2.
Optimizing subroutines in assembly language
An optimization guide for x86 platforms

By Agner Fog

Contents

1 Introduction

This is the second in a series of five manuals:

1. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms.

2. Optimizing subroutines in assembly language: An optimization guide for x86 platforms.

3. The microarchitecture of Intel and AMD CPU's: An optimization guide for assembly programmers and compiler writers.

4. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdown for Intel and AMD CPU's.

5. Calling conventions for different C++ compilers and operating systems.

The latest version of these manuals is always available from www.agner.org/optimize.

The present manual explains how to combine assembly code with a high level programming language and how to optimize CPU-intensive code for speed by using assembly code.

This manual is intended for advanced assembly programmers and compiler makers. It is assumed that the reader has a good understanding of assembly language and some experience with assembly coding. Beginners are advised to seek information elsewhere and get some programming experience before trying the optimization techniques described here. I can recommend the various introductions, tutorials, discussion forums and newsgroups on the internet (see links from www.agner.org/optimize) and the book "Introduction to 80x86 Assembly Language and Computer Architecture" by R. C. Detmer, 2. ed. 2006.

The present manual covers all platforms that use the x86 instruction set. This instruction set is used by most microprocessors from Intel and AMD. Operating systems that can use this instruction set include DOS (16 and 32 bits), Windows (16, 32 and 64 bits), Linux (32 and 64 bits), FreeBSD/Open BSD (32 and 64 bits), and Intel-based Mac OS (32 bits).

Optimization techniques that are not specific to assembly language are discussed in manual 1: "Optimizing software in C++". Details that are specific to a particular microprocessor are covered by manual 3: "The microarchitecture of Intel and AMD CPU's". Tables of instruction timings etc. are provided in manual 4: "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdown for Intel and AMD CPU's". Details about calling conventions for different operating systems and compilers are covered in manual 5: "Calling conventions for different C++ compilers and operating systems".

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! Please don't send your programming questions to me. Such mails will not be answered. 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 Reasons for using assembly code

Assembly coding is not used as much today as previously. However, there are still reasons for learning and using assembly code. The main reasons are:

1. Educational reason. It is important to know how microprocessors and compilers work at the instruction level in order to be able to predict which coding techniques are most efficient, to understand how various constructs in high level languages work, and to track hard-to-find errors.

2. Debugging and verifying. Looking at compiler-generated assembly code or the disassembly window in a debugger is useful for finding errors and for checking how well a compiler optimizes a particular piece of code.

3. Making compilers. Understanding assembly coding techniques is necessary for making compilers, debuggers and other development tools.

4. Embedded systems. Small embedded systems have fewer resources than PC's and mainframes. Assembly programming can be necessary for optimizing code for speed or size in small embedded systems.

5. Hardware drivers and system code. Accessing hardware, system control registers etc. may sometimes be difficult or impossible with high level code.

6. Accessing instructions that are not accessible from high level language. Certain assembly instructions have no high-level language equivalent.

7. Self-modifying code. Self-modifying code is generally not profitable because it interferes with efficient code caching. It may, however, be advantageous for example to include a small compiler in math programs where a user-defined function has to be calculated many times.

8. Optimizing code for size. Storage space and memory is so cheap nowadays that it is not worth the effort to use assembly language for reducing code size. However, cache size is still such a critical resource that it may be useful in some cases to optimize a critical piece of code for size in order to make it fit into the code cache.

9. Optimizing code for speed. Modern C++ compilers generally optimize code quite well in most cases. But there are still many cases where compilers perform poorly and where dramatic increases in speed can be achieved by careful assembly programming.

10. Function libraries. The total benefit of optimizing code is higher in function libraries that are used by many programmers.

11. Making function libraries compatible with multiple compilers and operating systems. It is possible to make library functions with multiple entries that are compatible with different compilers and different operating systems. This requires assembly programming.

The main focus in this manual is on optimizing code for speed, though some of the other topics are also discussed.

1.2 Reasons for not using assembly code

There are so many disadvantages and problems involved in assembly programming that it is advisable to consider the alternatives before deciding to use assembly code for a particular task. The most important reasons for *not* using assembly programming are:

1. Development time. Writing code in assembly language takes much longer time than in a high level language.

2. Reliability and security. It is easy to make errors in assembly code. The assembler is not checking if the calling conventions and register save conventions are obeyed. Nobody is checking for you if the number of PUSH and POP instructions is the same in all possible branches and paths. There are so many possibilities for hidden errors in assembly code that it affects the reliability and security of the project unless you have a very systematic approach to testing and verifying.

3. Debugging and verifying. Assembly code is more difficult to debug and verify because there are more possibilities for errors than in high level code.

4. Maintainability. Assembly code is more difficult to modify and maintain because the language allows unstructured spaghetti code and all kinds of dirty tricks that are difficult for others to understand. Thorough documentation and a consistent programming style is needed.

5. Portability. Assembly code is very platform-specific. Porting to a different platform is difficult.

6. System code can use intrinsic functions instead of assembly. The best modern C++ compilers have intrinsic functions for accessing system control registers and other system instructions. Assembly code is no longer needed for device drivers and other system code when intrinsic functions are available.

7. Application code can use intrinsic functions and vector classes instead of assembly. The best modern C++ compilers have intrinsic functions for vector operations and other special instructions that previously required assembly programming. It is no longer necessary to use old fashioned assembly code to take advantage of the XMM vector instructions. See page 26.

1.3 Microprocessors covered by this manual

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

Microprocessor name	Abbreviation
Intel Pentium (without name suffix)	P1
Intel Pentium MMX	PMMX
Intel Pentium Pro	PPro
Intel Pentium II	P2
Intel Pentium III	P3
Intel Pentium 4 (NetBurst)	P4
Intel Pentium 4 with EM64T, Pentium D, etc.	P4E
Intel Pentium M, Core Solo, Core Duo	PM
Intel Core 2	Core2
AMD Athlon 64	AMD64

See manual 3: "The microarchitecture of Intel and AMD CPU's" for details.

1.4 Operating systems covered by this manual

The following operating systems can use x86 family microprocessors:

16 bit: DOS, Windows 3.x.

32 bit: Windows, Linux, FreeBSD, OpenBSD, NetBSD, Intel-based Mac OS X.

64 bit: Windows, Linux, FreeBSD, OpenBSD, NetBSD.

All the UNIX-like operating systems (Linux, BSD, Mac OS) use the same calling conventions, with very few exceptions. Everything that is said in this manual about Linux also applies to other UNIX-like systems, possibly including systems not mentioned here.

2 Before you start

2.1 Things to decide before you start programming

Before you start to program in assembly, you have to think about why you want to use assembly language, which part of your program you need to make in assembly, and what programming method to use. If you haven't made your development strategy clear, then you will soon find yourself wasting time optimizing the wrong part of the program, doing things in assembly that could have been done in C++, attempting to optimize things that cannot be optimized further, making spaghetti code that is difficult to maintain, and making code that is full or errors and difficult to debug.

Here is a checklist of things to consider before you start programming:

- Never make the whole program in assembly. That is a waste of time. Assembly code should be used only where speed is critical and where a significant improvement in speed can be obtained. Most of the program should be made in C++. C and C++ are the programming languages that are most easily combined with assembly code.

- If the purpose of using assembly is to make system code or use special instructions that are not available in standard C++ then you should isolate the part of the program that needs these instructions in a separate function or class with a well defined functionality.

- If the purpose of using assembly is to optimize for speed then you have to identify the part of the program that consumes most CPU time, possibly with the use of a profiler. Check if the bottleneck is file access, memory access, CPU instructions, or something else, as described in manual 1: "Optimizing software in C++". Isolate the critical part of the program into a function or class with a well-defined functionality.

- If the purpose of using assembly is to make a function library then you should clearly define the functionality of the library. Decide whether to make a function library or a class library. Decide whether to use static linking (`.lib` in Windows, `.a` in Linux) or dynamic linking (`.dll` in Windows, `.so` in Linux). Static linking is usually more efficient, but dynamic linking may be necessary if the library is called from languages such as C# and Visual Basic. You may possibly make both a static and a dynamic link version of your library.

- If the purpose of using assembly is to optimize an embedded application for size or speed then find a development tool that supports both C/C++ and assembly and make as much as possible in C or C++.

- Decide if the code is reusable or application-specific. Spending time on careful optimization is more justified if the code is reusable. A reusable code is most appropriately implemented as a function library or class library.

- Decide if the code should support multithreading. A multithreading application can take advantage of microprocessors with multiple kernels. Any data that must be

preserved from one function call to the next on a per-thread basis should be stored in a C++ class or a per-thread buffer supplied by the calling program.

- Decide if portability is important for your application. Should the application work in both Windows, Linux and Intel-based Mac OS? Should it work in both 32 bit and 64 bit mode? Should it work on non-x86 platforms? This is important for the choice of compiler, assembler and programming method.

- Decide if your application should work on old microprocessors. If so, then you may make one version for microprocessors with, for example, the SSE2 instruction set, and another version which is compatible with old microprocessors. You may even make several versions, each optimized for a particular CPU. It is recommended to make automatic CPU dispatching (see page 109).

- There are three assembly programming methods to choose between: (1) Use intrinsic functions in a C++ compiler. (2) Use inline assembly in a C++ compiler. (3) Use an assembler. These three methods and their relative advantages and disadvantages are described in chapter 5, 6 and 7 respectively (page 26, 28 and 37 respectively).

- If you are using an assembler then you have to choose between different syntax dialects. It is preferred to use an assembler that is compatible with the assembly code that your C++ compiler can generate.

- Make your code in C++ first and optimize it as much as you can, using the methods described in manual 1: "Optimizing software in C++". Make the compiler translate the code to assembly. Look at the compiler-generated code and see if there are possibilities for improvement in the code.

- 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 calling convention and appropriate comments. Decide on a consistent strategy for code comments and documentation.

- Save the compiler, assembler and all other development tools together with the source code and project files for later maintenance. Compatible tools may not be available in a few years when updates and modifications in the code are needed.

2.2 Make a test strategy

Assembly code is error prone, difficult to debug, difficult to make in a clearly structured way, difficult to read, and difficult to maintain, as I have already mentioned. A consistent test strategy can ameliorate some of these problems and save you a lot of time.

My recommendation is to make the assembly code as an isolated module, function, class or library with a well-defined interface to the calling program. Make it all in C++ first. Then make a test program which can test all aspects of the code you want to optimize. It is easier and safer to use a test program than to test the module in the final application.

The test program has two purposes. The first purpose is to verify that the assembly code works correctly in all situations. And the second purpose is to test the speed of the assembly code without invoking the user interface, file access and other parts of the final application program that may make the speed measurements less accurate and less reproducible.

You should use the test program repeatedly after each step in the development process and after each modification of the code.

Make sure the test program works correctly. It is quite common to spend a lot of time looking for an error in the code under test when in fact the error is in the test program.

There are different test methods that can be used for verifying that the code works correctly. A white box test supplies a carefully chosen series of different sets of input data to make sure that all branches, paths and special cases in the code are tested. A black box test supplies a series of random input data and verifies that the output is correct. A very long series of random data from a good random number generator can sometimes find rarely occurring errors that the white box test hasn't found.

The test program may compare the output of the assembly code with the output of a C++ implementation to verify that it is correct. The test should cover all boundary cases and preferably also illegal input data to see if the code generates the correct error responses.

The speed test should supply a realistic set of input data. A significant part of the CPU time may be spent on branch mispredictions in code that contains a lot of branches. The amount of branch mispredictions depends on the degree of randomness in the input data. You may experiment with the degree of randomness in the input data to see how much it influences the computation time, and then decide on a realistic degree of randomness that matches a typical real application.

An automatic test program that supplies a long stream of test data will typically find more errors and find them much faster than testing the code in the final application. A good test program will find most errors, but you cannot be sure that it finds all errors. It is possible that some errors show up only in combination with the final application.

3 The basics of assembly coding

3.1 Assembly language syntaxes

Assembly programmers are in the unfortunate situation that there is no universally agreed syntax for x86 assembly. The different syntaxes that are used can be summarized as follows:

1. AT&T syntax. This syntax is used by the Gnu compiler and Gnu assembler. The Gnu compiler generates an output in this format which is subsequently fed into the Gnu assembler. This syntax is useful as a machine-generated intermediate, but it is very inconvenient for human-generated assembly code. The AT&T syntax has source operand before destination operand, and various prefixes like `%` and `$` for specifying operand types. Newer versions of Gnu compilers and assemblers have an option for using Intel syntax instead.

2. Intel syntax. This is an incomplete syntax which defines only the syntax for instruction codes, not for directives, functions, macros, etc. Various dialects of Intel syntax is part of the syntax for most assemblers. The destination operand comes before the source operand.

3. MASM syntax. This syntax is defined by Microsoft's macro assembler and is also used by Borland and Watcom assemblers. Minor changes in the syntax were made in MASM version 6 in order to fix various quirks and inconsistencies, but an option for backwards compatibility with version 5 is included. The lineage can be traced all the way back to the first IBM PC's which used assemblers produced by IBM, Intel

and Microsoft.

4. NASM syntax. Supported by the NASM and YASM assemblers. This syntax is similar to MASM but not fully compatible. Directives, macros and operators are different from MASM and allegedly more consistent.

5. High Level Assembler (HLA). Includes many high-level language constructs. Has source before destination in most cases.

6. Other open source assemblers. There are many open source assemblers available with each their syntax. NASM is the most popular of these.

7. Gnu inline assembly. Inline assembly in C and C++ programs compiled on a Gnu compatible compiler can use a special syntax for connection between the C++ part and the assembly part. This syntax is powerful but very complicated to use.

8. Microsoft inline assembly. Inline assembly in C and C++ programs compiled on a Microsoft compatible compiler can access C++ variables, functions and labels simply by inserting their names in the assembly code. This is easy, but does not currently support C++ register variables. See page 28.

9. C++ intrinsic functions. These are functions that can be used in C or C++ code. Each intrinsic function corresponds to one or a few assembly instructions. Supported by Microsoft, Intel and Gnu compilers. See page 26.

It is preferred to use an assembler that is fully compatible with the C++ compiler you are using. This allows you to use the compiler for translating C++ to assembly, modify the assembly code as you wish, and then assemble it.

The syntax I am using in this manual is MASM syntax. This is the syntax that is used in most textbooks, in manuals from Intel, AMD and Microsoft, and in the output from most C++ compilers. MASM is the closest we get to a de facto standard for x86 assembly language. The MASM syntax is described in Microsoft Macro Assembler Reference at msdn.microsoft.com.

Evidently, assemblers are not considered a profitable product by commercial software companies. The commercial assemblers are not maintained as well as one may wish. The best commercial assembler is MASM, but some important features are missing and several issues are never dealt with. Especially the 64-bit version has serious shortcomings.

Open source assemblers such as Gas and NASM are better in this respect, but unfortunately not fully compatible with the MASM output generated by many C++ compilers. An open source assembler that is fully compatible with the output of common C++ compilers (Microsoft, Borland, Intel) stands high on my wish list.

The MASM assembler is available for free as part of Microsoft's Platform Software Development Kit (PSDK). MASM is designed to run under 32-bit Windows. See page 50 for how to use MASM under Linux and similar operating systems.

See www.agner.org/optimize for links to various available assemblers, syntax manuals, coding manuals and discussion forums.

3.2 Register set and basic instructions

Registers in 16 bit mode

General purpose and integer registers

Full register bit 0 - 15	Partial register bit 8 - 15	Partial register bit 0 - 7
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL
SI		
DI		
BP		
SP		
Flags		
IP		

The 32-bit registers are also available in 16-bit mode if supported by the microprocessor and operating system. The high word of ESP should not be used because it is not saved during interrupts.

Floating point registers

Full register bit 0 - 79
ST(0)
ST(1)
ST(2)
ST(3)
ST(4)
ST(5)
ST(6)
ST(7)

MMX registers may be available if supported by the microprocessor. XMM registers may be available if supported by microprocessor and operating system.

Segment registers

Full register bit 0 - 15
CS
DS
ES
SS

Register FS and GS may be available.

Registers in 32 bit mode

General purpose and integer registers

Full register bit 0 - 31	Partial register bit 0 - 15	Partial register bit 8 - 15	Partial register bit 0 - 7
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI		
EDI	DI		
EBP	BP		
ESP	SP		
EFlags	Flags		
EIP	IP		

Floating point and 64-bit vector registers

Full register bit 0 - 79	Partial register bit 0 - 63
ST(0)	MM0
ST(1)	MM1
ST(2)	MM2
ST(3)	MM3
ST(4)	MM4
ST(5)	MM5
ST(6)	MM6
ST(7)	MM7

The MMX registers are only available if supported by the microprocessor. The ST and MMX registers cannot be used in the same part of the code. A section of code using MMX registers must be separated from any subsequent section using ST registers by executing an EMMS instruction.

128-bit integer and floating point vector registers

Full register bit 0 - 127
XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7

The XMM registers are only available if supported both by the microprocessor and the operating system. Scalar floating point instructions use only 32 or 64 bits of the XMM registers for single or double precision, respectively.

Segment registers

Full register bit 0 - 15
CS
DS
ES
FS
GS
SS

Registers in 64 bit mode

General purpose and integer registers

Full register bit 0 - 63	Partial register bit 0 - 31	Partial register bit 0 - 15	Partial register bit 8 - 15	Partial register bit 0 - 7
RAX	EAX	AX	AH	AL
RBX	EBX	BX	BH	BL
RCX	ECX	CX	CH	CL
RDX	EDX	DX	DH	DL
RSI	ESI	SI		SIL
RDI	EDI	DI		DIL
RBP	EBP	BP		BPL
RSP	ESP	SP		SPL
R8	R8D	R8W		R8B
R9	R9D	R9W		R9B

R10	R10D	R10W		R10B
R11	R11D	R11W		R11B
R12	R12D	R12W		R12B
R13	R13D	R13W		R13B
R14	R14D	R14W		R14B
R15	R15D	R15W		R15B
RFlags		Flags		
RIP				

The high 8-bit registers AH, BH, CH, DH can only be used in instructions that have no REX prefix.

Note that modifying a 32-bit partial register will set the rest of the register (bit 32-63) to zero, but modifying an 8-bit or 16-bit partial register does not affect the rest of the register. This can be illustrated by the following sequence:

```
; Example 3.1. 8, 16, 32 and 64 bit registers
mov  rax, 1111111111111111H  ; rax = 1111111111111111H
mov  eax, 22222222H          ; rax = 0000000022222222H
mov   ax, 3333H              ; rax = 0000000022223333H
mov   al, 44H                ; rax = 0000000022223344H
```

There is a good reason for this inconsistency. Setting the unused part of a register to zero is more efficient than leaving it unchanged because this removes a false dependence on previous values. But the principle of resetting the unused part of a register cannot be extended to 16 bit and 8 bit partial registers because this would break the backwards compatibility with 32-bit and 16-bit modes.

The only instruction that can have a 64-bit immediate data operand is MOV. Other integer instructions can only have a 32-bit sign extended operand. Examples:

```
; Example 3.2. Immediate operands, full and sign extended
mov  rax, 1111111111111111H  ; Full 64 bit immediate operand
mov  rax, -1                 ; 32 bit sign-extended operand
mov  eax, 0ffffffffH         ; 32 bit zero-extended operand
add  rax, 1                  ; 8  bit sign-extended operand
add  rax, 100H               ; 32 bit sign-extended operand
add  eax, 100H               ; 32 bit operand. result is zero-extended
mov  rbx, 100000000H         ; 64 bit immediate operand
add  rax, rbx                ; Use an extra register if big operand
```

It is not possible to use a 16-bit sign-extended operand. If you need to add an immediate value to a 64 bit register then it is necessary to first move the value into another register if the value is too big for fitting into a 32 bit sign-extended operand.

Floating point and 64-bit vector registers

Full register bit 0 - 79	Partial register bit 0 - 63
ST(0)	MM0
ST(1)	MM1
ST(2)	MM2
ST(3)	MM3
ST(4)	MM4
ST(5)	MM5
ST(6)	MM6
ST(7)	MM7

The ST and MMX registers cannot be used in the same part of the code. A section of code using MMX registers must be separated from any subsequent section using ST registers by

executing an EMMS instruction. The ST and MMX registers cannot be used in device drivers for 64-bit Windows.

128-bit integer and floating point vector registers

Full register bit 0 - 127
XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7
XMM8
XMM9
XMM10
XMM11
XMM12
XMM13
XMM14
XMM15

Scalar floating point instructions use only 32 or 64 bits of the XMM registers for single or double precision, respectively.

Segment registers

Full register bit 0 - 15
CS
FS
GS

Segment registers are only used for special purposes.

3.3 Addressing modes

Addressing in 16-bit mode

16-bit code uses a segmented memory model. A memory operand can have any of these components:

- A segment specification. This can be any segment register or a segment or group name associated with a segment register. (The default segment is DS, except if BP is used as base register). The segment can be implied from a label defined inside a segment.
- A label defining a relocatable offset. The offset relative to the start of the segment is calculated by the linker.
- An immediate offset. This is a constant. If there is also a relocatable offset then the values are added.
- A base register. This can only be BX or BP. Must appear inside [].
- An index register. This can only be SI or DI. Must appear inside []. There can be no scale factor.

A memory operand can have all of these components. An operand containing only an immediate offset is not interpreted as a memory operand, even if it has a []. Examples:

```
; Example 3.3. Memory operands in 16-bit mode
MOV  AX, DS:[100H]   ; Address has segment and immediate offset
ADD  AX, MEM[SI]+4   ; Has relocatable offset and index and immediate
```

Data structures bigger than 64 kb are handled in the following ways. In real mode and virtual mode (DOS): Adding 1 to the segment register corresponds to adding 10H to the offset. In protected mode (Windows 3.x): Adding 8 to the segment register corresponds to adding 10000H to the offset.

Addressing in 32-bit mode

32-bit code uses a flat memory model in most cases. Segmentation is possible but only used for special purposes (e.g. thread environment block in `FS`).

A memory operand can have any of these components:

- A segment specification. Not used in flat mode.
- A label defining a relocatable offset. The offset relative to the `FLAT` segment group is calculated by the linker.
- An immediate offset. This is a constant. If there is also a relocatable offset then the values are added.
- A base register. This can be any 32 bit register. Must appear inside `[]`.
- An index register. This can be any 32 bit register except `ESP`. Must appear inside `[]`.
- A scale factor applied to the index register. Allowed values are 1, 2, 4, 8.

A memory operand can have all of these components. Examples:

```
; Example 3.4. Memory operands in 32-bit mode
mov  eax, fs:[100H]     ; Address has segment and immediate offset
add  eax, mem[esi]      ; Has relocatable offset and index
add  eax, [esp+ecx*4+8] ; Base, index, scale and immediate offset
```

Addressing in 64-bit mode

64-bit code always uses a flat memory model. Segmentation is impossible except for `FS` and `GS` which are used for special purposes only (thread environment block, etc.).

There are several different addressing modes in 64-bit mode: 32-bit absolute, 64-bit absolute, RIP-relative, and relative to a base register.

32-bit absolute addresses in 64 bit mode

This works the same way as in 32-bit code. The 32-bit constant is sign-extended to 64 bits. This addressing mode works only if all addresses are in the interval between -2^{31} and 2^{31}.

The following examples address static arrays. The C++ code for this example is:

```
// Example 3.5a. Static arrays in 64 bit mode
// C++ code:
static int a[100], b[100];
for (int i = 0; i < 100; i++) {
   b[i] = -a[i];
}
```

The solution with 32-bit absolute addresses is as follows:

```
; Example 3.5b.
; Use 32-bit absolute addresses (64 bit Windows or Linux)
; Assumes that image base < 80000000H
.data
A    DD 100 dup (?)     ; Define static array A
B    DD 100 dup (?)     ; Define static array B

.code
xor  ecx, ecx           ; i = 0

TOPOFLOOP:              ; Top of loop
mov  eax, A[rcx*4]      ; 32-bit address + scaled index
neg  eax
mov  B[rcx*4], eax      ; 32-bit address + scaled index
add  ecx, 1
cmp  ecx, 100           ; i < 100
jb   TOPOFLOOP          ; Loop
```

The assembler will generate a 32-bit relocatable address for A and B in example 3.5b because it cannot combine a RIP-relative address with an index register.

This method is used by the Gnu compiler in 64-bit Linux to access static arrays. It is not used by any compiler for 64-bit Windows, I have seen, but it works in Windows as well if the address is less than 2^{31}. The image base is typically 2^{22} for application programs and between 2^{28} and 2^{29} for DLL's, so this method will work in most cases, but possibly not all.

Segmentation with CS, DS, ES and SS is not possible. Segments with FS and GS are possible and are used for thread environment blocks etc.

64-bit absolute addresses

This uses a 64-bit absolute virtual address. The address cannot contain any segment register, base or index register. 64-bit absolute addresses can only be used with the MOV instruction, and only with AL, AX, EAX or RAX as source or destination. This addressing mode is not supported by the MASM assembler.

RIP-relative addresses

This is the preferred addressing mode for static data. The address contains a 32-bit sign-extended offset relative to the instruction pointer. The address cannot contain any segment register or index register, and no base register other than RIP, which is implicit. Example:

```
; Example 3.6. RIP-relative memory operand in 64-bit mode
mov eax, mem
```

A static array cannot be addressed directly with this mode because there can be no index register on a RIP-relative address. This problem is solved either by loading the address of the array into a base register or by loading the image base into a base register.

The following solution loads the image base into register RBX by using a LEA instruction with a RIP-relative address:

```
; Example 3.5c.
; Address relative to image base (64 bit Windows)
.data
A    DD 100 dup (?)
B    DD 100 dup (?)
extern __ImageBase:byte

.code
lea  rbx, __ImageBase   ; Use RIP-relative address of image base
```

```
xor  ecx, ecx           ; i = 0

TOPOFLOOP:              ; Top of loop
; imagerel(A) = address of A relative to image base:
mov  eax, (imagerel A)[rbx + rcx*4]
neg  eax
mov  (imagerel B)[rbx + rcx*4], eax
add  ecx, 1
cmp  ecx, 100
jb   TOPOFLOOP
```

Unfortunately, the current version of the Microsoft assembler (version 8.00) has a bug with the `imagerel` operator. I don't know how to get the image base in Linux.

Another solution loads the address of array `A` into register `RBX` by using a `LEA` instruction with a `RIP`-relative address. The address of `B` is calculated relative to `A`.

```
; Example 3.5d.
; Load address of array into base register (64-bit Windows or Linux)
.data
A DD 100 dup (?)
B DD 100 dup (?)

.code
lea  rbx, A           ; Use RIP-relative address of A
xor  ecx, ecx         ; i = 0

TOPOFLOOP:            ; Top of loop
mov  eax, [rbx + 4*rcx] ; A[i]
neg  eax
mov  (B-A)[rbx + 4*rcx], eax ; Use offset of B relative to A
add  ecx, 1
cmp  ecx, 100
jb   TOPOFLOOP
```

Note that we can use a 32-bit instruction for incrementing the index (`ADD ECX,1`), even though we are using the 64-bit register for index (`RCX`). This works because we are sure that the index is less than 2^{32}.

Addressing relative to 64-bit base register

A memory operand in this mode can have any of these components:

- An immediate offset. This is a constant offset relative to the base register.
- A base register. This can be any 64 bit integer register. Must appear inside `[]`.
- An index register. This can be any 64 bit integer register except `RSP`. Must appear inside `[]`.
- A scale factor applied to the index register. The only possible values are 1, 2, 4, 8.

A base register is always needed for this addressing mode. The other components are optional. Examples:

```
; Example 3.7. Base register addressing in 64 bit mode
mov  eax, [rsi]
add  eax, [rsp + 4*rcx + 8]
```

3.4 Instruction code format

The format for instruction codes is described in detail in manuals from Intel and AMD. The basic principles of instruction encoding are explained here because of its relevance to microprocessor performance. In general, you can rely on the assembler for generating the smallest possible encoding of an instruction.

Each instruction can consist of the following elements, in the order mentioned:

1. Prefixes (0-5 bytes)
   These are prefixes that modify the meaning of the opcode that follows. There are several different kinds of prefixes as described in table 3.1 below.

2. Opcode (1-3 bytes)
   This is the instruction code. The first byte is 0Fh if there are two or three bytes. The second byte is 38h - 3Ah if there are three bytes.

3. mod-reg-r/m byte (0-1 byte)
   This byte specifies the operands. It consists of three fields. The mod field is two bits specifying the addressing mode, the reg field is three bits specifying a register for the first operand (most often the destination operand), the r/m field is three bits specifying the second operand (most often the source operand), which can be a register or a memory operand. The reg field can be part of the opcode if there is only one operand.

4. SIB byte (0-1 byte)
   This byte is used for memory operands with complex addressing modes, and only if there is a mod-reg-r/m byte. It has two bits for a scale factor, three bits specifying a scaled index register, and three bits specifying a base pointer register. A SIB byte is needed in the following cases:
   a. If a memory operand has two pointer or index registers,
   b. If a memory operand has a scaled index register,
   c. If a memory operand has the stack pointer (`ESP` or `RSP`) as base pointer,
   d. If a memory operand in 64-bit mode uses a 32-bit sign-extended direct memory address rather than a RIP-relative address.
   A SIB byte cannot be used in 16-bit addressing mode.

5. Displacement (0, 1, 2, 4 or 8 bytes)
   This is part of the address of a memory operand. It is added to the value of the pointer registers (base or index or both), if any.
   A 1-byte sign-extended displacement is possible in all addressing modes if a pointer register is specified.
   A 2-byte displacement is possible only in 16-bit addressing mode.
   A 4-byte displacement is possible in 32-bit addressing mode.
   A 4-byte sign-extended displacement is possible in 64-bit addressing mode. If there are any pointer registers specified then the displacement is added to these. If there is no pointer register specified and no SIB byte then the displacement is added to RIP. If there is a SIB byte and no pointer register then the sign-extended value is an absolute direct address.
   An 8-byte absolute direct address is possible in 64-bit addressing mode for a few `MOV` instructions that have no mod-reg-r/m byte.

6. Immediate operand (0, 1, 2, 4 or 8 bytes)
   This is a data constant which in most cases is a source operand for the operation.
   A 1-byte sign-extended immediate operand is possible in all modes for all instructions that can have immediate operands, except `MOV`, `CALL` and `RET`.
   A 2-byte immediate operand is possible for instructions with 16-bit operand size.
   A 4-byte immediate operand is possible for instructions with 32-bit operand size.

A 4-byte sign-extended immediate operand is possible for instructions with 64-bit operand size.
An 8-byte immediate operand is possible only for moves into a 64-bit register.

3.5 Instruction prefixes

The following table summarizes the use of instruction prefixes.

prefix for:	16 bit mode	32 bit mode	64 bit mode
8 bit operand size	none	none	none
16 bit operand size	none	66h	66h
32 bit operand size	66h	none	none
64 bit operand size	n.a.	n.a.	REX.W (48h)
packed integers in mmx register	none	none	none
packed integers in xmm register	66h	66h	66h
packed single-precision floats in xmm register	none	none	none
packed double-precision floats in xmm register	66h	66h	66h
scalar single-precision floats in xmm register	F3h	F3h	F3h
scalar double-precision floats in xmm register	F2h	F2h	F2h
16 bit address size	none	67h	n.a.
32 bit address size	67h	none	67h
64 bit address size	n.a.	n.a.	none
CS segment	2Eh	2Eh	n.a.
DS segment	3Eh	3Eh	n.a.
ES segment	26h	26h	n.a.
SS segment	36h	36h	n.a.
FS segment	64h	64h	64h
GS segment	65h	65h	65h
REP or REPE string operation	F3h	F3h	F3h
REPNE string operation	F2h	F2h	F2h
Locked memory operand	F0h	F0h	F0h
Register R8 - R15, XMM8 - XMM15 in reg field	n.a.	n.a.	REX.R (44h)
Register R8 - R15, XMM8 - XMM15 in r/m field	n.a.	n.a.	REX.B (41h)
Register R8 - R15 in SIB.base field	n.a.	n.a.	REX.B (41h)
Register R8 - R15 in SIB.index field	n.a.	n.a.	REX.X (42h)
Register SIL, DIL, BPL, SPL	n.a.	n.a.	REX (40h)
Predict branch taken first time	3Eh	3Eh	3Eh
Predict branch not taken first time	2Eh	2Eh	2Eh

Table 3.1. Instruction prefixes

Segment prefixes are rarely needed in a flat memory model. The DS segment prefix is only needed if a memory operand has base register BP, EBP or ESP and the DS segment is desired rather than SS.

The lock prefix is only allowed on certain instructions that read, modify and write a memory operand.

The branch prediction prefixes have little effect and are rarely needed.

There can be no more than one REX prefix. If more than one REX prefix is needed then the values are OR'ed into a single byte with a value in the range 40h to 4Fh. These prefixes are available only in 64-bit mode. The bytes 40h to 4Fh are instruction opcodes in 16-bit and 32-bit mode. These instructions (INC r and DEC r) are coded differently in 64-bit mode.

The prefixes can be inserted in any order, except for the REX prefix which must come after any other prefixes.

Meaningless, redundant or misplaced prefixes are ignored, except for the LOCK prefix. But prefixes that have no effect in a particular context may have an effect in future processors.

Unnecessary prefixes may be used instead of `NOP`'s for aligning code, but an excessive number of prefixes can slow down instruction decoding.

There can be any number of prefixes as long as the total instruction length does not exceed 15 bytes. For example, a `MOV EAX,EBX` with ten `ES` segment prefixes will still work correctly, but it takes a long time to decode.

4 ABI standards

ABI stands for Application Binary Interface. An ABI is a standard for how functions are called, how parameters and return values are transferred, and which registers a function is allowed to change. It is important to obey the appropriate ABI standard when combining assembly with high level language. The details of calling conventions etc. are covered in manual 5: "Calling conventions for different C++ compilers and operating systems". The most important rules are summarized here for your convenience.

4.1 Register usage

Table 2. Register usage

	16 bit DOS, Windows	32 bit Windows, Linux, MacOS	64 bit Windows	64 bit Linux
Registers that can be used freely	AX, BX, CX, DX, ES, ST(0)-ST(7)	EAX, ECX, EDX, ST(0)-ST(7), XMM0-XMM7	RAX, RCX, RDX, R8-R11, ST(0)-ST(7), XMM0-XMM5	RAX, RCX, RDX, RSI, RDI, R8-R11, ST(0)-ST(7), XMM0-XMM15
Registers that must be saved and restored	SI, DI, BP, DS	EBX, ESI, EDI, EBP	RBX, RSI, RDI, RBP, R12-R15, XMM6-XMM15	RBX, RBP, R12-R15
Registers that cannot be changed		DS, ES, FS, GS, SS		
Registers used for parameter transfer		(ECX)	RCX, RDX, R8, R9, XMM0-XMM3	RDI, RSI, RDX, RCX, R8, R9, XMM0-XMM7
Registers used for return values	AX, DX, ST(0)	EAX, EDX, ST(0)	RAX, XMM0	RAX, RDX, XMM0, XMM1, ST(0), ST(1)

The floating point registers `ST(0)` - `ST(7)` must be empty before any call or return, except when used for function return value. The MMX registers must be cleared by `EMMS` before any call or return.

The arithmetic flags can be changed freely. The direction flag may be set temporarily, but must be cleared before any call or return in 32-bit and 64-bit systems. The interrupt flag

cannot be cleared in protected operating systems. The floating point control word and bit 6-15 of the MXCSR register must be saved and restored in functions that modify them.

Register FS and GS are used for thread information blocks etc. and should not be changed. Other segment registers should not be changed, except in segmented 16-bit models.

4.2 Data storage

Variables and objects that are declared inside a function in C or C++ are stored on the stack and addressed relative to the stack pointer or a stack frame. 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 in 32-bit systems and with 32-bit RIP-relative addresses in 64-bit systems. 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.

4.3 Function calling conventions

Calling convention in 16 bit mode DOS and Windows 3.x

Function parameters are passed on the stack with the first parameter at the lowest address. This corresponds to pushing the last parameter first. The stack is cleaned up by the caller.

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. All stack parameters are aligned by 2.

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).

Calling convention in 32 bit Windows, Linux, BSD, Mac OS X

Function parameters are passed on the stack according to the following calling conventions:

Calling convention	Parameter order on stack	Parameters removed by
__cdecl	First par. at low address	Caller
__stdcall	First par. at low address	Subroutine
__fastcall Microsoft and Gnu	First 2 parameters in ecx, edx. Rest as __stdcall	Subroutine
__fastcall Borland	First 3 parameters in eax, edx, ecx. Rest as __stdcall	Subroutine
_pascal	First par. at high address	Subroutine
__thiscall Microsoft	this in ecx. Rest as __stdcall	Subroutine

The __cdecl calling convention is the default in Linux. In Windows, the __cdecl convention is also the default except for member functions, system functions and DLL's. Statically linked modules in .obj and .lib files should preferably use __cdecl, while dynamic link libraries in .dll files should use __stdcall. The Microsoft, Intel, Digital Mars and

Codeplay compilers use `__thiscall` by default for member functions under Windows, the Borland compiler uses `__cdecl` with '`this`' as the first parameter.

The fastest calling convention for functions with integer parameters is `__fastcall`, but this calling convention is not standardized.

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.

Parameters of 32 bits size or less use 4 bytes 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 aligned by 4.

Mac OS X and the Gnu compiler version 3 and later align the stack by 16 before every call instruction, though this behavior is not consistent. Sometimes the stack is aligned by 4. This discrepancy is an unresolved issue at the time of writing. See manual 5: "Calling conventions for different C++ compilers and operating systems" for details.

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, references and Booleans in `EAX`, 64-bit integers in `EDX:EAX`, and floating-point values in `ST(0)`.

See manual 5: "Calling conventions for different C++ compilers and operating systems" for details about parameters of composite types (`struct`, `class`, `union`) and vector types (`__m64`, `__m128`).

Calling conventions in 64 bit Windows

The first parameter is transferred in `RCX` if it is an integer or in `XMM0` if it is a `float` or `double`. The second parameter is transferred in `RDX` or `XMM1`. The third parameter is transferred in `R8` or `XMM2`. The fourth parameter is transferred in `R9` or `XMM3`. Note that `RCX` is not used for parameter transfer if `XMM0` is used, and vice versa. No more than four parameters can be transferred in registers, regardless of type. Any further parameters are transferred on the stack with the first parameter at the lowest address and aligned by 8. Member functions have '`this`' as the first parameter.

The caller must allocate 32 bytes of free space on the stack in addition to any parameters transferred on the stack. This is a shadow space where the called function can save the four parameter registers if it needs to. The shadow space is the place where the first four parameters would have been stored if they were transferred on the stack according to the `__cdecl` rule. The shadow space belongs to the called function which is allowed to store the parameters (or anything else) in the shadow space. The caller must reserve the 32 bytes of shadow space even for functions that have no parameters. The caller must clean up the stack, including the shadow space. Return values are in `RAX` or `XMM0`.

The stack pointer must be aligned by 16 before any `CALL` instruction, so that the value of `RSP` is 8 modulo 16 at the entry of a function. The function can rely on this alignment when storing `XMM` registers to the stack.

See manual 5: "Calling conventions for different C++ compilers and operating systems" for details about parameters of composite types (`struct`, `class`, `union`) and vector types (`__m64`, `__m128`).

Calling conventions in 64 bit Linux and BSD

The first six integer parameters are transferred in `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`, respectively. The first eight floating point parameters are transferred in `XMM0` - `XMM7`. All these registers can be used, so that a maximum of fourteen parameters can be transferred in registers. Any further parameters are transferred on the stack with the first parameters at the lowest address and aligned by 8. The stack is cleaned up by the caller if there are any parameters on the stack. There is no shadow space. Member functions have '`this`' as the first parameter. Return values are in `RAX` or `XMM0`.

The stack pointer must be aligned by 16 before any `CALL` instruction, so that the value of `RSP` is 8 modulo 16 at the entry of a function. The function can rely on this alignment when storing `XMM` registers to the stack.

The address range from `[RSP-1]` to `[RSP-128]` is called the red zone. A function can safely store data above the stack in the red zone as long as this is not overwritten by any `PUSH` or `CALL` instructions.

See manual 5: "Calling conventions for different C++ compilers and operating systems" for details about parameters of composite types (`struct`, `class`, `union`) and vector types (`__m64`, `__m128`).

4.4 Name mangling and name decoration

The support for function overloading in C++ makes it necessary to supply information about the parameters of a function to the linker. This is done by appending codes for the parameter types to the function name. This is called name mangling. The name mangling codes have traditionally been compiler specific. Fortunately, there is a growing tendency towards standardization in this area in order to improve compatibility between different compilers. The name mangling codes for different compilers are described in detail in manual 5: "Calling conventions for different C++ compilers and operating systems".

The problem of incompatible name mangling codes is most easily solved by using `extern "C"` declarations. Functions with `extern "C"` declaration have no name mangling. The only decoration is an underscore prefix in DOS and 32-bit Windows and Mac OS. There is some additional decoration of the name for functions with `__stdcall` and `__fastcall` declarations.

The `extern "C"` declaration cannot be used for member functions, overloaded functions, operators, and other constructs that are not supported in the C language. If it is necessary to turn off name mangling in object-oriented code then you can use the trick of translating member functions to friend functions, as illustrated in example 7.1b page 41.

4.5 Function examples

The following examples show how to code a function in assembly that obeys the calling conventions. First the code in C++:

```
// Example 4.1a
extern "C" double sinxpnx (double x, int n) {
   return sin(x) + n * x;
}
```

The same function can be coded in assembly. The following examples show the same function coded for different platforms.

```
; Example 4.1b. 16-bit DOS and Windows 3.x
ALIGN     4
```

```
_sinxpnx  PROC   NEAR
; parameter x = [SP+2]
; parameter n = [SP+10]
; return value = ST(0)

    push  bp                ; bp must be saved
    mov   bp, sp            ; stack frame
    fild  word ptr [bp+12]  ; n
    fld   qword ptr [bp+4]  ; x
    fmul  st(1), st(0)      ; n*x
    fsin                    ; sin(x)
    fadd                    ; sin(x) + n*x
    pop   bp                ; restore bp
    ret                     ; return value is in st(0)
_sinxpnx  ENDP
```

In 16-bit mode we need `BP` as a stack frame because `SP` cannot be used as base pointer. The integer `n` is only 16 bits. I have used the hardware instruction `FSIN` for the `sin` function.

```
; Example 4.1c. 32-bit Windows
EXTRN     _sin:near
ALIGN     4
_sinxpnx  PROC near
; parameter x = [ESP+4]
; parameter n = [ESP+12]
; return value = ST(0)

    fld   qword ptr [esp+4]  ; x
    sub   esp, 8             ; make space for parameter x
    fstp  qword ptr [esp]    ; store parameter for sin; clear st(0)
    call  _sin               ; library function for sin()
    add   esp, 8             ; clean up stack after call
    fild  dword ptr [esp+12] ; n
    fmul  qword ptr [esp+4]  ; n*x
    fadd                     ; sin(x) + n*x
    ret                      ; return value is in st(0)
_sinxpnx  ENDP
```

Here, I have chosen to use the library function `_sin` instead of `FSIN`. This may be faster in some cases because `FSIN` gives higher precision than needed. The parameter for `_sin` is transferred as 8 bytes on the stack.

```
; Example 4.1d. 32-bit Linux
EXTRN     sin:near
ALIGN     4
sinxpnx  PROC near
; parameter x = [ESP+4]
; parameter n = [ESP+12]
; return value = ST(0)

    fld   qword ptr [esp+4]  ; x
    sub   esp, 12            ; Keep stack aligned by 16 before call
    fstp  qword ptr [esp]    ; Store parameter for sin; clear st(0)
    call  sin                ; Library proc. may be faster than fsin
    add   esp, 12            ; Clean up stack after call
    fild  dword ptr [esp+12] ; n
    fmul  qword ptr [esp+4]  ; n*x
    fadd                     ; sin(x) + n*x
    ret                      ; Return value is in st(0)
sinxpnx  ENDP
```

In 32-bit Linux there is no underscore on function names. The stack must be kept aligned by 16 in Linux (GCC version 3 or later). The call to `sinxpnx` subtracts 4 from `ESP`. We are

subtracting 12 more from ESP so that the total amount subtracted is 16. We may subtract more, as long as the total amount is a multiple of 16. In example 4.1c we subtracted only 8 from ESP because the stack is only aligned by 4 in 32-bit Windows.

```
; Example 4.1e. 64-bit Windows
EXTRN     sin:near
ALIGN     4
sinxpnx   PROC
; parameter x = xmm0
; parameter n = edx
; return value = xmm0

    push     rbx              ; rbx must be saved
    movaps   [rsp+16],xmm6    ; save xmm6 in shadow space
    sub      rsp, 32          ; make shadow space for call to sin
    mov      ebx, edx         ; save n
    movsd    xmm6, xmm0       ; save x
    call     sin              ; xmm0 = sin(xmm0)
    cvtsi2sd xmm1, ebx        ; convert n to double
    mulsd    xmm1, xmm6       ; n * x
    addsd    xmm0, xmm1       ; sin(x) + n * x
    add      rsp, 32          ; restore stack pointer
    movaps   xmm6, [rsp+16]   ; restore xmm6
    pop      rbx              ; restore rbx
    ret                       ; return value is in xmm0
sinxpnx     ENDP
```

Function parameters are transferred in registers in 64-bit Windows. ECX is not used for parameter transfer because the first parameter is not an integer. We are using RBX and XMM6 for storing n and x across the call to sin. We have to use registers with callee-save status for this, and we have to save these registers on the stack before using them. The stack must be aligned by 16 before the call to sin. The call to sinxpnx subtracts 8 from RSP; the PUSH RBX instruction subtracts 8; and the SUB instruction subtracts 32. The total amount subtracted is 8+8+32 = 48, which is a multiple of 16. The extra 32 bytes on the stack is shadow space for the call to sin. Note that example 4.1e does not include support for exception handling. It is necessary to add tables with stack unwind information if the program relies on catching exceptions generated in the function sin.

```
; Example 4.1f. 64-bit Linux
EXTRN     sin:near
ALIGN     4
sinxpnx   PROC
PUBLIC    sinxpnx
; parameter x = xmm0
; parameter n = edi
; return value = xmm0

    push     rbx              ; rbx must be saved
    sub      rsp, 16          ; make local space and align stack by 16
    movaps   [rsp], xmm0      ; save x
    mov      ebx, edi         ; save n
    call     sin              ; xmm0 = sin(xmm0)
    cvtsi2sd xmm1, ebx        ; convert n to double
    mulsd    xmm1, [rsp]      ; n * x
    addsd    xmm0, xmm1       ; sin(x) + n * x
    add      rsp, 16          ; restore stack pointer
    pop      rbx              ; restore rbx
    ret                       ; return value is in xmm0
sinxpnx     ENDP
```

64-bit Linux does not use the same registers for parameter transfer as 64-bit Windows does. There are no XMM registers with callee-save status, so we have to save x on the stack

across the call to `sin`, even though saving it in a register might be faster (Saving `x` in a 64-bit integer register is possible, but slow). `n` can still be saved in a general purpose register with callee-save status. The stack is aligned by 16. There is no need for shadow space on the stack. The red zone cannot be used because it would be overwritten by the call to `sin`. Note that example 4.1f does not include support for exception handling. It is necessary to add tables with stack unwind information if the program relies on catching exceptions generated in the function `sin`.

5 Using intrinsic functions in C++

As already mentioned, there are three different ways of making assembly code: using intrinsic functions in C++, using inline assembly in C++, and making separate assembly modules. Intrinsic functions are described in this chapter. The other two methods are described in the following chapters.

The Microsoft, Intel and Gnu C++ compilers have support for the so-called intrinsic functions. Most of the intrinsic functions generate one machine instruction each. An intrinsic function is therefore equivalent to an assembly instruction.

Coding with intrinsic functions is a kind of high-level assembly. It can easily be combined with C++ language constructs such as `if`-statements, loops, functions, classes and operator overloading. Using intrinsic functions is an easier way of doing high level assembly coding than using `.if` constructs etc. in an assembler or using the so-called high level assembler (HLA).

The invention of intrinsic functions has made it much easier to do programming tasks that previously required coding with assembly syntax. The advantages of using intrinsic functions are:

- No need to learn assembly language syntax.
- Seamless integration into C++ code.
- Branches, loops, functions, classes, etc. are easily made with C++ syntax.
- The compiler takes care of calling conventions, register usage conventions, etc.
- The code is portable to almost all x86 platforms: 32-bit and 64-bit Windows, Linux, Mac OS, etc. Some intrinsic functions can even be used on Itanium and other non-x86 platforms.
- The code is compatible with Microsoft, Gnu and Intel compilers.
- The compiler takes care of register variables, register allocation and register spilling. The programmer doesn't have to care about which register is used for which variable.
- Different instances of the same inlined function or operator can use different registers for its parameters. This eliminates the need for register-to-register moves. The same function coded with assembly syntax would typically use a specific register for each parameter; and a move instruction would be required if the value happens to be in a different register.
- It is possible to define overloaded operators for the intrinsic functions. For example, the instruction that adds two 4-element vectors of floats is coded as `ADDPS` in

assembly language, and as `_mm_add_ps` when intrinsic functions are used. But an overloaded operator can be defined for the latter so that it is simply coded as a `+` when using the so-called vector classes. This makes the code look like plain old C++.

- The compiler can optimize the code further, for example by common subexpression elimination, loop-invariant code motion, scheduling and reordering, etc. This would have to be done manually if assembly syntax was used. The Gnu and Intel compilers provide the best optimization.

The disadvantages of using intrinsic functions are:

- Not all assembly instructions have intrinsic function equivalents.
- The function names are sometimes long and difficult to remember.
- An expression with many intrinsic functions looks kludgy and is difficult to read.
- Requires a good understanding of the underlying mechanisms.
- The compilers may not be able to optimize code containing intrinsic functions as good as other code, especially when constant propagation is needed.
- Unskilled use of intrinsic functions can make the code less efficient than simple C++ code.
- The compiler can modify the code or implement it in a less efficient way than the programmer intended. It may be necessary to look at the code generated by the compiler to see if it is optimized in the way the programmer intended.

5.1 Using intrinsic functions for system code

Intrinsic functions are useful for making system code and access system registers that are not accessible with standard C++. Some of these functions are listed below.

Functions for accessing system registers:
`__rdtsc`, `__readpmc`, `__readmsr`, `__readcr0`, `__readcr2`, `__readcr3`, `__readcr4`, `__readcr8`, `__writecr0`, `__writecr3`, `__writecr4`, `__writecr8`, `__writemsr`, `_mm_getcsr`, `_mm_setcsr`, `__getcallerseflags`.

Functions for input and output:
`__inbyte`, `__inword`, `__indword`, `__outbyte`, `__outword`, `__outdword`.

Functions for atomic memory read/write operations:
`_InterlockedExchange`, etc.

Functions for accessing `FS` and `GS` segments:
`__readfsbyte`, `__writefsbyte`, etc.

Cache control instructions (Require SSE or SSE2 instruction set):
`_mm_prefetch`, `_mm_stream_si32`, `_mm_stream_pi`, `_mm_stream_si128`, `_ReadBarrier`, `_WriteBarrier`, `_ReadWriteBarrier`, `_mm_sfence`.

Other system functions:
`__cpuid`, `__debugbreak`, `_disable`, `_enable`.

5.2 Using intrinsic functions for instructions not available in standard C++

Some simple instructions that are not available in standard C++ can be coded with intrinsic functions, for example functions for bit-rotate, bit-scan, etc.:
`_rotl8`, `_rotr8`, `_rotl16`, `_rotr16`, `_rotl`, `_rotr`, `_rotl64`, `_rotr64`, `_BitScanForward`, `_BitScanReverse`.

5.3 Using intrinsic functions for vector operations

Vector instructions are very useful for improving the speed of code with inherent parallelism. There are intrinsic functions for almost all vector operations using the MMX and XMM registers.

The use of these intrinsic functions for vector operations is thoroughly described in manual 1: "Optimizing software in C++".

5.4 Availability of intrinsic functions

The intrinsic functions are available on newer versions of Microsoft, Gnu and Intel compilers. Most intrinsic functions have the same names in all three compilers. You have to include a header file named `intrin.h` or `emmintrin.h` to get access to the intrinsic functions. The Codeplay compiler has limited support for intrinsic vector functions, but the function names are not compatible with the other compilers.

The intrinsic functions are listed in the help documentation for each compiler, in the appropriate header files, in msdn.microsoft.com and in Intel's "IA-32 Intel Architecture Software Developer's Manual", volume 2A and 2B (developer.intel.com).

6 Using inline assembly in C++

Inline assembly is another way of putting assembly code into a C++ file. The keyword `asm` or `_asm` or `__asm` or `__asm__` tells the compiler that the code is assembly. Different compilers have different syntaxes for inline assembly. The different syntaxes are explained below.

The advantages of using inline assembly are:

- It is easy to combine with C++.
- Variables and other symbols defined in C++ code can be accessed from the assembly code.
- Only the part of the code that cannot be coded in C++ is coded in assembly.
- All assembly instructions are available.
- The code generated is exactly what you write.
- It is possible to optimize in details.
- The compiler takes care of calling conventions, name mangling and saving registers.
- The compiler can inline a function containing inline assembly.
- Portable to different x86 platforms when using the Intel compiler.

The disadvantages of using inline assembly are:

- Different compilers use different syntax.
- Requires knowledge of assembly language.
- Requires a good understanding of how the compiler works. It is easy to make errors.
- The allocation of registers is mostly done manually. The compiler may allocate different registers for the same variables.
- The compiler cannot optimize well across the inline assembly code.
- It may be difficult to control function prolog and epilog code.
- It may not be possible to define data.
- It may not be possible to use macros and other directives.
- It may not be possible to make functions with multiple entries.
- The Microsoft compiler does not support inline assembly on 64-bit platforms.

The following sections illustrate how to make inline assembly with different compilers.

6.1 MASM style inline assembly

The most common syntax for inline assembly is a MASM-style syntax. This is the easiest way of making inline assembly and it is supported by most compilers, but not the Gnu compiler.

The following examples show a function that raises a floating point number `x` to an integer power `n`. The algorithm is to multiply x^1, x^2, x^4, x^8, etc. according to each bit in the binary representation of `n`. Actually, it is not necessary to code this in assembly because a good compiler will optimize it almost as good when you just write `pow(x,n)`. My purpose here is just to illustrate the syntax of inline assembly.

First the code in C++ to illustrate the algorithm:

```
// Example 6.1a. Raise double x to the power of int n.
double ipow (double x, int n) {
   unsigned int nn = abs(n);      // absolute value of n
   double y = 1.0;                // used for multiplication
   while (nn != 0) {              // loop for each bit in nn
      if (nn & 1) y *= x;         // multiply if bit = 1
      x *= x;                     // square x
      nn >>= 1;                   // get next bit of nn
   }
   if (n < 0) y = 1.0 / y;        // reciprocal if n is negative
   return y;                      // return y = pow(x,n)
}
```

And then the optimized code using inline assembly with MASM style syntax:

```
// Example 6.1b. MASM style inline assembly, 32 bit mode
double ipow (double x, int n) {
   __asm {
```

```
      mov eax, n          // Move n to eax
// abs(n) is calculated by inverting all bits and adding 1 if n < 0:
      cdq                 // Get sign bit into all bits of edx
      xor eax, edx        // Invert bits if negative
      sub eax, edx        // Add 1 if negative. Now eax = abs(n)
      fld1                // st(0) = 1.0
      jz L9               // End if n = 0
      fld qword ptr x     // st(0) = x, st(1) = 1.0
      jmp L2              // Jump into loop

   L1:                    // Top of loop
      fmul st(0), st(0)   // Square x
   L2:                    // Loop entered here
      shr eax, 1          // Get each bit of n into carry flag
      jnc L1              // No carry. Skip multiplication, goto next
      fmul st(1), st(0)   // Multiply by x squared i times for bit # i
      jnz L1              // End of loop. Stop when nn = 0

      fstp st(0)          // Discard st(0)
      test edx, edx       // Test if n was negative
      jns L9              // Finish if n was not negative
      fld1                // st(0) = 1.0, st(1) = x^abs(n)
      fdivr               // Reciprocal
   L9:                    // Finish
   }                      // Result is in st(0)
#pragma warning(disable:1011) // Don't warn for missing return value
}
```

Note that the function entry and parameters are declared with C++ syntax. The function body, or part of it, can then be coded with inline assembly. The parameters `x` and `n`, which are declared with C++ syntax, can be accessed directly in the assembly code using the same names. The compiler simply replaces `x` and `n` in the assembly code with the appropriate memory operands, probably `[esp+4]` and `[esp+12]`. If the inline assembly code needs to access a variable that happens to be in a register, then the compiler will store it to a memory variable on the stack and then insert the address of this memory variable in the inline assembly code.

The result is returned in `st(0)` according to the 32-bit calling convention. The compiler will normally issue a warning because there is no `return y;` statement in the end of the function. This statement is not needed if you know which register to return the value in. The `#pragma warning(disable:1011)` removes the warning. If you want the code to work with different calling conventions (e.g. 64-bit systems) then it is necessary to store the result in a temporary variable inside the assembly block:

```
// Example 6.1c. MASM style, independent of calling convention
double ipow (double x, int n) {
   double result;            // Define temporary variable for result
   __asm {
      mov eax, n
      cdq
      xor eax, edx
      sub eax, edx
      fld1
      jz L9
      fld qword ptr x
      jmp L2
   L1:fmul st(0), st(0)
   L2:shr eax, 1
      jnc L1
      fmul st(1), st(0)
      jnz L1
      fstp st(0)
```

```
      test edx, edx
      jns L9
      fld1
      fdivr
   L9:fstp qword ptr result  // store result to temporary variable
   }
   return result;
}
```

Now the compiler takes care of all aspects of the calling convention and the code works on all x86 platforms.

The compiler inspects the inline assembly code to see which registers are modified. The compiler will automatically save and restore these registers if required by the register usage convention. In some compilers it is not allowed to modify register `ebp` or `ebx` in the inline assembly code because these registers are needed for a stack frame. The compiler will generally issue a warning in this case.

It is possible to remove the automatically generated prolog and epilog code by adding `__declspec(naked)` to the function declaration. In this case it is the responsibility of the programmer to add any necessary prolog and epilog code and to save any modified registers if necessary. The only thing the compiler takes care of in a naked function is name mangling. Automatic variable name substitution may not work with naked functions because it depends on how the function prolog is made. A naked function cannot be inlined.

Accessing register variables

Register variables cannot be accessed directly by their symbolic names in MASM-style inline assembly. Accessing a variable by name in an inline assembly code will force the compiler to store the variable to a temporary memory location.

If you know which register a variable is in then you can simply write the name of the register. This makes the code more efficient but less portable.

For example, if the code in the above example is used in 64-bit Windows, then `x` will be in register `XMM0` and `n` will be in register `EDX`. Taking advantage of this knowledge, we can improve the code:

```
// Example 6.1d. MASM style, 64-bit Windows
double ipow (double x, int n) {
   const double one = 1.0;   // define constant 1.0
   __asm {                   // x is in xmm0
      mov eax, edx           // get n into eax
      cdq
      xor eax, edx
      sub eax, edx
      movsd xmm1, one        // load 1.0
      jz L9
      jmp L2
   L1:mulsd xmm0, xmm0       // square x
   L2:shr eax, 1
      jnc L1
      mulsd xmm1, xmm0       // Multiply by x squared i times
      jnz L1
      movsd xmm0, xmm1       // Put result in xmm0
      test edx, edx
      jns L9
      movsd xmm0, one
      divsd xmm0, xmm1       // Reciprocal
   L9:   }
#pragma warning(disable:1011) // Don't warn for missing return value
}
```

In 64-bit Linux we will have `n` in register `EDI` so the line `mov eax,edx` should be changed to `mov eax,edi`.

Accessing class members and structure members

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

```
// Example 6.2a. Accessing class data members
// define C++ class
class MyList {
   protected:
   int length;               // Number of items in list
   int buffer[100];          // Store items
   public:
   MyList();                 // Constructor
   void AttItem(int item);   // Add item to list
   int Sum();                // Compute sum of items
};

MyList::MyList() {           // Constructor
   length = 0;}

void MyList::AttItem(int item) { // Add item to list
   if (length < 100) {
      buffer[length++] = item;
   }
}

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

Below, I will show how to code the member function `MyList::Sum` in inline assembly. I have not tried to optimize the code, my purpose here is simply to show the syntax.

Class members are accessed by loading '`this`' into a pointer register and addressing class data members relative to the pointer with the dot operator (`.`).

```
// Example 6.2b. Accessing class members (32-bit)
int MyList::Sum() {
   __asm {
       mov ecx, this                // 'this' pointer
       xor eax, eax                 // sum = 0
       xor edx, edx                 // loop index, i = 0
       cmp [ecx].length, 0          // if (this->length != 0)
       je L9
   L1: add eax, [ecx].buffer[edx*4] // sum += buffer[i]
       add edx, 1                   // i++
       cmp edx, [ecx].length        // while (i < length)
       jb L1
   L9:
   }                                // Return value is in eax
   #pragma warning(disable:1011)
}
```

Here the '`this`' pointer is accessed by its name '`this`', and all class data members are addressed relative to '`this`'. The offset of the class member relative to '`this`' is obtained by writing the member name preceded by the dot operator. The index into the array named `buffer` must be multiplied by the size of each element in buffer `[edx*4]`.

Some 32-bit compilers for Windows put '`this`' in `ecx`, so the instruction `mov ecx,this` can be omitted. 64-bit systems require 64-bit pointers, so `ecx` should be replaced by `rcx` and `edx` by `rdx`. 64-bit Windows has '`this`' in `rcx`, while 64-bit Linux has '`this`' in `rdi`.

Structure members are accessed in the same way by loading a pointer to the structure into a register and using the dot operator. There is no syntax check against accessing `private` and `protected` members. There is no way to resolve the ambiguity if more than one structure or class has a member with the same name. The MASM assembler can resolve such ambiguities by using the `assume` directive or by putting the name of the structure before the dot, but this is not possible with inline assembly.

Calling functions

Functions are called by their name in inline assembly. Member functions can only be called from other member functions of the same class. Overloaded functions cannot be called because there is no way to resolve the ambiguity. It is not possible to use mangled function names. It is the responsibility of the programmer to put any function parameters on the stack or in the right registers before calling a function and to clean up the stack after the call. It is also the programmer's responsibility to save any registers you want to preserve across the function call, unless these registers have callee-save status.

Because of these complications, I will recommend that you go out of the assembly block and use C++ syntax when making function calls.

Syntax overview

The syntax for MASM-style inline assembly is not well described in any compiler manual I have seen. I will therefore summarize the most important rules here.

In most cases, the MASM-style inline assembly is interpreted by the compiler without invoking any assembler. You can therefore not assume that the compiler will accept anything that the assembler understands.

The inline assembly code is marked by the keyword `__asm`. Some compilers allow the alias `_asm`. The assembly code must be enclosed in curly brackets `{}` unless there is only one line. The assembly instructions are separated by newlines. Alternatively, you may separate the assembly instructions by the `__asm` keyword without any semicolons.

Instructions and labels are coded in the same way as in MASM. The size of memory operands can be specified with the `PTR` operator, for example `INC DWORD PTR [ESI]`. The names of instructions and registers are not case sensitive.

Variables, functions, and `goto` labels declared in C++ can be accessed by the same names in the inline assembly code. These names are case sensitive.

Data members of structures, classes and unions are accessed relative to a pointer register using the dot operator.

Comments are initiated with a semicolon (`;`) or a double slash (`//`).

Data definition statements are not allowed. For example, it is not possible to code an instruction that the inline assembler does not recognize by the use of `DB` directives.

Directives and macros are not allowed.

The compiler takes care of calling conventions and register saving for the function that contains inline assembly, but not for any function calls from the inline assembly.

Compilers using MASM style inline assembly

MASM-style inline assembly is supported by 16-bit and 32-bit Microsoft C++ compilers, but the 64-bit Microsoft compiler has no inline assembly.

The Intel C++ compiler supports MASM-style inline assembly in both 32-bit and 64-bit Windows as well as 32-bit and 64-bit Linux (and Mac OS ?). This is the preferred compiler for making portable code with inline assembly. The Intel compiler under Linux requires the command line option `-use-msasm` to recognize this syntax for inline assembly. Only the keyword `__asm` works in this case, not the synonyms `asm` or `__asm__`. The Intel compiler converts the MASM syntax to AT&T syntax before sending it to the assembler. The Intel compiler can therefore be useful as a syntax converter.

The MASM-style inline assembly is also supported by Borland, Digital Mars, Watcom and Codeplay compilers. The Borland assembler is not up to date and the Watcom compiler has some limitations on inline assembly.

6.2 Gnu style inline assembly

Inline assembly works quite differently on the Gnu compiler because the Gnu compiler compiles via assembly rather than producing object code directly, as most other compilers do. The assembly code is entered as a string constant which is passed through to the assembler with very little change. The default syntax is the AT&T syntax that the Gnu assembler uses.

The Gnu-style inline assembly has the advantage that you can pass on any instruction or directive to the assembler. Everything is possible. The disadvantage is that the syntax is very elaborate and difficult to learn, as the examples below show.

The Gnu-style inline assembly is supported by the Gnu compiler and the Intel compiler for Linux in both 32-bit and 64-bit mode.

AT&T syntax

Let's return to example 6.1b and translate it to Gnu style inline assembly with AT&T syntax:

```
// Example 6.1e. Gnu-style inline assembly, AT&T syntax
double ipow (double x, int n) {
   double y;
   __asm__ (
      "cltd                   \n" // cdq
      "xorl %%edx, %%eax      \n"
      "subl %%edx, %%eax      \n"
      "fld1                   \n"
      "jz 9f                  \n" // Forward jump to nearest 9:
      "fldl %[xx]             \n" // Substituted with x
      "jmp 2f                 \n" // Forward jump to nearest 2:
      "1:                     \n" // Local label 1:
      "fmul %%st(0), %%st(0)  \n"
      "2:                     \n" // Local label 2:
      "shrl $1, %%eax         \n"
      "jnc 1b                 \n" // Backward jump to nearest 1:
      "fmul %%st(0), %%st(1)  \n"
      "jnz 1b                 \n" // Backward jump to nearest 1:
      "fstp %%st(0)           \n"
      "testl %%edx, %%edx     \n"
```

```
        "jns 9f                \n" // Forward jump to nearest 9:
        "fld1                  \n"
        "fdivp %%st(0), %%st(1)\n"
        "9:                    \n" // Assembly string ends here

        // Use extended assembly syntax to specify operands:
        // Output operands:
        : "=t" (y)                 // Output top of stack to y

        // Input operands:
        : [xx] "m" (x), "a" (n)    // Input operand %[xx] = x, eax = n

        // Clobbered registers:
        : "%edx", "%st(1)"         // Clobber edx and st(1)
    );                             // __asm__ statement ends here

    return y;
}
```

We immediately notice that the syntax is very different from the previous examples. Many of the instruction codes have suffixes for specifying the operand size. Integer instructions use `b` for `BYTE`, `w` for `WORD`, `l` for `DWORD`, `q` for `QWORD`. Floating point instructions use `s` for `DWORD` (`float`), `l` for `QWORD` (`double`), `t` for `TBYTE` (`long double`). Some instruction codes are completely different, for example `CDQ` is changed to `CLTD`. The order of the operands is the opposite of MASM syntax. The source operand comes before the destination operand. Register names have `%%` prefix, which is changed to `%` before the string is passed on to the assembler. Constants have `$` prefix. Memory operands are also different. For example, `[ebx+ecx*4+20h]` is changed to `0x20(%%ebx,%%ecx,4)`.

Jump labels can be coded in the same way as in MASM, e.g. `L1:`, `L2:`, but I have chosen to use the syntax for local labels, which is a decimal number followed by a colon. The jump instructions can then use `jmp 1b` for jumping backwards to the nearest preceding `1:` label, and `jmp 1f` for jumping forwards to the nearest following `1:` label. The reason why I have used this kind of labels is that the compiler will produce multiple instances of the inline assembly code if the function is inlined, which is quite likely to happen. If we use normal labels like `L1:` then there will be more than one label with the same name in the final assembly file, which of course will produce an error. If you want to use normal labels then add `__attribute__((noinline))` to the function declaration to prevent inlining of the function.

The Gnu style inline assembly does not allow you to put the names of local C++ variables directly into the assembly code. Only global variable names will work because they have the same names in the assembly code. Instead you can use the so-called extended syntax as illustrated above. The extended syntax looks like this:

```
__asm__ ("assembly code string" : [output list] : [input list] :
[clobbered registers list] );
```

The assembly code string is a string constant containing assembly instructions separated by newline characters (`\n`).

In the above example, the output list is `"=t" (y)`. `t` means the top-of-stack floating point register `st(0)`, and `y` means that this should be stored in the C++ variable named `y` after the assembly code string.

There are two input operands in the input list. `[xx] "m" (x)` means replace `%[xx]` in the assembly string with a memory operand for the C++ variable `x`. `"a" (n)` means load the C++ variable `n` into register `eax` before the assembly string. There are many different

constraints symbols for specifying different kinds of operands and registers for input and output. See the GCC manual for details.

The clobbered registers list `"%edx", "%st(1)"` tells that registers `edx` and `st(1)` are modified by the inline assembly code. The compiler would falsely assume that these registers were unchanged if they didn't occur in the clobber list.

Intel syntax

The above example will be a little easier to code if we use Intel syntax for the assembly string. The Gnu assembler now accepts Intel syntax with the directive `.intel_syntax noprefix`. The `noprefix` means that registers don't need a `%`-sign as prefix.

```
// Example 6.1f. Gnu-style inline assembly, Intel syntax
double ipow (double x, int n) {
   double y;
   __asm__ (
      ".intel_syntax noprefix \n" // use Intel syntax for convenience
      "cdq                    \n"
      "xor eax, edx           \n"
      "sub eax, edx           \n"
      "fld1                   \n"
      "jz 9f                  \n"
      ".att_syntax prefix     \n" // AT&T syntax needed for %[xx]
      "fldl %[xx]             \n" // memory operand substituted with x
      ".intel_syntax noprefix \n" // switch to Intel syntax again
      "jmp 2f                 \n"
      "1:                     \n"
      "fmul st(0), st(0)      \n"
      "2:                     \n"
      "shr eax, 1             \n"
      "jnc 1b                 \n"
      "fmul st(1), st(0)      \n"
      "jnz 1b                 \n"
      "fstp st(0)             \n"
      "test edx, edx          \n"
      "jns 9f                 \n"
      "fld1                   \n"
      "fdivrp                 \n"
      "9:                     \n"
      ".att_syntax prefix     \n" // switch back to AT&T syntax

      // output operands:
      : "=t" (y)                  // output in top-of-stack goes to y
      // input operands:
      : [xx] "m" (x), "a" (n)     // input memory %[x] for x, eax for n
      // clobbered registers:
      : "%edx", "%st(1)" );       // edx and st(1) are modified

   return y;
}
```

Here, I have inserted `.intel_syntax noprefix` in the start of the assembly string which allows me to use Intel syntax for the instructions. The string must end with `.att_syntax prefix` to return to the default AT&T syntax, because this syntax is used in the subsequent code generated by the compiler. The instruction that loads the memory operand `x` must use AT&T syntax because the operand substitution mechanism uses AT&T syntax for the operand substituted for `%[xx]`. The instruction `fldl %[xx]` must therefore be written in AT&T syntax. I can still use AT&T-style local labels. The lists of output operands, input operands and clobbered registers are the same as in example 6.1e.

7 Using an assembler

There are certain limitations on what you can do with intrinsic functions and inline assembly. These limitations can be overcome by using an assembler. The principle is to write one or more assembly files containing the most critical functions of a program and writing the less critical parts in C++. The different modules are then linked together into an executable file.

The advantages of using an assembler are:

- There are almost no limitations on what you can do.
- You have complete control of all details of the final executable code.
- All aspects of the code can be optimized, including function prolog and epilog, parameter transfer methods, register usage, data alignment, etc.
- It is possible to make functions with multiple entries.
- It is possible to make code that is compatible with multiple compilers and multiple operating systems (see page 43).
- MASM and some other assemblers have a powerful macro language which opens up possibilities that are absent in most compiled high-level languages (see page 94).

The disadvantages of using an assembler are:

- Assembly language is difficult to learn. There are many instructions to remember.
- Coding in assembly takes more time than coding in a high level language.
- The assembly language syntax is not fully standardized.
- Assembly code tends to become poorly structured and spaghetti-like. It takes a lot of discipline to make assembly code well structured and readable for others.
- Assembly code is not portable between different microprocessor architectures.
- The programmer must know all details of the calling conventions and obey these conventions in the code.
- The assembler provides very little syntax checking. Many programming errors are not detected by the assembler.
- There are many things that you can do wrong in assembly code and the errors can have serious consequences.
- Errors in assembly code can be difficult to trace. For example, the error of not saving a register can cause a completely different part of the program to malfunction.
- Assembly language is not suitable for making a complete program. Most of the program has to be made in a different programming language.

The best way to start if you want to make assembly code is to first make the entire program in C or C++. Optimize the program with the use of the methods described in manual 1:

"Optimizing software in C++". If any part of the program needs further optimization then isolate this part in a separate module. Then translate the critical module from C++ to assembly. There is no need to do this translation manually. Most C++ compilers can produce assembly code. Turn on all relevant optimization options in the compiler when translating the C++ code to assembly. The assembly code thus produced by the compiler is a good starting point for further optimization. The compiler-generated assembly code is sure to have the calling conventions right. (The output produced by 64-bit compilers for Windows is not yet fully compatible with any assembler).

Inspect the assembly code produced by the compiler to see if there are any possibilities for further optimization. Sometimes compilers are very smart and produce code that is better optimized than what an average assembly programmer can do. In other cases, compilers are incredibly stupid and do things in very awkward and inefficient ways. It is in the latter case that it is justified to spend time on assembly coding.

Most IDE's (Integrated Development Environments) provide a way of including assembly modules in a C++ project. For example in Microsoft Visual Studio, you can define a "custom build step" for an assembly source file. The specification for the custom build step may, for example, look like this. Command line: `ml /c /Cx /Zi /coff $(InputName).asm`. Outputs: `$(InputName).obj`. Alternatively, you may use a makefile (see page 42) or a batch file.

The C++ files that call the functions in the assembly module should include a header file (`*.h`) containing function prototypes for the assembly functions. It is recommended to add `extern "C"` to the function prototypes to remove the compiler-specific name mangling codes from the function names.

Examples of assembly functions for different platforms are provided in paragraph 4.5, page 23ff.

7.1 Static link libraries

It is convenient to collect assembled code from multiple assembly files into a function library. The advantages of organizing assembly modules into function libraries are:

- The library can contain many functions and modules. The linker will automatically pick the modules that are needed in a particular project and leave out the rest so that no superfluous code is added to the project.

- A function library is easy and convenient to include in a C++ project. All C++ compilers and IDE's support function libraries.

- A function library is reusable. The extra time spent on coding and testing a function in assembly language is better justified if the code can be reused in different projects.

- Making as a reusable function library forces you to make well tested and well documented code with a well defined functionality and a well defined interface to the calling program.

- A reusable function library with a general functionality is easier to maintain and verify than an application-specific assembly code with a less well-defined responsibility.

- A function library can be used by other programmers who have no knowledge of assembly language.

A static link function library for Windows is built by using the library manager (e.g. `lib.exe`) to combine one or more `*.obj` files into a `*.lib` file.

A static link function library for Linux is built by using the archive manager (`ar`) to combine one or more `*.o` files into an `*.a` file.

A function library must be supplemented by a header file (`*.h`) containing function prototypes for the functions in the library. This header file is included in the C++ files that call the library functions (e.g. `#include "mylibrary.h"`).

It is convenient to use a makefile (see page 42) for managing the commands necessary for building and updating a function library.

7.2 Dynamic link libraries

The difference between static linking and dynamic linking is that the static link library is linked into the executable program file so that the executable file contains a copy of the necessary parts of the library. A dynamic link library (`*.dll` in Windows, `*.so` in Linux) is distributed as a separate file which is loaded at runtime by the executable file.

The advantages of dynamic link libraries are:

- Only one instance of the dynamic link library is loaded into memory when multiple programs running simultaneously use the same library.
- The dynamic link library can be updated without modifying the executable file.
- A dynamic link library can be called from most programming languages, such as Pascal, C#, Visual Basic (Calling from Java is possible but difficult).

The disadvantages of dynamic link libraries are:

- The whole library is loaded into memory even when only a small part of it is needed.
- Loading a dynamic link library takes extra time when the executable program file is loaded.
- Calling a function in a dynamic link library is less efficient than a static library because of extra call overhead and because of less efficient code cache use.
- The dynamic link library must be distributed together with the executable file.
- Multiple programs installed on the same computer must use the same version of a dynamic link library. This can cause many compatibility problems.

A DLL for Windows is made with the Microsoft linker (`link.exe`). The linker must be supplied one or more `.obj` or `.lib` files containing the necessary library functions and a `DllEntry` function, which just returns 1. A module definition file (`*.def`) is also needed. Note that DLL functions in 32-bit Windows use the `__stdcall` calling convention, while static link library functions use the `__cdecl` calling convention by default. An example source code can be found in www.agner.org/random/randoma.zip. Further instructions can be found in the Microsoft compiler documentation and in Iczelion's tutorials at win32asm.cjb.net.

I have no experience in making dynamic link libraries (shared objects) for Linux.

7.3 Libraries in source code form

A problem with subroutine libraries in binary form is that the compiler cannot optimize the function call. This problem can be solved by supplying the library functions as C++ source code.

If the library functions are supplied as C++ source code then the compiler can optimize away the function calling overhead by inlining the function. It can optimize register allocation across the function. It can do constant propagation. It can move invariant code when the function is called inside a loop, etc.

The compiler can only do these optimizations with C++ source code, not with assembly code. The code may contain inline assembly or intrinsic function calls. The compiler can do further optimizations if the code uses intrinsic function calls, but not if it uses inline assembly. Note that different compilers will not optimize the code equally well.

If the compiler uses whole program optimization then the library functions can simply be supplied as a C++ source file. If not, then the library code must be included with `#include` statements in order to enable optimization across the function calls. A function defined in an included file should be declared `static` and/or `inline` in order to avoid clashes between multiple instances of the function.

Some compilers with whole program optimization features can produce half-compiled object files that allow further optimization at the link stage. Unfortunately, the format of such files is not standardized - not even between different versions of the same compiler. It is possible that future compiler technology will allow a standardized format for half-compiled code. This format should, as a minimum, specify which registers are used for parameter transfer and which registers are modified by each function. It should preferably also allow register allocation at link time, constant propagation, common subexpression elimination across functions, and invariant code motion.

As long as such facilities are not available, we may consider using the alternative strategy of putting the entire innermost loop into an optimized library function rather than calling the library function from inside a C++ loop. This solution is used in Intel's Math Kernel Library (www.intel.com). If, for example, you need to calculate a thousand logarithms then you can supply an array of thousand arguments to a vector logarithm function in the library and receive an array of thousand results back from the library. This has the disadvantage that intermediate results have to be stored in arrays rather than transferred in registers.

7.4 Making classes in assembly

Classes are coded as structures in assembly and member functions are coded as functions that receive a pointer to the class/structure as a parameter.

It is not possible to apply the `extern "C"` declaration to a member function in C++ because `extern "C"` refers to the calling conventions of the C language which doesn't have classes and member functions. The most logical solution is to use the mangled function name. Returning to example 6.2a and b page 32, we can write the member function `int MyList::Sum()` with a mangled name as follows:

```
; Example 7.1a (Example 6.2b translated to stand alone assembly)
; Member function, 32-bit Windows, Microsoft compiler

; Define structure corresponding to class MyList:
MyList   STRUC
length_  DD  ?             ; length is a reserved name. Use length_
buffer   DD  100 DUP (?)   ; int buffer[100];
```

```
MyList   ENDS

; int MyList::Sum()
; Mangled function name compatible with Microsoft compiler (32 bit):
?Sum@MyList@@QAEHXZ PROC near
; Microsoft compiler puts 'this' in ECX
assume ecx: ptr MyList             ; ecx points to structure MyList
      xor eax, eax                 ; sum = 0
      xor edx, edx                 ; Loop index i = 0
      cmp [ecx].length_, 0         ; this->length
      je  L9                       ; Skip if length = 0
L1:   add eax, [ecx].buffer[edx*4] ; sum += buffer[i]
      add edx, 1                   ; i++
      cmp edx, [ecx].length_       ; while (i < length)
      jb  L1                       ; Loop
L9:   ret                          ; Return value is in eax
?Sum@MyList@@QAEHXZ ENDP           ; End of int MyList::Sum()
assume ecx: nothing                ; ecx no longer points to anything
```

The mangled function name `?Sum@MyList@@QAEHXZ` is compiler specific. Other compilers may have other name-mangling codes. Furthermore, other compilers may put '`this`' on the stack rather than in a register. These incompatibilities can be solved by using a `friend` function rather than a member function. This solves the problem that a member function cannot be declared `extern "C"`. The declaration in the C++ header file must then be changed to the following:

```
// Example 7.1b. Member function changed to friend function:

// An incomplete class declaration is needed here:
class MyList;

// Function prototype for friend function with 'this' as parameter:
extern "C" int MyList_Sum(MyList * ThisP);

// Class declaration:
class MyList {
   protected:
   int length;               // Data members:
   int buffer[100];

   public:
   MyList();                 // Constructor
   void AttItem(int item);   // Add item to list

   // Make MyList_Sum a friend:
   friend int MyList_Sum(MyList * ThisP);

   // Translate Sum to MyList_Sum by inline call:
   int Sum() {return MyList_Sum(this);}
};
```

The prototype for the friend function must come before the class declaration because some compilers do not allow `extern "C"` inside the class declaration. An incomplete class declaration is needed because the friend function needs a pointer to the class.

The above declarations will make the compiler replace any call to `MyList::Sum` by a call to `MyList_Sum` because the latter function is inlined into the former. The assembly implementation of `MyList_Sum` does not need a mangled name:

```
; Example 7.1c. Friend function, 32-bit mode

; Define structure corresponding to class MyList:
```

```
MyList   STRUC
length_  DD  ?             ; length is a reserved name. Use length_
buffer   DD  100 DUP (?)   ; int buffer[100];
MyList   ENDS

; extern "C" friend int MyList_Sum()
_MyList_Sum PROC near
; Parameter ThisP is on stack
      mov ecx, [esp+4]             ; ThisP
assume ecx: ptr MyList             ; ecx points to structure MyList
      xor eax, eax                 ; sum = 0
      xor edx, edx                 ; Loop index i = 0
      cmp [ecx].length_, 0         ; this->length
      je  L9                       ; Skip if length = 0
L1:   add eax, [ecx].buffer[edx*4] ; sum += buffer[i]
      add edx, 1                   ; i++
      cmp edx, [ecx].length_       ; while (i < length)
      jb  L1                       ; Loop
L9:   ret                          ; Return value is in eax
_MyList_Sum   ENDP                 ; End of int MyList_Sum()
assume ecx: nothing                ; ecx no longer points to anything
```

7.5 Thread-safe functions

A thread-safe or reentrant function is a function that works correctly when it is called simultaneously from more than one thread. Multithreading is used for taking advantage of computers with multiple CPU kernels. It is therefore reasonable to require that a function library intended for speed-critical applications should be thread-safe.

Functions are thread-safe when no variables are shared between threads, except for intended communication between the threads. Constant data can be shared between threads without problems. Variables that are stored on the stack are thread-safe because each thread has its own stack. The problem arises only with static variables stored in the data segment. Static variables are used when data have to be saved from one function call to the next. It is possible to make thread-local static variables, but this is inefficient and system-specific.

The best way to store data from one function call to the next in a thread-safe way is to let the calling function allocate storage space for these data. The most elegant way to do this is to encapsulate the data and the functions that need them in a class. Each thread must create an object of the class and call the member functions on this object. The previous paragraph shows how to make member functions in assembly.

If the thread-safe assembly function has to be called from C or another language that does not support classes, or does so in an incompatible way, then the solution is to allocate a storage buffer in each thread and supply a pointer to this buffer to the function.

7.6 Makefiles

A make utility is a universal tool to manage software projects. It keeps track of all the source files, object files, library files, executable files, etc. in a software project. It does so by means of a general set of rules based on the date/time stamps of all the files. If a source file is newer than the corresponding object file then the object file has to be re-made. If the object file is newer than the executable file then the executable file has to be re-made.

Any IDE (Integrated Development Environment) contains a make utility which is activated from a graphical user interface, but in most cases it is also possible to use a command-line version of the make utility. The command line make utility (called `make` or `nmake`) is based on a set of rules that you can define in a so-called makefile. The advantage of using a

makefile is that it is possible to define rules for any type of files, such as source files in any programming language, object files, library files, module definition files, resource files, executable files, zip files, etc. The only requirement is that a tool exists for converting one type of file to another and that this tool can be called from a command line with the file names as parameters.

The syntax for defining rules in a makefile is almost the same for all the different make utilities that come with different compilers for Windows and Linux.

Many IDE's also provide features for user-defined make rules for file types not known to the IDE, but these utilities are often less general and flexible than a stand-alone make utility.

The following is an example of a makefile for making a function library `mylibrary.lib` from three assembly source files `func1.asm`, `func2.asm`, `func3.asm` and packing it together with the corresponding header file `mylibrary.h` into a zip file `mylibrary.zip`.

```
# Example 7.2. makefile for mylibrary
mylibrary.zip: mylibrary.lib mylibrary.h
   wzzip $@ $?

mylibrary.lib: func1.obj func2.obj func3.obj
   lib /out:$@ $**

.asm.obj
   ml /c /Cx /coff /Fo$@ $*.asm
```

The line `mylibrary.zip: mylibrary.lib mylibrary.h` tells that the file `mylibrary.zip` is built from `mylibrary.lib` and `mylibrary.h`, and that it must be re-built if any of these has been modified later than the zip file. The next line, which must be indented, specifies the command needed for building the target file `mylibrary.zip` from its dependents `mylibrary.lib` and `mylibrary.h`. The line `.asm.obj` tells that any file with extension `.obj` can be built from a file with the same name and extension `.asm` by using the rule in the following indented line.

The build rules can use the following macros for specifying file names:

`$@` = Current target's full name
`$<` = Full name of dependent file
`$*` = Current target's base name without extension
`$**` = All dependents of the current target, separated by spaces (nmake)
`$+` = All dependents of the current target, separated by spaces (Gnu make)
`$?` = All dependents with a later timestamp than the current target

The make utility is activated with the command
`nmake /Fmakefile` or
`make -f makefile`.

See the manual for the particular make utility for details.

8 Making function libraries compatible with multiple compilers and platforms

There are a number of compatibility problems to take care of if you want to make a function library that is compatible with multiple compilers, multiple programming languages, and multiple operating systems. The most important compatibility problems have to do with:

1. Name mangling
2. Calling conventions
3. Object file formats

Methods to overcome these compatibility problems are discussed in the following paragraphs.

8.1 Supporting multiple name mangling schemes

The easiest way to deal with the problems of compiler-specific name mangling schemes is to turn off name mangling with the `extern "C"` directive, as explained on page 23.

The `extern "C"` directive cannot be used for class member functions. This problem may be solved by using friend functions instead of member functions, as explained on page 41. Likewise, overloaded functions may be translated to functions with specific names by inlining.

However, in some cases it is desired to preserve the name mangling. Either because it makes the C++ code simpler, or because the mangled names contain information about calling conventions and other compatibility issues.

An assembly function can be made compatible with multiple name mangling schemes simply by giving it multiple public names. Returning to example 4.1c page 24, we can add mangled names for multiple compilers in the following way:

```
; Example 8.1. (Example 4.1c rewritten)
; Function with multiple mangled names (32-bit mode)

; double sinxpnx (double x, int n) {return sin(x) + n*x;}

ALIGN     4
_sinxpnx  PROC NEAR            ; extern "C" name

; Make public names for each name mangling scheme:
?sinxpnx@@YANNH@Z  LABEL NEAR  ; Microsoft compiler
@sinxpnx$qdi       LABEL NEAR  ; Borland compiler
_Z7sinxpnxdi       LABEL NEAR  ; Gnu compiler for Linux
__Z7sinxpnxdi      LABEL NEAR  ; Gnu compiler for Windows and Mac OS
PUBLIC ?sinxpnx@@YANNH@Z, @sinxpnx$qdi, _Z7sinxpnxdi, __Z7sinxpnxdi

; parameter x = [ESP+4]
; parameter n = [ESP+12]
; return value = ST(0)

    fild  dword ptr [esp+12] ; n
    fld   qword ptr [esp+4]  ; x
    fmul  st(1), st(0)       ; n*x
    fsin                     ; sin(x)
    fadd                     ; sin(x) + n*x
    ret                      ; return value is in st(0)
_sinxpnx  ENDP
```

Example 8.1 works with most compilers in both 32-bit Windows and 32-bit Linux because the calling conventions are the same. A function can have multiple public names and the linker will simply search for a name that matches the call from the C++ file. But a function call cannot have more than one external name.

The syntax for name mangling for different compilers is described in manual 5: "Calling conventions for different C++ compilers and operating systems". Applying this syntax manually is a difficult job. It is much easier and safer to generate each mangled name by compiling the function in C++ with the appropriate compiler. Command line versions of most compilers are available for free or as trial versions.

The Intel, Digital Mars and Codeplay compilers for Windows are compatible with the Microsoft name mangling scheme. The Intel compiler for Linux is compatible with the Gnu name mangling scheme. Gnu compilers version 2.x and earlier have a different name mangling scheme which I have not included in example 8.1. Mangled names for the Watcom compiler contain special characters which are only allowed by the Watcom assembler.

8.2 Supporting multiple calling conventions in 32 bit mode

Member functions in 32-bit Windows do not always have the same calling convention. The Microsoft-compatible compilers use the `__thiscall` convention with '`this`' in register `ecx`, while Borland and Gnu compilers use the `__cdecl` convention with '`this`' on the stack. This problem can be solved by making a function with multiple entries.

The following example is a rewrite of example 7.1a page 40 with two entries for the two different calling conventions:

```
; Example 8.2a (Example 7.1a with two entries)
; Member function, 32-bit mode
; int MyList::Sum()

; Define structure corresponding to class MyList:
MyList   STRUC
length_  DD  ?
buffer   DD  100 DUP (?)
MyList   ENDS

_MyList_Sum       PROC NEAR   ; for extern "C" friend function

; Make mangled names for compilers with __cdecl convention:
@MyList@Sum$qv    LABEL NEAR  ; Borland compiler
_ZN6MyList3SumEv  LABEL NEAR  ; Gnu comp. for Linux
__ZN6MyList3SumEv LABEL NEAR  ; Gnu comp. for Windows and Mac OS
PUBLIC @MyList@Sum$qv, _ZN6MyList3SumEv, __ZN6MyList3SumEv

     ; Move 'this' from the stack to register ecx:
      mov ecx, [esp+4]

; Make mangled names for compilers with __thiscall convention:
?Sum@MyList@@QAEHXZ LABEL NEAR    ; Microsoft compiler
PUBLIC ?Sum@MyList@@QAEHXZ
assume ecx: ptr MyList             ; ecx points to structure MyList
      xor eax, eax                 ; sum = 0
      xor edx, edx                 ; Loop index i = 0
      cmp [ecx].length_, 0         ; this->length
      je  L9                       ; Skip if length = 0
L1:   add eax, [ecx].buffer[edx*4] ; sum += buffer[i]
      add edx, 1                   ; i++
      cmp edx, [ecx].length_       ; while (i < length)
      jb  L1                       ; Loop
L9:   ret                          ; Return value is in eax
_MyList_Sum ENDP                   ; End of int MyList::Sum()
assume ecx: nothing                ; ecx no longer points to anything
```

The difference in name mangling schemes is actually an advantage here because it enables the linker to lead the call to the entry that corresponds to the right calling convention.

The method becomes more complicated if the member function has more parameters. Consider the function `void MyList::AttItem(int item)` on page 32. The `__thiscall` convention has the parameter '`this`' in `ecx` and the parameter `item` on the stack at `[esp+4]` and requires that the stack is cleaned up by the function. The `__cdecl` convention has both parameters on the stack with '`this`' at `[esp+4]` and `item` at `[esp+8]` and the stack cleaned up by the caller. A solution with two function entries requires a jump:

```
; Example 8.2b
; void MyList::AttItem(int item);

_MyList_AttItem       PROC NEAR    ; for extern "C" friend function

; Make mangled names for compilers with __cdecl convention:
@MyList@AttItem$qi    LABEL NEAR   ; Borland compiler
_ZN6MyList7AttItemEi  LABEL NEAR   ; Gnu comp. for Linux
__ZN6MyList7AttItemEi LABEL NEAR   ; Gnu comp. for Windows and Mac OS
PUBLIC @MyList@AttItem$qi, _ZN6MyList7AttItemEi, __ZN6MyList7AttItemEi

      ; Move parameters into registers:
      mov  ecx, [esp+4]            ; ecx = this
      mov  edx, [esp+8]            ; edx = item
      jmp  L0                      ; jump into common section

; Make mangled names for compilers with __thiscall convention:
?AttItem@MyList@@QAEXH@Z LABEL NEAR; Microsoft compiler
PUBLIC ?AttItem@MyList@@QAEXH@Z
      pop  eax                     ; Remove return address from stack
      pop  edx                     ; Get parameter item from stack
      push eax                     ; Put return address back on stack

L0:   ; common section where parameters are in registers
      ; ecx = this, edx = item

      assume ecx: ptr MyList       ; ecx points to structure MyList
      mov  eax, [ecx].length_      ; eax = this->length
      cmp  eax, 100                ; Check if too high
      jnb  L9                      ; List is full. Exit
      mov  [ecx].buffer[eax*4],edx ; buffer[length] = item
      add  eax, 1                  ; length++
      mov  [ecx].length_, eax
L9:   ret
_MyList_AttItem ENDP               ; End of MyList::AttItem
assume ecx: nothing                ; ecx no longer points to anything
```

In example 8.2b, the two function entries each load all parameters into registers and then jumps to a common section that doesn't need to read parameters from the stack. The `__thiscall` entry removes parameters from the stack before the common section.

Another compatibility problem occurs when we want to have a static and a dynamic link version of the same function library in 32-bit Windows. The static link library uses the `__cdecl` convention by default, while the dynamic link library uses the `__stdcall` convention by default. The static link library is the most efficient solution for C++ programs, but the dynamic link library is needed by several other programming languages.

One solution to this problem is to specify the `__stdcall` convention for both libraries. Another solution is to make functions with two entries.

The following example shows the function from example 8.1 with two entries for the `__cdecl` and `__stdcall` calling conventions. Both conventions have the parameters on the stack. The difference is that the stack is cleaned up by the caller in the `__cdecl` convention and by the called function in the `__stdcall` convention.

```
; Example 8.3a (Example 8.1 with __stdcall and __cdecl entries)
; Function with entries for __stdcall and __cdecl (32-bit Windows):

ALIGN     4
; __stdcall entry:
; extern "C" double __stdcall sinxpnx (double x, int n);
_sinxpnx@12        PROC NEAR
    ; Get all parameters into registers
    fild  dword ptr [esp+12] ; n
    fld   qword ptr [esp+4]  ; x

    ; Remove parameters from stack:
    pop eax                  ; Pop return address
    add esp, 12              ; remove 12 bytes of parameters
    push eax                 ; Put return address back on stack
    jmp L0

; __cdecl entry:
; extern "C" double __cdecl sinxpnx (double x, int n);
_sinxpnx           LABEL NEAR
PUBLIC _sinxpnx
    ; Get all parameters into registers
    fild  dword ptr [esp+12] ; n
    fld   qword ptr [esp+4]  ; x
    ; Don't remove parameters from the stack. This is done by caller

L0: ; Common entry with parameters all in registers
; parameter x = st(0)
; parameter n = st(1)
    fmul  st(1), st(0)       ; n*x
    fsin                     ; sin(x)
    fadd                     ; sin(x) + n*x
    ret                      ; return value is in st(0)
_sinxpnx@12  ENDP
```

The method of removing parameters from the stack in the function prolog rather than in the epilog is admittedly rather kludgy. A more efficient solution is to use conditional assembly:

```
; Example 8.3b
; Function with versions for __stdcall and __cdecl (32-bit Windows)
; Choose function prolog according to calling convention:
IFDEF STDCALL_               ; If STDCALL_ is defined
    _sinxpnx@12  PROC NEAR   ; extern "C" __stdcall function name
ELSE
    _sinxpnx     PROC NEAR   ; extern "C" __cdecl function name
ENDIF

; Function body common to both calling conventions:
    fild  dword ptr [esp+12] ; n
    fld   qword ptr [esp+4]  ; x
    fmul  st(1), st(0)       ; n*x
    fsin                     ; sin(x)
    fadd                     ; sin(x) + n*x

; Choose function epilog according to calling convention:
IFDEF STDCALL_               ; If STDCALL_ is defined
    ret 12                   ; Clean up stack if __stdcall
    _sinxpnx@12  ENDP        ; End of function
```

```
ELSE
    ret                     ; Don't clean up stack if __cdecl
    _sinxpnx  ENDP          ; End of function
ENDIF
```

This solution requires that you make two versions of the object file, one with `__cdecl` calling convention for the static link library and one with `__stdcall` calling convention for the dynamic link library. The distinction is made on the command line for the assembler. The `__stdcall` version is assembled with `/DSTDCALL_` on the command line to define the macro `STDCALL_`, which is detected by the `IFDEF` conditional.

8.3 Supporting multiple calling conventions in 64 bit mode

Calling conventions are better standardized in 64-bit systems than in 32-bit systems. There is only one calling convention for 64-bit Windows and one calling convention for 64-bit Linux and other Unix-like systems. Unfortunately, the two sets of calling conventions are quite different. The most important differences are:

- Function parameters are transferred in different registers in the two systems.
- Registers `RSI`, `RDI`, and `XMM6` - `XMM15` have callee-save status in 64-bit Windows but not in 64-bit Linux.
- The caller must reserve a "shadow space" of 32 bytes on the stack for the called function in 64-bit Windows but not in Linux.
- A "red zone" of 128 bytes below the stack pointer is available for storage in 64-bit Linux but not in Windows.
- The Microsoft name mangling scheme is used in 64-bit Windows, the Gnu name mangling scheme is used in 64-bit Linux.

Both systems have the stack aligned by 16 and both systems have the stack cleaned up by the caller.

It is possible to make functions that can be used in both systems when the differences between the two systems are taken into account. The function should save the registers that have callee-save status in Windows or leave them untouched. The function should not use the shadow space or the red zone. The function should reserve a shadow space for any function it calls. The function needs two entries in order to resolve the differences in registers used for parameter transfer if it has at least one integer parameter.

Let's use example 4.1 page 23 once more and make an implementation that works in both 64-bit Windows and 64-bit Linux.

```
; Example 8.4 (Example 4.1e/f combined).
; Support for both 64-bit Windows and 64-bit Linux
; double sinxpnx (double x, int n) {return sin(x) + n * x;}

EXTRN     sin:near
ALIGN     8

; 64-bit Linux entry:
_Z7sinxpnxdi PROC NEAR       ; Gnu name mangling

    ; Linux has n in edi, Windows has n in edx. Move it:
    mov      edx, edi

; 64-bit Windows entry:
```

```
?sinxpnx@@YANNH@Z LABEL NEAR ; Microsoft name mangling
PUBLIC ?sinxpnx@@YANNH@Z
; parameter x = xmm0
; parameter n = edx
; return value = xmm0

    push     rbx             ; rbx must be saved
    sub      rsp, 48         ; space for x, shadow space f. sin, align
    movapd   [rsp+32], xmm0  ; save x across call to sin
    mov      ebx, edx        ; save n across call to sin
    call     sin             ; xmm0 = sin(xmm0)
    cvtsi2sd xmm1, ebx       ; convert n to double
    mulsd    xmm1, [rsp+32]  ; n * x
    addsd    xmm0, xmm1      ; sin(x) + n * x
    add      rsp, 48         ; restore stack pointer
    pop      rbx             ; restore rbx
    ret                      ; return value is in xmm0
_Z7sinxpnxdi  ENDP           ; End of function
```

We are not using `extern "C"` declaration here because we are relying on the different name mangling schemes for distinguishing between Windows and Linux. The two entries are used for resolving the differences in parameter transfer. If the function declaration had `n` before `x`, i.e. `double sinxpnx (int n, double x);`, then the Windows version would have `x` in `XMM1` and `n` in `ecx`, while the Linux version would still have `x` in `XMM0` and `n` in `EDI`.

The function is storing `x` on the stack across the call to `sin` because there are no `XMM` registers with callee-save status in 64-bit Linux. The function reserves 32 bytes of shadow space for the call to `sin` even though this is not needed in Linux.

8.4 Supporting different object file formats

Another compatibility problem stems from differences in the formats of object files.

Borland, Digital Mars and 16-bit Microsoft compilers use the OMF format for object files. Microsoft and Gnu compilers for 32-bit Windows use the PE32 format, also called COFF. The Gnu compiler under Linux prefers the ELF32 format. Gnu compilers for Mac OS X prefer the Mach-O format. The 32-bit Codeplay compiler supports both the OMF, PE32 and ELF32 formats.

All compilers for 64-bit Windows use the PE32+ format, while compilers for 64-bit Linux use the ELF64 format.

The MASM assembler can produce both OMF, PE32 and PE32+ format object files, but not ELF format. The NASM assembler can produce OMF, PE32 and ELF32 formats. The YASM assembler can produce OMF, PE32, ELF32, PE32+ and ELF64 formats. The Gnu assembler (Gas) can produce ELF32 and ELF64 formats.

There are various utilities for converting object files from one format to another. Some compilers (e.g. Digital Mars) include a utility called `coff2omf.exe` which converts `.obj` and `.lib` files from PE32 to OMF format. The Microsoft linker (`link.exe`) can convert from OMF to PE32 format. The `objcopy` utility that comes with the binutils package of Mingw32 can convert between PE32 and ELF32, though it has a bug on external calls. A freeware utility called EMXAOUT1 can translate object files from OMF format to the old a.out format that many Gnu linkers accept. I have not found any utility that can convert between PE32+ and ELF64, but I am considering making such an utility and publishing it.

It is possible to do cross-platform development if you have a suitable object file conversion utility or an assembler that supports all the object file formats you need. This is useful for making function libraries that work on multiple platforms.

It is also possible to use the MASM assembler under Linux. The Windows emulator called Wine allows you to call MASM under Linux. You can then use an appropriate object file conversion utility for converting the object file to ELF. The process is as follows:

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

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

8.5 Supporting 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, it is preferred 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 are usually implemented differently in different languages.

Many modern programming languages such as C# and Visual Basic.NET cannot link to static link libraries. You have to use a dynamic link library instead.

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

9 Optimizing for speed

9.1 Identify the most critical parts of your code

Optimizing software is not just a question of fiddling with the right assembly instructions. Many modern applications use much more time on loading modules, resource files, databases, interface frameworks, etc. than on actually doing the calculations the program is made for. Optimizing the calculation time does not help when the program spends 99.9% of its time on something else than calculation. It is important to find out where the biggest time consumers are before you start to optimize anything. Sometimes the solution can be to change from C# to C++, to use a different user interface framework, to organize file input and output differently, to cache network data, to avoid dynamic memory allocation, etc., rather than using assembly language. See manual 1: "Optimizing software in C++" for further discussion.

The use of assembly code for optimizing software is relevant only for highly CPU-intensive programs such as sound and image processing, encryption, sorting, data compression and complicated mathematical calculations.

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

different places in the code to see which part is executed most times. Use the methods described on page 132 for measuring how long time each part of the code takes.

It is important to study the algorithm used in the critical part of the code to see if it can be improved. Often you can gain more speed simply by choosing the optimal algorithm than by any other optimization method.

9.2 Out of order execution

All modern x86 processors can execute instructions out of order. Consider this example:

```
; Example 9.1a, Out-of-order execution
mov  eax, [mem1]
imul eax, 6
mov  [mem2], eax
mov  ebx, [mem3]
add  ebx, 2
mov  [mem4], ebx
```

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 it happens that `[mem1]` is not in the cache then the CPU has to wait many clock cycles while this operand is being fetched from main memory. The CPU will look for something else to do in the meantime. It cannot do the second instruction `imul eax,6` because it depends on the output of the first instruction. But the third instruction `mov ebx,[mem3]` is independent of the preceding instructions so it is possible to execute `mov ebx,[mem3]` and `add ebx,2` while it is waiting for `[mem1]`. The CPU's have many features to support efficient out-of-order execution. Most important is, of course, the ability to detect whether an instruction depends on the output of a previous instruction. Another important feature is register renaming. Assume that the we are using the same register for multiplying and adding in example 9.1a because there are no more spare registers:

```
; Example 9.1b, Out-of-order execution with register renaming
mov eax, [mem1]
imul eax, 6
mov [mem2], eax
mov eax, [mem3]
add eax, 2
mov [mem4], eax
```

Example 9.1b will work exactly as fast as example 9.1a because the CPU is able to use different physical registers for the same logical register `eax`. This works in a very elegant way. The CPU assigns a new physical register to hold the value of `eax` every time `eax` is written to. This means that the above code is changed inside the CPU to a code that uses four different physical registers for `eax`. The first register is used for the value loaded from `[mem1]`. The second register is used for the output of the `imul` instruction. The third register is used for the value loaded from `[mem3]`. And the fourth register is used for the output of the `add` instruction. The use of different physical registers for the same logical register enables the CPU to make the last three instructions in example 9.1b independent of the first three instructions. The CPU must have a lot of physical registers for this mechanism to work efficiently. The number of physical registers is different for different microprocessors, but you can generally assume that the number is sufficient for quite a lot of instruction reordering.

Some CPU's can keep different parts of a register separate, while other CPU's always treat a register as a whole. If we change example 9.1b so that the second part uses 16-bit registers then we have the problem of a false dependence:

```
; Example 9.1c, False dependence of partial register
mov eax, [mem1]       ; 32 bit memory operand
imul eax, 6
mov [mem2], eax
mov ax, [mem3]        ; 16 bit memory operand
add ax, 2
mov [mem4], ax
```

Here the instruction `mov ax,[mem3]` changes only the lower 16 bits of register `eax`, while the upper 16 bits retain the value they got from the `imul` instruction. Many CPU's from both Intel and AMD are unable to rename a partial register. The consequence is that the `mov ax,[mem3]` instruction has to wait for the `imul` instruction to finish because it needs to combine the 16 lower bits from `[mem3]` with the 16 upper bits from the `imul` instruction. Other CPU's are able to resolve the situation and avoid the false dependence in various more or less efficient ways. The problem is avoided by replacing `mov ax,[mem3]` with `movzx eax,[mem3]`. This resets the high bits of `eax` and breaks the dependence on any previous value of `eax`. In 64-bit mode, it is sufficient to write to the 32-bit register because this always resets the upper part of a 64-bit register. Thus, `movzx eax,[mem3]` and `movzx rax,[mem3]` are doing exactly the same thing. The 32-bit version of the instruction is one byte shorter than the 64-bit version. Any use of the high 8-bit registers `AH`, `BH`, `CH`, `DH` should be avoided because it can cause false dependences and less efficient code.

Another important feature is the splitting of instructions into micro-operations (abbreviated µops or uops). The following example shows the advantage of this.

```
; Example 9.2, Splitting instructions into uops
push  eax
call  SomeFunction
```

The `push eax` instruction does two things. It subtracts 4 from the stack pointer and stores `eax` to the address pointed to by the stack pointer. Assume now that `eax` is the result of a long and time-consuming calculation. This delays the `push` instruction. The `call` instruction depends on the value of the stack pointer which is modified by the `push` instruction. If instructions were not split into uops then the `call` instruction would have to wait until the `push` instruction was finished. But the CPU splits the `push eax` instruction into `sub esp,4` followed by `mov [esp],eax`. The `sub esp,4` micro-operation can be executed before `eax` is ready so the `call` instruction will wait only for `sub esp,4` but not for `mov [esp],eax`.

Out-of-order execution becomes even more efficient when the CPU can do more than one thing at the same time. Many CPU's can do two, three or four things at the same time if the things to do are independent of each other and do not use the same execution units in the CPU. Most CPU's have at least two integer ALU's (Arithmetic Logic Units) so that they can do two or more integer additions per clock cycle. There is usually one floating point add unit and one floating point multiplication unit so that it is possible to do a floating point addition and a multiplication at the same time. There is at least one memory read unit and one memory write unit so that it is possible to read and write to memory at the same time. The maximum average number of uops per clock cycle is three or four so that it is possible, for example, to do an integer operation, a floating point operation, and a memory operation in the same clock cycle. The maximum number of arithmetic operations (i.e. anything else than memory read or write) is limited to two or three uops per clock cycle, depending on the CPU.

Floating point operations are usually pipelined so that a new floating point addition can start before the previous addition is finished. MMX and XMM instructions use the floating point execution units even for integer instructions on many CPU's. The details about which instructions can be executed simultaneously or pipelined and how many clock cycles each instruction takes are CPU specific. The details for each type of CPU are explained manual 3: "The microarchitecture of Intel and AMD CPU's" and manual 4: "Instruction tables".

The most important things you have to be aware of in order to take maximum advantage of out-or-order execution are:

- At least the following registers can be renamed: all integer registers, the stack pointer, the flags register, floating point registers, MMX registers, and XMM registers. Some CPU's can also rename segment registers and the floating point control word.

- Prevent false dependences by writing to a full register rather than a partial register.

- The `INC` and `DEC` instructions should be avoided because they write to only part of the flags register (excluding the carry flag). Use `ADD` or `SUB` instead to avoid false dependences or inefficient splitting of the flags register.

- A chain of instructions where each instruction depends on the previous one cannot execute out of order. Avoid long dependence chains. (See page 55).

- Memory operands cannot be renamed.

- A memory read can execute before a preceding memory write to a different address. Any pointer or index registers should be calculated as early as possible so that the CPU can verify that the addresses of memory operands are different.

- A memory write cannot execute before a preceding write, but the write buffers can hold a number of pending writes, typically four or six.

- A memory read can execute before another preceding read on Intel processors, but not on AMD processors.

- The CPU can do more things simultaneously if the code contains a good mixture of instructions from different categories, such as: simple integer instructions, floating point addition, multiplication, memory read, memory write.

9.3 Instruction fetch, decoding and retirement

Instruction fetching can be a bottleneck. Many processors cannot fetch more than 16 bytes of instruction code per clock cycle. It may be necessary to make instructions as short as possible if this limit turns out to be critical. One way of making instructions shorter is to replace memory operands by pointers (see chapter 10 page 62). The address of memory operands can possibly be loaded into pointer registers outside of a loop if fetching is a bottleneck. Large constants can likewise be loaded into registers.

Instruction fetching is delayed by jumps on most processors. It is important to minimize the number of jumps in critical code. Branches that are not taken and correctly predicted do not delay instruction fetching. It is therefore advantageous to organize if-else branches so that the branch that is followed most commonly is the one where the conditional jump is not taken.

Some processors fetch instructions in aligned 16-byte blocks. It can be advantageous to align critical loop entries and subroutine entries by 16 in order to minimize the number of 16-byte boundaries in the code.

Instruction decoding is often a bottleneck. The organization of instructions that gives the optimal decoding is processor-specific. Intel PM processors require a 4-1-1 decode pattern. This means that instructions which generate 2, 3 or 4 uops should be interspersed by two

single-uop instructions. On Core2 processors the optimal decode pattern is 4-1-1-1. On AMD64 processors it is preferred to avoid instructions that generate more than 2 uops.

Instructions with multiple prefixes can slow down decoding. The maximum number of prefixes that an instruction can have without slowing down decoding is 1 on 32-bit Intel processors, 2 on 64-bit Intel processors, and 3 on 64-bit AMD processors. Avoid address size prefixes. Avoid operand size prefixes on instructions with an immediate operand. For example, it may be advantageous to replace `MOV AX,200` by `MOV EAX,200`.

Decoding is rarely a bottleneck on processors with a trace cache, but there are specific requirements for optimal use of the trace cache.

Some Intel processors have a problem called register read stalls. This occurs if the code has several registers which are often read from but seldom written to.

Instruction retirement can be a bottleneck on all processors. AMD processors and Intel PM and P4 processors can retire no more than 3 uops per clock cycle. Core2 processors can retire 4 uops per clock cycle. No more than one taken jump can retire per clock cycle.

All these details are processor-specific. See manual 3: "The microarchitecture of Intel and AMD CPU's" for details.

9.4 Instruction latency and throughput

The latency of an instruction is the number of clock cycles it takes from the instruction starts to execute till the result is ready. The time it takes to execute a dependence chain is the sum of the latencies of all instructions in the chain.

The throughput of an instruction is the maximum number of instructions of the same kind that can be executed per clock cycle if the instructions are independent. I prefer to list the reciprocal throughputs because this makes it easier to compare latency and throughput. The reciprocal throughput is the average time it takes from an instruction starts to execute till another independent instruction of the same type can start to execute, or the number of clock cycles per instruction in a series of independent instructions of the same kind. For example, floating point addition on a PM processor has a latency of 3 clock cycles and a reciprocal throughput of 1 clock per instruction. This means that the processor uses 3 clock cycles per addition if each addition depends on the result of the preceding addition, but only 1 clock cycle per addition if the additions are independent.

Manual 4: "Instruction tables" contains detailed lists of latencies and throughputs for almost all instructions on many different microprocessors from Intel and AMD.

The following list shows some typical values.

Instruction	Typical latency	Typical reciprocal throughput
Integer move	1	0.33-0.5
Integer addition	1	0.33-0.5
Integer Boolean	1	0.33-1
Integer shift	1	0.33-1
Integer multiplication	3-10	1-2
Integer division	39-79	20-40
Floating point addition	3-6	1
Floating point multiplication	4-8	1-2
Floating point division	20-45	20-45
Integer vector addition (XMM)	1-2	1-2
Integer vector multiplication (XMM)	3-7	2

Floating point vector addition (XMM)	3-5	2
Floating point vector multiplication (XMM)	5-7	2-4
Floating point vector division (XMM)	30-69	30-69
Memory read (cached)	3-4	0.5-1
Memory write (cached)	3-4	1
Jump or call	0	1-2

9.5 Break dependence chains

In order to take advantage of out-of-order execution, you have to avoid long dependence chains. Consider the following C++ example, which calculates the sum of 100 numbers:

```
// Example 9.3a, Loop-carried dependence chain
double list[100], sum = 0.;
for (int i = 0; i < 100; i++) sum += list[i];
```

This code is doing a hundred additions, and each addition depends on the result of the preceding one. This is a loop-carried dependence chain. A loop-carried dependence chain can be very long and completely prevent out-of-order execution for a long time. Only the calculation of `i` can be done in parallel with the floating point addition.

Assuming that floating point addition has a latency of 4 and a reciprocal throughput of 1, the optimal implementation will have four accumulators so that we always have four additions in the pipeline of the floating point adder. In C++ this will look like:

```
// Example 9.3b, Multiple accumulators
double list[100], sum1 = 0., sum2 = 0., sum3 = 0., sum4 = 0.;
for (int i = 0; i < 100; i += 4) {
   sum1 += list[i];
   sum2 += list[i+1];
   sum3 += list[i+2];
   sum4 += list[i+3];
}
sum1 = (sum1 + sum2) + (sum3 + sum4);
```

Here we have four dependence chains running in parallel and each dependence chain is one fourths as long as the original one. See page 55 for examples of assembly code for loops with multiple accumulators.

It may not be possible to obtain the theoretical maximum throughput. The more parallel dependence chains there are, the more difficult is it for the CPU to schedule and reorder the uops optimally. It is particularly difficult if the dependence chains are branched or entangled.

Dependence chains occur not only in loops but also in linear code. Such dependence chains can also be broken up. For example, `y = a + b + c + d` can be changed to `y = (a + b) + (c + d)` so that the two parentheses can be calculated in parallel.

Sometimes there are different possible ways of implementing the same calculation with different latencies. For example, you may have the choice between a branch and a conditional move. The branch has the shortest latency, but the conditional move avoids the risk of branch misprediction (see page 57). Which implementation is optimal depends on how predictable the branch is and how long the dependence chain is.

A common way of setting a register to zero is `XOR EAX,EAX` or `SUB EAX,EAX`. Some processors recognize 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. These instructions are useful for

breaking an unnecessary dependence. The following list summarizes which instructions are recognized as breaking dependence when source and destination are the same, on different processors:

Instruction	P3 and earlier	P4	PM	AMD64
XOR	-	x	-	x
SUB	-	x	-	x
SBB	-	-	-	x
PXOR	-	x	-	x
XORPS, XORPD	-	-	-	x
PANDN, PSUB	-	-	-	-
SUBPS, SUBPD	-	-	-	-

You cannot break a dependence by part of a register. For example `XOR AX,AX` does not break a dependence. The `SBB EAX,EAX` is of course dependent on the carry flag, even when it does not depend on `EAX`.

You may also use these instructions for breaking dependences on the flags. For example, rotate instructions have a false dependence on the flags in P4. This can be removed in the following way:

```
; Example 9.4, Break dependence on flags
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 that 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 dependences. So you may replace `SUB EDX,EDX` in the above example with `CMP ESP,ESP`. You cannot use `CLC` for breaking dependences on the carry flag.

9.6 Jumps and calls

Jumps, branches, calls and returns do not necessarily add to the execution time of a code because they will typically be executed in parallel with something else. The number of jumps etc. should nevertheless be kept at a minimum in critical code for the following reasons:

- Instruction prefetching is less efficient after a jump, especially if the target is near the end of a 16-byte block.
- The code cache becomes fragmented and less efficient when jumping around between noncontiguous subroutines.
- Microprocessors with a trace cache are likely to store multiple instances of the same code in the trace cache when the code contains many jumps.
- The branch target buffer (BTB) can store only a limited number of jump target addresses. A BTB miss costs many clock cycles.
- Conditional jumps are predicted according to advanced branch prediction mechanisms. Mispredictions are expensive, as explained below.
- On most processors, branches can interfere with each other in the global branch pattern history table and the branch history register. One branch may therefore

reduce the prediction rate of other branches.

- Returns are predicted by the use of a return stack buffer, which can only hold a limited number of return addresses, typically 8 or 16.

- Indirect jumps and indirect calls are poorly predicted on most processors.

All modern CPU's have an execution pipeline that contains stages for instruction prefetching, decoding, register renaming, uop reordering and scheduling, execution, retirement, etc. The number of stages in the pipeline range from 12 to 22, depending on the specific microarchitecture. When a branch instruction is fed into the pipeline then the CPU doesn't know for sure which instruction is the next one to fetch into the pipeline. It takes at least 12 more clock cycles before the branch instruction is executed so that it is known with certainty which way the branch goes. This uncertainty is likely to break the flow through the pipeline. Rather than waiting 12 or more clock cycles for an answer, the CPU attempts to guess which way the branch will go. The guess is based on the previous behavior of the branch. If the branch has gone the same way the last several times then it is predicted that it will go the same way this time. If the branch has alternated regularly between the two ways then it is predicted that it will continue to alternate.

If the prediction is right then the CPU has saved a lot of time by loading the right branch into the pipeline and started to decode and speculatively execute the instructions in the branch. If the prediction was wrong then the mistake is discovered after several clock cycles and the mistake has to be fixed by flushing the pipeline and discarding the results of the speculative executions. The cost of a branch misprediction ranges from 12 to more than 50 clock cycles, depending on the length of the pipeline and other details of the microarchitecture. This cost is so high that very advanced algorithms have been implemented in order to refine the branch prediction. These algorithms are explained in detail in manual 3: "The microarchitecture of Intel and AMD CPU's".

In general, you can assume that branches are predicted correctly most of the time in these cases:

- If the branch always goes the same way.

- If the branch follows a simple repetitive pattern and is inside a loop with few or no other branches.

- If the branch is correlated with a preceding branch.

- If the branch is a loop with a constant, small repeat count and few or no conditional jumps inside the loop.

The worst case is a branch that goes either way approximately 50% of the time, does not follow any regular pattern, and is not correlated with any preceding branch. Such a branch will be mispredicted 50% of the time. This is so costly that the branch should be replaced by conditional moves or a table lookup if possible.

In general, you should try to keep the number of poorly predicted branches at a minimum and keep the number of branches inside a loop at a minimum. It may be useful to split up or unroll a loop if this can reduce the number of branches inside the loop.

Indirect jumps and indirect calls are often poorly predicted. Most processors will simply predict an indirect jump or call to go the same way as it did last time. The PM (and Core2?) processors are able to recognize simple repetitive patterns for indirect jumps in some cases.

Returns are predicted by means of a so-called return stack buffer which is a first-in-last-out buffer that mirrors the return addresses pushed on the stack. A return stack buffer with 16 entries can correctly predict all returns for subroutines at a nesting level up to 16. If the subroutine nesting level is deeper than the size of the return stack buffer then the failure will be seen at the outer nesting levels, not the presumably more critical inner nesting levels. A return stack buffer size of 8 or 16 is therefore sufficient in most cases, except for deeply nested recursive functions.

The return stack buffer will fail if there is a call without a matching return or a return without a preceding call. It is therefore important to always match calls and returns. Do not jump out of a subroutine by any other means than by a `RET` instruction. And do not use the `RET` instruction as an indirect jump. Far calls should be matched with far returns.

Eliminating calls

It is possible to replace a call followed by a return by a jump:

```
; Example 9.5a, call/ret sequence (32-bit Windows)
Func1 PROC NEAR
    ...
    call  Func2
    ret
Func1 ENDP
```

This can be changed to:

```
; Example 9.5b, call+ret replaced by jmp
Func1 PROC NEAR
    ...
    jmp   Func2
Func1 ENDP
```

This modification does not conflict with the return stack buffer mechanism because the call to `Func1` is matched with the return from `Func2`. In systems with stack alignment, it is necessary to restore the stack pointer before the jump:

```
; Example 9.6a, call/ret sequence (64-bit Windows or Linux)
Func1 PROC NEAR
    sub   rsp, 8           ; Align stack by 16
    ...
    call  Func2            ; This call can be eliminated
    add   rsp, 8
    ret
Func1 ENDP
```

This can be changed to:

```
; Example 9.6b, call+ret replaced by jmp with stack aligned
Func1 PROC NEAR
    sub   rsp, 8
    ...
    add   rsp, 8           ; Restore stack pointer before jump
    jmp   Func2
Func1 ENDP
```

Eliminating unconditional jumps

It is often possible to eliminate a jump by copying the code that it jumps to. The code that is copied can typically be a loop epilog or function epilog. The following example is a function with an if-else branch inside a loop:

```
; Example 9.7a, Function with jump that can be eliminated
```

```
FuncA   PROC    NEAR
        push    ebp
        mov     ebp, esp
        sub     esp, StackSpaceNeeded
        lea     edx, EndOfSomeArray
        xor     eax, eax

Loop1:                                  ; Loop starts here
        cmp     [edx+eax*4], eax        ; if-else
        je      ElseBranch
        ...                             ; First branch
        jmp     End_If
ElseBranch:
        ...                             ; Second branch
End_If:
        add     eax, 1                  ; Loop epilog
        jnz     Loop1

        mov     esp, ebp                ; Function epilog
        pop     ebp
        ret
FuncA   ENDP
```

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

```
; Example 9.7b, Loop epilog copied to eliminate jump
FuncA   PROC    NEAR
        push    ebp
        mov     ebp, esp
        sub     esp, StackSpaceNeeded
        lea     edx, EndOfSomeArray
        xor     eax, eax

Loop1:                                  ; Loop starts here
        cmp     [edx+eax*4], eax        ; if-else
        je      ElseBranch
        ...                             ; First branch
        add     eax, 1                  ; Loop epilog for first branch
        jnz     Loop1
        jmp     AfterLoop
ElseBranch:
        ...                             ; Second branch
        add     eax, 1                  ; Loop epilog for second branch
        jnz     Loop1
AfterLoop:
        mov     esp, ebp                ; Function epilog
        pop     ebp
        ret
FuncA   ENDP
```

In example 9.7b, the unconditional jump inside the loop has been eliminated by making two copies of the loop epilog. The branch that is executed most often should come first because the first branch is fastest. The unconditional jump to `AfterLoop` can also be eliminated. This is done by copying the function epilog:

```
; Example 9.7b, Function epilog copied to eliminate jump
FuncA   PROC    NEAR
        push    ebp
        mov     ebp, esp
        sub     esp, StackSpaceNeeded
        lea     edx, EndOfSomeArray
        xor     eax, eax
```

```
Loop1:                                  ; Loop starts here
        cmp     [edx+eax*4], eax        ; if-else
        je      ElseBranch
        ...                             ; First branch
        add     eax, 1                  ; Loop epilog for first branch
        jnz     Loop1

        mov     esp, ebp                ; Function epilog 1
        pop     ebp
        ret

ElseBranch:
        ...                             ; Second branch
        add     eax, 1                  ; Loop epilog for second branch
        jnz     Loop1

        mov     esp, ebp                ; Function epilog 2
        pop     ebp
        ret
FuncA   ENDP
```

The gain that is obtained by eliminating the jump to `AfterLoop` is less than the gain obtained by eliminating the jump to `End_If` because it is outside the loop. But I have shown it here to illustrate the general method of duplicating a function epilog.

Tricks to avoid conditional jumps

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 using conditional moves or by ingenious manipulation of bits and flags, as shown in the following examples:

```
; Example 9.8, Set a bit in AL on an arbitrary flag condition:
SETcond  AL

; Example 9.9, Set all bits in EAX on an arbitrary flag condition:
sub     eax, eax
SETcond al
neg     eax
```

The carry flag is particularly useful for bit manipulation tricks:

```
; Example 9.10, Set carry flag if eax is zero:
cmp  eax, 1

; Example 9.11, Set carry flag if eax is not zero:
neg  eax

; Example 9.12, Increment eax if carry flag is set:
adc  eax, 0

; Example 9.13, Copy carry flag to all bits of eax:
sbb  eax, eax

; Example 9.14, Copy bits one by one from carry into a bit vector:
rcl  eax, 1
```

It is possible to calculate the absolute value of a signed integer without branching:

```
; Example 9.15, Calculate absolute value of eax
cdq               ; Copy sign bit of eax to all bits of edx
```

```
xor  eax, edx      ; Invert all bits if negative
sub  eax, edx      ; Add 1 if negative
```

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

```
; Example 9.16a, Find minimum of eax and ebx (unsigned):
sub  eax, ebx      ; = a-b
sbb  edx, edx      ; = (b > a) ? 0xFFFFFFFF : 0
and  edx, eax      ; = (b > a) ? a-b : 0
add  ebx, edx      ; Result is in ebx
```

Or, for signed numbers, ignoring overflow:

```
; Example 9.16b, Find minimum of eax and ebx (signed):
sub  eax, ebx      ; Will not work if overflow here
cdq                ; = (b > a) ? 0xFFFFFFFF : 0
and  edx, eax      ; = (b > a) ? a-b : 0
add  ebx, edx      ; Result is in ebx
```

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

```
; Example 9.17a, Choose between two numbers
test  eax, eax
mov   edx, ecx
cmovs edx, ebx     ; = (a < 0) ? b : c
```

Conditional moves are not very efficient on Intel processors and not available on old processors. Alternative implementations may be faster in some cases. The following example gives the same result as example 9.17a.

```
; Example 9.17b, Choose between two numbers without conditional move:
cdq                ; = (a < 0) ? 0xFFFFFFFF : 0
xor  ebx, ecx      ; b ^ c = bits that differ between b and c
and  edx, ebx      ; = (a < 0) ? (b ^ c) : 0
xor  edx, ecx      ; = (a < 0) ? b : c
```

Example 9.17b may be faster than 9.17a on processors where conditional moves are inefficient. Example 9.17b destroys the value of `ebx`.

Whether or not the various tricks listed above are faster than conditional jumps depends on how predictable a conditional jump would be, and how critical the dependence chain is. A branch is usually faster than a conditional move or other trick if the prediction of the branch is good. If a branch is poorly predicted then it is good to replace it by a conditional move or other trick as shown in the above examples.

Do not use conditional moves or other tricks if this increases the latency of a long or loop-carried dependence chain. For example, the code in example 12.13a page 95 works more efficiently with a branch inside the loop than with a conditional move, even if the branch is poorly predicted, because a conditional move will add to the latency of the loop-carried dependence chain.

It is also possible to do conditional moves in vector registers on an element-by-element basis. See page 97ff for details. There are special vector instructions for getting the minimum or maximum of two numbers. It may be faster to use vector registers than integer or floating point registers for finding minimums or maximums.

10 Optimizing for size

The code cache can hold from 8 to 32 kb of code, as explained in chapter 11 page 69. If there are problems keeping the critical parts of the code within the code cache, then you may consider reducing the size of the code. Reducing the code size can also sometimes improve the decoding of instructions. You may even want to reduce the size of the code at the cost of reduced speed 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. 64-bit code does not need more bytes for addresses than 32-bit code because it can use 32-bit RIP-relative addresses. 64-bit code may be slightly bigger than 32-bit code because of REX prefixes and other minor differences, but it may as well be smaller than 32-bit code because the increased number of registers reduces the need for memory variables.

10.1 Choosing shorter instructions

Certain instructions have short forms. `PUSH` and `POP` instructions with an integer register take only one byte. `XCHG EAX,reg32` is also a single-byte instruction and thus takes less space than a `MOV` instruction, but `XCHG` is slower than `MOV`. `INC` and `DEC` with a 32-bit register in 32-bit mode, or a 16-bit register in 16-bit mode take only one byte. The short form of `INC` and `DEC` is not available in 64-bit mode.

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

```
; Example 10.1. Instruction sizes
add eax,1000    is smaller than   add ebx,1000
mov eax,[mem]   is smaller than   mov ebx,[mem],  except in 64 bit mode.
```

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

```
; Example 10.2. Instruction sizes
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/RBP` as base pointer and no displacement and no index take one byte more than with other registers:

```
; Example 10.3. Instruction sizes
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:

```
; Example 10.4. Instruction sizes
lea eax,[ebx+ebx]   is shorter than   lea eax,[ebx*2].
```

Floating point calculations can be done either with the old floating point stack registers `ST(0)`-`ST(7)` or the new `XMM` registers on microprocessors with the SSE2 or later instruction set. The former instructions are more compact than the latter, for example:

```
; Example 10.5. Floating point instruction sizes
fadd  st(0), st(1)  ; 2 bytes
```

```
addsd xmm0, xmm1    ; 4 bytes
```

The use of ST(0)-ST(7) may be advantageous even if it requires extra FXCH instructions. There is no big difference in execution speed between the two types of floating point instructions.

10.2 Using shorter constants and addresses

Many jump addresses, data addresses, and data constants can be expressed as sign-extended 8-bit constants. This saves a lot of space. A sign-extended byte can only be used if the value is 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. The following example assumes that [mem1] and [mem2] are static memory addresses in the data segment and that the distance between them is less than 128 bytes:

```
; Example 10.6a, Static memory operands
mov ebx, [mem1]                 ; 6 bytes
add ebx, [mem2]                 ; 6 bytes
```

Reduce to:

```
; Example 10.6b, Replace addresses by pointer
mov eax, offset mem1            ; 5 bytes
mov ebx, [eax]                  ; 2 bytes
add ebx, [eax] + (mem2 - mem1)  ; 3 bytes
```

In 64-bit mode you need to replace `mov eax,offset mem1` with `lea rax,[mem1]`, which is one byte longer. The advantage of using a pointer obviously increases if you can use the same pointer many times. Storing data on the stack and using EBP or ESP as pointer will thus make the code smaller than if you use static memory locations and absolute addresses, provided of course that the data are within +/-127 bytes of the pointer. Using PUSH and POP to write and read temporary integer 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:

```
; Example 10.7, Sign-extended operands
push 200                    ; 5 bytes
push 100                    ; 2 bytes, sign extended

add ebx, 128                ; 6 bytes
sub ebx, -128               ; 3 bytes, sign extended
```

The only instructions with an immediate operand that do not have a short form with a sign-extended 8-bit constant are MOV, CALL and RET. Shorter alternatives for `MOV register, constant` are therefore useful. Examples:

```
; Example 10.8, Loading constants into 32-bit registers
mov eax, 0                  ; 5 bytes
sub eax, eax                ; 2 bytes

mov eax, 1                  ; 5 bytes
sub eax, eax / inc eax      ; 3 bytes
```

```
push 1 / pop eax            ; 3 bytes

mov eax, -1                 ; 5 bytes
or  eax, -1                 ; 3 bytes
```

You may also consider reducing the size of static data. Obviously, an array can be made smaller by using a smaller data size for the elements. For example 16-bit integers instead of 32-bit integers if the data are sure to fit into the smaller data size. The code for accessing 16-bit integers is slightly bigger than for accessing 32-bit integers, but the increase in code size is small compared to the decrease in data size for a large array.

10.3 Reusing constants

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:

```
; Example 10.9a, Loading 32-bit constants
mov [mem1], 200             ; 10 bytes
mov [mem2], 201             ; 10 bytes
mov eax, 100                ;  5 bytes
mov ebx, 150                ;  5 bytes
```

Replace with:

```
; Example 10.9b, Reuse constants
mov eax, 200                ;  5 bytes
mov [mem1], eax             ;  5 bytes
inc eax                     ;  1 byte
mov [mem2], eax             ;  5 bytes
sub eax, 101                ;  3 bytes
lea ebx, [eax+50]           ;  3 bytes
```

10.4 Constants in 64-bit mode

In 64-bit mode, there are three ways to move a constant into a 64-bit register: with a 64-bit constant, with a 32-bit sign-extended constant, and with a 32-bit zero-extended constant:

```
; Example 10.10, Loading constants into 64-bit registers
mov rax, 123456789abcdef0h  ; 10 bytes (64-bit constant)
mov rax, -100               ;  7 bytes (32-bit sign-extended)
mov eax, 100                ;  5 bytes (32-bit zero-extended)
```

Some assemblers use the sign-extended version rather than the shorter zero-extended version, even when the constant is within the range that fits into a zero-extended constant. You can force the assembler to use the zero-extended version by specifying a 32-bit destination register. Writes to a 32-bit register are always zero-extended into the 64-bit register.

10.5 Addresses and pointers in 64-bit mode

64-bit code should preferably use 64-bit register size for base and index in addresses, and 32-bit register size for everything else. Example:

```
; Example 10.11, 64-bit versus 32-bit registers
mov eax, [rbx + 4*rcx]
inc rcx
```

Here, you can save one byte by changing `inc rcx` to `inc ecx`. This will work because the value of the index register is certain to be less than 2^{32}. The base pointer however, may be bigger than 2^{32} in some systems so you can't replace `add rbx,4` by `add ebx,4`. Never use 32-bit registers as base or index inside the square bracket in 64-bit mode.

The rule of using 64-bit registers inside the square bracket of an indirect address and 32-bit registers everywhere else also applies to the `LEA` instruction. Examples:

```
; Example 10.12. LEA in 64-bit mode
lea eax, [ebx + ecx]   ; 4 bytes (needs address size prefix)
lea eax, [rbx + rcx]   ; 3 bytes (no prefix)
lea rax, [ebx + ecx]   ; 5 bytes (address size and REX prefix)
lea rax, [rbx + rcx]   ; 4 bytes (needs REX prefix)
```

The form with 32-bit destination and 64-bit address is preferred unless a 64-bit result is needed. This version takes no more time to execute than the version with 64-bit destination. The forms with address size prefix should never be used.

An array of 64-bit pointers in a 64-bit program can be made smaller by using 32-bit image-relative pointers instead. This makes the array of pointers smaller at the cost of making the code that uses the pointers bigger since it needs to add the image base. Whether this gives a net advantage depends on the size of the array. Example:

```
; Example 10.13a. Jump-table in 64-bit mode
.data
JumpTable DQ Label1, Label2, Label3, ..

.code
mov eax, [n]                ; Index
lea rdx, JumpTable          ; Address of jump table
jmp qword ptr [rdx+rax*8]   ; Jump to JumpTable[n]
```

Implementation with image-relative pointers:

```
; Example 10.13b. Image-relative jump-table in 64-bit Windows
.data
JumpTable DD imagerel(Label1),imagerel(Label2),imagerel(Label3),..
extrn __ImageBase:byte

.code
mov eax, [n]                ; Index
lea rdx, __ImageBase        ; Image base
mov eax, [rdx+rax*4+imagerel(JumpTable)] ; Load image rel. address
add rax, rdx                ; Add image base to address
jmp rax                     ; Jump to computed address
```

Unfortunately, the current version of the Microsoft assembler (version 8.00) has a bug with the `imagerel` operator.

A shorter alternative is to use 32-bit absolute pointers. This method can be used only if there is certainty that the addresses are less than 2^{31}:

```
; Example 10.13c. 32-bit absolute jump table in 64-bit mode
; Requires that addresses < 2^31
.data
JumpTable DD Label1, Label2, Label3, .. ; 32-bit addresses

.code
mov eax, [n]                ; Index
mov eax, JumpTable[rax*4]   ; Load 32-bit address
jmp rax                     ; Jump to zero-extended address
```

In example 10.13c, the address of JumpTable is a 32-bit relocatable address which is sign-extended to 64 bits. This works if the address is less than 2^{31}. The addresses of Label1, etc., are zero-extended, so this will work if the addresses are less than 2^{32}. The method of example 10.13c can be used if there is certainty that the image base plus the program size is less than 2^{31}, which will be true in most cases for application programs (see page 16). You may insert a check in the start of the program to generate an error message if the image base is too big. For DLL's, this check should be in the initialization procedure.

It is even possible to replace the 64-bit or 32-bit pointers with 16-bit offsets relative to a suitable reference point:

```
; Example 10.13d. 16-bit offsets to a reference point
.data
JumpTable DW  0, Label2-Label1, Label3-Label1, ..

.code
mov eax, [n]                ; Index
lea rdx, JumpTable          ; Address of table (RIP-relative)
movzx eax, word ptr [rdx+rax*2] ; Zero-extend 16-bit offset
lea rdx, Label1             ; Use Label1 as reference point
add rax, rdx                ; Add offset to reference point
jmp rax                     ; Jump to computed address
```

Example 10.13d uses Label1 as a reference point. It works only if all labels are within the interval from Label1 to Label1 + 2^{16}. The table contains the 16-bit offsets which are zero-extended and added to the reference point. It is not possible to use JumpTable as a reference point in this example because it is impossible to calculate the distance between a label in the data segment and a label in the code segment.

The examples above show different methods for storing code pointers. The same methods can be used for data pointers. A pointer can be stored as a 64-bit absolute address, a 32-bit image-relative address, a 32-bit absolute address, or a 16-bit offset relative to a suitable reference point. The methods that use pointers relative to the image base or a reference point are only worth the extra code if there are many pointers. This is typically the case in linked lists.

10.6 Making instructions longer for the sake of alignment

There are situations where it can be advantageous to reverse the advices in the previous paragraphs in order to make instructions longer. Most important is the case where a loop entry needs to be aligned (see p. 73). Rather than inserting NOP's to align the loop entry label you may make the preceding instructions longer than their minimum lengths in such a way that the loop entry becomes properly aligned. The longer versions of the instructions do not take longer time to execute, so we can save the time it takes to execute the NOP's.

The assembler will normally choose the shortest possible form of an instruction. It is often possible to choose a longer form of the same or an equivalent instruction. This can be done in several ways.

Use general form instead of short form of an instruction

The short forms of INC, DEC, PUSH, POP, XCHG, ADD, MOV do not have a mod-reg-r/m byte (see p. 18). The same instructions can be coded in the general form with a mod-reg-r/m byte. Examples:

```
; Example 10.14. Making instructions longer
inc  eax           ; short form. 1 byte (in 32-bit mode only)
DB   0FFH, 0C0H    ; long form of INC EAX, 2 bytes
```

```
push ebx            ; short form. 1 byte
DB   0FFH, 0F3H     ; long form of PUSH EBX, 2 bytes
```

Use an equivalent instruction that is longer

Examples:

```
; Example 10.15. Making instructions longer
inc  eax            ; 1 byte (in 32-bit mode only)
add  eax, 1         ; 3 bytes replacement for INC EAX

mov  eax, ebx       ; 2 bytes
lea  eax, [ebx]     ; can be any length from 2 to 8 bytes, see below
```

Use 4-bytes immediate operand

Instructions with a sign-extended 8-bit immediate operand can be replaced by the version with 32-bit immediate operand:

```
; Example 10.16. Making instructions longer
add  ebx,1          ; 3 bytes. Uses sign-extended 8-bit operand

add  ebx,9999       ; Use dummy constant too big for 8-bit operand
ORG  $ - 4          ; Go back 4 bytes..
DD   1              ; and overwrite the 9999 operand with a 1.
```

The above will encode `ADD EBX,1` using 6 bytes of code.

Add zero displacement to pointer

An instruction with a pointer can have a displacement of 1 or 4 bytes in 32-bit or 64-bit mode (1 or 2 bytes in 16-bit mode). A dummy displacement of zero can be used for making the instruction longer:

```
; Example 10.17. Making instructions longer
mov  eax,[ebx]      ; 2 bytes

mov  eax,[ebx+1]    ; Add 1-byte displacement. Total length = 3
ORG  $ - 1          ; Go back one byte..
DB   0              ; and overwrite displacement with 0

mov  eax,[ebx+9999]; Add 4-byte displacement. Total length = 6
ORG  $ - 4          ; Go back 4 bytes..
DD   0              ; and overwrite displacement with 0
```

The same can be done with `LEA EAX,[EBX+0]` as a replacement for `MOV EAX,EBX`.

Use SIB byte

An instruction with a memory operand can have a SIB byte (see p. 18). A SIB byte can be added to an instruction that doesn't already have one to make the instruction one byte longer. A SIB byte cannot be used in 16-bit mode or in 64-bit mode with a RIP-relative address. Example:

```
; Example 10.18. Making instructions longer
mov  eax, [ebx]          ; Length = 2 bytes
DB   8BH, 04H, 23H       ; Same with SIB byte. Length = 3 bytes
DB   8BH, 44H, 23H, 00H  ; With SIB byte and displacement. 4 bytes
```

Use prefixes

An easy way to make an instruction longer is to add unnecessary prefixes. All instructions with a memory operand can have a segment prefix. The DS segment prefix is rarely needed, but it can be added without changing the meaning of the instruction:

```
; Example 10.19. Making instructions longer
DB   3EH            ; DS segment prefix
mov  eax,[ebx]      ; prefix + instruction = 3 bytes
```

All instructions with a memory operand can have a segment prefix, including `LEA`. It is actually possible to add a segment prefix even to instructions without a memory operand. Such meaningless prefixes are simply ignored. But there is no absolute guarantee that the meaningless prefix will not have some meaning on future processors. For example, the P4 uses segment prefixes on branch instructions as branch prediction hints. The probability is very low, I would say, that segment prefixes will have any adverse effect on future processors for instructions that *could* have a memory operand, i.e. instructions with a mod-reg-r/m byte.

CS, DS, ES and SS segment prefixes have no effect in 64-bit mode, but they are still allowed, according to AMD64 Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions, 2003.

In 64-bit mode, you can also use an empty REX prefix to make instructions longer:

```
; Example 10.20. Making instructions longer
DB   40H            ; empty REX prefix
mov  eax,[rbx]      ; prefix + instruction = 3 bytes
```

Empty REX prefixes can safely be applied to almost all instructions in 64-bit mode that do not already have a REX prefix, except instructions that use `AH`, `BH`, `CH` or `DH`. REX prefixes cannot be used in 32-bit or 16-bit mode. A REX prefix must come after any other prefixes. Do not use more than one REX prefix.

AMD's optimization manual recommends the use of up to three operand size prefixes (66H) as fillers. But this prefix can only be used on instructions that are not affected by this prefix, i.e. `NOP` and floating point `st(x)` instructions. Segment prefixes are more widely applicable and have the same effect - or rather lack of effect.

It is possible to add multiple identical prefixes to any instruction as long as the total instruction length does not exceed 15. For example, you can have an instruction with two or three DS segment prefixes. But instructions with multiple prefixes may take extra time to decode. Do not use more than one prefix on 32-bit Intel processors, two prefixes on 64-bit Intel processors, or three prefixes on 64-bit AMD processors, including any necessary prefixes such as an operand size prefix.

It is not a good idea to use address size prefixes as fillers because this may slow down instruction decoding.

Do not place dummy prefixes immediately before a jump label to align it:

```
; Example 10.21. Wrong way of making instructions longer
L1:  mov  ecx,1000
     DB   3EH           ; DS segment prefix. Wrong!
L2:  mov  eax,[esi]     ; Executed both with and without prefix
```

In this example, the `MOV EAX,[ESI]` instruction will be decoded with a DS segment prefix when we come from `L1`, but without the prefix when we come from `L2`. This works in principle, but some microprocessors remember where the instruction boundaries are, and

such processors will be confused when the same instruction begins at two different locations. There may be a performance penalty for this.

It is recommended to check hand-coded instructions with a debugger or disassembler to make sure they are correct.

11 Optimizing memory access

Reading from the level-1 cache takes approximately 3 clock cycles. Reading from the level-2 cache takes in the order of magnitude of 10 clock cycles. And reading from main memory takes in the order of magnitude of 100 clock cycles. The access time is even longer if a DRAM page boundary is crossed, and extremely long if the memory area has been swapped to disk. I cannot give exact access times here because it depends on the hardware configuration and the figures keep changing with the fast technological development.

However, it is obvious from these numbers that caching of code and data is extremely important for the performance. If the code has many cache misses, and each cache miss costs more than a hundred clock cycles, then this can be a very serious bottleneck for the performance.

More advices on how to organize data for optimal caching are given in manual 1: "Optimizing software in C++". Processor-specific details are given in manual 3: "The microarchitecture of Intel and AMD CPU's" and in Intel's "IA-32 Intel® Architecture Optimization Reference Manual" and AMD's "Software Optimization Guide for AMD64 Processors".

11.1 How caching works

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 accessed in just a few clock cycles. 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 11.1. Level-1 cache contention
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 we 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`, we find that the cache line that held the value for `EAX` has now been discarded, so the processor takes the least recently used line, which is the one holding the `EBX` value, and so on. We 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 manual 4: "Instruction tables". The performance penalty for level-1 cache line contention can be quite considerable on older microprocessors, but on newer processors such as the P4 we loose only a few clock cycles because the data are likely to be prefetched from the level-2 cache, which is accessed quite fast through a full-speed 256 bit 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 we 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. We can take advantage of this by arranging data items that are used near each other together into aligned blocks of 64 bytes of memory.

The level-1 code cache works in the same way as the data cache, except on processors with a trace cache (see below). The level-2 cache is usually shared between code and data.

11.2 Trace cache

The P4 and P4E processors have a trace cache instead of a code cache. The trace cache stores the code after it has been translated to micro-operations (uops) while a normal code cache stores the raw code without translation. The trace cache makes the design more RISC-like and removes the bottleneck of instruction decoding. Another difference between a code cache and a trace cache is that the trace cache attempts to store the code in the order in which it is executed rather than the order in which it occurs in memory. This reduces the number of jumps in the trace cache.

The main disadvantage of a trace cache is that the code takes more space in a trace cache than in a code cache, because uops take more space than CISC code and because the same code may occur in several traces. The trace cache is therefore most advantageous when the critical hot spot of the program is a small innermost loop that fits into the trace cache. The trace cache is not an advantage when the critical part of the code is distributed over so many functions and loops that it can fill up the trace cache.

11.3 Alignment of data

All data in RAM should be aligned at addresses divisible by a power of 2 according to this scheme:

Operand size	Alignment
1 (byte)	1
2 (word)	2
4 (dword)	4
6 (fword)	8
8 (qword)	8
10 (tbyte)	16
16 (oword, xmmword)	16

The following example illustrates alignment of static data.

```
; Example 11.2, alignment of static data
.data
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  ?, ?

.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.

All microprocessors have a penalty of several clock cycles when accessing misaligned data that cross a cache line boundary. AMD processors also have a penalty when misaligned data cross an 8-byte boundary, and some early Intel processors (P1, PMMX) have a penalty for misaligned data crossing a 4-byte boundary. Most processors have a penalty when reading a misaligned operand shortly after writing to the same operand.

Reading and writing misaligned 16-byte XMM operands is only possible with the instructions MOVDQU, MOVUPS, MOVUPD, LDDQU, and this is very inefficient. Aligning XMM operands is therefore necessary.

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 11.3a, Explicit alignment of stack (32-bit Windows)
_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
        mov     eax, [ebp+8]       ; Function parameter = array
        movdqu  xmm0,[eax]         ; Load from unaligned array
        movdqa  [esp],xmm0         ; Store in aligned space
        call    SomeOtherFunction  ; Call some other function
        ...
        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.

All 64-bit operating systems, and some 32-bit operating systems (Mac OS and later versions of Linux) keep the stack aligned by 16 at all CALL instructions. This eliminates the need for the AND instruction and the frame pointer. It is necessary to propagate this alignment from one CALL instruction to the next by proper adjustment of the stack pointer in each function:

```
; Example 11.3b, Propagate stack alignment (32-bit Linux)
FuncWithAlign PROC NEAR
        sub     esp, 28            ; Allocate space for local data
        mov     eax, [esp+32]      ; Function parameter = array
        movdqu  xmm0,[eax]         ; Load from unaligned array
        movdqa  [esp],xmm0         ; Store in aligned space
        call    SomeOtherFunction  ; This call must be aligned
        ...
        ret
FuncWithAlign ENDP
```

In example 11.3b we are relying on the fact that the stack pointer is aligned by 16 before the call to FuncWithAlign. The CALL FuncWithAlign instruction (not shown here) has pushed the return address on the stack, whereby 4 is subtracted from the stack pointer. We have to subtract another 12 from the stack pointer before it is aligned by 16 again. The 12 is not enough for the local variable that needs 16 bytes so we have to subtract 28 to keep the stack pointer aligned by 16. 4 for the return address + 28 = 32, which is divisible by 16. Remember to include any PUSH instructions in the calculation. If, for example, there had been one PUSH instruction in the function prolog then we would subtract 24 from ESP to keep it aligned by 16. Example 11.3b needs to align the stack for two reasons. The MOVDQA instruction needs an aligned operand, and the CALL SomeOtherFunction needs to be aligned in order to propagate the correct stack alignment to SomeOtherFunction.

The principle is the same in 64-bit mode:

```
; Example 11.3c, Propagate stack alignment (64-bit Linux)
FuncWithAlign PROC
        sub     rsp, 24            ; Allocate space for local data
        mov     rax, rdi           ; Function parameter rdi = array
        movdqu  xmm0,[rax]         ; Load from unaligned array
        movdqa  [rsp],xmm0         ; Store in aligned space
        call    SomeOtherFunction  ; This call must be aligned
        ...
        ret
FuncWithAlign ENDP
```

Here, the return address takes 8 bytes and we subtract 24 from RSP, so that the total amount subtracted is 8 + 24 = 32, which is divisible by 16. Every PUSH instruction subtracts 8 from RSP in 64-bit mode.

Alignment issues are also important when mixing C++ and assembly language. Consider this C++ structure:

```
// Example 11.4a, C++ structure
struct abcd {
   unsigned char a;    // takes 1 byte storage
   int b;              // 4 bytes storage
   short int c;        // 2 bytes storage
   double d;           // 8 bytes storage
} x;
```

Most compilers (but not all) will insert three empty bytes between a and b, and six empty bytes between c and d in order to give each element its natural alignment. You may change the structure definition to:

```
// Example 11.4b, C++ structure
struct abcd {
   double d;           // 8 bytes storage
   int b;              // 4 bytes storage
   short int c;        // 2 bytes storage
   unsigned char a;    // 1 byte storage
   char unused[1];     // fill up to 16 bytes
} x;
```

This has several advantages: The implementation is identical on compilers with and without automatic alignment, the structure is easily translated to assembly, all members are properly aligned, and there are fewer unused bytes. The extra unused character in the end makes sure that all elements in an array of structures are properly aligned.

11.4 Alignment of code

Microprocessors like Intel PM and AMD processors fetch code in aligned 16-byte blocks. If an important subroutine entry or jump label happens to be near the end of a 16-byte block then the microprocessor will only get a few useful bytes of code when fetching that block of code. It may have to fetch the next 16 bytes too before it can decode the first instructions after the label. This can be avoided by aligning important subroutine entries and loop entries by 16. Aligning by 8 will assure that at least 8 bytes of code can be loaded with the first instruction fetch, which may be sufficient if the instructions are small. We may align subroutine entries by the cache line size (typically 64 bytes) if the subroutine is part of a critical hot spot and the preceding code is unlikely to be executed in the same context.

A disadvantage of code alignment is that some cache space is lost to empty spaces before the aligned code entries.

In most cases, the effect of code alignment is minimal. So my recommendation is to align code only in the most critical cases like critical subroutines and critical innermost loops.

Aligning a subroutine entry is as simple as putting as many NOP's as needed before the subroutine entry to make the address divisible by 8, 16, 32 or 64, as desired. The assembler does this with the ALIGN directive. The NOP's that are inserted will not slow down the performance because they are never executed.

It is more problematic to align a loop entry because the preceding code is also executed. It may require up to 15 NOP's to align a loop entry by 16. These NOP's will be executed before the loop is entered and this will cost processor time. It is more efficient to use longer instructions that do nothing than to use a lot of single-byte NOP's. The best modern assemblers will do just that and use instructions like `MOV EAX,EAX` and `LEA EBX,[EBX+00000000H]` to fill the space before an `ALIGN nn` statement. The LEA instruction is particularly flexible. It is possible to give an instruction like `LEA EBX,[EBX]` any length from 2 to 8 by variously adding a SIB byte, a segment prefix and an offset of one or four bytes of zero. Don't use a two-byte offset in 32-bit mode as this will slow down decoding. And don't use more than one prefix because this will slow down decoding on 32-bit Intel processors.

Using an instruction like `LEA EBX,[EBX+0]` as a NOP has the disadvantage that it has a false dependence on EBX. If this causes unwanted dependencies then you may alternatively use the multi-byte NOP instruction which has the code `0FH,1FH,mod-000-rm`. This code can have any combination of offset, SIB byte, and prefix without doing anything. This multi-byte

NOP instruction is available in Intel processors with conditional move or SSE and in AMD processors K7 and later.

A particularly efficient filler is FXCH ST(0) because this instruction is resolved by register renaming without going to any execution port on most modern Intel processors. Remember that FXCH cannot be used in code that uses MMX registers.

A more efficient way of aligning a loop entry is to code the preceding instructions in ways that are longer than necessary. In most cases, this will not add to the execution time, but possibly to the instruction fetch time. See page 66 for details on how to code instructions in longer versions.

The most efficient way to align an innermost loop is to move the preceding subroutine entry. The following example shows how to do this:

```
; Example 11.5, Aligning loop entry
ALIGN 16
X1 = 9                              ; Replace value with whatever X2 is.
DB (-X1 AND 0FH) DUP (90H)          ; Insert calculated number of NOP's.
INNERFUNCTION PROC NEAR             ; This address will be adjusted
    mov  eax,[esp+4]
    mov  ecx,10000
INNERLOOP:                          ; Loop entry will be aligned by 16
X2 = INNERLOOP - INNERFUNCTION      ; This value is needed above
.ERRNZ X1 NE X2                     ; Make error message if X1 != X2
    ; ...
    sub  ecx, 1
    jnz  INNERLOOP
    ret
INNERFUNCTION ENDP
```

This code looks awkward because currently available assemblers cannot resolve the forward reference to a difference between two labels in this case. X2 is the distance from INNERFUNCTION to INNERLOOP. X1 must be manually adjusted to the same value as X2 by looking at the assembly listing. The .ERRNZ line will generate an error message from the assembler if X1 and X2 are different. The number of bytes to insert before INNERFUNCTION in order to align INNERLOOP by 16 is ((-X1) modulo 16). The modulo 16 is calculated here by AND'ing with 15. The DB line calculates this value and inserts the appropriate number of NOP's with opcode 90H.

INNERLOOP is aligned here by misaligning INNERFUNCTION. The cost of misaligning INNERFUNCTION is negligible compared to the gain by aligning INNERLOOP because the latter label is jumped to 10000 times as often.

11.5 Organizing data for improved caching

The caching of data works best if critical data are contained in a small contiguous area of memory. The best place to store critical data is on the stack. The stack space that is allocated by a subroutine is released when the subroutine returns. The same stack space is then reused by the next subroutine that is called. Reusing the same memory area gives the optimal caching. Variables should therefore be stored on the stack rather than in the data segment when possible.

Floating point constants are typically stored in the data segment. This is a problem because it is difficult to keep the constants used by different subroutines contiguous. An alternative is to store the constants in the code. In 64-bit mode it is possible to load a double precision constant via an integer register to avoid using the data segment. Example:

```
; Example 11.6a. Loading double constant from data segment
.data
C1  DQ  SomeConstant

.code
movsd xmm0, C1
```

This can be changed to:

```
; Example 11.6b. Loading double constant from register (64-bit mode)
.code
mov rax, SomeConstant
movq xmm0, rax      ; Some assemblers use 'movd' for this instruction
```

See page 104 for various methods of generating constants without loading data from memory. This is advantageous if data cache misses are expected, but not if data caching is efficient.

Constant tables are typically stored in the data segment. It may be advantageous to copy such a table from the data segment to the stack outside the innermost loop if this can improve caching inside the loop.

Static variables are variables that are preserved from one function call to the next. Such variables are typically stored in the data segment. It may be a better alternative to encapsulate the function together with its data in a class. The class may be declared in the C++ part of the code even when the member function is coded in assembly.

Data structures that are too large for the data cache should preferably be accessed in a linear, forward way for optimal prefetching and caching. Non-sequential access can cause cache line contentions if the stride is a high power of 2. Manual 1: "Optimizing software in C++" contains examples of how to avoid access strides that are high powers of 2.

11.6 Organizing code for improved caching

The caching of code works best if the critical part of the code is contained within a contiguous area of memory no bigger than the code cache. Avoid scattering critical subroutines around at random memory addresses. Rarely accessed code such as error handling routines should be kept separate from the critical hot spot code.

It may be useful to split the code segment into different segments for different parts of the code. For example, you may make a hot code segment for the code that is executed most often and a cold code segment for code that is not speed-critical.

Alternatively, you may control the order in which modules are liked, so that modules that are used in the same part of the program are linked at addresses near each other.

Dynamic linking of function libraries (DLL's or shared objects) make code caching less efficient. Dynamic link libraries are typically loaded at round memory addresses. This can cause cache contentions if the distances between multiple DLL's are divisible by high powers of 2.

11.7 Cache control instructions

Memory writes are more expensive than reads when cache misses occur in a write-back cache. A whole cache line has to be read from memory, modified, and written back in case of a cache miss. This can be avoided by using the nontemporal write instructions `MOVNTI`, `MOVNTQ`, `MOVNTDQ`, `MOVNTPD`, `MOVNTPS`. These instructions should be used when writing to a memory location that is unlikely to be cached and unlikely to be read from again before the

would-be cache line is evicted. MOVNTQ and MOVNTPS require the SSE instruction set, MOVNTI, MOVNTDQ and MOVNTPD require the SSE2 instruction set. Don't mix nontemporal writes with normal writes or reads to the same memory area (i.e. the area that would belong to the same cache line).

Explicit data prefetching with the PREFETCH instructions can sometimes improve cache performance, but in most cases the automatic prefetching is sufficient.

12 Loops

The critical hot spot of a CPU-intensive program is almost always a loop. The clock frequency of modern computers is so high that even the most time-consuming instructions, cache misses and inefficient exceptions are finished in a fraction of a microsecond. The delay caused by inefficient code is only noticeable when repeated millions of times. Such high repeat counts are likely to be seen only in the innermost level of a series of nested loops. The things that can be done to improve the performance of loops is discussed in this chapter.

12.1 Minimize loop overhead

The loop overhead is the instructions needed for jumping back to the beginning of the loop and to determine when to exit the loop. Optimizing these instructions is a fairly general technique that can be applied in many situations. Optimizing the loop overhead is not needed, however, if some other bottleneck is limiting the speed. See page 80ff for a description of possible bottlenecks in a loop.

A typical loop in C++ may look like this:

```
// Example 12.1a. Typical for-loop in C++
for (int i = 0; i < n; i++) {
   // (loop body)
}
```

Without optimization, the assembly implementation will look like this:

```
; Example 12.1b. For-loop, not optimized
    mov  ecx, n        ; Load n
    xor  eax, eax      ; i = 0
LoopTop:
    cmp  eax, ecx      ; i < n
    jge  LoopEnd       ; Exit when i >= n
    ; (loop body)      ; Loop body goes here
    add  eax, 1        ; i++
    jmp  LoopTop       ; Jump back
LoopEnd:
```

Don't use the inc instruction for adding 1 to the loop counter. The inc instruction has a problem with writing to only part of the flags register, which makes it less efficient than the add instruction.

The most important problem with the loop in example 12.1b is that there are two jump instructions. We can eliminate one jump from the loop by putting the branch instruction in the end:

```
; Example 12.1c. For-loop with branch in the end
    mov  ecx, n        ; Load n
    test ecx, ecx      ; Test n
```

```
    jng  LoopEnd       ; Skip if n <= 0
    xor  eax, eax      ; i = 0
LoopTop:
    ; (loop body)      ; Loop body goes here
    add  eax, 1        ; i++
    cmp  eax, ecx      ; i < n
    jl   LoopTop       ; Loop back if i < n
LoopEnd:
```

Now we have got rid of the unconditional jump instruction in the loop by putting the loop exit branch in the end. We have to put an extra check before the loop to cover the case where the loop should run zero times. Without this check, the loop would run one time when `n` = 0.

The method of putting the loop exit branch in the end can even be used for complicated loop structures that have the exit condition in the middle. Consider a C++ loop with the exit condition in the middle:

```
// Example 12.2a. C++ loop with exit in the middle
int i = 0;
while (true) {
   FuncA();                  // Upper loop body
   if (++i >= n) break;      // Exit condition here
   FuncB();                  // Lower loop body
}
```

This can be implemented in assembly by reorganizing the loop so that the exit comes in the end and the entry comes in the middle:

```
; Example 12.2b. Assembly loop with entry in the middle
    xor  eax, eax          ; i = 0
    jmp  LoopEntry         ; Jump into middle of loop
LoopTop:
    call FuncB             ; Lower loop body comes first
LoopEntry:
    call FuncA             ; Upper loop body comes last
    add  eax, 1
    cmp  eax, n
    jge  LoopTop           ; Exit condition in the end
```

The `cmp` instruction in example 12.1c and 12.2b can be eliminated if the counter ends at zero because we can rely on the `add` instruction for setting the zero flag. This can be done by counting down from `n` to zero rather counting up from zero to `n`:

```
; Example 12.3. Loop with counting down
    mov  ecx, n        ; Load n
    test ecx, ecx      ; Test n
    jng  LoopEnd       ; Skip if n <= 0
LoopTop:
    ; (loop body)      ; Loop body goes here
    sub  ecx, 1        ; n--
    jnz  LoopTop       ; Loop back if not zero
LoopEnd:
```

Now the loop overhead is reduced to just two instructions, which is the best possible. The instructions `jecxz` and `loop` should be avoided because they are less efficient.

The solution in example 12.3 is not good if `i` is needed inside the loop, for example for an array index. The following example adds 1 to all elements in an integer array:

```
; Example 12.4a. For-loop with array
    mov  ecx, n        ; Load n
```

```
    test ecx, ecx      ; Test n
    jng  LoopEnd       ; Skip if n <= 0
    xor  eax, eax      ; i = 0
    lea  esi, Array    ; Pointer to an array
LoopTop:
    ; Loop body: Add 1 to all elements in Array:
    add  dword ptr [esi+4*eax], 1
    add  eax, 1        ; i++
    cmp  eax, ecx      ; i < n
    jl   LoopTop       ; Loop back if i < n
LoopEnd:
```

The address of the start of the array is in `esi` and the index in `eax`. The index is multiplied by 4 in the address calculation because the size of each array element is 4 bytes.

It is possible to modify example 12.4a to make it count down rather than up, but the data cache is optimized for accessing data forwards, not backwards. Therefore it is better to count up through negative values from `-n` to zero. This is possible by making a pointer to the end of the array and using a negative offset from the end of the array:

```
; Example 12.4b. For-loop with negative index from end of array
    mov  ecx, n            ; Load n
    lea  esi, Array[4*ecx] ; Point to end of array
    neg  ecx               ; i = -n
    jnl  LoopEnd           ; Skip if (-n) >= 0
LoopTop:
    ; Loop body: Add 1 to all elements in Array:
    add  dword ptr [esi+4*ecx], 1
    add  ecx, 1            ; i++
    js   LoopTop           ; Loop back if i < 0
LoopEnd:
```

A slightly different solution is to multiply `n` by 4 and count from `-4*n` to zero:

```
; Example 12.4c. For-loop with neg. index multiplied by element size
    mov  ecx, n            ; Load n
    shl  ecx, 2            ; n * 4
    jng  LoopEnd           ; Skip if (4*n) <= 0
    lea  esi, Array[ecx]   ; Point to end of array
    neg  ecx               ; i = -4*n
LoopTop:
    ; Loop body: Add 1 to all elements in Array:
    add  dword ptr [esi+ecx], 1
    add  ecx, 4            ; i += 4
    js   LoopTop           ; Loop back if i < 0
LoopEnd:
```

There is no difference in speed between example 12.4b and 12.4c, but the latter method is useful if the size of the array elements is not 1, 2, 4 or 8 so that we cannot use the scaled index addressing.

The loop counter should always be an integer because floating point compare instructions are less efficient than integer compare instructions, even with the new SSE2 instruction set. Some loops have a floating point exit condition by nature. A well-known example is a Taylor expansion which is ended when the terms become sufficiently small. It may be useful in such cases to always use the worst-case maximum repeat count. The cost of repeating the loop more times than necessary is in some cases less than what is saved by avoiding the calculation of the exit condition in the loop and using an integer counter as loop control. A further advantage of this method is that the loop exit branch becomes more predictable. Even when the loop exit branch is mispredicted, the cost of the misprediction is smaller with

an integer counter because the integer instructions are likely to be executed way ahead of the slower floating point instructions so that the misprediction can be resolved much earlier.

12.2 Induction variables

If the floating point value of the loop counter is needed for some other purpose then it is better to have both an integer counter and a floating point counter. Consider the example of a loop that makes a sine table:

```
// Example 12.5a. C++ loop to make sine table
double Table[100];  int i;
for (i = 0; i < 100; i++) Table[i] = sin(0.01 * i);
```

This can be changed to:

```
// Example 12.5b. C++ loop to make sine table
double Table[100], x;  int i;
for (i = 0, x = 0.; i < 100; i++, x += 0.01) Table[i] = sin(x);
```

Here we have an integer counter `i` for the loop control and array index, and a floating point counter `x` for replacing `0.01*i`. The calculation of `x` by adding `0.01` to the previous value is much faster than converting `i` to floating point and multiplying by `0.01`. The assembly implementation looks like this:

```
; Example 12.5c. Assembly loop to make sine table
.data
align 8
M0_01   dq   0.01          ; Define constant 0.01
_Table  dq   100 dup (?)   ; Define Table

.code
    xor  eax, eax          ; i = 0
    fld  M0_01             ; Load constant 0.01
    fldz                   ; x = 0.
LoopTop:
    fld  st(0)             ; Copy x
    fsin                   ; sin(x)
    fstp _Table[eax*8]     ; Table[i] = sin(x)
    fadd st(0), st(1)      ; x += 0.01
    add  eax, 1            ; i++
    cmp  eax, 100          ; i < n
    jb   LoopTop           ; Loop
    fcompp                 ; Discard st(0) and st(1)
```

There is no need to optimize the loop overhead in this case because the speed is limited by the floating point calculations. Another possible optimization is to use a library function that calculates two or four sine values at a time in an xmm register. Such a function is found in the "Short Vector Math Library" that comes with the Intel C++ compiler (See "Integrating Fast Math Libraries for the Intel Pentium 4 Processor", www.intel.com).

The method of calculating `x` in example 12.5c by adding `0.01` to the previous value rather than multiplying `i` by `0.01` is commonly known as using an induction variable. Induction variables are useful whenever it is easier to calculate some value in a loop from the previous value than to calculate it from the loop counter. An induction variable can be integer or floating point. The most common use of induction variables is for calculating array addresses, as in example 12.4c, but induction variables can also be used for more complex expressions. Any function that is an n'th degree polynomial of the loop counter can be calculated with just n additions and no multiplications by the use of n induction variables. See manual 1: "Optimizing software in C++" for an example.

The calculation of a function of the loop counter by the induction variable method makes a loop-carried dependence chain. If this chain is too long then it may be advantageous to calculate each value from a value that is two or more iterations back.

12.3 Move loop-invariant code

The calculation of any expression that doesn't change inside the loop should be moved out of the loop.

The same applies to if-else branches with a condition that doesn't change inside the loop. Such a branch can be avoided by making two loops, one for each branch, and making a branch that chooses between the two loops.

12.4 Find the bottlenecks

There are a number of possible bottlenecks that can limit the performance of a loop. The most likely bottlenecks are:

- Cache misses and cache contentions
- Loop-carried dependence chains
- Instruction fetching
- Instruction decoding
- Instruction retirement
- Register read stalls
- Execution port throughput
- Execution unit throughput
- Suboptimal reordering and scheduling of uops
- Branch mispredictions
- Floating point exceptions and denormal operands

If one particular bottleneck is limiting the performance then it doesn't help to optimize anything else. It is therefore very important to analyze the loop carefully in order to identify which bottleneck is the limiting factor. Only when the narrowest bottleneck has successfully been removed does it make sense to look at the next bottleneck. The various bottlenecks are discussed in the following sections. All these details are processor-specific. See manual 3: "The microarchitecture of Intel and AMD CPU's" for explanation of the processor-specific details mentioned below.

Sometimes a lot of experimentation is needed in order to find and fix the limiting bottleneck. It is important to remember that a solution found by experimentation is CPU-specific and unlikely to be optimal on CPU's with a different microarchitecture.

12.5 Instruction fetch, decoding and retirement in a loop

The details about how to optimize instruction fetching, decoding, retirement, etc. is processor-specific, as mentioned on page 53.

If code fetching is a bottleneck then it is necessary to align the loop entry by 16 and reduce instruction sizes in order to minimize the number of 16-byte boundaries in the loop.

If instruction decoding is a bottleneck then it is necessary to observe the CPU-specific rules about decoding patterns. Avoid complex instructions such as `LOOP`, `JECXZ`, `LODS`, `STOS`, etc.

Register read stalls can occur in PM and similar processors. If register read stalls are likely to occur then it can be necessary to reorder instructions or to refresh registers that are read multiple times but not written to inside the loop.

Jumps and calls inside the loop should be avoided because it delays code fetching. Subroutines that are called inside the loop should be inlined if possible.

Branches inside the loop should be avoided if possible because they interfere with the prediction of the loop exit branch. However, branches should not be replaced by conditional moves if this increases the length of a loop-carried dependence chain.

If instruction retirement is a bottleneck then it may be preferred to make the total number of uops inside the loop a multiple of the retirement rate (4 for Core2, 3 for all other processors). Some experimentation is needed in this case.

12.6 Distribute uops evenly between execution units

Manual 4: "Instruction tables" contains tables of how many uops each instruction generates and which execution port each uop goes to. This information is CPU-specific, of course. It is necessary to calculate how many uops the loop generates in total and how many of these uops go to each execution port and each execution unit.

The time it takes to retire all instructions in the loop is the total number of uops divided by the retirement rate. The retirement rate is 4 uops per clock cycle for Core2 processors and 3 for all other processors. The calculated retirement time is the minimum execution time for the loop. This value is useful as a norm which other potential bottlenecks can be compared against.

The throughput for an execution port is 1 uop per clock cycle, except for simple integer instructions on P4 and P4E. AMD processors do not have execution ports. The load on a particular execution port is calculated as the number of uops that goes to this port divided by the throughput of the port. If this value exceeds the retirement time as calculated above, then this particular execution port is likely to be a bottleneck.

There may be more than one execution unit on each execution port. Most execution units have the same throughput as the execution port. If this is the case then the execution unit cannot be a narrower bottleneck than the execution port. But an execution unit can be a bottleneck in the following situations: (1) if the throughput of the execution unit is lower than the throughput of the execution port, e.g. for multiplication and division; (2) if the execution unit is accessible though more than one execution port, e.g. floating point addition on PM; and (3) on AMD processors that have no execution ports.

The load on a particular execution unit is calculated as the total number of uops going to that execution unit multiplied by the reciprocal throughput for that unit. If this value exceeds the retirement time as calculated above, then this particular execution unit is likely to be a bottleneck.

12.7 An example of analysis for bottlenecks

The way to do these calculations is illustrated in the following example, which is the so-called DAXPY algorithm used in linear algebra:

```
// Example 12.6a. C++ code for DAXPY algorithm
int i;  const int n = 100;
double X[n];  double Y[n];  double DA;
for (i = 0; i < n; i++)  Y[i] = Y[i] - DA * X[i];
```

The following implementation is for a processor with the SSE2 instruction set in 32-bit mode, assuming that `X` and `Y` are aligned by 16:

```
; Example 12.6b. DAXPY algorithm, 32-bit mode
n   = 100                   ; Define constant n (even and positive)
    mov ecx, n * 8          ; Load n * sizeof(double)
    xor eax, eax            ; i = 0
    lea esi, X              ; X must be aligned by 16
    lea edi, Y              ; Y must be aligned by 16
    movsd  xmm2, DA         ; Load DA
    shufpd xmm2, xmm2, 0    ; Get DA into both qwords of xmm2

; This loop does 2 DAXPY calculations per iteration, using vectors:
L1: movapd xmm1, [esi+eax]  ; X[i], X[i+1]
    mulpd  xmm1, xmm2       ; X[i] * DA, X[i+1] * DA
    movapd xmm0, [edi+eax]  ; Y[i], Y[i+1]
    subpd  xmm0, xmm1       ; Y[i]-X[i]*DA, Y[i+1]-X[i+1]*DA
    movapd [edi+eax], xmm0  ; Store result
    add eax, 16             ; Add size of two elements to index
    cmp eax, ecx            ; Compare with n*8
    jl L1                   ; Loop back
```

Now let's analyze this code for bottlenecks on a Pentium M processor, assuming that there are no cache misses. The CPU-specific details that I am referring to are explained in manual 3: "The microarchitecture of Intel and AMD CPU's".

We are only interested in the loop, i.e. the code after `L1`. We need to list the uop breakdown for all instructions in the loop, using the table in manual 4: "Instruction tables". The list looks as follows:

Instruction	uops fused	uops for each execution port						execution units	
		port 0	port 1	port 0 or 1	port 2	port 3	port 4	FADD	FMUL
`movapd xmm1,[esi+eax]`	2				2				
`mulpd  xmm1, xmm2`	2	2							2
`movapd xmm0,[edi+eax]`	2				2				
`subpd  xmm0, xmm1`	2			2				2	
`movapd [edi+eax],xmm0`	2					2	2		
`add eax, 16`	1			1					
`cmp eax, ecx`	1			1					
`jl L1`	1		1						
Total	13	2	1	4	4	2	2	2	2

Table 12.1. Total uops for DAXPY loop on Pentium M

The total number of uops going to all the ports is 15. The 4 uops from `movapd [edi+eax],xmm0` are fused into 2 uops so the total number of uops in the fused domain is 13. The total retirement time is 13 uops / 3 uops per clock cycle = 4.33 clock cycles.

Now, we will calculate the number of uops going to each port. There are 2 uops for port 0, 1 for port 1, and 4 which can go to either port 0 or port 1. This totals 7 uops for port 0 and port 1 combined so that each of these ports will get 3.5 uops on average per iteration. Each port has a throughput of 1 uop per clock cycle so the total load corresponds to 3.5 clock cycles. This is less than the 4.33 clock cycles for retirement, so port 0 and 1 will not be bottlenecks. Port 2 receives 4 uops per clock cycle so it is close to being saturated. Port 3 and 4 receive only 2 uops each.

The next analysis concerns the execution units. The FADD unit has a throughput of 1 uop per clock and it can receive uops from both port 0 and 1. The total load is 2 uops / 1 uop per clock = 2 clocks. Thus the FADD unit is far from saturated. The FMUL unit has a throughput of 0.5 uop per clock and it can receive uops only from port 0. The total load is 2 uops / 0.5 uop per clock = 4 clocks. Thus the FMUL unit is close to being saturated.

There is a dependence chain in the loop. The latencies are: 2 for memory read, 5 for multiplication, 3 for subtraction, and 3 for memory write, which totals 13 clock cycles. This is three times as much as the retirement time but it is not a loop-carried dependence because the results from each iteration are saved to memory and not reused in the next iteration. The out-of-order execution mechanism and pipelining makes it possible that each calculation can start before the preceding calculation is finished. The only loop-carried dependence is `add eax,16` which has a latency of only 1.

The time needed for instruction fetching can be calculated from the lengths of the instructions. The total length of the eight instructions in the loop is 30 bytes. The processor needs to fetch two 16-byte blocks if the loop entry is aligned by 16, or at most three 16-byte blocks in the worst case. Instruction fetching is therefore not a bottleneck.

Instruction decoding in the PM processor follows the 4-1-1 pattern. The pattern of (fused) uops for each instruction in the loop in example 12.6b is 2-2-2-2-2-1-1-1. This is not optimal, and it will take 6 clock cycles to decode. This is more than the retirement time, so we can conclude that instruction decoding is the bottleneck in example 12.6b. The total execution time is 6 clock cycles per iteration or 3 clock cycles per calculated `Y[i]` value. The decoding can be improved by moving one of the 1-uop instructions and by changing the sign of `xmm2` so that `movapd xmm0,[edi+eax]` and `subpd xmm0,xmm1` can be combined into one instruction `addpd xmm1,[edi+eax]`. We will therefore change the code of the loop as follows:

```
; Example 12.6c. Loop of DAXPY algorithm with improved decoding
.data
align   16
SignBit DD  0, 80000000H     ; qword with sign bit set
n   = 100                    ; Define constant n (even and positive)

.code
    mov ecx, n * 8           ; Load n * sizeof(double)
    xor eax, eax             ; i = 0
    lea esi, X               ; X must be aligned by 16
    lea edi, Y               ; Y must be aligned by 16
    movsd  xmm2, DA          ; Load DA
    xorpd  xmm2, SignBit     ; Change sign
    shufpd xmm2, xmm2, 0     ; Get -DA into both qwords of xmm2

L1: movapd xmm1, [esi+eax]   ; X[i], X[i+1]
    mulpd  xmm1, xmm2        ; X[i] * (-DA), X[i+1] * (-DA)
    addpd  xmm1, [edi+eax]   ; Y[i]-X[i]*DA, Y[i+1]-X[i+1]*DA
    add eax, 16              ; Add size of two elements to index
    movapd [edi+eax-16],xmm1 ; Address corrected for changed eax
    cmp eax, ecx             ; Compare with n*8
    jl L1                    ; Loop back
```

The number of uops in the loop is the same as before, but the decode pattern is now 2-2-4-1-2-1-1 and the decode time is reduced from 6 to 4 clock cycles so that decoding is no longer a bottleneck.

An experimental test shows that the expected improvement is not obtained. The execution time is only reduced from 6.0 to 5.6 clock cycles per iteration. There are three reasons why the execution time is higher than the expected 4.3 clocks per iteration.

The first reason is register read stalls. The reorder buffer can handle no more than three reads per clock cycle from registers that have not been modified recently. Register `esi`, `edi`, `ecx` and `xmm2` are not modified inside the loop. `xmm2` counts as two because it is implemented as two 64-bit registers. `eax` also contributes to the register read stalls because its value has enough time to retire before it is used again. This causes a register read stall in two out of three iterations. The execution time can be reduced to 5.4 by moving the `add eax,16` instruction up before the `mulpd` so that the reads of register `esi`, `xmm2` and `edi` are separated further apart and therefore prevented from going into the reorder buffer in the same clock cycle.

The second reason is retirement. The number of fused uops in the loop is not divisible by 3. Therefore, the taken jump to L1 will not always go into the first slot of the retirement station which it has to. This can sometimes cost a clock cycle.

The third reason is that the two uops for `addpd` may be issued in the same clock cycle through port 0 and port 1, respectively, though there is only one execution unit for floating point addition. This is the consequence of a bad design, as explained in manual 3: "The microarchitecture of Intel and AMD CPU's".

A solution which happens to work better is to get rid of the `cmp` instruction by using a negative index from the end of the arrays:

```
; Example 12.6d. Loop of DAXPY algorithm with negative indexes
.data
align   16
SignBit DD  0, 80000000H     ; qword with sign bit set
n   = 100                    ; Define constant n (even and positive)

.code
    mov eax, -n * 8          ; Index = -n * sizeof(double)
    lea esi, X + 8 * n       ; Point to end of array X (aligned)
    lea edi, Y + 8 * n       ; Point to end of array Y (aligned)
    movsd  xmm2, DA          ; Load DA
    xorpd  xmm2, SignBit     ; Change sign
    shufpd xmm2, xmm2, 0     ; Get -DA into both qwords of xmm2

L1: movapd xmm1, [esi+eax]   ; X[i], X[i+1]
    mulpd  xmm1, xmm2        ; X[i] * (-DA), X[i+1] * (-DA)
    addpd  xmm1, [edi+eax]   ; Y[i]-X[i]*DA, Y[i+1]-X[i+1]*DA
    movapd [edi+eax],xmm1    ; Store result
    add eax, 16              ; Add size of two elements to index
    js L1                    ; Loop back
```

This removes one uop from the loop. My measurements show an execution time for example 12.6d of 5.0 clock cycles per iteration on a PM processor. The theoretical minimum is 4. The register read stalls have disappeared because `eax` now has less time to retire before it is used again. The retirement is also improved because the number of fused uops in the loop is now 12, which is divisible by the retirement rate of 3. The problem with the floating point addition uops clashing remains and this is responsible for the extra clock cycle. This problem can only be targeted by experimentation. I found that the optimal order of the instructions has the `add` instruction immediately after the `mulpd`:

```
; Example 12.6e. Loop of DAXPY. Optimal solution for PM
.data
align   16
SignBit DD  0, 80000000H     ; qword with sign bit set
n   = 100                    ; Define constant n (even and positive)

.code
    mov eax, -n * 8          ; Index = -n * sizeof(double)
    lea esi, X + 8 * n       ; Point to end of array X (aligned)
    lea edi, Y + 8 * n       ; Point to end of array Y (aligned)
    movsd  xmm2, DA          ; Load DA
    xorpd  xmm2, SignBit     ; Change sign
    shufpd xmm2, xmm2, 0     ; Get -DA into both qwords of xmm2

L1: movapd xmm1, [esi+eax]   ;
    mulpd  xmm1, xmm2        ;
    add eax, 16              ; Optimal position of add instruction
    addpd  xmm1, [edi+eax-16]; Address corrected for changed eax
    movapd [edi+eax-16],xmm1 ; Address corrected for changed eax
    js L1                    ;
```

The execution time is now reduced to 4.0 clock cycles per iteration, which is the theoretical minimum. An analysis of the bottlenecks in the loop of example 12.6e gives the following results: The decoding time is 4 clock cycles. The retirement time is 12 / 3 = 4 clock cycles. Port 0 and 1 are used at 75% or their capacity. Port 2 is used 100%. Port 3 and 4 are used 50%. The FMUL execution unit is used at 100% of its maximum throughput. The FADD unit is used 50%. The conclusion is that the speed is limited by four equally narrow bottlenecks and that no further improvement is possible.

The fact that we found an instruction order that removes the floating point uop clashing problem is sheer luck. In more complicated cases there may not exist a solution that eliminates this problem.

12.8 Loop unrolling

A loop that does n repetitions can be replaced by a loop that repeats n / r times and does r calculations for each repetition, where r is the unroll factor. n should preferably be divisible by r.

Loop unrolling can be used for the following purposes:

- Reducing loop overhead. The loop overhead per calculation is divided by the loop unroll factor r. This is only useful if the loop overhead contributes significantly to the calculation time. There is no reason to unroll a loop if some other bottleneck limits the execution speed. For example, the loop in example 12.6e above cannot benefit from further unrolling.

- Vectorization. A loop must be rolled out by r or a multiple of r in order to use vector registers with r elements. The loop in example 12.6e is rolled out by 2 in order to use vectors of two double-precision numbers. If we had used single-precision numbers then we would have rolled out the loop by 4 and used vectors of 4 elements.

- Improve branch prediction. The prediction of the loop exit branch can be improved by unrolling the loop so much that the repeat count n / r does not exceed the maximum repeat count that can be predicted on a specific CPU.

- Improve caching. If the loop suffers from many data cache misses then it may be advantageous to schedule memory reads and writes in the way that is optimal for a

specific processor. See the optimization manual from the microprocessor vendor for details.

- Eliminate integer divisions. If the loop contains an expression where the loop counter i is divided by an integer r or the modulo of i by r is calculated, then the integer division can be avoided by unrolling the loop by r.

- Eliminate branch inside loop. If there is a branch or a `switch` statement inside the loop with a repetitive pattern of period r then this can be eliminated by unrolling the loop by r. For example, if an if-else branch goes either way every second time then this branch can be eliminated by rolling out by 2.

- Break loop-carried dependence chain. A loop-carried dependence chain can in some cases be broken up by using multiple accumulators. The unroll factor r is equal to the number of accumulators. See example 9.3b page 55.

- Reduce dependence of induction variable. If the latency of calculating an induction variable from the value in the previous iteration is so long that it becomes a bottleneck then it may be possible to solve this problem by unrolling by r and calculate each value of the induction variable from the value that is r places behind in the sequence.

- Improving uop retirement. If the number of uops in the loop is not divisible by the retirement rate and retirement is inefficient for this reason then this problem may in some cases be solved by loop unrolling.

- Parallel operations on microprocessors without out-of-order capabilities. The old P1 and PMMX processors cannot do calculations out of order, but they can execute two instructions in parallel. Rolling out by 2 can facilitate this.

- Complete unrolling. A loop is completely unrolled when $r = n$. This eliminates the loop overhead completely. Every expression that is a function of the loop counter can be replaced by constants. Every branch that depends only on the loop counter can be eliminated. See page 94 for examples.

There is a problem with loop unrolling when the repeat count n is not divisible by the unroll factor r. There will be a remainder of n modulo r extra calculations that are not done inside the loop. These extra calculations have to be done either before or after the main loop.

Getting the extra calculations right can be somewhat tricky if we are using a negative index as in example 12.6d and e. The following example shows the DAXPY algorithm again, this time with single precision and unrolled by 4. In this example n is a variable which may or may not be divisible by 4. The arrays `X` and `Y` must be aligned by 16. (The optimization that was specific to the PM processor has been omitted for the sake of clarity).

```
; Example 12.7. Unrolled Loop of DAXPY, single precision.
.data
align      16
SignBitS   DD  80000000H        ; dword with sign bit set

.code
    mov    eax, n               ; Number of calculations, n
    sub    eax, 4               ; n - 4
    lea    esi, [X + eax*4]     ; Point to X[n-4]
    lea    edi, [Y + eax*4]     ; Point to Y[n-4]
    movss  xmm2, DA             ; Load DA
    xorps  xmm2, SignBitS       ; Change sign
    shufps xmm2, xmm2, 0        ; Get -DA into all four dwords of xmm2
    neg    eax                  ; -(n-4)
```

```
    jg     L2                  ; Skip main loop if n < 4

L1: ; Main loop rolled out by 4
    movaps xmm1, [esi+eax*4]   ; Load 4 values from X
    mulps  xmm1, xmm2          ; Multiply with -DA
    addps  xmm1, [edi+eax*4]   ; Add 4 values from Y
    movaps [edi+eax*4],xmm1    ; Store 4 results in Y
    add    eax, 4              ; i += 4
    jle     L1                 ; Loop as long as <= 0

L2: ; Check for remaining calculations
    sub    eax, 4              ; = -remainder
    jns    L4                  ; Skip extra loop if remainder = 0

L3: ; Extra loop for up to 3 remaining calculations
    movss  xmm1, [esi+eax*4+16] ; Load 1 value from X
    mulss  xmm1, xmm2          ; Multiply with -DA
    addss  xmm1, [edi+eax*4+16] ; Add 1 value from Y
    movss  [edi+eax*4+16],xmm1  ; Store 1 result in Y
    add    eax, 1              ; i += 1
    js     L3                  ; Loop as long as negative
L4:
```

An alternative solution for an unrolled loop that does calculations on arrays is to extend the arrays with up to r-1 unused spaces and rounding up the repeat count n to the nearest multiple of the unroll factor r. This eliminates the need for calculating the remainder (n mod r) and for the extra loop for the remaining calculations. The unused array elements must be initialized to zero or some other valid floating point value in order to avoid denormal numbers, NAN, overflow, underflow, or any other condition that can slow down the floating point calculations. If the arrays are of integer type then the only condition you have to avoid is division by zero.

Loop unrolling should only be used when there is a reason to do so and a significant gain in speed can be obtained. Excessive loop unrolling should be avoided. The disadvantages of loop unrolling are:

- The code becomes bigger and takes more space in the code cache. This can cause code cache misses that cost more than what is gained by the unrolling. Note that the code cache misses are not detected when the loop is tested in isolation.

- The need to do extra calculations outside the unrolled loop in case n is not divisible by r makes the code more complicated and clumsy and increases the number of branches.

- The unrolled loop may need more registers, e.g. for multiple accumulators.

12.9 Optimize caching

Memory access is likely to take more time than anything else in a loop that accesses uncached memory. Data should be held contiguous if possible and accessed sequentially, as explained in chapter 11 page 69.

The number of arrays accessed in a loop should not exceed the number of read/write buffers in the microprocessor. One way of reducing the number of data streams is to combine multiple arrays into an array of structures so that the multiple data streams are interleaved into a single stream.

Some microprocessors have advanced data prefetching mechanisms. These mechanisms can detect regularities in the data access pattern such as accessing data with a particular stride. It is recommended to take advantage of such prefetching mechanisms by keeping

the number of different data streams at a minimum and keeping the access stride constant if possible. Automatic data prefetching often works better than explicit data prefetching when the data access pattern is sufficiently regular.

Explicit prefetching of data with the `prefetch` instructions may be necessary in cases where the data access pattern is too irregular to be predicted by the automatic prefetch mechanisms. A good deal of experimentation is often needed to find the optimal prefetching strategy for a program that accesses data in an irregular manner.

It is possible to put the data prefetching into a separate thread if the system has multiple CPU kernels. The Intel C++ compiler has a feature for doing this.

Data access with a stride that is a high power of 2 is likely to cause cache line contentions. This can be avoided by changing the stride or by loop blocking. See the chapter on optimizing memory access in manual 1: "Optimizing software in C++" for details.

The non-temporal write instructions are useful for writing to uncached memory that is unlikely to be accessed again soon. You may use vector instructions in order to minimize the number of non-temporal write instructions.

12.10 Parallelization

The most important way of improving the performance of CPU-intensive code is to do things in parallel. The main methods of doing things in parallel are:

- Improve the possibilities of the CPU to do out-of-order execution. This is done by breaking long dependence chains (see page 55) and distributing uops evenly between the different execution units or execution ports (see page 81).

- Use vector instructions. See chapter 13 page 96.

- Use multiple threads. See chapter 14 page 108.

Loop-carried dependence chains can be broken by using multiple accumulators, as explained on page 55. The optimal number of accumulators if the CPU has nothing else to do is the latency of the most critical instruction in the dependence chain divided by the reciprocal throughput for that instruction. For example, the latency of floating point addition on an AMD processor is 4 clock cycles and the reciprocal throughput is 1. This means that the optimal number of accumulators is 4. Example 12.8b below shows a loop that adds numbers with four floating point registers as accumulators.

```
// Example 12.8a, Loop-carried dependence chain
// (Same as example 9.3a page 55)
double list[100], sum = 0.;
for (int i = 0; i < 100; i++) sum += list[i];
```

An implementation with 4 floating point registers as accumulators looks like this:

```
; Example 12.8b, Four floating point accumulators
    lea   esi,  list             ; Pointer to list
    fld   qword ptr [esi]        ; accum1 = list[0]
    fld   qword ptr [esi+8]      ; accum2 = list[1]
    fld   qword ptr [esi+16]     ; accum3 = list[2]
    fld   qword ptr [esi+24]     ; accum4 = list[3]
    fxch  st(3)                  ; Get accum1 to top
    add   esi, 800               ; Point to end of list
    mov   eax, 32-800            ; Index to list[4] from end of list
L1:
    fadd  qword ptr [esi+eax]    ; Add list[i]
```

```
    fxch  st(1)                 ; Swap accumulators
    fadd  qword ptr [esi+eax+8] ; Add list[i+1]
    fxch  st(2)                 ; Swap accumulators
    fadd  qword ptr [esi+eax+16]; Add list[i+2]
    fxch  st(3)                 ; Swap accumulators
    add   eax, 24               ; i += 3
    js    L1                    ; Loop

    faddp st(1), st(0)          ; Add two accumulators together
    fxch  st(1)                 ; Swap accumulators
    faddp st(2), st(0)          ; Add the two other accumulators
    faddp st(1), st(0)          ; Add these sums
    fstp  qword ptr [sum]       ; Store the result
```

In example 12.8b, I have loaded the four accumulators with the first four values from `list`. Then the number of additions to do in the loop happens to be divisible by the rollout factor, which is 3. The funny thing about using floating point registers as accumulators is that the number of accumulators is equal to the rollout factor *plus one*. This is a consequence of the way the `fxch` instructions are used for swapping the accumulators. You have to play computer and follow the position of each accumulator on the floating point register stack to verify that the four accumulators are actually rotated one place after each iteration of the loop so that each accumulator is used for every fourth addition despite the fact that the loop is only rolled out by three.

The loop in example 12.8b takes 1 clock cycle per addition, which is the maximum throughput of the floating point adder. The latency of 4 clock cycles for floating point addition is taken care of by using four accumulators. The `fxch` instructions have zero latency because they are translated to register renaming on both Intel and AMD processors.

The `fxch` instructions can be avoided on processors with the SSE2 instruction set by using XMM registers instead of floating point stack registers as shown in example 12.8c. The latency of floating point vector addition is 4 on an AMD and the reciprocal throughput is 2 so the optimal number of accumulators is 2 vector registers.

```
; Example 12.8c, Two XMM vector accumulators
    lea     esi,  list          ; list must be aligned by 16
    movapd  xmm0, [esi]         ; list[0], list[1]
    movapd  xmm1, [esi+16]      ; list[2], list[3]
    add     esi,  800           ; Point to end of list
    mov     eax,  32-800        ; Index to list[4] from end of list
L1:
    addpd   xmm0, [esi+eax]     ; Add list[i],   list[i+1]
    addpd   xmm1, [esi+eax+16]  ; Add list[i+2], list[i+3]
    add     eax,  32            ; i += 4
    js      L1                  ; Loop

    addpd   xmm0, xmm1          ; Add the two accumulators together
    movhlps xmm1, xmm0          ; There is no movhlpd instruction
    addsd   xmm0, xmm1          ; Add the two vector elements
    movsd   [sum], xmm0         ; Store the result
```

Example 12.8b and 12.8c are exactly equally fast on an AMD processor because both are limited by the throughput of the floating point adder.

Example 12.8c is faster than 12.8b on an Intel Core2 processor because this processor has a 128 bits wide floating point adder that can handle a whole vector in one operation.

12.11 Analyzing dependences

A loop may have several interlocked dependence chains. Such complex cases require a careful analysis.

The next example is a Taylor expansion. As you probably know, many functions can be approximated by a Taylor 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 12.9a. Taylor expansion
.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

.code
    movsd  xmm2, [x]                  ; xmm2 = x
    movsd  xmm1, [one]                ; xmm1 = x^i
    pxor   xmm0, xmm0                 ; xmm0 = sum. init. to 0
    mov    eax,  offset coeff         ; point to c[i]
L1: 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 coeff_end      ; stop at end of list
    jb     L1
```

(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. This time we will consider a P4E processor. 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 12.1.

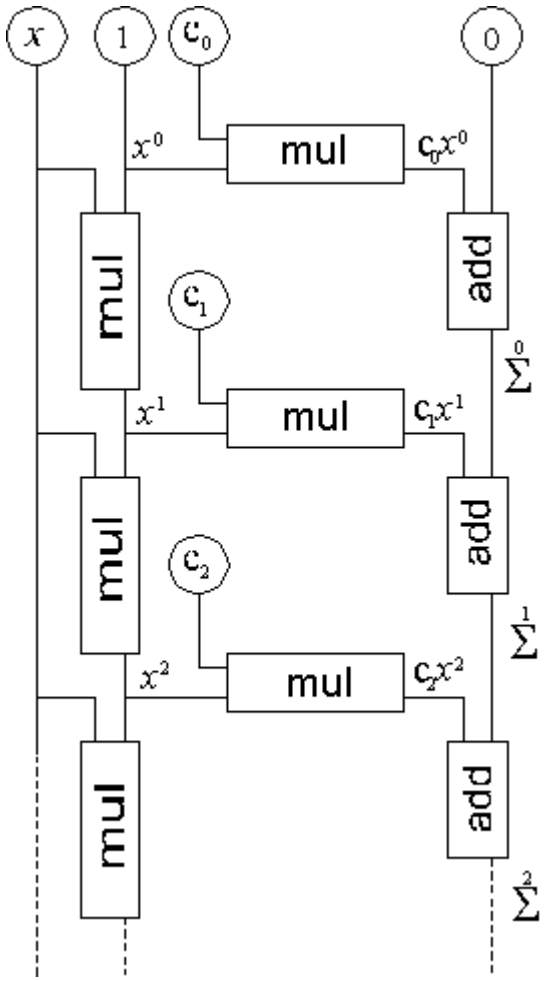

Figure 12.1: Dependences in example 12.9.

There are two continued dependence chains, one for calculating x^i and one for calculating the sum. The `mulsd` instruction has a latency of 7, while the `addsd` has a latency of 5 on P4E. The vertical multiplication chain is therefore more critical than the addition chain. The additions have to wait for $c_i x^i$, which come 7 clocks after x^i, and later than the preceding additions. If nothing else limits the performance, then we can expect this code to take 7 clocks per iteration.

Throughput appears not to be a limiting factor because the multiplication unit can start 3.5 multiplications in the 7 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 7 clock cycles per iteration as expected, but 9. 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 12.1 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 12.1 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 12.9b. Taylor expansion optimized for P4E
    movsd   xmm2, [x]                ; xmm2 = x
    movsd   xmm1, [one]              ; xmm1 = x^i
    pxor    xmm0, xmm0               ; xmm0 = sum. initialize to 0
```

```
    mov    eax,  offset coeff       ; point to c[i]
L1: 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 coeff_end    ; stop at end of list
    addsd  xmm0, xmm3               ; sum += c[i] * x^i
    jb     L1
```

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 7 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 zero 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 zero. 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 an experimental test that includes 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 worst case maximum number of terms in order to avoid the floating point comparison and to keep the repetition count constant so that the loop control branch is not mispredicted. The misprediction penalty is 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.

12.12 Loops on processors without out-of-order execution

The P1 and PMMX processors have no capabilities for out-of-order execution. Instead you have to care about pairing opportunities.

I have chosen the simple example of 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:

```
// Example 12.10a. Loop to change sign
void ChangeSign (int * A, int * B, int N) {
   for (int i = 0; i < N; i++) B[i] = -A[i];
}
```

An assembly implementation optimized for P1 may look like this:

```
; Example 12.10b
        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 DAXPY loop of example 12.6 page 82. An optimal solution for P1 is as follows:

```
; Example 12.11. DAXPY optimized for P1 and PMMX
    mov   eax, [n]                  ; number of elements
    mov   esi, [X]                  ; pointer to X
    mov   edi, [Y]                  ; pointer to Y
    xor   ecx, ecx
    lea   esi, [esi+8*eax]          ; point to end of X
    sub   ecx, eax                  ; -n
    lea   edi, [edi+8*eax]          ; point to end of Y
    jz    short L3                  ; test for n = 0
    fld   qword ptr [DA]            ; start first calc.
    fmul  qword ptr [esi+8*ecx]     ; DA * X[0]
    jmp   short L2                  ; jump into loop
L1: fld   qword ptr [DA]
    fmul  qword ptr [esi+8*ecx]     ; DA * X[i]
    fxch                            ; get old result
    fstp  qword ptr [edi+8*ecx-8]   ; store Y[i]
L2: fsubr qword ptr [edi+8*ecx]     ; subtract from Y[i]
    inc   ecx                       ; increment index
    jnz   L1                        ; loop
    fstp  qword ptr [edi+8*ecx-8]   ; store last result
L3:
```

Here we are using the loop counter as array index and counting through negative values up to zero.

Each operation begins before the previous one is finished, in order to improve calculation overlaps. Processors with out-of-order execution will do this automatically, but for processors with no out-of-order capabilities we have to do this overlapping explicitly.

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 unrolled solutions published elsewhere.

12.13 Macro loops

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 code cache if the repeat count is high.

The MASM syntax 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:

```
// Example 12.12a. Loop to make list of squares
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 12.12b. Macro loop to produce data
.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 12.12b generates no executable code, only data. The macro preprocessor will translate the above code to:

```
; Example 12.12c. Resuls of macro loop expansion
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
```

Macro loops are also useful for generating code. 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:

```
// Example 12.13a. Calculate 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;}
```

If n is known at assembly time, then the power function can be implemented using the following macro loop:

```
; Example 12.13b.
; 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
            mulsd Y, X       ; multiply Y with a power of X
         ELSE                ; If this is first time Y is used:
            movsd 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
      mulsd 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:

```
; Example 12.13c. Macro invocation
INTPOWER xmm0, xmm1, 12
```

This will be expanded to:

```
; Example 12.13d. Result of macro expansion
mulsd   xmm0, xmm0        ; x^2
mulsd   xmm0, xmm0        ; x^4
movsd   xmm1, xmm0        ; save x^4
mulsd   xmm0, xmm0        ; x^8
mulsd   xmm1, xmm0        ; x^4 * x^8 = x^12
```

This even has fewer instructions than an optimized assembly loop without unrolling. The macro can also work on vectors when `mulsd` is replaced by `mulpd` and `movsd` is replaced by `movapd`.

13 Vector 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. There are more than two hundred different SIMD instructions available. These instructions are listed in "IA-32 Intel Architecture Software Developer's Manual" vol. 2A and 2B, and in "AMD64 Architecture Programmer's Manual", vol. 4.

Multiple data can be packed into 64-bit MMX registers or 128-bit XMM 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 109. 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 109 for how to check if the use of XMM registers is enabled by the operating system.

It is advantageous to choose the smallest data size that fits the purpose in order to pack as many data as possible into one vector 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, you have to consider whether the resulting code will be faster than the simple instructions without vectors. Simple operations such as integer additions have lower throughput in SIMD registers than in general purpose registers on AMD and Intel P4 processors. The SIMD instructions are therefore only advantageous for integer additions on these processors if they can handle at least four data in parallel. Loading and storing memory operands take no longer for XMM registers than for general purpose registers. Integer shift and multiplication is faster in XMM registers than in general purpose 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 13.4 below is an example of this.

For floating-point calculations, 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 on P4/P4E, 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 XMM instructions have to be aligned by 16. See page 73 for how to align data in memory.

13.1 Conditional moves in SIMD registers

Consider this C++ code which finds the biggest values in four pairs of values:

```
// Example 13.1a. Loop to find maximums
float a[4], b[4], c[4];
for (int i = 0; i < 4; i++) {
   c[i] = a[i] > b[i] ? a[i] : b[i];
}
```

If we want to implement this code with XMM registers then we cannot use a conditional jump for the branch inside the loop because the branch condition is not the same for all four elements. Fortunately, there is a maximum instruction that does the same:

```
; Example 13.1b. Maximum in XMM
movaps   xmm0, a        ; Load a vector
maxps    xmm0, b        ; max(a,b)
movaps   c, xmm0        ; c = a > b ? a : b
```

Minimum and maximum vector instructions exist for single and double precision floats and for 8-bit and 16-bit integers. The absolute value of floating point vector elements is calculated by AND'ing out the sign bit, as shown in example 13.7 page 106. Instructions for the absolute value of integer vector elements exist in the SSE4 instruction set. The integer saturated addition vector instructions (e.g. `PADDSW`) can also be used for finding maximum or minimum or for limiting values to a specific range.

These methods are not very general, however. A more general way of doing conditional moves in vector registers is to use Boolean vector instructions. The following example is a modification of the above example where we cannot use the `MAXPS` instruction:

```
// Example 13.2a. Branch in loop
float a[4], b[4], c[4], x[4], y[4];
for (int i = 0; i < 4; i++) {
   c[i] = x[i] > y[i] ? a[i] : b[i];
}
```

The necessary conditional move is done by making a mask that consists of all 1's when the condition is true and all 0's when the condition is false. `a[i]` is AND'ed with this mask and `b[i]` is AND'ed with the inverted mask:

```
; Example 13.2b. Conditional move in XMM
movaps   xmm1, y        ; Load y vector
cmpltps  xmm1, x        ; Compare with x. xmm1 = mask for y < x
movaps   xmm0, a        ; Load a vector
andps    xmm0, xmm1     ; a AND mask
andnps   xmm1, b        ; b AND NOT mask
orps     xmm0, xmm1     ; (a AND mask) OR (b AND NOT mask)
movaps   c, xmm0        ; c = x > y ? a : b
```

The vectors that make the condition (`x` and `y` in example 13.2b) and the vectors that are selected (`a` and `b` in example 13.2b) need not be the same type. For example, `x` and `y` could be integers. But they should have the same number of elements per vector. If `a` and `b` are `double`'s with two elements per vector, and `x` and `y` are 32-bit integers with four elements per vector, then we have to duplicate each element in `x` and `y` in order to get the right size of the mask (See example 13.4b below).

Note that the AND-NOT instruction (`andnps`, `andnpd`, `pandn`) inverts the destination operand, not the source operand. This means that it destroys the mask. Therefore we must have `andps` before `andnps` in example 13.2b. If the mask is needed more than once then it may be more efficient to AND the mask with an XOR combination of `a` and `b`. This is illustrated in the next example which makes a conditional swapping of `a` and `b`:

```
// Example 13.3a. Conditional swapping in loop
float a[4], b[4], x[4], y[4], temp;
for (int i = 0; i < 4; i++) {
   if (x[i] > y[i]) {
      temp = a[i];       // Swap a[i] and b[i] if x[i] > y[i]
      a[i] = b[i];
      b[i] = temp;
   }
}
```

And now the assembly code using XMM vectors:

```
; Example 13.3b. Conditional move in XMM registers
movaps   xmm2, y        ; Load y vector
cmpltps  xmm2, x        ; Compare with x. xmm2 = mask for y < x
movaps   xmm0, a        ; Load a vector
movaps   xmm1, b        ; Load b vector
xorps    xmm0, xmm1     ; a XOR b
andps    xmm2, xmm0     ; (a XOR b) AND mask
xorps    xmm1, xmm2     ; b XOR ((a XOR b) AND mask)
xorps    xmm2, a        ; a XOR ((a XOR b) AND mask)
movaps   b, xmm1        ; (x[i] > y[i]) ? a[i] : b[i]
movaps   a, xmm2        ; (x[i] > y[i]) ? b[i] : a[i]
```

The `xorps xmm0,xmm1` instruction generates a pattern of the bits that differ between `a` and `b`. This bit pattern is AND'ed with the mask so that `xmm2` contains the bits that need to be changed if `a` and `b` should be swapped, and zeroes if they should not be swapped. The last two `xorps` instructions flip the bits that have to be changed if `a` and `b` should be swapped and leave the values unchanged if not.

The mask used for conditional moves can also be generated by copying the sign bit into all bit positions using the arithmetic shift right instruction `psrad`. This is illustrated in the next example where vector elements are raised to different integer powers. We are using the method in example 12.13a page 95 for calculating powers.

```
// Example 13.4a. Raise vector elements to different integer powers
double x[2], y[2];  unsigned int n[2];
for (int i = 0; i < 2; i++) {
   y[i] = pow(x[i],n[i]);
}
```

If the elements of `n` are equal then the simplest solution is to use a branch. But if the powers are different then we have to use conditional moves:

```
; Example 13.4b. Raise vector to power, using integer mask
.data                          ; Data segment
align 16                       ; Must be aligned
ONE  DQ  1.0, 1.0              ; Make constant 1.0
X    DQ  ?, ?                  ; x[0], x[1]
Y    DQ  ?, ?                  ; y[0], y[1]
N    DD  ?, ?                  ; n[0], n[1]

.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 initialized to 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 are both zero
    bsr       ecx,  eax      ; Compute repeat count for max(n0,n1)

L1: movdqa    xmm3, xmm2     ; Copy i
    pslld     xmm3, 31       ; Get least significant bit of i
    psrad     xmm3, 31       ; Copy to all bit positions to make mask
    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        ; Loop counter
    jns       L1             ; Repeat ecx+1 times
    movapd    [Y],  xmm1     ; Store result
```

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 13.4b in P4E: There are four continued dependence chains: `xmm0`: 7 clocks, `xmm1`: 7 clocks, `xmm2`: 4 clocks, `ecx`: 1 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 23 and 19 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.

Conditional moves in general purpose registers using `CMOVcc` and floating point registers using `FCMOVcc` are no faster than in XMM registers.

13.2 Using XMM instructions with other types of data than they are intended for

Most XMM instructions are 'typed' in the sense that they are intended for a particular type of data. For example, it doesn't make sense to use an instruction for adding integers on floating point data. But instructions that only move data around will work with any type of data even though they are intended for one particular type of data. This can be quite useful if an equivalent instruction doesn't exist for the type of data you have or if an instruction for another type of data is more efficient.

All XMM instructions that move, shuffle or shift data as well as the Boolean instructions can be used for other types of data than they are intended for. But instructions that do any kind

of arithmetic operation, type conversion or precision conversion can only be used for the type of data it is intended for. For example, the `FLD` instruction does more than move floating point data, it also converts to a different precision. If you try to use `FLD` and `FSTP` for moving integer data then you may get exceptions for denormal operands in case the integer data do not happen to represent a normal floating point number. The instruction may even change the value of the data in some cases. But the instruction `MOVAPS`, which is also intended for moving floating point data, does not convert precision or anything else. It just moves the data. Therefore, it is OK to use `MOVAPS` for moving integer data.

If you are in doubt whether a particular instruction will work with any type of data then check the software manual from Intel or AMD. If the instruction can generate any kind of "floating point exception" then it should not be used for any other kind of data than it is intended for.

There is a penalty for using the wrong type of instructions on AMD processors in some cases. The reason is that the processor stores extra information about floating point numbers in XMM registers in order to remember if the number is zero, normal or denormal. This information is lost if an instruction intended for integer vectors is used for moving the floating point data. The processor needs one or two clock cycles extra for re-generating the lost information. This is called a reformatting delay. The reformatting delay occurs whenever the output of an integer XMM instruction is used as input for a floating point XMM instruction, except when the floating point XMM instruction does nothing else than writing the value to memory. Interestingly, there is no reformatting delay when using single-precision XMM instructions for double-precision data or vice versa.

The reformatting delay occurs only in AMD processors. I have observed no reformatting delays on any Intel processor.

Using an instruction of a wrong type can be advantageous in cases where there is no reformatting delay and in cases where the gain by using a particular instruction is more than the reformatting delay. Some cases are described below.

Using the shortest instruction

The instructions for packed single precision floating point numbers, with names ending in `PS`, are one byte shorter than equivalent instructions for double precision or integers. For example, you may use `MOVAPS` instead of `MOVAPD` or `MOVDQA` for moving data to or from memory or between registers. A reformatting delay occurs in AMD processors when using `MOVAPS` for moving the result of an integer instruction to another register, but not when moving data to or from memory.

Using the most efficient instruction

There are several different ways of reading an XMM register from unaligned memory. The typed instructions are `MOVDQU`, `MOVUPD`, and `MOVUPS`. These are all quite inefficient. `LDDQU` is faster, but requires the SSE3 instruction set. On many processors, the most efficient way of reading an XMM register from unaligned memory is to read 64 bits at a time using `MOVLPS` and `MOVHPS`. Likewise, the fastest way of writing to unaligned memory may be to use `MOVLPS` and `MOVHPS`.

An efficient way of setting a vector register to zero is `PXOR XMM0,XMM0`. The P4 and P4E processors recognize this instruction as being independent of the previous value of `XMM0`, while it does not recognize this for `XORPS` or `XORPD`. The `PXOR` instruction is therefore preferred for setting a register to zero.

The integer versions of the Boolean vector instructions (`PAND`, `PANDN`, `POR`, `PXOR`) can use the FADD or FMUL unit in an AMD64 processor, while the floating point versions can use only the FMUL unit.

Using an instruction that is not available for other types of data

There are many situations where it is advantageous to use an instruction intended for a different type of data simply because an equivalent instruction doesn't exist for the type of data you have.

The instructions for single precision float vectors are available in the SSE instruction set, while the equivalent instructions for double precision and integers require the SSE2 instruction set. Using `MOVAPS` instead of `MOVAPD` or `MOVDQA` for moving data makes the code compatible with processors that have SSE but not SSE2.

There are many useful instructions for data shuffling that are available for only one type of data. These instructions can easily be used for other types of data than they are intended for. The reformatting delay, if any, is likely to be less than the cost of alternative solutions. The data shuffling instructions are listed in the next paragraph.

13.3 Shuffling data

Vectorized code sometimes needs a lot of instructions for swapping and copying vector elements and putting data into the right positions in the vectors. The need for these extra instructions is reducing the advantage of using vector operations. It can often be an advantage to use a shuffling instruction that is intended for a different type of data than you have, as explained in the previous paragraph. Some instructions that are useful for data shuffling are listed below.

Moving data between different elements of a register (Use same register for source and destination)

Instruction	Block size, bits	Description	Instruction set
`PSHUFD`	32	Universal shuffle	SSE2
`PSHUFLW`	16	Shuffles low half of register only	SSE2
`PSHUFHW`	16	Shuffles high half of register only	SSE2
`SHUFPS`	32	Shuffle	SSE
`SHUFPD`	64	Shuffle	SSE2
`PSLLDQ`	8	Shifts to a different position and sets the original position to zero	SSE2
`PSHUFB`	8	Shuffle	SSE4

Table 13.1. Shuffle instructions

Moving data from one register to different elements of another register

Instruction	Block size, bits	Description	Instruction set
`PSHUFD`	32	Universal shuffle	SSE2
`PSHUFLW`	16	Shuffles low half of register only	SSE2
`PSHUFHW`	16	Shuffles high half of register only	SSE2

Table 13.2. Move-and-shuffle instructions

Using the `PSHUFD` instruction can often save a move instruction. Can be used for higher element sizes as well.

Combining data from two different sources

Instruction	Block size, bits	Description	Instruction set
SHUFPS	32	Lower 2 dwords from any position of source higher 2 dwords from any position of destination	SSE
SHUFPD	64	Low qword from any position of source high qword from any position of destination	SSE2
MOVLPS/D	64	Low qword from memory, high qword unchanged	SSE/SSE2
MOVHPS/D	64	High qword from memory, low qword unchanged	SSE/SSE2
MOVLHPS	64	Low qword unchanged, high qword from low of source	SSE
MOVHLPS	64	Low qword from high of source, high qword unchanged	SSE
MOVSS	32	Lowest dword from source (register only), bits 32-127 unchanged	SSE
MOVSD	64	Low qword from source (register only), high qword unchanged	SSE2
PUNPCKLBW	8	Low 8 bytes from source and destination interleaved	SSE2
PUNPCKLWD	16	Low 4 words from source and destination interleaved	SSE2
PUNPCKLDQ	32	Low 2 dwords from source and destination interleaved	SSE2
PUNPCKLQDQ	64	Low qword unchanged, high qword from low of source	SSE2
PUNPCKHBW	8	High 8 bytes from source and destination interleaved	SSE2
PUNPCKHWD	16	High 4 words from source and destination interleaved	SSE2
PUNPCKHDQ	32	High 2 dwords from source and destination interleaved	SSE2
PUNPCKHQDQ	64	Low qword from high of destination, high qword from high of source	SSE2
PACKUSWB	8	Low 8 bytes from 8 words of destination, high 8 bytes from 8 words of source. Converted with unsigned saturation.	SSE2
PACKSSWB	8	Low 8 bytes from 8 words of destination, high 8 bytes from 8 words of source. Converted with signed saturation.	SSE2
PACKSSDW	16	Low 4 words from 4 dwords of destination, high 4 words from 4 dwords of source. Converted with signed saturation.	SSE2
MOVQ	64	Low qword from source, high qword set to zero	SSE2
PINSRW	16	Modify any word, all other words unchanged	SSE2

Table 13.3. Combine data

Copying data to multiple elements of a register (broadcast)
(Use same register for source and destination)

Instruction	Block size, bits	Description	Instruction set
PSHUFD	32	Broadcast any dword	SSE2
SHUFPS	32	Broadcast dword	SSE
SHUFPD	64	Broadcast qword	SSE2
MOVLHPS	64	Broadcast qword	SSE2
MOVHLPS	64	Broadcast high qword	SSE2
MOVDDUP	64	Broadcast qword	SSE3
MOVSLDUP	32	Copy dword 0 to 1, copy dword 2 to 3	SSE3
MOVSHDUP	32	Copy dword 1 to 0, copy dword 3 to 2	SSE3
PUNPCKLBW	8	Duplicate each of the lower 8 bytes	SSE2
PUNPCKLWD	16	Duplicate each of the lower 4 words	SSE2
PUNPCKLDQ	32	Duplicate each of the lower 2 dwords	SSE2
PUNPCKLQDQ	64	Broadcast qword	SSE2
PUNPCKHBW	8	Duplicate each of the higher 8 bytes	SSE2
PUNPCKHWD	16	Duplicate each of the higher 4 words	SSE2
PUNPCKHDQ	32	Duplicate each of the higher 2 dwords	SSE2
PUNPCKHQDQ	64	Broadcast high qword	SSE2

Table 13.4. Broadcast data

Copy data from one register to all elements of another register (broadcast)

Instruction	Block size, bits	Description	Instruction set
PSHUFD xmm2,xmm1,0	32	Broadcast dword	SSE2
PSHUFD xmm2,xmm1,0EEH	64	Broadcast qword	SSE2
MOVDDUP	64	Broadcast qword	SSE3
MOVSLDUP	32	2 copies of each of dword 0 and 2	SSE3
MOVSHDUP	32	2 copies of each of dword 1 and 3	SSE3

Table 13.5. Move and broadcast data

Example: Horizontal addition

The following examples show how to add all elements of a vector.

```
; Example 13.5a. Add 16 elements in vector of 8-bit unsigned integers
movaps     xmm0, source    ; Source vector, 16 8-bit unsigned integers
movaps     xmm1, source    ; Same
pxor       xmm2, xmm2      ; 0
punpcklbw  xmm0, xmm2      ; Zero-extend low  8 elements to words
punpckhbw  xmm1, xmm2      ; Zero-extend high 8 elements to words
paddw      xmm0, xmm1      ; Add elements as words
pshufd     xmm1, xmm0, 0EH ; Get bit 64-127 from xmm1 (or use movhlps)
paddw      xmm0, xmm1      ; Sums are in 4 words
pshufd     xmm1, xmm0, 01H ; Get bit 32-63 from xmm0
paddw      xmm0, xmm1      ; Sums are in 2 words
pshuflw    xmm1, xmm0, 01H ; Get bit 16-31 from xmm0
paddw      xmm0, xmm1      ; Sum is in one word
movd       eax,  xmm0      ; Sum is in low word of eax
mov        [sum], ax       ; Store sum


; Example 13.5b. Add eight elements in vector of 16-bit integers
movaps     xmm0, source    ; Source vector, 8 16-bit integers
pshufd     xmm1, xmm0, 0EH ; Get bit 64-127 from xmm1 (or use movhlps)
```

```
    paddw      xmm0, xmm1      ; Sums are in 4 words
    pshufd     xmm1, xmm0, 01H ; Get bit 32-63 from xmm0
    paddw      xmm0, xmm1      ; Sums are in 2 words
    pshuflw    xmm1, xmm0, 01H ; Get bit 16-31 from xmm0
    paddw      xmm0, xmm1      ; Sum is in one word
    movd       eax,  xmm0      ; Sum is in low word of eax
    mov        [sum], ax       ; Store sum
```

(If SSE4 instruction set is available, use `PHADDW`).

```
    ; Example 13.5c. Add four elements in vector of 32-bit integers
    movaps     xmm0, source    ; Source vector, 4 32-bit integers
    pshufd     xmm1, xmm0, 0EH ; Get bit 64-127 from xmm1 (or use movhlps)
    paddd      xmm0, xmm1      ; Sums are in 2 dwords
    pshufd     xmm1, xmm0, 01H ; Get bit 32-63 from xmm0
    paddd      xmm0, xmm1      ; Sum is in one dword
    movd       [sum],xmm0      ; Store sum
```

(If SSE4 instruction set is available, use `PHADDD`).

```
    ; Example 13.5d. Add two elements in vector of 64-bit integers
    movaps     xmm0, source    ; Source vector, 4 32-bit integers
    pshufd     xmm1, xmm0, 0EH ; Get bit 64-127 from xmm1 (or use movhlps)
    paddq      xmm0, xmm1      ; Sum is in one qword
    movlps     [sum],xmm0      ; Store sum
```

```
    ; Example 13.5e. Add four elements in vector of 32-bit floats
    movaps     xmm0, source    ; Source vector, 4 32-bit integers
    movhlps    xmm1, xmm0      ; Get bit 64-127 from xmm1
    addps      xmm0, xmm1      ; Sums are in 2 dwords
    pshufd     xmm1, xmm0, 01H ; Get bit 32-63 from xmm0
    addss      xmm0, xmm1      ; Sum is in one dword
    movss      [sum],xmm0      ; Store sum
```

(If SSE3 instruction set is available, use `HADDPS` twice).

```
    ; Example 13.5f. Add two elements in vector of 64-bit doubles
    movaps     xmm0, source    ; Source vector, 2 64-bit doubles
    movhlps    xmm1, xmm0      ; Get bit 64-127 from xmm1
    addsd      xmm0, xmm1      ; Sum is in one qword
    movsd      [sum],xmm0      ; Store sum
```

(If SSE3 instruction set is available, use `HADDPD`).

13.4 Generating constants

There is no instruction for moving a constant into an XMM register. The default way of putting a constant into an XMM register is to load it from a memory constant. This is also the most efficient way if cache misses are rare. But if cache misses are frequent then we may look for alternatives.

One alternative is to copy the constant from a static memory location to the stack outside the innermost loop and use the stack copy inside the innermost loop. A memory location on the stack is less likely to cause cache misses than a memory location in a constant data segment. However, this option may not be possible in library functions.

A second alternative is to store the constant to stack memory using integer instructions and then load the value from the stack memory to the XMM register.

A third alternative is to generate the constant by clever use of various instructions. This does not use the data cache but takes more space in the code cache. The code cache is less likely to cause cache misses because the code is contiguous.

The constants may be reused many times as long as the register is not needed for something else.

The table 13.6 below shows how to make various integer constants in XMM registers. The same value is generated in all cells in the vector:

Making constants for integer vectors in XMM registers

Value	8 bit	16 bit	32 bit	64 bit
0	`pxor xmm0,xmm0`	`pxor xmm0,xmm0`	`pxor xmm0,xmm0`	`pxor xmm0,xmm0`

1	`pcmpeqw xmm0,xmm0`
	`psrlw xmm0,15`
	`packuswb xmm0,xmm0`
		`pcmpeqw xmm0,xmm0`
		`psrlw xmm0,15`
			`pcmpeqd xmm0,xmm0`
			`psrld xmm0,31`
				`pcmpeqw xmm0,xmm0`
				`psrlq xmm0,63`

2	`pcmpeqw xmm0,xmm0`
	`psrlw xmm0,15`
	`psllw xmm0,1`
	`packuswb xmm0,xmm0`
		`pcmpeqw xmm0,xmm0`
		`psrlw xmm0,15`
		`psllw xmm0,1`
			`pcmpeqd xmm0,xmm0`
			`psrld xmm0,31`
			`pslld xmm0,1`
				`pcmpeqw xmm0,xmm0`
				`psrlq xmm0,63`
				`psllq xmm0,1`

3	`pcmpeqw xmm0,xmm0`
	`psrlw xmm0,14`
	`packuswb xmm0,xmm0`
		`pcmpeqw xmm0,xmm0`
		`psrlw xmm0,14`
			`pcmpeqd xmm0,xmm0`
			`psrld xmm0,30`
				`pcmpeqw xmm0,xmm0`
				`psrlq xmm0,62`

4	`pcmpeqw xmm0,xmm0`
	`psrlw xmm0,15`
	`psllw xmm0,2`
	`packuswb xmm0,xmm0`
		`pcmpeqw xmm0,xmm0`
		`psrlw xmm0,15`
		`psllw xmm0,2`
			`pcmpeqd xmm0,xmm0`
			`psrld xmm0,31`
			`pslld xmm0,2`
				`pcmpeqw xmm0,xmm0`
				`psrlq xmm0,63`
				`psllq xmm0,2`

-1	`pcmpeqw xmm0,xmm0`	`pcmpeqw xmm0,xmm0`	`pcmpeqd xmm0,xmm0`	`pcmpeqw xmm0,xmm0`

-2	`pcmpeqw xmm0,xmm0`
	`psllw xmm0,1`
	`packsswb xmm0,xmm0`
		`pcmpeqw xmm0,xmm0`
		`psllw xmm0,1`
			`pcmpeqd xmm0,xmm0`
			`pslld xmm0,1`
				`pcmpeqw xmm0,xmm0`
				`psllq xmm0,1`

Other value	`mov eax,`
	`  value*01010101H`
	`movd xmm0,eax`
	`pshufd xmm0,xmm0,0`
		`mov eax,`
		`  value*10001H`
		`movd xmm0,eax`
		`pshufd xmm0,xmm0,0`
			`mov eax,value`
			`movd xmm0,eax`
			`pshufd xmm0,xmm0,0`
				`mov rax,value`
				`movq xmm0,rax`
				`punpcklqdq xmm0,xmm0`
				`(64 bit mode only)`

Table 13.6. Generate integer vector constants

Table 13.7 below shows how to make various floating point constants in XMM registers. The same value is generated in one or all cells in the vector:

Making floating point constants in XMM registers

Value	scalar single	scalar double	vector single	vector double
0.0	`pxor xmm0,xmm0`	`pxor xmm0,xmm0`	`pxor xmm0,xmm0`	`pxor xmm0,xmm0`

0.5	`pcmpeqw xmm0,xmm0`
	`pslld xmm0,26`
	`psrld xmm0,2`
		`pcmpeqw xmm0,xmm0`
		`psllq xmm0,55`
		`psrlq xmm0,2`
			`pcmpeqw xmm0,xmm0`
			`pslld xmm0,26`
			`psrld xmm0,2`
				`pcmpeqw xmm0,xmm0`
				`psllq xmm0,55`
				`psrlq xmm0,2`

1.0	`pcmpeqw xmm0,xmm0`
	`pslld xmm0,25`
	`psrld xmm0,2`
		`pcmpeqw xmm0,xmm0`
		`psllq xmm0,54`
		`psrlq xmm0,2`
			`pcmpeqw xmm0,xmm0`
			`pslld xmm0,25`
			`psrld xmm0,2`
				`pcmpeqw xmm0,xmm0`
				`psllq xmm0,54`
				`psrlq xmm0,2`

1.5	`pcmpeqw xmm0,xmm0`
	`pslld xmm0,24`
	`psrld xmm0,2`
		`pcmpeqw xmm0,xmm0`
		`psllq xmm0,53`
		`psrlq xmm0,2`
			`pcmpeqw xmm0,xmm0`
			`pslld xmm0,24`
			`psrld xmm0,2`
				`pcmpeqw xmm0,xmm0`
				`psllq xmm0,53`
				`psrlq xmm0,2`

2.0	`pcmpeqw xmm0,xmm0`
	`pslld xmm0,31`
	`psrld xmm0,1`
		`pcmpeqw xmm0,xmm0`
		`psllq xmm0,63`
		`psrlq xmm0,1`
			`pcmpeqw xmm0,xmm0`
			`pslld xmm0,31`
			`psrld xmm0,1`
				`pcmpeqw xmm0,xmm0`
				`psllq xmm0,63`
				`psrlq xmm0,1`

-2.0	`pcmpeqw xmm0,xmm0` `pslld xmm0,30`	`pcmpeqw xmm0,xmm0` `psllq xmm0,62`	`pcmpeqw xmm0,xmm0` `pslld xmm0,30`	`pcmpeqw xmm0,xmm0` `psllq xmm0,62`
sign bit	`pcmpeqw xmm0,xmm0` `pslld xmm0,31`	`pcmpeqw xmm0,xmm0` `psllq xmm0,63`	`pcmpeqw xmm0,xmm0` `pslld xmm0,31`	`pcmpeqw xmm0,xmm0` `psllq xmm0,63`
not sign bit	`pcmpeqw xmm0,xmm0` `psrld xmm0,1`	`pcmpeqw xmm0,xmm0` `psrlq xmm0,1`	`pcmpeqw xmm0,xmm0` `psrld xmm0,1`	`pcmpeqw xmm0,xmm0` `psrlq xmm0,1`
Other value	`mov eax, value` `movd xmm0,eax`	`mov eax, value>>32` `movd xmm0,eax`	`mov eax, value` `movd xmm0,eax`	`mov eax, value>>32` `movd xmm0,eax`

value (32 bit mode)		`psllq xmm0,32`	`shufps xmm0,xmm0,0`	`pshufd xmm0,xmm0,22H`

Other value (64 bit mode)	`mov eax, value`
	`movd xmm0,eax`
		`mov rax, value`
		`movq xmm0,rax`
			`mov eax, value`
			`movd xmm0,eax`
			`shufps xmm0,xmm0,0`
				`mov rax, value`
				`movq xmm0,rax`
				`shufpd xmm0,xmm0,0`

Table 13.7. Generate floating point vector constants

The "sign bit" is a value with the sign bit set and all other bits = 0. This is used for changing or setting the sign of a variable. For example to change the sign of a 2*double vector in `xmm0`:

```
; Example 13.6. Change sign of 2*double vector
pcmpeqw xmm7, xmm7    ; All 1's
psllq   xmm7, 63      ; Shift out the lower 63 1's
xorpd   xmm0, xmm7    ; Flip sign bit of xmm0
```

The "not sign bit" is the inverted value of "sign bit". It has the sign bit = 0 and all other bits = 1. This is used for getting the absolute value of a variable. For example to get the absolute value of a 2*double vector in `xmm0`:

```
; Example 13.7. Absolute value of 2*double vector
pcmpeqw xmm6, xmm6    ; All 1's
psrlq   xmm6, 1       ; Shift out the highest bit
andpd   xmm0, xmm6    ; Set sign bit to 0
```

Generating an arbitrary double precision value in 32-bit mode is more complicated. The method in table 13.7 uses only the upper 32 bits of the 64-bit representation of the number, assuming that the lower binary decimals of the number are zero or that an approximation is acceptable. For example, to generate the double value 9.25, we first use a compiler or assembler to find that the hexadecimal representation of 9.25 is 4022800000000000H. The lower 32 bits can be ignored, so we can do as follows:

```
; Example 13.8a. Set 2*double vector to arbitrary value (32 bit mode)
mov        eax,  40228000H  ; High 32 bits of 9.25
movd       xmm0, eax        ; Move to xmm0
pshufd     xmm0, xmm0, 22H  ; Get value into dword 1 and 3
```

In 64-bit mode, we can use 64-bit integer registers:

```
; Example 13.8b. Set 2*double vector to arbitrary value (64 bit mode)
mov        rax,  4022800000000000H  ; Full representation of 9.25
movq       xmm0, rax        ; Move to xmm0
shufpd     xmm0, xmm0, 0    ; Broadcast
```

Note that some assemblers use the very misleading name `movd` instead of `movq` for the instruction that moves 64 bits between a general purpose register and an `XMM` register.

13.5 Vector operations in general purpose registers

Sometimes it is possible to handle packed data in 32-bit or 64-bit general purpose registers. You may use this method on processors where integer operations are faster than vector operations or where vector operations are not available.

A 64-bit register can hold two 32-bit integers, four 16-bit integers, eight 8-bit integers, or 64 Booleans. When doing calculations on packed integers in 32-bit or 64-bit registers, you have to take special care to avoid carries from one operand going into the next operand if

overflow is possible. Carry does not occur if all operands are positive and so small that overflow cannot occur. For example, you can pack four positive 16-bit integers into RAX and use ADD RAX,RBX instead of PADDW MM0,MM1 if you are sure that overflow will not occur. If carry cannot be ruled out then you have to mask out the highest bit, as in the following example, which adds 2 to all four bytes in EAX:

```
; Example 13.9. Byte vector in 32-bit register
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 13.10a, optimized strlen procedure (32-bit):
_strlen PROC    NEAR
; extern "C" int strlen (const char * s);
; Works in 32-bit Window and 32-bit Linux.
; In Linux, remove the underscore from the function name.
        push    ebx
        mov     eax, [esp+8]           ; get pointer s
        lea     edx, [eax+3]           ; pointer+3 used in the end
l1:     mov     ebx, [eax]             ; read 4 bytes of string
        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]           ; .. and increment pointer by 2
        cmovz   eax, ebx
        add     cl,  cl                ; test first byte
        sbb     eax, edx               ; compute length
        pop     ebx
        ret
_strlen ENDP

; Example 13.11b, optimized strlen procedure (64-bit):
strlen PROC    NEAR
; extern "C" int strlen (const char * s);
; This code uses 64-bit Windows calling conventions with s in RCX.
; For 64-bit Linux, replace RCX by RDI and ECX by EDI and remove
; PUSH RSI and POP RSI.
        push    rsi
        mov     rsi, 0101010101010101H ; load constants
        mov     r9,  8080808080808080H
        lea     r8d, [rcx+7]           ; begin of string + 7
```

```
l1:     mov     rax, [rcx]              ; read 8 bytes of string
        mov     rdx, rax
        add     rcx, 8                  ; increment pointer to string
        sub     rax, rsi                ; subtract 1 from each byte
        not     rdx                     ; invert all bytes
        and     rax, rdx                ; and these two
        and     rax, r9                 ; test all sign bits
        jz      l1                      ; continue if no 0 byte found
        mov     rdx, rax
        shr     rdx, 32
        test    eax, eax                ; test first four bytes
        cmovz   eax, edx                ; shift if not in first four
        lea     edx, [rcx+4]            ; .. and increment pointer by 4
        cmovz   ecx, edx
        mov     edx, eax
        shr     edx, 16
        test    ax,  ax                 ; test first two bytes
        cmovz   eax, edx                ; shift if not in first two
        lea     edx, [rcx+2]            ; .. and increment pointer by 2
        cmovz   ecx, edx
        add     al,  al                 ; test first byte
        sbb     ecx, r8d                ; decrement pointer
        mov     eax, ecx
        pop     rsi
        ret
strlen  ENDP
```

The string should preferably be aligned by 4 or 8, respectively. The code may read past the end of the string, so the string should not be placed at the end of a segment. Handling four or eight bytes simultaneously can be quite difficult. The code in example 13.10 searches for 0-bytes by subtracting one and testing if the sign bit changes from 0 to 1. This makes it possible to test all four or eight bytes in one operation. This algorithm involves the subtraction of 1 from all bytes. I have not masked out the highest bit of each byte before subtracting, as I did in example 13.9, 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 only. If you want to search for a byte value other than zero, then you may `XOR` all bytes with the value you are searching for, and then use the method above to search for the first zero.

14 Multithreading

There is a limit to how much processing power you can get out of a single CPU. Therefore, many modern computer systems have multiple CPU kernels. The way to make use of multiple CPU kernels is to divide the computing job between multiple threads. The optimal number of threads is equal to the number of CPU kernels. The workload should ideally be divided evenly between the threads.

Multithreading is useful where the code has an inherent parallelism that is coarse-grained. Multithreading cannot be used for fine-grained parallelism because there is a considerable overhead cost of starting and stopping threads and synchronizing the threads. Communication between threads can be quite costly. Therefore, the computing job should preferably be divided into threads at the highest possible level. If the outermost loop can be parallelized, then it should be divided into one loop for each thread, each doing its share of the whole job.

Thread-local storage should preferably use the stack, because accessing thread-local static memory is inefficient.

See manual 1: "Optimizing software in C++" for more details on multithreading.

15 CPU dispatching

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 runtime 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.

If you are in doubt how many different versions of the code to make then you should consider the expected lifetime of your software application and what kind of hardware the typical user can be expected to have. Most software applications have a lifetime of at least several years. The newest microprocessors that are available today will probably be the most common microprocessors in a few years time. Therefore, it makes sense to optimize for the newest microarchitecture. Few users will run a speed-critical application on an old microprocessor. At the time of writing (June 2006) it looks like the Core2 microarchitecture will be used in all future Intel processors while the P4 microarchitecture (called NetBurst) is on its way out. My recommendation in this situation would be to make one version which is optimized for the Core2 architecture and another version which is compatible with older processors. If the SSE3 and SSE4 instructions are not used then it will be compatible with most modern CPU's. If the version optimized for Core2 doesn't work well on AMD processors then make a third version optimized specifically for the newest AMD processors. It is important to include a version that works on old microprocessors because there can always be categories of users that you haven't thought of who don't have the newest computers.

I have provided a library of subroutines that check the processor type and determine which instructions are supported. This library can be downloaded from www.agner.org/optimize/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 sets, you may use the macros at www.agner.org/optimize/macros.zip for coding the new instructions.

15.1 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 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 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 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/optimize/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/optimize/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.

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. Some operating systems provide system functions that tell which instruction set is supported, but the method mentioned above is independent of the operating system.

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.

16 Problematic Instructions

16.1 LEA instruction (all processors)

The LEA instruction is useful for many purposes because it can do a shift operation, two additions, and a move in just one instruction. Example:

```
; Example 16.1a, LEA instruction
lea eax, [ebx+8*ecx-1000]
```

is much faster than

```
; Example 16.1b
mov  eax, ecx
shl  eax, 3
add  eax, ebx
sub  eax, 1000
```

A typical use of LEA is as a three-register addition: `lea eax,[ebx+ecx]`. The LEA instruction can also be used for doing an addition or shift without changing the flags.

The processors have no documented addressing mode with a scaled index register and nothing else. Therefore, an instruction like `lea eax,[ebx*2]` is actually coded as `lea eax,[ebx*2+00000000H]` with an immediate displacement of 4 bytes. The size of this instruction can be reduces by writing `lea eax,[ebx+ebx]`. 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:

```
; Example 16.2, LEA instruction without base pointer
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 with a scaled index register.

The size of the base and index registers can be changed with an address size prefix. The size of the destination register can be changed with an operand size prefix (See prefixes, page 19). If the operand size is less than the address size then the result is truncated. If the operand size is more than the address size then the result is zero-extended.

The shortest version of LEA in 64-bit mode has 32-bit operand size and 64-bit address size, e.g. `LEA EAX,[RBX+RCX]`, see page 65. Use this version when the result is sure to be less than 2^{32}. Use the version with a 64-bit destination register for address calculation in 64-bit mode when the address may be bigger than 2^{32}.

The preferred version in 32-bit mode has 32-bit operand size and 32-bit address size. `LEA` with a 16-bit operand size is slow on AMD processors. `LEA` with a 16-bit address size in 32-bit mode should be avoided because the decoding of this instruction is slow on many processors.

`LEA` can also be used in 64-bit mode for loading a RIP-relative address. A RIP-relative address cannot be combined with base or index registers.

16.2 INC and DEC (all Intel processors)

The `INC` and `DEC` instructions do not modify the carry flag but they do modify the other arithmetic flags. Writing to only part of the flags register costs an extra uop on P4 and P4E, and it can cause partial flags stalls on P2, P3 and PM processors. Furthermore, it can cause a false dependence on the carry flag from a previous instruction.

Use `ADD` and `SUB` when optimizing for speed. Use `INC` and `DEC` when optimizing for size.

16.3 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.

The `XCHG` instruction with register operands may be useful when optimizing for size as explained on page 62.

16.4 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.

16.5 Rotates through carry (all processors)

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

16.6 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, P3 and PM, bit tests with a memory operand should be avoided. `BTC`, `BTR`, and `BTS` use 2 uops on AMD processors. Bit test instructions are useful when optimizing for size.

16.7 LAHF and SAHF (all processors)

`LAHF` is slow on P4 and P4E. Use `SETcc` instead for storing the value of a flag.

`SAHF` is slow on P4E and AMD processors. Use `TEST` instead for testing a bit in `AH`. Use `FCOMI` if available as a replacement for the sequence `FCOM` / `FNSTSW AX` / `SAHF`.

`LAHF` and `SAHF` are not available in 64 bit mode on some early 64-bit Intel processors.

16.8 Integer multiplication (all processors)

An integer multiplication takes from 3 to 14 clock cycles, depending on the processor. 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 or MMX instructions.

16.9 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 in manual 1: "Optimizing software in C++". 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:

```
; Example 16.3. Divide unsigned integer by 2^N
shr  eax, N
```

Dividing a signed integer by 2^N:

```
; Example 16.4. Divide 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 $q = x / d$ in integers with a word size of w bits, 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 $w+b$, where b is the number of binary digits in d minus 1. (b is the highest integer for which $2^b \le d$). Use $r = w+b$ 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 (re-)invented this method):

b = (the number of significant bits in d) - 1
$r = w + b$
$f = 2^r / d$
If f is an integer then d is a power of 2: go to 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: go to case B.
If the fractional part of f > 0.5: go to case C.

case A ($d = 2b$):

result = x SHR b

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

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

Example:
Assume that you want to divide by 5.
5 = 101B.
w = 32.
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:

```
; Example 16.5a. Divide unsigned integer by 5
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:

```
; Example 16.5b. Divide unsigned integer, case B
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:

```
; Example 16.5c. Divide unsigned integer, case B, check for overflow
         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 = w = 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:

```
; Example 16.5d. Divide unsigned integer by 5, limited range
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: $q = (x+1)*3333H$:

```
; Example 16.5e. Divide unsigned integer by 10, limited range
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.

The division method can also be used for vector operands. Example 16.5f divides eight unsigned 16-bit integers by 10:

```
; Example 16.5f. Divide vector of unsigned integers by 10
.data
align 16
RECIPDIV DW 8 dup (0CCCDH) ; Vector of reciprocal divisor

.code
pmulhuw xmm0, RECIPDIV
psrlw   xmm0, 3
```

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 16.6, 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     DSHIFT                 ; divisor is power of 2
        add    eax, edx               ; correct for rounding error
        jc     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 $0 \le x < 2^{32}$, $0 < d < 2^{32}$.

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)

Two or more floating point divisions can be combined into one, using the method described in manual 1: "Optimizing software in C++".

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, 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 on most processors.

Using reciprocal instruction for fast division (processors with SSE)

On processors with the SSE instruction set, 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 16.7, fast division, single precision (SSE)
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 23 clock cycles on a PM 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 39 clock cycles on the PM. It gives exactly the same results as the `DIV` instruction for 0 ≤ dividend ≤ 7FFFFH and 0 < divisor ≤ 7FFFFH:

```
; Example 16.8, integer division with packed 16-bit words (SSE2):
; 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.

16.10 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 largest word size possible (`DWORD` in 32-bit mode, `QWORD` in 64-bit mode), and make sure that both source and destination are aligned by the word size.

Moving data in XMM registers is often faster than `REP MOVSD` and `REP STOSD`. See page 131 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 align both source and destination by 8. Never use `MOVSB` or `MOVSW` in optimized code, not even in 16-bit mode.

On PM, `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 difference between `EDI` and `ESI` must be numerically greater than or equal to 64
- 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 speed is approximately 5-6 bytes per clock cycle for both instructions, which is more than 4 times as fast as when the above conditions are not met. The speed is higher for high values of `ECX`.

On P4, the number of clock cycles for `REP MOVSD` is difficult to predict, but it is always faster to use `MOVAPS` 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 107 for alternatives to `REPNE SCASB`.

16.11 WAIT instruction (all processors)

You can often increase speed by omitting the `WAIT` instruction (also known as `FWAIT`). 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:

```
; Example 16.9. Uses of WAIT:
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 functionality in point A is never needed on any other processors than the old 8087. Unless you want your 16-bit 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 even the oldest 32-bit processors 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 might overwrite the value to load. FNOP may be faster than WAIT on some processors and serve the same purpose.

16.12 FCOM + FSTSW AX (all processors)

The FNSTSW instruction is very slow on all processors. Most processors have FCOMI instructions to avoid the slow FNSTSW. Using FCOMI instead of the common sequence FCOM / FNSTSW AX / SAHF will save 4 - 8 clock cycles. 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:

```
; Example 16.10a.
    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:

```
; Example 16.10b.
    fld    [a]
    fcomp  [b]
    fnstsw ax
    shr    ah,1
    jc     ASmallerThanB
```

Testing for zero or equality:

```
; Example 16.10c.
    ftst
    fnstsw ax
    and    ah, 40H     ; Don't use TEST instruction, it's not pairable
    jnz    IsZero      ; (the zero flag is inverted!)
```

Test if greater:

```
; Example 16.10d.
    fld    [a]
    fcomp  [b]
    fnstsw ax
    and    ah, 41H
    jz     AGreaterThanB
```

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. You may fill this gap with integer instructions.

It is sometimes faster to use integer instructions for comparing floating-point values, as described on page 128 and 130.

16.13 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 127 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.

16.14 FRNDINT (all processors)

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

```
; Example 16.11.
    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 127 on how to truncate on processors that don't have truncate instructions. On processors with SSE instructions, use the conversion instructions such as CVTSS2SI and CVTTSS2SI.

16.15 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:

```
; Example 16.12a.
    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:

```
; Example 16.12b.
    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:

```
; Example 16.12c.
    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 processors with SSE2 instructions, you can make these operations in XMM registers without the need for a memory intermediate (see page 129).

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

```
; Example 16.13. Exponential function
; extern "C" long double 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
```

16.16 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.

16.17 FSQRT (SSE processors)

A fast way of calculating an approximate square root on processors with SSE 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.

16.18 FLDCW (Most Intel processors)

The PPro, P2, P3 and PM 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 127 on how to convert floating-point numbers to integers without changing the control word. On processors with SSE, use truncation instructions such as CVTTSS2SI instead.

16.19 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:

```
; Example 16.14a. Emulate reverse bit scan
        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:

```
; Example 16.14b. Emulate forward bit scan
        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.

17 Special topics

17.1 XMM versus floating point registers (Processors with SSE)

Processors with the SSE instruction set can do single precision floating point calculations in `XMM` registers. Processors with the SSE2 instruction set can also do double precision calculations in `XMM` registers. Floating point calculations are approximately equally fast in `XMM` registers and `ST()` registers. The decision of whether to use the floating point stack registers `ST(0)` - `ST(7)` or the new `XMM` registers depends on the following factors.

Advantages of using `ST()` registers:

- Compatible with old processors without SSE or SSE2.
- Compatible with old operating systems without XMM support.
- Supports long double precision.
- Intermediate results are always calculated with long double precision.
- Precision conversions are free in the sense that they require no extra instructions and take no extra time. Use `ST()` registers for expressions where operands have mixed precision.
- Mathematical functions such as logarithms and trigonometric functions are supported by hardware instructions. These functions are useful when optimizing for size, but not necessarily faster than library functions using XMM registers.
- Conversions to and from decimal numbers can use the `FBLD` and `FBSTP` instructions when optimizing for size.
- Floating point instructions using `ST()` registers are smaller than the corresponding instructions using `XMM` registers. For example, `FADD ST(0),ST(1)` is 2 bytes, while `ADDSD XMM0,XMM1` is 4 bytes.

Advantages of using `XMM` registers:

- Can do two double precision or four single precision operations with a single vector instruction.
- Avoids the need to use FXCH for getting the desired register to the top of the stack.
- No need to clean up the register stack after use.
- Can be used together with MMX instructions.
- No need for memory intermediates when converting between integers and floating point numbers.
- 64-bit systems have 16 XMM registers, but only 8 ST() registers.
- ST() registers cannot be used in device drivers in 64-bit Windows.

17.2 MMX versus XMM registers (Processors with SSE2)

Integer vector instructions can use either the 64-bit MMX registers or the 128-bit XMM registers.

Advantages of using MMX registers:

- Compatible with older microprocessors since the PMMX.
- Compatible with old operating systems without XMM support.
- No need for data alignment.

Advantages of using XMM registers:

- The number of elements per vector in XMM registers is double the number in MMX registers.
- MMX registers cannot be used together with ST() registers.
- A series of MMX instructions must end with EMMS.
- 64-bit systems have 16 XMM registers, but only 8 MMX registers.
- MMX registers cannot be used in device drivers in 64-bit Windows.
- XMM instructions have higher throughput than MMX instructions on Core2 processors.

17.3 Freeing floating-point registers (all processors)

You have to free all used floating-point ST() 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 (PM) or port load (P4).

It is not recommended to use FFREE.

17.4 Transitions between floating-point and MMX instructions (Processors with MMX)

It is not possible to use 64-bit MMX registers and 80-bit floating-point ST() registers in the same part of the code. You must issue an EMMS instruction after the last instruction that uses 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 processors with out-of-order execution there is no such penalty.

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

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

```
; Example 17.1.
fistp dword ptr [TEMP]
mov eax, [TEMP]
```

On many processors, and especially the 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 many 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 processors with SSE or SSE2 instructions 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.

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.

```
; Example 17.2a. Rounding to nearest or even:
; extern "C" int round (double x);
_round  PROC    NEAR   ; (32 bit mode)
PUBLIC  _round
```

```
        fld     qword ptr [esp+4]
        fistp   dword ptr [esp+4]
        mov     eax, dword ptr [esp+4]
        ret
_round  ENDP

; Example 17.2b. Truncation towards zero:
; extern "C" int truncate (double x);
_truncate PROC    NEAR   ; (32 bit mode)
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

; Example 17.2c. Truncation towards minus infinity:
; extern "C" int ifloor (double x);
_ifloor PROC    NEAR   ; (32 bit mode)
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.

17.6 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

```
; Example 17.3a. Moving floating point data
fld qword ptr [esi]
fstp qword ptr [edi]
```

can be replaced by:

```
; Example 17.3b
```

```
mov eax,[esi]
mov ebx,[esi+4]
mov [edi],eax
mov [edi+4],ebx
```

or:

```
; Example 17.3c
movq mm0,[esi]
movq [edi],mm0
```

In 64-bit mode, use:

```
; Example 17.3d
mov rax,[rsi]
mov [rdi],rax
```

Many other manipulations are possible if you know how floating-point numbers are represented in binary format. See the chapter "Using integer operations for manipulating floating point variables" in manual 1: "Optimizing software in C++".

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

Table 17.1. Floating point formats

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.

It is possible to generate simple floating-point constants without using data in memory as explained on page 104.

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:

```
; Example 17.4a. Testing floating point value for zero
fld    dword ptr [ebx]
ftst
fnstsw ax
and    ah, 40h
jnz    IsZero
```

can be replaced by

```
; Example 17.4b. Testing floating point value for zero
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:

```
; Example 17.4c. Testing double value for zero
```

```
fld    qword ptr [ebx]
ftst
fnstsw ax
and    ah, 40h
jnz    IsZero
```

can be replaced by

```
; Example 17.4d. Testing double value for zero
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):

```
; Example 17.5. Testing floating point value for negative
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:

```
; Example 17.6. 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:

```
; Example 17.7. 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:

```
; Example 17.8. 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 by a power of 2 by simply adding to the exponent:

```
; Example 17.9. Multiply vector by power of 2
movaps  xmm0, [x]   ; four single-precision floats
movdqa  xmm1, [n]   ; four 32-bit positive 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 $0 \le x < 1.0$.

```
; Example 17.10. Convert bits to value between 0 and 1
.data
one    dq   1.0
x      dq   ?
n      dd   ?
.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
orpd   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 value in the interval $1.0 \le x < 2.0$. The `SUBSD` instruction subtracts 1.0 to get x into the desired interval. This is useful for random number generators.

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):

```
; Example 17.11a. Compare single precision float numbers
fld    [a]
fcomp  [b]
fnstsw ax
and    ah, 1
jnz    ASmallerThanB
```

can be replaced by:

```
; Example 17.11b. Compare single precision float numbers
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.

17.7 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. See page 99 for details.

Converting binary to decimal numbers

The FBLD and FBSTP instructions provide a simple and convenient way of converting numbers between binary and decimal, although not necessarily the fastest method.

17.8 Moving blocks of data (All processors)

There are several ways to move large blocks of data. The most common method is REP MOVS. See page 118 about the speed of this instruction.

In many cases it is faster to use instructions that move the largest possible number of bytes per operation. 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 processors without MMX 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 processors with MMX it is advantageous to use MMX registers for moving 8 bytes at a time. The loop may be rolled out by two.

On processors with SSE, the fastest way of moving data is to use the MOVAPS instruction if the conditions for fast REP MOVS on page 118 are not met or if the destination is in the level 1 or level 2 cache.

On processors with SSE 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.

REP MOVS takes only slightly longer time than a loop with MOVAPS. You may use REP MOVS for the sake of convenience, for covering cases with misaligned operands or odd-size memory blocks, for reducing code size, or to avoid branch misprediction if the block size is varying.

Other string instructions, such as REP STOS, may be replaced by more efficient loops.

For further advices on improving memory access see Intel's "IA-32 Intel Architecture Optimization Reference Manual" and AMD's "Software Optimization Guide for AMD64 Processors".

17.9 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, P3, PM. 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. The trick of putting the code in a data segment doesn't work in newer systems.

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.

Self-modifying code can be useful, for example in a math program where a user-defined function has to be evaluated many times. The program may contain a small compiler that converts the function to binary code.

18 Measuring performance

18.1 Testing speed

Many compilers have a profiler which makes it possible to measure how many times each function in a program is called and how long time it takes. This is very useful for finding any hot spot in the program. If a particular hot spot is taking a high proportion of the total execution time then this hot spot should be the target for your optimization efforts.

Many profilers are not very accurate, and certainly not accurate enough for fine-tuning a small part of the code. The most accurate way of testing the speed of a piece of code is to use the so-called time stamp counter. This is an internal 64-bit clock counter which can be read into `EDX:EAX` using the instruction `RDTSC` (read time stamp counter). The time stamp counter counts at the CPU clock frequency so that one count equals one clock cycle, which is the smallest relevant time unit. Some overclocked processors count at a slightly different frequency.

The time stamp counter is very useful because it can measure exactly how many clock cycles a piece of code takes.

On processors with out-of-order execution, 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 any DOS programs using `RDTSC` on these processors must run in real mode.

The biggest problem when counting clock ticks is to avoid interrupts. Protected operating systems do not allow you to clear the interrupt flag, so you cannot avoid interrupts and task switches during the test. This makes test results inaccurate and irreproducible. 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 not too long 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.

I have made a series of test programs that use method 1, 2 and possibly 6. These programs are available at www.agner.org/optimize/testp.zip.

You will soon observe when you are measuring clock cycles that a piece of code always takes longer time the first time it is executed where it is not in the cache. Furthermore, it may take two or three iterations before the branch predictor has adapted to the code. The first measurement gives the execution time when code and data are not in the cache. The subsequent measurements give the execution time with the best possible caching.

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 in manual 3: "The microarchitecture of Intel and AMD CPUs".

Other 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.

Most x86 processors also have a set of so-called performance monitor counters which can count events such as cache misses, misalignments, branch mispredictions, etc. These are very useful for diagnosing performance problems. The performance monitor counters are processor-specific. You need a different test setup for each type of CPU.

Details about the performance monitor counters can be found in Intel's "IA-32 Intel Architecture Software Developer's Manual", vol. 3 and in AMD's "BIOS and Kernel Developer's Guide".

You need privileged access to set up the performance monitor counters. This is done most conveniently with a device driver. The test programs at www.agner.org/optimize/testp.zip give access to the performance monitor counters under 32-bit and 64-bit Windows and 16-bit real mode DOS. These test program support the different kinds of performance monitor counters in most Intel and AMD processors.

Intel and AMD are providing profilers that use the performance monitor counters of their respective processors. Intel's profiler is called Vtune and AMD's profiler is called CodeAnalyst.

19 Literature

The present manual is part of a series available from www.agner.org/optimize as mentioned in the introduction on page 4.

A lot of useful literature can be downloaded for free from developer.intel.com and www.amd.com, 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.

The most important manuals from Intel are:

- IA-32 Intel Architecture Optimization Reference Manual.
- IA-32 Intel Architecture Software Developer's Manual, vol. 2A and 2B: Instruction Set Reference.

The most important manuals from AMD are:

- Software Optimization Guide for AMD64 Processors.
- AMD64 Architecture Programmer's Manual Volume 1: Application Programming.
- AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions.
- AMD64 Architecture Programmer's Manual Volume 4: 128-Bit Media Instructions.

A lot of other sources 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/optimize.

Some useful textbooks:

- R. C. Detmer: Introduction to 80x86 Assembly Language and Computer Architecture, 2'nd ed. Jones & Bartlett, 2006. Essentials of 80x86 Assembly Language. Jones & Bartlett, 2006.
- J. L. Hennessy and D. A. Patterson: Computer Architecture: A Quantitative Approach, 3'rd ed. 2002.
- S. Goedecker and A. Hoisie: Performance Optimization of Numerically Intensive Codes. SIAM, 2001.
- John R. Levine: Linkers and Loaders. Morgan Kaufmann, 2000.