go the same way as last time they were executed. These two instructions should be avoided in time-critical code for PMMX. (In PPro, PII and PIII they are predicted using pattern recognition, but the loop instruction is still inferior to `DEC ECX / JNZ`).

22.2.11 Returns (PMMX, PPro, PII and PIII)

The PMMX, PPro, PII and PIII processors have a Return Stack Buffer (RSB) which is used for predicting return instructions. The RSB works as a First-In-Last-Out buffer. Each time a call instruction is executed, the corresponding return address is pushed into the RSB. And each time a return instruction is executed, a return address is pulled out of the RSB and used for prediction of the return. This mechanism makes sure that return instructions are correctly predicted when the same subroutine is called from several different locations.

In order to make sure this mechanism works correctly, you must make sure that all calls and returns are matched. Never jump out of a subroutine without a return and never use a return as an indirect jump if speed is critical.

The RSB can hold four entries in the PMMX, sixteen in the PPro, PII and PIII. In the case where the RSB is empty, the return instruction is predicted in the same way as an indirect jump, i.e. it is expected to go to the same target as it did last time.

On the PMMX, when subroutines are nested deeper than four levels then the innermost four levels use the RSB, whereas all subsequent returns from the outer levels use the simpler prediction mechanism as long as there are no new calls. A return instruction which uses the RSB still occupies a BTB entry. Four entries in the RSB of the PMMX doesn't sound of much, but it is probably sufficient. Subroutine nesting deeper than four levels is certainly not unusual, but only the innermost levels matter in terms of speed, except possibly for recursive procedures.

On the PPro, PII and PIII, when subroutines are nested deeper than sixteen levels then the innermost 16 levels use the RSB, whereas all subsequent returns from the outer levels are mispredicted. Recursive subroutines should therefore not go deeper than 16 levels.

22.2.12 Static prediction in PMMX

A control transfer instruction which has not been seen before or which is not in the BTB is always predicted to fall through on the PMMX. It doesn't matter whether it goes forward or backwards.

A branch instruction will not get a BTB entry if it always falls through. As soon as it is taken once, it will get into the BTB and stay there no matter how many times it falls through. A control transfer instruction can only go out of the BTB when it is pushed out by another control transfer instruction which steals its BTB entry.

Any control transfer instruction which jumps to the address immediately following itself will not get a BTB entry. Example:

```
        JMP SHORT LL
LL:
```

This instruction will never get a BTB entry and therefore always have a misprediction penalty.

22.2.13 Static prediction in PPro, PII and PIII

On PPro, PII and PIII, a control transfer instruction which has not been seen before or which is not in the BTB is predicted to fall through if it goes forwards, and to be taken if it goes backwards (e.g. a loop). Static prediction takes longer time than dynamic prediction on these processors.

If your code is unlikely to be cached then it is preferred to have the most frequently executed branch fall through in order to improve prefetching.

22.2.14 Close jumps (PMMX)

On the PMMX, there is a risk that two control transfer instructions will share the same BTB entry if they are too close to each other. The obvious result is that they will always be mispredicted.

The BTB entry for a control transfer instruction is identified by bits 2-31 of the address of the last byte in the instruction. If two control transfer instructions are so close together that they differ only in bits 0-1 of the address, then we have the problem of a shared BTB entry. Example:

```
        CALL    P
        JNC     SHORT L
```

If the last byte of the `CALL` instruction and the last byte of the `JNC` instruction lie within the same dword of memory, then we have the penalty. You have to look at the output list file from the assembler to see whether the two addresses are separated by a DWORD boundary or not. (A DWORD boundary is an address divisible by 4).

There are various ways to solve this problem:
1. Move the code sequence a little up or down in memory so that you get a dword boundary between the two addresses.
2. Change the short jump to a near jump (with 4 bytes displacement) so that the end of the instruction is moved further down. There is no way you can force the assembler to use anything but the shortest form of an instruction so you have to hard-code the near branch if you choose this solution.
3. Put in some instruction between the `CALL` and the `JNC` instructions. This is the easiest method, and the only method if you don't know where DWORD boundaries are because your segment is not dword aligned or because the code keeps moving up and down as you make changes in the preceding code:

```
        CALL    P
        MOV     EAX,EAX           ; two bytes filler to be safe
        JNC     SHORT L
```

If you want to avoid problems on the PPlain too, then put in two `NOP`'s instead to prevent pairing (see section 22.1.3 above).

The `RET` instruction is particularly prone to this problem because it is only one byte long:

```
        JNZ     NEXT
        RET
```

Here you may need up to three bytes of fillers:

```
        JNZ     NEXT
        NOP
        MOV     EAX,EAX
        RET
```

22.2.15 Consecutive calls or returns (PMMX)

There is a penalty when the first instruction pair following the target label of a call contains another call instruction or if a return follows immediately after another return. Example:

```
FUNC1   PROC    NEAR
        NOP             ; avoid call after call
        NOP
        CALL    FUNC2
        CALL    FUNC3
        NOP             ; avoid return after return
        RET
FUNC1   ENDP
```

Two `NOP`'s are required before `CALL FUNC2` because a single `NOP` would pair with the `CALL`. One `NOP` is enough before the `RET` because `RET` is unpairable. No `NOP`'s are required between the two `CALL` instructions because there is no penalty for call after return. (On the PPlain you would need two `NOP`'s here too).

The penalty for chained calls only occurs when the same subroutines are called from more than one location (probably because the RSB needs updating). Chained returns always have a penalty. There is sometimes a small stall for a jump after a call, but no penalty for return after call; call after return; jump, call, or return after jump; or jump after return.

22.2.16 Chained jumps (PPro, PII and PIII)

A jump, call, or return cannot be executed in the first clock cycle after a previous jump, call, or return. Therefore, chained jumps will take two clock cycles for each jump, and you may want to make sure that the processor has something else to do in parallel. For the same reason, a loop will take at least two clock cycles per iteration on these processors.

22.2.17 Designing for branch predictabiligy (PMMX, PPro, PII and PIII)

Multiway branches (switch/case statements) are implemented either as an indirect jump using a list of jump addresses, or as a tree of branch instructions. Since indirect jumps are poorly predicted, the latter method may be preferred if easily predicted patterns can be expected and you have enough BTB entries. In case you decide to use the former method, then it is recommended that you put the list of jump addresses in the data segment.

You may want to reorganize your code so that branch patterns which are not predicted perfectly can be replaced by other patterns which are. Consider, for example, a loop which always executes 20 times. The conditional jump at the bottom of the loop is taken 19 times and falls through every 20'th time. This pattern is regular, but not recognized by the pattern recognition mechanism, so the fall-through is always mispredicted. You may make two nested loops by four and five, or unroll the loop by four and let it execute 5 times, in order to have only recognizable patterns. This kind of complicated schemes are only worth the extra code on the PPro, PII and PIII processors where mispredictions are very expensive. For higher loop counts there is no reason to do anything about the single misprediction.

22.3. Avoiding jumps (all processors)

There can be many reasons why you may want reduce the number of jumps, calls and returns:

- jump mispredictions are very expensive,
- there are various penalties for consecutive or chained jumps, depending on the processor,
- jump instructions may push one another out of the branch target buffer because of the random replacement algorithm,
- a return takes 2 clocks on PPlain and PMMX, calls and returns generate 4 uops on PPro, PII and PIII.
- on PPro, PII and PIII, instruction fetch may be delayed after a jump (chapter 15), and retirement may be slightly less effective for taken jumps then for other uops (chapter 18).

Calls and returns can be avoided by replacing small procedures with inline macros. And in many cases it is possible to reduce the number of jumps by restructuring your code. For example, a jump to a jump should be replaced by a jump to the final target. In some cases this is even possible with conditional jumps if the condition is the same or is known. A jump to a return can be replaced by a return. If you want to eliminate a return to a return, then you should not manipulate the stack pointer because that would interfere with the prediction mechanism of the return stack buffer. Instead, you can replace the preceding call with a jump. For example `CALL PRO1 / RET` can be replaced by `JMP PRO1` if `PRO1` ends with the same kind of `RET`.

You may also eliminate a jump by dublicating the code jumped to. This can be useful if you have a two-way branch inside a loop or before a return. Example:

```
A:      CMP     [EAX+4*EDX],ECX
        JE      B
        CALL    X
        JMP     C
B:      CALL    Y
C:      INC     EDX
        JNZ     A
        MOV     ESP, EBP
        POP     EBP
        RET
```

The jump to `C` may be eliminated by dublicating the loop epilog:

```
A:      CMP [EAX+4*EDX],ECX
        JE      B
        CALL    X
        INC     EDX
        JNZ     A
        JMP     D
B:      CALL    Y
C:      INC     EDX
        JNZ     A
D:      MOV     ESP, EBP
        POP     EBP
        RET
```

The most often executed branch should come first here. The jump to `D` is outside the loop and therefore less critical. If this jump is executed so often that it needs optimizing too, then replace it with the three instructions following `D`.

22.4. Avoiding conditional jumps by using flags (all processors)

The most important jumps to eliminate are conditional jumps, especially if they are poorly predictable. Sometimes it is possible to obtain the same effect as a branch by ingenious manipulation of bits and flags. For example you may calculate the absolute value of a signed number without branching:

```
CDQ
XOR EAX,EDX
SUB EAX,EDX
```

(On PPlain and PMMX, use `MOV EDX,EAX / SAR EDX,31` instead of `CDQ`).

The carry flag is particularly useful for this kind of tricks:
Setting carry if a value is zero: `CMP [VALUE],1`
Setting carry if a value is not zero: `XOR EAX,EAX / CMP EAX,[VALUE]`
Incrementing a counter if carry: `ADC EAX,0`
Setting a bit for each time the carry is set: `RCL EAX,1`
Generating a bit mask if carry is set: `SBB EAX,EAX`
Setting a bit on an arbitrary condition: `SETcond AL`
Setting all bits on an arbitrary condition: `XOR EAX,EAX / SETNcond AL / DEC EAX`
(remember to reverse the condition in the last example)

The following example finds the minimum of two unsigned numbers: if (b < a) a = b;

```
SUB EBX,EAX
SBB ECX,ECX
AND ECX,EBX
ADD EAX,ECX
```

The next example chooses between two numbers: if (a != 0) a = b; else a = c;

```
CMP EAX,1
SBB EAX,EAX
XOR ECX,EBX
AND EAX,ECX
XOR EAX,EBX
```

Whether or not such tricks are worth the extra code depends on how predictable a conditional jump would be, whether the extra pairing or scheduling opportunities of the branch-free code can be utilized, and whether there are other jumps following immediately after which could suffer the penalties of consecutive jumps.

22.5. Replacing conditional jumps by conditional moves (PPro, PII and PIII)

The PPro, PII and PIII processors have conditional move instructions intended specifically for avoiding branches because branch misprediction is very time-consuming on these processors. There are conditional move instructions for both integer and floating point registers. For code that will run only on these processors you may replace poorly predictable branches with conditional moves whenever possible. If you want your code to run on all processors then you may make two versions of the most critical parts of the code, one for processors that support conditional move instructions and one for those that don't (see chapter 27.10 for how to detect if conditional moves are supported).

The misprediction penalty for a branch may be so high that it is advantageous to replace it with conditional moves even when it costs several extra instructions. But a conditional move instruction has the disadvantage that it makes dependency chains longer. The conditional move waits for both register operands to be ready even though only one of them is needed. A conditional move is waiting for three operands to be ready: the condition flag and the two move operands. You have to consider if any of these three operands are likely to be delayed by dependency chains or cache misses. If the condition flag is available long before the move operands then you may as well use a branch, because a possible branch misprediction could be resolved while waiting for the move operands. In situations where you have to wait long for a move operand that may not be needed after all, the branch will be faster than the conditional move despite a possible misprediction penalty. The opposite situation is when the condition flag is delayed while both move operands are available early. In this situation the conditional move is preferred over the branch if misprediction is likely.

23. Reducing code size (all processors)

As explained in chapter 7, the code cache is 8 or 16 kb. If you have problems keeping the critical parts of your code within the code cache, then you may consider reducing the size of your code.

32 bit code is usually bigger than 16 bit code because addresses and data constants take 4 bytes in 32 bit code and only 2 bytes in 16 bit code. However, 16 bit code has other penalties such as prefixes and problems with accessing adjacent words simultaneously (see chapter 10.2 above). Some other methods for reducing the size or your code are discussed below.

Both jump addresses, data addresses, and data constants take less space if they can be expressed as a sign-extended byte, i.e. if they are within the interval from -128 to +127.

For jump addresses this means that short jumps take two bytes of code, whereas jumps beyond 127 bytes take 5 bytes if unconditional and 6 bytes if conditional.

Likewise, data addresses take less space if they can be expressed as a pointer and a displacement between -128 and +127. Example:
`MOV EBX,DS:[100000] / ADD EBX,DS:[100004] ; 12 bytes`
Reduce to:
`MOV EAX,100000 / MOV EBX,[EAX] / ADD EBX,[EAX+4] ; 10 bytes`

The advantage of using a pointer obviously increases if you use it many times. Storing data on the stack and using `EBP` or `ESP` as pointer will thus make your code smaller than if you use static memory locations and absolute addresses, provided of course that your data are within +/-127 bytes of the pointer. Using `PUSH` and `POP` to write and read temporary data is even shorter.

Data constants may also take less space if they are between -128 and +127. Most instructions with immediate operands have a short form where the operand is a sign-extended single byte. Examples:

```
PUSH 200      ; 5 bytes
PUSH 100      ; 2 bytes

ADD EBX,128   ; 6 bytes
SUB EBX,-128  ; 3 bytes
```

The most important instruction with an immediate operand which doesn't have such a short form is `MOV`.
Examples:

```
MOV EAX, 0              ; 5 bytes
```

May be changed to:

```
XOR EAX,EAX             ; 2 bytes
```

And

```
MOV EAX, 1              ; 5 bytes
```

May be changed to:

```
XOR EAX,EAX / INC EAX   ; 3 bytes
```

or:

```
PUSH 1 / POP EAX        ; 3 bytes
```

And

```
MOV EAX, -1             ; 5 bytes
```

May be changed to:

```
OR EAX, -1              ; 3 bytes
```

If the same address or constant is used more than once then you may load it into a register. A `MOV` with a 4-byte immediate operand may sometimes be replaced by an arithmetic instruction if the value of the register before the `MOV` is known. Example:

```
MOV     [mem1],200              ; 10 bytes
MOV     [mem2],200              ; 10 bytes
MOV     [mem3],201              ; 10 bytes
MOV     EAX,100                 ;  5 bytes
MOV     EBX,150                 ;  5 bytes
```

Assuming that `mem1` and `mem3` are both within -128/+127 bytes of `mem2`, this may be changed to:

```
MOV     EBX, OFFSET mem2        ;  5 bytes
MOV     EAX,200                 ;  5 bytes
MOV     [EBX+mem1-mem2],EAX     ;  3 bytes
MOV     [EBX],EAX               ;  2 bytes
INC     EAX                     ;  1 byte
MOV     [EBX+mem3-mem2],EAX     ;  3 bytes
SUB     EAX,101                 ;  3 bytes
LEA     EBX,[EAX+50]            ;  3 bytes
```

Be aware of the AGI stall in the `LEA` instruction (for PPlain and PMMX).

You may also consider that different instructions have different lengths. The following instructions take only one byte and are therefore very attractive: `PUSH reg`, `POP reg, INC reg32, DEC reg32`.
`INC` and `DEC` with 8 bit registers take 2 bytes, so `INC EAX` is shorter than `INC AL`.

`XCHG EAX,reg` is also a single-byte instruction and thus takes less space than `MOV EAX,reg`, but it is slower.

Some instructions take one byte less when they use the accumulator than when they use any other register.
Examples:

```
MOV EAX,DS:[100000]  is smaller than  MOV EBX,DS:[100000]
ADD EAX,1000         is smaller than  ADD EBX,1000
```

Instructions with pointers take one byte less when they have only a base pointer (not `ESP`) and a displacement than when they have a scaled index register, or both base pointer and index register, or `ESP` as base pointer.
Examples:

```
MOV EAX,[array][EBX]  is smaller than  MOV EAX,[array][EBX*4]
MOV EAX,[EBP+12]      is smaller than  MOV EAX,[ESP+12]
```

Instructions with `EBP` as base pointer and no displacement and no index take one byte more than with other registers:

```
MOV EAX,[EBX]    is smaller than  MOV EAX,[EBP],  but
MOV EAX,[EBX+4]  is same size as  MOV EAX,[EBP+4].
```

Instructions with a scaled index pointer and no base pointer must have a four byte displacement, even when it is 0:

```
LEA EAX,[EBX+EBX]  is shorter than  LEA EAX,[2*EBX].
```

24. Scheduling floating point code (PPlain and PMMX)

Floating point instructions cannot pair the way integer instructions can, except for one special case, defined by the following rules:

- the first instruction (executing in the U-pipe) must be `FLD, FADD, FSUB, FMUL, FDIV, FCOM, FCHS,` or `FABS`.
- the second instruction (in V-pipe) must be `FXCH`
- the instruction following the `FXCH` must be a floating point instruction, otherwise the `FXCH` will pair imperfectly and take an extra clock cycle.

This special pairing is important, as will be explained shortly.

While floating point instructions in general cannot be paired, many can be pipelined, i.e. one instruction can begin before the previous instruction has finished. Example:

```
FADD ST(1),ST(0)   ; clock cycle 1-3
FADD ST(2),ST(0)   ; clock cycle 2-4
FADD ST(3),ST(0)   ; clock cycle 3-5
FADD ST(4),ST(0)   ; clock cycle 4-6
```

Obviously, two instructions cannot overlap if the second instruction needs the result of the first. Since almost all floating point instructions involve the top of stack register, `ST(0)`, there are seemingly not very many possibilities for making an instruction independent of the result of previous instructions. The solution to this problem is register renaming. The `FXCH` instruction does not in reality swap the contents of two registers, it only swaps their names. Instructions which push or pop the register stack also work by renaming. Floating point register renaming has been highly optimized on the Pentiums so that a register may be renamed while in use. Register renaming never causes stalls - it is even possible to rename a register more than once in the same clock cycle, as for example when you pair `FLD` or `FCOMPP` with `FXCH`.

By the proper use of `FXCH` instructions you may obtain a lot of overlapping in your floating point code. Example:

```
FLD     [a1]    ; clock cycle 1
FADD    [a2]    ; clock cycle 2-4
FLD     [b1]    ; clock cycle 3
FADD    [b2]    ; clock cycle 4-6
FLD     [c1]    ; clock cycle 5
FADD    [c2]    ; clock cycle 6-8
FXCH    ST(2)   ; clock cycle 6
FADD    [a3]    ; clock cycle 7-9
FXCH    ST(1)   ; clock cycle 7
FADD    [b3]    ; clock cycle 8-10
FXCH    ST(2)   ; clock cycle 8
FADD    [c3]    ; clock cycle 9-11
FXCH    ST(1)   ; clock cycle 9
FADD    [a4]    ; clock cycle 10-12
FXCH    ST(2)   ; clock cycle 10
FADD    [b4]    ; clock cycle 11-13
FXCH    ST(1)   ; clock cycle 11
FADD    [c4]    ; clock cycle 12-14
FXCH    ST(2)   ; clock cycle 12
```

In the above example we are interleaving three independent threads. Each `FADD` takes 3 clock cycles, and we can start a new `FADD` in each clock cycle. When we have started an `FADD` in the 'a' thread we have time to start two new `FADD` instructions in the '`b`' and '`c`' threads before returning to the '`a`' thread, so every third `FADD` belongs to the same thread. We are using `FXCH` instructions every time to get the register that belongs to the desired thread into `ST(0)`. As you can see in the example above, this generates a regular pattern, but note well that the `FXCH` instructions repeat with a period of two while the threads have a period of three. This can be quite confusing, so you have to 'play computer' in order to know which registers are where.

All versions of the instructions `FADD, FSUB, FMUL,` and `FILD` take 3 clock cycles and are able to overlap, so that these instructions may be scheduled using the method described above. Using a memory operand does not take more time than a register operand if the memory operand is in the level 1 cache and properly aligned.

By now you must be used to rules having exceptions, and the overlapping rule is no exception: You cannot start an `FMUL` instruction one clock cycle after another `FMUL` instruction, because the `FMUL` circuitry is not perfectly pipelined. It is recommended that you put another instruction in between two `FMUL`'s. Example:

```
FLD     [a1]    ; clock cycle 1
FLD     [b1]    ; clock cycle 2
FLD     [c1]    ; clock cycle 3
FXCH    ST(2)   ; clock cycle 3
FMUL    [a2]    ; clock cycle 4-6
FXCH            ; clock cycle 4
FMUL    [b2]    ; clock cycle 5-7    (stall)
FXCH    ST(2)   ; clock cycle 5
FMUL    [c2]    ; clock cycle 7-9    (stall)
FXCH            ; clock cycle 7
FSTP    [a3]    ; clock cycle 8-9
FXCH            ; clock cycle 10     (unpaired)
FSTP    [b3]    ; clock cycle 11-12
FSTP    [c3]    ; clock cycle 13-14
```

Here you have a stall before `FMUL [b2]` and before `FMUL [c2]` because another `FMUL` started in the preceding clock cycle. You can improve this code by putting `FLD` instructions in between the `FMUL`'s:

```
FLD     [a1]    ; clock cycle 1
FMUL    [a2]    ; clock cycle 2-4
FLD     [b1]    ; clock cycle 3
FMUL    [b2]    ; clock cycle 4-6
FLD     [c1]    ; clock cycle 5
FMUL    [c2]    ; clock cycle 6-8
FXCH    ST(2)   ; clock cycle 6
FSTP    [a3]    ; clock cycle 7-8
FSTP    [b3]    ; clock cycle 9-10
FSTP    [c3]    ; clock cycle 11-12
```

In other cases you may put `FADD, FSUB`, or anything else in between `FMUL`'s to avoid the stalls.

Overlapping floating point instructions requires of course that you have some independent threads that you can interleave. If you have only one big formula to execute, then you may compute parts of the formula in parallel to achieve overlapping. If, for example, you want to add six numbers, then you may split the operations into two threads with three numbers in each, and add the two threads in the end:

```
FLD     [a]     ; clock cycle 1
FADD    [b]     ; clock cycle 2-4
FLD     [c]     ; clock cycle 3
FADD    [d]     ; clock cycle 4-6
FXCH            ; clock cycle 4
FADD    [e]     ; clock cycle 5-7
FXCH            ; clock cycle 5
FADD    [f]     ; clock cycle 7-9    (stall)
FADD            ; clock cycle 10-12  (stall)
```

Here we have a one clock stall before `FADD [f]` because it is waiting for the result of `FADD [d]` and a two clock stall before the last `FADD` because it is waiting for the result of `FADD [f]`. The latter stall can be hidden by filling in some integer instructions, but the first stall can not because an integer instruction at this place would make the `FXCH` pair imperfectly.

The first stall can be avoided by having three threads rather than two, but that would cost an extra `FLD` so we do not save anything by having three threads rather than two unless there are at least eight numbers to add.

Not all floating point instructions can overlap. And some floating point instructions can overlap more subsequent integer instructions than subsequent floating point instructions. The `FDIV` instruction, for example, takes 39 clock cycles. All but the first clock cycle can overlap with integer instructions, but only the last two clock cycles can overlap with floating point instructions. Example:

```
FDIV         ; clock cycle 1-39  (U-pipe)
FXCH         ; clock cycle 1-2   (V-pipe, imperfect pairing)
SHR EAX,1    ; clock cycle 3     (U-pipe)
INC EBX      ; clock cycle 3     (V-pipe)
CMC          ; clock cycle 4-5   (non-pairable)
FADD [x]     ; clock cycle 38-40 (U-pipe, waiting while FPU busy)
FXCH         ; clock cycle 38    (V-pipe)
FMUL [y]     ; clock cycle 40-42 (U-pipe, waiting for result of FDIV)
```

The first `FXCH` pairs with the `FDIV`, but takes an extra clock cycle because it is not followed by a floating point instruction. The `SHR / INC` pair starts before the `FDIV` is finished, but has to wait for the `FXCH` to finish. The `FADD` has to wait till clock 38 because new floating point instructions can only execute during the last two clock cycles of the `FDIV`. The second `FXCH` pairs with the `FADD`. The `FMUL` has to wait for the `FDIV` to finish because it uses the result of the division.

If you have nothing else to put in after a floating point instruction with a large integer overlap, such as `FDIV` or `FSQRT`, then you may put in a dummy read from an address which you expect to need later in the program to make sure it is in the level one cache. Example:

```
        FDIV    QWORD PTR [EBX]
        CMP     [ESI],ESI
        FMUL    QWORD PTR [ESI]
```

Here we use the integer overlap to pre-load the value at `[ESI]` into the cache while the `FDIV` is being computed (we don't care what the result of the `CMP` is).

Chapter 28 gives a complete listing of floating point instructions, and what they can pair or overlap with.

There is no penalty for using a memory operand on floating point instuctions because the arithmetic unit is one step later in the pipeline than the read unit. The tradeoff of this comes when you store floating point data to memory. The `FST` or `FSTP` instruction with a memory operand takes two clock cycles in the execution stage, but it needs the data one clock earlier so you will get a one clock stall if the value to store is not ready one clock cycle in advance. This is analogous to an AGI stall. Example:

```
    FLD     [a1]    ; clock cycle 1
    FADD    [a2]    ; clock cycle 2-4
    FLD     [b1]    ; clock cycle 3
    FADD    [b2]    ; clock cycle 4-6
    FXCH            ; clock cycle 4
    FSTP    [a3]    ; clock cycle 6-7
    FSTP    [b3]    ; clock cycle 8-9
```

The `FSTP [a3]` stalls for one clock cycle because the result of `FADD [a2]` is not ready in the preceding clock cycle. In many cases you cannot hide this type of stall without scheduling your floating point code into four threads or putting some integer instructions in between. The two clock cycles in the execution stage of the `FST(P)` instruction cannot pair or overlap with any subsequent instructions.

Instructions with integer operands such as `FIADD, FISUB, FIMUL, FIDIV, FICOM` may be split up into simpler operations in order to improve overlapping. Example:

```
    FILD    [a]     ; clock cycle 1-3
    FIMUL   [b]     ; clock cycle 4-9
```

Split up into:

```
    FILD    [a]     ; clock cycle 1-3
    FILD    [b]     ; clock cycle 2-4
    FMUL            ; clock cycle 5-7
```

In this example, you save two clocks by overlapping the two FILD instructions.

25. Loop optimization (all processors)

When analyzing a program you often find that most of the time consumption lies in the innermost loop. The way to improve the speed is to carefully optimize the most time-consuming loop using assembly language. The rest of the program may be left in high-level language.

In all the following examples it is assumed that all data are in the level 1 cache. If the speed is limited by cache misses then there is no reason to optimize the instructions. Rather, you should concentrate on organizing your data in a way that minimizes cache misses (see chapter 7).

25.1. Loops in PPlain and PMMX

A loop generally contains a counter controlling how many times to iterate, and often array access reading or writing one array element for each iteration. I have chosen as example a procedure which reads integers from an array, changes the sign of each integer, and stores the results in another array.

A C language code for this procedure would be:

```
void ChangeSign (int * A, int * B, int N) {
  int i;
  for (i=0; i<N; i++) B[i] = -A[i];}
```

Translating to assembly, we might write the procedure like this:

Example 1.1:

```
_ChangeSign PROC NEAR
        PUSH    ESI
        PUSH    EDI
A       EQU     DWORD PTR [ESP+12]
B       EQU     DWORD PTR [ESP+16]
N       EQU     DWORD PTR [ESP+20]
        MOV     ECX, [N]
        JECXZ   L2
        MOV     ESI, [A]
        MOV     EDI, [B]
        CLD
L1:     LODSD
        NEG     EAX
        STOSD
        LOOP    L1
L2:     POP     EDI
        POP     ESI
        RET                         ; (no extra pop if _cdecl calling convention)
_ChangeSign     ENDP
```

This looks like a nice solution, but it is not optimal because it uses slow non-pairable instructions. It takes 11 clock cycles per iteration if all data are in the level one cache.

Using pairable instructions only (PPlain and PMMX)

Example 1.2:

```
        MOV     ECX, [N]
        MOV     ESI, [A]
        TEST    ECX, ECX
        JZ      SHORT L2
        MOV     EDI, [B]
L1:     MOV     EAX, [ESI]       ; u
        XOR     EBX, EBX         ; v (pairs)
        ADD     ESI, 4           ; u
        SUB     EBX, EAX         ; v (pairs)
        MOV     [EDI], EBX       ; u
        ADD     EDI, 4           ; v (pairs)
        DEC     ECX              ; u
        JNZ     L1               ; v (pairs)
L2:
```

Here we have used pairable instructions only, and scheduled the instructions so that everything pairs. It now takes only 4 clock cycles per iteration. We could have obtained the same speed without splitting the NEG instruction, but the other unpairable instructions should be split up.

Using the same register for counter and index

Example 1.3:

```
        MOV     ESI, [A]
        MOV     EDI, [B]
        MOV     ECX, [N]
        XOR     EDX, EDX
        TEST    ECX, ECX
        JZ      SHORT L2
L1:     MOV     EAX, [ESI+4*EDX]          ; u
        NEG     EAX                       ; u
        MOV     [EDI+4*EDX], EAX          ; u
        INC     EDX                       ; v (pairs)
        CMP     EDX, ECX                  ; u
        JB      L1                        ; v (pairs)
L2:
```

Using the same register for counter and index gives us fewer instructions in the body of the loop, but it still takes 4 clocks because we have two unpaired instructions.

Letting the counter end at zero (PPlain and PMMX)

We want to get rid of the `CMP` instruction in example 1.3 by letting the counter end at zero and use the zero flag for detecting when we are finished as we did in example 1.2. One way to do this would be to execute the loop backwards taking the last array elements first. However, data caches are optimized for accessing data forwards, not backwards, so if cache misses are likely, then you should rather start the counter at -N and count through negative values up to zero. The base registers should then point to the end of the arrays rather than the beginning:

Example 1.4:

```
        MOV     ESI, [A]
        MOV     EAX, [N]
        MOV     EDI, [B]
        XOR     ECX, ECX
        LEA     ESI, [ESI+4*EAX]          ; point to end of array A
        SUB     ECX, EAX                  ; -N
        LEA     EDI, [EDI+4*EAX]          ; point to end of array B
        JZ      SHORT L2
L1:     MOV     EAX, [ESI+4*ECX]          ; u
        NEG     EAX                       ; u
        MOV     [EDI+4*ECX], EAX          ; u
        INC     ECX                       ; v (pairs)
        JNZ     L1                        ; u
L2:
```

We are now down at five instructions in the loop body but it still takes 4 clocks because of poor pairing. (If the addresses and sizes of the arrays are constants we may save two registers by substituting `A+SIZE A` for `ESI` and `B+SIZE B` for `EDI`). Now let's see how we can improve pairing.

Pairing calculations with loop overhead (PPlain and PMMX)

We may want to improve pairing by intermingling calculations with the loop control instructions. If we want to put something in between `INC ECX` and `JNZ L1`, it has to be something that doesn't affect the zero flag. The `MOV [EDI+4*ECX],EBX` instruction after `INC ECX` would generate an AGI delay, so we have to be more ingenious:

Example 1.5:

```
        MOV     EAX, [N]
        XOR     ECX, ECX
        SHL     EAX, 2                  ; 4 * N
        JZ      SHORT L3
        MOV     ESI, [A]
        MOV     EDI, [B]
        SUB     ECX, EAX                ; - 4 * N
        ADD     ESI, EAX                ; point to end of array A
        ADD     EDI, EAX                ; point to end of array B
        JMP     SHORT L2
L1:     MOV     [EDI+ECX-4], EAX        ; u
L2:     MOV     EAX, [ESI+ECX]          ; v (pairs)
        XOR     EAX, -1                 ; u
        ADD     ECX, 4                  ; v (pairs)
        INC     EAX                     ; u
        JNC     L1                      ; v (pairs)
        MOV     [EDI+ECX-4], EAX
L3:
```

I have used a different way to calculate the negative of `EAX` here: inverting all bits and adding one. The reason why I am using this method is that I can use a dirty trick with the `INC` instruction: `INC` doesn't change the carry flag, whereas `ADD` does. I am using `ADD` rather than `INC` to increment my loop counter and testing the carry flag rather than the zero flag. It is then possible to put the `INC EAX` in between without affecting the carry flag. You may think that we could have used `LEA EAX,[EAX+1]` here instead of `INC EAX`, at least that doesn't change any flags, but the `LEA` instruction would have an AGI stall so that's not the best solution. Note that the trick with the `INC` instruction not changing the carry flag is useful only on PPlain and PMMX, but will cause a partial flags stall on PPro, PII and PIII.

I have obtained perfect pairing here and the loop now takes only 3 clock cycles. Whether you want to increment the loop counter by 1 (as in example 1.4) or by 4 (as in example 1.5) is a matter of taste, it makes no difference in loop timing.

Overlapping the end of one operation with the beginning of the next (PPlain and PMMX)

The method used in example 1.5 is not very generally applicable so we may look for other methods of improving pairing opportunities. One way is to reorganize the loop so that the end of one operation overlaps with the beginning of the next. I will call this convoluting the loop. A convoluted loop has an unfinished operation at the end of each loop iteration which will be finished in the next run. Actually, example 1.5 did pair the last `MOV` of one iteration with the first `MOV` of the next, but we want to explore this method further:

Example 1.6:

```
        MOV     ESI, [A]
        MOV     EAX, [N]
        MOV     EDI, [B]
        XOR     ECX, ECX
        LEA     ESI, [ESI+4*EAX]          ; point to end of array A
        SUB     ECX, EAX                  ; -N
        LEA     EDI, [EDI+4*EAX]          ; point to end of array B
        JZ      SHORT L3
        XOR     EBX, EBX
        MOV     EAX, [ESI+4*ECX]
        INC     ECX
        JZ      SHORT L2
L1:     SUB     EBX, EAX                  ; u
        MOV     EAX, [ESI+4*ECX]          ; v (pairs)
        MOV     [EDI+4*ECX-4], EBX        ; u
        INC     ECX                       ; v (pairs)
        MOV     EBX, 0                    ; u
        JNZ     L1                        ; v (pairs)
L2:     SUB     EBX, EAX
        MOV     [EDI+4*ECX-4], EBX
L3:
```

Here we begin reading the second value before we have stored the first, and this of course improves pairing opportunities. The `MOV EBX,0` instruction has been put in between `INC ECX` and `JNZ L1` not to improve pairing but to avoid AGI stall.

Rolling out a loop (PPlain and PMMX)

The most generally applicable way to improve pairing opportunities is to do two operations for each run and do half as many runs. This is called rolling out a loop:

Example 1.7:

```
        MOV     ESI, [A]
        MOV     EAX, [N]
        MOV     EDI, [B]
        XOR     ECX, ECX
        LEA     ESI, [ESI+4*EAX]          ; point to end of array A
        SUB     ECX, EAX                  ; -N
        LEA     EDI, [EDI+4*EAX]          ; point to end of array B
        JZ      SHORT L2
        TEST    AL,1                      ; test if N is odd
        JZ      SHORT L1
        MOV     EAX, [ESI+4*ECX]          ; N is odd. do the odd one
        NEG     EAX
        MOV     [EDI+4*ECX], EAX
        INC     ECX                       ; make counter even
        JZ      SHORT L2                  ; N = 1
L1:     MOV     EAX, [ESI+4*ECX]          ; u
        MOV     EBX, [ESI+4*ECX+4]        ; v (pairs)
        NEG     EAX                       ; u
        NEG     EBX                       ; u
        MOV     [EDI+4*ECX], EAX          ; u
        MOV     [EDI+4*ECX+4], EBX        ; v (pairs)
        ADD     ECX, 2                    ; u
        JNZ     L1                        ; v (pairs)
L2:
```

Now we are doing two operations in parallel which gives the best pairing opportunities. We have to test if `N` is odd and if so do one operation outside the loop because the loop can only do an even number of operations.

The loop has an AGI stall at the first `MOV` instruction because `ECX` has been incremented in the preceding clock cycle. The loop therefore takes 6 clock cycles for two operations.

Reorganizing a loop to remove AGI stall (PPlain and PMMX)

Example 1.8:

```
        MOV     ESI, [A]
        MOV     EAX, [N]
        MOV     EDI, [B]
        XOR     ECX, ECX
        LEA     ESI, [ESI+4*EAX]          ; point to end of array A
        SUB     ECX, EAX                  ; -N
        LEA     EDI, [EDI+4*EAX]          ; point to end of array B
        JZ      SHORT L3
        TEST    AL,1                      ; test if N is odd
        JZ      SHORT L2
        MOV     EAX, [ESI+4*ECX]          ; N is odd. do the odd one
        NEG     EAX                       ; no pairing opportunity
        MOV     [EDI+4*ECX-4], EAX
        INC     ECX                       ; make counter even
        JNZ     SHORT L2
        NOP                       ; add NOP's if JNZ L2 not predictable
        NOP
        JMP     SHORT L3                  ; N = 1
L1:     NEG     EAX                       ; u
        NEG     EBX                       ; u
        MOV     [EDI+4*ECX-8], EAX        ; u
        MOV     [EDI+4*ECX-4], EBX        ; v (pairs)
L2:     MOV     EAX, [ESI+4*ECX]          ; u
        MOV     EBX, [ESI+4*ECX+4]        ; v (pairs)
        ADD     ECX, 2                    ; u
        JNZ     L1                        ; v (pairs)
        NEG     EAX
        NEG     EBX
        MOV     [EDI+4*ECX-8], EAX
        MOV     [EDI+4*ECX-4], EBX
L3:
```

The trick is to find a pair of instructions that do not use the loop counter as index and reorganize the loop so that the counter is incremented in the preceding clock cycle. We are now down at 5 clock cycles for two operations which is close to the best possible.

If data caching is critical, then you may improve the speed further by interleaving the `A` and `B` arrays into one structured array so that each `B[i]` comes immediately after the corresponding `A[i]`. If the structured array is aligned by at least 8 then `B[i]` will always be in the same cache line as `A[i]`, so you will never have a cache miss when writing `B[i]`. This may of course have a tradeoff in other parts of the program so you have to weigh the costs against the benefits.

Rolling out by more than 2 (PPlain and PMMX)

You may think of doing more than two operations per iteration in order to reduce the loop overhead per operation. But since the loop overhead in most cases can be reduced to only one clock cycle per iteration, then rolling out the loop by 4 rather than by 2 would only save 1/4 clock cycle per operation, which is hardly worth the effort. Only if the loop overhead cannot be reduced to one clock cycle and if N is very big, should you think of unrolling by 4.

The drawbacks of excessive loop unrolling are:

1. You need to calculate N MODULO R, where R is the unrolling factor, and do N

MODULO R operations before or after the main loop in order to make the remaining number of operations divisible by R. This takes a lot of extra code and poorly predictable branches. And the loop body of course also becomes bigger.
2. A Piece of code usually takes much more time the first time it executes, and the penalty of first time execution is bigger the more code you have, especially if N is small.
3. Excessive code size makes the utilization of the code cache less effective.

Handling multiple 8 or 16 bit operands simultaneously in 32 bit registers (PPlain and PMMX)

If you need to manipulate arrays of 8 or 16 bit operands, then there is a problem with unrolled loops because you may not be able to pair two memory access operations. For example `MOV AL,[ESI] / MOV BL,[ESI+1]` will not pair if the two operands are within the same dword of memory. But there may be a much smarter method, namely to handle four bytes at a time in the same 32 bit register.

The following example adds 2 to all elements of an array of bytes:

Example 1.9:

```
        MOV     ESI, [A]         ; address of byte array
        MOV     ECX, [N]         ; number of elements in byte array
        TEST    ECX, ECX         ; test if N is 0
        JZ      SHORT L2
        MOV     EAX, [ESI]       ; read first four bytes
L1:     MOV     EBX, EAX         ; copy into EBX
        AND     EAX, 7F7F7F7FH   ; get lower 7 bits of each byte in EAX
        XOR     EBX, EAX         ; get the highest bit of each byte
        ADD     EAX, 02020202H   ; add desired value to all four bytes
        XOR     EBX, EAX         ; combine bits again
        MOV     EAX, [ESI+4]     ; read next four bytes
        MOV     [ESI], EBX       ; store result
        ADD     ESI, 4           ; increment pointer
        SUB     ECX, 4           ; decrement loop counter
        JA      L1               ; loop
L2:
```

This loop takes 5 clock cycles for every 4 bytes. The array should of course be aligned by 4. If the number of elements in the array is not divisible by four, then you may padd it in the end with a few extra bytes to make the length divisible by four. This loop will always read past the end of the array, so you should make sure the array is not placed at the end of a segment to avoid a general protection error.

Note that I have masked out the highest bit of each byte to avoid a possible carry from each byte into the next when adding. I am using `XOR` rather than `ADD` when putting in the high bit again to avoid carry.

The `ADD ESI,4` instruction could have been avoided by using the loop counter as index as in example 1.4. However, this would give an odd number of instructions in the loop body, so there would be one unpaired instruction and the loop would still take 5 clocks. Making the branch instruction unpaired would save one clock after the last operation when the branch is mispredicted, but we would have to spend an extra clock cycle in the prolog code to setup a pointer to the end of the array and calculate -N, so the two methods will be exactly equally fast. The method presented here is the simplest and shortest.

The next example finds the length of a zero-terminated string by searching for the first byte of zero. It is faster than using `REP SCASB`:

Example 1.10:

```
STRLEN  PROC    NEAR
        MOV     EAX,[ESP+4]                ; get pointer
        MOV     EDX,7
        ADD     EDX,EAX                    ; pointer+7 used in the end
        PUSH    EBX
        MOV     EBX,[EAX]                  ; read first 4 bytes
        ADD     EAX,4                      ; increment pointer
L1:     LEA     ECX,[EBX-01010101H]        ; subtract 1 from each byte
        XOR     EBX,-1                     ; invert all bytes
        AND     ECX,EBX                    ; and these two
        MOV     EBX,[EAX]                  ; read next 4 bytes
        ADD     EAX,4                      ; increment pointer
        AND     ECX,80808080H              ; test all sign bits
        JZ      L1                         ; no zero bytes, continue loop
        TEST    ECX,00008080H              ; test first two bytes
        JNZ     SHORT L2
        SHR     ECX,16                     ; not in the first 2 bytes
        ADD     EAX,2
L2:     SHL     CL,1                       ; use carry flag to avoid a branch
        POP     EBX
        SBB     EAX,EDX                    ; compute length
        RET
STRLEN  ENDP
```

Again we have used the method of overlapping the end of one operation with the beginning of the next to improve pairing. I have not unrolled the loop because it is likely to repeat relatively few times. The string should of course be aligned by 4. The code will always read past the end of the string, so the string should not be placed at the end of a segment.

The loop body has an odd number of instructions so there is one unpaired. Making the branch instruction unpaired rather than one of the other instructions has the advantage that it saves 1 clock cycle when the branch is mispredicted.

The `TEST ECX,00008080H` instruction is non-pairable. You could use the pairable instruction `OR CH,CL` here instead, but then you would have to put in a `NOP` or something to avoid the penalties of consecutive branches. Another problem with `OR CH,CL` is that it would cause a partial register stall on a PPro, PII and PIII. So I have chosen to keep the unpairable `TEST` instruction.

Handling 4 bytes simultaneously can be quite difficult. The code uses a formula which generates a nonzero value for a byte if, and only if, the byte is zero. This makes it possible to test all four bytes in one operation. This algorithm involves the subtraction of 1 from all bytes (in the `LEA` instruction). I have not masked out the highest bit of each byte before subtracting, as I did in the previous example, so the subtraction may generate a borrow to the next byte, but only if it is zero, and this is exactly the situation where we don't care what the next byte is, because we are searching forwards for the first zero. If we were searching backwards then we would have to re-read the dword after detecting a zero, and then test all four bytes to find the last zero, or use `BSWAP` to reverse the order of the bytes.

If you want to search for a byte value other than zero, then you may `XOR` all four bytes with the value you are searching for, and then use the method above to search for zero.

Loops with MMX operations (PMMX)

Handling multiple operands in the same register is easier on the MMX processors because they have special instructions and special 64 bit registers for exactly this purpose.

Returning to the problem of adding two to all bytes in an array, we may take advantage of the MMX instructions:

Example 1.11:

```
.data
ALIGN   8
ADDENTS DQ      0202020202020202h       ; specify byte to add eight times
A       DD      ?                       ; address of byte array
N       DD      ?                       ; number of iterations

.code
        MOV     ESI, [A]
        MOV     ECX, [N]
        MOVQ    MM2, [ADDENTS]
        JMP     SHORT L2
        ; top of loop
L1:     MOVQ    [ESI-8], MM0    ; store result
L2:     MOVQ    MM0, MM2        ; load addents
        PADDB   MM0, [ESI]      ; add eight bytes in one operation
        ADD     ESI, 8
        DEC     ECX
        JNZ     L1
        MOVQ    [ESI-8], MM0    ; store last result
        EMMS
```

The store instruction is moved to after the loop control instructions in order to avoid a store stall.

This loop takes 4 clocks because the `PADDB` instruction doesn't pair with `ADD ESI,8`. (A MMX instruction with memory access cannot pair with a non-MMX instruction or with another MMX instruction with memory access). We could get rid of `ADD ESI,8` by using `ECX` as index, but that would give an AGI stall.

Since the loop overhead is considerable we might want to unroll the loop:

Example 1.12:

```
.data
ALIGN   8
ADDENTS DQ      0202020202020202h       ; specify byte to add eight
times
A       DD      ?                       ; address of byte array
N       DD      ?                       ; number of iterations

.code
        MOVQ    MM2, [ADDENTS]
        MOV     ESI, [A]
        MOV     ECX, [N]
        MOVQ    MM0, MM2
        MOVQ    MM1, MM2
L3:     PADDB   MM0, [ESI]
        PADDB   MM1, [ESI+8]
        MOVQ    [ESI], MM0
        MOVQ    MM0, MM2
        MOVQ    [ESI+8], MM1
        MOVQ    MM1, MM2
        ADD     ESI, 16
        DEC     ECX
        JNZ     L3
        EMMS
```

This unrolled loop takes 6 clocks per iteration for adding 16 bytes. The `PADDB` instructions are not paired. The two threads are interleaved to avoid a store stall.

Using the MMX instructions has a high penalty if you are using floating point instructions shortly afterwards, so there may still be situations where you want to use 32 bit registers as in example 1.9.

Loops with floating point operations (PPlain and PMMX)

The methods of optimizing floating point loops are basically the same as for integer loops, although the floating point instructions are overlapping rather than pairing.

Consider the C language code:

```
int i, n;  double * X;  double * Y;  double DA;
for (i=0; i<n; i++)  Y[i] = Y[i] - DA * X[i];
```

This piece of code (called DAXPY) has been studied extensively because it is the key to solving linear equations.

Example 1.13:

```
DSIZE   = 8                                     ; data size
        MOV     EAX, [N]                        ; number of elements
        MOV     ESI, [X]                        ; pointer to X
        MOV     EDI, [Y]                        ; pointer to Y
        XOR     ECX, ECX
        LEA     ESI, [ESI+DSIZE*EAX]            ; point to end of X
        SUB     ECX, EAX                        ; -N
        LEA     EDI, [EDI+DSIZE*EAX]            ; point to end of Y
        JZ      SHORT L3                        ; test for N = 0
        FLD     DSIZE PTR [DA]
        FMUL    DSIZE PTR [ESI+DSIZE*ECX]       ; DA * X[0]
        JMP     SHORT L2                        ; jump into loop
L1:     FLD     DSIZE PTR [DA]
        FMUL    DSIZE PTR [ESI+DSIZE*ECX]       ; DA * X[i]
        FXCH                                    ; get old result
        FSTP    DSIZE PTR [EDI+DSIZE*ECX-DSIZE] ; store Y[i]
L2:     FSUBR   DSIZE PTR [EDI+DSIZE*ECX]       ; subtract from Y[i]
        INC     ECX                             ; increment index
        JNZ     L1                              ; loop
        FSTP    DSIZE PTR [EDI+DSIZE*ECX-DSIZE] ; store last result
L3:
```

Here we are using the same methods as in example 1.6: Using the loop counter as index register and counting through negative values up to zero. The end of one operation overlaps with the beginning of the next.

The interleaving of floating point operations work perfectly here: The 2 clock stall between `FMUL` and `FSUBR` is filled with the `FSTP` of the previous result. The 3 clock stall between `FSUBR` and `FSTP` is filled with the loop overhead and the first two instructions of the next operation. An AGI stall has been avoided by reading the only parameter that doesn't depend on the index in the first clock cycle after the index has been incremented.

This solution takes 6 clock cycles per operation, which is better than the unrolled solution published by Intel!

Unrolling floating point loops (PPlain and PMMX)

The DAXPY loop unrolled by 3 is quite complicated:

Example 1.14:

```
DSIZE = 8                                ; data size
IF DSIZE EQ 4
SHIFTCOUNT = 2
ELSE
SHIFTCOUNT = 3
ENDIF

        MOV     EAX, [N]                 ; number of elements
        MOV     ECX, 3*DSIZE             ; counter bias
        SHL     EAX, SHIFTCOUNT          ; DSIZE*N
        JZ      L4                       ; N = 0
        MOV     ESI, [X]                 ; pointer to X
        SUB     ECX, EAX                 ; (3-N)*DSIZE
        MOV     EDI, [Y]                 ; pointer to Y
        SUB     ESI, ECX                  ; end of pointer - bias
        SUB     EDI, ECX
        TEST    ECX, ECX
        FLD     DSIZE PTR [ESI+ECX]       ; first X
        JNS     SHORT L2                  ; less than 4 operations
L1:     ; main loop
        FMUL    DSIZE PTR [DA]
        FLD     DSIZE PTR [ESI+ECX+DSIZE]
        FMUL    DSIZE PTR [DA]
        FXCH
        FSUBR   DSIZE PTR [EDI+ECX]
        FXCH
        FLD     DSIZE PTR [ESI+ECX+2*DSIZE]
        FMUL    DSIZE PTR [DA]
        FXCH
        FSUBR   DSIZE PTR [EDI+ECX+DSIZE]
        FXCH    ST(2)
        FSTP    DSIZE PTR [EDI+ECX]
        FSUBR   DSIZE PTR [EDI+ECX+2*DSIZE]
        FXCH
        FSTP    DSIZE PTR [EDI+ECX+DSIZE]
        FLD     DSIZE PTR [ESI+ECX+3*DSIZE]
        FXCH
        FSTP    DSIZE PTR [EDI+ECX+2*DSIZE]
        ADD     ECX, 3*DSIZE
        JS      L1                        ; loop
L2:     FMUL    DSIZE PTR [DA]            ; finish leftover operation
        FSUBR   DSIZE PTR [EDI+ECX]
        SUB     ECX, 2*DSIZE              ; change pointer bias
        JZ      SHORT L3                  ; finished
        FLD     DSIZE PTR [DA]            ; start next operation
        FMUL    DSIZE PTR [ESI+ECX+3*DSIZE]
        FXCH
        FSTP    DSIZE PTR [EDI+ECX+2*DSIZE]
        FSUBR   DSIZE PTR [EDI+ECX+3*DSIZE]
        ADD     ECX, 1*DSIZE
        JZ      SHORT L3                  ; finished
        FLD     DSIZE PTR [DA]
        FMUL    DSIZE PTR [ESI+ECX+3*DSIZE]
        FXCH
        FSTP    DSIZE PTR [EDI+ECX+2*DSIZE]
        FSUBR   DSIZE PTR [EDI+ECX+3*DSIZE]
        ADD     ECX, 1*DSIZE
L3:     FSTP    DSIZE PTR [EDI+ECX+2*DSIZE]
L4:
```

The reason why I am showing you how to unroll a loop by 3 is not to recommend it, but to warn you how difficult it is! Be prepared to spend a considerable amount of time debugging and verifying your code when doing something like this. There are several problems to take care of: In most cases, you cannot remove all stalls from a floating point loop unrolled by less than 4 unless you convolute it (i.e. there are unfinished operations at the end of each run which are being finished in the next run). The last `FLD` in the main loop above is the beginning of the first operation in the next run. It would be tempting here to make a solution which reads past the end of the array and then discards the extra value in the end, as in example 1.9 and 1.10, but that is not recommended in floating point loops because the reading of the extra value might generate a denormal operand exception in case the memory position after the array doesn't contain a valid floating point number. To avoid this, we have to do at least one more operation after the main loop.

The number of operations to do outside an unrolled loop would normally be N MODULO R, where N is the number of operations, and R is the unrolling factor. But in the case of a convoluted loop, we have to do one more, i.e. (N-1) MODULO R + 1, for the abovementioned reason.

Normally, we would prefer to do the extra operations before the main loop, but here we have to do them afterwards for two reasons: One reason is to take care of the leftover operand from the convolution. The other reason is that calculating the number of extra operations requires a division if R is not a power of 2, and a division is time consuming. Doing the extra operations after the loop saves the division.

The next problem is to calculate how to bias the loop counter so that it will change sign at the right time, and adjust the base pointers so as to compensate for this bias. Finally, you have to make sure the leftover operand from the convolution is handled correctly for all values of N.

The epilog code doing 1-3 operations could have been implemented as a separate loop, but that would cost an extra branch misprediction, so the solution above is faster.

Now that I have scared you by demonstrating how difficult it is to unroll by 3, I will show you that it is much easier to unroll by 4:

Example 1.15:

```
DSIZE   = 8                             ; data size
        MOV     EAX, [N]                ; number of elements
        MOV     ESI, [X]                ; pointer to X
        MOV     EDI, [Y]                ; pointer to Y
        XOR     ECX, ECX
        LEA     ESI, [ESI+DSIZE*EAX]    ; point to end of X
        SUB     ECX, EAX                ; -N
        LEA     EDI, [EDI+DSIZE*EAX]    ; point to end of Y
        TEST    AL,1                    ; test if N is odd
        JZ      SHORT L1
        FLD     DSIZE PTR [DA]          ; do the odd operation
        FMUL    DSIZE PTR [ESI+DSIZE*ECX]
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX]
        INC     ECX                     ; adjust counter
        FSTP    DSIZE PTR [EDI+DSIZE*ECX-DSIZE]
L1:     TEST    AL,2          ; test for possibly 2 more operations
        JZ      L2
        FLD     DSIZE PTR [DA]        ; N MOD 4 = 2 or 3. Do two more
        FMUL    DSIZE PTR [ESI+DSIZE*ECX]
        FLD     DSIZE PTR [DA]
        FMUL    DSIZE PTR [ESI+DSIZE*ECX+DSIZE]
        FXCH
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX]
        FXCH
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX+DSIZE]
        FXCH
        FSTP    DSIZE PTR [EDI+DSIZE*ECX]
        FSTP    DSIZE PTR [EDI+DSIZE*ECX+DSIZE]
        ADD     ECX, 2                 ; counter is now divisible by 4
L2:     TEST    ECX, ECX
        JZ      L4                         ; no more operations
L3:     ; main loop:
        FLD     DSIZE PTR [DA]
        FLD     DSIZE PTR [ESI+DSIZE*ECX]
        FMUL    ST,ST(1)
        FLD     DSIZE PTR [ESI+DSIZE*ECX+DSIZE]
        FMUL    ST,ST(2)
        FLD     DSIZE PTR [ESI+DSIZE*ECX+2*DSIZE]
        FMUL    ST,ST(3)
        FXCH    ST(2)
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX]
        FXCH    ST(3)
        FMUL    DSIZE PTR [ESI+DSIZE*ECX+3*DSIZE]
        FXCH
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX+DSIZE]
        FXCH    ST(2)
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX+2*DSIZE]
        FXCH
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX+3*DSIZE]
        FXCH    ST(3)
        FSTP    DSIZE PTR [EDI+DSIZE*ECX]
        FSTP    DSIZE PTR [EDI+DSIZE*ECX+2*DSIZE]
        FSTP    DSIZE PTR [EDI+DSIZE*ECX+DSIZE]
        FSTP    DSIZE PTR [EDI+DSIZE*ECX+3*DSIZE]
        ADD     ECX, 4                          ; increment index by 4
        JNZ     L3                              ; loop
L4:
```

It is usually quite easy to find a stall-free solution when unrolling by 4, and there is no need for convolution. The number of extra operations to do outside the main loop is N MODULO 4, which can be calculated easily without division, simply by testing the two lowest bits in N. The extra operations are done before the main loop rather than after, to make the handling of the loop counter simpler.

The tradeoff of loop unrolling is that the extra operations outside the loop are slower due to incomplete overlapping and possible branch mispredictions, and the first time penalty is higher because of increased code size.

As a general recommendation, I would say that if N is big or if convoluting the loop without unrolling cannot remove enough stalls, then you should unroll critical integer loops by 2 and floating point loops by 4.

25.2 Loops in PPro, PII and PIII

In the previous chapter (25.1) I explained how to use convolution and loop unrolling in order to improve pairing in PPlain and PMMX. On the PPro, PII and PIII there is no reason to do this thanks to the out-of-order execution mechanism. But there are other quite difficult problems to take care of, most importantly ifetch boundaries and register read stalls.

I have chosen the same example as in chapter 25.1 for the previous microprocessors: a procedure which reads integers from an array, changes the sign of each integer, and stores the results in another array.

A C language code for this procedure would be:

```
void ChangeSign (int * A, int * B, int N) {
  int i;
  for (i=0; i<N; i++) B[i] = -A[i];}
```

Translating to assembly, we might write the procedure like this:

Example 2.1:

```
_ChangeSign PROC NEAR
        PUSH    ESI
        PUSH    EDI
A       EQU     DWORD PTR [ESP+12]
B       EQU     DWORD PTR [ESP+16]
N       EQU     DWORD PTR [ESP+20]

        MOV     ECX, [N]
        JECXZ   L2
        MOV     ESI, [A]
        MOV     EDI, [B]
        CLD
L1:     LODSD
        NEG     EAX
        STOSD
        LOOP    L1
L2:     POP     EDI
        POP     ESI
        RET
_ChangeSign     ENDP
```

This looks like a nice solution, but it is not optimal because it uses the non-optimal instructions `LOOP, LODSD` and `STOSD` that generate many uops. It takes 6-7 clock cycles per iteration if all data are in the level one cache. Avoiding these instructions we get:

Example 2.2:

```
        MOV     ECX, [N]
        JECXZ   L2
        MOV     ESI, [A]
        MOV     EDI, [B]
ALIGN   16
L1:     MOV     EAX, [ESI]        ; len=2, p2rESIwEAX
        ADD     ESI, 4            ; len=3, p01rwESIwF
        NEG     EAX               ; len=2, p01rwEAXwF
        MOV     [EDI], EAX        ; len=2, p4rEAX, p3rEDI
        ADD     EDI, 4            ; len=3, p01rwEDIwF
        DEC     ECX               ; len=1, p01rwECXwF
        JNZ     L1                ; len=2, p1rF
L2:
```

The comments are interpreted as follows: the `MOV EAX,[ESI]` instruction is 2 bytes long, it generates one uop for port 2 that reads `ESI` and writes to (renames) `EAX`. This information is needed for analyzing the possible bottlenecks.

Let's first analyze the instruction decoding (chapter 14): One of the instructions generates 2 uops (`MOV [EDI],EAX`). This instruction must go into decoder D0. There are three decode groups in the loop so it can decode in 3 clock cycles.

Next, let's look at the instruction fetch (chapter 15): If an ifetch boundary prevents the first three instructions from decoding together then there will be three decode groups in the last ifetch block so that the next iteration will have the ifetch block starting at the first instruction where we want it, and we will get a delay only in the first iteration. A worse situation would be a 16-byte boundary and an ifetch boundary in one of the last three instructions. According to the ifetch table, this will generate a delay of 1 clock and cause the next iteration to have its first ifetch block aligned by 16, so that the problem continues through all iterations. The result is a fetch time of 4 clocks per iteration rather than 3. There are two ways to prevent this situation: the first method is to control where the ifetch blocks lie on the first iteration; the second method is to control where the 16-byte boundaries are. The latter method is the easiest. Since the entire loop has only 15 bytes of code you can avoid any 16-byte boundary by aligning the loop entry by 16, as shown above. This will put the entire loop into a single ifetch block so that no further analysis of instruction fetching is needed.

The third problem to look at is register read stalls (chapter 16). No register is read in this loop without being written to at least a few clock cycles before, so there can be no register read stalls.

The fourth analysis is execution (chapter 17). Counting the uops for the different ports we get:
port 0 or 1: 4 uops
port 1 only: 1 uop
port 2: 1 uop
port 3: 1 uop
port 4: 1 uop
Assuming that the uops that can go to either port 0 or 1 are distributed optimally, the execution time will be 2.5 clocks per iteration.

The last analysis is retirement (chapter 18). Since the number of uops in the loop is not divisible by 3, the retirement slots will not be used optimally when the jump has to retire in the first slot. The time needed for retirement is the number of uops divided by 3, and rounded up to nearest integer. This gives 3 clocks for retirement.

In conclusion, the loop above can execute in 3 clocks per iteration if the loop entry is aligned by 16. I am assuming that the conditional jump is predicted every time except on the exit of the loop (chapter 22.2).

Using the same register for counter and index and letting the counter end at zero (PPro, PII and PIII)

Example 2.3:

```
        MOV     ECX, [N]
        MOV     ESI, [A]
        MOV     EDI, [B]
        LEA     ESI, [ESI+4*ECX]         ; point to end of array A
        LEA     EDI, [EDI+4*ECX]         ; point to end of array B
        NEG     ECX                      ; -N
        JZ      SHORT L2
ALIGN   16
L1:     MOV     EAX, [ESI+4*ECX]         ; len=3, p2rESIrECXwEAX
        NEG     EAX                      ; len=2, p01rwEAXwF
        MOV     [EDI+4*ECX], EAX         ; len=3, p4rEAX, p3rEDIrECX
        INC     ECX                      ; len=1, p01rwECXwF
        JNZ     L1                       ; len=2, p1rF
L2:
```

Here we have reduced the number of uops to 6 by using the same register as counter and index. The base pointers point to the end of the arrays so that the index can count up through negative values to zero.

Decoding: There are two decode groups in the loop so it will decode in 2 clocks.

Instruction fetch: A loop always takes at least one clock cycle more than the the number of 16 byte blocks. Since there are only 11 bytes of code in the loop it is possible to have it all in one ifetch block. By aligning the loop entry by 16 we can make sure that we don't get more than one 16-byte block so that it is possible to fetch in 2 clocks.

Register read stalls: The `ESI` and `EDI` registers are read, but not modified inside the loop. They will therefore be counted as permanent register reads, but not in the same triplet. Register `EAX, ECX`, and flags are modified inside the loop and read before they are written back so they will cause no permanent register reads. The conclusion is that there are no register read stalls.

Execution:
port 0 or 1: 2 uops
port 1: 1 uop
port 2: 1 uop
port 3: 1 uop
port 4: 1 uop
Execution time: 1.5 clocks.

Retirement:
6 uops = 2 clocks.

Conclusion: this loop takes only 2 clock cycles per iteration.

If you use absolute addresses instead of `ESI` and `EDI` then the loop will take 3 clocks because it cannot be contained in a single 16-byte block.

Unrolling a loop (PPro, PII and PIII)

Doing more than one operation in each run and doing correspondingly fewer runs is called loop unrolling. In previous processors you would unroll loops to get parallel execution by pairing (chapter 25.1). In PPro, PII and PIII this is not needed because the out-of-order execution mechanism takes care of that. There is no need to use two different registers either, because register renaming takes care of this. The purpose of unrolling here is to reduce the loop overhead per iteration.

The following example is the same as example 2.2 , but unrolled by 2, which means that you do two operations per iteration and half as many iterations

Example 2.4:

```
        MOV     ECX, [N]
        MOV     ESI, [A]
        MOV     EDI, [B]
        SHR     ECX, 1            ; N/2
        JNC     SHORT L1          ; test if N was odd
        MOV     EAX, [ESI]        ; do the odd one first
        ADD     ESI, 4
        NEG     EAX
        MOV     [EDI], EAX
        ADD     EDI, 4
L1:     JECXZ   L3

ALIGN   16
L2:     MOV     EAX, [ESI]        ; len=2, p2rESIwEAX
        NEG     EAX               ; len=2, p01rwEAXwF
        MOV     [EDI], EAX        ; len=2, p4rEAX, p3rEDI
        MOV     EAX, [ESI+4]      ; len=3, p2rESIwEAX
        NEG     EAX               ; len=2, p01rwEAXwF
        MOV     [EDI+4], EAX      ; len=3, p4rEAX, p3rEDI
        ADD     ESI, 8            ; len=3, p01rwESIwF
        ADD     EDI, 8            ; len=3, p01rwEDIwF
        DEC     ECX               ; len=1, p01rwECXwF
        JNZ     L2                ; len=2, p1rF
L3:
```

In example 2.2 the loop overhead (i.e. adjusting pointers and counter, and jumping back) was 4 uops and the 'real job' was 4 uops. When unrolling the loop by two you do the 'real job' twice and the overhead once, so you get 12 uops in all. This reduces the overhead from 50% to 33% of the uops. Since the unrolled loop can do only an even number of operations you have to check if N is odd and if so do one operation outside the loop.

Analyzing instruction fetching in this loop we find that a new ifetch block begins in the `ADD ESI,8` instruction, forcing it into decoder D0. This makes the loop decode in 5 clock cycles and not 4 as we wanted. We can solve this problem by coding the preceding instruction in a longer version. Change `MOV [EDI+4],EAX` to:

```
    MOV [EDI+9999],EAX     ; make instruction with long displacement
    ORG $-4
    DD 4                   ; rewrite displacement to 4
```

This will force a new ifetch block to begin at the long `MOV [EDI+4],EAX` instruction, so that decoding time is now down at 4 clocks. The rest of the pipeline can handle 3 uops per clock so that the expected execution time is 4 clocks per iteration, or 2 clocks per operation.

Testing this solution shows that it actually takes a little more. My measurements showed approximately 4.5 clocks per iteration. This is probably due to a sub-optimal reordering of the uops. Possibly, the ROB doesn't find the optimal execution-order for the uops but submits them in a less than optimal order. This problem was not predicted, and only testing can reveal such a problem. We may help the ROB by doing some of the reordering manually:

Example 2.5:

```
ALIGN   16
L2:     MOV     EAX, [ESI]        ; len=2, p2rESIwEAX
        MOV     EBX, [ESI+4]      ; len=3, p2rESIwEBX
        NEG     EAX               ; len=2, p01rwEAXwF
        MOV     [EDI], EAX        ; len=2, p4rEAX, p3rEDI
        ADD     ESI, 8            ; len=3, p01rwESIwF
        NEG     EBX               ; len=2, p01rwEBXwF
        MOV     [EDI+4], EBX      ; len=3, p4rEBX, p3rEDI
        ADD     EDI, 8            ; len=3, p01rwEDIwF
        DEC     ECX               ; len=1, p01rwECXwF
        JNZ     L2                ; len=2, p1rF
L3:
```

The loop now executes in 4 clocks per iteration. This solution also solves the problem with instruction fetch blocks. The cost is that we need an extra register because we cannot take advantage of register renaming.

Rolling out by more than 2

Loop unrolling is recommended when the loop overhead constitutes a high proportion of the total execution time. In example 2.3 the overhead is only 2 uops, so the gain by unrolling is little, but I will show you how to unroll it anyway, just for the exercise.

The 'real job' is 4 uops and the overhead 2. Unrolling by two we get 2*4+2 = 10 uops. The retirement time will be 10/3, rounded up to an integer, that is 4 clock cycles. This calculation shows that nothing is gained by unrolling this by two. Unrolling by four we get:

Example 2.6:

```
        MOV     ECX, [N]
        SHL     ECX, 2                      ; number of bytes to handle
        MOV     ESI, [A]
        MOV     EDI, [B]
        ADD     ESI, ECX                    ; point to end of array A
        ADD     EDI, ECX                    ; point to end of array B
        NEG     ECX                         ; -4*N
        TEST    ECX, 4                      ; test if N is odd
        JZ      SHORT L1
        MOV     EAX, [ESI+ECX]              ; N is odd. do the odd one
        NEG     EAX
        MOV     [EDI+ECX], EAX
        ADD     ECX, 4
L1:     TEST    ECX, 8                      ; test if N/2 is odd
        JZ      SHORT L2
        MOV     EAX, [ESI+ECX]              ; N/2 is odd. do two extra
        NEG     EAX
        MOV     [EDI+ECX], EAX
        MOV     EAX, [ESI+ECX+4]
        NEG     EAX
        MOV     [EDI+ECX+4], EAX
        ADD     ECX, 8
L2:     JECXZ   SHORT L4

ALIGN   16
L3:     MOV     EAX, [ESI+ECX]              ; len=3, p2rESIrECXwEAX
        NEG     EAX                         ; len=2, p01rwEAXwF
        MOV     [EDI+ECX], EAX              ; len=3, p4rEAX, p3rEDIrECX
        MOV     EAX, [ESI+ECX+4]            ; len=4, p2rESIrECXwEAX
        NEG     EAX                         ; len=2, p01rwEAXwF
        MOV     [EDI+ECX+4], EAX            ; len=4, p4rEAX, p3rEDIrECX
        MOV     EAX, [ESI+ECX+8]            ; len=4, p2rESIrECXwEAX
        MOV     EBX, [ESI+ECX+12]           ; len=4, p2rESIrECXwEAX
        NEG     EAX                         ; len=2, p01rwEAXwF
        MOV     [EDI+ECX+8], EAX            ; len=4, p4rEAX, p3rEDIrECX
        NEG     EBX                         ; len=2, p01rwEAXwF
        MOV     [EDI+ECX+12], EBX           ; len=4, p4rEAX, p3rEDIrECX
        ADD     ECX, 16                     ; len=3, p01rwECXwF
        JS      L3                          ; len=2, p1rF
L4:
```

The ifetch blocks are where we want them. Decode time is 6 clocks.

Register read stalls is a problem here because `ECX` has retired near the end of the loop and we need to read both `ESI, EDI,` and `ECX`. The instructions have been reordered in order to avoid reading ESI near the bottom so that we can avoid a register read stall. In other words, the reason for reordering instructions and use an extra register (`EBX`) is not the same as in the previous example.

There are 12 uops and the loop executes in 6 clocks per iteration, or 1.5 clocks per operation.

It may be tempting to unroll loops by a high factor in order to get the maximum speed. But since the loop overhead in most cases can be reduced to something like one clock cycle per iteration then unrolling the loop by 4 rather than by 2 would save only 1/4 clock cycle per operation which is hardly worth the effort. Only if the loop overhead is high compared to the rest of the loop and N is very big should you think of unrolling by 4. Unrolling by more than 4 does not make sense.

The drawbacks of excessive loop unrolling are:

1. You need to calculate N MODULO R, where R is the unrolling factor, and do N MODULO R operations before or after the main loop in order to make the remaining number of operations divisible by R. This takes a lot of extra code and poorly predictable branches. And the loop body of course also becomes bigger.
2. A Piece of code usually takes much more time the first time it executes, and the penalty of first time execution is bigger the more code you have, especially if N is small.
3. Excessive code size makes the utilization of the code cache less effective.

Using an unrolling factor which is not a power of 2 makes the calculation of N MODULO R quite difficult, and is generally not recommended unless N is known to be divisible by R. Example 1.14 shows how to unroll by 3.

Handling multiple 8 or 16 bit operands simultaneously in 32 bit registers (PPro, PII and PIII)

It is sometimes possible to handle four bytes at a time in the same 32 bit register. The following example adds 2 to all elements of an array of bytes:

Example 2.7:

```
        MOV     ESI, [A]         ; address of byte array
        MOV     ECX, [N]         ; number of elements in byte array
        JECXZ   L2
ALIGN   16
        DB   7  DUP (90H)        ; 7 NOP's for controlling alignment

 L1:    MOV     EAX, [ESI]       ; read four bytes
        MOV     EBX, EAX         ; copy into EBX
        AND     EAX, 7F7F7F7FH   ; get lower 7 bits of each byte in EAX
        XOR     EBX, EAX         ; get the highest bit of each byte
        ADD     EAX, 02020202H   ; add desired value to all four bytes
        XOR     EBX, EAX         ; combine bits again
        MOV     [ESI], EBX       ; store result
        ADD     ESI, 4           ; increment pointer
        SUB     ECX, 4           ; decrement loop counter
        JA      L1               ; loop
L2:
```

Note that I have masked out the highest bit of each byte to avoid a possible carry from each byte into the next one when adding. I am using `XOR` rather than `ADD` when putting in the high bit again to avoid carry. The array should of course be aligned by 4.

This loop should ideally take 4 clocks per iteration, but it takes somewhat more due to the dependency chain and difficult reordering. On PII and PIII you can do the same more effectively using MMX registers.

The next example finds the length of a zero-terminated string by searching for the first byte of zero. It is much faster than using `REPNE SCASB`:

Example 2.8:

```
_strlen PROC    NEAR
        PUSH    EBX
        MOV     EAX,[ESP+8]           ; get pointer to string
        LEA     EDX,[EAX+3]           ; pointer+3 used in the end
L1:     MOV     EBX,[EAX]             ; read first 4 bytes
        ADD     EAX,4                 ; increment pointer
        LEA     ECX,[EBX-01010101H]   ; subtract 1 from each byte
        NOT     EBX                   ; invert all bytes
        AND     ECX,EBX               ; and these two
        AND     ECX,80808080H         ; test all sign bits
        JZ      L1                    ; no zero bytes, continue loop
        MOV     EBX,ECX
        SHR     EBX,16
        TEST    ECX,00008080H         ; test first two bytes
        CMOVZ   ECX,EBX               ; shift right if not in the first 2 bytes
        LEA     EBX,[EAX+2]
        CMOVZ   EAX,EBX
        SHL     CL,1                  ; use carry flag to avoid branch
        SBB     EAX,EDX               ; compute length
        POP     EBX
        RET
_strlen ENDP
```

This loop takes 3 clocks for each iteration testing 4 bytes. The string should of course be aligned by 4. The code may read past the end of the string, so the string should not be placed at the end of a segment.

Handling 4 bytes simultaneously can be quite difficult. This code uses a formula which generates a nonzero value for a byte if, and only if, the byte is zero. This makes it possible to test all four bytes in one operation. This algorithm involves the subtraction of 1 from all bytes (in the `LEA ECX` instruction). I have not masked out the highest bit of each byte before subtracting, as I did in example 2.7, so the subtraction may generate a borrow to the next byte, but only if it is zero, and this is exactly the situation where we don't care what the next byte is, because we are searching forwards for the first zero. If we were searching backwards then we would have to re-read the dword after detecting a zero, and then test all four bytes to find the last zero, or use `BSWAP` to reverse the order of the bytes. If you want to search for a byte value other than zero, then you may `XOR` all four bytes with the value you are searching for, and then use the method above to search for zero.

Loops with MMX instructions (PII and PIII)

Using MMX instructions we can compare 8 bytes in one operation:

Example 2.9:

```
_strlen PROC    NEAR
        PUSH    EBX
        MOV     EAX,[ESP+8]
        LEA     EDX,[EAX+7]
        PXOR    MM0,MM0
L1:     MOVQ    MM1,[EAX]         ; len=3 p2rEAXwMM1
        ADD     EAX,8             ; len=3 p01rEAX
        PCMPEQB MM1,MM0           ; len=3 p01rMM0rMM1
        MOVD    EBX,MM1           ; len=3 p01rMM1wEBX
        PSRLQ   MM1,32            ; len=4 p1rMM1
        MOVD    ECX,MM1           ; len=3 p01rMM1wECX
        OR      ECX,EBX           ; len=2 p01rECXrEBXwF
        JZ      L1                ; len=2 p1rF
        MOVD    ECX,MM1
        TEST    EBX,EBX
        CMOVZ   EBX,ECX
        LEA     ECX,[EAX+4]
        CMOVZ   EAX,ECX
        MOV     ECX,EBX
        SHR     ECX,16
        TEST    BX,BX
        CMOVZ   EBX,ECX
        LEA     ECX,[EAX+2]
        CMOVZ   EAX,ECX
        SHR     BL,1
        SBB     EAX,EDX
        EMMS
        POP     EBX
        RET
_strlen ENDP
```

This loop has 7 uops for port 0 and 1 which gives an average execution time of 3.5 clocks per iteration. The measured time is 3.8 clocks which shows that the ROB handles the situation reasonably well despite a dependency chain that is 6 uops long. Testing 8 bytes in less than 4 clocks is incredibly much faster than `REPNE SCASB`.

Loops with floating point instructions (PPro, PII and PIII)

The methods for optimizing floating point loops are basically the same as for integer loops, but you should be more aware of dependency chains because of the long latencies of instruction execution.

Consider the C language code:

```
  int i, n;  double * X;  double * Y;  double DA;
  for (i=0; i<n; i++)  Y[i] = Y[i] - DA * X[i];
```

This piece of code (called DAXPY) has been studied extensively because it is the key to solving linear equations.

Example 2.10:

```
DSIZE   = 8                      ; data size (4 or 8)
        MOV     ECX, [N]         ; number of elements
        MOV     ESI, [X]         ; pointer to X
        MOV     EDI, [Y]         ; pointer to Y
        JECXZ   L2               ; test for N = 0
        FLD     DSIZE PTR [DA]   ; load DA outside loop
ALIGN   16
        DB    2 DUP (90H)        ; 2 NOP's for alignment
L1:     FLD     DSIZE PTR [ESI]  ; len=3 p2rESIwST0
        ADD     ESI,DSIZE        ; len=3 p01rESI
        FMUL    ST,ST(1)         ; len=2 p0rST0rST1
        FSUBR   DSIZE PTR [EDI]  ; len=3 p2rEDI, p0rST0
        FSTP    DSIZE PTR [EDI]  ; len=3 p4rST0, p3rEDI
        ADD     EDI,DSIZE        ; len=3 p01rEDI
        DEC     ECX              ; len=1 p01rECXwF
        JNZ     L1               ; len=2 p1rF
        FSTP    ST               ; discard DA
L2:
```

The dependency chain is 10 clock cycles long, but the loop takes only 4 clocks per iteration because it can begin a new operation before the previous one is finished. The purpose of the alignment is to prevent a 16-byte boundary in the last ifetch block.

Example 2.11:

```
DSIZE   = 8                                ; data size (4 or 8)
        MOV     ECX, [N]                   ; number of elements
        MOV     ESI, [X]                   ; pointer to X
        MOV     EDI, [Y]                   ; pointer to Y
        LEA     ESI, [ESI+DSIZE*ECX]       ; point to end of array
        LEA     EDI, [EDI+DSIZE*ECX]       ; point to end of array
        NEG     ECX                        ; -N
        JZ      SHORT L2                   ; test for N = 0
        FLD     DSIZE PTR [DA]             ; load DA outside loop
ALIGN   16
L1:     FLD     DSIZE PTR [ESI+DSIZE*ECX]  ; len=3 p2rESIrECXwST0
        FMUL    ST,ST(1)                   ; len=2 p0rST0rST1
        FSUBR   DSIZE PTR [EDI+DSIZE*ECX]  ; len=3 p2rEDIrECX, p0rST0
        FSTP    DSIZE PTR [EDI+DSIZE*ECX]  ; len=3 p4rST0, p3rEDIrECX
        INC     ECX                        ; len=1 p01rECXwF
        JNZ     L1                         ; len=2 p1rF
        FSTP    ST                         ; discard DA
L2:
```

Here we have used the same trick as in example 2.3. Ideally, this loop should take 3 clocks, but measurements say approximately 3.5 clocks due to the long dependency chain. Unrolling the loop doesn't save much.

Loops with XMM instructions (PIII)

The XMM instructions on the PIII allow you to operate on four single precision floating point numbers in parallel. The operands must be aligned by 16.

The DAXPY algorithm is not very suited for XMM instructions because the precision is poor, it may not be possible to align the operands by 16, and you need some extra code if the number of operations is not a multiple of four. I am showing the code here anyway, just to give an example of a loop with XMM instructions:

Example 2.12:

```
        MOV     ECX, [N]                    ; number of elements
        MOV     ESI, [X]                    ; pointer to X
        MOV     EDI, [Y]                    ; pointer to Y
        SHL     ECX, 2
        ADD     ESI, ECX                    ; point to end of X
        ADD     EDI, ECX                    ; point to end of Y
        NEG     ECX                         ; -4*N
        MOV     EAX, [DA]                   ; load DA outside loop
        XOR     EAX, 80000000H              ; change sign of DA
        PUSH    EAX
        MOVSS   XMM1, [ESP]                 ; -DA
        ADD     ESP, 4
        SHUFPS  XMM1, XMM1, 0               ; copy -DA to all four positions
        CMP     ECX, -16
        JG      L2
L1:     MOVAPS  XMM0, [ESI+ECX]             ; len=4 2*p2rESIrECXwXMM0
        ADD     ECX, 16                     ; len=3 p01rwECXwF
        MULPS   XMM0, XMM1                  ; len=3 2*p0rXMM0rXMM1
        CMP     ECX, -16                    ; len=3 p01rECXwF
        ADDPS   XMM0, [EDI+ECX-16]          ; len=5 2*p2rEDIrECX, 2*p1rXMM0
        MOVAPS  [EDI+ECX-16], XMM0          ; len=5 2*p4rXMM0, 2*p3rEDIrECX
        JNG     L1                          ; len=2 p1rF
L2:     JECXZ   L4                          ; check if finished
        MOVAPS  XMM0, [ESI+ECX]             ; 1-3 operations missing, do 4 more
        MULPS   XMM0, XMM1
        ADDPS   XMM0, [EDI+ECX]
        CMP     ECX, -8
        JG      L3
        MOVLPS  [EDI+ECX], XMM0             ; store two more results
        ADD     ECX, 8
        MOVHLPS XMM0, XMM0
L3:     JECXZ   L4
        MOVSS   [EDI+ECX], XMM0             ; store one more result
L4:
```

The `L1` loop takes 5-6 clocks for 4 operations. The `ECX` instructions have been placed before and after the `MULPS XMM0, XMM1` instruction in order to avoid a register read port stall generated by the reading of the two parts of the `XMM1` register together with `ESI` or `EDI` in the RAT. The extra code after `L2` takes care of the situation where N is not divisible by 4. Note that this code may read past the end of A and B. This may delay the last operation if the extra memory positions read do not contain normal floating point numbers. If possible, put in some dummy extra data to make the number of operations divisible by 4 and leave out the extra code after `L2`.

26. Problematic Instructions

26.1 XCHG (all processors)

The `XCHG register,[memory]` instruction is dangerous. By default this instruction has an implicit `LOCK` prefix which prevents it from using the cache. This instruction is therefore very time consuming, and should always be avoided.

26.2 Rotates through carry (all processors)

`RCR` and `RCL` with a count different from one are slow and should be avoided.

26.3 String instructions (all processors)

String instructions without a repeat prefix are too slow and should be replaced by simpler instructions. The same applies to `LOOP` on all processors and to `JECXZ` on PPlain and PMMX.

`REP MOVSD` and `REP STOSD` are quite fast if the repeat count is not too small. Always use the DWORD version if possible, and make sure that both source and destination are aligned by 8.

Some other methods of moving data are faster under certain conditions. See chapter 27.8 for details.

Note that while the `REP MOVS` instruction writes a word to the destination, it reads the next word from the source in the same clock cycle. You can have a cache bank conflict if bit 2-4 are the same in these two addresses. In other words, you will get a penalty of one clock extra per iteration if `ESI+(wordsize)-EDI` is divisible by 32. The easiest way to avoid cache bank conflicts is to use the DWORD version and align both source and destination by 8. Never use `MOVSB` or `MOVSW` in optimized code, not even in 16 bit mode.

`REP MOVS` and `REP STOS` can perform very fast by moving an entire cache line at a time on PPro, PII and PIII. This happens only when the following conditions are met:

- both source and destination must be aligned by 8
- direction must be forward (direction flag cleared)
- the count (`ECX`) must be greater than or equal to 64
- the difference between `EDI` and `ESI` must be numerically greater than or equal to 32
- the memory type for both source and destination must be either writeback or write-combining (you can normally assume this).

Under these conditions the number of uops issued is approximately 215+2*`ECX` for `REP MOVSD` and 185+1.5*`ECX` for `REP STOSD,` giving a speed of approximately 5 bytes per clock cycle for both instructions, which is almost 3 times as fast as when the above conditions are not met.

The byte and word versions also benefit from this fast mode, but they are less effective than the dword versions.

`REP STOSD` is optimal under the same conditions as `REP MOVSD`.

`REP LOADS, REP SCAS,` and `REP CMPS` are not optimal, and may be replaced by loops. See example 1.10, 2.8 and 2.9 for alternatives to `REPNE SCASB. REP CMPS` may suffer cache bank conflicts if bit 2-4 are the same in `ESI` and `EDI`.

26.4 Bit test (all processors)

`BT, BTC, BTR`, and `BTS` instructions should preferably be replaced by instructions like `TEST, AND, OR, XOR`, or shifts on PPlain and PMMX. On PPro, PII and PIII, bit tests with a memory operand should be avoided.

26.5 Integer multiplication (all processors)

An integer multiplication takes approximately 9 clock cycles on PPlain and PMMX and 4 on PPro, PII and PIII. It is therefore often advantageous to replace a multiplication by a constant with a combination of other instructions such as `SHL, ADD, SUB`, and `LEA`.
Example:
`IMUL EAX,10`
can be replaced with
`MOV EBX,EAX / ADD EAX,EAX / SHL EBX,3 / ADD EAX,EBX`
or
`LEA EAX,[EAX+4*EAX] / ADD EAX,EAX`

Floating point multiplication is faster than integer multiplication on PPlain and PMMX, but the time spent on converting integers to float and converting the product back again is usually more than the time saved by using floating point multiplication, except when the number of conversions is low compared with the number of multiplications. MMX multiplication is fast, but is only available with 16-bit operands.

26.6 WAIT instruction (all processors)

You can often increase speed by omitting the `WAIT` instruction. The `WAIT` instruction has three functions:

a. The old 8087 processor requires a `WAIT` before every floating point instruction to make sure the coprocessor is ready to receive it.

b. `WAIT` is used for coordinating memory access between the floating point unit and the integer unit. Examples:

```
b.1.  FISTP [mem32]
      WAIT             ; wait for FPU to write before..
      MOV EAX,[mem32]  ; reading the result with the integer unit

b.2.  FILD [mem32]
      WAIT             ; wait for FPU to read value..
      MOV [mem32],EAX  ; before overwriting it with integer unit

b.3.  FLD QWORD PTR [ESP]
      WAIT             ; prevent an accidental interrupt from..
      ADD ESP,8        ; overwriting value on stack
```

c. `WAIT` is sometimes used to check for exceptions. It will generate an interrupt if an unmasked exception bit in the floating point status word has been set by a preceding floating point instruction.

Regarding a:
The function in point a is never needed on any other processors than the old 8087. Unless you want your code to be compatible with the 8087 you should tell your assembler not to put in these `WAIT`'s by specifying a higher processor. A 8087 floating point emulator also inserts `WAIT` instructions. You should therefore tell your assembler not to generate emulation code unless you need it.

Regarding b:
`WAIT` instructions to coordinate memory access are definitely needed on the 8087 and 80287 but not on the Pentiums. It is not quite clear whether it is needed on the 80387 and 80486. I have made several tests on these Intel processors and not been able to provoke any error by omitting the `WAIT` on any 32 bit Intel processor, although Intel manuals say that the `WAIT` is needed for this purpose except after `FNSTSW` and `FNSTCW`. Omitting `WAIT` instructions for coordinating memory access is not 100 % safe, even when writing 32 bit code, because the code may be able to run on the very rare combination of a 80386 main processor with a 287 coprocessor, which requires the `WAIT`. Also, I have no information on non-Intel processors, and I have not tested all possible hardware and software combinations, so there may be other situations where the `WAIT` is needed.

If you want to be certain that your code will work on any 32 bit processor (including non-Intel processors) then I would recommend that you include the `WAIT` here in order to be safe.

Regarding c:
The assembler automatically inserts a `WAIT` for this purpose before the following instructions: `FCLEX, FINIT, FSAVE, FSTCW, FSTENV, FSTSW`. You can omit the `WAIT` by writing FNCLEX, etc. My tests show that the WAIT is unneccessary in most cases because these instructions without `WAIT` will still generate an interrupt on exceptions except for `FNCLEX` and `FNINIT` on the 80387. (There is some inconsistency about whether the `IRET` from the interrupt points to the `FN..` instruction or to the next instruction).

Almost all other floating point instructions will also generate an interrupt if a previous floating point instruction has set an unmasked exception bit, so the exception is likely to be detected sooner or later anyway. You may insert a `WAIT` after the last floating point instruction in your program to be sure to catch all exceptions.

You may still need the `WAIT` if you want to know exactly where an exception occurred in order to be able to recover from the situation. Consider, for example, the code under b.3 above: If you want to be able to recover from an exception generated by the `FLD` here, then you need the `WAIT` because an interrupt after `ADD ESP,8` would overwrite the value to load. `FNOP` may be faster than `WAIT` and serve the same purpose.

26.7 FCOM + FSTSW AX (all processors)

The `FNSTSW` instruction is very slow on all processors. The PPro, PII and PIII processors have `FCOMI` instructions to avoid the slow `FNSTSW`. Using `FCOMI` instead of the common sequence `FCOM / FNSTSW AX / SAHF` will save you 8 clock cycles. You should therefore use `FCOMI` to avoid `FNSTSW` wherever possible, even in cases where it costs some extra code.

On processors without `FCOMI` instructions, the usual way of doing floating point comparisons is:

```
FLD [a]
FCOMP [b]
FSTSW AX
SAHF
JB ASmallerThanB
```

You may improve this code by using `FNSTSW AX` rather than `FSTSW AX` and test `AH` directly rather than using the non-pairable `SAHF` (TASM version 3.0 has a bug with the `FNSTSW AX` instruction):

```
FLD [a]
FCOMP [b]
FNSTSW AX
SHR AH,1
JC ASmallerThanB
```

Testing for zero or equality:

```
FTST
FNSTSW AX
AND AH,40H
JNZ IsZero     ; (the zero flag is inverted!)
```

Test if greater:

```
FLD [a]
FCOMP [b]
FNSTSW AX
AND AH,41H
JZ AGreaterThanB
```

Do not use `TEST AH,41H` as it is not pairable on PPlain and PMMX.

On the PPlain and PMMX, the `FNSTSW` instruction takes 2 clocks, but it is delayed for an additional 4 clocks after any floating point instruction because it is waiting for the status word to retire from the pipeline. This delay comes even after `FNOP` which cannot change the status word, but not after integer instructions. You can fill the latency between `FCOM` and `FNSTSW` with integer instructions taking up to four clock cycles. A paired `FXCH` immediately after `FCOM` doesn't delay the `FNSTSW`, not even if the pairing is imperfect:

```
FCOM                   ; clock 1
FXCH                   ; clock 1-2 (imperfect pairing)
INC DWORD PTR [EBX]    ; clock 3-5
FNSTSW AX              ; clock 6-7
```

You may want to use `FCOM` rather than `FTST` here because `FTST` is not pairable.

Remember to include the `N` in `FNSTSW`. `FSTSW` (without `N`) has a `WAIT` prefix which delays it further.

It is sometimes faster to use integer instructions for comparing floating point values, as described in chapter 27.6.

26.8 FPREM (all processors)

The `FPREM` and `FPREM1` instructions are slow on all processors. You may replace it by the following algorithm: Multiply by the reciprocal divisor, get the fractional part by subtracting the truncated value, then multiply by the divisor. (see chapter 27.5 on how to truncate)

Some documents say that these instructions may give incomplete reductions and that it is therefore necessary to repeat the `FPREM` or `FPREM1` instruction until the reduction is complete. I have tested this on several processors beginning with the old 8087 and I have found no situation where a repetition of the `FPREM` or `FPREM1` was needed.

26.9 FRNDINT (all processors)

This instruction is slow on all processors. Replace it by:

```
FISTP QWORD PTR [TEMP]
FILD  QWORD PTR [TEMP]
```

This code is faster despite a possible penalty for attempting to read from `[TEMP]` before the write is finished. It is recommended to put other instructions in between in order to avoid this penalty. See chapter 27.5 on how to truncate.

26.10 FSCALE and exponential function (all processors)

`FSCALE` is slow on all processors. Computing integer powers of 2 can be done much faster by inserting the desired power in the exponent field of the floating point number. To calculate 2^N, where N is a signed integer, select from the examples below the one that fits your range of N:

For $|N| < 2^7-1$ you can use single precision:

```
MOV     EAX, [N]
SHL     EAX, 23
ADD     EAX, 3F800000H
MOV     DWORD PTR [TEMP], EAX
FLD     DWORD PTR [TEMP]
```

For $|N| < 2^{10}-1$ you can use double precision:

```
MOV     EAX, [N]
SHL     EAX, 20
ADD     EAX, 3FF00000H
MOV     DWORD PTR [TEMP], 0
MOV     DWORD PTR [TEMP+4], EAX
FLD     QWORD PTR [TEMP]
```

For $|N| < 2^{14}-1$ use long double precision:

```
    MOV     EAX, [N]
    ADD     EAX, 00003FFFH
    MOV     DWORD PTR [TEMP],   0
    MOV     DWORD PTR [TEMP+4], 80000000H
    MOV     DWORD PTR [TEMP+8], EAX
    FLD     TBYTE PTR [TEMP]
```

`FSCALE` is often used in the calculation of exponential functions. The following code shows an exponential function without the slow `FRNDINT` and `FSCALE` instructions:

```
; extern "C" long double _cdecl exp (double x);
_exp    PROC    NEAR
PUBLIC  _exp
        FLDL2E
        FLD     QWORD PTR [ESP+4]             ; x
        FMUL                                  ; z = x*log2(e)
        FIST    DWORD PTR [ESP+4]             ; round(z)
        SUB     ESP, 12
        MOV     DWORD PTR [ESP], 0
        MOV     DWORD PTR [ESP+4], 80000000H
        FISUB   DWORD PTR [ESP+16]            ; z - round(z)
        MOV     EAX, [ESP+16]
        ADD     EAX,3FFFH
        MOV     [ESP+8],EAX
        JLE     SHORT UNDERFLOW
        CMP     EAX,8000H
        JGE     SHORT OVERFLOW
        F2XM1
        FLD1
        FADD                                  ; 2^(z-round(z))
        FLD     TBYTE PTR [ESP]               ; 2^(round(z))
        ADD     ESP,12
        FMUL                                  ; 2^z = e^x
        RET

UNDERFLOW:
        FSTP    ST
        FLDZ                                  ; return 0
        ADD     ESP,12
        RET

OVERFLOW:
        PUSH    07F800000H                    ; +infinity
        FSTP    ST
        FLD     DWORD PTR [ESP]               ; return infinity
        ADD     ESP,16
        RET

_exp    ENDP
```

26.11 FPTAN (all processors)

According to the manuals, `FPTAN` returns two values X and Y and leaves it to the programmer to divide Y with X to get the result, but in fact it always returns 1 in X so you can save the division. My tests show that on all 32 bit Intel processors with floating point unit or coprocessor, `FPTAN` always returns 1 in X regardless of the argument. If you want to be absolutely sure that your code will run correctly on all processors, then you may test if X is 1, which is faster than dividing with X. The Y value may be very high, but never infinity, so you don't have to test if Y contains a valid number if you know that the argument is valid.

26.12 FSQRT (PIII)

A fast way of calculating an approximate squareroot on the PIII is to multiply the reciprocal squareroot of x by x:
SQRT(x) = x * RSQRT(x)
The instruction `RSQRTSS` or `RSQRTPS` gives the reciprocal squareroot with a precision of 12 bits. You can improve the precision to 23 bits by using the Newton-Raphson formula described in Intel's application note AP-803:
x_0 = `RSQRTSS`(a)
$x_1 = 0.5 * x_0 * (3 - (a * x_0)) * x_0)$
where x_0 is the first approximation to the reciprocal squareroot of a, and x_1 is a better approximation. The order of evaluation is important. You must use this formula before multiplying with a to get the squareroot.

26.13 MOV [MEM], ACCUM (PPlain and PMMX)

The instructions `MOV [mem],AL   MOV [mem],AX   MOV [mem],EAX` are treated by the pairing circuitry as if they were writing to the accumulator. Thus the following instructions do not pair:

```
MOV [mydata], EAX
MOV EBX, EAX
```

This problem occurs only with the short form of the `MOV` instruction which can not have a base or index register, and which can only have the accumulator as source. You can avoid the problem by using another register, by reordering your instructions, by using a pointer, or by hard-coding the general form of the `MOV` instruction.

In 32 bit mode you can write the general form of `MOV [mem],EAX`:

```
DB 89H, 05H
DD OFFSET DS:mem
```

In 16 bit mode you can write the general form of `MOV [mem],AX`:

```
DB 89H, 06H
DW OFFSET DS:mem
```

To use `AL` instead of `(E)AX`, you replace `89H` with `88H`

This flaw has not been fixed in the PMMX.

26.14 TEST instruction (PPlain and PMMX)

The `TEST` instruction with an immediate operand is only pairable if the destination is `AL`, `AX`, or `EAX`.

`TEST register,register` and `TEST register,memory` is always pairable.

Examples:

```
    TEST ECX,ECX                 ; pairable
    TEST [mem],EBX               ; pairable
    TEST EDX,256                 ; not pairable
    TEST DWORD PTR [EBX],8000H   ; not pairable
```

To make it pairable, use any of the following methods:

```
    MOV EAX,[EBX] / TEST EAX,8000H
    MOV EDX,[EBX] / AND  EDX,8000H
    MOV AL,[EBX+1] / TEST AL,80H
    MOV AL,[EBX+1] / TEST AL,AL  ; (result in sign flag)
```

(The reason for this non-pairability is probably that the first byte of the 2-byte instruction is the same as for some other non-pairable instructions, and the processor cannot afford to check the second byte too when determining pairability.)

26.15 Bit scan (PPlain and PMMX)

`BSF` and `BSR` are the poorest optimized instructions on the PPlain and PMMX, taking approximately 11 + 2*n clock cycles, where n is the number of zeros skipped.

The following code emulates `BSR ECX,EAX`:

```
        TEST    EAX,EAX
        JZ      SHORT BS1
        MOV     DWORD PTR [TEMP],EAX
        MOV     DWORD PTR [TEMP+4],0
        FILD    QWORD PTR [TEMP]
        FSTP    QWORD PTR [TEMP]
        WAIT    ; WAIT only needed for compatibility with old 80287 processor
        MOV     ECX, DWORD PTR [TEMP+4]
        SHR     ECX,20        ; isolate exponent
        SUB     ECX,3FFH      ; adjust
        TEST    EAX,EAX       ; clear zero flag
BS1:
```

The following code emulates `BSF ECX,EAX`:

```
        TEST    EAX,EAX
        JZ      SHORT BS2
        XOR     ECX,ECX
        MOV     DWORD PTR [TEMP+4],ECX
        SUB     ECX,EAX
        AND     EAX,ECX
        MOV     DWORD PTR [TEMP],EAX
        FILD    QWORD PTR [TEMP]
        FSTP    QWORD PTR [TEMP]
        WAIT    ; WAIT only needed for compatibility with old 80287 processor
        MOV     ECX, DWORD PTR [TEMP+4]
        SHR     ECX,20
        SUB     ECX,3FFH
        TEST    EAX,EAX       ; clear zero flag
BS2:
```

These emulation codes should not be used on the PPro, PII and PIII, where the bit scan instructions take only 1 or 2 clocks, and where the emulation codes shown above have two partial memory stalls.

26.16 FLDCW (PPro, PII and PIII)

The PPro, PII and PIII have a serious stall after the `FLDCW` instruction if followed by any floating point instruction which reads the control word (which almost all floating point instructions do).

When C or C++ code is compiled it often generates a lot of `FLDCW` instructions because conversion of floating point numbers to integers is done with truncation while other floating point instructions use rounding. After translation to assembly, you can improve this code by using rounding instead of truncation where possible, or by moving the `FLDCW` out of a loop where truncation is needed inside the loop.

See chapter 27.5 on how to convert floating point numbers to integers whitout changing the control word.

27. Special topics

27.1 LEA instruction (all processors)

The `LEA` instruction is useful for many purposes because it can do a shift, two additions, and a move in just one instruction taking one clock cycle. Example:
`LEA EAX,[EBX+8*ECX-1000]`
is much faster than
`MOV EAX,ECX / SHL EAX,3 / ADD EAX,EBX / SUB EAX,1000`
The `LEA` instruction can also be used to do an add or shift without changing the flags. The source and destination need not have the same word size, so `LEA EAX,[BX]` is a possible replacement for `MOVZX EAX,BX`, although suboptimal on most processors.

You must be aware, however, that the `LEA` instruction will suffer an AGI stall on the PPlain and PMMX if it uses a base or index register which has been written to in the preceding clock cycle.

Since the `LEA` instruction is pairable in the v-pipe on PPlain and PMMX and shift instructions are not, you may use `LEA` as a substitute for a `SHL` by 1, 2, or 3 if you want the instruction to execute in the V-pipe.

The 32 bit processors have no documented addressing mode with a scaled index register and nothing else, so an instruction like `LEA EAX,[EAX*2]` is actually coded as `LEA EAX,[EAX*2+00000000]` with an immediate displacement of 4 bytes. You may reduce the instruction size by instead writing `LEA EAX,[EAX+EAX]` or even better `ADD EAX,EAX`. The latter code cannot have an AGI delay in PPlain and PMMX. If you happen to have a register which is zero (like a loop counter after a loop), then you may use it as a base register to reduce the code size:

```
LEA EAX,[EBX*4]     ; 7 bytes
LEA EAX,[ECX+EBX*4] ; 3 bytes
```

27.2 Division (all processors)

Division is quite time consuming. On PPro, PII and PIII an integer division takes 19, 23, or 39 clocks for byte, word, and dword divisors respectively. On PPlain and PMMX an unsigned integer division takes approximately the same, while a signed integer division takes somewhat more. It is therefore preferable to use the smallest operand size possible that won't generate an overflow, even if it costs an operand size prefix, and use unsigned division if possible.

Integer division by a constant (all processors)

Integer division by a power of two can be done by shifting right. Dividing an unsigned integer by 2^N:

```
        SHR     EAX, N
```

Dividing a signed integer by 2^N:

```
        CDQ
        AND     EDX, (1 SHL N) -1  ; or  SHR EDX, 32-N
        ADD     EAX, EDX
        SAR     EAX, N
```

The `SHR` alternative is shorter than the `AND` if N > 7, but can only go to execution port 0 (or u-pipe), whereas `AND` can go to either port 0 or 1 (u or v-pipe).

Dividing by a constant can be done by multiplying with the reciprocal. To calculate the unsigned integer division q = x / d, you first calculate the reciprocal of the divisor, f = 2^r / d, where r defines the position of the binary decimal point (radix point). Then multiply x with f and shift right r positions. The maximum value of r is 32+b, where b is the number of binary digits in d minus 1. (b is the highest integer for which 2^b <= d). Use r = 32+b to cover the maximum range for the value of the dividend x.

This method needs some refinement in order to compensate for rounding errors. The following algorithm will give you the correct result for unsigned integer division with truncation, i.e. the same result as the `DIV` instruction gives (Thanks to Terje Mathisen who invented this method):

```
b = (the number of significant bits in d) - 1
r = 32 + b
f = 2^r / d
If f is an integer then d is a power of 2: goto case A.
If f is not an integer, then check if the fractional part of f is < 0.5
If the fractional part of f < 0.5: goto case B.
If the fractional part of f > 0.5: goto case C.

case A: (d = 2^b)
result = x SHR b

case B: (fractional part of f < 0.5)
round f down to nearest integer
result = ((x+1) * f) SHR r

case C: (fractional part of f > 0.5)
round f up to nearest integer
result = (x * f) SHR r
```

Example:
Assume that you want to divide by 5.
5 = 00000101b.
b = (number of significant binary digits) - 1 = 2
r = 32+2 = 34
f = 2^{34} / 5 = 3435973836.8 = 0CCCCCCCC.CCC...(hexadecimal)
The fractional part is greater than a half: use case C.
Round f up to 0CCCCCCCDh.

The following code divides `EAX` by 5 and returns the result in `EDX`:

```
        MOV     EDX,0CCCCCCCDh
        MUL     EDX
        SHR     EDX,2
```

After the multiplication, `EDX` contains the product shifted right 32 places. Since r = 34 you have to shift 2 more places to get the result. To divide by 10 you just change the last line to `SHR EDX,3`.

In case B you would have:

```
        INC     EAX
        MOV     EDX,f
        MUL     EDX
        SHR     EDX,b
```

This code works for all values of x except 0FFFFFFFFH which gives zero because of overflow in the `INC` instruction. If x = 0FFFFFFFFH is possible, then change the code to:

```
        MOV     EDX,f
        ADD     EAX,1
        JC      DOVERFL
        MUL     EDX
DOVERFL:SHR     EDX,b
```

If the value of x is limited, then you may use a lower value of r, i.e. fewer digits. There can be several reasons to use a lower value of r:

- you may set r = 32 to avoid the `SHR EDX,b` in the end.
- you may set r = 16+b and use a multiplication instruction that gives a 32 bit result rather than 64 bits. This will free the `EDX` register:

  ```
  IMUL EAX,0CCCDh / SHR EAX,18
  ```

- you may choose a value of r that gives case C rather than case B in order to avoid the `INC EAX` instruction

The maximum value for x in these cases is at least 2^{r-b}, sometimes higher. You have to do a systematic test if you want to know the exact maximum value of x for which your code works correctly.

You may want to replace the slow multiplication instruction with faster instructions as explained in chapter 26.5.

The following example divides `EAX` by 10 and returns the result in `EAX`. I have chosen r=17 rather than 19 because it happens to give a code, which is easier to optimize, and covers the same range for x. f = 2^{17} / 10 = 3333h, case B: q = (x+1)*3333h:

```
        LEA     EBX,[EAX+2*EAX+3]
        LEA     ECX,[EAX+2*EAX+3]
        SHL     EBX,4
        MOV     EAX,ECX
        SHL     ECX,8
        ADD     EAX,EBX
        SHL     EBX,8
        ADD     EAX,ECX
        ADD     EAX,EBX
        SHR     EAX,17
```

A systematic test shows that this code works correctly for all x < 10004H.

Repeated integer division by the same value (all processors)

If the divisor is not known at assembly time, but you are dividing repeatedly with the same divisor, then you may use the same method as above. The code has to distinguish between case A, B and C and calculate f before doing the divisions.

The code that follows shows how to do multiple divisions with the same divisor (unsigned division with truncation). First call `SET_DIVISOR` to specify the divisor and calculate the reciprocal, then call `DIVIDE_FIXED` for each value to divide by the same divisor.

```
.data

RECIPROCAL_DIVISOR DD ?           ; rounded reciprocal divisor
CORRECTION         DD ?           ; case A: -1, case B: 1, case C: 0
BSHIFT             DD ?           ; number of bits in divisor - 1

.code

SET_DIVISOR PROC NEAR             ; divisor in EAX
        PUSH    EBX
        MOV     EBX,EAX
        BSR     ECX,EAX           ; b = number of bits in divisor - 1
        MOV     EDX,1
        JZ      ERROR             ; error: divisor is zero
        SHL     EDX,CL            ; 2^b
        MOV     [BSHIFT],ECX      ; save b
        CMP     EAX,EDX
        MOV     EAX,0
        JE      SHORT CASE_A      ; divisor is a power of 2
        DIV     EBX               ; 2^(32+b) / d
        SHR     EBX,1             ; divisor / 2
        XOR     ECX,ECX
        CMP     EDX,EBX           ; compare remainder with divisor/2
        SETBE   CL                ; 1 if case B
        MOV     [CORRECTION],ECX  ; correction for rounding errors
        XOR     ECX,1
        ADD     EAX,ECX           ; add 1 if case C
        MOV     [RECIPROCAL_DIVISOR],EAX ; rounded reciprocal divisor
        POP     EBX
        RET
CASE_A: MOV     [CORRECTION],-1   ; remember that we have case A
        POP     EBX
        RET
SET_DIVISOR     ENDP

DIVIDE_FIXED PROC NEAR                ; dividend in EAX, result in EAX
        MOV     EDX,[CORRECTION]
        MOV     ECX,[BSHIFT]
        TEST    EDX,EDX
        JS      SHORT DSHIFT          ; divisor is power of 2
        ADD     EAX,EDX               ; correct for rounding error
        JC      SHORT DOVERFL         ; correct for overflow
        MUL     [RECIPROCAL_DIVISOR]  ; multiply with reciprocal divisor
        MOV     EAX,EDX
DSHIFT: SHR     EAX,CL                ; adjust for number of bits
        RET
DOVERFL:MOV     EAX,[RECIPROCAL_DIVISOR] ; dividend = 0FFFFFFFFH
        SHR     EAX,CL                ; do division by shifting
        RET
DIVIDE_FIXED    ENDP
```

This code gives the same result as the `DIV` instruction for 0 <= x < 2^{32}, 0 < d < 2^{32}. Note: The line `JC DOVERFL` and its target are not needed if you are certain that x < 0FFFFFFFFH.

If powers of 2 occur so seldom that it is not worth optimizing for them, then you may leave out the jump to `DSHIFT` and instead do a multiplication with `CORRECTION` = 0 for case A.

If the divisor is changed so often that the procedure `SET_DIVISOR` needs optimizing, then you may replace the `BSR` instruction with the code given in chapter 26.15 for the PPlain and PMMX processors.

Floating point division (all processors)

Floating point division takes 38 or 39 clock cycles for the highest precision. You can save time by specifying a lower precision in the floating point control word (On PPlain and PMMX, only `FDIV` and `FIDIV` are faster at low precision; on PPro, PII and PIII, this also applies to `FSQRT`. No other instructions can be speeded up this way).

Parallel division (PPlain and PMMX)

On PPlain and PMMX, it is possible to do a floating point division and an integer division in parallel to save time. On PPro, PII and PIII this is not possible, because integer division and floating point division use the same circuitry.
Example: A = A1 / A2; B = B1 / B2

```
FILD    [B1]
FILD    [B2]
MOV     EAX, [A1]
MOV     EBX, [A2]
CDQ
FDIV
DIV     EBX
FISTP   [B]
MOV     [A], EAX
```

(make sure you set the floating point control word to the desired rounding method)

Using reciprocal instruction for fast division (PIII)

On PIII you can use the fast reciprocal instruction `RCPSS` or `RCPPS` on the divisor and then multiply with the dividend. However, the precision is only 12 bits. You can increase the precision to 23 bits by using the Newton-Raphson method described in Intel's application note AP-803:
x_0 = `RCPSS`(d)
$x_1 = x_0 * (2 - d * x_0) = 2*x_0 - d * x_0 * x_0$
where x_0 is the first approximation to the reciprocal of the divisor, d, and x_1 is a better approximation. You must use this formula before multiplying with the dividend:

```
MOVAPS  XMM1, [DIVISORS]        ; load divisors
RCPPS   XMM0, XMM1              ; approximate reciprocal
MULPS   XMM1, XMM0              ; Newton-Raphson formula
MULPS   XMM1, XMM0
ADDPS   XMM0, XMM0
SUBPS   XMM0, XMM1
MULPS   XMM0, [DIVIDENDS]       ; results in XMM0
```

This makes four divisions in 18 clock cycles with a precision of 23 bits. Increasing the precision further by repeating the Newton-Raphson formula in the floating point registers is possible, but not very advantageous.

If you want to use this method for integer divisions then you have to check for rounding errors. The following code makes four divisions with truncation on packed word size integers in approximately 42 clock cycles. It gives exact results for 0 <= dividend < 7FFFFH and 0 < divisor <= 7FFFFH:

```
MOVQ MM1, [DIVISORS]      ; load four divisors
MOVQ MM2, [DIVIDENDS]     ; load four dividends
PUNPCKHWD MM4, MM1        ; unpack divisors to DWORDs
PSRAD MM4, 16
PUNPCKLWD MM3, MM1
PSRAD MM3, 16
CVTPI2PS XMM1, MM4        ; convert divisors to float, upper two operands
MOVLHPS XMM1, XMM1
CVTPI2PS XMM1, MM3        ; convert lower two operands
PUNPCKHWD MM4, MM2        ; unpack dividends to DWORDs
PSRAD MM4, 16
PUNPCKLWD MM3, MM2
PSRAD MM3, 16
CVTPI2PS XMM2, MM4        ; convert dividends to float, upper two operands
MOVLHPS XMM2, XMM2
CVTPI2PS XMM2, MM3        ; convert lower two operands
RCPPS XMM0, XMM1          ; approximate reciprocal of divisors
MULPS XMM1, XMM0          ; improve precision with Newton-Raphson method
PCMPEQW MM4, MM4          ; make four integer 1's in the meantime
PSRLW MM4, 15
MULPS XMM1, XMM0
ADDPS XMM0, XMM0
SUBPS XMM0, XMM1          ; reciprocal divisors with 23 bit precision
MULPS XMM0, XMM2          ; multiply with dividends
CVTTPS2PI MM0, XMM0       ; truncate lower two results
MOVHLPS XMM0, XMM0
CVTTPS2PI MM3, XMM0       ; truncate upper two results
PACKSSDW MM0, MM3         ; pack the four results into MM0
MOVQ MM3, MM1             ; multiply results with divisors...
PMULLW MM3, MM0           ; to check for rounding errors
PADDSW MM0, MM4           ; add 1 to compensate for later subtraction
PADDSW MM3, MM1           ; add divisor. this should be > dividend
PCMPGTW MM3, MM2          ; check if too small
PADDSW MM0, MM3           ; subtract 1 if not too small
MOVQ [QUOTIENTS], MM0     ; save the four results
```

This code checks if the result is too small and makes the appropriate correction. It is not necessary to check if the result is too big.

Avoiding divisions (all processors)

Obviously, you should always try to minimize the number of divisions. Floating point division with a constant or repeated division with the same value should of course be done by multiplying with the reciprocal. But there are many other situations where you can reduce the number of divisions. For example: if (A/B > C)... can be rewritten as if (A > B*C)... when B is positive, and the opposite when B is negative.

A/B + C/D can be rewritten as (A*D + C*B) / (B*D)

If you are using integer division, then you should be aware that the rounding errors may be different when you rewrite the formulas.

27.3 Freeing floating point registers (all processors)

You have to free all used floating point registers before exiting a subroutine, except for any register used for the result.

The fastest way of freeing one register is `FSTP ST`. The fastest way of freeing two registers is `FCOMPP` on PPlain and PMMX; on PPro, PII and PIII you may use either `FCOMPP` or two times `FSTP ST`, whichever fits best into the decoding sequence.

It is not recommended to use `FFREE`.

27.4 Transitions between floating point and MMX instructions (PMMX, PII and PIII)

You must issue an `EMMS` instruction after your last MMX instruction if there is a possibility that floating point code follows later.

On PMMX there is a high penalty for switching between floating point and MMX instructions. The first floating point instruction after an `EMMS` takes approximately 58 clocks extra, and the first MMX instruction after a floating point instruction takes approximately 38 clocks extra.

On PII and PIII there is no such penalty. The delay after `EMMS` can be hidden by putting in integer instructions between `EMMS` and the first floating point instruction.

27.5 Converting from floating point to integer (All processors)

All conversions from floating point to integer, and vice versa, must go via a memory location:

```
FISTP DWORD PTR [TEMP]
MOV EAX, [TEMP]
```

On PPro, PII and PIII, this code is likely to have a penalty for attempting to read from `[TEMP]` before the write to `[TEMP]` is finished because the `FIST` instruction is slow (see chapter 17). It doesn't help to put in a `WAIT` (see chapter 26.6). It is recommended that you put in other instructions between the write to `[TEMP]` and the read from `[TEMP]` if possible in order to avoid this penalty. This applies to all the examples that follow.

The specifications for the C and C++ language requires that conversion from floating point numbers to integers use truncation rather than rounding. The method used by most C libraries is to change the floating point control word to indicate truncation before using an

`FISTP` instruction and changing it back again afterwords. This method is very slow on all processors. On PPro, PII and PIII, the floating point control word cannot be renamed, so all subsequent floating point instructions must wait for the `FLDCW` instruction to retire.

Whenever you have a conversion from floating point to integer in C or C++, you should think of whether you can use rounding to nearest integer instead of truncation. If your standard library doesn't have a fast round function then make your own using the code examples listed below.

If you need truncation inside a loop then you should change the control word only outside the loop if the rest of the floating point instructions in the loop can work correctly in truncation mode.

You may use various tricks for truncating without changing the control word, as illustrated in the examples below. These examples presume that the control word is set to default, i.e. rounding to nearest or even.

Rounding to nearest or even

```
; extern "C" int round (double x);
_round  PROC    NEAR
PUBLIC  _round
        FLD     QWORD PTR [ESP+4]
        FISTP   DWORD PTR [ESP+4]
        MOV     EAX, DWORD PTR [ESP+4]
        RET
_round  ENDP
```

Truncation towards zero

```
; extern "C" int truncate (double x);
_truncate PROC    NEAR
PUBLIC  _truncate
        FLD     QWORD PTR [ESP+4]   ; x
        SUB     ESP, 12             ; space for local variables
        FIST    DWORD PTR [ESP]     ; rounded value
        FST     DWORD PTR [ESP+4]   ; float value
        FISUB   DWORD PTR [ESP]     ; subtract rounded value
        FSTP    DWORD PTR [ESP+8]   ; difference
        POP     EAX                 ; rounded value
        POP     ECX                 ; float value
        POP     EDX                 ; difference (float)
        TEST    ECX, ECX            ; test sign of x
        JS      SHORT NEGATIVE
        ADD     EDX, 7FFFFFFFH      ; produce carry if difference < -0
        SBB     EAX, 0              ; subtract 1 if x-round(x) < -0
        RET
NEGATIVE:
        XOR     ECX, ECX
        TEST    EDX, EDX
        SETG    CL                  ; 1 if difference > 0
        ADD     EAX, ECX            ; add 1 if x-round(x) > 0
        RET
_truncate ENDP
```

Truncation towards minus infinity

```
; extern "C" int ifloor (double x);
_ifloor PROC    NEAR
PUBLIC  _ifloor
        FLD     QWORD PTR [ESP+4]   ; x
        SUB     ESP, 8              ; space for local variables
        FIST    DWORD PTR [ESP]     ; rounded value
        FISUB   DWORD PTR [ESP]     ; subtract rounded value
        FSTP    DWORD PTR [ESP+4]   ; difference
        POP     EAX                 ; rounded value
        POP     EDX                 ; difference (float)
        ADD     EDX, 7FFFFFFFH      ; produce carry if difference < -0
        SBB     EAX, 0              ; subtract 1 if x-round(x) < -0
        RET
_ifloor ENDP
```

These procedures work for $-2^{31} < x < 2^{31}-1$. They do not check for overflow or NAN's.

The PIII has instructions for truncation of single precision floating point numbers: `CVTTSS2SI` and `CVTTPS2PI`. These instructions are very useful if the single precision is satisfactory, but if you are converting a float with higher precision to single precision in order to use these truncation instructions then you have the problem that the number may be rounded up in the conversion to single precision.

Alternative to FISTP instruction (PPlain and PMMX)

Converting a floating point number to integer is normally done like this:

```
        FISTP   DWORD PTR [TEMP]
        MOV     EAX, [TEMP]
```

An alternative method is:

```
.DATA
ALIGN 8
TEMP    DQ      ?
MAGIC   DD      59C00000H   ; f.p. representation of 2^51 + 2^52

.CODE
        FADD    [MAGIC]
        FSTP    QWORD PTR [TEMP]
        MOV     EAX, DWORD PTR [TEMP]
```

Adding the 'magic number' of $2^{51} + 2^{52}$ has the effect that any integer between -2^{31} and $+2^{31}$ will be aligned in the lower 32 bits when storing as a double precision floating point number. The result is the same as you get with `FISTP` for all rounding methods except truncation towards zero. The result is different from `FISTP` if the control word specifies truncation or in case of overflow. You may need a `WAIT` instruction for compatibility with the old 80287 processor, see chapter 26.6.

This method is not faster than using `FISTP`, but it gives better scheduling opportunities on PPlain and PMMX because there is a 3 clock void between `FADD` and `FSTP` which may be filled with other instrucions. You may multiply or divide the number by a power of 2 in the same operation by doing the opposite to the magic number. You may also add a constant by adding it to the magic number, which then has to be double precision.

27.6 Using integer instructions to do floating point operations (all processors)

Integer instructions are generally faster than floating point instructions, so it is often advantageous to use integer instructions for doing simple floating point operations. The most obvious example is moving data. Example:

```
FLD QWORD PTR [ESI] / FSTP QWORD PTR [EDI]
```

Change to:

```
MOV EAX,[ESI] / MOV EBX,[ESI+4] / MOV [EDI],EAX / MOV [EDI+4],EBX
```

Testing if a floating point value is zero:

The floating point value of zero is usually represented as 32 or 64 bits of zero, but there is a pitfall here: The sign bit may be set! Minus zero is regarded as a valid floating point number, and the processor may actually generate a zero with the sign bit set if for example multiplying a negative number with zero. So if you want to test if a floating point number is zero, you should not test the sign bit. Example:

```
FLD DWORD PTR [EBX] / FTST / FNSTSW AX / AND AH,40H / JNZ IsZero
```

Use integer instructions instead, and shift out the sign bit:

```
MOV EAX,[EBX] / ADD EAX,EAX / JZ IsZero
```

If the floating point number is double precision (QWORD) then you only have to test bit 32-62. If they are zero, then the lower half will also be zero if it is a normal floating point number.

Testing if negative:

A floating point number is negative if the sign bit is set and at least one other bit is set. Example:

```
MOV EAX,[NumberToTest] / CMP EAX,80000000H / JA IsNegative
```

Manipulating the sign bit:

You can change the sign of a floating point number simply by flipping the sign bit. Example:

```
XOR BYTE PTR [a] + (TYPE a) - 1, 80H
```

Likewise you may get the absolute value of a floating point number by simply ANDing out the sign bit.

Comparing numbers:

Floating point numbers are stored in a unique format which allows you to use integer instructions for comparing floating point numbers, except for the sign bit. If you are certain that two floating point numbers both are normal and positive then you may simply compare them as integers. Example:

```
FLD [a] / FCOMP [b] / FNSTSW AX / AND AH,1 / JNZ ASmallerThanB
```

Change to:

`MOV EAX,[a] / MOV EBX,[b] / CMP EAX,EBX / JB ASmallerThanB`
This method only works if the two numbers have the same precision and you are certain that none of the numbers have the sign bit set.

If negative numbers are possible, then you have to convert the negative numbers to 2-complement, and do a signed compare:

```
        MOV     EAX, [a]
        MOV     EBX, [b]
        MOV     ECX, EAX
        MOV     EDX, EBX
        SAR     ECX, 31             ; copy sign bit
        AND     EAX, 7FFFFFFFH      ; remove sign bit
        SAR     EDX, 31
        AND     EBX, 7FFFFFFFH
        XOR     EAX, ECX      ; make 2-complement if sign bit was set
        XOR     EBX, EDX
        SUB     EAX, ECX
        SUB     EBX, EDX
        CMP     EAX, EBX
        JL      ASmallerThanB       ; signed comparison
```

This method works for all normal floating point numbers, including -0.

27.7 Using floating point instructions to do integer operations (PPlain and PMMX)

Integer multiplication (PPlain and PMMX)

Floating point multiplication is faster than integer multiplication on the PPlain and PMMX, but the price for converting integer factors to float and converting the result back to integer is high, so floating point multiplication is only advantageous if the number of conversions needed is low compared with the number of multiplications. (It may be tempting to use denormal floating point operands to save some of the conversions here, but the handling of denormals is very slow, so this is not a good idea!)

On the PMMX, MMX multiplication instructions are faster than integer multiplication, and can be pipelined to a throughput of one multiplication per clock cycle, so this may be the best solution for doing fast multiplication on the PMMX, if you can live with 16 bit precision.

Integer multiplication is faster than floating point on PPro, PII and PIII.

Integer division (PPlain and PMMX)

Floating point division is not faster than integer division, but you can do other integer operations (including integer division, but not integer multiplication) while the floating point unit is working on the division (See example above).

Converting binary to decimal numbers (all processors)

Using the `FBSTP` instruction is a simple and convenient way of converting a binary number to decimal, although not necessarily the fastest method.

27.8 Moving blocks of data (all processors)

There are several ways of moving blocks of data. The most common method is `REP MOVSD`, but under certain conditions other methods are faster.

On PPlain and PMMX it is faster to move 8 bytes at a time using floating point registers if the destination is not in the cache:

```
TOP:    FILD    QWORD PTR [ESI]
        FILD    QWORD PTR [ESI+8]
        FXCH
        FISTP   QWORD PTR [EDI]
        FISTP   QWORD PTR [EDI+8]
        ADD     ESI, 16
        ADD     EDI, 16
        DEC     ECX
        JNZ     TOP
```

The source and destination should of course be aligned by 8. The extra time used by the slow `FILD` and `FISTP` instructions is compensated for by the fact that you only have to do half as many write operations. Note that this method is only advantageous on the PPlain and PMMX and only if the destination is not in the level 1 cache. You cannot use `FLD` and `FSTP` (without `I`) on arbitrary bit patterns because denormal numbers are handled slowly and certain bit patterns are not preserved unchanged.

On the PMMX processor it is faster to use MMX instructions to move eight bytes at a time if the destination is not in the cache:

```
TOP:    MOVQ    MM0,[ESI]
        MOVQ    [EDI],MM0
        ADD     ESI,8
        ADD     EDI,8
        DEC     ECX
        JNZ     TOP
```

There is no need to unroll this loop or optimize it further if cache misses are expected, because memory access is the bottleneck here, not instruction execution.

On PPro, PII and PIII processors the `REP MOVSD` instruction is particularly fast when the following conditions are met (see chapter 26.3):

- both source and destination must be aligned by 8
- direction must be forward (direction flag cleared)
- the count (`ECX`) must be greater than or equal to 64
- the difference between `EDI` and `ESI` must be numerically greater than or equal to 32
- the memory type for both source and destination must be either writeback or write-combining (you can normally assume this).

On the PII it is faster to use MMX registers if the above conditions are not met and the destination is likely to be in the level 1 cache. The loop may be rolled out by two, and the source and destination should of course be aligned by 8.

On the PIII the fastest way of moving data is to use the `MOVAPS` instruction if the above conditions are not met or if the destination is in the level 1 or level 2 cache:

```
        SUB     EDI, ESI
TOP:    MOVAPS  XMM0, [ESI]
        MOVAPS  [ESI+EDI], XMM0
        ADD     ESI, 16
        DEC     ECX
        JNZ     TOP
```

Unlike `FLD`, `MOVAPS` can handle any bit pattern without problems. Remember that source and destination must be aligned by 16.

If the number of bytes to move is not divisible by 16 then you may round up to the nearest number divisible by 16 and put some extra space at the end of the destination buffer to receive the superfluous bytes. If this is not possible then you have to move the remaining bytes by other methods.

On the PIII you also have the option of writing directly to RAM memory without involving the cache by using the `MOVNTQ` or `MOVNTPS` instruction. This can be useful if you don't want the destination to go into a cache. `MOVNTPS` is only slightly faster than `MOVNTQ`.

27.9 Self-modifying code (All processors)

The penalty for executing a piece of code immediately after modifying it is approximately 19 clocks for PPlain, 31 for PMMX, and 150-300 for PPro, PII and PIII. The 80486 and earlier processors require a jump between the modifying and the modified code in order to flush the code cache.

To get permission to modify code in a protected operating system you need to call special system functions: In 16-bit Windows call ChangeSelector, in 32-bit Windows call VirtualProtect and FlushInstructionCache (or put the code in a data segment).

Self-modifying code is not considered good programming practice, but it may be justified if the gain in speed is considerable.

27.10 Detecting processor type (All processors)

I think it is fairly obvious by now that what is optimal for one microprocessor may not be optimal for another. You may make the most critical parts of you program in different versions, each optimized for a specific microprocessor and selecting the desired version at run time after detecting which microprocessor the program is running on. If you are using instructions that are not supported by all microprocessors (i.e. conditional moves, `FCOMI`, MMX and XMM instructions) then you must first check if the program is running on a microprocessor that supports these instructions. The subroutine below checks the type of microprocessor and the features supported.

```
; define CPUID instruction if not known by assembler:
CPUID   MACRO
        DB      0FH, 0A2H
ENDM

; C++ prototype:
; extern "C" long int DetectProcessor (void);

; return value:
; bits 8-11 = family (5 for PPlain and PMMX, 6 for PPro, PII and PIII)
; bit  0 = floating point instructions supported
; bit 15 = conditional move and FCOMI instructions supported
; bit 23 = MMX instructions supported
; bit 25 = XMM instructions supported

_DetectProcessor PROC NEAR
PUBLIC  _DetectProcessor
        PUSH    EBX
        PUSH    ESI
        PUSH    EDI
        PUSH    EBP
        ; detect if CPUID instruction supported by microprocessor:
        PUSHFD
        POP     EAX
        MOV     EBX, EAX
        XOR     EAX, 1 SHL 21    ; check if CPUID bit can toggle
        PUSH    EAX
        POPFD
        PUSHFD
        POP     EAX
        XOR     EAX, EBX
        AND     EAX, 1 SHL 21
        JZ      SHORT DPEND      ; CPUID instruction not supported
        XOR     EAX, EAX
        CPUID                    ; get number of CPUID functions
        TEST    EAX, EAX
        JZ      SHORT DPEND      ; CPUID function 1 not supported
        MOV     EAX, 1
        CPUID                    ; get family and features
        AND     EAX, 000000F00H  ; family
        AND     EDX, 0FFFFF0FFH  ; features flags
        OR      EAX, EDX         ; combine bits
DPEND:  POP     EBP
        POP     EDI
        POP     ESI
        POP     EBX
        RET
_DetectProcessor ENDP
```

Note that some operating systems do not allow XMM instructions. Information on how to check for operating system support of XMM instructions can be found in Intel's application note AP-900: "Identifying support for Streaming SIMD Extensions in the Processor and Operating System". More information on microprocessor identification can be found in Intel's application note AP-485: "Intel Processor Identification and the CPUID Instruction".

To code the conditional move, MMX, XMM instructions etc. on an assembler that doesn't have these instructions use the macros at www.agner.org/assem/macros.zip

28. List of instruction timings for PPlain and PMMX

28.1 Integer instructions

Explanations:
Operands:
r = register, m = memory, i = immediate data, sr = segment register
m32 = 32 bit memory operand, etc.

Clock cycles:
The numbers are minimum values. Cache misses, misalignment, and exceptions may increase the clock counts considerably.

Pairability:
u = pairable in u-pipe, v = pairable in v-pipe, uv = pairable in either pipe, np = not pairable.

Instruction	Operands	Clock cycles	Pairability
NOP		1	uv
MOV	r/m, r/m/i	1	uv
MOV	r/m, sr	1	np
MOV	sr , r/m	>= 2 b)	np
MOV	m , accum	1	uv h)
XCHG	(E)AX, r	2	np
XCHG	r , r	3	np
XCHG	r , m	>15	np
XLAT		4	np
PUSH	r/i	1	uv
POP	r	1	uv
PUSH	m	2	np
POP	m	3	np
PUSH	sr	1 b)	np
POP	sr	>= 3 b)	np
PUSHF		3-5	np
POPF		4-6	np
PUSHA POPA		5-9 i)	np

PUSHAD POPAD		5	np
LAHF SAHF		2	np
MOVSX MOVZX	r , r/m	3 a)	np
LEA	r , m	1	uv
LDS LES LFS LGS LSS	m	4 c)	np
ADD SUB AND OR XOR	r , r/i	1	uv
ADD SUB AND OR XOR	r , m	2	uv
ADD SUB AND OR XOR	m , r/i	3	uv
ADC SBB	r , r/i	1	u
ADC SBB	r , m	2	u
ADC SBB	m , r/i	3	u
CMP	r , r/i	1	uv
CMP	m , r/i	2	uv
TEST	r , r	1	uv
TEST	m , r	2	uv
TEST	r , i	1	f)
TEST	m , i	2	np
INC DEC	r	1	uv
INC DEC	m	3	uv
NEG NOT	r/m	1/3	np
MUL IMUL	r8/r16/m8/m16	11	np
MUL IMUL	all other versions	9 d)	np
DIV	r8/m8	17	np
DIV	r16/m16	25	np
DIV	r32/m32	41	np
IDIV	r8/m8	22	np
IDIV	r16/m16	30	np
IDIV	r32/m32	46	np
CBW CWDE		3	np
CWD CDQ		2	np

SHR SHL SAR SAL	r , i	1	u
SHR SHL SAR SAL	m , i	3	u
SHR SHL SAR SAL	r/m, CL	4/5	np
ROR ROL RCR RCL	r/m, 1	1/3	u
ROR ROL	r/m, i(><1)	1/3	np
ROR ROL	r/m, CL	4/5	np
RCR RCL	r/m, i(><1)	8/10	np
RCR RCL	r/m, CL	7/9	np
SHLD SHRD	r, i/CL	4 a)	np
SHLD SHRD	m, i/CL	5 a)	np
BT	r, r/i	4 a)	np
BT	m, i	4 a)	np
BT	m, i	9 a)	np
BTR BTS BTC	r, r/i	7 a)	np
BTR BTS BTC	m, i	8 a)	np
BTR BTS BTC	m, r	14 a)	np
BSF BSR	r , r/m	7-73 a)	np
SETcc	r/m	1/2 a)	np
JMP CALL	short/near	1 e)	v
JMP CALL	far	>= 3 e)	np
conditional jump	short/near	1/4/5/6 e)	v
CALL JMP	r/m	2/5 e	np
RETN		2/5 e	np
RETN	i	3/6 e)	np
RETF		4/7 e)	np
RETF	i	5/8 e)	np
J(E)CXZ	short	4-11 e)	np
LOOP	short	5-10 e)	np
BOUND	r , m	8	np
CLC STC CMC CLD STD		2	np

CLI STI		6-9	np
LODS		2	np
REP LODS		7+3*n g)	np
STOS		3	np
REP STOS		10+n g)	np
MOVS		4	np
REP MOVS		12+n g)	np
SCAS		4	np
REP(N)E SCAS		9+4*n g)	np
CMPS		5	np
REP(N)E CMPS		8+4*n g)	np
BSWAP		1 a)	np
CPUID		13-16 a)	np
RDTSC		6-13 a) j)	np

Notes:
a) this instruction has a `0FH` prefix which takes one clock cycle extra to decode on a PPlain unless preceded by a multicycle instruction (see chapter 12).
b) versions with `FS` and `GS` have a `0FH` prefix. see note a.
c) versions with `SS, FS`, and `GS` have a `0FH` prefix. see note a.
d) versions with two operands and no immediate have a `0FH` prefix, see note a.
e) see chapter 22
f) only pairable if register is accumulator. see chapter 26.14.
g) add one clock cycle for decoding the repeat prefix unless preceded by a multicycle instruction (such as `CLD`. see chapter 12).
h) pairs as if it were writing to the accumulator. see chapter 26.14.
i) 9 if `SP` divisible by 4. See 10.2
j) on PPlain: 6 in priviledged or real mode, 11 in nonpriviledged, error in virtual mode. On PMMX: 8 and 13 clocks respectively.

28.2 Floating point instructions

Explanations:
Operands:
r = register, m = memory, m32 = 32 bit memory operand, etc.

Clock cycles:
The numbers are minimum values. Cache misses, misalignment, denormal operands, and exceptions may increase the clock counts considerably.

Pairability:
+ = pairable with `FXCH`, np = not pairable with `FXCH`.

i-ov:
Overlap with integer instructions. i-ov = 4 means that the last four clock cycles can overlap with subsequent integer instructions.

fp-ov:
Overlap with floating point instructions. fp-ov = 2 means that the last two clock cycles can overlap with subsequent floating point instructions. (`WAIT` is considered a floating point instruction here)

Instruction	Operand	Clock cycles	Pairability	i-ov	fp-ov
FLD	r/m32/m64	1	+	0	0
FLD	m80	3	np	0	0
FBLD	m80	48-58	np	0	0
FST(P)	r	1	np	0	0
FST(P)	m32/m64	2 m)	np	0	0
FST(P)	m80	3 m)	np	0	0
FBSTP	m80	148-154	np	0	0
FILD	m	3	np	2	2
FIST(P)	m	6	np	0	0
FLDZ FLD1		2	np	0	0
FLDPI FLDL2E etc.		5 s)	np	2	2
FNSTSW	AX/m16	6 q)	np	0	0
FLDCW	m16	8	np	0	0
FNSTCW	m16	2	np	0	0
FADD(P)	r/m	3	+	2	2
FSUB(R)(P)	r/m	3	+	2	2
FMUL(P)	r/m	3	+	2	2 n)
FDIV(R)(P)	r/m	19/33/39 p)	+	38 o)	2
FCHS FABS		1	+	0	0
FCOM(P)(P) FUCOM	r/m	1	+	0	0
FIADD FISUB(R)	m	6	np	2	2
FIMUL	m	6	np	2	2

FIDIV(R)	m	22/36/42 p)	np	38 o)	2
FICOM	m	4	np	0	0
FTST		1	np	0	0
FXAM		17-21	np	4	0
FPREM		16-64	np	2	2
FPREM1		20-70	np	2	2
FRNDINT		9-20	np	0	0
FSCALE		20-32	np	5	0
FXTRACT		12-66	np	0	0
FSQRT		70	np	69 o)	2
FSIN FCOS		65-100 r)	np	2	2
FSINCOS		89-112 r)	np	2	2
F2XM1		53-59 r)	np	2	2
FYL2X		103 r)	np	2	2
FYL2XP1		105 r)	np	2	2
FPTAN		120-147 r)	np	36 o)	0
FPATAN		112-134 r)	np	2	2
FNOP		1	np	0	0
FXCH	r	1	np	0	0
FINCSTP FDECSTP		2	np	0	0
FFREE	r	2	np	0	0
FNCLEX		6-9	np	0	0
FNINIT		12-22	np	0	0
FNSAVE	m	124-300	np	0	0
FRSTOR	m	70-95	np	0	0
WAIT		1	np	0	0

Notes:
m) The value to store is needed one clock cycle in advance.
n) 1 if the overlapping instruction is also an `FMUL`.
o) Cannot overlap integer multiplication instructions.

p) `FDIV` takes 19, 33, or 39 clock cycles for 24, 53, and 64 bit precision respectively. `FIDIV` takes 3 clocks more. The precision is defined by bit 8-9 of the floating point control word.
q) The first 4 clock cycles can overlap with preceding integer instructions. See chapter 26.7.
r) clock counts are typical. Trivial cases may be faster, extreme cases may be slower.
s) may be up to 3 clocks more when output needed for `FST`, `FCHS`, or `FABS`.

28.3 MMX instructions (PMMX)

A list of MMX instruction timings is not needed because they all take one clock cycle, except the MMX multiply instructions which take 3. MMX multiply instructions can be overlapped and pipelined to yield a throughput of one multiplication per clock cycle.

The `EMMS` instruction takes only one clock cycle, but the first floating point instruction after an `EMMS` takes approximately 58 clocks extra, and the first MMX instruction after a floating point instruction takes approximately 38 clocks extra. There is no penalty for an MMX instruction after `EMMS` on the PMMX (but a possible small penalty on the PII and PIII).

There is no penalty for using a memory operand in an MMX instruction because the MMX arithmetic unit is one step later in the pipeline than the load unit. But the penalty comes when you store data from an MMX register to memory or to a 32 bit register: The data have to be ready one clock cycle in advance. This is analogous to the floating point store instructions.

All MMX instructions except `EMMS` are pairable in either pipe. Pairing rules for MMX instructions are described in chapter 10.

29. List of instruction timings and micro-op breakdown for PPro, PII and PIII

Explanations:
Operands:
r = register, m = memory, i = immediate data, sr = segment register, m32 = 32 bit memory operand, etc.

Micro-ops:
The number of micro-ops that the instruction generates for each execution port.
p0: port 0: ALU, etc.
p1: port 1: ALU, jumps
p01: instructions that can go to either port 0 or 1, whichever is vacant first.
p2: port 2: load data, etc.
p3: port 3: address generation for store
p4: port 4: store data

Delay:
This is the delay that the instruction generates in a dependency chain. (This is not the same as the time spent in the execution unit. Values may be inaccurate in situations where

they cannot be measured exactly, especially with memory operands). The numbers are minimum values. Cache misses, misalignment, and exceptions may increase the clock counts considerably. Floating point operands are presumed to be normal numbers. Denormal numbers, NANs and infinity increase the delays by 50-150 clocks, except in XMM move, shuffle and boolean instructions. Floating point overflow, underflow, denormal or NAN results give a similar delay.

Throughput:
The maximum throughput for several instructions of the same kind. For example, a throughput of 1/2 for `FMUL` means that a new `FMUL` instruction can start executing every 2 clock cycles.

29.1 Integer instructions									
Instruction	Operands	micro-ops						delay	throughput
		p0	p1	p01	p2	p3	p4		
NOP				1					
MOV	r,r/i			1					
MOV	r,m				1				
MOV	m,r/i					1	1		
MOV	r,sr			1					
MOV	m,sr			1		1	1		
MOV	sr,r	8						5	
MOV	sr,m	7			1			8	
MOVSX MOVZX	r,r			1					
MOVSX MOVZX	r,m				1				
CMOVcc	r,r	1		1					
CMOVcc	r,m	1		1	1				
XCHG	r,r			3					
XCHG	r,m			4	1	1	1	high b)	
XLAT				1	1				
PUSH	r/i			1		1	1		
POP	r			1	1				
POP	(E)SP			2	1				
PUSH	m			1	1	1	1		

POP	m			5	1	1	1		
PUSH	sr			2		1	1		
POP	sr			8	1				
PUSHF(D)		3		11		1	1		
POPF(D)		10		6	1				
PUSHA(D)				2		8	8		
POPA(D)				2	8				
LAHF SAHF				1					
LEA	r,m	1						1 c)	
LDS LES LFS LGS LSS	m			8	3				
ADD SUB AND OR XOR	r,r/i			1					
ADD SUB AND OR XOR	r,m			1	1				
ADD SUB AND OR XOR	m,r/i			1	1	1	1		
ADC SBB	r,r/i			2					
ADC SBB	r,m			2	1				
ADC SBB	m,r/i			3	1	1	1		
CMP TEST	r,r/i			1					
CMP TEST	m,r/i			1	1				
INC DEC NEG NOT	r			1					
INC DEC NEG NOT	m			1	1	1	1		
AAS DAA DAS			1						
AAD		1		2				4	
AAM		1	1	2				15	
MUL IMUL	r,(r),(i)	1						4	1/1
MUL IMUL	(r),m	1			1			4	1/1
DIV IDIV	r8	2		1				19	1/12
DIV IDIV	r16	3		1				23	1/21

DIV IDIV	r32	3		1				39	1/37
DIV IDIV	m8	2		1	1			19	1/12
DIV IDIV	m16	2		1	1			23	1/21
DIV IDIV	m32	2		1	1			39	1/37
CBW CWDE				1					
CWD CDQ		1							
SHR SHL SAR ROR ROL	r,i/CL	1							
SHR SHL SAR ROR ROL	m,i/CL	1			1	1	1		
RCR RCL	r,1	1		1					
RCR RCL	r8,i/CL	4		4					
RCR RCL	r16/32,i/CL	3		3					
RCR RCL	m,1	1		2	1	1	1		
RCR RCL	m8,i/CL	4		3	1	1	1		
RCR RCL	m16/32,i/CL	4		2	1	1	1		
SHLD SHRD	r,r,i/CL	2							
SHLD SHRD	m,r,i/CL	2		1	1	1	1		
BT	r,r/i			1					
BT	m,r/i	1		6	1				
BTR BTS BTC	r,r/i			1					
BTR BTS BTC	m,r/i	1		6	1	1	1		
BSF BSR	r,r		1	1					
BSF BSR	r,m		1	1	1				
SETcc	r			1					
SETcc	m			1		1	1		
JMP	short/near		1						1/2
JMP	far	21			1				
JMP	r		1						1/2
JMP	m(near)		1		1				1/2

JMP	m(far)	21			2				
conditional jump	short/near		1						1/2
CALL	near		1	1		1	1		1/2
CALL	far	28			1	2	2		
CALL	r		1	2		1	1		1/2
CALL	m(near)		1	4	1	1	1		1/2
CALL	m (far)	28			2	2	2		
RETN			1	2	1				1/2
RETN	i		1	3	1				1/2
RETF		23			3				
RETF	i	23			3				
J(E)CXZ	short		1	1					
LOOP	short	2	1	8					
LOOP(N)E	short	2	1	8					
ENTER	i,0			12		1	1		
ENTER	a,b	ca. 18+4b				b-1	2b		
LEAVE				2	1				
BOUND	r,m	7		6	2				
CLC STC CMC				1					
CLD STD				4					
CLI		9							
STI		17							
INTO				5					
LODS					2				
REP LODS				10+6n					
STOS					1	1	1		
REP STOS				ca. 5n a)					
MOVS				1	3	1	1		
REP MOVS				ca. 6n a)					

SCAS				1	2				
REP(N)E SCAS				12+7n					
CMPS				4	2				
REP(N)E CMPS				12+9n					
BSWAP		1		1					
CPUID		23-48							
RDTSC		31							
IN		18						>300	
OUT		18						>300	
PREFETCHNTA d)	m				1				
PREFETCHT0 d)	m				1				
PREFETCHT1 d)	m				1				
PREFETCHT2 d)	m				1				
SFENCE d)						1	1		1/6

Notes:
a) faster under certain conditions: see chapter 26.3.
b) see chapter 26.1
c) 3 if constant without base or index register
d) PIII only.

29.2 Floating point instructions									
Instruction	Operands	micro-ops						delay	throughput
		p0	p1	p01	p2	p3	p4		
FLD	r	1							
FLD	m32/64				1			1	
FLD	m80	2			2				
FBLD	m80	38			2				
FST(P)	r	1							
FST(P)	m32/m64					1	1	1	
FSTP	m80	2				2	2		
FBSTP	m80	165				2	2		

FXCH	r							0	3/1 f)
FILD	m	3			1			5	
FIST(P)	m	2				1	1	5	
FLDZ		1							
FLD1 FLDPI FLDL2E etc.		2							
FCMOVcc	r	2						2	
FNSTSW	AX	3						7	
FNSTSW	m16	1				1	1		
FLDCW	m16	1		1	1			10	
FNSTCW	m16	1				1	1		
FADD(P) FSUB(R)(P)	r	1						3	1/1
FADD(P) FSUB(R)(P)	m	1			1			3-4	1/1
FMUL(P)	r	1						5	1/2 g)
FMUL(P)	m	1			1			5-6	1/2 g)
FDIV(R)(P)	r	1						38 h)	1/37
FDIV(R)(P)	m	1			1			38 h)	1/37
FABS		1							
FCHS		3						2	
FCOM(P) FUCOM	r	1						1	
FCOM(P) FUCOM	m	1			1			1	
FCOMPP FUCOMPP		1		1				1	
FCOMI(P) FUCOMI(P)	r	1						1	
FCOMI(P) FUCOMI(P)	m	1			1			1	
FIADD FISUB(R)	m	6			1				
FIMUL	m	6			1				
FIDIV(R)	m	6			1				
FICOM(P)	m	6			1				

FTST		1						1	
FXAM		1						2	
FPREM		23							
FPREM1		33							
FRNDINT		30							
FSCALE		56							
FXTRACT		15							
FSQRT		1						69	e,i)
FSIN FCOS		17-97						27-103	e)
FSINCOS		18-110						29-130	e)
F2XM1		17-48						66	e)
FYL2X		36-54						103	e)
FYL2XP1		31-53						98-107	e)
FPTAN		21-102						13-143	e)
FPATAN		25-86						44-143	e)
FNOP		1							
FINCSTP FDECSTP		1							
FFREE	r	1							
FFREEP	r	2							
FNCLEX				3					
FNINIT		13							
FNSAVE		141							
FRSTOR		72							
WAIT				2					

Notes:
e) not pipelined
f) `FXCH` generates 1 micro-op that is resolved by register renaming without going to any port.

g) `FMUL` uses the same circuitry as integer multiplication. Therefore, the combined throughput of mixed floating point and integer multiplications is 1 `FMUL` + 1 `IMUL` per 3 clock cycles.
h) `FDIV` delay depends on precision specified in control word: precision 64 bits gives delay 38, precision 53 bits gives delay 32, precision 24 bits gives delay 18. Division by a power of 2 takes 9 clocks. Throughput is 1/(delay-1).
i) faster for lower precision.

29.3 MMX instructions (PII and PIII)									
Instruction	Operands	micro-ops						delay	throughput
		p0	p1	p01	p2	p3	p4		
MOVD MOVQ	r,r			1					2/1
MOVD MOVQ	r64,m32/64				1				1/1
MOVD MOVQ	m32/64,r64					1	1		1/1
PADD PSUB PCMP	r64,r64			1					1/1
PADD PSUB PCMP	r64,m64			1	1				1/1
PMUL PMADD	r64,r64	1						3	1/1
PMUL PMADD	r64,m64	1			1			3	1/1
PAND PANDN POR PXOR	r64,r64			1					2/1
PAND PANDN POR PXOR	r64,m64			1	1				1/1
PSRA PSRL PSLL	r64,r64/i		1						1/1
PSRA PSRL PSLL	r64,m64		1		1				1/1
PACK PUNPCK	r64,r64		1						1/1
PACK PUNPCK	r64,m64		1		1				1/1
EMMS		11						6 k)	
MASKMOVQ d)	r64,r64			1		1	1	2-8	1/30-1/2
PMOVMSKB d)	r32,r64		1					1	1/1
MOVNTQ d)	m64,r64					1	1		1/30-1/1
PSHUFW d)	r64,r64,i		1					1	1/1
PSHUFW d)	r64,m64,i		1		1			2	1/1
PEXTRW d)	r32,r64,i		1	1				2	1/1
PISRW d)	r64,r32,i		1					1	1/1

PISRW d)	r64,m16,i		1		1			2	1/1
PAVGB PAVGW d)	r64,r64			1				1	2/1
PAVGB PAVGW d)	r64,m64			1	1			2	1/1
PMINUB PMAXUB PMINSW PMAXSW d)	r64,r64			1				1	2/1
PMINUB PMAXUB PMINSW PMAXSW d)	r64,m64			1	1			2	1/1
PMULHUW d)	r64,r64	1						3	1/1
PMULHUW d)	r64,m64	1			1			4	1/1
PSADBW d)	r64,r64	2		1				5	1/2
PSADBW d)	r64,m64	2		1	1			6	1/2

Notes:
d) PIII only.
k) you may hide the delay by inserting other instructions between `EMMS` and any subsequent floating point instruction.

29.4 XMM instructions (PIII)									
Instruction	Operands	micro-ops						delay	throughput
		p0	p1	p01	p2	p3	p4		
MOVAPS	r128,r128			2				1	1/1
MOVAPS	r128,m128				2			2	1/2
MOVAPS	m128,r128					2	2	3	1/2
MOVUPS	r128,m128				4			2	1/4
MOVUPS	m128,r128		1			4	4	3	1/4
MOVSS	r128,r128			1				1	1/1
MOVSS	r128,m32			1	1			1	1/1
MOVSS	m32,r128					1	1	1	1/1
MOVHPS MOVLPS	r128,m64			1				1	1/1
MOVHPS MOVLPS	m64,r128					1	1	1	1/1

MOVLHPS MOVHLPS	r128,r128			1				1	1/1
MOVMSKPS	r32,r128	1						1	1/1
MOVNTPS	m128,r128					2	2		1/15-1/2
CVTPI2PS	r128,r64		2					3	1/1
CVTPI2PS	r128,m64		2		1			4	1/2
CVTPS2PI CVTTPS2PI	r64,r128		2					3	1/1
CVTPS2PI	r64,m128		1		2			4	1/1
CVTSI2SS	r128,r32		2		1			4	1/2
CVTSI2SS	r128,m32		2		2			5	1/2
CVTSS2SI CVTTSS2SI	r32,r128		1		1			3	1/1
CVTSS2SI	r32,m128		1		2			4	1/2
ADDPS SUBPS	r128,r128		2					3	1/2
ADDPS SUBPS	r128,m128		2		2			3	1/2
ADDSS SUBSS	r128,r128		1					3	1/1
ADDSS SUBSS	r128,m32		1		1			3	1/1
MULPS	r128,r128	2						4	1/2
MULPS	r128,m128	2			2			4	1/2
MULSS	r128,r128	1						4	1/1
MULSS	r128,m32	1			1			4	1/1
DIVPS	r128,r128	2						48	1/34
DIVPS	r128,m128	2			2			48	1/34
DIVSS	r128,r128	1						18	1/17
DIVSS	r128,m32	1			1			18	1/17
ANDPS ANDNPS ORPS XORPS	r128,r128		2					2	1/2

ANDPS ANDNPS ORPS XORPS	r128,m128		2		2			2	1/2
MAXPS MINPS	r128,r128		2					3	1/2
MAXPS MINPS	r128,m128		2		2			3	1/2
MAXSS MINSS	r128,r128		1					3	1/1
MAXSS MINSS	r128,m32		1		1			3	1/1
CMPccPS	r128,r128		2					3	1/2
CMPccPS	r128,m128		2		2			3	1/2
CMPccSS	r128,r128		1		1			3	1/1
CMPccSS	r128,m32		1		1			3	1/1
COMISS UCOMISS	r128,r128		1					1	1/1
COMISS UCOMISS	r128,m32		1		1			1	1/1
SQRTPS	r128,r128	2						56	1/56
SQRTPS	r128,m128	2			2			57	1/56
SQRTSS	r128,r128	2						30	1/28
SQRTSS	r128,m32	2			1			31	1/28
RSQRTPS	r128,r128	2						2	1/2
RSQRTPS	r128,m128	2			2			3	1/2
RSQRTSS	r128,r128	1						1	1/1
RSQRTSS	r128,m32	1			1			2	1/1
RCPPS	r128,r128	2						2	1/2
RCPPS	r128,m128	2			2			3	1/2
RCPSS	r128,r128	1						1	1/1
RCPSS	r128,m32	1			1			2	1/1
SHUFPS	r128,r128,i		2	1				2	1/2
SHUFPS	r128,m128,i		2		2			2	1/2

UNPCKHPS UNPCKLPS	r128,r128		2	2				3	1/2
UNPCKHPS UNPCKLPS	r128,m128		2		2			3	1/2
LDMXCSR	m32	11						15	1/15
STMXCSR	m32	6						7	1/9
FXSAVE	m4096	116						62	
FXRSTOR	m4096	89						68	

30. Testing speed

The Pentium family of processors have an internal 64 bit clock counter which can be read into `EDX:EAX` using the instruction `RDTSC` (read time stamp counter). This is very useful for testing exactly how many clock cycles a piece of code takes.

The program below is useful for measuring the number of clock cycles a piece of code takes. The program executes the code to test 10 times and stores the 10 clock counts. The program can be used in both 16 and 32 bit mode on the PPlain and PMMX:

```
;************   Test program for PPlain and PMMX:    ********************

ITER    EQU     10              ; number of iterations
OVERHEAD EQU    15              ; 15 for PPlain, 17 for PMMX

RDTSC   MACRO                   ; define RDTSC instruction
        DB      0FH,31H
ENDM
;************   Data segment:                 ********************
.DATA                           ; data segment
ALIGN   4
COUNTER DD      0               ; loop counter
TICS    DD      0               ; temporary storage of clock
RESULTLIST  DD  ITER DUP (0)    ; list of test results
;************   Code segment:                 ********************
.CODE                           ; code segment
BEGIN:  MOV     [COUNTER],0     ; reset loop counter
TESTLOOP:                       ; test loop
;************   Do any initializations here:    ********************
        FINIT
;************   End of initializations          ********************
        RDTSC                   ; read clock counter
        MOV     [TICS],EAX      ; save count
        CLD                     ; non-pairable filler
REPT    8
        NOP                     ; eight NOP's to avoid shadowing effect
ENDM

;************   Put instructions to test here:  ********************
        FLDPI                   ; this is only an example
        FSQRT
        RCR     EBX,10
        FSTP    ST
;***************** End of instructions to test  ********************

        CLC                     ; non-pairable filler with shadow
        RDTSC                   ; read counter again
        SUB     EAX,[TICS]      ; compute difference
        SUB     EAX,OVERHEAD    ; subtract clocks used by fillers etc.
        MOV     EDX,[COUNTER]   ; loop counter
        MOV     [RESULTLIST][EDX],EAX   ; store result in table
        ADD     EDX,TYPE RESULTLIST     ; increment counter
        MOV     [COUNTER],EDX           ; store counter
        CMP     EDX,ITER * (TYPE RESULTLIST)
        JB      TESTLOOP                ; repeat ITER times

; insert here code to read out the values in RESULTLIST
```

The 'filler' instructions before and after the piece of code to test are are included in order to get consistent results on the PPlain. The `CLD` is a non-pairable instruction which has been inserted to make sure the pairing is the same the first time as the subsequent times. The eight `NOP` instructions are inserted to prevent any prefixes in the code to test to be decoded in the shadow of the preceding instructions on the PPlain. Single byte instructions are used here to obtain the same pairing the first time as the subsequent times. The `CLC` after the code to test is a non-pairable instruction which has a shadow under which the `0FH` prefix of the `RDTSC` can be decoded so that it is independent of any shadowing effect from the code to test on the PPlain.

On The PMMX you may want to insert `XOR EAX,EAX / CPUID` before the instructions to test if you want the FIFO instruction buffer to be empty, or some time-consuming instruction (f.ex. `CLI` or `AAD`) if you want the FIFO buffer to be full (`CPUID` has no shadow under which prefixes of subsequent instructions can decode).

On the PPro, PII and PIII you have to insert `XOR EAX,EAX / CPUID` before and after each `RDTSC` to prevent it from executing in parallel with anything else, and remove the filler instructions. (`CPUID` is a serializing instruction which means that it flushes the pipeline and waits for all pending operations to finish before proceeding. This is useful for testing purposes.)

The `RDTSC` instruction cannot execute in virtual mode on the PPlain and PMMX, so if you are running DOS programs you must run in real mode. (Press F8 while booting and select "safe mode command prompt only" or "bypass startup files").

The complete test program is available from www.agner.org/assem/.

The Pentium processors have special performance monitor counters which can count events such as cache misses, misalignments, various stalls, etc. Details about how to use the performance monitor counters are not covered by this manual but can be found in "Intel Architecture Software Developer's Manual", vol. 3, Appendix A.

31. Comparison of the different microprocessors

The following table summarizes some important differences between the microprocessors in the Pentium family:

	PPlain	PMMX	PPro	PII	PIII
code cache, kb	8	16	8	16	16
data cache, kb	8	16	8	16	16
built in level 2 cache, kb	0	0	256	512 *)	512 *)
MMX instructions	no	yes	no	yes	yes
XMM instructions	no	no	no	no	yes
conditional move instructruct.	no	no	yes	yes	yes
out of order execution	no	no	yes	yes	yes
branch prediction	poor	good	good	good	good
branch target buffer entries	256	256	512	512	512
return stack buffer size	0	4	16	16	16

branch misprediction penalty	3-4	4-5	10-20	10-20	10-20
partial register stall	0	0	5	5	5
FMUL latency	3	3	5	5	5
FMUL throughput	1/2	1/2	1/2	1/2	1/2
IMUL latency	9	9	4	4	4
IMUL throughput	1/9	1/9	1/1	1/1	1/1

*) Celeron: 0-128, Xeon: 512 or more, many other variants available. On some versions the level 2 cache runs at half speed.

Comments to the table:
Code cache size is important if the critical part of your program is not limited to a small memory space.

Data cache size is important for all programs that handle more than small amounts of data in the critical part.

MMX and XMM instructions are useful for programs that handle massively parallel data, such as sound and image processing. In other applications it may not be possible to take advantage of the MMX and XMM instructions.

Conditional move instructructions are useful for avoiding poorly predictable conditional jumps.

Out of order execution improves performance, especially on non-optimized code. It includes automatic instruction reordering and register renaming.

Processors with a good branch prediction method can predict simple repetitive patterns. A good branch prediction is most important if the branch misprediction penalty is high.

A return stack buffer improves prediction of return instructions when a subroutine is called alternatingly from different locations.

Partial register stalls make handling of mixed data sizes (8, 16, 32 bit) more difficult.

The latency of a multiplication instruction is the time it takes in a dependency chain. A throughput of 1/2 means that the execution can be pipelined so that a new multiplication can begin every second clock cycle. This defines the speed for handling parallel data.

Most of the optimizations described in this document have little or no negative effects on other microprocessors, including non-Intel processors, but there are some problems to be aware of.

Scheduling floating point code for the PPlain and PMMX often requires a lot of extra `FXCH` instructions. This will slow down execution on older microprocessors, but not on the Pentium family and advanced non-Intel processors.

Taking advantage of the MMX instructions in the PMMX, PII and PIII processors or the conditional moves in the PPro, PII and PIII will create problems if you want your code to be compatible with earlier microprocessors. The solution may be to write several versions of your code, each optimized for a particular processor. Your program should detect which processor it is running on and select the appropriate version of code (chapter 27.10).