How to optimize for the Pentium family of microprocessors

By Agner Fog, Ph.D.

Contents

1 Introduction

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

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. Mechanistic explanations in this manual should be regarded as a model which is useful for predicting microprocessor behavior. I have no way of knowing whether it is in accordance with the actual physical structure of the microprocessors. The only purpose of providing this information here is to enable programmers to optimize their code. My findings are sometimes in disagreement with data published by Intel. Reasons for this discrepancy might be that Intel data are theoretical while my data are obtained experimentally under a particular set of testing conditions. It is possible that different testing conditions may lead to different results. Most tests are done in 32-bit protected mode without interrupts. Far jumps and calls are tested in 16-bit mode.

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 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. I am therefore not able to tell which brand is best.

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. A good textbook on the subject is "Introduction to 80x86 Assembly Language and Computer Architecture" by R. C. Detmer, 2001.

Please don't send your programming questions to me, I am not gonna do your homework for you! There are various discussion forums on the Internet where you can get answers to your programming questions if you cannot find the answers in the relevant books and manuals.

Good luck with your hunt for nanoseconds!

1.1 Assembly language syntax

The assembly language syntax used in this manual is MASM syntax. MASM - the Microsoft assembler - is now available for free. It is the most advanced and most used assembler available, and the MASM syntax has always been a *de facto* standard. Textbooks and manuals use this syntax, and most C++ compilers can translate C++ code to assembly code with MASM syntax. (Some versions of the Gnu C++ compiler can only produce AT&T syntax assembly code).

There are two different versions of MASM syntax. MASM implementations up to version 5.10 use a somewhat lax syntax with incomplete syntax checking. Most of the problems and ambiguities in the older syntax have been resolved in MASM version 6 and later, which uses a slightly different syntax. The old MASM 5.10 syntax is still supported in later versions of MASM when appropriate options are set.

The Borland assembler, called TASM, supports MASM 5.10 syntax but not MASM 6.x syntax (TASM is no longer commercially available). Many other assemblers with non-standardized syntaxes are listed at Programmer's heaven. An open source assembler that runs under several operating systems and supports MASM syntax stands high on my wish list.

MASM 6.x is designed to run under 32-bit Windows. See page 21 for instructions on how to use MASM under Linux and similar operating systems.

All examples in this manual work with MASM 5.x, MASM 6.x and TASM. Instructions not covered by older assemblers can be emulated with the macros available from www.agner.org/assem/macros.zip.

MASM syntax manuals can be found at Microsoft's MSDN library and at Randal Hyde's assembly page.

1.2 Microprocessor versions covered by this manual

The following versions of Intel x86-family microprocessors are discussed in this manual:

Name	Abbreviation
Pentium (without name suffix)	P1
Pentium MMX	PMMX
Pentium Pro	PPro
Pentium II	P2
Pentium III	P3
Pentium 4	P4

The name *Celeron* applies to Pentium II and later models with less cache than the standard versions. The name *Xeon* applies to Pentium II and later models with more cache than the standard versions.

The P1 and PMMX processors represent the fifth generation in the Intel x86 series of microprocessors, and their processor kernels are very similar. PPro, P2 and P3 all have the sixth generation kernel. These three processors are almost identical except for the fact that new instructions are added to each new model. P4 is the first processor in the seventh generation which, for obscure reasons, is not called seventh generation in Intel documents. Quite unexpectedly, the generation number returned by the CPUID instruction in the P4 is not 7 but 15. The reader should be aware that the 5'th, 6'th and 7'th generation micro-processors behave very differently. What is optimal for one generation may not be optimal for the others.

2 Getting started with optimization

2.1 Choice of algorithm

The first thing to do when you want to optimize a piece of software is to find the best algorithm. Optimizing a poor algorithm is a waste of time. So don't even think of converting your code to assembly before you have explored all possibilities for optimizing your algorithm and the implementation of your algorithm.

2.2 Choice of programming language

Before starting a new software project, you have to decide which programming language to use. Low-level languages are good for optimizing execution speed or program size, while high-level languages are good for making clear and well-structured code. A typical programmer spends more time finding errors and making additions and modifications than on writing new code. Therefore, most software is written in high-level languages that are easier to document and maintain. The backside of the coin is that the code gets slower and the demands on hardware performance gets bigger and bigger. At the opposite extreme, we have assembly language which produces very compact and fast code, but is very difficult to debug and maintain.

Today, most universities teach Java as the first programming language for pedagogical reasons. The advantages of Java are that it is consistent, well structured, and portable. But it is not fast, because in most cases it runs on a virtual Java machine that interprets code rather than executing it. If execution speed is important, then the best choice will be C++. This language has the best of both worlds. The C++ language has more features and options than most other programming languages. Advanced features like classes, polymorphism, template libraries and exception handling enable you to make well-structured and reusable code at a high level of abstraction. On the other hand, the C++ language is a superset of the old C language, which gives you access to fiddle with every bit and byte and to use low-level programming techniques. Furthermore, most C++ developing systems provide easy access to mix C++ and assembly language. It is therefore possible to optimize the most critical part of your code using assembly language, and leave the rest of the project in high-level C++.

2.3 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 that supports this mode. The code examples shown in this manual assume a 32-bit flat memory model, unless otherwise specified.

2.4 Finding the hot spots

Before you try to optimize anything, 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. If this is the case then you should isolate this hot spot in a separate subroutine that you can optimize for speed, while the rest of your program can be optimized for clarity and maintainability.

You may translate the critical subroutine to assembly and leave everything else in high-level language. Many assembly programmers waste a lot of energy optimizing the wrong parts of their programs. There are even people who make entire Windows programs in assembly. Most of the code in a typical program goes to the user interface and to calling system routines. A user interface with menus and dialog boxes is certainly not something that is being executed a thousand times per second. People who try to optimize something like this in assembly may be spending hours - or more likely months - making the program respond ten nanoseconds faster to a mouse click on a system where the screen is refreshed 60 times per second. There are certainly better ways of investing your programming skills! The same applies to program sections that consist mainly of calls to system routines. Such calls are usually well optimized by C++ compilers and there is no reason to use assembly language here.

Assembly language should be used only for loops that are executed so many times that it really matters in terms of CPU time, and that is very many. A 2 GHz Pentium 4 can do $6 \cdot 10^9$ integer additions per second. So it is probably not worth the effort to optimize a loop that makes "only" one million additions. It will suffice to change from Java to C++.

Typical applications where assembly language can be useful for optimizing speed include processing of sound and images, compression and encryption of large amounts of data, simulation of complex systems, and mathematical calculations that involve iteration.

Assembly language is also useful when optimizing code for size. This is typically used when a piece of code has to fit into a ROM. Using assembly language for optimizing an application program for size is not worth the effort because data storage is so cheap.

The highest priorities in modern software development are typically not execution time but development time, as well as clarity, documentation, verification, security, reusability, and - most importantly - maintainability. Software programmers are typically spending much more time on debugging, maintaining and modifying than on primary programming. These priorities require that the code is divided into well-defined and well-demarcated modules. This is the reason why the trend in software development tools is going in the direction of object orientation and high levels of abstraction.

Assembly language is the opposite of all this. Assembly language should therefore be used only where a significant gain in performance can be expected. The assembly code should be encapsulated into small modules with a well-defined interface to the high-level language from where it is called. These modules should be thoroughly tested and documented.

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 or a better graphics library.

Some high level language compilers offer relatively good optimization for specific processors, but in most cases further optimization by hand can make it much better. When the possibilities for optimizing in C++ have been exhausted, then you can make your C++ compiler translate the critical subroutine to assembly, and do further optimizations by hand.

2.5 Literature

A lot of useful literature can be downloaded for free from Intel's web 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 instruction set. However, the documents from Intel are not always accurate. Especially the first Pentium tutorials had many errors.

The most important manuals are "Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual", and "IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference". 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.

VTUNE is a software tool from Intel for optimizing code. I have not tested it and can therefore not give any evaluation 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 resources follow the links from www.agner.org/assem.

Some useful textbooks are "Introduction to 80x86 Assembly Language and Computer Architecture" by R. C. Detmer, 2001; and "Computer Architecture: A Quantitative Approach" by J. L. Hennessy and D. A. Patterson, 3'rd ed. 2002.

3 Optimizing in C++

Here, I will give you some general advices for improving the speed of C++ code. Most of these advices apply to other compiled programming languages as well.

3.1 Use optimization options

Study the optimization options provided by your compiler and use the ones that are applicable to your project. Turn off all debugging options when you are finished using them and want to make the final distributable code.

3.2 Identify the most critical parts of your code

In computation-intensive software programs, you will often find that 99% of the CPU time is used in the innermost loop. Identifying the most critical part of your software is therefore necessary if you want to improve the speed of computation. Optimizing less critical parts of your code will not only be a waste of time, it also makes your code less clear, and less easy to debug and maintain. If it is not obvious which part of your code is most critical, then you may use a profiler. If you don't have a profiler, then set up a number of counter variables that are incremented at different places in your code to see which part is executed most times.

Study the algorithm used in the critical part of your code and see if it can be improved. Often you can gain more speed simply by optimizing the algorithm than by any other optimization method.

3.3 Break dependence chains

Modern microprocessors can do out-of-order execution. This means that if a piece of software specifies the calculation of A and then B, and the calculation of A is slow, then the microprocessor can begin the calculation of B before the calculation of A is finished. Obviously, this is only possible if the value of A is not needed for the calculation of B.

In order to take advantage of out-of-order execution, you have to avoid long dependence chains. A dependence chain is a series of calculations, where each calculation depends on the result of the preceding one. Consider the following example, which calculates the sum of 100 numbers:

```
double list[100], sum = 0.;
for (int i = 0; i < 100; i++) sum += list[i];
```

This is a long dependence chain. If a floating-point addition takes 5 clock cycles, then this loop will take approximately 500 clock cycles. You can improve the performance dramatically by splitting the dependence chain in two:

```
double list[100], sum1 = 0., sum2 = 0.;
for (int i = 0; i < 100; i += 2) {
   sum1 += list[i];
   sum2 += list[i+1];}
sum1 += sum2;
```

If the microprocessor is doing an addition to `sum1` from time T to T+5, then it can do another addition to `sum2` from time T+1 to T+6, and the whole loop will take only 256 clock cycles.

3.4 Use local variables

Variables and objects that are declared inside a function, and not static, will be stored on the stack. The same applies to function parameters. The memory space occupied by these variables is released when the function returns, and can be reused by the next function. Using the same memory space again and again makes the caching of memory more efficient. Unless you have very big arrays and objects on your stack, you can be almost certain that your local variables are in the level-1 cache inside the microprocessor, from where they can be accessed many times faster than other parts of the memory. Static and global variables and objects are stored at a fixed place in memory, and are less likely to be cached.

If you need global variables then you may make these variables part of a class and access them through member functions. This may save space in the level-2 cache and the trace cache because addresses relative to the '`this`' pointer can be expressed with an 8-bit or 16-bit offset, while absolute addresses require 32 bits. The drawback is that extra code is required for transferring the '`this`' pointer to the member functions.

Allocating objects with `new` or `malloc` is inefficient, and should be avoided in critical parts of your code. For example, a first-in-first-out queue can be implemented in an array with wrap-around, rather than a linked list. If you are using container class templates, then make your own templates that do not use dynamic memory allocation.

3.5 Use array of structures rather than structure of arrays

Variables that are used together should preferably be stored near each other in order to improve caching. If you have two arrays, `a` and `b`, and the elements are accessed in the order `a[0]`, `b[0]`, `a[1]`, `b[1]`, ... then you may improve the performance by making an array of a structure which contains one `a` and one `b`.

3.6 Alignment of data

A variable is accessed most efficiently if it is stored at a memory address which is divisible by the size of the variable. For example, a `double` takes 8 bytes of storage space. It should therefore preferably be stored at an address divisible by 8. The size should always be a power of 2. Objects bigger than 16 bytes should be stored at an address divisible by 16.

Not all compilers give you access to control data alignment. But you can control the alignment of structure and class members. Consider this structure:

```
struct abc {
   unsigned char a;    // takes 1 byte storage
   int b;              // 4 bytes storage
   double c;           // 8 bytes storage
} x;
```

Assume that `x` is stored at address N, which is divisible by 8. Then `x.a` will be at address N, `x.b` at address N+1, and `x.c` at address N+5. So `x.b` and `x.c` will not be properly aligned. You may change the structure definition to:

```
struct abc {
   double c;           // 8 bytes storage
   int b;              // 4 bytes storage
   unsigned char a;    // 1 byte storage
   char unused[3];     // fill up to 16 bytes
} x;
```

Now all elements are properly aligned, provided that `x` is aligned by 8. The 3 extra unused characters make sure that if you have an array of structures, then all elements in the array will be properly aligned.

3.7 Division

Division takes much longer time than addition, subtraction and multiplication. You should therefore minimize the number of divisions.

You can divide an integer by 2^n by shifting the binary number n places to the right. All modern compilers will use this trick if the divisor is a power of 2 and known as a constant at compile time. Likewise, multiplication by a power of 2 will be done using a left shift. The method is simpler if the dividend is unsigned. Example:

```
int n = 1000;
int divisor = 8;
int fraction = n / divisor;
```

Change this to:

```
int n = 1000;
const int divisor = 8;
int fraction = (unsigned)n / divisor;    // (will do n >> 3)
```

Making `divisor` a `const` (or simply writing `8` instead of `divisor`) makes sure that the compiler can use this optimization. Making `n` unsigned improves the code even further (assuming that you are certain that `n` can never be negative).

Floating-point division by a constant should be done by multiplying with the reciprocal:

```
double n;
double divisor = 1.2345;
double fraction = n / divisor;
```

Change this to:

```
double n;
const double factor = (1. / 1.2345);
double fraction = n * factor;
```

The compiler will calculate `1./1.2345` at compile time and insert the reciprocal in the code, so you will never spend time doing the division. In fact, any expression that contains only constants and does not involve function calls, will be evaluated by the compiler and replaced by the result.

Divisions can sometimes be eliminated completely. For example:

```
    if (a > b / c)
```

can often be replaced by

```
    if (a * c > b)
```

Pitfalls: The inequality sign must be reversed if `c` < 0. The division is inexact if `b` and `c` are integers, while the multiplication is exact. The multiplication may cause overflow.

Multiple divisions can be combined. For example, `a1/b1 + a2/b2` should be replaced by `(a1*b2 + a2*b1) / (b1*b2)` which has one division instead of two. The trick of using a common denominator can even be used on completely independent divisions. Example:

```
    double a1, a2, b1, b2, y1, y2;
    y1 = a1 / b1;
    y2 = a2 / b2;
```

This can be changed to:

```
    double a1, a2, b1, b2, y1, y2, reciprocal_divisor;
    reciprocal_divisor = 1. / (b1 * b2);
    y1 = a1 * b2 * reciprocal_divisor;
    y2 = a2 * b1 * reciprocal_divisor;
```

You can even do four divisions in one:

```
    double a1, a2, a3, a4, b1, b2, b3, b4, y1, y2, y3, y4;
    y1 = a1 / b1;
    y2 = a2 / b2;
    y3 = a3 / b3;
    y4 = a4 / b4;
```

can be replaced by:

```
    double a1, a2, a3, a4, b1, b2, b3, b4, y1, y2, y3, y4;
    double b12, b34, reciprocal_divisor;
    b12 = b1 * b2;  b34 = b3 * b4;
    reciprocal_divisor = 1. / (b12 * b34);
    y1 = a1 * b2 * b34 * reciprocal_divisor;
    y2 = a2 * b1 * b34 * reciprocal_divisor;
    y3 = a3 * b4 * b12 * reciprocal_divisor;
    y4 = a4 * b3 * b12 * reciprocal_divisor;
```

It is not recommended to combine more than four divisions; because the time saved by having only one division will be spent on the increased number of multiplications.

3.8 Function calls

When a function with parameters is called, the parameters are stored on the stack by the caller and read again by the called function. This causes some delay if a parameter is part of a critical dependence chain, especially on the P4 processor. There are several alternative ways to avoid this:

1. keep the most critical dependence chain entirely inside one function

2. use `inline` functions. An inline function will be expanded like a macro without parameter transfer, if possible.

3. use `#define` macros with parameters instead of functions.
   But beware that macro parameters are evaluated every time they are used.
   Example:
   ```
     #define max(a,b) (a > b ? a : b)
     y = max(sin(x),cos(x));
   ```
   In this example, `sin(x)` and `cos(x)` are both calculated twice because the macro is referencing them twice. This is certainly not optimal.

4. declare functions `__fastcall`. The first two or three (depending on compiler) integer parameters will be transferred in registers rather than on the stack when the function is declared `__fastcall`. Floating-point parameters are always stored on the stack. The implicit '`this`' pointer in member functions (methods) is also treated like a parameter, so there may be only one free register left for transferring your parameters. Therefore, make sure that the most critical integer parameter comes first when you are using `__fastcall`.

5. declare functions `static`. Static functions have no external linkage. This enables the compiler to optimize across function calls.

Your compiler may ignore the optimization hints given by the `inline` and `static` modifiers, while `__fastcall` is certain to have an effect on the first one or two integer parameters. Using `#define` macros is likely to have an effect on floating-point parameters as well.

If a large object is transferred to a function as a parameter, then the entire object is copied. The copy constructor is called if there is one. If copying the object is not necessary for the logic of your algorithm, then you can save time by transferring a pointer or reference to the object rather than a copy of the object, or by making the function a member of the object's class. Whether you choose to use a pointer, a reference, or a member function, is a matter of programming style. All three methods will produce the same compiled code, which will be more efficient than copying the object. In general, pointers, references, and member functions are quite efficient. Feel free to use them whenever it is useful for the logic structure of your program. Function pointers and virtual functions are somewhat less efficient.

3.9 Conversion from floating-point numbers to integers

According to the standards for the C++ language, all conversions from floating-point numbers to integers use truncation towards zero, rather than rounding. This is unfortunate because truncation takes much longer time than rounding on most microprocessors. It is beyond my comprehension why there is no round function in standard C++ libraries. If you cannot avoid conversions from `float` or `double` to `int` in the critical part of your code, then you may make your own round function using assembly language:

```
inline int round (double x) {
   int n;
   __asm fld x;
   __asm fistp n;
   return n;}
```

This code is not portable, and will work only on Intel-compatible microprocessors. The round function is also available in the function library at www.agner.org/assem/asmlib.zip.

The P3 and P4 processors have fast truncation instructions, but these instructions are not compatible with previous microprocessors and can therefore only be used in code that is written exclusively for these microprocessors.

Conversion of unsigned integers to floating-point numbers is also slow. Use signed integers for efficient conversion to float.

3.10 Use old-fashioned character arrays for text strings

Modern C++ libraries define a class named `string` or `CString` which facilitates the manipulation of text strings. These classes use dynamic memory allocation and are much less efficient than the old method of using character arrays for text strings. If you don't know how to do this, then find the explanation in an old C++ textbook, or study the documentation for the functions `strcpy`, `strncpy`, `strcat`, `strlen`, `strcmp`, `sprintf`. Remember that it is your own responsibility that the length of a string never exceeds the length of the array minus 1.

4 Combining assembly and high level language

Before you start to code a function in assembly language, you should code it in C++, using the optimization guidelines given in the previous chapter (page 7). Only the most critical part of your program needs to be optimized using assembly language.

4.1 Inline assembly

The simplest way to combine C++ and assembly language is to inert inline assembly in the C++ code. See the compiler manual for syntax details.

Note that not all registers can be used freely in inline assembly. To be safe, avoid modifying EBP, ESP and EBX. It is recommended to let the C++ compiler make an assembly file so you can check if the inline assembly code interfaces correctly with the surrounding C++ code and that no reserved register is modified without saving. See the chapters below on register usage.

The C++ compiler may interpret the most common assembly instructions using a built-in assembler. But in many cases the compiler needs to translate all the surrounding C++ code to assembly and run everything through an assembler. It may be possible to specify which assembler to use for inline assembly, but the assembler must be compatible with the assembly generated by the compiler as well as with the inline assembly.

If you are using the Gnu compiler, you have to use the primitive AT&T syntax for inline assembly or move the assembly code to a separate module.

An alternative to inline assembly is to make entire functions in separate assembly language modules. This gives better control of register use and function prolog and epilog code. The following chapters give more details on how to make assembly language modules that can be linked with C++ programs.

4.2 Calling conventions

An application binary interface (ABI) is a set of standards for programs running under a particular system. When linking assembly language modules together with modules written in other programming languages, it is essential that your assembly code conform to all standards. It is possible to use your own standards for assembly procedures that are called only from other assembly procedures, but it is highly recommended to follow as many of the existing standards as possible. On the 32-bit Intel x86-compatible platform, there are several different conventions for transferring parameters to procedures:

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
member function	compiler specific	compiler specific

The `__cdecl` calling convention is the default for C and C++ functions, while `__stdcall` is the default for system functions. Remember that the stack pointer is decreased when a value is pushed on the stack. This means that the parameter pushed first will be at the highest address, in accordance with the `_pascal` convention. You must push parameters in reverse order to satisfy the `__cdecl` and `__stdcall` conventions.

The `__fastcall` convention allows parameters to be transferred in registers. This is considerably faster, especially on the P4. Unfortunately, the `__fastcall` convention is different for different compilers. You may improve execution speed by using registers for parameter transfer on assembly procedures that are called only from other assembly language procedures.

4.3 Data storage in C++

Variables and objects that are declared inside a function in C++ will be stored on the stack and addressed by `ESP` or `EBP`. This is the most efficient way of storing data, for two reasons. Firstly, the stack space used for local storage is released when the function returns and may be reused by the next function that is called. Using the same memory area repeatedly improves data caching. The second reason is that data stored on the stack can often be addressed with an 8-bit offset relative to a pointer rather than the 32 bits required for addressing data in the data segment. This makes the code more compact so that it takes less space in the code cache or trace cache.

Global and static data in C++ are stored in the data segment and addressed with 32-bit absolute addresses. A third way of storing data in C++ is to allocate space with `new` or `malloc`. This method should be avoided if speed is critical.

The following example shows a simple C++ function with local data stored on the stack, and the same function translated to assembly. (Calling details will be explained on page 16 below).

```
; Example 4.1
extern "C" double SinPlusPow (double a, double b, double c) {
  double x, y;
  x = sin(a);
  y = pow(b,c);
  return x + y;}
```

Same in assembly, with `__cdecl` calling convention:

```
_SinPlusPow PROC NEAR
SMAP        STRUC      ; make a map of data on stack
CALLPARM1   DQ  ?      ; parameter 1 for call to sin and pow
CALLPARM2   DQ  ?      ; parameter 2 for call to pow
X           DQ  ?      ; local variable X
Y           DQ  ?      ; local variable Y
RETURNADDR  DD  ?      ; return address for _SinCosPlusOne
A           DQ  ?      ; parameters for _SinCosPlusOne
B           DQ  ?
```

```
C_          DQ  ?     ; (C is a reserved word in MASM 6)
SMAP        ENDS

; compute space required for data not already on stack:
LOCALDATASPACE = SMAP.RETURNADDR - SMAP.CALLPARM1

        SUB    ESP, LOCALDATASPACE  ; make space for local data
        FLD    [ESP].SMAP.A         ; load a
        FSTP   [ESP].SMAP.CALLPARM1 ; store a on top of stack
        CALL   _sin                 ; _sin reads parameter CALLPARM1
        FSTP   [ESP].SMAP.X         ; store x
        FLD    [ESP].SMAP.B         ; load b
        FSTP   [ESP].SMAP.CALLPARM1 ; store b on top of stack
        FLD    [ESP].SMAP.C_        ; load c
        FSTP   [ESP].SMAP.CALLPARM2 ; store c next on stack
        CALL   _pow                ;_pow reads CALLPARM1 and CALLPARM2
        FST    [ESP].SMAP.Y         ; store y
        FADD   [ESP].SMAP.X         ; x + y
        ADD    ESP, LOCALDATASPACE  ; release local data space
        RET                         ; return value in ST(0)
_SinPlusPow ENDP

PUBLIC  _SinPlusPow                 ; public function
EXTRN   _sin:near, _pow:near        ; external functions
```

In this function, we are allocating space for local data by subtracting the size of CALPARM1, CALLPARM2, X and Y from the stack pointer. The stack pointer must be restored to its original value before the RET. Before calling the functions _sin and _pow, we must place the function parameters for these calls at the right place relative to the current value of ESP. Therefore, we have placed CALLPARM1 and CALLPARM2 at the beginning of STACKMAP. It is more common to push the parameters before a function call and pop the stack after the call, but this method is faster. We are assuming here that the _sin and _pow functions use the __cdecl calling convention so that ESP still points to CALLPARM1 after the call. Therefore, we don't need to adjust the stack pointer between the two calls.

Remember, when using ESP as a pointer, that the value of ESP is changed every time you have a PUSH or POP. If you are using simplified function directives (MASM 6.x syntax), such as:

```
    SinPlusPow PROC NEAR C, a:REAL8, b:REAL8, c:REAL8
```

then you have an implicit PUSH EBP in the prolog code which you must include in your stack map. You may use .LISTALL to see the prolog code. Remember, also, that the size of the stack map must be a multiple of 4.

This function could be further optimized. We might use integer registers for moving A, B and C; we don't need to store Y; and we might use the FSIN instruction rather than calling the external function _sin. The purpose of the above example is just to show how data are stored and transferred on the stack.

If your assembly code contains many calls to high-level language functions or system functions, then you are in all likelihood optimizing the wrong part of your program. The critical innermost loop where most of the CPU time is used should be placed in a separate function that does not call any other functions.

4.4 Register usage in 16 bit mode DOS or Windows

Function parameters are passed on the stack according to the calling conventions listed on page 12. Parameters of 8 or 16 bits size use one word of stack space. Parameters bigger than 16 bits are stored in little-endian form, i.e. with the least significant word at the lowest address.

Function return values are passed in registers in most cases. 8-bit integers are returned in AL, 16-bit integers and near pointers in AX, 32-bit integers and far pointers in DX:AX, Booleans in AX, and floating-point values 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. The high word of ESP cannot be used because it is modified by interrupts and task switches.

4.5 Register usage in 32 bit Windows

Function parameters are passed on the stack according to the calling conventions listed on page 12. Parameters of 32 bits size or less use one DWORD of stack space. Parameters bigger than 32 bits are stored in little-endian form, i.e. with the least significant DWORD at the lowest address, and DWORD aligned.

Function return values are passed in registers in most cases. 8-bit integers are returned in AL, 16-bit integers in AX, 32-bit integers, pointers, and Booleans in EAX, 64-bit integers in EDX:EAX, and floating-point values in ST(0). Structures and class objects not exceeding 64 bits size are returned in the same way as integers, even if the structure contains floating point values. Structures and class objects bigger than 64 bits are returned through a pointer passed to the function as the first parameter and returned in EAX. Compilers that don't support 64-bit integers may return structures bigger than 32 bits through a pointer. The Borland compiler also returns structures through a pointer if the size is not a power of 2.

Registers EAX, ECX and EDX may be changed by a procedure. All other general-purpose registers (EBX, ESI, EDI, EBP) must be saved and restored if they are used. The value of ESP must be divisible by 4 at all times, so don't push 16-bit data on the stack. Segment registers cannot be changed, not even temporarily. CS, DS, ES, and SS all point to the flat segment group. FS is used for a thread environment block. GS is unused, but reserved. Flags may be changed by a 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 can 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.

4.6 Register usage in Linux

The rules for register usage in Linux appear to be almost the same as for 32-bit windows. Registers EAX, ECX, and EDX may be changed by a procedure. All other general-purpose registers must be saved. There appears to be no rule for the direction flag. Function return values are transferred in the same way as under Windows. Calling conventions are the same, except for the fact that no underscore is prefixed to public names. I have no information about the use of FS and GS in Linux. It is not difficult to make an assembly function that works under both Windows and Linux, if only you take these minor differences into account.

4.7 Making compiler-independent code

Functions

By default, C++ compilers use a method called name mangling for distinguishing between different versions of overloaded functions. A code that defines the number and type of function parameters, and possibly the calling convention and return type, is appended to the function name. These name mangling codes are compiler-specific. It is therefore recommended to turn off name mangling when linking C++ and assembly code together. You can avoid the mangling of a function name by adding `extern "C"` to the function prototype in the C++ file. `extern "C"` indicates that the function should be linked according to the conventions of the C language, rather than C++. Therefore, `extern "C"` cannot be used for constructs that don't exist in the C language, such as member functions. You may also add `__cdecl` to the function prototype to make the calling convention explicit.

The assembly code must have the function name prefixed by an underscore (_) if called under Windows or DOS. The underscore is not required in newer versions of other operating systems such as Linux. Your C++ compiler may have an option for adding or removing the underscore, but you have to recompile all function libraries if you change this option (and risk name clashes).

Thus, the best way of linking C++ and assembly code together is to add `extern "C"` to the function prototype in C++, and prefix the function name with an underscore in the assembly file. You must assemble with case sensitivity on externals. Example:

```
; Example 4.2
; extern "C" int square (int x);
  _square PROC NEAR              ; integer square function
  PUBLIC  _square
          MOV     EAX, [ESP+4]   ; read x from stack
          IMUL    EAX            ; x * x
          RET                    ; return value in EAX
  _square ENDP
```

Global objects

Some C++ compilers also mangle the names of global objects. Add `extern "C"` to the declaration of global variables and objects if you want them to be accessible from assembly modules. Even better, avoid global objects if possible.

Member functions

Let's take as an example a simple C++ class containing a list of integers:

```
// Example 4.3
// define C++ class
class MyList {
  int length;                 // number of items in list
  int buffer[100];            // store items
  public:
  MyList();                   // constructor
  void AddItem(int item);     // add item to list
  int Sum();};                // compute sum of items

MyList::MyList() {            // constructor in C++
  length = 0;}

void MyList::AddItem(int item) { // member function AddItem in C++
  if (length < 100) buffer[length++] = item;}

int MyList::Sum() {           // member function Sum in C++
  int i, s = 0;
  for (i=0; i<length; i++) s += buffer[i];
  return s;}
```

The implementation of this class is compiler-specific because different C++ compilers use different calling conventions and name mangling methods for member functions. If you want to place one or more of the member functions in an assembly module and you don't want to mess with compiler-specific intricacies, then you may replace the member function by a `friend` function. The '`this`' pointer is not transferred automatically to a `friend` function so you have to make it an explicit parameter to the function. The C++ class definition is then changed as follows:

```
    // Example 4.3 with friend function
    // predefine class name:
    class MyList;

    // define external friend function:
    extern "C" void MyListAddItem(MyList * p_this, int item);

    // changed definition of class MyList:
    class MyList {
      int length;               // number of items in list
      int buffer[100];          // store items
      public:
      MyList();                 // constructor
      // make external function a friend:
      friend void MyListAddItem(MyList *, int);
      void AddItem(int item) {  // wrap MyListAddItem in AddItem
        MyListAddItem(this,item);} //tranfer 'this' explicitly to function
      int Sum();};              // compute sum of items
```

The `friend` function `MyListAddItem` can be coded in assembly without name mangling:

```
    ; define data members of class MyList:
    MyList  STRUC
    LENGTH_  DD   ?
    BUFFER  DD   100 DUP (?)
    MyList  ENDS

    ; define friend function MyListAddItem
    _MyListAddItem PROC NEAR
    PUBLIC _MyListAddItem
            MOV     ECX, [ESP+4]                ; p_this
            MOV     EAX, [ESP+8]                ; item
            MOV     EDX, [ECX].MyList.LENGTH_   ; length
            CMP     EDX, 100
            JNB     ADDITEM9
            MOV     [ECX+4*EDX].MyList.BUFFER, EAX
            ADD     EDX, 1
            MOV     [ECX].MyList.LENGTH_, EDX
    ADDITEM9: RET
    _MyListAddItem ENDP
```

Wrapping `MyListAddItem` in `AddItem` does not slow down execution because it is inlined when called inside the class definition. An optimizing compiler will simply call `MyListAddItem` instead of `AddItem`. The `extern "C"` linking makes this solution compatible with all compilers.

Member function pointers

The implementation of member function pointers is compiler-specific. If your assembly code needs a pointer to member functions then replace the member functions by friend functions as explained above, and replace the member function pointer by a friend function pointer.

Data member pointers

The implementation of data member pointers is compiler-specific. Some compilers add 1 to the offset in order to distinguish a pointer to the first data member from a null pointer. Furthermore, some compilers use 64 bits for the member pointer in order to handle more complicated constructs.

If your assembly code needs a pointer to structure or class data members (corresponding to the C++ operators `.*` or `->*`), then try if it can be replaced by a simple pointer or an array index. If this is not possible then replace the data member pointer by an integer which contains the offset of the member relative to the class object. You need to typecast addresses to integers in order to use such a member pointer in C++ code.

Virtual functions and polymorphous classes

Each object of a polymorphous class contains a pointer to a table of pointers to the virtual functions. Borland and Microsoft compilers place the pointer to this so-called virtual table at the beginning of the object, while the Gnu compiler places it at the end of the object. This makes the code compiler-specific, even if only non-virtual member functions are coded in assembly. If you want to make the code compiler-independent then you have to replace *all* virtual member functions by friend functions. Insert one or more friend function pointers as members of the class and initialize these pointers in the constructors to emulate the virtual tables.

Long double

A `long double` in C++ corresponds to a `TBYTE` in assembly, using 10 bytes. The Microsoft C++ compiler does not support this type, but replaces it with a `double`, using 8 bytes. The Borland compiler allocates 10 bytes for a `long double`, while the Gnu compiler allocates 12 (or 16) bytes for the sake of alignment. If you want a structure or class containing `long double` members to be compiler-independent (and properly aligned) then use a union:

```
    union ld {
      long double a;
      char filler[16];};
```

The Microsoft compiler can still not access the `long double` without using assembly language.

Thread local objects

Thread-local data or objects should be defined or allocated in C++, but they can be accessed in assembly code through pointers.

Assembly language functions can be made reentrant (thread safe) without the need for thread-local storage when all variables are stored on the stack.

4.8 Adding support for multiple compilers in .asm modules

An alternative to making compiler-independent code is to use the proper name mangling, calling conventions, etc. for the C++ compiler in question. To do this, you have to write the code in C++ first and make the compiler translate the C++ to assembly. Use the assembly code produced by the C++ compiler to get the right mangled names, calling conventions, data formats, etc.

Functions

In many cases, it is possible to make assembly libraries containing functions that are compatible with more than one C++ compiler by adding the proper mangled names for each compiler.

For example, overloaded functions cannot be made without name mangling, but by adding several mangled names, the function can be made compatible with several different compilers. The following example shows a square function with two overloaded versions:

```
; Example 4.4: Overloaded function
; int square (int x);           // C++ prototype
SQUARE_I PROC NEAR              ; integer square function
@square$qi LABEL NEAR           ; link name for Borland compiler
?square@@YAHH@Z LABEL NEAR      ; link name for Microsoft compiler
_square__Fi LABEL NEAR          ; link name for Gnu compiler (Windows)
square__Fi LABEL NEAR           ; link for Gnu (Redhat, Debian, BSD)
_Z6squarei LABEL NEAR           ; link for Gnu (Mandrake, UNIX)
PUBLIC @square$qi,?square@@YAHH@Z,_square__Fi,square__Fi,_Z6squarei
        MOV     EAX, [ESP+4]    ; x
        IMUL    EAX
        RET
SQUARE_I ENDP

; double square (double x);     // C++ prototype
SQUARE_D PROC NEAR              ; double precision float square funct.
@square$qd LABEL NEAR           ; link name for Borland compiler
?square@@YANN@Z LABEL NEAR      ; link name for Microsoft compiler
_square__Fd LABEL NEAR          ; link name for Gnu compiler (Windows)
square__Fd LABEL NEAR           ; link for Gnu (Redhat, Debian, BSD)
_Z6squared LABEL NEAR           ; link for Gnu (Mandrake, UNIX)
PUBLIC @square$qd,?square@@YANN@Z,_square__Fd,square__Fd,_Z6squared
        FLD     QWORD PTR [ESP+4]  ; x
        FMUL    ST(0), ST(0)
        RET
SQUARE_D ENDP
```

Member functions

The above method works because all the compilers use the `__cdecl` calling convention by default for overloaded functions. For member functions (methods) however, the compilers differ. Borland and most Gnu compilers use the `__stdcall` convention; Gnu for Mandrake uses `__cdecl`; and Microsoft compilers uses a hybrid of `__stdcall` and `__fastcall`, with the '`this`' pointer in `ECX`. To obtain binary compatibility, we want to force the compilers to use the same calling convention. Most compilers allow you to explicitly specify the calling convention to member functions, but a few compilers (e.g. Gnu for Mandrake) do not allow this specification. However, you can force all compilers to use the `__cdecl` method for member functions by specifying a variable number of parameters. Returning to example 4.3 page 16, we can provide support for almost all compilers in this way:

```
// Example 4.3 with support for multiple compilers
// define C++ class
class MyList {
  int length;                        // number of items in list
  int buffer[100];                   // store items
  public:
  MyList();                          // constructor
  void AddItem(int item, ...);       // add item to list
  int Sum(...);};                    // compute sum of items


; Assembly code for MyList::AddItem with mangled names for several
; compilers:

; define data members of class MyList:
MyList  STRUC
LENGTH_ DD   ?
BUFFER  DD   100 DUP (?)
MyList  ENDS
```

```
; define member function MyListAddItem with __cdecl calling method:
MyListAddItem PROC NEAR                   ; extern "C" friend (UNIX)
_MyListAddItem LABEL NEAR                 ; extern "C" friend (Windows)
@MyList@AddItem$qie LABEL NEAR            ; Borland
?AddItem@MyList@@QAAXHZZ LABEL NEAR       ; Microsoft
_AddItem__6MyListie LABEL NEAR            ; Gnu (Windows)
AddItem__6MyListie LABEL NEAR             ; Gnu (Redhat, Debian, BSD)
_ZN6MyList7AddItemEiz LABEL NEAR          ; Gnu (Mandrake, UNIX)
PUBLIC MyListAddItem, _MyListAddItem, @MyList@AddItem$qie
PUBLIC ?AddItem@MyList@@QAAXHZZ, _AddItem__6MyListie
PUBLIC AddItem__6MyListie, _ZN6MyList7AddItemEiz

        MOV     ECX, [ESP+4]              ; 'this'
        MOV     EAX, [ESP+8]              ; item
        MOV     EDX, [ECX].MyList.LENGTH_   ; length
        CMP     EDX, 100
        JNB     ADDITEM9
        MOV     [ECX+4*EDX].MyList.BUFFER, EAX
        ADD     EDX, 1
        MOV     [ECX].MyList.LENGTH_, EDX
ADDITEM9: RET
MyListAddItem ENDP
```

This method works for member functions and constructors. Destructors and overloaded operators cannot have a variable number of parameters. It is possible to explicitly specify the calling convention for member functions and overloaded operators on most compilers. The only compiler I have come across that doesn't allow this specification is Gnu for Mandrake, which uses the `__cdecl` convention for member functions and overloaded operators. Thus, you can make assembly code for overloaded operators compatible by specifying the `__cdecl` convention on all other compilers. Note that the mangled names are changed if you change the specified calling convention, even if the generated code is identical.

Member functions that return an object bigger than 8 bytes are not binary compatible for any calling method because the Microsoft compiler places the `this` pointer first, while other compilers place the return pointer first. In this case you may return the object through an explicit pointer parameter.

You may want to use the friend function method described on page 17 or use inline assembly to avoid these problems and intricacies for both member functions, constructors, destructors, and overloaded operators.

4.9 Further compiler incompatibilities

There are still incompatibilities that cannot be handled with the methods described in the preceding chapters. Do not expect your assembly code to be compatible with multiple compilers if it contains `__fastcall` functions, `new`, `delete`, global objects with constructors or destructors, or exception handling. The ways of name-mangling standard library functions and system functions may also differ.

4.10 Object file formats

Another compatibility problem stems from differences in the formats of object files. Borland, Symantec and 16-bit Microsoft compilers use the OMF format for object files. Microsoft and Gnu compilers for 32-bit Windows use MS-COFF format, also called PE. The Gnu compiler under Linux, BSD, and similar systems prefers ELF format.

The MASM assembler can produce both OMF and MS-COFF format object files, but not ELF format. It is often possible to translate object files from one format to another. The linker and library manager that come with Microsoft compilers can translate object files from OMF to MS-COFF format. A freeware utility called EMXAOUT1 can translate object files from OMF format to the old a.out format that many Gnu linkers accept. The Gnu objcopy utility is a more versatile tool for converting object formats. Which object formats it can handle depends on the build options. The version of objcopy that comes with the MingW32 package can convert between MS-COFF format and ELF format (Download binutils.xxx.tar.gz from www.mingw.org). With this utility, it is possible to use the same assembly module with several different compilers under several different operating systems.

More details about object file formats can be found in the book "Linkers and Loaders" by J. R. Levine (Morgan Kaufmann Publ. 2000).

4.11 Using MASM under Linux

The Gnu assembler that comes with Linux, BSD and similar operating systems uses the terrible AT&T syntax. As I am in favor of standardizing assembly syntax, I will recommend using the MASM assembler under Linux.

Tools for converting assembly files in MASM syntax to AT&T syntax are not reliable, but tools for converting object files seem to work well. You may assemble your code with MASM under Windows or under Linux using Wine, the Windows emulator. Let MASM generate object files in MS-COFF format and convert them to ELF format using the objcopy utility from MingW32 mentioned above. The object files in ELF format can then be linked into a C++ project using g++ or ld. Under Linux, the process goes like this:

```
wine -- ml.exe /c /Cx /coff myfile.asm
wine -- objcopy.exe -Oelf32-i386 myfile.obj myfile.o
g++ somefile.cpp myfile.o
```

You may include this sequence in a make script or shell script. It may seem awkward to use the MING version of objcopy which needs Wine to run under Linux. It is probably possible to rebuild the native objcopy utility to add support for the MS-COFF format (called pe-i386 in this context), but I haven't figured out how.

If you have leading underscores on your function names then add the option `--remove-leading-char` to the objcopy command line.

If you want to build a function library that can be used under several different operating systems, then make a .lib file under Windows using the lib.exe utility that comes with Microsoft compilers and convert the .lib file to ELF format with the command

```
objcopy -Oelf32-i386 --remove-leading-char myfile.lib myfile.a
```

The library myfile.a can then be used under Linux, BSD, UNIX, etc. Make sure your library doesn't call any system functions.

4.12 Object oriented programming

As explained on page 6, object oriented programming principles may be required for the sake of clarity and maintainability of a software project. Object oriented programming means classes containing member data (properties) and member functions (methods). You may expect this extra complexity to slow down program performance, but this is not necessarily the case.

Well-designed C++ member functions are expected to access no other data than their member data and parameters. The function parameters are stored on the stack, and so are the member data if the object is declared locally (automatic) inside some other function. This means that all the data can be kept together within a small area of memory so that data caching will be very efficient. A further advantage is that you avoid using 32-bit addresses for global data. This makes the code more compact so that it takes less space in the code cache or trace cache.

The disadvantage of using member functions, in terms of performance, is that the `this` pointer has to be transferred to the member function as an extra parameter. The register that is used for the `this` pointer might otherwise have been used for other purposes. These disadvantages may outweigh the advantages for small member functions, but not for bigger member functions.

It is not necessary to translate all the member functions of a C++ class to assembly language. It is possible to make the most critical member function in assembly and leave the rest of the member functions, including constructor and destructor, in C++.

Different compilers for the x86 platform are not compatible on object-oriented code. As explained on page 16ff, you have the choice between several different strategies for overcoming this problem:

1. use inline assembly inside C++ code
2. make assembly modules containing simple functions rather than member functions
3. make compiler-independent code using `friend` functions, etc.
4. make compiler-specific code and add support for several compilers in the assembly module if necessary
5. make the entire program in assembly (not recommended except for extremely simple programs)

4.13 Other high level languages

If you are using other high-level languages than C++, and the compiler manual has no information on how to link with assembly, then see if the manual has any information on how to link with C or C++ modules. You can probably find out how to link with assembly from this information.

In general, you have to use simple functions without name mangling, compatible with the `extern "C"` and `__cdecl` or `__stdcall` conventions in C++. This will work with most compiled languages. Arrays and strings may be implemented differently in different languages.

To call assembly code from Java, you have to compile the code to a DLL and use the Java Native Interface (JNI).

5 Debugging and verifying assembly code

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 C++. Next, write a test program that can test your subroutine thoroughly. Make sure the test program goes into all branches and boundary cases.

When your C++ subroutine works with your test program then you are ready to translate the code to assembly language. Most C++ compilers can translate C++ to assembly.

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 methods described in chapter 20 page 131. If the code is significantly slower than expected, then check the list of possible bottlenecks on page 74 for PPro, P2 and P3, and page 93 for P4.

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.

6 Reducing code size

As explained in chapter 9 page 28, the code cache is 8 or 16 kb on P1, PMMX, PPro, P2 and P3. 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. You may also want to reduce the size of your code if speed is not important.

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, especially because of segment prefixes. 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 that does not have such a short form is `MOV`. Examples:

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

May be changed to:

```
        SUB EAX,EAX             ; 2 bytes
```

And

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

May be changed to:

```
        SUB 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
```

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 bytes displacement, even when it is 0:

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

7 Detecting processor type

What is optimal for one microprocessor may not be optimal for another. Therefore, you may make the most critical part of your program in different versions, each optimized for a specific microprocessor, and select the desired version at run time after detecting which microprocessor the program is running on. The CPUID instruction tells which instructions the microprocessor supports. If you are using instructions that are not supported by all microprocessors, then you must first check if the program is running on a microprocessor that supports these instructions. If your program can benefit significantly from using Single-Instruction-Multiple-Data (SIMD) instructions, then you may make one version of a critical part of the program that uses these instructions, and another version which does not and which is compatible with old microprocessors.

I have provided a library of subroutines that check the processor type and determine which instructions are supported. This can be downloaded from www.agner.org/assem/asmlib.zip. These subroutines can be called from assembly as well as from high-level language. Obviously, it is recommended to store the output from such a subroutine rather than calling it again each time the information is needed.

For assemblers that don't support the newest instruction set, you may use the macros at www.agner.org/assem/macros.zip.

7.1 Checking for operating system support for XMM registers

Unfortunately, the information that can be obtained from the CPUID instruction is not sufficient for determining whether it is possible to use the SSE and SSE2 instructions, which use the 128-bit XMM registers. The operating system has to save these registers during a task switch and restore them when the task is resumed. The microprocessor can disable the use of the XMM registers in order to prevent their use under old operating systems that do not save these registers. Operating systems that support the use of XMM registers must set bit 9 of the control register CR4 to enable the use of XMM registers and indicate its ability to save and restore these registers during task switches. (Saving and restoring registers is actually faster when XMM registers are enabled).

Unfortunately, the CR4 register can only be read in privileged mode. Application programs therefore have a serious problem determining whether they are allowed to use the XMM registers or not. According to official Intel documents, the only way for an application program to determine whether the operating system supports the use of XMM registers is to try to execute an XMM instruction and see if you get an invalid opcode exception. This is ridiculous, because not all operating systems, compilers and programming languages provide facilities for application programs to catch invalid opcode exceptions. The advantage of using the XMM registers evaporates completely if you have no way of knowing whether you can use these registers without crashing your software.

These serious problems led me to search for an alternative way of checking if the operating system supports the use of XMM registers, and fortunately I have found a way that works reliably. If XMM registers are enabled, then the `FXSAVE` and `FXRSTOR` instructions can read and modify the XMM registers. If XMM registers are disabled, then `FXSAVE` and `FXRSTOR` cannot access these registers. It is therefore possible to check if XMM registers are enabled by trying to read and write these registers with `FXSAVE` and `FXRSTOR`. The subroutines in www.agner.org/assem/asmlib.zip use this method. These subroutines can be called from assembly as well as from high-level languages, and provide an easy way of detecting whether XMM registers can be used.

In order to verify that this detection method works correctly with all microprocessors, I first checked various manuals. The 1999 version of Intel's software developer's manual says about the `FXRSTOR` instruction: *"The Streaming SIMD Extension fields in the save image (XMM0-XMM7 and MXCSR) will not be loaded into the processor if the CR4.OSFXSR bit is not set."* AMD's Programmer’s Manual says effectively the same. However, the 2003 version of Intel's manual says that this behavior is implementation dependent. In order to clarify this, I contacted Intel Technical Support and got the reply, *"If the OSFXSR bit in CR4 in not set, then XMMx registers are not restored when FXRSTOR is executed"*. They further confirmed that this is true for all versions of Intel microprocessors and all microcode updates. I regard this as a guarantee from Intel that my detection method will work on all Intel microprocessors. We can rely on the method working correctly on AMD processors as well since the AMD manual is unambiguous on this question. It appears to be safe to rely on this method working correctly on future microprocessors as well, because any microprocessor that deviates from the above specification would introduce a security problem as well as failing to run existing programs. Compatibility with existing programs is of great concern to microprocessor producers.

The subroutines in www.agner.org/assem/asmlib.zip are constructed so that the detection will give a correct answer unless `FXSAVE` and `FXRSTOR` are *both* buggy. My detection method has been further verified by testing on many different versions of Intel and AMD processors and different operating systems (Test program available at www.agner.org/assem/xmmtest.zip).

The detection method recommended in Intel manuals has the drawback that it relies on the ability of the compiler and the operating system to catch invalid opcode exceptions. A Windows application, for example, using Intel's detection method would therefore have to be tested in all compatible operating systems, including various Windows emulators running under a number of other operating systems. My detection method does not have this problem because it is independent of compiler and operating system. My method has the further advantage that it makes modular programming easier, because a module, subroutine library, or DLL using XMM instructions can include the detection procedure so that the problem of XMM support is of no concern to the calling program, which may even be written in a different programming language.

The above discussion has relied on the following documents:

Intel application note AP-900: "Identifying support for Streaming SIMD Extensions in the Processor and Operating System". 1999.

Intel application note AP-485: "Intel Processor Identification and the CPUID Instruction". 2002.

"Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference", 1999.

"IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference", 2003.

"AMD64 Architecture Programmer's Manual, Volume 4: 128-Bit Media Instructions", 2003.

8 Alignment

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

operand size	alignment	
	P1 and PMMX	PPro, P2, P3, P4
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

```
; Example 8.1, alignment of static data
_DATA SEGMENT PARA PUBLIC USE32 'DATA' ; must be paragraph aligned
A     DQ  ?, ?          ; A is aligned by 16
B     DB  32 DUP (?)
C     DD  ?
D     DW  ?
ALIGN 16                ; E must be aligned by 16
E     DQ  ?, ?
_DATA ENDS
_CODE SEGMENT BYTE PUBLIC 'CODE'
      MOVDQA  XMM0, [A]
      MOVDQA  [E], XMM0
```

In the above example, A, B and C all start at addresses divisible by 16. D starts at an address divisible by 4, which is more than sufficient because it only needs to be aligned by 2. An alignment directive must be inserted before E because the address after D is not divisible by 16 as required by the MOVDQA instruction. Alternatively, E could be placed after A or B to make it aligned.

On P1 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, P2 and P3, misaligned data will cost 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. On P4, there is little or no penalty for misaligned operands smaller than 16 bytes if reads do not occur shortly after writes to the same address. Unaligned 128 bit (16 bytes) operands can only be accessed with the MOVDQU instruction or with two separate 64-bit operations.

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

```
; Example 8.2, alignment of data on stack
_FuncWithAlign PROC NEAR
        PUSH    EBP                 ; Prolog code
        MOV     EBP, ESP
        SUB     ESP, LocalSpace    ; Allocate space for local data
        AND     ESP, 0FFFFFFF0H    ; (= -16) Align ESP by 16
        MOVDQU  XMM0,[EBP+8]       ; Unaligned function parameter
        MOVDQA  [ESP],XMM0         ; Store something in aligned space
        ...
        MOV     ESP, EBP           ; Epilog code. Restore ESP
        POP     EBP                ; Restore EBP
        RET
_FuncWithAlign ENDP
```

This function uses EBP to address function parameters, and ESP to address aligned local data.

9 Cache

A cache is a temporary storage that is closer to the microprocessor than the main memory. Data and code that is used often, or that is expected to be used soon, is stored in a cache so that it is accessed faster. Different microprocessors have one, two or three levels of cache. The level-1 cache is close to the microprocessor kernel and is often accessed in a single clock cycle. A bigger level-2 cache is placed on the same chip or at least in the same housing.

The level-1 data cache in the P4 processor, for example, can contain 8 kb of data. It is organized as 128 lines of 64 bytes each. The cache is 4-way set-associative. This means that the data from a particular memory address cannot be assigned to an arbitrary cache line, but only to one of four possible lines. The line length in this example is 2^6 = 64. So each line must be aligned to an address divisible by 64. The least significant 6 bits, i.e. bit 0 - 5, of the memory address are used for addressing a byte within the 64 bytes of the cache line. As each set comprises 4 lines, there will be 128 / 4 = 32 = 2^5 different sets. The next five bits, i.e. bits 6 - 10, of a memory address will therefore select between these 32 sets. The remaining bits can have any value. The conclusion of this mathematical exercise is that if bits 6 - 10 of two memory addresses are equal, then they will be cached in the same set of cache lines. The 64-byte memory blocks that contend for the same set of cache lines are spaced 2^{11} = 2048 bytes apart. No more than 4 such addresses can be cached at the same time.

Let me illustrate this by the following piece of code, where EDI holds an address divisible by 64:

```
; Example 9.1
AGAIN:  MOV  EAX, [EDI]
        MOV  EBX, [EDI + 0804H]
        MOV  ECX, [EDI + 1000H]
        MOV  EDX, [EDI + 5008H]
        MOV  ESI, [EDI + 583CH]
        SUB  EBP, 1
        JNZ  AGAIN
```

The five addresses used here all have the same set-value because the differences between the addresses with the lower 6 bits truncated are multiples of 2048 = 800H. This loop will perform poorly because at the time you read ESI, there is no free cache line with the proper set-value, so the processor takes the least recently used of the four possible cache lines, that is the one which was used for EAX, and fills it with the data from [EDI+5800H] to [EDI+583FH] and reads ESI. 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. Here, you have nothing but cache misses, but if the 5'th line is changed to `MOV ESI,[EDI + 5840H]` then we have crossed a 64 byte boundary, so that we do not have the same set-value as in the first four lines, and there will be no problem assigning a cache line to each of the five addresses.

The cache sizes, cache line sizes, and set associativity on different microprocessors are listed in the table on page 154. The performance penalty for cache line contention can be quite considerable on older microprocessors, but on the P4 you loose only a few clock cycles because the level-2 cache is accessed quite fast through a full-speed 256 bits data bus. The improved efficiency of the level-2 cache in the P4 compensates for the smaller level-1 data cache.

The cache lines are always aligned to physical addresses divisible by the cache line size (in the above example 64). When you have read a byte at an address divisible by 64, then the next 63 bytes will be cached as well, and can be read or written to at almost no extra cost. You can take advantage of this by arranging data items that are used near each other together into aligned blocks of 64 bytes of memory. If, for example, you have a loop that 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 a big array or other data structure is a multiple of 64 bytes, then you should preferably align it by 64. The cache line size on older microprocessors is 32.

The rules for the data cache also apply to the code cache for processors prior to the P4. The code cache in the P4 is a trace cache. This means that code instructions are translated into micro-operations (uops) before being cached. The trace cache has to be quite big, because the uops often take more space than the instruction codes before translation. The major reason for using a trace cache is that the decoding of instructions often is a bottleneck which limits the performance. The problem is that it is fairly complicated to determine the length of an instruction. While the microprocessor may execute several instructions in parallel if they are independent, it cannot decode them in parallel because it doesn't know where the second instruction begins before it has determined the length of the first instruction. The PPro, P2 and P3 can decode up to three instructions per clock cycle, which is quite impressive. But at higher clock frequencies, this may be more difficult. And one obvious solution is to decode instructions before caching them. Each entry in the P4 trace cache is at least 36 bits wide, probably more. There are 12288 entries in the trace cache, so the size is more than 55 kb.

It is important that the critical part of your code (the innermost loop) fits in the trace cache or code cache. Frequently used pieces of code, or routines which are used together, should preferably be stored near each other. Seldom used branches or procedures should be put away in the bottom of your code or somewhere else.

It may be very difficult to determine if your data addresses or code addresses contend for the same cache sets, especially if they are scattered around in different segments. The best thing you can do to avoid problems of cache line contention is to keep all data used in the critical part or your program within one contiguous block not bigger than the cache, or up to four contiguous blocks no bigger than a fourth of that. This will make sure that your cache lines are used optimally.

Since you need stack space anyway for subroutine parameters and return addresses, the best way of keeping data together is to copy all frequently used static data to dynamic variables on the stack before entering the most critical part of your program, and copy them back again outside the critical loop if they have been changed.

The delay caused by a level-1 cache miss depends on how far away the needed data are. The level-2 cache can be accessed quite fast; the level-3 cache somewhat slower; and accesses to the main memory takes many clock cycles. The delay is even longer if a DRAM page boundary is crossed, and extremely long if the memory area has been swapped to disk.

I am not giving any time estimates here because the timings depend very much on the hardware configuration; and the fast technological development make any figures obsolete in half a year.

On PPro, P2, P3 and P4, a write miss will normally load a cache line, but it is possible to set up an area of memory to perform differently, for example video RAM (See "IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide").

Other ways of speeding up memory reads and writes are discussed on page 130.

The P1 and PPro have two write buffers, PMMX, P2 and P3 have four, and P4 has six. On the P4 you may have up to six unfinished writes to uncached memory without delaying the subsequent instructions. Each write buffer can handle operands up to 128 bits wide (64 bits on earlier processors).

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 (see page 27).

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 resource 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` in 16-bit mode because the timer interrupt will modify the high word of `ESP` at unpredictable times).

For further advices on improving cache efficiency see the Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual.

10 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 data cache, which may take much more time than executing the instructions. The data are more likely to be in the cache when the code is repeated.

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 12 page 34.

4. In many processors, decoding the length of each instruction is a bottleneck. The P1 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 P1 the first time they are executed, unless the first of the two instructions is only one byte long. The PMMX, PPro, P2 and P3 have no penalty on first time decoding. On the P4, instructions go directly from the decoder to the execution unit the first time they are executed. On subsequent executions they are submitted from the trace cache at the rate of 3 uops per clock.

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 trace cache or 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.

11 Out-of-order execution (PPro, P2, P3, P4)

The most important improvement that came with the sixth generation of microprocessors is out-of-order execution. The idea is that if the execution of a particular instruction is delayed because the input data are not available yet, then the microprocessor will try to find later instructions that it can do first, if the input data for the latter instructions are ready. Obviously, the microprocessor has to check if the latter instructions need the output from the former instruction. If each instruction depends on the result of the preceding instruction, then we have no opportunities for out-of-order execution. The logic for determining input dependences and the mechanisms for doing instructions as soon as the necessary inputs are ready, gives us the further advantage that the microprocessor can do several things at the same time. If we need to do an addition and a multiplication, and neither instruction depends on the output of the other, then we can do both at the same time, because they are using two different execution units. But we cannot do two multiplications at the same time, because both will need to use the same execution unit.

Typically, everything in these microprocessors is highly pipelined in order to improve the throughput. If, for example, a floating-point addition takes 5 clock cycles, and the execution unit is fully pipelined, then we can start one addition at time T, which will be finished at time T+5, and start another addition at time T+1, which will be finished at time T+6. The advantage of this technology is therefore highest if we can organize our code so that there are as few dependencies as possible between successive instructions.

11.1 Instructions are split into uops

The microprocessors with out-of-order execution are translating all instructions into micro-operations - abbreviated *uops*. A simple instruction such as `ADD EAX,EBX` generates only one uop, while an instruction like `ADD EAX,[MEM1]` generates two: one for reading from memory into a temporary (unnamed) register, and one for adding the contents of the temporary register to `EAX`. The instruction `ADD [MEM1],EAX` generates three uops: one for reading from memory, one for adding, and one for writing the result back to memory. The advantage of this is that the uops can be executed out of order. Example:

```
    MOV EAX, [MEM1]
    IMUL EAX, 5
    ADD EAX, [MEM2]
    MOV [MEM3], EAX
```

Here, the `ADD EAX,[MEM2]` instruction is split into two uops. The advantage of this is that the microprocessor can fetch the value of `[MEM2]` at the same time as it is doing the multiplication. If none of the data are in the cache, then the microprocessor will start to fetch `[MEM2]` immediately after starting to fetch `[MEM1]`, and long before the multiplication can start. This principle also makes the stack work more efficiently. Consider the example:

```
    PUSH EAX
    CALL FUNC
```

The PUSH EAX instruction is split into two uops which can be represented as SUB ESP,4 and MOV [ESP],EAX. The advantage of this is that the SUB ESP,4 uop can be executed even if the value of EAX is not ready yet. The CALL operation needs the new value of ESP, so the CALL would have to wait for the value of EAX if the PUSH instruction was not split into uops. Thanks to the use of uops, the value of the stack pointer almost never causes delays in normal programs.

11.2 Register renaming

Consider the example:

```
MOV EAX, [MEM1]
IMUL EAX, 6
MOV [MEM2], EAX
MOV EAX, [MEM3]
ADD EAX, 2
MOV [MEM4], EAX
```

This piece of code is doing two things that have nothing to do with each other: multiplying [MEM1] by 6 and adding 2 to [MEM3]. If we were using a different register in the last three instructions, then the independence would be obvious. And, in fact, the microprocessor is actually smart enough to do just that. It is using a different temporary register in the last three instructions so that it can do the multiplication and the addition in parallel. The instruction set gives us only seven general-purpose 32-bit registers, and often we are using them all. So we cannot afford the luxury of using a new register for every calculation. But the microprocessor has more temporal registers to use. The PPro, P2 and P3 have 40 universal temporary registers, and the P4 probably has 126. The microprocessor can *rename* any of these temporary registers to represent the logical register EAX.

Register renaming works fully automatically and in a very simple way. Every time an instruction writes to or modifies a logical register, the microprocessor assigns a new temporary register to that logical register. The first instruction in the above example will assign one temporary register to EAX. The second instruction is putting a new value into EAX, so a new temporary register will be assigned here. In other words, the multiplication instruction will use two different registers, one for input and another one for output. The next example illustrates the advantage of this:

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

Assume, now, that [MEM1] is in the cache, while [MEM2] is not. This means that the multiplication can start before the addition. The advantage of using a new temporary register for the result of the multiplication is that we still have the old value of EAX, which has to be kept until EBX is ready for the addition. If we had used the same register for the input and output of the multiplication, then the multiplication would have to wait until the loading of EBX and the addition was finished.

After all the operations are finished, the value in the temporary register that represents the last value of EAX in the code sequence is written to a permanent EAX register. This process is called retirement (see page 69 and 87).

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.

11.3 Dependence chains

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

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

In this example, the `IMUL` instructions generate several uops each, one for reading from memory, and one or more for multiplying. The read uops can execute out or order, while the multiplication uops must wait for the previous uops to finish. This dependence chain takes quite a long time, because multiplication is slow. You can improve this code by breaking up the dependence chain. The trick 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.

Floating-point instructions often have a longer latency than integer instructions, so you should definitely break up long dependence 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 by register renaming so they hardly put any load on the execution units. If the dependence chain is long and the latencies of the instructions are long compared to the throughputs, then you may use more than two accumulators.

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 this address is finished. This penalty is particularly high in the P4. 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 this 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 dependence chains. The throughput for these instructions is one jump per one or two clock cycles. It is therefore recommended that you give the microprocessor something else to do between jumps.

12 Branch prediction (all processors)

The pipeline in a modern microprocessor contains many stages, including instruction fetch, decoding, register allocation and renaming, uop reordering, execution, and retirement. Handling instructions in a pipelined manner allows the microprocessor to do many things at the same time. While one instruction is being executed, the next instructions are being fetched and decoded. The biggest problem with pipelining is branches in the code. For example, a conditional jump allows the instruction flow to go in any of two directions. If there is only one pipeline, then the microprocessor doesn't know which of the two branches to feed into the pipeline until the branch instruction has been executed. The longer the pipeline, the more time does the microprocessor waste if it has fed the wrong branch into the pipeline.

As the pipelines become longer and longer in every new microprocessor generation, the problem of branch misprediction becomes so big that the microprocessor designers go to great lengths to reduce it. The designers are inventing more and more sophisticated mechanisms for predicting which way a branch will go, in order to minimize the frequency of branch mispredictions. The history of branch behavior is stored in order to use past history for predicting future behavior. This prediction has two aspects: predicting whether a conditional jump will be taken or not, and predicting the target address that it jumps to. A cache called Branch Target Buffer (BTB) stores the target address of all jumps. The first time an unconditional jump is executed, the target address is stored in the BTB. The second time the same jump is executed, the target address in the BTB is used for fetching the predicted target into the pipeline, even though the true target is not calculated until the jump reaches the execution stage. The predicted target is very likely to be correct, but not certain, because the BTB may not be big enough to contain all jumps in a program, so different jumps may replace each other's entries in the BTB.

12.1 Prediction methods for conditional jumps

When a conditional jump is encountered, the microprocessor has to predict not only the target address, but also whether the conditional jump is taken or not taken. If the guess is right and the right target is loaded, then the pipeline goes smoothly and fast. But if the prediction is wrong and the microprocessor has loaded the wrong target into the pipeline, then the pipeline has to be flushed, and the time that has been spent on fetching, decoding and perhaps speculatively executing instructions in the wrong branch is wasted.

Saturating counter

A relatively simple method is to store information in the BTB about what the branch has done most in the past. This can be done with a saturating counter, as shown in figure 12.1.

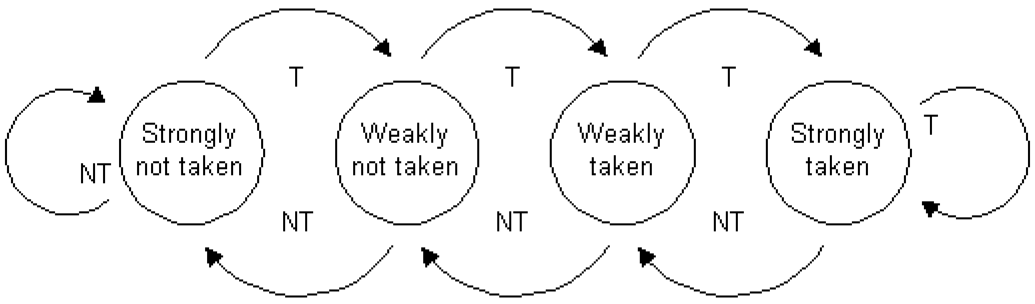

Figure 12.1. Saturating 2-bit counter

This counter has four states. Every time the branch is taken, the counter goes up to the next state, unless it already is in the highest state. Every time the branch is not taken, the counter goes down one step, unless it already is in the lowest state. When the counter is in one of the highest two states, it predicts that the branch will be taken the next time. When the counter is in one of the lowest two states, it predicts that the branch will not be taken the next time. If the branch has been not taken several times in a row, the counter will be in the lowest state, called "strongly not taken". The branch then has to be taken twice for the prediction to change to taken. Likewise, if the branch has been taken several times in a row, it will be in state "Strongly taken". It has to be not taken twice before the prediction changes to not taken. In other words, the branch has to deviate twice from what it has done most in the past before the prediction changes.

This method is good for a branch that does the same most of the time, but not good for a branch that changes often. The P1 uses this method, though with a flaw, as explained on page 38.

Two-level adaptive predictor

Consider the behavior of the counter in figure 12.1 for a branch that is taken every second time. If it starts in state "strongly not taken", then the counter will alternate between state "strongly not taken" and "weakly not taken". The prediction will always be "not taken", which will be right only 50% of the time. Likewise, if it starts in state "strongly taken" then it will predict "taken" all the time. The worst case is if it happens to start in state "weakly taken" and alternates between "weakly not taken" and "weakly taken". In this case, the branch will be mispredicted all the time.

A method of improving the prediction rate for branches with such a regularly recurring pattern is to remember the history of the last *n* occurrences of the branch and use one saturating counter for each of the possible 2^n history patterns. This method, which was invented by T.-Y. Yeh and Y. N. Patt, is illustrated in figure 12.2.

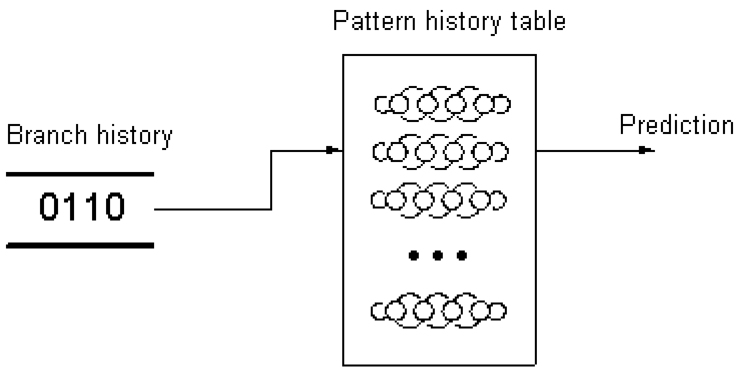

Figure 12.2. Adaptive two-level predictor

Consider the example of *n* = 2. This means that the last two occurrences of the branch are stored in a 2-bit shift register. This branch history register can have 4 different values: 00, 01, 10, and 11; where 0 means "not taken" and 1 means "taken". Now, we make a pattern history table with four entries, one for each of the possible branch histories. Each entry in the pattern history table contains a 2-bit saturating counter of the same type as in figure 12.1. The branch history register is used for choosing which of the four saturating counters to use. If the history is 00 then the first counter is used. If the history is 11 then the last of the four counters is used.

In the case of a branch that is alternatingly taken and not taken, the branch history register will always contain either 01 or 10. When the history is 01 we will use the counter with the binary number 01B in the pattern history table. This counter will soon learn that after 01 comes a 0. Likewise, counter number 10B will learn that after 10 comes a 1. After a short learning period, the predictor will make 100% correct predictions. Counters number 00B and 11B will not be used in this case.

A branch that is alternatingly taken twice and not taken twice will also be predicted 100% by this predictor. The repetitive pattern is 0011-0011-0011. Counter number 00B in the pattern history table will learn that after 00 comes a 1. Counter number 01B will learn that after a 01 comes a 1. Counter number 10B will learn that after 10 comes a 0. And counter number 11B will learn that after 11 comes a 0. But the repetitive pattern 0001-0001-0001 will not be predicted correctly all the time because 00 can be followed by either a 0 or a 1.

The mechanism in figure 12.2 is called a two-level adaptive predictor. The general rule for a two-level adaptive predictor with an *n*-bit branch history register is as follows:

> Any repetitive pattern with a period of *n*+1 or less can be predicted perfectly after a warm-up time no longer than three periods. A repetitive pattern with a period *p* higher than *n*+1 and less than or equal to 2^n can be predicted perfectly if all the *p* *n*-bit subsequences are different.

To illustrate this rule, consider the repetitive pattern 0011-0011-0011 in the above example. The 2-bit subsequences are 00, 01, 11, 10. Since these are all different, they will use different counters in the pattern history table of a two-level predictor with *n* = 2. With *n* = 4, we can predict the repetitive pattern 000011-000011-000011 with period 6, because the six 4-bit subsequences: 0000, 0001, 0011, 0110, 1100, 1000, are all different. But the pattern 000001-000001-000001, which also has period 6, cannot be predicted perfectly, because the subsequence 0000 can be followed by either a 0 or a 1.

The PMMX, PPro, P2 and P3 all use a two-level adaptive predictor with *n* = 4. This requires 36 bits of storage for each branch: two bits for each of the 16 counters in the pattern history table, and 4 bits for the branch history register.

The agree predictor

Since the storage requirement for the two-level predictor grows exponentially with the number of history bits *n*, there is a practical limit to how big we can make *n*. One way of overcoming this limitation is to share the branch history buffer and the pattern history table among all the branches rather than having one set for each branch.

Imagine a two-level predictor with a global branch history register, storing the history of the last *n* branches, and a shared pattern history table. The prediction of a branch is made on the basis of the last *n* branch events. Some or all of these events may be occurrences of the same branch. If the innermost loop contains *m* conditional jumps, then the prediction of a branch within this loop can rely on $\text{floor}(n/m)$ occurrences of the same branch in the branch history register, while the rest of the entries come from other branches. If this is enough for defining the pattern of this branch, or if it is highly correlated with the other branches, then we can expect the prediction rate to be good.

The disadvantage of this method is that branches that behave differently may share the same entry in the pattern history table. This problem can be reduced by storing a biasing bit for each branch. The biasing bit indicates whether the branch is mostly taken or not taken. The predictors in the pattern history table now no longer indicate whether the branch is predicted to be taken or not, but whether it is predicted to go the same way or the opposite way of the biasing bit. Since the prediction is more likely to agree than to disagree with the biasing bit, the probability of negative interference between branches that happen to use the same entry in the pattern history table is reduced, but not eliminated. My research indicates that the P4 is using one version of this method, as shown in figure 12.3.

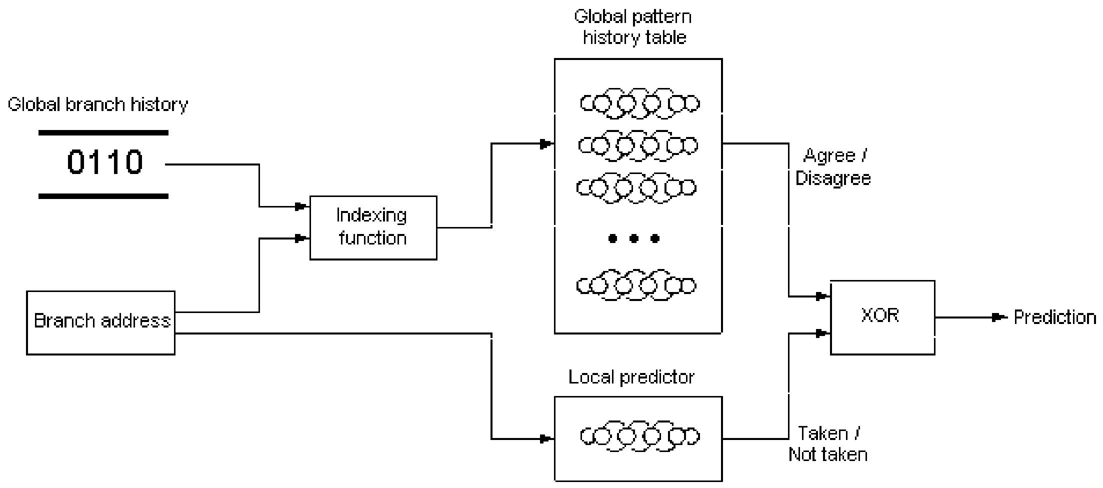

Figure 12.3. Agree predictor

Each branch has a local predictor, which is simply a saturating counter of the same type as shown in figure 12.1. The global pattern history table, which is indexed by the global branch history register, indicates whether the branch is predicted to agree or disagree with the output of the local predictor.

The global branch history register has 16 bits in the P4. Since, obviously, some of the 2^{16} different history patterns are more common than others, we have the problem that some entries in the pattern history table will be used by several branches, while many other entries will never be used at all, if the pattern history table is indexed by the branch history alone. In order to make the use of these entries more evenly distributed, and thus reduce the probability that two branches use the same entry, the pattern history table may be indexed by a combination of the global history and the branch address. The literature recommends that the index into the pattern history table is generated by an XOR combination of the history bits and the branch address (E. Sprangle, et. al.: The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference. Proceedings of the 24th International Symposium on Computer Architecture, Denver, June 1997). However, my experimental results do not confirm such a design. The indexing function in figure 12.3 may be a more complex hashing function of the history and the branch address, or it may involve the branch target address, BTB entry address or trace cache address.

Since the indexing function is not known, it is impossible to predict whether two branches will use the same entry in the pattern history table. For the same reason, I have not been able to calculate the size of the pattern history table. The most logical value is 2^{16}, but in theory it may possibly be both bigger and smaller.

Future branch prediction methods

It is likely that branch prediction methods will be further improved in the future. According to the technical literature and the patent literature, the following developments are likely:

- alloyed predictor. The two-level predictor can be improved by using a combination of local and global history bits as index into the pattern history table. This eliminates the need for the agree predictor and improves the prediction of branches that are not correlated with any preceding branch.

- hybrid predictor. There may be two or more different predictors, for example one based on local history and one based on global history. A local selector keeps track of which of the two predictors has been most successful in the past and selects the output of the best predictor for a particular branch.

- switch counter. A local counter keeps track of how many consecutive "taken" and how many consecutive "not taken" each branch makes. This is useful for predicting loops that always repeat the same number of times. The switch counter may be part of a hybrid predictor.

- keeping unimportant branches out of global history register. In typical programs, a large proportion of the branches always go the same way. Such branches may be kept out of the global history register in order to increase its information contents.

- decoding both branches. Part or all of the pipeline may be duplicated so that both branches can be decoded and speculatively executed simultaneously. It may decode both branches whenever possible, or only if the prediction is uncertain.

- neural networks. The storage requirement for the two-level predictor grows exponentially with *n*, and the warm-up time may also grow exponentially with *n*. This limits the performance that can be achieved with the two-level predictor. Other methods with less storage requirements are likely to be implemented. Such new methods may use the principles of neural networks.

- reducing the effect of context switches. The information that the predictors have collected is often lost due to task switches and other context switches. As more advanced prediction methods require longer warm-up time, it is likely that new methods will be implemented to reduce the loss during context switches.

12.2 Branch prediction in P1

The branch prediction mechanism for the P1 is very different from the other processors. Information found in Intel documents and elsewhere on this subject is misleading, and following the advices given is such documents is likely to lead to sub-optimal code.

The P1 has a branch target buffer (BTB), which can hold information for up to 256 jump instructions. The BTB is organized as 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.

Each entry contains a saturation counter of the type shown in figure 12.1. Apparently, the designers couldn't afford to use an extra bit for indicating whether the BTB entry is used or not. Instead, they have equated state "strongly not taken" with "entry unused". This makes sense because a branch with no BTB entry is predicted to be not taken anyway, in the P1. A branch doesn't get a BTB entry until the first time it is taken. Unfortunately, the designers have decided that a branch that is taken the first time it is seen should go to state "strongly taken". This makes the state diagram for the predictor look like this:

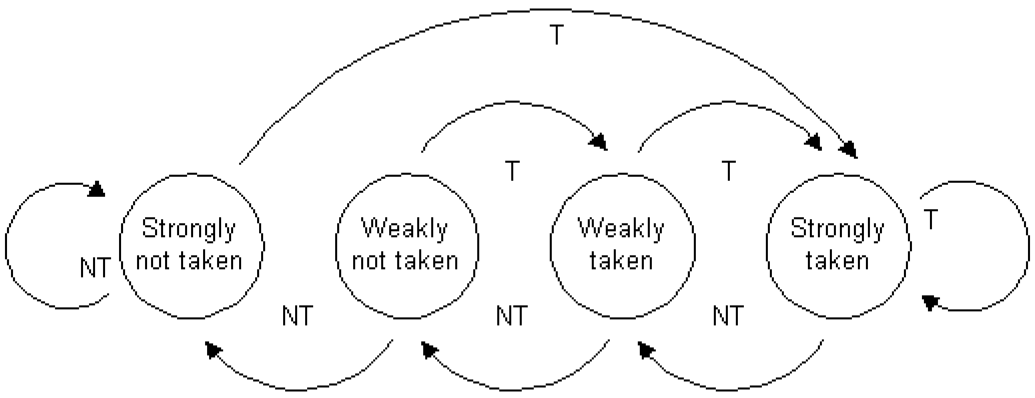

Figure 12.4. Branch predictor in P1

This is of course a sub-optimal design, and I have strong indications that it is a design flaw. In a tight loop with no more than four instruction pairs, where the loop control branch is seen again before the BTB has had the time to update, the output of the saturation counter is forwarded directly to the prefetcher. In this case the state can go from "strongly not taken" to "weakly not taken". This indicates that the originally intended behavior is as in figure 12.1. Intel engineers have been unaware of this flaw until I published my findings in an earlier version of this manual.

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. You may take this asymmetry into account by organizing your branches so that they are taken more often than not.

BTB is looking ahead (P1)

The BTB mechanism in the P1 is counting instruction pairs, rather than single instructions, so you have to know how instructions are pairing (see page 51) in order to analyze where a BTB entry is stored. The BTB entry for any control transfer 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 predicts the branch to be taken, then the P1 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.

Instructions are seldom pairing the first time they are executed (see page 51). 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 P1 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.

Consecutive branches

When a jump is mispredicted, then the pipeline gets flushed. If the next instruction pair executed also contains a control transfer instruction, then the P1 will not 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 we 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 that 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.

To summarize, consecutive jumps can lead to the following problems in the P1:

- failure to load a jump target when the pipeline is being flushed by a preceding mispredicted jump.

- a BTB entry being misapplied to non-jumping instructions and predicting them to jump.

- a second consequence of the above is that a misapplied 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 in the P1. 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
```

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 misapplied 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 misapplied 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 misapplied 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 misapplied 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.

12.3 Branch prediction in PMMX, PPro, P2, and P3

BTB organization

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, P2 and P3 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, P2 and P3 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 that 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.

Misprediction penalty

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, P2 and P3, the misprediction penalty is higher 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, P2 and P3.

Pattern recognition for conditional jumps

The PMMX, PPro, P2 and P3 all use a two-level adaptive branch predictor with a local 4-bit history, as explained on page 35. Simple repetitive patterns are predicted well by this mechanism. For example, a branch which is alternatingly taken twice and not taken twice, will be predicted all the time after a short learning period. The rule on page 36 tells which repetitive branch patterns can be predicted perfectly. All patterns with a period of five or less are predicted perfectly. This means that a loop which always repeats five times will have no mispredictions, but a loop that repeats six or more times will not be predicted. Repetitive patterns with a longer period can also be predicted if all 4-bit subsequences are different. For example, a branch which is alternatingly taken four times and not taken four times, will be predicted perfectly.

The branch prediction mechanism is also 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. Likewise, a branch which switches back and forth between two different regular patterns is predicted well.

Completely random patterns

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 taken and not taken:

fraction of taken/not taken	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	0.417
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 that has no regularities.

Tight loops (PMMX)

Branch prediction in the PMMX 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.

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 that 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.

Tight loops on PPro, P2 and P3 are predicted normally, and take minimum two clock cycles per iteration.

Indirect jumps and calls (PMMX, PPro, P2 and P3)

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.

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, P2 and P3 they are predicted using pattern recognition, but the `LOOP` instruction is still inferior to `DEC ECX` / `JNZ`.

12.4 Branch prediction in P4

The organization of the branch target buffer (BTB) in the P4 is not known in detail. It probably has 512 entries or more organized as 8 ways * 64 sets. Apparently, control transfer instructions that are spaced 4096 bytes apart have the same set-value and may therefore occasionally push each other out of the BTB. Far jumps, calls and returns are not predicted in the P4.

The P4 allocates a BTB entry to any near control transfer instruction the first time it jumps. A branch instruction which never jumps will stay out of the BTB on the P4, but not out of the branch history buffer. 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. All conditional jumps, including `JECXZ` and `LOOP`, contribute to the branch history buffer. Unconditional and indirect jumps, calls and returns do not contribute to the branch history.

Branch mispredictions are much more expensive on the P4 than on previous generations of microprocessors. The time it takes to recover from a misprediction is rarely less than 24 clock cycles, and typically 45 uops. Apparently, the microprocessor cannot cancel a bogus uop before it has reached the retirement stage. This means that if you have a lot of uops with long latency or poor throughput, then the penalty for a misprediction may be as high as 100 clock cycles or more. It is therefore very important to organize code so that the number of mispredictions is minimized.

Pattern recognition for conditional jumps

The P4 uses an "agree" predictor with a 16-bit global history, as explained on page 37. According to the prediction rule on page 36, the P4 can predict any repetitive pattern with a period of 17 or less, as well as some patterns with higher history. However, this applies to the global history, not the local history. You therefore have to look at the preceding branches in order to determine whether a branch is likely to be well predicted. I will explain this with the following example:

```
    MOV  EAX, 100
A:  ...
    ...
    MOV  EBX, 16
B:  ...
    SUB  EBX, 1
    JNZ  B
    TEST EAX, 1
    JNZ  X1
    CALL EAX_IS_EVEN
    JMP  X2
X1: CALL EAX_IS_ODD
X2: ...
    MOV  ECX, 0
C1: CMP  ECX, 10
    JNB  C2
    ...
    ADD  ECX, 1
    JMP  C1
C2: ...
    SUB  EAX, 1
```

```
        JNZ  A
```

The A loop repeats 100 times. The JNZ A instruction is taken 99 times and falls through 1 time. It will be mispredicted when it falls through. The B and C loops are inside the A loop. The B loop repeats 16 times, so without considering the prehistory, we would expect it to be predictable. But we have to consider the prehistory. With the exception of the first time, the prehistory for JNZ B will look like this: JNB C2: not taken 10 times, taken 1 time (JMP C1 does not count because it is unconditional); JNZ A taken; JNZ B taken 15 times, not taken 1 time. This totals 17 consecutive taken branches in the global history before JNZ B is not taken. It will therefore be mispredicted once or twice for each cycle. There is a way to avoid this misprediction. If you insert a dummy branch that always falls through anywhere between the A: and B: labels, then JNZ B is likely to be predicted perfectly, because the prehistory now has a not taken before the 15 times taken. The time saved by predicting JNZ B well is far more than the cost of an extra dummy branch. The dummy branch may, for example, be TEST ESP,ESP / JC B.

JNZ X1 is taken every second time and is not correlated with any of the preceding 16 conditional jump events, so it will not be predicted well.

Assuming that the called procedures do not contain any conditional jumps, the prehistory for JNB C2 is the following: JNZ B taken 15 times, not taken 1 time; JNZ X1 taken or not taken; JNB C2: not taken 10 times, taken 1 time. The prehistory of JNB C2 is thus always unique. In fact, it has 22 different and unique prehistories, and it will be predicted well. If there was another conditional jump inside the C loop, for example if the JMP C1 instruction was conditional, then the JNB C2 loop would not be predicted well, because there would be 20 instances between each time JNB C2 is taken.

In general, a loop cannot be predicted well on the P4 if the repeat count multiplied by the number of conditional jumps inside the loop exceeds 17.

Alternating branches

While the C loop in the above example is predictable, and the B loop can be made predictable by inserting a dummy branch, we still have a big problem with the JNZ X1 branch. This branch is alternatingly taken and not taken, and it is not correlated with any of the preceding 16 branch events. Let's study the behavior of the predictors in this case. If the local predictor starts in state "weakly not taken", then it will alternate between "weakly not taken" and "strongly not taken" (see figure 12.1). If the entry in the global pattern history table starts in an agree state, then the branch will be predicted to fall through every time, and we will have 50% mispredictions (see figure 12.3). If the global predictor happens to start in state "strongly disagree", then it will be predicted to be taken every time, and we still have 50% mispredictions. The worst case is if the global predictor starts in state "weakly disagree". It will then alternate between "weakly agree" and "weakly disagree", and we will have 100% mispredictions. There is no way to control the starting state of the global predictor, but we can control the starting state of the local predictor. The local predictor starts in state "weakly not taken" or "weakly taken", according to the rules of static prediction, explained on page 47 below. If we swap the two branches and replace JNZ with JZ, so that the branch is taken the first time, then the local predictor will alternate between state "weakly not taken" and "weakly taken". The global predictor will soon go to state "strongly disagree", and the branch will be predicted correctly all the time. A backward branch that alternates would have to be organized so that it is not taken the first time, to obtain the same effect. Instead of swapping the two branches, we may insert a 3EH prediction hint prefix immediately before the JNZ X1 to change the static prediction to "taken" (see p. 47). This will have the same effect.

While this method of controlling the initial state of the local predictor solves the problem in most cases, it is not completely reliable. It may not work if the first time the branch is seen is after a mispredicted preceding branch. Furthermore, the sequence may be broken by a task switch or other event that pushes the branch out of the BTB. We have no way of knowing whether the branch is taken or not taken the first time it is seen after such an event. Fortunately, it appears that the designers have been aware of this problem and implemented a way to solve it, though the method is undocumented. While researching these mechanisms, I discovered an undocumented prefix, `64H`, which does the trick on the P4. This prefix doesn't change the static prediction, but it controls the state of the local predictor after the first event so that it will toggle between state "weakly not taken" and "weakly taken", regardless of whether the branch is taken or not taken the first time. This trick can be summarized in the following rule:

A branch which is taken exactly every second time, and which doesn't correlate with any of the preceding 16 branch events, can be predicted well on the P4 if it is preceded by a `64H` prefix. This prefix is coded in the following way:

```
    DB   64H          ; hint prefix for alternating branch
    JNZ  X1           ; branch instruction
```

No prefix is needed if the branch can see a previous instance of itself in the 16-bit prehistory.

The `64H` prefix has no effect and causes no harm on any other microprocessor. It is an `FS` segment prefix. The x86 family microprocessors are designed to ignore redundant and meaningless prefixes. The `64H` prefix cannot be used together with the `2EH` and `3EH` static prediction prefixes.

Completely random branch patterns

The powerful capability of branch pattern recognition has a minor drawback in the case of completely random sequences with no regularities. 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. The list of misprediction rates for random branches on page 43 also applies to the P4.

12.5 Indirect jumps (all processors)

While an unconditional jump always goes to the same target, indirect jumps, indirect calls, and returns may go to a different address each time. The prediction method for an indirect jump or indirect call is, in all processors, simply to predict that it will go to the same target as last time it was executed. The first time an indirect jump or indirect call is seen, it is predicted to go to the immediately following instruction. Therefore, an indirect jump or call should always be followed by valid code. Don't place a list of jump addresses immediately after an indirect jump or call. Such a list should preferably be placed in the data segment, rather than the code segment.

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.

12.6 Returns (all processors except P1)

A better method is used for returns. A Last-In-First-Out buffer, called the return stack buffer, remembers the return address every time a call instruction is executed, and uses this for predicting where the corresponding return will go. This mechanism makes sure that return instructions are correctly predicted when the same subroutine is called from several different locations.

The P1 has no return stack buffer, but uses the same method for returns as for indirect jumps. Later processors have a return stack buffer. The size of this buffer is 4 in the PMMX, and 16 in PPro, P2, P3, and P4. This size may seem rather small, but it is sufficient in most cases because only the innermost subroutines matter in terms of execution time. The return stack buffer may be insufficient, though, in the case of a deeply nesting recursive function.

In order to make this mechanism work, you must make sure that all calls are matched with returns. Never jump out of a subroutine without a return and never use a return as an indirect jump. It is OK, however, to replace a `CALL MYPROC` / `RET` sequence with `JMP MYPROC`.

On the P4, you also must make sure that far calls are matched with far returns and near calls with near returns. This may be problematic because the assembler will replace a far call to a procedure in the same segment with `PUSH CS` followed by a near call. Even if you prevent the assembler from doing this by hard-coding the far call, the linker is likely to translate the far call to `PUSH CS` and a near call. Use the NOFARCALLTRANSLATION option in the linker to prevent this. It is recommended to use a small or flat memory model so that you don't need far calls, because far calls and returns are expensive anyway.

12.7 Static prediction

The first time a branch instruction is seen, a prediction is made according to the principles of static prediction.

Static prediction in P1 and PMMX

A control transfer instruction which has not been seen before or which is not in the branch target buffer (BTB) is always predicted to fall through on the P1 and PMMX.

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. On the PMMX, it will stay in the BTB no matter how many times it falls through. Any control transfer instruction which jumps to the address immediately following itself will not get a BTB entry and will therefore always have a misprediction penalty.

Static prediction in PPro, P2, P3 and P4

On PPro, P2, P3 and P4, 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.

On the P4, you can change the static prediction by adding prediction hint prefixes. The prefix `3EH` will make the branch predicted taken the first time, and prefix `2EH` will make it predicted not taken the first time. These prefixes can be coded in this way:

```
    DB  3EH    ; prediction hint prefix
    JBE LL     ; predicted taken first time
```

The prediction hint prefixes are in fact segment prefixes, which have no effect and cause no harm on previous processors.

It is rarely worth the effort to take static prediction into account. Almost any branch that is executed sufficiently often for its timing to have any significant effect is likely to stay in the BTB so that only the dynamic prediction counts. Static prediction only has a significant effect if context switches or task switches occur very often.

Normally you don't have to care about the penalty of static mispredictions. It is more important to organize branches so that the most common path is not taken, because this improves code prefetching, trace cache use, and retirement.

Static prediction does have an influence on the way traces are organized in the trace cache, but this is not a lasting effect because traces may be reorganized after several iterations.

12.8 Close jumps

Close jumps on 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. The `RET` instruction is particularly prone to this problem because it is only one byte long. 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 a 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 jump if you choose this solution.
3. Put in some instruction between the two control transfer 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.

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.

The penalty for chained calls only occurs when the same subroutines are called from more than one location. 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.

Chained jumps on PPro, P2 and P3

A jump, call, or return cannot be executed in the first clock cycle after a previous jump, call, or return on the PPro, P2 and P3. 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.

Chained jumps on P4

The retirement station can handle only one taken jump, call or return per clock cycle, and only in the first of the three retirement slots. Therefore, preferably, no more than every third uop should be a jump.

12.9 Avoiding jumps (all processors)

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

- jump mispredictions are very expensive.

- jump instructions may push one another out of the branch target buffer.
- on the P4, branches may interfere with each other in the global pattern history table.
- on the P4, branches fill up the global branch history register. This may reduce the predictability of subsequent branches.
- there are various penalties for consecutive or chained jumps, depending on the processor.
- a return takes 2 clocks on P1 and PMMX, calls and returns generate 3 - 4 uops on PPro, P2, P3 and P4.
- on PPro, P2 and P3, instruction fetch may be delayed after a jump.
- on PPro, P2, P3 and P4, retirement is less effective for taken jumps then for other uops.
- on P4, the utilization of the trace cache and the delivery from the trace cache is less effective if the code contains many branches.

Eliminating unconditional jumps and calls

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 the 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 this would interfere with the prediction mechanism of the return stack buffer. Instead, you can replace the preceding call with a jump. For example `CALL MYPROC` / `RET` can be replaced by `JMP MYPROC` if `MYPROC` ends with the same kind of `RET`.

You may also eliminate a jump by duplicating 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:      ADD     EDX, 1
        JNZ     A
        MOV     ESP, EBP
        POP     EBP
        RET
```

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

```
A:      CMP     [EAX+4*EDX],ECX
        JE      B
        CALL    X
        ADD     EDX, 1
        JNZ     A
        JMP     D
B:      CALL    Y
C:      ADD     EDX, 1
        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.

Tricks to avoid conditional jumps (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 integer without branching:

```
    CDQ                 ; copy sign of EAX to all bits of EDX
    XOR EAX,EDX         ; toggle all bits if negative
    SUB EAX,EDX         ; add 1 if negative
```

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: `SUB 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: `SUB EAX,EAX` / `SETcond AL` / `NEG EAX`

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

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

Or, for signed numbers:

```
    SUB EAX,EBX
    CDQ
    AND EDX,EAX
    ADD EBX,EDX
```

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

```
    CDQ
    XOR EBX,ECX
    AND EDX,EBX
    XOR EDX,ECX
```

Whether or not such tricks are worth the extra code depends on how predictable a conditional jump would be, and the latency of the extra code. The examples that use `CDQ` are faster than conditional moves on the P4.

Another way of avoiding branches in newer processors is to use the `MAX..`, `MIN..`, `PMAX..` and `PMIN..` instructions and the saturating `PADD..` and `PSUB..` instructions.

You can conditionally move data in memory by using `REP MOVS` with `ECX` = 0 when you don't want to move.

Replacing conditional jumps by conditional moves (PPro, P2, P3, P4)

The PPro, P2, P3 and P4 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 (See page 109 for how to make conditional moves in MMX and XMM registers). For code that will not run on old processors you may replace poorly predictable branches with conditional moves.

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

```
CMP    EAX,EBX
CMOVNB EAX,EBX
```

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 conditional move instructions have two important disadvantages:

1. they make dependence chains longer

2. they introduce an unnecessary dependence on the operand not chosen

A conditional move is waiting for three operands to be ready before it can be executed: the condition flag and the two move operands. You have to consider if any of these three operands are likely to be delayed by dependence 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 may 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.

13 Optimizing for P1 and PMMX

13.1 Pairing integer instructions

Perfect pairing

The P1 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 the 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 page 133).

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 that 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 pointer registers in 16 bit mode or 16 bit pointer registers in 32 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 P1, 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 P1, 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 P1 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.

9. Both instructions must be preloaded and decoded. This is explained in chapter 10 page 30.

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 page 57).
- an MMX instruction which accesses memory or integer registers can execute only in the U-pipe and cannot pair with a non-MMX instruction.

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 instruction suffers an AGI stall (see page 55).

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. The preceding rule is extended to the case where bits 2 - 4 are the same in the two addresses (cache line conflict). For DWORD addresses this means that the difference between the two addresses should not be divisible by 32.

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 that 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 that 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.

13.2 Address generation interlock

It takes one clock cycle to calculate the address needed by an instruction that 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 we have to wait an extra clock cycle for the address to be calculated. This is called an AGI stall.

```
    ; Example 13.1
    ADD EBX,4
    MOV EAX,[EBX] ; AGI stall
```

The stall in this example can be removed by putting some other instructions in between these two, or by rewriting the code to:

```
    ; Example 13.2
    MOV EAX,[EBX+4]
    ADD EBX,4
```

You can also get an AGI stall with instructions that 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 P1 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 that has been changed in the preceding clock cycle. Example:

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

PPro, P2 and P3 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.

13.3 Splitting complex instructions into simpler ones

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 that 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 that 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 does not 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, P2 and P3, except when the split instructions generate fewer uops.

13.4 Prefixes

An instruction with one or more prefixes may not be able to execute in the V-pipe (see page 52), and it may take more than one clock cycle to decode.

On the P1, 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 P1 is that any instruction that 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 that takes more than one clock cycle to execute, and any instruction that 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 were 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 P1 (The PMMX has no decoding delay for the 0FH).

13.5 Scheduling floating-point code

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 one. 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 that 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 FLD or FCOMPP is paired with FXCH.

By the proper use of FXCH instructions you may obtain a lot of overlapping in your floating-point code. 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 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    ST(1)   ; clock cycle 4
        FMUL    [b2]    ; clock cycle 5-7    (stall)
        FXCH    ST(2)   ; clock cycle 5
        FMUL    [c2]    ; clock cycle 7-9    (stall)
        FXCH    ST(1)   ; clock cycle 7
        FSTP    [a3]    ; clock cycle 8-9
        FXCH    ST(1)   ; 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.

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. A complete listing of floating-point instructions, and what they can pair or overlap with, is given on page 135.

There is no penalty for using a memory operand on floating-point instructions because the arithmetic unit is one step later in the pipeline than the read unit. The tradeoff of this comes when a floating-point value is stored 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. In many cases you cannot hide this type of stall without scheduling the 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] / FIMUL [b]
```

Split up into:

```
        FILD [a] / FILD [b] / FMUL
```

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

14 Optimizing for PPro, P2, and P3

14.1 The pipeline in PPro, P2 and P3

The architecture of the PPro, P2 and P3 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. Next, it goes to the instruction decoders. There are three decoders so that up to three instructions can be decoded 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 uops. Simple instructions generate only one uop, while more complex instructions may generate several uops. 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 works on temporary registers, which are later written to the permanent registers EAX, EBX, etc, as explained on page 32. The purpose of the RAT is to allow register renaming and to tell the uops which temporary registers to use.

After the RAT, the uops go to 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.

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

Instruction decoding

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 below)

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 interspersed 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 section, 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 starting on page 138.

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

1. 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
```

2. 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.

3. Segment prefixes are used when you address data in a non-default data segment. Segment prefixes give no penalty on the PPro, P2 and P3.

4. Repeat prefixes and lock prefixes give no penalty in the decoders.

5. There is always a penalty if you have more than one prefix. This penalty is usually one clock per prefix.

Instruction fetch

The code is fetched in aligned 16-bytes chunks from the code cache and placed in the double buffer. The code is then taken from the double buffer and fed to the decoders in IFETCH blocks, which are usually 16 bytes long, but not necessarily aligned by 16. The purpose of the double buffer is to allow instruction fetching across 16 byte boundaries.

The double buffer can fetch one 16-byte 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 page 42 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	16-byte boundary in this IFETCH	16-byte boundary in first instruction after	decoder delay	alignment of first IFETCH after

block containing jump	block	jump		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 this 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. 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 count 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 biggest problem is to know 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 on page 42, 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 partially serializing instructions such as `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. Some 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 page 65). 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 in its shortest form. There is also a 2-byte version (DB 0FFH, 0C9H), and SUB ECX,1 is 3 bytes. You can even code a 6 bytes long version with a long immediate operand using this trick:

```
    SUB ECX, 99999
    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. There is no guarantee, however, that meaningless prefixes will have no effect on future processors. In fact, many of the new instructions that have been added to the P3 and P4 processors are coded by adding repeat or operand size prefixes to existing opcodes. It is quite safe, however, to add a DS: segment prefix to any instruction that has a memory operand, including LEA.

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

14.2 Register renaming

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.

14.3 Register read stalls

A possible limitation, which applies only to the PPro, P2 and P3 processors, is that the ROB can only read two different permanent registers 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, dependence 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 that 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 a 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. Example:

```
FSTP QWORD PTR [EBX+8*ECX]
```

The first uop reads `ST(0)`, the second uop reads `EBX` and `ECX`.

Read and modify
An instruction that reads a memory operand and modifies a register by some arithmetic or logic 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]
```

The first uop reads `ESI`, the 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 that 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.

14.4 Out of order execution

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.

Writes to memory cannot execute out of order relative to other writes in the PPro, P2 and P3. 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 a 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 best 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. On page 138 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 on page 138 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 execution unit. 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 square root 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 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+8] / 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.

14.5 Retirement

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 page 42) 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.

14.6 Partial register stalls

Partial register stall is a problem that occurs in PPro, P2 and P3 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, P2 and P3, 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, P2 and P3 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, and this 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 dependence on earlier instructions on PPro, P2 and P3 (but it does on P4). Example:

```
    DIV EBX
    MOV [MEM], EAX
    MOV EAX, 0                ; break dependence
    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
```

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 in the assembly code 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
    JZ  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.

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.

14.7 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
```

14.8 Bottlenecks in PPro, P2, P3

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 another bottleneck 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 (page 28), and avoid long dependence chains after a data read cache miss.

If you have many divisions, then try to reduce them (page 114) and make sure the processor has something else to do during the divisions.

Dependence chains tend to hamper out-of-order execution (page 33). Try to break long dependence 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 poorly predictable conditional jumps with conditional moves if possible, and replace small procedures with macros (page 48).

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 (page 70).

If you aim at a throughput of 3 uops per clock cycle then be aware of possible delays in instruction fetch and decoding (page 59), especially in small loops. Instruction decoding is often the narrowest bottleneck in these processors, and unfortunately this factor is quite complicated to optimize.

The limit of two permanent register reads per clock cycle may reduce your throughput to less than 3 uops per clock cycle (page 65). 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 (68).

The retirement station can handle 3 uops per clock, but may be slightly less effective for taken jumps (page 69).

15 Optimizing for P4

15.1 Trace cache

The pipeline of the 7'th generation microprocessor P4 is similar to the pipeline of the 6'th generation microprocessors, with one important difference: Instructions are cached *after* being decoding into uops. Rather than storing instruction opcodes in the level-1 cache, it stores decoded uops. An important reason for this is that the decoding step was a bottleneck on earlier processors. An opcode can have any length from 1 to 15 bytes. It is quite complicated to determine the length of an instruction opcode; and you have to know the length of the first opcode in order to know where the second opcode begins. Therefore, it is difficult to determine opcode lengths in parallel. The 6'th generation microprocessors could decode three instructions per clock cycle. This may be more difficult at higher clock speeds (though I am convinced that it is possible). If uops all have the same size, then the processor can handle them in parallel, and the bottleneck disappears. This is the principle of RISC processors. Caching uops rather than opcodes enables the P4 to use RISC technology on a CISC instruction set. The cache that holds uops is called a trace cache. It stores traces of consecutive uops. A trace is a string of uops that are executed in sequence, even if they are not sequential in the original code.

The uops take more space than opcodes on average. It can be calculated from the information contents that each uop requires at least 36 bits. We can therefore estimate that the size of each entry in the trace cache is at least 36 bits, probably more.

The on-chip level-2 cache is used for both code and data. The size of the level-2 cache is 512 kb in most models. It runs at full speed with a 256 bits wide data bus to the central processor, and is quite efficient.

The trace cache seems to be organized as 2048 lines of 6 entries each, 4-way set-associa-tive. 16 of the bits in each entry are reserved for data. This means that a uop that requires more than 16 bits of data storage must use two entries. You can calculate whether a uop uses one or two trace cache entries by the following rules, which have been obtained experimentally.

1. A uop with no immediate data and no memory operand uses only one trace cache entry.

2. A uop with an 8-bit or 16-bit immediate operand uses one trace cache entry.

3. A uop with a 32-bit immediate operand in the interval from -32768 to +32767 uses one trace cache entry. The immediate operand is stored as a 16-bit signed integer. If an opcode contains a 32-bit constant, then the decoder will investigate if this constant is within the interval that allows it to be represented as a 16-bit signed integer. If this is the case, then the uop can be contained in a single trace cache entry.

4. If a uop has an immediate 32-bit operand outside the $\pm 2^{15}$ interval so that it cannot be represented as a 16-bit signed integer, then it will use two trace cache entries unless it can borrow storage space from a nearby uop.

5. A uop in need of extra storage space can borrow 16 bits of extra storage space from a nearby uop that doesn't need its own data space. Almost any uop that has no immediate operand and no memory operand will have an empty 16-bit data space for other uops to borrow. A uop that requires extra storage space can borrow space from the next uop as well as from any of the preceding 3 - 5 uops (5 if it is not number 2 or 3 in a trace cache line), even if they are not in the same trace cache line. A uop cannot borrow space from a preceding uop if any uop between the two is double size or has borrowed space. Space is preferentially borrowed from preceding

rather than subsequent uops.

6. The displacement of a near jump, call or conditional jump is stored as a 16-bit signed integer, if possible. An extra trace cache entry is used if the displacement is outside the $\pm 2^{15}$ range and no extra storage space can be borrowed according to rule 5 (Displacements outside this range are rare).

7. A memory load or store uop will store the address or displacement as a 16-bit integer, if possible. This integer is signed if there is a base or index register, otherwise unsigned. Extra storage space is needed if a direct address is $\geq 2^{16}$ or an indirect address (i.e. with one or two pointer registers) has a displacement outside the $\pm 2^{15}$ interval.

8. Memory load uops can *not* borrow extra storage space from other uops. If 16 bits of storage is insufficient then an extra trace cache entry will be used, regardless of borrowing opportunities.

9. Most memory store instructions generate two uops: The first uop, which goes to port 3, calculates the memory address. The second uop, which goes to port 0, transfers the data from the source operand to the memory location calculated by the first uop. The first uop can always borrow storage space from the second uop. This space cannot be borrowed to any other uop, even if it is empty.

10. Store operations with an 8, 16, or 32-bit register as source, and no SIB byte, can be contained in a single uop. These uops can borrow storage space from other uops, according to rule 5 above.

11. Segment prefixes do not require extra storage space.

12. A uop cannot have both a memory operand and an immediate operand. An instruction that contains both will be split into two or more uops. No uop can use more than two trace cache entries.

13. A uop that requires two trace cache entries cannot cross a trace cache line boundary. If a double-space uop would cross a 6-entry boundary in the trace cache then an empty space will be inserted and the uop will use the first two entries of the next trace cache line.

The difference between load and store operations needs an explanation. My theory is as follows: No uop can have more than two input dependencies (not including segment registers). Any instruction that has more than two input dependencies needs to be split up into two or more uops. Examples are `ADC` and `CMOVcc`. A store instruction like `MOV [ESI+EDI],EAX` also has three input dependencies. It is therefore split up into two uops. The first uop calculates the address `[ESI+EDI]`, the second uop stores the value of `EAX` to the calculated address. In order to optimize the most common store instructions, a single-uop version has been implemented to handle the situations where there is no more than one pointer register. The decoder makes the distinction by seeing if there is a SIB byte in the address field of the instruction. A SIB byte is needed if there is more than one pointer register, or a scaled index register, or `ESP` as base pointer. Load instructions, on the other hand, can never have more than two input dependencies. Therefore, load instructions are implemented as single-uop instructions in the most common cases. The load uops need to contain more information than the store uops. In addition to the type and number of the destination register, it needs to store any segment prefix, base pointer, index pointer, scale factor, and displacement. The size of the trace cache entries has probably been chosen to be exactly enough to contain this information. Allocating a few more bits for the load uop to indicate where it is borrowing storage space from would mean that all trace cache entries would have a bigger size. Given the physical constraints on the trace cache, this would mean fewer entries. This is probably the reason why memory load uops cannot borrow storage space. The store instructions do not have this problem because the necessary information is already split up between two uops unless there is no SIB byte, and hence less information to contain.

The following examples will illustrate the rules for trace cache use:

```
ADD EAX,10000   ; The constant 10000 uses 32 bits in the opcode, but
                ; can be contained in 16 bits in uop. Uses 1 space.
ADD EBX,40000   ; The constant is bigger than 2^15, but it can borrow
                ; storage space from the next uop.
ADD EBX,ECX     ; Uses 1 space. Gives storage space to preceding uop.
MOV EAX,[MEM1]  ; Requires 2 spaces, assuming that address ≥ 2^16;
                ; preceding borrowing space is already used.
MOV EAX,[ESI+4] ; Requires 1 space.
MOV [SI],AX     ; Requires 1 space.
MOV AX,[SI]     ; Requires 2 uops taking one space each.
MOVZX EAX,WORD PTR[SI]    ; Requires 1 space.
MOVDQA XMM1,ES:[ESI+100H] ; Requires 1 space.
FLD QWORD PTR ES:[EBP+8*edx+16] ; Requires 1 space.
MOV [EBP+4], EBX ; Requires 1 space.
MOV [ESP+4], EBX ; Requires 2 uops because SIB byte needed.
FSTP DWORD PTR [MEM2] ; Requires 2 uops. The first uop borrows
                      ; space from the second one.
```

No further data compression is used in the trace cache besides the methods mentioned above. A program that has a lot of direct memory addresses will typically use two trace cache entries for each data access, even if all memory addresses are within the same narrow range. In a flat memory model, the address of a direct memory operand uses 32 bits in the opcode. Your assembler listing will typically show addresses lower than 2^{16}, but these addresses are relocated twice before the microprocessor sees them. The first relocation is done by the linker; the second relocation is done by the loader when the program is loaded into memory. When a flat memory model is used, the loader will typically place the entire program at a virtual address space beginning at a value > 2^{16}. In some systems it is possible to specify that the program should be loaded at a lower virtual address. This may save space in the trace cache if you have many direct memory references in the critical part of your code. However, a better solution might be to access data in the critical part of your program through pointers. In high-level languages like C++, local data are always saved on the stack and accessed through pointers. Direct addressing of global and static data can be avoided by using classes and member functions. Similar methods may be applied in assembly programs.

Uops can be delivered from the trace cache to the next steps in the execution pipeline at a rate of one trace cache line every two clock cycles. This corresponds to a throughput of 3 single-size uops per clock cycle for contiguous code. If there are no bottlenecks further down the pipeline, then you can get a throughput of 3 trace cache entries per clock cycle.

You can therefore calculate with a time consumption of ⅓ clock cycle for each single-entry uop, and ⅔ clock cycles for a double-size uop. If a double-size uop happens to cross a trace cache line boundary, then you will get an extra empty entry at the cost of a further ⅓ clock cycle. The chance of this happening is 1/6. So the average time consumption for a double-size uop is ⅔ + 1/6 * ⅓ = 0.72 clock cycles.

You can prevent double-size uops from crossing 6-entry boundaries by scheduling them so that there is an even number (including 0) of single-size uops between any two double-size uops (A long, continuous 2-1-2-1 pattern will also do). Example:

```
MOV EAX, [MEM1]   ; 1 uop, 2 TC entries
ADD EAX, 1        ; 1 uop, 1 TC entry
MOV EBX, [MEM2]   ; 1 uop, 2 TC entries
MOV [MEM3], EAX   ; 1 uop, 2 TC entries
```

```
    ADD EBX, 1          ; 1 uop, 1 TC entry
```

If we assume, for example, that the first uop here starts at 6-entry boundary, then the `MOV [MEM3],EAX` uop will cross the next 6-entry boundary at the cost of an empty entry. This can be prevented by re-arranging the code:

```
    MOV EAX, [MEM1]     ; 1 uop, 2 TC entries
    MOV EBX, [MEM2]     ; 1 uop, 2 TC entries
    ADD EAX, 1          ; 1 uop, 1 TC entry
    ADD EBX, 1          ; 1 uop, 1 TC entry
    MOV [MEM3], EAX     ; 1 uop, 2 TC entries
```

We cannot know whether the first two uops are crossing any 6-entry boundary as long as we haven't looked at the preceding code, but we can be certain that the `MOV [MEM3],EAX` uop will not cross a boundary, because the second entry of the first uop cannot be the first entry in a trace cache line. If a long code sequence is arranged so that there is never an odd number of single-size uops between any two double-size uops then we will not waste any trace cache entries. The preceding two examples assume that direct memory operands are bigger than 2^{16}, which is usually the case. For the sake of simplicity, I have used only instructions that generate 1 uop each in these examples. For instructions that generate more than one uop, you have to consider each uop separately.

Branches

The uops in the trace cache are not stored in the same order as the original code. If a branching uop jumps most of the time, then the traces will usually be organized so that the jumping uop is followed by the uops jumped to, rather than the uops that follows it in the original code. This reduces the number of jumps between traces. The same sequence of uops can appear more than once in the trace cache if it is jumped to from different places.

Sometimes it is possible to control which of the two branches are stored after a branching uop by using branch hint prefixes (see page 47), but my experiments have shown no consistent advantage of doing so. Even in the cases where there is an advantage by using branch hint prefixes, this effect does not last very long, because the traces are rearranged quite often to fit the behavior of the branch uops. You can therefore assume that traces are usually organized according to the way branches go most often.

The throughput is typically less than 3 uops per clock cycle if the code contains many branches. If, for example, the first entry in a trace cache line is a uop that branches to another trace stored elsewhere in the trace cache, then the next 5 uops in the same trace cache line are lost. This worst-case loss is equivalent to 5/3 = 1.67 clock cycles. There is no loss if the branching uop is the last uop in a trace cache line. The average loss for jumping to a different trace is 5/6 = 0.833 clock. In theory, it might be possible to organize code so that branch uops appear in the end of trace cache lines in order to avoid losses. But attempts to do so are rarely successful because it is almost impossible to predict where each trace begins. Sometimes, a small loop containing branches can be improved by organizing it so that each branch contains a number of trace cache entries divisible by 6. A number of trace cache entries that is slightly less than a value divisible by 6 is better than a number slightly more than a value divisible by 6.

Obviously, these considerations are only relevant if the throughput is not limited by any other bottleneck in the execution units, and the branches are predictable.

Guidelines for improving trace cache performance

The following guidelines can improve performance if the delivery of uops from the trace cache is a bottleneck:

- Prefer instructions that generate few uops. A list of how many uops each instruction generates is given on page 145.

- Keep immediate operands in the range between -2^{15} and $+2^{15}$ if possible. If a uop has an immediate 32-bit operand outside this range, then you should preferably have a uop with no immediate operand and no memory operand before or immediately after the uop with the big operand.

- Avoid direct memory addresses in 32-bit mode. The performance can be improved by using pointers if the same pointer can be used repeatedly and the addresses are within $\pm 2^{15}$ of the pointer register.

- Avoid having an odd number of single-size uops between any two double-size uops. Instructions that generate double-size uops include memory loads with direct memory operands, and other uops with an unmet need for extra storage space.

- Replace branch instructions by conditional moves if this does not imply large extra costs.

15.2 Instruction decoding

In most cases, the decoder generates 1 - 4 uops for each instruction. For complex instructions that require more than 4 uops, the uops are submitted from microcode ROM. The table on page 145 lists the number of decoder uops and microcode uops that each instruction generates.

The decoder can handle instructions at a rate of one instruction per clock cycle. There are a few cases where the decoding of an instruction takes more than one clock cycle:

An instruction that generates micro-code may take more than one clock cycle to decode, sometimes much more. The following instructions, which may (in some cases) generate micro-code, do not take significantly more time to decode: moves to and from segment registers, `ADC`, `SBB`, `IMUL`, `MUL`, `MOVDQU`, `MOVUPS`, `MOVUPD`.

An instruction that has more than one prefix takes one clock cycle for each prefix to decode. Instruction prefixes are used in the following cases:

- Instructions using XMM registers have a prefix specifying the size and precision of data, except for packed single precision float data.
- Instructions using 16-bit registers in 32-bit mode or vice versa have an operand size prefix (except instructions that can only have one data size, such as `FNSTSW AX`).
- Instructions using 16-bit addressing in 32-bit mode or vice versa have an address size prefix.
- Instructions using a non-default data segment have a segment prefix. The default data segment for all explicit memory operands is `DS`, except when `EBP` or `ESP` is used as base pointer, using `SS` as segment. String instructions that use `EDI` as implicit memory pointer use `ES` in association with `EDI`, regardless of segment prefix.
- Repeated string instructions have a repeat prefix.
- Instructions that read, modify, and write a memory operand can have a `LOCK` prefix which prevents other microprocessors from accessing the same memory location until the operation is finished.
- Branch instructions can have a prefix as a hint to aid prediction.
- A few new instructions are formed by old instructions with a prefix added.

Many newer instructions begin with a `0FH` byte. This byte doesn't count as a prefix on this microprocessor.

With a flat memory model, you will probably never need more than one instruction prefix. In a segmented memory model, you will need two prefixes when a segment prefix is used in addition to an operand size prefix or a prefix for an XMM instruction. The order of the prefixes is unimportant.

Decoding time is not important for small loops that fit entirely into the trace cache. If the critical part of your code is too big for the trace cache, or scattered around in many small pieces, then the uops may go directly from the decoder to the execution pipeline, and the decoding speed may be a bottleneck. The level-2 cache is so efficient that you can safely assume that it delivers code to the decoder at a sufficient speed.

If it takes longer time to execute a piece of code than to decode it, then the trace may not stay in the trace cache. This has no negative influence on the performance, because the code can run directly from the decoder again next time it is executed, without delay. This mechanism tends to reserve the trace cache for the pieces of code that execute faster than they decode. I have not found out which algorithm the microprocessor uses to decide whether a piece of code should stay in the trace cache or not, but the algorithm seems to be rather conservative, rejecting code from the trace cache only in extreme cases.

15.3 Execution units

Uops from the trace cache or from the decoder are queued when they are waiting to be executed. After register renaming and reordering, each uop goes through a port to an execution unit. Each execution unit has one or more subunits which are specialized for particular operations, such as addition or multiplication. The organization of ports, execution units, and subunits can be represented as follows:

port	execution unit	subunit	speed
0	alu0	add, sub, mov	double
		logic	
		store integer	
		branch	
0	mov	move and store fp, mmx, xmm	single
		fxch	
1	alu1	add, sub, mov	double
1	int	misc.	single
		borrows subunits from fp and mmx	
1	fp	fp add	half
		fp mul	
		fp div	
		fp misc.	
1	mmx	mmx alu	half
		mmx shift	
		mmx misc.	
2	load	all loads	single
3	store	store address	single

Further explanation can be found in "Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual". The table above deviates slightly from diagrams in the Intel manual in order to account for various delays.

A uop can be executed when the following conditions are met:

- all operands for the uop are ready
- an appropriate execution port is ready
- an appropriate execution unit is ready
- an appropriate execution subunit is ready

Two of the execution units run at double clock speed. This is alu0 and alu1, which are used for integer addition, subtraction and moves. Alu0 can also do logical instructions (and, or, xor), memory store, and branches. These units are highly optimized in order to execute the most common uops as fast as possible. The double clock speed enables these two units to receive a new uop every half-clock cycle. An instruction like `ADD EAX,EBX` can execute in either of these two units. This means that the execution core can handle four integer additions per clock cycle.

Some of the execution units run at half speed. These units are doubled so that they can receive a new uop every clock cycle (see page 82).

The trace cache can submit only three uops per clock cycle to the queue. This sets a limit to the execution speed if all uops are of the type that can execute in alu0 and alu1. The throughput of four uops per clock cycle can thus only be obtained if uops have been queued during a preceding period of lower throughput (due to slow instructions or cache misses). My measurements show that a throughput of four uops per clock cycle can be obtained for a maximum of 11 consecutive clock cycles if the queue has been filled during a preceding period of lower throughput.

Each port can receive one uop for every whole clock tick. Port 0 and port 1 can each receive one additional uop at every half-clock tick, if the additional uop is destined for alu0 or alu1. This means that if a code sequence consists of only uops that go to alu0 then the throughput is two uops per clock cycle. If the uops can go to either alu0 or alu1 then the throughput at this stage can be four uops per clock cycle. If all uops go to the single-speed and half-speed execution units under port 1 then the throughput is limited to one uop per clock cycle. If all ports and units are used evenly, then the throughput at this stage may be as high as six uops per clock cycle.

The single-speed and half-speed execution units can each receive one uop per clock cycle. Some subunits have a lower throughput. For example, the fp-div subunit cannot start a new division before the preceding division is finished, which takes from 23 to 43 clock cycles. Other subunits are perfectly pipelined. For example, a floating-point addition takes 5 clock cycles, but the fp-add subunit can start a new `FADD` operation every clock cycle. In other words, if the first `FADD` operation goes from time T to T+5, then the second `FADD` can start at time T+1 and end at time T+6, and the third `FADD` goes from time T+2 to T+7, etc. Obviously, this is only possible if each `FADD` operation is independent of the results of the preceding ones.

Details about uops, execution units, subunits, throughput and latencies are listed in the tables starting on page 145 for almost all P4 instructions. The following examples will illustrate how to use this table for making time calculations.

```
    FADD ST,ST(1)           ; 0 - 5
    FADD QWORD PTR [ESI]    ; 5 - 10
```

The first `FADD` instruction has a latency of 5 clock cycles. If it starts at time T=0, it will be finished at time T=5. The second `FADD` depends on the result of the first one. Hence, the time is determined by the latency, not the throughput of the fp-add unit. The second addition will start at time T=5 and be finished at time T=10. The second `FADD` instruction generates an additional uop that loads the memory operand. Memory loads go to port 0, while floating-point arithmetic operations go to port 1. The memory load uop can start at time T=0 simultaneously with the first `FADD` or perhaps even earlier. If the operand is in the level-1 or level-2 data cache then we can expect it to be ready before it is needed.

The second example shows how to calculate throughput:

```
    PMULLW XMM1,XMM0    ; 0 - 6
    PADDW XMM2,XMM0     ; 1 - 3
```

```
PADDW MM1,MM0       ; 3 - 5
PADDW XMM3,[ESI]    ; 4 - 6
```

The 128-bit packed multiplication has a latency of 6 and a reciprocal throughput of 2. The subsequent addition uses a different execution unit. It can therefore start as soon as port 1 is vacant. The 128-bit packed additions have a reciprocal throughput of 2, while the 64-bit versions have a reciprocal throughput of 1. Reciprocal throughput is also called issue latency. A reciprocal throughput of 2 means that the second `PADD` can start 2 clocks after the first one. The second `PADD` operates on 64-bit registers, but uses the same execution subunit. It has a throughput of 1, which means that the third `PADD` can start one clock later. As in the previous example, the last instruction generates an additional memory load uop. As the memory load uop goes to port 0, while the other uops go to port 1, the memory load does not affect the throughput. None of the instructions in this example depend on the results of the preceding ones. Consequently, only the throughput matters, not the latency. We cannot know if the four instructions are executed in program order or they are reordered. However, reordering will not affect the overall throughput of the code sequence.

15.4 Do the floating-point and MMX units run at half speed?

Looking at the table on page 151, we notice that almost all the latencies for 64-bit and 128-bit integer and floating-point instructions are even numbers. This suggests that the MMX and FP execution units run at half clock speed. The first explanation that comes to mind is:

Hypothesis 1

We may assume that the P4 has two 64-bit MMX units working together at half speed. Each 128-bit uop will use both units and take 2 clock cycles, as illustrated on fig 15.1. A 64-bit uop can use either of the two units so that independent 64-bit uops can execute at a throughput of one uop per clock cycle, assuming that the half-speed units can start at both odd and even clocks. Dependent 64-bit uops will have a latency of 2 clocks, as shown in fig 15.1.

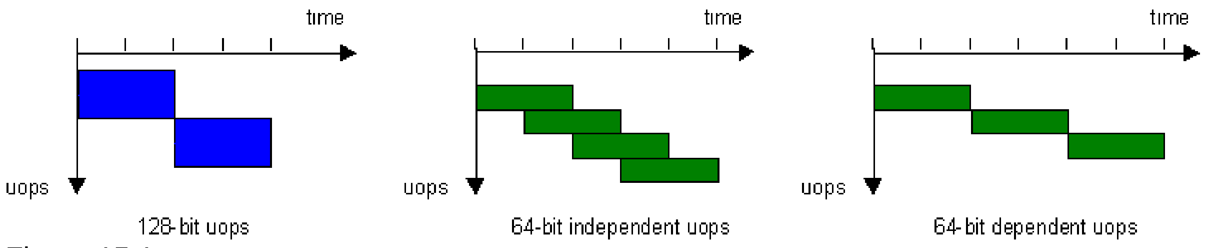

Figure 15.1

The measured latencies and throughputs are in accordance with this hypothesis. In order to test this hypothesis, I have made an experiment with a series of alternating 128-bit and 64-bit uops. Under hypothesis 1, it will be impossible for a 64-bit uop to overlap with a 128-bit uop, because the 128-bit uop uses both 64-bit units. A long sequence of *n* 128-bit uops alternating with *n* 64-bit uops should take 4· *n* clocks, as shown in figure 15.2.

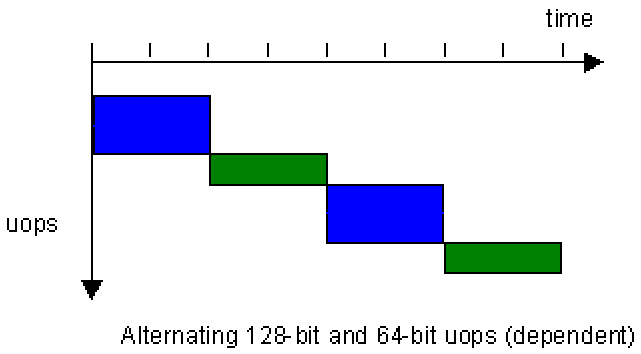

Figure 15.2

However, my experiment shows that this sequence takes only 3· *n* clocks. (I have made the 64-bit uops interdependent, so that they cannot overlap with each other). We therefore have to reject hypothesis 1.

Hypothesis 2

We may modify hypothesis 1 with the assumption that the internal data bus is only 64 bits wide, so that a 128-bit operand is transferred to the execution units in two clock cycles. If we still assume that there are two 64-bit execution units running at half speed, then the first 64-bit unit can start at time T = 0 when the first half of the 128-bit operand arrives, while the second 64-bit unit will start one clock later, when the second half of the operand arrives (see figure 15.3). The first 64-bit unit will then be able to accept a new 64-bit operand at time T=2, before the second 64-bit unit is finished with the second half of the 128-bit operand. If we have a sequence of alternating 128-bit and 64-bit uops, then the third uop, which is 128-bit, can start with its first half operand at time T=3, using the second 64-bit execution unit, while the second operand starts at T=4 using the first 64-bit execution unit. As figure 15.3 shows, this can explain the observation that a sequence of *n* 128-bit uops alternating with *n* 64-bit uops takes 3· *n* clocks.

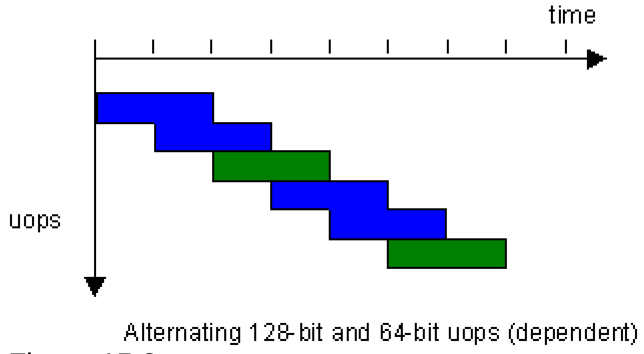

Figure 15.3

The measured latency of simple 128-bit uops is not 3 clocks, but 2. In order to explain this, we have to look at how a dependence chain of 128-bit uops is executed. Figure 15.4 shows the execution of a chain of interdependent 128-bit uops.

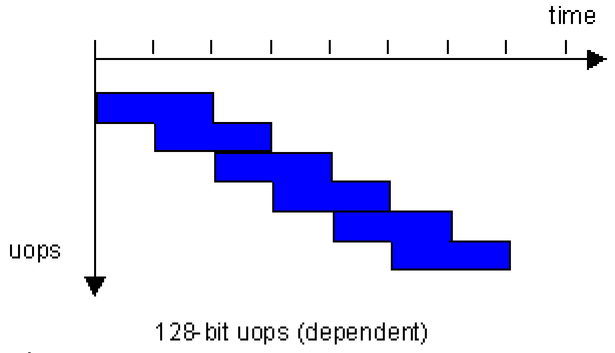

Figure 15.4

The first uop handles the first half of its operand from time T = 0 to 2, while the second half of the operand is handled from time T = 1 to time 3. The second uop can start to handle its first half operand already at time T = 2, even though the second half operand is not ready until time T = 3. A sequence of *n* interdependent 128-bit uops of this kind will thus take 2· *n*+1 clocks. The extra 1 clock in the end will appear to be part of the latency of the final instruction in the chain, which stores the result to memory. Thus, for practical purposes, we can calculate with a latency of 2 clocks for simple 128-bit uops.

Hypothesis 3

The assumption is now that there is only one 64-bit arithmetic unit running at full speed. It has a latency of 2 clocks and is fully pipelined, so that it can accept a new 64-bit operand every clock cycle. Under this assumption, the sequence of alternating 128-bit and 64-bit uops will still be executed as shown in figure 15.3.

There is no experimental way to distinguish between hypothesis 2 and 3 if the two units assumed under hypothesis 2 are identical, because all inputs and outputs to the execution units occur at the same time under both of these hypotheses. However, hypothesis 3 seems less likely than hypothesis 2 because we have no explanation of why the execution unit would require two pipeline stages.

It would be possible to prove hypothesis 2 and reject hypothesis 3 if there were some 64-bit operations that could execute only in one of the two assumed 64-bit units, but I have not found any such operations.

Hypothesis 4

In the P3, simple 128-bit instructions are split into two 64-bit uops. If this is also the case in the P4, then the uops in figure 15.2 can be executed out of order to allow overlap with the 64-bit instructions. However, this is not in accordance with the uop counts that can be measured with the performance monitor counters.

Those 128-bit uops where the two 64-bit halves are not independent of each other all have a latency of 4 clocks. This is in accordance with hypothesis 2 and 3.

We may thus conclude that hypothesis 2 is the most probable explanation. It is also a logical choice. Two 64-bit units running at half speed will give the same latency and throughput on 128-bit operands as a single 64-bit execution unit running at full speed with a latency of 1. If the former implementation is cheaper than the latter, reduces the power consumption, or allows a higher clock speed, then this will be a reasonable choice for a microprocessor designer. Hypothesis 2 is partly confirmed by the following sentence in Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual: "Intel NetBurst micro-architecture [...] uses a deeply pipelined design to enable high clock rates with different parts of the chip running at different clock rates, some faster and some slower than the nominally-quoted clock frequency of the processor". Letting different units run at different speeds may actually be a better design decision than letting the slowest unit determine the overall clock frequency. A further reason for this choice may be to reduce power consumption and optimize the thermal design.

Floating-point addition and multiplication uops operating on 80-bit registers have latencies that are one clock cycle more than the latencies of similar uops in 128-bit registers. The latencies of these instructions are thus odd values. Under hypothesis 2, the extra clock cycle can be explained as the extra time it takes to transfer an 80-bit operand over a 64-bit data bus. Under hypothesis 3, the extra clock cycle can be explained as the time needed to generate the extra 80-bit precision.

Scalar floating-point operations in 80-bit registers have a throughput of 1 uop per clock cycle, while scalar floating-point operations in 128-bit registers have half throughput, even though they only use 32 or 64 of the 128 bits. This is probably because the remaining 96 or 64 bits of the destination operand, which remain unchanged, are going through the execution unit to the new (renamed) destination register.

Divisions behave differently. There is a separate division unit which uses iteration and is not pipelined. Divisions can have both odd and even latencies, so it is likely that at least part of the division unit runs at full speed, even though it uses the FP-mul unit. Square roots also use the division unit.

15.5 Transfer of data between execution units

The latency of an operation is in most cases longer if the next dependent operation is not executed in the same execution unit. Example:

```
                    ; clock      ex.unit   subunit
PADDW XMM0,XMM1     ;  0 -  2    mmx       alu
PSLLW XMM0,4        ;  2 -  4    mmx       shift
PMULLW XMM0,XMM2    ;  5 - 11    fp        mul
PSUBW XMM0,XMM3     ; 12 - 14    mmx       alu
POR XMM6,XMM7       ;  3 -  5    mmx       alu
MOVDQA XMM1,XMM0    ; 15 - 21    mov
PAND XMM1,XMM4      ; 21 - 23    mmx       alu
```

The first instruction PADDW runs in the MMX unit under port 1, and has a latency of 2. The shift instruction PSLLW runs in the same execution unit, though in a different subunit. There is no extra delay, so it can start at time T=2. The multiplication instruction PMULLW runs in a different execution unit, the FP unit, because there is no multiplication subunit in the MMX execution unit. This gives an extra delay of one clock cycle. The multiplication cannot start until T=5, even though the shift operation finished at T=4. The next instruction, PSUBW, goes back to the MMX unit, so again we have a delay of one clock cycle from the multiplication is finished till the subtraction can begin. The POR does not depend on any of the preceding instructions, so it can start as soon as port 1 and the mmx-alu subunit are both vacant. The MOVDQA instruction goes to the mov unit under port 0, which gives us another delay of one clock cycle after the PSUBW has finished. The last instruction, PAND, goes back to the MMX unit under port 1. However, there is no additional delay after a move instruction. The whole sequence takes 23 clock cycles.

There is no delay between the two double-speed units, alu0 and alu1, but there is an additional delay of ½ clock cycle from these units to any other (single-speed) execution unit. Example:

```
                    ;   clock         ex.unit   subunit
AND EAX,0FH         ;  0.0 -  0.5     alu0      logic
XOR EBX,30H         ;  0.5 -  1.0     alu0      logic
ADD EAX,1           ;  0.5 -  1.0     alu1      add
SHL EAX,3           ;  2.0 -  6.0     int       mmx shift
SUB EAX,ECX         ;  7.0 -  7.5     alu0/1    add
MOV EDX,EAX         ;  7.5 -  8.0     alu0/1    mov
IMUL EDX,100        ;  9.0 - 23.0     int       fp mul
OR  EDX,EBX         ; 23.0 - 23.5     alu0/1    mov
```

The first instruction, AND, starts at time T=0 in alu0. Running at double speed, it is finished at time 0.5. The XOR instruction starts as soon as alu0 is vacant, at time 0.5. The third instruction, ADD, needs the result of the first instruction, but not the second. Since alu0 is occupied by the XOR, the ADD has to go to alu1. There is no delay from alu0 to alu1, so the ADD can start at time T=0.5, simultaneously with the XOR, and finish at T=1.0. The SHL instruction runs in the single-speed int unit. There is a ½ clock delay from alu0 or alu1 to any other unit, so the int unit cannot receive the result of the ADD until time T=1.5. Running at single speed, the int unit cannot start at a half-clock tick so it will wait until time T=2.0 and finish at T=6.0. The next instruction, SUB, goes back to alu0 or alu1. There is a one-clock delay from the SHL instruction to any other execution unit, so the SUB instruction is delayed until time T=7.0. After the two double-speed instructions, SUB and MOV, we have a ½ clock delay again before the IMUL running in the int unit. The IMUL, running again at single speed, cannot start at time T=8.5 so it is delayed until T=9.0. There is no additional delay after IMUL, so the last instruction can start at T=23.0 and end at T=23.5.

There are several ways to improve this code. The first improvement is to swap the order of ADD and SHL (then we have to add 1 SHL 3 = 8):

```
    AND EAX,00FH        ;  0.0 -  0.5    alu0      logic
    XOR EBX,0F0H        ;  0.5 -  1.0    alu0      logic
    SHL EAX,3           ;  1.0 -  5.0    int       mmx shift
    ADD EAX,8           ;  6.0 -  6.5    alu1      add
    SUB EAX,ECX         ;  6.5 -  7.0    alu0/1    add
    MOV EDX,EAX         ;  7.0 -  7.5    alu0/1    mov
    IMUL EDX,100        ;  8.0 - 22.0    int       fp mul
    OR  EDX,EBX         ; 22.0 - 22.5    alu0/1    mov
```

Here we are saving ½ clock before the `SHL` and ½ clock before the `IMUL` by making the data for these instructions ready at a half-clock tick so that they are available to the single-speed unit ½ clock later, at an integral time. The trick is to reorder your instructions so that you have an odd number of double-speed uops between any two single-speed or half-speed uops in a chain of interdependent instructions. You can improve the code further by minimizing the number of transitions between execution units. Even better, of course, is to keep all operations in the same execution unit, and preferably the double-speed units. `SHL EAX,3` can be replaced by 3 × (`ADD EAX,EAX`). See page 113 for how to replace integer multiplications by additions.

If we want to know why there is an additional delay when going from one execution unit to another, there are three possible explanations:

Explanation A

The physical distance between execution units on the silicon chip causes a propagation delay in the traveling of electrical signals from one unit to another because of the induction and capacity in the wires.

Explanation B

The "logical distance" between execution units means that the data have to travel through various buffers, ports, buses and multiplexers to get to the right destination. The designers have implemented various shortcuts to bypass these delaying elements when the output of an instruction is needed immediately afterwards in the same or a nearby execution unit.

If we assume that on a less sophisticated design, a simple operation like integer addition uses half a clock cycle for doing the calculation and the rest of the clock cycle for directing the result to the right address. Then, bypassing the latter process may be the trick that enables the P4 to do some calculations at double speed when the result of the operation is needed only in the same execution unit.

Explanation C

If 128-bit operands are handled 64 bits at a time, as figure 15.4 suggests, then we will have a 1 clock delay at the end of a chain of 128-bit instructions when the two halves have to be united. Consider, for example, the addition of packed double precision floating-point numbers in 128-bit registers. If the addition of the lower 64-bit operand starts at time T=0, it will finish at T=4. The upper 64-bit operand can start at time T=1 and finish at T=5. If the next dependent operation is also a packed addition, then the second addition can start to work on the lower 64-bit operand at time T=4, before the upper operand is ready.

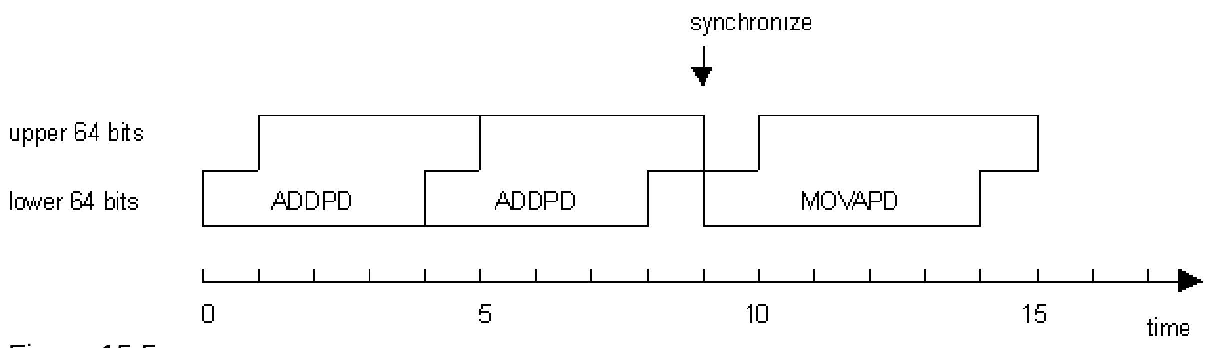

Figure 15.5

The latency for a chain of such instructions will appear to be 4 clock cycles per operation. If all operations on 128-bit registers can overlap in this way, then we will never see the 128-bit operations having higher latency than the corresponding 64-bit operations. But if the transport of the data to another execution unit requires that all 128 bits travel together, then we get an additional delay of 1 clock cycle for the synchronization of the upper and lower operands, as figure 15.5 shows. It is not known whether the data buses between execution units are 64 bits or 128 bits wide.

Obviously, explanation C cannot explain additional delays in 32-bit operations, so we have to accept explanation A or B, at least for the double-speed units. Explanation C may still apply in some situations, such as memory loads and stores, as well as register-to-register moves that use the same execution unit as memory stores.

15.6 Retirement

The retirement of executed uops works in the same way in the P4 as in the 6'th generation processors. This process is explained on page 69.

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 trace cache. But the retirement station has the further limitation that taken jumps must retire in the first of the three slots in the retirement station. This sometimes limits the throughput of small loops. If the number of uops in the loop is not a multiple of 3, then the jump-back instruction in the bottom of the loop may go into the wrong retirement slot, at the penalty of one clock cycle per iteration. It is therefore recommended that the number of uops (not instructions) in small critical loops should be a multiple of 3. In some cases, you can actually save one clock cycle per iteration by adding one or two NOP's to the loop to make the number of uops divisible by 3. This applies only if a throughput of 3 uops per clock cycle is expected.

15.7 Partial registers and partial flags

Registers AL, AH, and AX are all parts of the EAX register. These are called partial registers. On 6'th generation microprocessors, the partial registers could be split into separate temporary registers, so that different parts could be handled independently of each other. This caused a serious delay whenever there was a need to join different parts of a register into a single full register. This problem is explained on page 70.

To prevent this problem, the P4 always stores the whole register together. This solution has other drawbacks, however. The first drawback is that it introduces false dependences. Any read or write to AL will be delayed if a preceding write to AH is delayed.

Another drawback is that access to a partial register sometimes requires an extra uop. Examples:

```
    MOV EAX,[MEM32]        ; 1 uop
    MOV AX,[MEM16]         ; 2 uops
    MOV AL,[MEM8]          ; 2 uops
    MOV AH,[MEM8]          ; 2 uops
    MOVZX EAX,[MEM8]       ; 1 uop
    MOVSX EAX,[MEM8]       ; 2 uops
    ADD AL,BL              ; 1 uop
    ADD AH,BH              ; 1 uop
    ADD AL,BH              ; 2 uops
    ADD AH,BL              ; 2 uops
```

For optimal performance, you may follow the following guidelines when working with 8-bit and 16-bit operands:

- Avoid using the high 8-bit registers `AH`, `BH`, `CH`, `DH`.
- When reading from an 8-bit or 16-bit memory operand, use `MOVZX` to read the entire 32-bit register, even in 16-bit mode.
- Alternatively, use MMX or XMM registers to handle 8-bit and 16-bit integers, if they can be packed.

The problems with partial access also apply to the flags register when an instruction modifies some of the flags but leaves other flags unchanged.

For historical reasons, the `INC` and `DEC` instructions leave the carry flag unchanged, while the other arithmetic flags are written to. This causes a false dependence on the previous value of the flags and costs an extra uop. To avoid these problems, it is recommended that you always use `ADD` and `SUB` instead of `INC` and `DEC`. For example, `INC EAX` should be replaced by `ADD EAX,1`.

`SAHF` leaves the overflow flag unchanged but changes the other arithmetic flags. This causes a false dependence on the previous value of the flags, but no extra uop.

`BSF` and `BSR` change the zero flag but leave the other flags unchanged. This causes a false dependence on the previous value of the flags and costs an extra uop.

`BT`, `BTC`, `BTR`, and `BTS` change the carry flag but leave the other flags unchanged. This causes a false dependence on the previous value of the flags and costs an extra uop. Use `TEST`, `AND`, `OR` or `XOR` instead of these instructions.

15.8 Partial memory access

The problems with accessing part of a memory operand are much bigger than when accessing part of a register. These problems are the same as for previous processors, see page 73. Example:

```
    MOV DWORD PTR  [MEM1], EAX
    MOV DWORD PTR  [MEM1+4], 0
    FILD QWORD PTR [MEM1]              ; large penalty
```

You can save 10-20 clocks by changing this to:

```
    MOVD XMM0, EAX
    MOVQ QWORD PTR [MEM1], XMM0
    FILD QWORD PTR [MEM1]              ; no penalty
```

15.9 Memory intermediates in dependencies

The P4 has an unfortunate proclivity for trying to read a memory operand before it is ready. If you write

```
    IMUL EAX,5
    MOV [MEM1],EAX
    MOV EBX,[MEM1]
```

Then the microprocessor may try to read the value of `[MEM1]` into `EBX` before the `IMUL` and the memory write have finished. It soon discovers that the value it has read is invalid, so it will discard `EBX` and try again. It will keep replaying the read instruction until the data in `[MEM1]` are ready. There seems to be no limit to how many times it can replay a memory read, and this process steals resources from other processes. In a long dependence chain, this may typically cost 10 - 20 clock cycles! Using the `MFENCE` instruction to serialize memory access does not solve the problem because this instruction is even more costly. On previous microprocessors, the penalty for reading a memory operand immediately after writing to the same memory position is only a few clock cycles.

The best way to avoid this problem is, of course, to replace `MOV EBX,[MEM1]` with `MOV EBX,EAX` in the above example. Another possible solution is to give the processor plenty of work to do between the store and the load from the same address.

However, there are two situations where it is not possible to keep data in registers. The first situation is the transfer of parameters in high-level language procedure calls; the second situation is transferring data between floating-point registers and other registers.

Transferring parameters to procedures

Calling a function with one integer parameter in C++ will typically look like this:

```
    PUSH EAX                ; save parameter on stack
    CALL _FF                ; call function _FF
    ADD ESP,4               ; clean up stack after call
    ...
    _FF PROC NEAR           ; function entry
    PUSH EBP                ; save EBP
    MOV EBP,ESP             ; copy stack pointer
    MOV EAX,[EBP+8]         ; read parameter from stack
    ...
    POP EBP                 ; restore EBP
    RET                     ; return from function
    _FF ENDP
```

As long as either the calling program or the called function is written in high-level language, you may have to stick to the convention of transferring parameters on the stack. Most C++ compilers can transfer 2 or 3 integer parameters in registers when the function is declared `__fastcall`. However, this method is not standardized. Different compilers use different registers for parameter transfer, so you may need one version of your procedure for each compiler. To avoid the problem, you may have to keep the entire dependence chain in assembly language. See also page 10 for a discussion of how to handle this problem in C++.

Transferring data between floating-point and other registers

There is no way to transfer data between floating-point registers and other registers, except through memory. Example:

```
    IMUL EAX,EBX
    MOV [TEMP],EAX       ; transfer data from integer register to f.p.
    FILD [TEMP]
    FSQRT
    FISTP [TEMP]         ; transfer data from f.p. register to integer
    MOV EAX,[TEMP]
```

Here we have the problem of transferring data through memory twice. You may avoid the problem by keeping the entire dependence chain in floating-point registers, or by using XMM registers instead of floating-point registers.

Another way to prevent premature reading of the memory operand is to make the read address depend on the data. The first transfer can be done like this:

```
MOV [TEMP],EAX
AND EAX,0          ; make EAX = 0, but keep dependence
FILD [TEMP+EAX]    ; make read address depend on EAX
```

The `AND EAX,0` instruction sets `EAX` to zero but keeps a false dependence on the previous value. By putting `EAX` into the address of the `FILD` instruction, we prevent it from trying to read before `EAX` is ready.

It is a little more complicated to make a similar dependence when transferring data from floating-point registers to integer registers. The simplest way to solve the problem is:

```
FISTP [TEMP]
FNSTSW AX             ; transfer status after FISTP to AX
AND EAX,0             ; set to 0
MOV EAX,[TEMP+EAX]    ; make dependent on EAX
```

Two other methods are a little faster:

```
FIST [TEMP]           ; store without popping
FCOMIP ST,ST          ; compare and pop, make flags depend on ST
SETC AL               ; make AL depend on flags
AND EAX,0             ; set to 0
MOV EAX,[TEMP+EAX]    ; make dependent on EAX
```

and:

```
FNSTSW AX             ; transfer status to AX before FISTP
FISTP [TEMP]
REPT 5                ; repeat 5 times
   NEG EAX            ; exactly match latency of FISTP
ENDM
AND EAX,0             ; set to 0
MOV EAX,[TEMP+EAX]    ; make dependent on EAX
```

15.10 Breaking dependencies

A common way of setting a register to zero is `XOR EAX,EAX` or `SUB EBX,EBX`. The P4 processor recognizes that these instructions are independent of the prior value of the register. So any instruction that uses the new value of the register will not have to wait for the value prior to the `XOR` or `SUB` instruction to be ready. The same applies to the `PXOR` instruction with a 64-bit or 128-bit register, but not to any of the following instructions: `XOR` or `SUB` with an 8-bit or 16-bit register, `SBB`, `PANDN`, `PSUB`, `XORPS`, `XORPD`, `SUBPS`, `SUBPD`, `FSUB`.

The instructions `XOR`, `SUB` and `PXOR` are useful for breaking an unnecessary dependence. On PPro, P2 and P3, you have to write `MOV EAX,0` to break the dependence.

You may also use these instructions for breaking dependencies on the flags. For example, rotate instructions have a false dependence on the flags. This can be removed in the following way:

```
ROR EAX,1
SUB EDX,EDX    ; remove false dependence on the flags
ROR EBX,1
```

If you don't have a spare register for this purpose, then use an instruction which doesn't change the register, but only the flags, such as `CMP` or `TEST`. The stack pointer may be preferred for this purpose because it is the least likely register to be delayed by prior dependencies. So you may replace `SUB EDX,EDX` in the above example with `CMP ESP,ESP`. You cannot use `CLC` for breaking dependencies on the carry flag.

15.11 Choosing the optimal instructions

There are many possibilities for replacing less efficient instructions with more efficient ones. The most important cases are summarized below.

INC and DEC

These instructions have a problem with partial flag access, as explained on page 88. Always replace `INC EAX` with `ADD EAX,1`, etc.

8-bit and 16-bit integers

Replace `MOV AL,BYTE PTR [MEM8]` by `MOVZX EAX,BYTE PTR [MEM8]`
Replace `MOV BX,WORD PTR [MEM16]` by `MOVZX EBX,WORD PTR [MEM16]`

Avoid using the high 8-bit registers `AH`, `BH`, `CH`, `DH`.

If 8-bit or 16-bit integers can be packed and handled in parallel, then use MMX or XMM registers.

These rules apply even in 16-bit mode.

Memory stores

Most memory store instructions use 2 uops. Simple store instructions of the type `MOV [MEM],EAX` use only one uop if the memory operand has no SIB byte. A SIB byte is needed if there is more than one pointer register, if there is a scaled index register, or if `ESP` is used as base pointer. The short-form store instructions can use a 32-bit register, a 16-bit register, or a low 8-bit register (see page 76). Examples:

```
MOV ARRAY[ECX], EAX      ; 1 uop
MOV ARRAY[ECX*4], EAX    ; 2 uops because of scaled index
MOV [ECX+EDI], EAX       ; 2 uops because of two index registers
MOV [EBP+8], EBX         ; 1 uop
MOV [ESP+8], EBX         ; 2 uops because ESP used
MOV ES:[MEM8], CL        ; 1 uop
MOV ES:[MEM8], CH        ; 2 uops because high 8-bit register used
MOVQ [ESI], MM1          ; 2 uops because not a general purp.register
FSTP [MEM32]             ; 2 uops because not a general purp.register
```

The corresponding memory load instructions all use only 1 uop. A consequence of these rules is that a procedure which has many stores to local variables on the stack should use `EBP` as pointer, while a procedure which has many loads and few stores may use `ESP` as pointer, and save `EBP` for other purposes.

Shifts and rotates

Shifts and rotates on integer registers are quite slow on the P4 because the integer execution unit transfers the data to the MMX shift unit and back again. Shifts to the left may be replaced by additions. For example, `SHL EAX,3` can be replaced by 3 times `ADD EAX,EAX`.

Rotates through carry (`RCL`, `RCR`) by a value different from 1 or by `CL` should be avoided.

If your code contains many integer shifts and multiplications, then it may be advantageous to execute it in MMX or XMM registers.

Integer multiplication

Integer multiplication is slow on the P4 because the integer execution unit transfers the data to the FP-MUL unit and back again. If your code has many integer multiplications then it may be advantageous to handle the data in MMX or XMM registers.

Integer multiplication by a constant can be replaced by additions. See page 113 for a description of this method. Replacing a single multiply instruction by a long sequence of `ADD` instructions should, of course, only be done in critical dependence chains.

LEA

The `LEA` instruction is split into additions and shifts on the P4. `LEA` instructions with a scale factor may preferably be replaced by additions. This applies only to the `LEA` instruction, not to any other instructions with a memory operand containing a scale factor.

Register-to-register moves bigger than 32 bits

The following instructions, which copy one register into another, all have a latency of 6 clocks: `MOVQ R64,R64`, `MOVDQA R128,R128`, `MOVAPS R128,R128`, `MOVAPD R128,R128`, `FLD R80`, `FST R80`, `FSTP R80`. These instructions have no additional latency. A possible reason for the long latency of these instructions is that they use the same execution unit as memory stores (port 0, mov).

There are several ways to avoid this delay:

- The need for copying a register can sometimes be eliminated by using the same register repeatedly as source, rather than destination, for other instructions.

- With floating-point registers, the need for moving data from one register to another can often be eliminated by using `FXCH`. The `FXCH` instruction has no latency.

- If the value of a register needs to be copied, then use the old copy in the most critical dependency path, and the new copy in a less critical path. The following example calculates Y = $(a+b)^{2.5}$:
  ```
  FLD [A]
  FADD [B]     ; a+b
  FLD ST       ; copy a+b
  FXCH         ; get old copy
  FSQRT        ; (a+b)^0.5
  FXCH         ; get new (delayed) copy
  FMUL ST,ST   ; (a+b)^2
  FMUL         ; (a+b)^2.5
  FSTP [Y]
  ```
  The old copy is used for the slow square root, while the new copy, which is available 6 clocks later, is used for the multiplication.

If none of these methods solve the problem, and latency is more important than throughput, then use faster alternatives:

- For 80-bit floating-point registers:
  ```
  FLD ST                 ; copy register
  ```
  can be replaced by
  ```
  FLDZ                   ; make an empty register
  SUB EAX,EAX            ; set zero flag
  FCMOVZ ST,ST(1)        ; conditional move
  ```

- For 64-bit MMX registers:
  ```
  MOVQ MM1,MM0
  ```
  can be replaced by the shuffle instruction
  ```
  PSHUFW MM1,MM0,11100100B
  ```

- For 128-bit XMM registers containing integers:
  ```
  MOVDQA XMM1,XMM0
  ```
  can be replaced by the shuffle instruction
  ```
  PSHUFD XMM1,XMM0,11100100B
  ```
  or even faster:
  ```
  PXOR XMM1,XMM1    ; set new register to 0
  POR XMM1,XMM0     ; OR with desired value
  ```

- For 128-bit XMM registers containing packed single precision floats:
  ```
  MOVAPS XMM1,XMM0
  ```
  can be replaced by
  ```
  PXOR XMM1,XMM1    ; set new register to 0
  ORPS XMM1,XMM0    ; OR with desired value
  ```
  Here, I have used `PXOR` rather than the more correct `XORPS` because the former breaks any dependence on previous values, the latter does not.

- For 128-bit XMM registers containing packed double precision floats:
  ```
  MOVAPD XMM1,XMM0
  ```
  can be replaced by
  ```
  PXOR XMM1,XMM1    ; set new register to 0
  ORPD XMM1,XMM0    ; OR with desired value
  ```
  Again, I have used `PXOR` rather than the more correct `XORPD` because the former breaks any dependence on previous values, the latter does not.

These methods all have lower latencies than the register-to-register moves. However, a drawback of these tricks is that they use port 1 which is also used for all calculations on these registers. If port 1 is saturated, then it may be better to use the slow moves, which go to port 0.

15.12 Bottlenecks in P4

It is important, when optimizing a piece of code, to find the limiting factor that controls execution speed. Tuning the wrong factor is unlikely to have any beneficial effect. In the following paragraphs, I will explain each of the possible limiting factors. You have to consider each factor in order to determine which one is the narrowest bottleneck, and then concentrate your optimization effort on that factor until it is no longer the narrowest bottleneck. As explained before, you have to concentrate on only the most critical part of your program - usually the innermost loop.

Memory access

If you are accessing large amounts of data, or if your data are scattered around everywhere in the memory, then you will have many data cache misses. Accessing uncached data is so time consuming that all other optimization considerations are unimportant. The caches are organized as aligned lines of 64 bytes each. If one byte within an aligned 64-bytes block has been accessed, then you can be certain that all 64 bytes will be loaded into the level-1 data cache and can be accessed at no extra cost. To improve caching, it is recommended that data that are used in the same part of the program be stored together. You may align large arrays and structures by 64. Store local variables on the stack if you don't have enough registers.

The level-1 data cache is only 8 kb on the P4. This may not be enough to hold all your data, but the level-2 cache is more efficient on the P4 than on previous processors. Fetching data from the level-2 cache will cost you only a few clock cycles extra.

Data that are unlikely to be cached may be prefetched before they are used. If memory addresses are accessed consecutively, then they will be prefetched automatically. You should therefore preferably organize your data in a linear fashion so that they can be accessed consecutively, and access no more than four large arrays, preferably less, in the critical part of your program.

The `PREFETCH` instructions can improve performance in situations where you access uncached data and cannot rely on automatic prefetching. However, excessive use of the `PREFETCH` instructions can slow down program throughput. If you are in doubt whether a `PREFETCH` instruction will benefit your program, then you may simply load the data needed into a spare register rather than using a `PREFETCH` instruction. If you have no spare register then use an instruction which reads the memory operand without changing any register, such as `CMP` or `TEST`. As the stack pointer is unlikely to be part of a critical dependence chain, a useful way to prefetch data is `CMP ESP,[MEM]`, which will change only the flags.

When writing to a memory location that is unlikely to be accessed again soon, you may use the non-temporal write instructions `MOVNTI`, etc., but excessive use of non-temporal moves will slow down performance.

Not only data, but also code, should be arranged for optimal caching. Subroutines that are used in the same part of the program should preferably be stored together in the same memory address range, rather than be scattered around at different addresses. Put seldom used branches away in the bottom of your code. Make sure the critical part of your program is not too big for the trace cache.

Further guidelines regarding memory access can be found in "Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual".

Execution latency

The executing time for a dependence chain can be calculated from the latencies listed in the tables starting on page145. Most instructions have an additional latency of ½ or 1 clock cycle when the subsequent instruction goes to a different execution unit. See page 85 for further explanation.

The longest dependence chains occur in loops where each iteration depends on the result of the preceding one. Such loops can often be improved by handling data in parallel. Use multiple accumulators or SIMD instructions to handle data in parallel (see page 33).

If long dependence chains limit the performance of your program then you may improve performance by choosing instructions with low latency, minimizing the number of transitions between execution units, breaking up dependence chains, and utilizing all opportunities for calculating subexpressions in parallel.

Always avoid memory intermediates in dependence chains, as explained on page 88.

Execution unit throughput

If your dependence chains are short, or if you are working on several dependence chains in parallel, then your program may be limited by throughput rather than latency. Different execution units have different throughputs. Alu0 and alu1, which handle simple integer instructions and other common uops, both have a throughput of 2 instructions per clock cycle. Most other execution units have a throughput of one instruction per clock cycle. When working with 128-bit registers, the throughput is usually one instruction per two clock cycles. Division and square roots have the lowest throughputs. The instruction list on page 145 indicates reciprocal throughputs for all instructions on the P4 processor. Each throughput measure applies to all uops executing in the same execution subunit (see page 81).

If execution throughput limits your code then try to move some calculations to other execution subunits.

Port throughput

Each of the execution ports can receive one uop per clock cycle. Port 0 and port 1 can receive an additional uop at each half-clock tick if these uops go to the double-speed units alu0 and alu1. If all uops in the critical part of your code go to the single-speed and half-speed units under port 1, then the throughput will be limited to 1 uop per clock cycle. If the uops are optimally distributed between the four ports, then the throughput may be as high as 6 uops per clock cycle. Such a high throughput can only be achieved in short bursts, because the trace cache and the retirement station limit the average throughput to 3 uops per clock cycle.

If port throughput limits your code then try to move some uops to other ports.

Trace cache delivery

The trace cache can deliver a maximum of 3 uops per clock cycle. Some uops require more than one trace cache entry, as explained on page 75. The delivery rate can be less than 3 uops per clock cycle for code that contains many branches and for tiny loops with branches inside (see page 78).

If none of the abovementioned factors limit program performance, then you may aim at a throughput of 3 uops per clock cycle.

Choose the instructions that generate the smallest number of uops. Avoid uops that require more than one trace cache entry, and avoid having an odd number of single-size uops between any two double-size uops (see page 75).

Uop retirement

The retirement station can handle 3 uops per clock cycle. Taken branches can only be handled by the first of the three slots in the retirement station.

If you aim at an average throughput of 3 uops per clock cycle then avoid an excessive number of jumps, calls and branches. Small critical loops should preferably have a number of uops divisible by 3 (see page 87).

Instruction decoding

If the critical part of your code doesn't fit into the trace cache, then the limiting stage may be instruction decoding. The decoder can handle one instruction per clock cycle, provided that the instruction generates no more than 4 uops and no microcode, and has no more than one prefix (see page 79). If decoding is a bottleneck, then you may try to minimize the number of instructions rather than the number of uops.

Branch prediction

The calculations of latencies and throughputs are only valid if all branches are predicted. Branch mispredictions can seriously slow down performance when latency or throughput is the limiting factor.

Avoid poorly predictable branches in critical parts of your code unless the alternative (e.g. conditional moves) outweighs the advantage by adding costly extra dependencies and latency. See page 44 for details.

16 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 likely to be in the cache most of the time. 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 page 28).

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 that 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) {
  for (int i=0; i<N; i++) B[i] = -A[i];}
```

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

```
; Example 16.1:
_ChangeSign PROC NEAR
        PUSH    ESI
        PUSH    EDI
A       EQU     DWORD PTR [ESP+12]  ; addresses of parameters on stack
B       EQU     DWORD PTR [ESP+16]
N       EQU     DWORD PTR [ESP+20]
        MOV     ECX, [N]
        JECXZ   L2                  ; skip if N = 0
        MOV     ESI, [A]            ; pointer to source A
        MOV     EDI, [B]            ; pointer to destination B
        CLD
L1:     LODSD                       ; read
        NEG     EAX                 ; change sign
        STOSD                       ; write
        LOOP    L1                  ; repeat
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 the complex instructions `JECXZ`, `CLD`, `LODSD`, `STOSD` and `LOOP`, which are inefficient on most processors. It can be improved by avoiding these instructions:

```
; Example 16.2:
        MOV     ECX, [N]        ; ECX = counter
        MOV     ESI, [A]
        TEST    ECX, ECX
        JZ      SHORT L2        ; skip if N = 0
        MOV     EDI, [B]
L1:     MOV     EAX, [ESI]      ; read
        ADD     ESI, 4          ; increment source pointer
        NEG     EAX             ; change sign
        MOV     [EDI], EAX      ; write
        ADD     EDI, 4          ; increment destination pointer
        SUB     ECX, 1          ; decrement loop counter
        JNZ     L1              ; loop
L2:
```

Here I am using SUB ECX,1 instead of DEC ECX because the latter instruction uses one extra uop in the P4. It is an advantage to have the loop control branch in the bottom of the loop because if the branch were at the top of the loop then we would need an extra jump in the bottom of the loop.

Using the same register for counter and index reduces the number of instructions:

```
; Example 16.3:
        MOV     ESI, [A]
        MOV     EDI, [B]
        MOV     ECX, [N]
        SUB     EDX, EDX            ; set counter EDX = 0
        TEST    ECX, ECX
        JZ      SHORT L2
L1:     MOV     EAX, [ESI+4*EDX]    ; use base pointer and index
        NEG     EAX
        MOV     [EDI+4*EDX], EAX
        ADD     EDX, 1              ; increment loop counter
        CMP     EDX, ECX
        JB      L1
L2:
```

We can get rid of the CMP instruction in example 16.3 by letting the loop counter end at zero and use the zero flag for detecting when the loop is finished as we did in example 16.2. One way of doing 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. This is possible if you let the base registers point to the end of the arrays rather than the beginning:

```
; Example 16.4:
        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                ; skip if N = 0
L1:     MOV     EAX, [ESI+4*ECX]
        NEG     EAX
        MOV     [EDI+4*ECX], EAX
        ADD     ECX, 1
        JNZ     L1
L2:
```

Now we are down to two simple instructions for loop overhead, which is as low as you can get.

On processors with out-of-order execution, it is likely that the second calculation will start before the first calculation is finished. In some situations, the processor may start several iterations until the maximum throughput of the execution units is reached.

The out-of-order capabilities cannot be utilized, however, in the case where each calculation depends on the result of the previous one. Such a continued dependence chain is the worst situation you can have, and you should definitely try to find a way to break down the dependence chain. Assume, for example, that we have to multiply a long series of integers. The C++ code looks like this:

```
int MultiplyList (int * List, int N) {
  int product = 1, i;
  for (i=0; i<N; i++) product *= List[i];
```

```
      return product;}
```

The best thing you can do here is to roll out the loop and use two accumulators:

```
    ; Example 16.5:
    _MultiplyList PROC NEAR
            PUSH    ESI
            PUSH    EBX
    List    EQU     DWORD PTR [ESP+12]  ; addresses of parameters on stack
    N       EQU     DWORD PTR [ESP+16]
            MOV     ESI, [List]
            MOV     ECX, [N]
            MOV     EAX, 1               ; accumulator one
            MOV     EBX, EAX             ; accumulator two
            SUB     EDX, EDX             ; counter starts at 0
            SUB     ECX, 1               ; N-1
            JS      SHORT L3             ; N-1 < 0 means N = 0
            SHL     ECX, 2               ; 4*(N-1)
    L1:     IMUL    EAX, [ESI+EDX]       ; multiply first accumulator
            IMUL    EBX, [ESI+EDX+4]     ; multiply second accumulator
            ADD     EDX, 8               ; add 2*(data size) to counter
            CMP     EDX, ECX             ; do we have at least 2 more?
            JB      L1
            JA      SHORT L2             ; finished if counter > N-1
            IMUL    EAX, [ESI+EDX]       ; N is odd, do one more
    L2:     IMUL    EAX, EBX             ; combine the two accumulators
    L3:     POP     EBX
            POP     ESI
            RET
    _MultiplyList ENDP
```

Now we are doing two operations in parallel and the long dependence chain is split into two parallel dependence chains of half the length. This will reduce the calculation time with almost 50% if the multiplication unit is pipelined.

When we roll out the loop by two, we have to check if the number of factors in `List` is odd. If `N` is odd, then we have to do the odd multiplication outside the loop. We can do the odd one either before or after the main loop. It may be more efficient to do it after the loop if `List` is aligned by 8.

In general, if you roll out a loop by R, i.e. if you do R calculations per loop iteration, then the number of extra calculations to do outside the loop is $E = N \bmod R$. If you want to do the extra E calculations before the main loop, then you have to calculate E, which requires a division if R is not a power of 2. This can be avoided by doing the extra calculations after the main loop, as shown in example 16.5. Of course it is an advantage to choose R so that N is divisible by R, if possible.

A suitable roll-out factor R can be found by dividing the latency of the most critical calculation instruction with the reciprocal throughput of the same instruction. Remember that it not necessary to roll out the loop if there is no continued dependence chain. Excessive loop unrolling will only fill up the code cache or trace cache without any significant speed advantage. However, you may choose to roll out a loop if this improves the prediction of the loop control branch.

In example 16.5, we are using three instructions for counter and loop control. It is possible to reduce this to two instructions, as in example 16.4, but this will make the code quite awkward. In most cases, the loop control instructions can execute in parallel with the calculations so you don't have to care about minimizing the loop control overhead.

16.1 Loops in P1 and PMMX

The P1 and PMMX processors have no capabilities for out-of-order execution. Instead you have to care about pairing opportunities. The code in example 16.4 may be changed to:

```
; Example 16.6:
        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                  ; start first calculation
        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                  ; end last calculation
        MOV     [EDI+4*ECX-4], EBX
L3:
```

Here the iterations are overlapped in order to improve pairing opportunities. We begin reading the second value before we have stored the first one. The `MOV EBX,0` instruction has been put in between `INC ECX` and `JNZ L1`, not to improve pairing, but to avoid the AGI stall that would result from using `ECX` as address index in the first instruction pair after it has been incremented.

Loops with floating-point operations are somewhat different because 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 16.7:
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]                  ; start first calc.
        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 the previous examples: using the loop counter as index register and counting through negative values up to zero. Each operation begins before the previous one is finished; in order to improve calculation overlaps.

The interleaving of floating-point operations works 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 counter in the first clock cycle after the index has been incremented.

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

16.2 Loops in PPro, P2, and P3

There are six important things that you have to analyze when optimizing a loop for the 6'th generation Intel processors: Instruction fetch, instruction decoding, register reads, execution ports, retirement, and branch mispredictions (see page 74). After a slight modification, example 16.4 now looks like this:

```
; Example 16.8:
        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                  ; skip if N = 0
ALIGN   16
L1:     MOV     EAX, [ESI+4*ECX]          ; len=3, p2rESIrECXwEAX
        NEG     EAX                       ; len=2, p01rwEAXwFlags
        MOV     [EDI+4*ECX], EAX          ; len=3, p4 rEAX, p3rEDIrECX
        INC     ECX                       ; len=1, p01rwECXwFlags
        JNZ     L1                        ; len=2, p1rFlags
L2:
```

The comments are interpreted as follows: The MOV EAX,[ESI+4*ECX] instruction is 3 bytes long, it generates one uop for port 2 that reads ESI and ECX, and writes to (renames) EAX. This information is needed for analyzing the possible bottlenecks.

First, we have to analyze the instruction decoding (page 59): One of the instructions generates 2 uops (MOV [EDI+4*ECX],EAX). This instruction must go into decoder D0. There are two decode groups in the loop so it can decode in 2 clock cycles.

Next, we have to analyze the instruction fetch (page 61): A loop always takes at least one clock cycle more than 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. If ESI and EDI are replaced by absolute addresses, then the loop will take 3 clocks because it cannot be contained in a single 16-byte block.

The third thing we have to analyze is register read stalls (page 65): 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.

The fourth analysis concerns the distribution of uops to the execution ports:
port 0 or 1: 2 uops
port 1: 1 uop
port 2: 1 uop
port 3: 1 uop
port 4: 1 uop
If both port 0 and 1 are fully used then the execution of two iterations will take 3 clock cycles. This gives an average execution time of 1.5 clocks per iteration.

The next analysis concerns retirement. The retirement station can handle 3 uops per clock cycle. The taken branch uop must go to the first slot of the retirement station. This means that the number of uops in the loop should preferably be divisible by 3. There are 6 uops and the retirement will take 2 clock cycles per iteration.

The `JNZ L1` branch in the end of the loop will be predicted correctly if `N` is no more than 5, and always the same value. Higher values can be made predictable by making nested loops. For example, if `N` is always equal to 20, then make one loop that repeats 5 times inside another loop that repeats 4 times. Further nesting is not worth the effort.

16.3 Loops in P4

To optimize a loop for the P4 processor, you have to analyze each of the possible bottlenecks mentioned on page 93 to determine which one is the limiting factor.

Let's first look at example 16.3 on page 97, and see how it performs on each of the possible bottlenecks:

1. memory access: The loop accesses two arrays. If these arrays are unlikely to be cached then forget about the rest, the performance is limited by memory access.

2. trace cache delivery: The loop contains 7 uops; and none of these require more than one entry in the trace cache. A delivery rate of 3 uops per clock from the trace cache means that the loop requires 2.33 clock cycle at the trace cache delivery stage. In general, there is no need to unroll the loop to avoid jumps in the trace cache, because this will be done automatically. The trace cache is likely to store two or more copies of small loop bodies consecutively to reduce the number of trace cache jumps.

3. retirement: The retirement station can handle 3 uops per clock cycle, but the taken branch at the bottom of the loop must go to the first slot in the retirement station. The retirement is therefore likely to take 3 clock cycles per iteration. The performance would be better if the number of uops were divisible by 3.

4. execution latency: There are no long or continued dependence chains in this loop. Each operation can be overlapped with the preceding ones, thanks to out-of-order execution. The only continued dependence chain is `ADD EDX,1` which requires ½ clock per iteration. This loop is therefore not limited by latency.

5. execution unit throughput: The 7 uops are well distributed between different execution units and subunits. The `NEG` and `JNZ` uops go to alu0. The `ADD` and `CMP` instructions can go to either alu0 or alu1. The memory load and store uops all go to different units. The memory store has the lowest throughput and requires 2 clock cycles per iteration.

6. port throughput: The 7 uops are well distributed between the execution ports: Port 0: 3 uops, Port 1: 2 uops, Port 2: 1 uop, Port 3: 1 uop. Most of the uops to port 0 and 1 can be accepted at half-clock ticks, so the time required for port throughput is 1.5 clock per iteration.

7. branch prediction: The loop control is likely to be predicted if the iteration count is no more than 17 and always the same, provided that there is a not-taken branch no more than 16 steps back in the prehistory before the last execution of the loop control branch (see page 44). If you are not certain that there is a not-taken branch before the loop, then insert a dummy branch instruction that is never taken, as explained on page 45. If the repetition count is between 17 and 32, and always the same, then unroll the loop by 2.

On previous processors, you can improve loop prediction by making nested loops, as explained on page 101. This method does not work on the P4 because the branch prediction in this processor relies on global branch history rather than local branch history (page 44). The only loop nesting scheme that improves prediction on the P4 is when all but the innermost loop have a repeat count of 2 and a `64H` branch prefix, as explained on page 46, and the innermost loop has a repeat count not exceeding 17 and a dummy never-taken branch before the loop entry. This solution becomes too clumsy if the repeat count is high. If the repeat count is high, then you can accept a single misprediction in the end.

8. conclusion

The conclusion of this analysis is that retirement is the limiting factor for the loop in example 16.3 if the loop branch is predicted or the repeat count is high. The execution will take approximately 3 clocks per iteration. The execution time can be reduced to 2 clocks by replacing the code with example 16.4 on page 97, which has 6 uops in the loop. Retirement is optimized when the number of uops is divisible by 3.

An alternative way of reducing the uop count of example 16.3 is to replace the `MOV [EDI+4*EDX],EAX`, which uses 2 uops, with a version without scaled index, which uses only 1 uop:

```
; Example 16.9:
      MOV     ESI, [A]
      MOV     EDI, [B]
      MOV     ECX, [N]
      TEST    ECX, ECX
      JZ      SHORT L2
      LEA     ECX, [EDI+4*ECX]      ; point to end of destination
      SUB     ESI, EDI              ; difference between the two arrays
L1:   MOV     EAX, [EDI+ESI]        ; compute source from destination
      NEG     EAX
      MOV     [EDI], EAX            ; destination
      ADD     EDI, 4                ; increment destination pointer
      CMP     EDI, ECX              ; compare with end address
      JB      L1
L2:
```

Example 16.9 has 6 uops in the loop body, which gives the same performance as example 16.4. Remember to use the simple addressing mode for the destination, not the source.

When counting uops, you should remember that `ADD` and `SUB` use 1 uop, while `INC` and `DEC` use 2 uops on the P4.

Analyzing dependences

The next example is a Taylor expansion. As you probably know, many functions can be approximated by a polynomial of the form

$$f(x) \approx \sum_{i=0}^{n} c_i x^i$$

Each power x^i is conveniently calculated by multiplying the preceding power x^{i-1} with x. The coefficients c_i are stored in a table:

```
;   Example 16.10
```

```
DATA SEGMENT PARA PUBLIC 'DATA'
x            dq   ?                  ; x
one          dq   1.0                ; 1.0
coeff        dq   c0, c1, c2, ...    ; Taylor coefficients
coeff_end    label qword             ; end of coeff. list
DATA ENDS
CODE SEGMENT BYTE PUBLIC 'CODE'
    MOVSD  XMM2, [X]                 ; XMM2 = x
    MOVSD  XMM1, [ONE]               ; XMM1 = x^i
    PXOR   XMM0, XMM0                ; XMM0 = sum. Init. to 0
    MOV    EAX,  OFFSET DS:coeff     ; point to c[i]
A:  MOVSD  XMM3, [EAX]               ; c[i]
    MULSD  XMM3, XMM1                ; c[i] * x^i
    MULSD  XMM1, XMM2                ; x^(i+1)
    ADDSD  XMM0, XMM3                ; sum += c[i] * x^i
    ADD    EAX,  8                   ; point to c[i+1]
    CMP    EAX, OFFSET DS:coeff_end ; stop at end of list
    JB     A
```

(If your assembler confuses the `MOVSD` instruction with the string instruction of the same name, then code it as `DB 0F2H / MOVUPS`).

And now to the analysis. The list of coefficients is so short that we can expect it to stay cached. Trace cache and retirement are obviously not limiting factors in this example.

In order to check whether latencies are important, we have to look at the dependences in this code. The dependences are shown in figure 16.1.

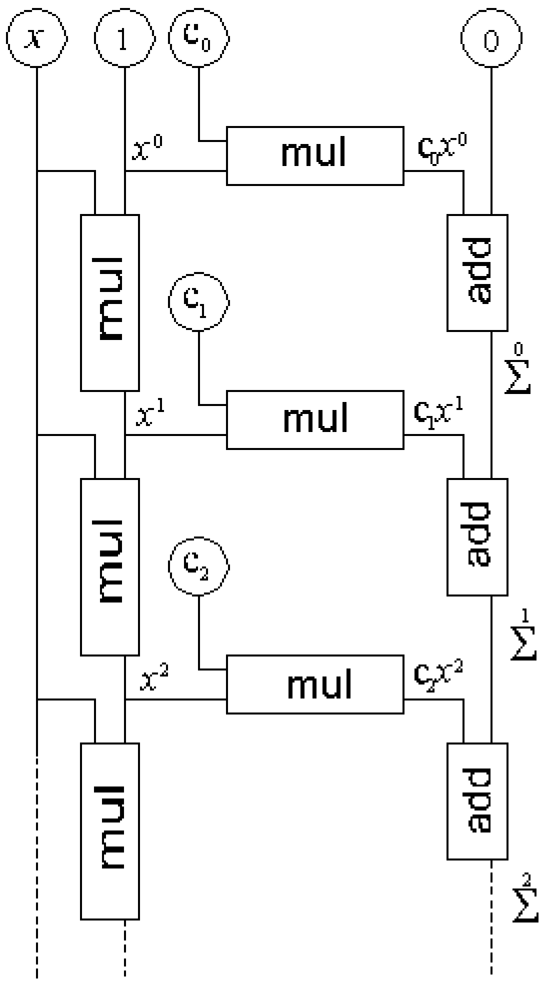

Figure 16.1: Dependences in example 16.10.

There are two continued dependence chains, one for calculating x^i and one for calculating the sum. The MULSD instruction has a latency of 6, while the ADDSD has a latency of 4. The vertical multiplication chain is therefore more critical than the addition chain. The additions have to wait for $c_i x^i$, which come 6 clocks after x^i, and later than the preceding additions. If nothing else limits the performance, then we can expect this code to take 6 clocks per iteration.

Throughput appears not to be a limiting factor because the multiplication unit can start 3 multiplications in the 6 clock cycles, and we need only 2. There are 3 uops to port 1, so port throughput is not a limiting factor either.

However, this loop does not take 6 clock cycles per iteration as expected, but 8. The explanation is as follows: Both multiplications have to wait for the value of x^{i-1} in XMM1 from the preceding iteration. Thus, both multiplications are ready to start at the same time. We would like the vertical multiplication in the ladder of figure 16.1 to start first, because it is part of the most critical dependence chain. But the microprocessor sees no reason to swap the order of the two multiplications, so the horizontal multiplication on figure 16.1 starts first. The vertical multiplication is delayed for 2 clock cycles, which is the reciprocal throughput of the floating-point multiplication unit. This explains the extra delay of 2 clocks per iteration.

The problem can be solved by delaying the horizontal multiplication:

```
; Example 16.11 (example 16.10 improved):
    MOVSD  XMM2, [X]                  ; XMM2 = x
    MOVSD  XMM1, [ONE]                ; XMM1 = x^i
    PXOR   XMM0, XMM0                 ; XMM0 = sum. Initialize to 0
    MOV    EAX,  OFFSET DS:coeff      ; point to c[i]
A:  MOVSD  XMM3, [ZERO]               ; set to 0
    ORPD   XMM3, XMM1                 ; set to XMM1 = x^i
    MULSD  XMM1, XMM2                 ; x^(i+1)   (vertical multipl.)
    MULSD  XMM3, [EAX]                ; c[i]*x^i  (horizontal multipl.)
    ADD    EAX,  8                    ; point to c[i+1]
    CMP    EAX, OFFSET DS:coeff_end   ; stop at end of list
    ADDSD  XMM0, XMM3                 ; sum += c[i] * x^i
    JB     A
```

XMM1 is now copied to XMM3 by setting XMM3 = 0 OR XMM1. This delays the horizontal multiplication by 4 clocks, so that the vertical multiplication can start first. ORPD uses a different execution unit so that it suffers an additional latency of 1 clock for transferring data to the other unit. Therefore, the ORPD does not use port 1 when the vertical multiplication needs it. The loop can now be executed in 6 clocks per iteration. The price we have to pay for this is that the last addition is delayed by an extra 4 clocks. (The 4 clocks are calculated as 1 clock additional latency before and after the ORPD for going to a different execution unit and back again + 2 clock latency for the ORPD). We might use MOVAPD XMM3,XMM1 instead of this weird way of copying XMM1 to XMM3, but MOVAPD has a longer latency so that we run the risk that the horizontal multiplication uses the multiplication unit when the vertical multiplication in the *next* iteration needs it.

You may set XMM3 to zero in the above code by copying 0 from a memory location using MOVSD or from a register that has been set to zero outside the loop using MOVAPD. But don't use PXOR XMM3,XMM3 inside the loop for setting XMM3 to 0. This would put an extra load on port 1 and thereby increase the risk that port 1 is occupied when the critical vertical multiplication needs it.

In situations like this, it is difficult to predict whether a port will be vacant when a critical uop needs it. This can only be determined by experimental testing, including the preceding code.

I have used XMM registers rather than floating-point registers in this example because of the shorter latency. The upper half of the XMM registers are not used in my example, but the upper half of the registers could be used at no extra cost for another Taylor expansion or for calculating every second term in the sum.

It is common to stop a Taylor expansion when the terms become negligible. However, it may be wise to always include the maximum number of terms in order to keep the repetition count constant so that the loop control branch is not mispredicted. The misprediction penalty is far more than the price of a few extra iterations. Set the MXCSR register to "Flush to zero" mode in order to avoid the possible penalty of underflows.

Loops with branches inside

The next example calculates x^n, where x is a floating-point number and n is a positive integer. This is done most efficiently by repeatedly squaring x and multiplying together the factors that correspond to the binary digits in n. The algorithm can be expressed by the C++ code:

```
// calculate power = pow(x,n) where n is a positive integer:
double x, xp, power;
unsigned int n, i;
xp = x;  power = 1.0;
for (i = n; i != 0; i >>= 1) {
   if (i & 1) power *= xp;
   xp *= xp;}
```

The corresponding assembly code is:

```
;   Example 16.12
.DATA
X      DQ    ?
POWER DQ    ?
ONE    DD    1.0
N      DD    ?

.CODE
      FLD   [ONE]          ; power init. = 1.0
      FLD   [X]            ; ST(0) = xp, ST(1) = power
      MOV   EAX, [N]       ; EAX = i
A:    SHR   EAX, 1         ; shift right i
      JNC   B              ; test the shifted-out bit
      FMUL  ST(1),ST(0)    ; power *= xp
B:    FMUL  ST(0),ST(0)    ; xp *= xp
      JNZ   A              ; stop when i = 0
      FSTP  ST(0)          ; discard xp
      FSTP  [POWER]        ; store result
```

This loop has two continued dependence chains, xp and power. Both have a latency of 7 for the FMUL instruction. The first multiplication is sometimes skipped, so the second multiplication is the limiting factor. We have the same problem as in the previous example that the two multiplications are ready to start simultaneously, and the least critical multiplication comes first. The reciprocal throughput for FMUL is 2, so the loop will take 7+2 = 9 clocks per iteration if both branches are predicted perfectly.

Branches inside small loops should generally be avoided on the P4 for three reasons:

1. branches reduce the uop delivery rate from the trace cache (see page 78).
2. branch mispredictions are expensive, especially in long dependence chains. A misprediction typically costs 45 uops on the P4.
3. branches inside a loop may hamper the prediction of the loop control branch.

Trace cache delivery is not a limiting factor in example 16.12. The JNC B branch follows a pattern defined by the binary bits of n. The branch predictor is generally good at predicting such patterns, so we may have perfect prediction if n is constant. The JNZ A branch is correlated with the preceding branch and will stop the loop after a distinct history pattern of the `JNC B`. It may therefore, for certain values of *n*, be predicted even when the loop count exceeds 17.

However, this applies only as long as *n* is constant. Any change in *n* may lead to several mispredictions and the performance will be extremely poor. We may therefore replace the inner branch by conditional moves:

```
;   Example 16.13
      FLD     [ONE]          ; temporary 1.0
      FLD     [ONE]          ; power init. = 1.0
      FLD     [X]            ; ST(0) = xp, ST(1) = power
      MOV     EAX, [N]       ; EAX = i
A:    FLD     ST(0)          ; load a temporary copy of xp
      SHR     EAX, 1         ; shift right i
      FCMOVNC ST(0),ST(3)    ; replace xp by 1.0 if bit = 0
      FMULP   ST(2),ST(0)    ; power *= (i & 1) ? xp : 1.0
      FMUL    ST(0),ST(0)    ; xp *= xp
      JNZ     A              ; stop when i = 0
      FSTP    ST(0)          ; discard xp
      FSTP    [POWER]        ; store result
      FSTP    ST(0)          ; discard temporary 1.0
```

Here, we are keeping the conditional move out of the critical dependence chain by choosing between `xp` and `1.0`, rather than between `power` and `power*xp`. Otherwise, we would add the latency of the conditional move to the clock count per iteration. Furthermore, we have reduced the execution time to 7 clocks per iteration by using the same method as in example 16.11. `FLD ST(0)` plays the same role in example 16.13 as `ORPD XMM3,XMM1` in example 16.11.

The repetition count for this loop is the number of significant bits in *n*. If this value often changes, then you may repeat the loop the maximum number of times in order to make the loop control branch predictable. This requires, of course, that there is no risk of overflow in the multiplications.

Changing the code of example 16.13 to use XMM registers is no advantage, unless you can handle data in parallel, because conditional moves in XMM registers are complicated to implement (see page 109).

16.4 Macro loops (all processors)

If the repetition count for a loop is small and constant, then it is possible to unroll the loop completely. The advantage of this is that calculations that depend only on the loop counter can be done at assembly time rather than at execution time. The disadvantage is, of course, that it takes up more space in the trace cache or code cache.

The MASM language includes a powerful macro language that is useful for this purpose. If, for example, we need a list of square numbers, then the C++ code may look like this:

```
int squares[10];
for (int i=0; i<10; i++) squares[i] = i*i;
```

The same list can be generated by a macro loop in MASM language:

```
; Example 16.14
.DATA
squares LABEL DWORD    ; label at start of array
I = 0                  ; temporary counter
REPT 10                ; repeat 10 times
   DD  I * I           ; define one array element
   I = I + 1           ; increment counter
ENDM                   ; end of REPT loop
```

Here, `I` is a preprocessing variable. The `I` loop is run at assembly time, not at execution time. The variable `I` and the statement `I = I + 1` never make it into the final code, and hence take no time to execute. In fact, example 16.14 generates no executable code, only data. The macro preprocessor will translate the above code to:

```
squares LABEL DWORD    ; label at start of array
   DD  0
   DD  1
   DD  4
   DD  9
   DD  16
   DD  25
   DD  36
   DD  49
   DD  64
   DD  81
```

Now, let's return to the power example (example 16.12). If *n* is known at assembly time, then the power function can be implemented using the following macro:

```
; This macro will raise two packed double-precision floats in X
; to the power of N, where N is a positive integer constant.
; The result is returned in Y. X and Y must be two different
; XMM registers. X is not preserved.
; (Only for processors with SSE2)
INTPOWER MACRO X, Y, N
   LOCAL I, YUSED              ; define local identifiers
   I = N                       ; I used for shifting N
   YUSED = 0                   ; remember if Y contains valid data
   REPT 32                     ; maximum repeat count is 32
      IF I AND 1               ; test bit 0
         IF YUSED              ; If Y already contains data
            MULPD Y, X         ; multiply Y with a power of X
         ELSE                  ; If this is first time Y is used:
            MOVAPD Y, X        ; copy data to Y
            YUSED = 1          ; remember that Y now contains data
         ENDIF                 ; end of IF YUSED
      ENDIF                    ; end of IF I AND 1
      I = I SHR 1              ; shift right I one place
      IF I EQ 0                ; stop when I = 0
         EXITM                 ; exit REPT 32 loop prematurely
      ENDIF                    ; end of IF I EQ 0
      MULPD X, X               ; square X
   ENDM                        ; end of REPT 32 loop
ENDM                           ; end of INTPOWER macro definition
```

This macro generates the minimum number of instructions needed to do the job. There is no loop overhead, prolog or epilog in the final code. And, most importantly, no branches. All branches have been resolved by the macro preprocessor. To calculate `XMM0` to the power of 12, you write:

```
INTPOWER XMM0, XMM1, 12
```

This will be resolved to:

```
MULPD   XMM0, XMM0        ; x^2
MULPD   XMM0, XMM0        ; x^4
MOVAPD  XMM1, XMM0        ; save x^4
MULPD   XMM0, XMM0        ; x^8
MULPD   XMM1, XMM0        ; x^4 * x^8 = x^12
```

This even has fewer instructions than the optimized loop (example 16.13). The expanded macro takes 25 clock cycles in this example.

17 Single-Instruction-Multiple-Data programming

Since there are technological limits to the maximum clock frequency of microprocessors, the trend goes towards increasing processor throughput by handling multiple data in parallel.

When optimizing code, it is important to consider if there are data that can be handled in parallel. The principle of Single-Instruction-Multiple-Data (SIMD) programming is that a vector or set of data are packed together in one large register and handled together in one operation.

Multiple data can be packed into 64-bit or 128-bit registers in the following ways:

data type	data per pack	register size	instruction set	microprocessor
8-bit integer	8	64 bit (MMX)	MMX	PMMX and later
16-bit integer	4	64 bit (MMX)	MMX	PMMX and later
32-bit integer	2	64 bit (MMX)	MMX	PMMX and later
64-bit integer	1	64 bit (MMX)	SSE2	P4 and later
32-bit float	2	64 bit (MMX)	3DNow	AMD only
8-bit integer	16	128 bit (XMM)	SSE2	P4 and later
16-bit integer	8	128 bit (XMM)	SSE2	P4 and later
32-bit integer	4	128 bit (XMM)	SSE2	P4 and later
64-bit integer	2	128 bit (XMM)	SSE2	P4 and later
32-bit float	4	128 bit (XMM)	SSE	P3 and later
64-bit float	2	128 bit (XMM)	SSE2	P4 and later

All these packing modes are available on the latest microprocessors from Intel and AMD, except for the 3DNow mode, which is available only on AMD processors. Whether the different instruction sets are supported on a particular microprocessor can be determined with the `CPUID` instruction, as explained on page 25. The 64-bit MMX registers cannot be used together with the floating-point registers. The 128-bit XMM registers can only be used if supported by the operating system. See page 25 for how to check if the use of XMM registers is enabled.

You may make two or more versions of the critical part of your code: one that will run on old microprocessors, and one that uses the most advantageous packing mode and instruction set. The program should automatically select the version of the code that is appropriate for the system on which it is running.

Choose the smallest data size that fits your purpose in order to pack as many data as possible into one register. Mathematical computations may require double precision (64-bit) floats in order to avoid loss of precision in the intermediate calculations, even if single precision is sufficient for the final result.

Before you choose to use SIMD instructions for integer operations, you have to consider whether the resulting code will be faster than the simple integer instructions in 32-bit registers. Simple operations such as integer additions take four times as long in SIMD registers as in 32-bit registers on the P4. The SIMD instructions are therefore only advantageous for integer additions if they can handle at least four data in parallel. Loading and storing memory operands take no longer for 64-bit and 128-bit registers than for 32-bit registers. Integer shift and multiplication is faster in 64-bit and 128-bit registers than in 32-bit registers on the P4. With SIMD code, you may spend more instructions on trivial things such as moving data into the right positions in the registers and emulating conditional moves, than on the actual calculations. Example 17.1 below is an example of this.

For floating-point calculations on the P4, it is often advantageous to use XMM registers, even if there are no opportunities for handling data in parallel. The latency of floating-point operations is shorter in XMM registers than in floating-point registers, and you can make conversions between integers and floating-point numbers without using a memory intermediate. Furthermore, you get rid of the annoying floating-point register stack.

Memory operands for SIMD instructions have to be properly aligned. See page 27 for how to align data in memory. The alignment requirement makes it complicated to pass 64-bit and 128-bit function parameters on the stack (see Intel's optimization reference manual). It is therefore recommended to pass 64-bit and 128-bit parameters in registers or through a pointer, rather than on the stack.

Conditional moves in SIMD registers

Let's repeat example 16.13 page 106 with two double-precision floats in XMM registers. This enables us to compute x_0^{n0} and x_1^{n1} in parallel:

```
; Example 17.1 (P4)
DATA SEGMENT PARA PUBLIC 'DATA'
ONE  DQ  1.0, 1.0
X    DQ  ?, ?                ; X0, X1
N    DD  ?, ?                ; N0, N1
DATA ENDS
CODE SEGMENT BYTE PUBLIC 'CODE'
; register use:
; XMM0 = xp
; XMM1 = power
; XMM2 = i  (i0 and i1 each stored twice as DWORD integers)
; XMM3 = 1.0 if not(i & 1)
; XMM4 = xp if (i & 1)

    MOVQ      XMM2, [N]      ; load N0, N1
    PUNPCKLDQ XMM2, XMM2     ; copy to get N0, N0, N1, N1
    MOVAPD    XMM0, [X]      ; load X0, X1
    MOVAPD    XMM1, [ONE]    ; power init. = 1.0
    MOV       EAX, [N]       ; N0
    OR        EAX, [N+4]     ; N0 OR N1 to get highest significant bit
    XOR       ECX, ECX       ; 0 if N0 and N1 both zero
    BSR       ECX, EAX       ; compute repeat count for max(N0,N1)

A:  MOVDQA    XMM3, XMM2     ; copy i
    PSLLD     XMM3, 31       ; get least significant bit of i
    PSRAD     XMM3, 31       ; copy to all bit positions
    PSRLD     XMM2, 1        ; i >>= 1
    MOVAPD    XMM4, XMM0     ; copy of xp
    ANDPD     XMM4, XMM3     ; xp if bit = 1
    ANDNPD    XMM3, [ONE]    ; 1.0 if bit = 0
    ORPD      XMM3, XMM4     ; (i & 1) ? xp : 1.0
    MULPD     XMM1, XMM3     ; power *= (i & 1) ? xp : 1.0
    MULPD     XMM0, XMM0     ; xp *= xp
    SUB       ECX, 1
    JNS       A              ; repeat ECX+1 times
```

Conditional moves in SIMD registers are implemented by generating a mask of all 1's representing the condition; AND'ing the first operand with the mask; AND'ing the second operand with the inverted mask; and OR'ing the two together. The mask can be generated by a compare instruction or, as here, by shifting a bit into the most significant position and then shifting it arithmetically to copy it into all positions. The arithmetic shift does not exist in a 64-bit version, so we have to use the 32-bit version with two identical copies of the condition operand.

The repeat count of the loop is calculated separately outside the loop in order to reduce the number of instructions inside the loop.

Timing analysis for example 17.1 in P4: There are four continued dependence chains: XMM0: 6 clocks, XMM1: 6 clocks, XMM2: 2 clocks, ECX: ½ clock. Throughput for the different execution units: MMX-SHIFT: 3 uops, 6 clocks. MMX-ALU: 3 uops, 6 clocks. FP-MUL: 2 uops, 4 clocks. Throughput for port 1: 8 uops, 8 clocks. Thus, the loop appears to be limited by port 1 throughput. The best timing we can hope for is 8 clocks per iteration which is the number of uops that must go to port 1. However, three of the continued dependence chains are interconnected by two broken, but quite long, dependence chains involving XMM3 and XMM4, which take 22 and 18 clocks, respectively. This tends to hinder the optimal reordering of uops. The measured time is approximately 10 uops per iteration. This timing actually requires a quite impressive reordering capability, considering that several iterations must be overlapped and several dependence chains interwoven in order to satisfy the restrictions on all ports and execution units.

In situations like this where it is difficult to obtain the optimal reordering of uops, it may require some experimentation to find the optimal solution. By experimentation I found that the code in example 17.1 can be made approximately 16 clocks faster in total by modifying it to the following:

```
    ; Example 17.2 (P4)
    MOVQ       XMM2, [N]      ; load N0, N1
    PUNPCKLDQ  XMM2, XMM2     ; copy to get N0, N0, N1, N1
    MOVAPD     XMM1, [ONE]    ; power init. = 1.0
    MOVAPD     XMM0, [X]      ; load X0, X1
    MOV        EAX, [N]       ; N0
    OR         EAX, [N+4]     ; N0 OR N1 to get highest significant bit
    XOR        ECX, ECX       ; 0 if N0 and N1 both zero
    BSR        ECX, EAX       ; compute repeat count for max(N0,N1)

A:  MOVDQA     XMM3, XMM2     ; copy i
    MOVDQA     XMM4, [ONE]    ; temporary 1.0
    PSLLD      XMM3, 31       ; get least significant bit of i
    PSRAD      XMM3, 31       ; copy to all bit positions
    PSRLD      XMM2, 1        ; i >>= 1
    PXOR       XMM4, XMM0     ; get bits that differ between xp and 1.0
    PANDN      XMM3, XMM4     ; mask out if (i & 1)
    XORPD      XMM3, XMM0     ; (i & 1) ? xp : 1.0
    MULPD      XMM0, XMM0     ; xp *= xp
    MULPD      XMM1, XMM3     ; power *= (i & 1) ? xp : 1.0
    SUB        ECX, 1
    JNS        A              ; repeat ECX+1 times
```

Here I have used PXOR and PANDN, rather than XORPD and ANDNPD, where the result of the operation may not be a valid floating-point number.

These examples shows that conditional moves in SIMD registers are quite complicated and involve considerable dependence chains. However, conditional moves are unavoidable in SIMD programming if the conditions are not the same for all operands in a pack. In the above example, the conditional moves can only be replaced by branches if n_0 and n_1 have the same value.

Conditional moves in 32-bit registers are no faster than in SIMD registers. The reason why conditional moves are so complex and inefficient is that they have three dependencies, while the hardware design does not allow any uop to have more than two dependencies.

Packing operands in 32-bit registers

Sometimes it is possible to handle packed data in 32-bit registers. You may use this method to take advantage of the fact that 32-bit operations are fast, or to make the code compatible with old microprocessors.

A 32-bit register can hold two 16-bit integers, four 8-bit integers, or 32 Booleans. When doing calculations on packed integers in 32-bit registers, you have to take special care to avoid carries from one operand going into the next operand if overflow is possible. The following example adds 2 to all four bytes in EAX:

```
; Example 17.3
MOV     EAX, [ESI]       ; read 4-bytes operand
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     EAX, EBX         ; combine bits again
MOV     [EDI], EAX       ; store result
```

Here the highest bit of each byte is masked out to avoid a possible carry from each byte into the next one when adding. The code is using XOR rather than ADD to put back the high bit again, in order to avoid carry. If the second addend may have the high bit set as well, it must be masked too. No masking is needed if none of the two addends have the high bit set.

The next example finds the length of a zero-terminated string by searching for the first byte of zero. It is faster than using REPNE SCASB if the string is long or the branch misprediction penalty is not severe:

```
        ; Example 17.4
_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 if not in 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
```

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. The code in example 17.4 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 example 17.3, 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 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.

18 Problematic Instructions

18.1 XCHG (all processors)

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

18.2 Shifts and rotates (P4)

Shifts and rotates on general purpose registers are slow on the P4. You may consider using MMX or XMM registers instead or replacing left shifts by additions.

18.3 Rotates through carry (all processors)

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

18.4 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 some processors.

`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.

Moving data in XMM registers is generally faster than `REP MOVSD` and `REP STOSD`. See page 130 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 on P2 and P3. 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.

On PPro, P2 and P3, `REP MOVS` and `REP STOS` can perform fast by moving an entire cache line at a time. 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 write-back 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.

On P4, the number of clock cycles for `REP MOVSD` is difficult to predict, but it is always faster to use `MOVDQA` for moving data, except possibly for small repeat counts if a loop would suffer a branch misprediction penalty.

`REP LOADS`, `REP SCAS`, and `REP CMPS` take more time per iteration than simple loops.

See page 111 for alternatives to `REPNE SCASB`. `REP CMPS` may suffer cache bank conflicts on PPro, P2 and P3 if bit 2-4 are the same in `ESI` and `EDI`.

18.5 Bit test (all processors)

`BT`, `BTC`, `BTR`, and `BTS` instructions should preferably be replaced by instructions like `TEST`, `AND`, `OR`, `XOR`, or shifts on P1, PMMX and P4. On PPro, P2 and P3, bit tests with a memory operand should be avoided.

18.6 Integer multiplication (all processors)

An integer multiplication takes approximately 9 clock cycles on P1 and PMMX; 4 on PPro, P2 and P3; and 14 on P4. 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`. For example `IMUL EAX,5` can be replaced by `LEA EAX,[EAX+4*EAX]`. On the P4, `SHL` and `LEA` with a scale factor are also relatively slow, so the fastest way on this processor is to use additions.

Multiplying a register with a constant can be done with the following macro, which uses only additions:

```
    ; This macro multiplies an integer by a constant, using only
    ; additions. Parameters:
    ; REG1: an 8-bit, 16-bit or 32-bit register containing the number
    ;       to multiply. The result will be returned in REG1.
    ; REG2: a spare register of the same size. (will not be used if
    ;       FACTOR is a power of 2).
    ; FACTOR: a positive integer constant to multiply by.
    MULTIPLY MACRO REG1, REG2, FACTOR
    LOCAL N, REG2USED
      N = FACTOR            ; N will be shifted to get the bits of FACTOR
      REG2USED = 0          ; remember when REG2 is used
      REPT 32               ; loop through all bits of FACTOR
        IF N EQ 1           ; finished when N = 1
          IF REG2USED
            add REG1, REG2  ; add the two registers
          ENDIF
          EXITM             ; REPT loop always exits here
        ENDIF
        IF N AND 1          ; add value of REG1 if N odd
          IF REG2USED
            add REG2, REG1  ; REG2 already contains data, add more data
          ELSE
            mov REG2, REG1  ; copy data to REG2
            REG2USED = 1    ; remember that REG2 contains data
          ENDIF
        ENDIF
        add REG1, REG1      ; multiply by 2
        N = N SHR 1         ; shift right N one place
      ENDM                  ; end of REPT loop
    ENDM                    ; end of MULTIPLY macro
```

For example, `IMUL EAX,100` can be replaced by

```
    MULTIPLY EAX, EBX, 100
```

which will be expanded to:
```
    ADD EAX,EAX   ; 2*a
    ADD EAX,EAX   ; 4*a
    MOV EBX,EAX   ; copy 4*a
    ADD EAX,EAX   ; 8*a
    ADD EAX,EAX   ; 16*a
    ADD EAX,EAX   ; 32*a
    ADD EBX,EAX   ; (32+4)*a
    ADD EAX,EAX   ; 64*a
    ADD EAX,EBX   ; (64+32+4)*a
```

This method is considerably faster than using `MUL` or `IMUL` on the P4, unless the factor is very big. However, since this method uses many instructions, it should only be used if latency is critical and throughput is not critical. If you have opportunities for handling data in parallel, or if your code contains many integer multiply and shift operations, then it may be faster to use MMX or XMM registers.

18.7 Division (all processors)

Both integer division and floating-point division are quite time consuming on all processors. Various methods for reducing the number of divisions are explained on page 9. Several methods to improve code that contains division are discussed below.

Integer division by a power of 2 (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
```

Obviously, you should prefer the unsigned version if the dividend is certain to be non-negative.

Integer division by a constant (all processors)

Dividing by a constant can be done by multiplying with the reciprocal. A useful algorithm for integer division by a constant is as follows:

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 $2^b \le d$). 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.

<u>case A (d = 2b):</u>
result = x SHR b

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

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

Example:
Assume that you want to divide by 5.
5 = 101B.
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, just change the last line to `SHR EDX,3`.

In case B we would have:

```
    ADD     EAX, 1
    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 `ADD EAX,1` 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 for using 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 `ADD EAX,1` 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 the code works correctly.

You may want to replace the slow multiplication instruction with faster instructions as explained on page 113.

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 that 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.

```
    Example 18.1, repeated integer division with 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 w. 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 $0 \leq x < 2^{32}$, $0 < d < 2^{32}$.

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.

Floating-point division (all processors)

The time it takes to make a floating-point division depends on the precision. When floating-point registers are used, you can make division faster by specifying a lower precision in the floating-point control word. This also speeds up the FSQRT instruction (except on P1 and PMMX), but not any other instructions. When XMM registers are used, you don't have to change any control word. Just use single-precision instructions if your application allows this.

It is not possible to do a floating-point division and an integer division at the same time because they are using the same execution unit, except on P1 and PMMX.

Using reciprocal instruction for fast division (P3 and P4)

On P3 and P4, 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:

```
x0 = RCPSS(d)
x1 = x0 * (2 - d * x0) = 2 * x0 - d * x0 * x0
```

where x0 is the first approximation to the reciprocal of the divisor d, and x1 is a better approximation. You must use this formula before multiplying with the dividend.

```
; Example 18.2, fast division, single precision (P3, P4)
```

```
    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 approximately 26 clock cycles (P4) with a precision of 23 bits. Increasing the precision further by repeating the Newton-Raphson formula with double precision 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 integer divisions with truncation on packed word size integers in approximately 45 clock cycles on the P4. It gives exactly the same results as the `DIV` instruction for 0 ≤ dividend ≤ 7FFFFH and 0 < divisor ≤ 7FFFFH:

```
    ; Example 18.3, integer division with packed 16-bit words (P4):
    ; compute QUOTIENTS = DIVIDENDS / DIVISORS
    MOVQ      XMM1, [DIVISORS]    ; load four divisors
    MOVQ      XMM2, [DIVIDENDS]   ; load four dividends
    PXOR      XMM0, XMM0          ; temporary 0
    PUNPCKLWD XMM1, XMM0          ; convert divisors to DWORDs
    PUNPCKLWD XMM2, XMM0          ; convert dividends to DWORDs
    CVTDQ2PS  XMM1, XMM1          ; convert divisors to floats
    CVTDQ2PS  XMM2, XMM2          ; convert dividends to floats
    RCPPS     XMM0, XMM1          ; approximate reciprocal of divisors
    MULPS     XMM1, XMM0          ; improve precision with..
    MULPS     XMM1, XMM0          ; Newton-Raphson method
    ADDPS     XMM0, XMM0
    SUBPS     XMM0, XMM1          ; reciprocal divisors (23 bit precision)
    MULPS     XMM0, XMM2          ; multiply with dividends
    CVTTPS2DQ XMM0, XMM0          ; truncate result of division
    PACKSSDW  XMM0, XMM0          ; convert quotients to WORD size
    MOVQ      XMM1, [DIVISORS]    ; load divisors again
    MOVQ      XMM2, [DIVIDENDS]   ; load dividends again
    PSUBW     XMM2, XMM1          ; dividends - divisors
    PMULLW    XMM1, XMM0          ; divisors * quotients
    PCMPGTW   XMM1, XMM2          ; -1 if quotient not too small
    PCMPEQW   XMM2, XMM2          ; make integer -1's
    PXOR      XMM1, XMM2          ; -1 if quotient too small
    PSUBW     XMM0, XMM1          ; correct quotient
    MOVQ      [QUOTIENTS], XMM0   ; save the four corrected quotients
```

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.

18.8 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. 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 addition 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.

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+00000000H]` 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`. If you happen to have a register that 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
```

`LEA` with a scale factor is slow on the P4, and may be replaced by additions. This applies only to the `LEA` instruction, not to instructions accessing memory.

18.9 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:

```
    B1:   FISTP [mem32]
          WAIT             ; wait for FPU to write before..
          MOV EAX,[mem32]  ; reading the result with the integer unit

    B2:   FILD [mem32]
          WAIT             ; wait for FPU to read value..
          MOV [mem32],EAX  ; before overwriting it with integer unit

    B3:   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. An 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 then I would recommend that you include the `WAIT` here in order to be safe. If rare and obsolete hardware platforms such as the combination of 80386 and 80287 can be ruled out, then you may omit the `WAIT`.

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 unnecessary 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 `B3` 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` on some processors and serve the same purpose.

18.10 FCOM + FSTSW AX (all processors)

The `FNSTSW` instruction is very slow on all processors. The PPro, P2, P3 and P4 processors have `FCOMI` instructions to avoid the slow `FNSTSW`. Using `FCOMI` instead of the common sequence `FCOM` / `FNSTSW AX` / `SAHF` will save 8 clock cycles on PPro, P2 and P3, and 4 clock cycles on P4. You should therefore use `FCOMI` to avoid `FNSTSW` wherever possible, even in cases where it costs some extra code.

On P1 and PMMX processors, which don't have `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 P1 and PMMX.

On the P1 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.

It is sometimes faster to use integer instructions for comparing floating-point values, as described on page 127 and 128.

18.11 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, and then multiply by the divisor. (See page 125 on how to truncate on processors that don't have truncate instructions).

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.

18.12 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 page 125 on how to truncate on processors that don't have truncate instructions. On P3 and P4, use the conversion instructions such as `CVTSS2SI` and `CVTTSS2SI`.

18.13 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]
```

On P4, you can make these operations in XMM registers without the need for a memory intermediate (see page 128).

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
```

18.14 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.

18.15 FSQRT (P3 and P4)

A fast way of calculating an approximate square root on the P3 and P4 is to multiply the reciprocal square root of x by x:

```
    SQRT(x) = x * RSQRT(x)
```

The instruction RSQRTSS or RSQRTPS gives the reciprocal square root 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:

```
    x0 = RSQRTSS(a)
    x1 = 0.5 * x0 * (3 - (a * x0)) * x0)
```

where x0 is the first approximation to the reciprocal square root of a, and x1 is a better approximation. The order of evaluation is important. You must use this formula before multiplying with a to get the square root.

18.16 FLDCW (PPro, P2, P3, P4)

The PPro, P2 and P3 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.

On the P4, this stall is even longer, approximately 143 clocks. But the P4 has made a special case out of the situation where the control word is alternating between two different values. This is the typical case in C++ programs where the control word is changed to specify truncation when a floating-point number is converted to integer, and changed back to rounding after this conversion. The latency for FLDCW is 3 when the new value loaded is the same as the value of the control word before the preceding FLDCW. The latency is still 143, however, when loading the same value into the control word as it already has, if this is not the same as the value it had one time earlier.

See page 125 on how to convert floating-point numbers to integers without changing the control word. On P3 and P4, use truncation instructions such as CVTTSS2SI instead.

18.17 Bit scan (P1 and PMMX)

BSF and BSR are the poorest optimized instructions on the P1 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
        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
        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 later processors.

19 Special topics

19.1 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 P1 and PMMX. On later processors you may use either `FCOMPP` or twice `FSTP ST`, whichever fits best into the decoding sequence (PPro, P2, P3) or port load (P4).

It is not recommended to use `FFREE`.

19.2 Transitions between floating-point and MMX instructions (PMMX, P2, P3, P4)

It is not possible to use 64-bit MMX registers and 80-bit floating-point registers in the same part of the code. You must issue an `EMMS` instruction after the last instruction that uses 64-bit MMX registers if there is a possibility that later code uses floating-point registers. You may avoid this problem by using 128-bit XMM registers instead.

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 P2, P3 and P4 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.

19.3 Converting from floating-point to integer (All processors)

All conversions between floating-point registers and integer registers must go via a memory location:

```
        FISTP DWORD PTR [TEMP]
        MOV EAX, [TEMP]
```

On PPro, P2, P3 and especially P4, this code is likely to have a penalty for attempting to read from `[TEMP]` before the write to `[TEMP]` is finished. It doesn't help to put in a `WAIT`. 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 afterwards. This method is very slow on all processors. On PPro and later processors, the floating-point control word cannot be renamed, so all subsequent floating-point instructions must wait for the `FLDCW` instruction to retire. See page 123.

On the P3 and P4 you can avoid all these problems by using XMM registers instead of floating-point registers and use the `CVT..` instructions to avoid the memory intermediate. (On the P3, these instructions are only available in single precision).

Whenever you have a conversion from a floating-point register to an integer register, you should think of whether you can use rounding to nearest integer instead of truncation.

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.

19.4 Using integer instructions for floating-point operations

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. For example

```
FLD QWORD PTR [ESI] / FSTP QWORD PTR [EDI]
```

can be replaced by:

```
MOV EAX,[ESI] / MOV EBX,[ESI+4] / MOV [EDI],EAX / MOV [EDI+4],EBX
```

or:

```
MOVQ MM0,[ESI] / MOVQ [EDI],MM0
```

Many other manipulations are possible if you know how floating-point numbers are represented in binary format. The floating-point format used in registers as well as in memory is in accordance with the IEEE-754 standard. Future implementations are certain to use the same format. The floating-point format consists of three parts: the sign s, mantissa m, and exponent e:

$$x = s \cdot m \cdot 2^e.$$

The sign s is represented as one bit, where a zero means +1 and a one means -1. The mantissa is a value in the interval $1 \leq m < 2$. The binary representation of m always has a 1 before the radix point. This 1 is not stored, except in the long double (80 bits) format. Thus, the left-most bit of the mantissa represents ½, the next bit represents ¼, etc. The exponent e can be both positive and negative. It is not stored in the usual 2-complement signed format, but in a biased format where 0 is represented by the value that has all but the most significant bit = 1. This format makes comparisons easier. The value x = 0.0 is represented by setting all bits of m and e to zero. The sign bit may be 0 or 1 so we can actually distinguish between +0.0 and -0.0, but comparisons must of course treat +0.0 and -0.0 as equal. The bit positions are shown in this table:

precision	**mantissa**	**always 1**	**exponent**	**sign**
single (32 bits)	bit 0 - 22		bit 23 - 30	bit 31
double (64 bits)	bit 0 - 51		bit 52 - 62	bit 63
long double (80 bits)	bit 0 - 62	bit 63	bit 64 - 78	bit 79

From this table we can find that the value 1.0 is represented as 3F80,0000H in single precision format, 3FF0,0000,0000,0000H in double precision, and 3FFF,8000,0000,0000,0000H in long double precision.

Generating constants

It is possible to generate simple floating-point constants without using data in memory:

```
; generate four single-precision values = 1.0
PCMPEQD XMM0,XMM0   ; generate all 1's
PSRLD   XMM0,25     ; seven 1's
PSLLD   XMM0,23     ; shift into exponent field
```

To generate the constant 0.0, it is better to use `PXOR XMM0,XMM0` than `XORPS`, `XORPD`, `SUBPS`, etc., because the `PXOR` instruction is recognized by the P4 processor to be independent of the previous value of the register if source and destination are the same, while this is not the case for the other instructions.

Testing if a floating-point value is zero

To test if a floating-point number is zero, we have to test all bits except the sign bit, which may be either 0 or 1. For example:

```
FLD DWORD PTR [EBX] / FTST / FNSTSW AX / AND AH,40H / JNZ IsZero
```

can be replaced by

```
MOV EAX,[EBX] / ADD EAX,EAX / JZ IsZero
```

where the `ADD EAX,EAX` shifts out the sign bit. Double precision floats have 63 bits to test, but if denormal numbers can be ruled out, then you can be certain that the value is zero if the exponent bits are all zero. Example:

```
FLD QWORD PTR [EBX] / FTST / FNSTSW AX / AND AH,40H / JNZ IsZero
```

can be replaced by

```
MOV EAX,[EBX+4] / ADD EAX,EAX / JZ IsZero
```

Manipulating the sign bit

A floating-point number is negative if the sign bit is set and at least one other bit is set. Example (single precision):

```
MOV EAX,[NumberToTest] / CMP EAX,80000000H / JA IsNegative
```

You can change the sign of a floating-point number simply by flipping the sign bit. This is useful when XMM registers are used, because there is no XMM change sign instruction. Example:

```
; change sign of four single-precision floats in XMM0
CMPEQD XMM1,XMM1    ; generate all 1's
PSLLD  XMM1,31      ; 1 in the leftmost bit of each DWORD only
XORPS  XMM0,XMM1    ; change sign of XMM0
```

You can get the absolute value of a floating-point number by AND'ing out the sign bit:

```
; absolute value of four single-precision floats in XMM0
CMPEQD XMM1,XMM1    ; generate all 1's
PSRLD  XMM1,1       ; 1 in all but the leftmost bit of each DWORD
ANDPS  XMM0,XMM1    ; set sign bits to 0
```

You can extract the sign bit of a floating-point number:

```
; generate a bit-mask if single-precision floats in XMM0 are..
; negative or -0.0
PSRAD  XMM0,31      ; copy sign bit into all bit positions
```

Manipulating the exponent

You can multiply a non-zero number with a power of 2 by simply adding to the exponent:

```
MOVAPS  XMM0, [X]   ; four single-precision floats
MOVDQA  XMM1, [N]   ; four 32-bit integers
PSLLD   XMM1, 23    ; shift integers into exponent field
PADDD   XMM0, XMM1  ; X * 2^N
```

Likewise, you can divide by a power of 2 by subtracting from the exponent. Note that this code does not work if X is zero or if overflow or underflow is possible.

Manipulating the mantissa

You can convert an integer to a floating-point number in an interval of length 1.0 by putting bits into the mantissa field. The following code computes $x = n / 2^{32}$, where n in an unsigned integer in the interval $0 \le n < 2^{32}$, and the resulting x is in the interval $0 \le x < 1.0$.

```
DATA SEGMENT PARA PUBLIC 'DATA'
ONE    DQ  1.0
X      DQ   ?
N      DD   ?
DATA ENDS
CODE SEGMENT BYTE PUBLIC 'CODE'
MOVSD  XMM0, [ONE]   ; 1.0, double precision
MOVD   XMM1, [N]     ; N, 32-bit unsigned integer
PSLLQ  XMM1, 20      ; align N left in mantissa field
POR    XMM1, XMM0    ; combine mantissa and exponent
SUBSD  XMM1, XMM0    ; subtract 1.0
MOVSD  [X], XMM1     ; store result
```

In the above code, the exponent from 1.0 is combined with a mantissa containing the bits of n. This gives a double-precision float in the interval $1.0 \le x < 2.0$. The SUBSD instruction subtracts 1.0 to get x into the desired interval.

Comparing numbers

Thanks to the fact that the exponent is stored in the biased format and to the left of the mantissa, it is possible to use integer instructions for comparing positive floating-point numbers. Example (single precision):

```
FLD [a] / FCOMP [b] / FNSTSW AX / AND AH,1 / JNZ ASmallerThanB
```

can be replaced by:

```
    MOV EAX,[a] / MOV EBX,[b] / CMP EAX,EBX / JB ASmallerThanB
```

This method works only if you are certain that none of the numbers have the sign bit set. You may compare absolute values by shifting out the sign bit of both numbers. For double-precision numbers, you can make an approximate comparison by comparing the upper 32 bits using integer instructions.

19.5 Using floating-point instructions for integer operations

While there are no problems using integer instructions for moving floating-point data, it is not always safe to use floating-point instructions for moving integer data. For example, you may be tempted to use `FLD QWORD PTR [ESI]` / `FSTP QWORD PTR [EDI]` to move 8 bytes at a time. However, this method may fail if the data do not represent valid floating-point numbers. The `FLD` instruction may generate an exception and it may even change the value of the data. If you want your code to be compatible with processors that don't have MMX and XMM registers then you can only use the slower `FILD` and `FISTP` for moving 8 bytes at a time.

However, some floating-point instructions can handle integer data without generating exceptions or modifying data. For example, the `MOVAPS` instruction can be used for moving 16 bytes at a time on the P3 processor that doesn't have the `MOVDQA` instruction. You can determine whether a floating-point instruction can handle integer data by looking at the documentation in the "IA-32 Intel Architecture Software Developer's Manual" Volume 2. If the instruction can generate any floating-point exception, then it cannot be used for integer data. If the documentation says "none" for floating-point exceptions, then this instruction can be used for integer data. It is reasonable to assume that such code will work correctly on future processors, but there is no guarantee that it will work equally fast on future processors.

Most SIMD instructions are "typed", in the sense that they are intended for one type of data only. It seems quite odd, for example, that the P4 has three different instructions for OR'ing 128-bit registers. The instructions `POR`, `ORPS` and `ORPD` are doing exactly the same thing. Replacing one with another has absolutely no consequence on the P4 processor. However, in some cases there is a performance penalty for using the wrong type on the P3. It is unknown whether future processors and processors from other vendors also have a penalty for using the wrong type. The reason for this performance penalty is, I guess, that the processor may do certain kinds of optimizations on a chain of dependent floating-point instructions, which is only possible when the instructions are dedicated to floating-point data only. It is therefore recommended to always use the right type of instruction if such an instruction is available.

There are certain floating-point instructions that have no integer equivalent and which may be useful for handling integer data. This includes the `MOVAPS` and `MOVNTPS` instructions which are useful for moving 16 bytes of data from and to memory on the P3. The P4 has integer versions of the same instructions, `MOVDQA` and `MOVNTDQ`.

The `MOVSS` instruction may be useful for moving 32 bits of data from one XMM register to another, while leaving the rest of the destination register unchanged (not if source is a memory operand).

Most other movements of integer data within and between registers can be done with the various shuffle, pack, unpack and shift instructions.

Converting binary to decimal numbers

The `FBSTP` instruction provides a simple and convenient way of converting a binary number to decimal, although not necessarily the fastest method.

19.6 Moving blocks of data (All processors)

There are several ways to move large blocks of data. The most common method is REP MOVSD. See page 112 about the speed of this instruction.

In many cases it is faster to use instructions that move more than 4 bytes at a time. Make sure that both source and destination are aligned by 8 if you are moving 8 bytes at a time, and aligned by 16 if you are moving 16 bytes at a time. If the size of the block you want to move is not a multiple of 8, respectively 16, then it is better to pad the buffers with extra space in the end and move a little more data than needed, than to move the extra data using other methods.

On P1 and PMMX it is advantageous to use FILD and FISTP with 8 byte operands if the destination is not in the cache. You may roll out the loop by two (FILD / FILD / FXCH / FISTP / FISTP).

On P2 and P3 it is advantageous to use MMX registers for moving 8 bytes at a time 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.

On the P3, the fastest way of moving data is to use the MOVAPS instruction if the conditions on page 112 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
```

On the P3 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.

On the P4, the fastest way of moving blocks of data is to use MOVDQA. You may use MOVNTDQ if you don't want the destination to be cached, but MOVDQA is often faster. REP MOVSD may still be the best choice for small blocks of data if the block size is varying and a loop would suffer a branch misprediction.

For further advices on improving memory access see the Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual.

19.7 Self-modifying code (All processors)

The penalty for executing a piece of code immediately after modifying it is approximately 19 clocks for P1, 31 for PMMX, and 150-300 for PPro, P2 and P3. The P4 will purge the entire trace cache after self-modifying code. 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. It should only be used if the gain in speed is substantial and the modified code is executed so many times that the advantage outweighs the penalties for using self-modifying code.

20 Testing speed

The microprocessors in the Pentium family 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 measuring exactly how many clock cycles a piece of code takes.

On the PPro, P2, P3 and P4 processors, you have to insert `XOR EAX,EAX` / `CPUID` before and after each `RDTSC` to prevent it from executing in parallel with anything else. `CPUID` is a serializing instruction, which means that it flushes the pipeline and waits for all pending operations to finish before proceeding. This is very useful for testing purposes.

The `RDTSC` instruction cannot execute in virtual mode on the P1 and PMMX, so if you are testing DOS programs on these processors you must run in real mode.

The biggest problem when counting clock ticks is to avoid interrupts. Protected operating systems may not allow you to clear the interrupt flag, so you cannot avoid interrupts and task switches during the test. There are several alternative ways to overcome this problem:

1. Run the test code with a high priority to minimize the risk of interrupts and task switches.

2. If the piece of code you are testing is relatively short then you may repeat the test several times and assume that the lowest of the clock counts measured represents a situation where no interrupt has occurred.

3. If the piece of code you are testing takes so long time that interrupts are unavoidable then you may repeat the test many times and take the average of the clock count measurements.

4. Make a virtual device driver to clear the interrupt flag.

5. Use an operating system that allows clearing the interrupt flag (e.g. Windows 98 without network, in console mode).

6. Start the test program in real mode using the old DOS operating system.

My test programs use method 1, 2, 5 and 6. These programs are available at www.agner.org/assem/testp.zip. The test programs that use method 6 set up a segment descriptor table and switch to 32-bit protected mode with the interrupt flag cleared. You can insert the code you want to test into these test programs. You need a bootable disk with Windows 98 or earlier to get access to run the test programs in real mode.

Remember when you are measuring clock ticks that a piece of code always takes longer time the first few times it is executed where it is not in the code cache or trace cache. Furthermore, it may take three iterations before the branch predictor has adapted to the code.

The alignment effects on the PPro, P2 and P3 processors make time measurements very difficult on these processors. Assume that you have a piece code and you want to make a change which you expect to make the code a few clocks faster. The modified code does not have exactly the same size as the original. This means that the code below the modification will be aligned differently and the instruction fetch blocks will be different. If instruction fetch and decoding is a bottleneck, which is often the case on these processors, then the change in the alignment may make the code several clock cycles faster or slower. The change in the alignment may actually have a larger effect on the clock count than the modification you have made. So you may be unable to verify whether the modification in itself makes the code faster or slower. It can be quite difficult to predict where each instruction fetch block begins, as explained on page 61.

The P1, PMMX and P4 processors do not have these alignment problems. The P4 does, however, have a somewhat similar, though less severe, effect. This effect is caused by changes in the alignment of uops in the trace cache. The time it takes to jump to the least common (but predicted) branch after a conditional jump instruction may differ by up to two clock cycles on different alignments if trace cache delivery is the bottleneck. The alignment of uops in the trace cache lines is difficult to predict (see page 78).

The processors in the Pentium family have special performance monitor counters which can count events such as cache misses, misalignments, branch mispredictions, etc. You need privileged access to set up these counters. The performance monitor counters are model specific. This means that you must use a different test setup for each microprocessor model. Details about how to use the performance monitor counters can be found in Intel's Software Developer's Manuals.

The test programs at www.agner.org/assem/testp.zip give access to the performance monitor counters when run under real mode DOS.

21 List of instruction timings for P1 and PMMX

21.1 Integer instructions (P1 and PMMX)

Explanation of column headings:

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 P1 unless preceded by a multicycle instruction (see page 56).
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 page 38 (P1) and 42 (PMMX).
f) only pairable if register is `AL`, `AX` or `EAX`.
g) add one clock cycle for decoding the repeat prefix unless preceded by a multicycle instruction (such as `CLD`).
h) pairs as if it were writing to the accumulator.
i) 9 if `SP` divisible by 4. See page 53.
j) on P1: 6 in privileged or real mode; 11 in nonprivileged; error in virtual mode. On PMMX: 8 and 13 clocks respectively.

21.2 Floating-point instructions (P1 and PMMX)

Explanation of column headings:
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 page 120.

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`.

21.3 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 P2 and P3).

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 on page 53.

22 List of instruction timings and uop breakdown for PPro, P2 and P3

Explanation of column headings:

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

Latency: This is the delay that the instruction generates in a dependence 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.

Reciprocal throughput: One divided by the maximum throughput for several instructions of the same kind. This is also called issue latency. For example, a reciprocal throughput of 2 for `FMUL` means that a new `FMUL` instruction can start executing 2 clock cycles after a previous `FMUL`.

22.1 Integer instructions (PPro, P2 and P3)

Instruction	Operands	Micro-ops						Latency	Reciprocal 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
MUL IMUL	(r),m	1			1			4	1
DIV IDIV	r8	2		1				19	12
DIV IDIV	r16	3		1				23	21
DIV IDIV	r32	3		1				39	37
DIV IDIV	m8	2		1	1			19	12
DIV IDIV	m16	2		1	1			23	21
DIV IDIV	m32	2		1	1			39	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						2
JMP	far	21			1				
JMP	r		1						2
JMP	m(near)		1		1				2
JMP	m(far)	21			2				
conditional jump	short/near		1						2
CALL	near		1	1		1	1		2
CALL	far	28			1	2	2		
CALL	r		1	2		1	1		2
CALL	m(near)		1	4	1	1	1		2
CALL	m(far)	28			2	2	2		
RETN			1	2	1				2
RETN	i		1	3	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				
PREFETCHT0/1/2 d)	m				1				
SFENCE d)						1	1		6

Notes:
a) faster under certain conditions: see page 112.
b) see page 112.
c) 3 if constant without base or index register
d) P3 only.

22.2 Floating-point instructions (PPro, P2 and P3)

Instruction	Operands	Micro-ops						Latency	Reciprocal 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	⅓ 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
FADD(P) FSUB(R)(P)	m	1			1			3-4	1
FMUL(P)	r	1						5	2 g)
FMUL(P)	m	1			1			5-6	2 g)
FDIV(R)(P)	r	1						38 h)	37
FDIV(R)(P)	m	1			1			38 h)	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		17-9	7					27-103	e)
FSINCOS		18-1	10					29-130	e)
F2XM1		17-4	8					66	e)
FYL2X		36-5	4					103	e)
FYL2XP1		31-5	3					98-107	e)
FPTAN		21-1	02					13-143	e)
FPATAN		25-8	6					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 uop 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 latency depends on precision specified in control word: 64 bits precision gives latency 38, 53 bits precision gives latency 32, 24 bits precision gives latency 18. Division by a power of 2 takes 9 clocks. Reciprocal throughput is 1/(latency-1).
i) faster for lower precision.

22.3 MMX instructions (P2 and P3)

Instruction	Operands	Micro-ops						Latency	Reciprocal throughput
		p0	p1	p01	p2	p3	p4		
MOVD MOVQ	r,r			1				1	½
MOVD MOVQ	r64,m32/64				1				1
MOVD MOVQ	m32/64,r64					1	1		1
PADD PSUB PCMP	r64,r64			1				1	1
PADD PSUB PCMP	r64,m64			1	1				1
PMUL PMADD	r64,r64	1						3	1
PMUL PMADD	r64,m64	1			1			3	1
PAND(N) POR PXOR	r64,r64			1				1	½
PAND(N) POR PXOR	r64,m64			1	1				1
PSRA PSRL PSLL	r64,r64/i		1					1	1
PSRA PSRL PSLL	r64,m64		1		1				1
PACK PUNPCK	r64,r64		1					1	1
PACK PUNPCK	r64,m64		1		1				1
EMMS		11						6 k)	
MASKMOVQ d)	r64,r64			1		1	1	2-8	2 - 30

PMOVMSKB d)	r32,r64		1					1	1
MOVNTQ d)	m64,r64					1	1		1 - 30
PSHUFW d)	r64,r64,i		1					1	1
PSHUFW d)	r64,m64,i		1		1			2	1
PEXTRW d)	r32,r64,i		1	1				2	1
PISRW d)	r64,r32,i		1					1	1
PISRW d)	r64,m16,i		1		1			2	1
PAVGB PAVGW d)	r64,r64			1				1	½
PAVGB PAVGW d)	r64,m64			1	1			2	1
PMIN/MAXUB/SW d)	r64,r64			1				1	½
PMIN/MAXUB/SW d)	r64,m64			1	1			2	1
PMULHUW d)	r64,r64	1						3	1
PMULHUW d)	r64,m64	1			1			4	1
PSADBW d)	r64,r64	2		1				5	2
PSADBW d)	r64,m64	2		1	1			6	2

Notes:
d) P3 only.
k) you may hide the delay by inserting other instructions between `EMMS` and any subsequent floating-point instruction.

22.4 XMM instructions (P3)

Instruction	**Operands**	**Micro-ops**						**Latency**	**Reciprocal throughput**
		p0	**p1**	**p01**	**p2**	**p3**	**p4**		
MOVAPS	r128,r128			2				1	1
MOVAPS	r128,m128				2			2	2
MOVAPS	m128,r128					2	2	3	2
MOVUPS	r128,m128				4			2	4
MOVUPS	m128,r128		1			4	4	3	4
MOVSS	r128,r128			1				1	1
MOVSS	r128,m32			1	1			1	1
MOVSS	m32,r128					1	1	1	1
MOVHPS MOVLPS	r128,m64			1				1	1
MOVHPS MOVLPS	m64,r128					1	1	1	1
MOVLHPS MOVHLPS	r128,r128			1				1	1
MOVMSKPS	r32,r128	1						1	1
MOVNTPS	m128,r128					2	2		2 - 15
CVTPI2PS	r128,r64		2					3	1
CVTPI2PS	r128,m64		2		1			4	2
CVT(T)PS2PI	r64,r128		2					3	1
CVTPS2PI	r64,m128		1		2			4	1
CVTSI2SS	r128,r32		2		1			4	2
CVTSI2SS	r128,m32		2		2			5	2
CVT(T)SS2SI	r32,r128		1		1			3	1
CVTSS2SI	r32,m128		1		2			4	2
ADDPS SUBPS	r128,r128		2					3	2
ADDPS SUBPS	r128,m128		2		2			3	2
ADDSS SUBSS	r128,r128		1					3	1
ADDSS SUBSS	r128,m32		1		1			3	1
MULPS	r128,r128	2						4	2
MULPS	r128,m128	2			2			4	2

MULSS	r128,r128	1						4	1
MULSS	r128,m32	1			1			4	1
DIVPS	r128,r128	2						48	34
DIVPS	r128,m128	2			2			48	34
DIVSS	r128,r128	1						18	17
DIVSS	r128,m32	1			1			18	17
AND(N)PS ORPS XORPS	r128,r128		2					2	2
AND(N)PS ORPS XORPS	r128,m128		2		2			2	2
MAXPS MINPS	r128,r128		2					3	2
MAXPS MINPS	r128,m128		2		2			3	2
MAXSS MINSS	r128,r128		1					3	1
MAXSS MINSS	r128,m32		1		1			3	1
CMPccPS	r128,r128		2					3	2
CMPccPS	r128,m128		2		2			3	2
CMPccSS	r128,r128		1					3	1
CMPccSS	r128,m32		1		1			3	1
COMISS UCOMISS	r128,r128		1					1	1
COMISS UCOMISS	r128,m32		1		1			1	1
SQRTPS	r128,r128	2						56	56
SQRTPS	r128,m128	2			2			57	56
SQRTSS	r128,r128	2						30	28
SQRTSS	r128,m32	2			1			31	28
RSQRTPS	r128,r128	2						2	2
RSQRTPS	r128,m128	2			2			3	2
RSQRTSS	r128,r128	1						1	1
RSQRTSS	r128,m32	1			1			2	1
RCPPS	r128,r128	2						2	2
RCPPS	r128,m128	2			2			3	2
RCPSS	r128,r128	1						1	1
RCPSS	r128,m32	1			1			2	1
SHUFPS	r128,r128,i		2	1				2	2
SHUFPS	r128,m128,i		2		2			2	2
UNPCKHPS UNPCKLPS	r128,r128		2	2				3	2
UNPCKHPS UNPCKLPS	r128,m128		2		2			3	2
LDMXCSR	m32	11						15	15
STMXCSR	m32	6						7	9
FXSAVE	m4096	116						62	
FXRSTOR	m4096	89						68	

23 List of instruction timings and uop breakdown for P4

Explanation of column headings:

Instruction: instruction name. cc means any condition code. For example, Jcc can be `JB`, `JNE`, etc.

Operands: r means any register, r32 means 32-bit register, etc.; m means any memory operand including indirect operands, m64 means 64-bit memory operand, etc.; i means any immediate constant.

Uops: number of micro-ops issued from instruction decoder and stored in trace cache.

Microcode: number of additional uops issued from microcode ROM.

Latency: the number of clock cycles from the execution of an instruction begins to the next *dependent* instruction can begin, if the latter instruction starts in the same execution unit. 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, infinity and exceptions increase the delays. The latency of moves to and from memory cannot be measured accurately because of the problem with memory intermediates explained on page 88. You should avoid making optimizations that rely on the latency of memory operations.

Additional latency: add this number to the latency if the next dependent instruction is in a different execution unit. There is no additional latency between ALU0 and ALU1.

Reciprocal throughput: This is also called issue latency. This value indicates the number of clock cycles from the execution of an instruction begins to a subsequent *independent* instruction can begin to execute in the same execution subunit. A value of 0.25 indicates 4 instructions per clock cycle.

Port: the port through which each uop goes to an execution unit. Two independent uops can start to execute simultaneously only if they are going through different ports.

Execution unit: Use this information to determine additional latency. When an instruction with more than one uop uses more than one execution unit, only the first and the last execution unit is listed.

Execution subunit: throughput measures apply only to instructions executing in the same subunit.

Backwards compatibility: Indicates the first microprocessor in the Intel 80x86 family that supported the instruction. The history sequence is: 8086, 80186, 80286, 80386, 80486, P1, PPro, PMMX, P2, P3, P4. Availability in processors prior to 80386 does not apply for 32-bit operands. Availability in PMMX and P2 does not apply to 128-bit packed instructions. Availability in P3 does not apply to 128-bit packed integer instructions and double precision floating-point instructions.

23.1 integer instructions

Instruction	Operands	Uops	Microcode	Latency	Additional latency	Reciprocal throughput	Port	Execution unit	Subunit	Backwards compatibility	Notes
Move instructions											
MOV	r,r	1	0	0.5	0.5-1	0.25	0/1	alu0/1		86	c
MOV	r,i	1	0	0.5	0.5-1	0.25	0/1	alu0/1		86	
MOV	r32,m	1	0	1	0	1	2	load		86	
MOV	r8/r16,m	2	0	1	0	1	2	load		86	
MOV	m,r	1	0	1		2	0	store		86	b,c
MOV	m,i	3	0			2	0,3	store		86	
MOV	r,sr	4	2			6				86	
MOV	sr,r/m	4	4	12	0	14				86	a,k
MOVNTI	m,r32	2	0			≈33				p4	
MOVZX	r,r	1	0	0.5	0.5-1	0.25	0/1	alu0/1		386	c
MOVZX	r,m	1	0	1	0	1	2	load		386	
MOVSX	r,r	1	0	0.5	0.5-1	0.5	0	alu0		386	c
MOVSX	r,m	2	0	1.5	0.5-1	1	2,0			386	
CMOVcc	r,r/m	3	0	6	0	3				ppro	a,e
XCHG	r,r	3	0	1.5	0.5-1	1	0/1	alu0/1		86	
XCHG	r,m	4	8	>100						86	
XLAT		4	0	3						86	
PUSH	r	2	0	1		2				86	
PUSH	i	2	0	1		2				186	
PUSH	m	3	0			2				86	
PUSH	sr	4	4			7				86	
POP	r	2	0	1	0	1				86	
POP	m	4	8			14				86	
POP	sr	4	5			13				86	
PUSHF(D)		4	4			10				86	
POPF(D)		4	8			52				86	
PUSHA(D)		4	10			19				186	
POPA(D)		4	16			14				186	
LEA	r,[r+r/i]	1	0	0.5	0.5-1	0.25	0/1	alu0/1		86	
LEA	r,[r+r+i]	2	0	1	0.5-1	0.5	0/1	alu0/1		86	
LEA	r,[r*i]	3	0	4	0.5-1	1	1	int,alu		386	
LEA	r,[r+r*i]	2	0	4	0.5-1	1	1	int,alu		386	
LEA	r,[r+r*i+i]	3	0	4	0.5-1	1	1	int,alu		386	
LAHF		1	0	4	0	4	1	int		86	
SAHF		1	0	0.5	0.5-1	0.5	0/1	alu0/1		86	d
SALC		3	0	5	0	1	1	int		86	
LDS, LES, ...	r,m	4	7			15				86	
LODS		4	3	6		6				86	
REP LODS		4	5n	≈ 4n+36						86	
STOS		4	2	6		6				86	
REP STOS		4	2n+3	≈ 3n+10						86	
MOVS		4	4	6		4				86	
REP MOVS		4	≈163+1.1n			≈ n				86	
BSWAP	r	3	0	7	0	2		int,alu		486	

IN, OUT	r,r/i	8	64			>1000				86	
PREFETCHCNTA	m	4	2			6				p3	
PREFETCHT0/1/2	m	4	2			6				p3	
SFENCE		4	2			40				p3	
LFENCE		4	2			38				p4	
MFENCE		4	2			100				p4	
Arithmetic instructions											
ADD, SUB	r,r	1	0	0.5	0.5-1	0.25	0/1	alu0/1		86	c
ADD, SUB	r,m	2	0	1	0.5-1	1				86	c
ADD, SUB	m,r	3	0	≥ 8		≥ 4				86	c
ADC, SBB	r,r	4	4	6	0	6	1	int,alu		86	
ADC, SBB	r,i	3	0	6	0	6	1	int,alu		86	
ADC, SBB	r,m	4	6	8	0	8	1	int,alu		86	
ADC, SBB	m,r	4	7	≥ 9		8				86	
CMP	r,r	1	0	0.5	0.5-1	0.25	0/1	alu0/1		86	c
CMP	r,m	2	0	1	0.5-1	1				86	c
INC, DEC	r	2	0	0.5	0.5-1	0.5	0/1	alu0/1		86	
INC, DEC	m	4	0	4		≥ 4				86	
NEG	r	1	0	0.5	0.5-1	0.5	0	alu0		86	
NEG	m	3	0			≥ 3				86	
AAA, AAS		4	27	90						86	
DAA, DAS		4	57	100						86	
AAD		4	10	22			1	int	fpmul	86	
AAM		4	22	56			1	int	fpdiv	86	
MUL, IMUL	r8/r32	4	6	16	0	8	1	int	fpmul	86	
MUL, IMUL	r16	4	7	17	0	8	1	int	fpmul	86	
MUL, IMUL	m8/m32	4	7-8	16	0	8	1	int	fpmul	86	
MUL, IMUL	m16	4	10	16	0	8	1	int	fpmul	86	
IMUL	r32,r	4	0	14	0	4.5	1	int	fpmul	386	
IMUL	r32,(r),i	4	0	14	0	4.5	1	int	fpmul	386	
IMUL	r16,r	4	5	16	0	9	1	int	fpmul	386	
IMUL	r16,r,i	4	5	15	0	8	1	int	fpmul	186	
IMUL	r16,m16	4	7	15	0	10	1	int	fpmul	186	
IMUL	r32,m32	4	0	14	0	8	1	int	fpmul	186	
IMUL	r,m,i	4	7	14	0	10	1	int	fpmul	186	
DIV	r8/m8	4	20	61	0	24	1	int	fpdiv	86	a
DIV	r16/m16	4	18	53	0	23	1	int	fpdiv	86	a
DIV	r32/m32	4	21	50	0	23	1	int	fpdiv	386	
IDIV	r8/m8	4	24	61	0	24	1	int	fpdiv	86	a
IDIV	r16/m16	4	22	53	0	23	1	int	fpdiv	86	a
IDIV	r32/m32	4	20	50	0	23	1	int	fpdiv	386	a
CBW		2	0	1	0.5-1	1	0	alu0		86	
CWD, CDQ		2	0	1	0.5-1	0.5	0/1	alu0/1		86	
CWDE		1	0	0.5	0.5-1	0.5	0	alu0		386	
SCAS		4	3			6				86	
REP SCAS		4	≈ 40+6n			≈4n				86	
CMPS		4	5			8				86	
REP CMPS		4	≈ 50+8n			≈4n				86	
Logic											
AND, OR, XOR	r,r	1	0	0.5	0.5-1	0.5	0	alu0		86	c
AND, OR, XOR	r,m	2	0	≥ 1	0.5-1	≥ 1				86	c
AND, OR, XOR	m,r	3	0	≥ 8		≥ 4				86	c
TEST	r,r	1	0	0.5	0.5-1	0.5	0	alu0		86	c
TEST	r,m	2	0	≥ 1	0.5-1	≥ 1				86	c
NOT	r	1	0	0.5	0.5-1	0.5	0	alu0		86	
NOT	m	4	0			≥ 4				86	
SHL, SHR, SAR	r,i	1	0	4	1	1	1	int	mmxsh	186	
SHL, SHR, SAR	r,CL	2	0	6	0	1	1	int	mmxsh	86	d
ROL, ROR	r,i	1	0	4	1	1	1	int	mmxsh	186	d

ROL, ROR	r,CL	2	0	6	0	1	1	int	mmxsh	86	d
RCL, RCR	r,1	1	0	4	1	1	1	int	mmxsh	86	d
RCL, RCR	r,i	4	15	16	0	15	1	int	mmxsh	186	d
RCL, RCR	r,CL	4	15	16	0	14	1	int	mmxsh	86	d
shl,shr,sar,rol,ror	m,i/CL	4	7-8	10	0	10	1	int	mmxsh	86	d
RCL, RCR	m,1	4	7	10	0	10	1	int	mmxsh	86	d
RCL, RCR	m,i/CL	4	18	18-28		14	1	int	mmxsh	86	d
SHLD, SHRD	r,r,i/CL	4	14	14	0	14	1	int	mmxsh	386	
SHLD, SHRD	m,r,i/CL	4	18	14	0	14	1	int	mmxsh	386	
BT	r,i	3	0	4	0	2	1	int	mmxsh	386	d
BT	r,r	2	0	4	0	1	1	int	mmxsh	386	d
BT	m,i	4	0	4	0	2	1	int	mmxsh	386	d
BT	m,r	4	12	12	0	12	1	int	mmxsh	386	d
BTR, BTS, BTC	r,i	3	0	6	0	2	1	int	mmxsh	386	
BTR, BTS, BTC	r,r	2	0	6	0	4	1	int	mmxsh	386	
BTR, BTS, BTC	m,i	4	7	18	0	8	1	int	mmxsh	386	
BTR, BTS, BTC	m,r	4	15	14	0	14	1	int	mmxsh	386	
BSF, BSR	r,r	2	0	4	0	2	1	int	mmxsh	386	
BSF, BSR	r,m	3	0	4	0	3	1	int	mmxsh	386	
SETcc	r	3	0	5	0	1	1	int		386	
SETcc	m	4	0	5	0	3	1	int		386	
CLC, STC		3	0	10	0	2				86	d
CMC		3	0	10	0	2				86	
CLD		4	7	52	0	52				86	
STD		4	5	48	0	48				86	
CLI		4	5	35		35				86	
STI		4	12	43		43				86	
Jump and call											
JMP	short/near	1	0	0	0	1	0	alu0	branch	86	
JMP	far	4	28	118		118	0			86	
JMP	r	3	0	4		4	0	alu0	branch	86	
JMP	m(near)	3	0	4		4	0	alu0	branch	86	
JMP	m(far)	4	31	11		11	0			86	
Jcc	short/near	1	0	0		2-4	0	alu0	branch	86	
J(E)CXZ	short	4	4	0		2-4	0	alu0	branch	86	
LOOP	short	4	4	0		2-4	0	alu0	branch	86	
CALL	near	3	0	2		2	0	alu0	branch	86	
CALL	far	4	34				0			86	
CALL	r	4	4	8			0	alu0	branch	86	
CALL	m(near)	4	4	9			0	alu0	branch	86	
CALL	m(far)	4	38				0			86	
RETN		4	0	2			0	alu0	branch	86	
RETN	i	4	0	2			0	alu0	branch	86	
RETF		4	33	11			0			86	
RETF	i	4	33	11			0			86	
IRET		4	48	24			0			86	
ENTER	i,0	4	12	26		26				186	
ENTER	i,n	4	45+24n			128+16n				186	
LEAVE		4	0	3		3				186	
BOUND	m	4	14	14		14				186	
INTO		4	5	18		18				86	
INT	i	4	84	644						86	
Other											
NOP		1	0	0		0.25	0/1	alu0/1		86	
PAUSE		4	2							p4	
CPUID		4	39-81			200-500				p5	
RDTSC		4	7			80				p5	

Notes:

a) Add 1 uop if source is a memory operand.

b) Uses an extra uop (port 3) if SIB byte used. A SIB byte is needed if the memory operand has more than one pointer register, or a scaled index, or ESP is used as base pointer.
c) Add 1 uop if source or destination, but not both, is a high 8-bit register (AH, BH, CH, DH).
d) Has false dependence on the flags in most cases.
e) Not available on PMMX
k) Latency is 12 in 16-bit real or virtual mode, 24 in 32-bit protected mode.

23.2 Floating-point instructions

Instruction	Operands	Uops	Microcode	Latency	Additional latency	Reciprocal throughput	Port	Execution unit	Subunit	Backwards compatibility	Notes
Move instructions											
FLD	r	1	0	6	0	1	0	mov		87	
FLD	m32/m64	1	0	≈ 7	0	1	2	load		87	
FLD	m80	3	4			6	2	load		87	
FBLD	m80	3	75			90	2	load		87	
FST(P)	r	1	0	6	0	1	0	mov		87	
FST(P)	m32/m64	2	0	≈ 7		2-3	0	store		87	
FSTP	m80	3	8			8	0	store		87	
FBSTP	m80	3	311			400	0	store		87	
FXCH	r	1	0	0	0	1	0	mov		87	
FILD	m32/64	2	0	≈ 10		1	2	load		87	
FILD	m16	3	3	≈ 10		6	2	load		87	
FIST	m32/64	2	0	≈ 10		2-3	0	store		87	
FIST	m16	3	0	≈ 10		2-4	0	store		87	
FISTP	m	3	0	≈ 10		2-4	0	store		87	
FLDZ		1	0			2	0	mov		87	
FLD1		2	0			2	0	mov		87	
FCMOVcc	st(0),r	4	0	2-4	1	4	1	fp		ppro	e
FFREE	r	3	0			4	0	mov		87	
FINCSTP, FDECSTP		1	0	0	0	1	0	mov		87	
FNSTSW	AX	4	0	11	0	3	1			287	
FSTSW	AX	6	0	11	0	3	1			287	
FNSTSW	m16	4	4			6	0			87	
FNSTCW	m16	4	4			6	0			87	
FLDCW	m16	4	7	(3)		(8)	0,2			87	f
Arithmetic instructions											
FADD(P),FSUB(R)(P)	r	1	0	5	1	1	1	fp	add	87	
FADD,FSUB(R)	m	2	0	5	1	1	1	fp	add	87	
FIADD,FISUB(R)	m32	3	0	5	1	2	1	fp	add	87	
FIADD,FISUB(R)	m16	3	4	6	0	6	1	fp	add	87	
FMUL(P)	r	1	0	7	1	2	1	fp	mul	87	
FMUL	m	2	0	7	1	2	1	fp	mul	87	
FIMUL	m32	3	4	7	1	6	1	fp	mul	87	
FIMUL	m16	3	0	7	1	2	1	fp	mul	87	
FDIV(R)(P)	r	1	0	43	0	43	1	fp	div	87	g,h
FDIV(R)	m	2	0	43	0	43	1	fp	div	87	g,h
FIDIV(R)	m32	3	0	43	0	43	1	fp	div	87	g,h
FIDIV(R)	m16	3	4	43	0	43	1	fp	div	87	g,h

FABS		1	0	2	1	1	1	fp	misc	87	
FCHS		1	0	2	1	1	1	fp	misc	87	
FCOM(P), FUCOM(P)	r	1	0	2	0	1	1	fp	misc	87	
FCOM(P)	m	2	0	2	0	1	1	fp	misc	387	
FCOMPP, FUCOMPP		2	0	2	0	1	1	fp	misc	87	
FCOMI(P)	r	3	0	10	0	3	0,1	fp	misc	ppro	
FICOM(P)	m32	3	0	2	0	2	1,2	fp	misc	87	
FICOM(P)	m16	4	4			6	1	fp	misc	87	
FTST		1	0	2	0	1	1	fp	misc	87	
FXAM		1	0	2	0	1	1	fp	misc	87	
FRNDINT		3	15	23	0	15	0,1			87	
FPREM		6	84	212			1	fp		87	
FPREM1		6	84	212			1	fp		387	
Math											
FSQRT		1	0	43	0	43	1	fp	div	87	g,h
FLDPI, etc.		2	0			3	1	fp		87	
FSIN		6	≈150	≈180		≈170	1	fp		387	
FCOS		6	≈175	≈207		≈207	1	fp		387	
FSINCOS		7	≈178	≈216		≈211	1	fp		387	
FPTAN		6	≈160	≈230		≈200	1	fp		87	
FPATAN		3	92	≈187		≈153	1	fp		87	
FSCALE		3	24	57		66	1	fp		87	
FXTRACT		3	15	20		20	1	fp		87	
F2XM1		3	45	≈165		63	1	fp		87	
FYL2X		3	60	≈200		90	1	fp		87	
FYL2XP1		11	134	≈242		≈220	1	fp		87	
Other											
FNOP		1	0	1	0	1	0		mov	87	
(F)WAIT		2	0	0	0	1	0		mov	87	
FNCLEX		4	4			96	1			87	
FNINIT		6	29			172				87	
FNSAVE		4	174	456		420	0,1			87	
FRSTOR		4	96	528		532				87	
FXSAVE		4	69	132		96				p3	i
FXRSTOR		4	94	208		208				p3	i

e) Not available on PMMX
f) The latency for `FLDCW` is 3 when the new value loaded is the same as the value of the control word before the preceding `FLDCW`, i.e. when alternating between the same two values. In all other cases, the latency and reciprocal throughput is 143. See page 123.
g) Latency and reciprocal throughput depend on the precision setting in the F.P. control word. Single precision: 23, double precision: 38, long double precision (default): 43.
h) Throughput of FP-MUL unit is reduced during the use of the FP-DIV unit.
i) Takes 6 uops more and 40-80 clocks more when XMM registers are disabled.

23.3 SIMD integer instructions

Instruction	Operands	Uops	Microcode	Latency	Additional latency	Reciprocal throughput	Port	Execution unit	Subunit	Backwards compatibility	Notes
Move instructions											
MOVD	r32, r64	2	0	5	1	1	0	fp		PMMX	
MOVD	r64, r32	2	0	2	0	2	1	mmx	alu	PMMX	
MOVD	r64,m32	1	0	≈ 8	0	1	2	mmx		PMMX	
MOVD	r32, r128	2	0	10	1	2	0	fp		PMMX	
MOVD	r128, r32	2	0	6	1	2	1	mmx	shift	PMMX	
MOVD	r128,m32	1	0	≈ 8	0	1	2	load		PMMX	
MOVD	m32, r	2	0	≈ 8		2	0,1			PMMX	
MOVQ	r64,r64	1	0	6	0	1	0	mov		PMMX	
MOVQ	r128,r128	1	0	2	1	2	1	mmx	shift	PMMX	
MOVQ	r,m64	1	0	≈ 8		1	2	load		PMMX	
MOVQ	m64,r	2	0	≈ 8		2	0	mov		PMMX	
MOVDQA	r128,r128	1	0	6	0	1	0	mov		p4	
MOVDQA	r128,m	1	0	≈ 8		1	2	load		p4	
MOVDQA	m,r128	2	0	≈ 8		2	0	mov		p4	
MOVDQU	r128,m	4	0			2	2	load		p4	k
MOVDQU	m,r128	4	6			2	0	mov		p4	k
MOVDQ2Q	r64,r128	3	0	8	1	2	0,1	mov-mmx		p4	
MOVQ2DQ	r128,r64	2	0	8	1	2	0,1	mov-mmx		p4	
MOVNTQ	m,r64	3	0			75	0	mov		p3	
MOVNTDQ	m,r128	2	0			18	0	mov		p4	
PACKSSWB/DW PACKUSWB	r64,r/m	1	0	2	1	1	1	mmx	shift	PMMX	a
PACKSSWB/DW PACKUSWB	r128,r/m	1	0	4	1	2	1	mmx	shift	PMMX	a
PUNPCKH/LBW/WD/DQ	r64,r/m	1	0	2	1	1	1	mmx	shift	PMMX	a
PUNPCKHBW/WD/DQ/QDQ	r128,r/m	1	0	4	1	2	1	mmx	shift	p4	a
PUNPCKLBW/WD/DQ/QDQ	r128,r/m	1	0	2	1	2	1	mmx	shift	p4	a
PSHUFD	r128,r128,i	1	0	4	1	2	1	mmx	shift	p4	
PSHUFL/HW	r128,r128,i	1	0	2	1	2	1	mmx	shift	p3	
PSHUFW	r64,r64,i	1	0	2	1	1	1	mmx	shift	p3	
MASKMOVQ	r64,r64	4	4			7	0	mov		p3	
MASKMOVDQU	r128,r128	4	6			10	0	mov		p4	
PMOVMSKB	r32,r	2	0	7	1	3	0,1	mmx-alu0		p3	
PEXTRW	r32,r64,i	3	0	8	1	2	1	mmx-int		p3	
PEXTRW	r32,r128,i	3	0	9	1	2	1	mmx-int		p4	
PINSW	r64,r32,i	2	0	3	1	2	1	int-mmx		p3	
PINSW	r128,r32,i	2	0	4	1	2	1	int-mmx		p4	
Arithmetic instructions											
PADDB/W/D PADD(U)SB/W	r,r/m	1	0	2	1	1,2	1	mmx	alu	PMMX	a,j
PSUBB/W/D PSUB(U)SB/W	r,r/m	1	0	2	1	1,2	1	mmx	alu	PMMX	a,j
PADDQ, PSUBQ	r64,r/m	1	0	2	1	1	1	mmx	alu	p4	a
PADDQ, PSUBQ	r128,r/m	1	0	4	1	2	1	mmx	alu	p4	a
PCMPEQB/W/D PCMPGTB/W/D	r,r/m	1	0	2	1	1,2	1	mmx	alu	PMMX	a,j
PMULLW PMULHW	r,r/m	1	0	6	1	1,2	1	fp	mul	PMMX	a,j
PMULHUW	r,r/m	1	0	6	1	1,2	1	fp	mul	p3	a,j

PMADDWD	r,r/m	1	0	6	1	1,2	1	fp	mul	PMMX	a,j
PMULUDQ	r,r/m	1	0	6	1	1,2	1	fp	mul	p4	a,j
PAVGB/W	r,r/m	1	0	2	1	1,2	1	mmx	alu	p3	a,j
PMIN/MAXUB	r,r/m	1	0	2	1	1,2	1	mmx	alu	p3	a,j
PMIN/MAXSW	r,r/m	1	0	2	1	1,2	1	mmx	alu	p3	a,j
PAVGB/W	r,r/m	1	0	2	1	1,2	1	mmx	alu	p3	a,j
PSADBW	r,r/m	1	0	4	1	1,2	1	mmx	alu	p3	a,j
Logic											
PAND, PANDN	r,r/m	1	0	2	1	1,2	1	mmx	alu	PMMX	a,j
POR, PXOR	r,r/m	1	0	2	1	1,2	1	mmx	alu	PMMX	a,j
PSL/RLW/D/Q, PSRAW/D	r,i/r/m	1	0	2	1	1,2	1	mmx	shift	PMMX	a,j
PSLLDQ, PSRLDQ	r128,i/r/m	1	0	4	1	2	1	mmx	shift	P4	a
Other											
EMMS		4	11	12		12	0			PMMX	

Notes:
a) Add 1 uop if source is a memory operand.
j) Reciprocal throughput is 1 for 64 bit operands, and 2 for 128 bit operands.
k) It may be advantageous to replace this instruction by two 64-bit moves

23.4 SIMD floating-point instructions

Instruction	Operands	Uops	Microcode	Latency	Additional latency	Reciprocal throughput	Port	Execution unit	Subunit	Backwards compatibility	Notes
Move instructions											
MOVAPS/D	r,r	1	0	6	0	1	0	mov		p3	
MOVAPS/D	r,m	1	0	≈ 7	0	1	2			p3	
MOVAPS/D	m,r	2	0	≈ 7		2	0			p3	
MOVUPS/D	r,r	1	0	6	0	1	0	mov		p3	
MOVUPS/D	r,m	4	0			2	2			p3	k
MOVUPS/D	m,r	4	6			8	0			p3	k
MOVSS	r,r	1	0	2	0	2	1	fp		p3	
MOVSD	r,r	1	0	2	1	2	1	fp		p3	
MOVSS, MOVSD	r,m	1	0	≈ 7	0	1	2			p3	
MOVSS, MOVSD	m,r	2	0			2	0			p3	
MOVHLPS	r,r	1	0	4	0	2	1	fp		p3	
MOVLHPS	r,r	1	0	2	0	2	1	fp		p3	
MOVHPS/D, MOVLPS/D	r,m	3	0			4	2			p3	
MOVHPS/D, MOVLPS/D	m,r	2	0			2	0			p3	
MOVNTPS/D	m,r	2	0			4	0			p3	
MOVMSKPS/D	r32,r	2	0	6	1	3	1	fp		p3	
SHUFPS/D	r,r/m,i	1	0	4	1	2	1	mmx	shift	p3	
UNPCKHPS/D	r,r/m	1	0	4	1	2	1	mmx	shift	p3	
UNPCKLPS/D	r,r/m	1	0	2	1	2	1	mmx	shift	p3	
Conversion											
CVTPS2PD	r,r/m	4	0	7	1	4	1	mmx	shift	p4	a
CVTPD2PS	r,r/m	2	0	10	1	2	1	fp-mmx		p4	a

CVTSD2SS	r,r/m	4	0	14	1	6	1	mmx	shift	p4	a
CVTSS2SD	r,r/m	4	0	10	1	6	1	mmx	shift	p4	a
CVTDQ2PS	r,r/m	1	0	4	1	2	1	fp		p4	a
CVTDQ2PD	r,r/m	3	0	9	1	4	1	mmx-fp		p4	a
CVT(T)PS2DQ	r,r/m	1	0	4	1	2	1	fp		p4	a
CVT(T)PD2DQ	r,r/m	2	0	9	1	2	1	fp-mmx		p4	a
CVTPI2PS	r128,r64/m	4	0	10	1	4	1	mmx		p3	a
CVTPI2PD	r128,r64/m	4	0	11	1	5	1	fp		p4	a
CVT(T)PS2PI	r64,r128/m	3	0	7	0	2	0,1	fp-mmx		p3	a
CVT(T)PD2PI	r64,r128/m	3	0	11	1	3	0,1	fp		p4	a
CVTSI2SS	r128,r32/m	3	0	10	1	3	1	fp-mmx		p3	a
CVTSI2SD	r128,r32/m	4	0	15	1	6	1	fp-mmx		p4	a
CVT(T)SD2SI	r32,r128/m	2	0	8	1	2.5	1	fp		p4	a
CVT(T)SS2SI	r32,r128/m	2	0	8	1	2.5	1	fp		p3	a
Arithmetic											
ADDPS/D ADDSS/D	r,r/m	1	0	4	1	2	1	fp	add	p3	a
SUBPS/D SUBSS/D	r,r/m	1	0	4	1	2	1	fp	add	p3	a
MULPS/D MULSS/D	r,r/m	1	0	6	1	2	1	fp	mul	p3	a
DIVSS	r,r/m	1	0	23	0	23	1	fp	div	p3	a,h
DIVPS	r,r/m	1	0	39	0	39	1	fp	div	p3	a,h
DIVSD	r,r/m	1	0	38	0	38	1	fp	div	p4	a,h
DIVPD	r,r/m	1	0	69	0	69	1	fp	div	p4	a,h
RCPPS PCPSS	r,r/m	2	0	4	1	4	1	mmx		p3	a
MAXPS/D MAXSS/D MINPS/D MINSS/D	r,r/m	1	0	4	1	2	1	fp	add	p3	a
CMPccPS/D CMPccSS/D	r,r/m	1	0	4	1	2	1	fp	add	p3	a
COMISS/D UCOMISS/D	r,r/m	2	0	6	1	3	1	fp		p3	a
Logic											
ANDPS/D ANDNPS/D ORPS/D XORPS/D	r,r/m	1	0	2	1	2	1	mmx	alu	p3	a
Math											
SQRTSS	r,r/m	1	0	23	0	23	1	fp	div	p3	a,h
SQRTPS	r,r/m	1	0	39	0	39	1	fp	div	p3	a,h
SQRTSD	r,r/m	1	0	38	0	38	1	fp	div	p4	a,h
SQRTPD	r,r/m	1	0	69	0	69	1	fp	div	p4	a,h
RSQRTSS	r,r/m	2	0	4	1	3	1	mmx		p3	a
RSQRTPS	r,r/m	2	0	4	1	4	1	mmx		p3	a
Other											
LDMXCSR	m	4	8	98		100	1			p3	
STMXCSR	m	4	4			6	1			p3	

Notes:
a) Add 1 uop if source is a memory operand.
h) Throughput of FP-MUL unit is reduced during the use of the FP-DIV unit.
k) It may be advantageous to replace this instruction by two 64-bit moves.

24 Comparison of the different microprocessors

The following table summarizes some important differences between the microprocessors in the Pentium family:

	P1	**PMMX**	**PPro**	**P2**	**P3**	**P4**
code cache, kb	8	16	8	16	16	≈ 60
code cache associativity, ways	2	4	4	4	4	4
data cache, kb	8	16	8	16	16	8
data cache associativity, ways	2	4	2	4	4	4
data cache line size	32	32	32	32	32	64
built-in level 2 cache, kb	0	0	256 *)	512 *)	512 *)	512 *)
level 2 cache associativity, ways	0	0	4	4	8	8
level 2 cache bus size, bits	0	0	64	64	256	256
MMX instructions	no	yes	no	yes	yes	yes
XMM instructions	no	no	no	no	yes	yes
conditional move instructions	no	no	yes	yes	yes	yes
out of order execution	no	no	yes	yes	yes	yes
branch prediction	poor	good	good	good	good	good
branch target buffer entries	256	256	512	512	512	512?
return stack buffer size	0	4	16	16	16	16
branch misprediction penalty	3-4	4-5	10-20	10-20	10-20	≥ 24
partial register stall	0	0	5	5	5	0
FMUL latency	3	3	5	5	5	6-7
FMUL reciprocal throughput	2	2	2	2	2	1
IMUL latency	9	9	4	4	4	14
IMUL reciprocal throughput	9	9	1	1	1	5-10

*) Celeron: 0-128, Xeon: 512 or more, many other variants available. On some versions the level 2 cache runs at half speed.