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

By Agner Fog. Technical University of Denmark.

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, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers.

4. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs.

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

The latest versions of these manuals are always available from www.agner.org/optimize. Copyright conditions are listed on page 156 below.

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 and x86-64 instruction set. This instruction set is used by most microprocessors from Intel, AMD, and VIA. Operating systems that can use this instruction set include DOS, Windows, Linux, FreeBSD/Open BSD, and Intel-based Mac OS. The manual covers the newest microprocessors and the newest instruction sets. See manual 3 and 4 for details about individual microprocessor models.

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, AMD, and VIA CPUs". Tables of instruction timings etc. are provided in manual 4: "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs". 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 do not 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 reasons. 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 cases where compilers perform poorly and where significant 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. System code can use intrinsic functions instead of assembly. 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.

6. Application code can use intrinsic functions or vector classes instead of assembly. 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 Single-Instruction-Multiple-Data (SIMD) instructions. See page 33.

7. Portability. Assembly code is very platform-specific. Porting to a different platform is difficult. Code that uses intrinsic functions instead of assembly are portable to all x86 and x86-64 platforms.

8. Compilers have been improved a lot in recent years. The best compilers are now better than an experienced assembly programmer in many situations.

9. Compiled code may be faster than assembly code because compilers can make inter-procedural optimization and whole-program optimization. The assembly programmer usually has to make well-defined functions with a well-defined call interface that obeys all calling conventions in order to make the code testable and verifiable. This prevents many of the optimization methods that compilers use, such as function inlining, register allocation, constant propagation, common sub-expression elimination across functions, scheduling across functions, etc. These advantages can be obtained by using C++ code with intrinsic functions or vector classes instead of assembly code.

1.3 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, Intel-based Mac OS X.

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 parts 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 or C++. These 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. Use intrinsic functions instead (see p. 33) if possible.

- If the purpose of using assembly is to optimize for speed, then you have to identify the part of the program that consumes the 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 other languages than C or C++. You may possibly make both a static and a dynamic link version of the 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 cores. 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 AVX512 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 132).

- There are three assembly programming methods to choose between: (1) Use intrinsic functions and vector classes 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 33, 36, and 44 respectively).

- If you are using an assembler, then you have to choose between different syntax dialects. It may be 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 any 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.

2.3 Common coding pitfalls

The following list points out some of the most common programming errors in assembly code.

1. Forgetting to save registers. Some registers have callee-save status, for example EBX. These registers must be saved in the prolog of a function and restored in the epilog if they are modified inside the function. Remember that the order of POP instructions must be the opposite of the order of PUSH instructions. See page 28 for a list of callee-save registers.

2. Unmatched PUSH and POP instructions. The number of PUSH and POP instructions must be equal for all possible paths through a function. Example:

   ```
   Example 2.1. Unmatched push/pop
   push ebx
   test ecx, ecx
   jz   Finished
   ...
   pop  ebx
   Finished:       ; Wrong! Label should be before pop ebx
   ret
   ```

   Here, the value of EBX that is pushed is not popped again if ECX is zero. The result is that the RET instruction will pop the former value of EBX and jump to a wrong address.

3. Using a register that is reserved for another purpose. Some compilers reserve the use of EBP or EBX for a frame pointer or other purpose. Using these registers for a different purpose in inline assembly can cause errors.

4. Stack-relative addressing after push. When addressing a variable relative to the stack pointer, you must take into account all preceding instructions that modify the stack pointer. Example:

```
Example 2.2. Stack-relative addressing
mov  [esp+4], edi
push ebp
push ebx
cmp  esi, [esp+4]          ; Wrong!
```

   Here, the programmer probably intended to compare `ESI` with `EDI`, but the value of `ESP` that is used for addressing has been changed by the two `PUSH` instructions, so that `ESI` is in fact compared with `EBP` instead.

5. Confusing value and address of a variable. Example:

```
Example 2.3. Value versus addresssection .data
; NASM syntax:
MyVariable  DD  0          ; Define variable

section .text
mov eax, [MyVariable]      ; Gets value of MyVariable
mov eax, MyVariable        ; Gets address of MyVariable
lea eax, [MyVariable]      ; Gets address of MyVariable
mov ebx, [eax]             ; Gets value of MyVariable through pointer

; MASM syntax is even more confusing:
mov eax, MyVariable        ; Gets value of MyVariable
mov eax, offset MyVariable; Gets address of MyVariable
lea eax, MyVariable        ; Gets address of MyVariable
mov ebx, [eax]             ; Gets value of MyVariable through pointer
mov ebx, [100]             ; Gets the constant 100 despite brackets
mov ebx, ds:[100]          ; Gets value from address 100
```

6. Ignoring calling conventions. It is important to observe the calling conventions for functions, such as the order of parameters, whether parameters are transferred on the stack or in registers, and whether the stack is cleaned up by the caller or the called function. See page 27.

7. Function name mangling. A C++ code that calls an assembly function should use `extern "C"` to avoid name mangling. Some systems require that an underscore (_) is put in front of the name in the assembly code. See page 31.

8. Forgetting return. A function declaration must end with both `RET` and `ENDP`. Using one of these is not enough. The execution will continue in the code after the procedure if there is no `RET`.

9. Forgetting stack alignment. The stack pointer must point to an address divisible by 16 before any call statement, except in 16-bit systems and 32-bit Windows. See page 27.

10. Forgetting shadow space in 64-bit Windows. It is required to reserve 32 bytes of empty stack space before any function call in 64-bit Windows. See page 30.

11. Mixing calling conventions. The calling conventions in 64-bit Windows and 64-bit Linux are different. See page 27.

12. Forgetting to clean up floating point register stack. All floating point stack registers that are used by a function must be cleared, typically with `FSTP ST(0)`, before the function returns, except for `ST(0)` if it is used for return value. It is necessary to keep

track of exactly how many floating point registers are in use. If a functions pushes more values on the floating point register stack than it pops, then the register stack will grow each time the function is called. An exception is generated when the stack is full. This exception may occur somewhere else in the program.

13. Forgetting to clear MMX state. A function that uses MMX registers must clear these with the `EMMS` instruction before any call or return.

14. Forgetting to clear YMM state. A function that uses YMM or ZMM registers must clear these with the `VZEROUPPER` or `VZEROALL` instruction before any call or return.

15. Forgetting to clear direction flag. Any function that sets the direction flag with `STD` must clear it with `CLD` before any call or return.

16. Mixing signed and unsigned integers. Unsigned integers are compared using the `JB` and `JA` instructions. Signed integers are compared using the `JL` and `JG` instructions. Mixing signed and unsigned integers can have unintended consequences.

17. Forgetting to scale an array index. An array index must be multiplied by the size of one array element. For example `mov eax, MyIntegerArray[ebx*4]`.

18. Exceeding array bounds. An array with n elements is indexed from 0 to n - 1, not from 1 to n. A defective loop writing outside the bounds of an array can cause errors elsewhere in the program that are hard to find.

19. Loop with `ECX` = 0. A loop that ends with the `LOOP` instruction will repeat 2^{32} times if `ECX` is zero. Be sure to check if `ECX` is zero before the loop.

20. Reading carry flag after `INC` or `DEC`. The `INC` and `DEC` instructions do not change the carry flag. Do not use instructions that read the carry flag, such as `ADC`, `SBB`, `JC`, `JBE`, `SETA`, etc. after `INC` or `DEC`. Use `ADD` and `SUB` instead of `INC` and `DEC` to avoid this problem.

3 The basics of assembly coding

3.1 Assemblers available

There are several assemblers available for the x86 instruction set. Assembly programmers are in the unfortunate situation that there is no universally agreed syntax for x86 assembly. Different assemblers use different syntax variants. The most common assemblers are listed below.

MASM

The Microsoft assembler is included with Microsoft C++ compilers. Free versions can sometimes be obtained by downloading the Microsoft Windows driver kit (WDK) or the platforms software development kit (SDK) or as an add-on to the free Visual C++ Express Edition. MASM has been a de-facto standard in the Windows world for many years, but is now falling into disuse. The assembly output of several Windows compilers still uses MASM syntax. MASM syntax is somewhat messy and inconsistent due to a heritage that dates back to the very first assemblers for the 8086 processor. Microsoft is still maintaining MASM in order to provide a complete set of development tools for Windows, but it is obviously not profitable, and the maintenance of MASM is apparently kept at a minimum. Newer versions can run only when the compiler is installed.

GAS

The Gnu assembler is part of the Gnu Binutils package that is included with most distributions of Linux, BSD and Mac OS X. The Gnu and Clang compilers produce assembly output that goes through the Gnu assembler before it is linked. The Gnu assembler traditionally uses the AT&T syntax that works well for machine-generated code, but it is very inconvenient for human-generated assembly code. The AT&T syntax has the operands in an order that differs from all other x86 assemblers and from the instruction documentation published by Intel and AMD. It uses various prefixes like `%` and `$` for specifying operand types. The Gnu assembler is available for all x86 platforms.

Fortunately, the Gnu assembler has an option for using Intel syntax instead. The Gnu-Intel syntax is similar to MASM syntax. The Gnu-Intel syntax defines only the syntax for instruction codes, not for directives, functions, macros, etc. The directives still use the old Gnu-AT&T syntax. Specify `.intel_syntax noprefix` to use the Intel syntax. Specify `.att_syntax prefix` to return to the AT&T syntax before leaving inline assembly in C or C++ code.

NASM

NASM is a free open source assembler with support for several platforms and object file formats. The syntax is more clear and consistent than MASM syntax. NASM is updated regularly with new instruction sets. NASM is currently the best multi-platform assembler and is widely used by assembly programmers.

YASM

YASM is very similar to NASM and uses exactly the same syntax. YASM and NASM may be used interchangeably. YASM has not been updated since 2014.

FASM

The Flat assembler is another open source assembler for multiple platforms. The syntax is not compatible with other assemblers. FASM is itself written in assembly language – an enticing idea, but unfortunately this makes the development and maintenance of it less efficient.

UASM

UASM is the latest development in a chain of initiatives by different people. The origin is the WASM assembler included with Watcom C++ compiler. WASM was continued as an independent assembler under the name JWASM, and later HJWASM and UASM. It is compatible with MASM syntax, including advanced macro and high level directives. It may be used as a plugin to Visual Studio IDE.

HLA

High Level Assembler is actually a high level language compiler that allows assembly-like statements and produces assembly output. This was probably a good idea at the time it was invented, but today where many C++ compilers support intrinsic functions, I believe that HLA is no longer needed. HLA is apparently no longer maintained.

TASM

Borland Turbo Assembler was included with the once popular Borland C++ compiler and later continued by companies CodeGear and Embarcadero. TASM is obsolete and no longer available.

Inline assembly

Intel C++ compilers support inline assembly using a subset of the MASM syntax.

Microsoft compilers supports inline assembly only in 32-bit mode. It is possible to access C++ variables, functions, and labels simply by inserting their names in the assembly code. This is easy, but does not support C++ register variables. See page 36.

The Gnu and Clang compilers support inline assembly with access to the full range of instructions and directives of the Gnu assembler in both Intel and AT&T syntax. The access to C++ variables from assembly uses a versatile but complicated syntax.

The Intel compilers for Linux and Mac systems support both the Microsoft style and the Gnu style of inline assembly.

Intrinsic functions in C++

This is the most convenient way of combining low-level and high-level code. Intrinsic functions are high-level language representatives of machine instructions. For example, you can do a vector addition in C++ by calling the intrinsic function that is equivalent to an assembly instruction for vector addition. Furthermore, it is possible to define a vector class with an overloaded `+` operator so that a vector addition is obtained simply by writing `+`. Intrinsic functions are supported by Microsoft, Intel, Gnu, and Clang compilers. See page 33 and manual 1: "Optimizing software in C++".

Which assembler to choose?

In most cases, the easiest solution is to use intrinsic functions in C++ code. The compiler can take care of most of the optimization so that the programmer can concentrate on choosing the best algorithm and organizing the data into vectors. System programmers can access system instructions by using intrinsic functions without having to use assembly language.

Where real low-level programming is needed, such as in highly optimized function libraries or device drivers, you may use an assembler.

It may be preferred to use an assembler that is compatible with the C++ compiler you are using. This allows you to use the compiler for translating C++ to assembly, optimize the assembly code further, and then assemble it. If the assembler is not compatible with the syntax generated by the compiler, then you may generate an object file with the compiler and then disassemble the object file to the assembly syntax you need. The objconv disassembler supports several different syntax dialects.

The NASM assembler is a good choice for many purposes because it supports many platforms and object file formats, it is well maintained, and usually up to date with the latest instruction sets.

The examples in this manual use NASM syntax, unless otherwise noted.

See www.agner.org/optimize for links to various 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		

Table 3.1. General purpose registers in 16 bit mode.

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)

Table 3.2. Floating point stack registers

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

Table 3.3. Segment registers in 16 bit mode

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		

Table 3.4. General purpose registers in 32 bit mode

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

Table 3.5. Floating point and MMX registers

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- and bigger integer and floating point vector registers

Full or partial register bit 0 - 127	Full or partial register bit 0 - 255	Full register bit 0 - 511
XMM0	YMM0	ZMM0
XMM1	YMM1	ZMM1
XMM2	YMM2	ZMM2
XMM3	YMM3	ZMM3
XMM4	YMM4	ZMM4
XMM5	YMM5	ZMM5
XMM6	YMM6	ZMM6
XMM7	YMM7	ZMM7

Table 3.6. XMM, YMM and ZMM registers in 32 bit mode

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. The YMM registers are available only if the processor and the operating system supports the AVX instruction set. The ZMM registers are available if the processor supports the AVX-512 instruction set.

Segment registers

Full register bit 0 - 15
CS
DS
ES

FS
GS
SS

Table 3.7. Segment registers in 32 bit mode

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				

Table 3.8. Registers in 64 bit mode

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

Table 3.9. Floating point and MMX registers

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 and larger integer and floating point vector registers

Full or partial register bit 0 - 127	Full or partial register bit 0 - 255	Full register bit 0 - 511
XMM0	YMM0	ZMM0
XMM1	YMM1	ZMM1
XMM2	YMM2	ZMM2
XMM3	YMM3	ZMM3
XMM4	YMM4	ZMM4
XMM5	YMM5	ZMM5
XMM6	YMM6	ZMM6
XMM7	YMM7	ZMM7
XMM8	YMM8	ZMM8
XMM9	YMM9	ZMM9
XMM10	YMM10	ZMM10
XMM11	YMM11	ZMM11
XMM12	YMM12	ZMM12
XMM13	YMM13	ZMM13
XMM14	YMM14	ZMM14
XMM15	YMM15	ZMM15
		ZMM16
		ZMM17
		ZMM18
		ZMM19
		ZMM20
		ZMM21
		ZMM22
		ZMM23
		ZMM24
		ZMM25
		ZMM26
		ZMM27
		ZMM28

		ZMM29
		ZMM30
		ZMM31

Table 3.10. XMM, YMM and ZMM registers in 64 bit mode

Scalar floating point instructions use only 32 or 64 bits of the XMM registers for single or double precision, respectively. The YMM registers are available only if the processor and the operating system supports the AVX instruction set. The ZMM registers are available only if the processor supports the AVX512 instruction set. It is possible to use XMM16-31 and YMM16-31 when AVX512VL is supported by the processor.

Segment registers

Full register bit 0 - 15
CS
FS
GS

Table 3.11. Segment registers in 64 bit mode

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.
- An index register. This can only be SI or DI. 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 by the MASM assembler, even if it has a []. Examples:

```
; Example 3.3. Memory operands in 16-bit mode, MASM syntax
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. The value added to the segment must be a multiple of 8.

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.
- An index register. This can be any 32 bit register except `ESP`.
- 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:10H]       ; 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
```

Position-independent code in 32-bit mode

Position-independent code is required for making shared objects (`*.so`) in 32-bit Unix-like systems. The method commonly used for making position-independent code in 32-bit Linux and BSD is to use a global offset table (GOT) containing the addresses of all static objects. The GOT method is quite inefficient because the code has to fetch an address from the GOT every time it reads or writes data in the data segment. A faster method is to use an arbitrary reference point, as shown in the following example. Note that this works with the NASM assembler, but not with several other assemblers:

```
; Example 3.5a. Position-independent code, 32 bit, NASM syntax
SECTION .data
alpha:  dd      1
beta:   dd      2

SECTION .text

funca:  ; This function returns alpha + beta
        call    get_thunk_ecx             ; get ecx = eip
refpoint:                                 ; ecx points here
        mov     eax, [ecx+alpha-refpoint] ; relative address
        add     eax, [ecx+beta -refpoint] ; relative address
        ret

get_thunk_ecx:  ; Function for reading instruction pointer
        mov     ecx, [esp]
        ret
```

The only instruction that can read the instruction pointer in 32-bit mode is the `call` instruction. In example 3.5 we are using `call get_thunk_ecx` for reading the instruction pointer (`eip`) into `ecx`. `ecx` will then point to the first instruction after the call. This is our reference point, named `refpoint`.

The Gnu compiler for 32-bit Mac OS X uses a slightly different version of this method:

```
Example 3.5b. Bad method!
funca:  ; This function returns alpha + beta
        call    refpoint                ; get eip on stack
refpoint:
        pop     ecx                     ; pop eip from stack
        mov     eax, [ecx+alpha-refpoint] ; relative address
        add     eax, [ecx+beta -refpoint] ; relative address
        ret
```

The method used in example 3.5b is bad because it has a call instruction that is not matched by a return. This will cause subsequent returns to be mispredicted on many CPUs. (See manual 3: "The microarchitecture of Intel, AMD and VIA CPUs" for an explanation of return prediction).

This method is commonly used in Mac systems, where the Mach-O file format supports references relative to an arbitrary point. Other file formats do not support this kind of reference, but it is possible to use a self-relative reference with an offset. The NASM and Gnu assemblers will do this automatically, while most other assemblers are unable to handle this situation. It is therefore necessary to use a NASM or Gnu assembler if you want to generate position-independent code in 32-bit mode with this method. The code may look strange in a debugger or disassembler, but it executes without any problems in 32-bit Linux, BSD and Windows systems. In 32-bit Mac systems, the loader may not identify the section of the target correctly unless you are using an assembler that supports the reference point method (I am not aware of any other assembler than the Gnu assembler that can do this correctly).

The GOT method would use the same reference point method as in example 3.5a for addressing the GOT, and then use the GOT to read the addresses of `alpha` and `beta` into other pointer registers. This is an unnecessary waste of time and registers because if we can access the GOT relative to the reference point, then we can just as well access `alpha` and `beta` relative to the reference point.

The pointer tables used in `switch`/`case` statements can use the same reference point for the table and for the pointers in the table:

```
; Example 3.6. Position-independent switch, 32 bit, NASM syntax
SECTION .data

jumptable: dd case1-refpoint, case2-refpoint, case3-refpoint

SECTION .text

global funcb:function
funcb:  ; This function implements a switch statement
        mov     eax, [esp+4]            ; function parameter
        call    get_thunk_ecx           ; get ecx = eip
refpoint:                               ; ecx points here
        cmp     eax, 3
        jnb     case_default            ; index out of range
        mov     eax, [ecx+eax*4+jumptable-refpoint] ; read table entry
; The jump addresses are relative to refpoint, get absolute address:
        add     eax, ecx
        jmp     eax                     ; jump to desired case

case1:  ...
        ret
case2:  ...
        ret
case3:  ...
```

```
        ret
case_default:
        ...
        ret

get_thunk_ecx:  ; Function for reading instruction pointer
        mov     ecx, [esp]
        ret
```

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: RIP-relative, 32-bit absolute, 64-bit absolute, and relative to a base register.

RIP-relative addressing

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 3.7a. RIP-relative memory operand, NASM syntax
default rel
mov eax, [mem]

; Example 3.7b. RIP-relative memory operand, MASM syntax
mov eax, [mem]

; Example 3.7c. RIP-relative memory operand, Gas/Intel syntax
mov eax, [mem+rip]
```

The MASM assembler always generates RIP-relative addresses for static data when no explicit base or index register is specified. On other assemblers you must remember to specify relative addressing.

32-bit absolute addressing in 64 bit mode

A 32-bit constant address is sign-extended to 64 bits. This addressing mode works only if it is certain that all addresses are below 2^{31} (or above -2^{31} for system code).

It is safe to use 32-bit absolute addressing in Linux and BSD main executables, where all addresses are below 2^{31} by default, but it cannot be used in shared objects. 32-bit absolute addresses will often work in Windows main executables as well (but not DLLs), but no Windows compiler is using this possibility because .exe files may be relocated to higher addresses in special cases.

32-bit absolute addresses cannot be used in Mac OS X, where addresses are above 2^{32} by default.

Note that NASM and Gnu assemblers can make 32-bit absolute addresses when you do not explicitly specify rip-relative addresses. You have to specify `default rel` in NASM or `[mem+rip]` in Gas to avoid 32-bit absolute addresses.

There is absolutely no reason to use absolute addresses for simple memory operands. Rip-relative addresses make instructions shorter, they eliminate the need for relocation at load time, and they are safe to use in all systems.

Absolute addresses are needed only for accessing arrays where there is an index register, e.g.

```
; Example 3.8. 32 bit absolute addresses in 64-bit mode
mov al,  [abs chararray + rsi]
mov ebx, [abs intarray  + rsi*4]
```

This method can be used only if the address is guaranteed to be < 2^{31}, as explained above. See below for alternative methods of addressing static arrays.

The MASM assembler generates absolute addresses only when a base or index register is specified together with a memory label as in example 3.8 above.

The index register should preferably be a 64-bit register, not a 32-bit register. Segmentation is possible only with `FS` or `GS`.

64-bit absolute addressing

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.

```
; Example 3.9. 64 bit absolute address, NASM syntax
mov eax, dword [qword a]
```

This addressing mode is not supported by the MASM assembler, but it is supported by most other assemblers.

Addressing relative to 64-bit base register

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

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

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

```
; Example 3.10. Base register addressing in 64 bit mode
mov  eax, [rsi]
add  eax, [rsp + 4*rcx + 8]
```

This addressing mode is used for data on the stack, for structure and class members and for arrays.

Addressing static arrays in 64 bit mode

It is not possible to access static arrays with RIP-relative addressing and an index register. There are several possible alternatives.

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

```
// Example 3.11a. 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 simplest solution is to use 32-bit absolute addresses. This is possible as long as the addresses are below 2^{31}.

```
; Example 3.11b. Use 32-bit absolute addresses
; 64 bit Linux only
; Assumes that image base < 80000000H

SECTION .bss
A:   resd 100           ; Define static array A
B:   resd 100           ; Define static array B

SECTION .text
xor  ecx, ecx           ; i = 0

TOPOFLOOP:              ; Top of loop
mov  eax, [abs A+rcx*4] ; 32-bit address + scaled index
neg  eax
mov  [abs 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 3.11b because it cannot combine a RIP-relative address with an index register.

This method is used in 64-bit Linux to access static arrays. It is not used by any compiler for 64-bit Windows because it is not 100% safe. The image base is typically 2^{22} for application programs and between 2^{28} and 2^{29} for DLL's in Windows, so this method will work in most cases, but not all. This method cannot be used in 64-bit Mac systems because all addresses are above 2^{32} by default.

The second method is to use image-relative addressing. The following solution loads the image base into register `RBX` by using a `LEA` instruction with a `RIP`-relative address:

```
; Example 3.11c. Address relative to image base
; 64 bit, Windows only, MASM assembler
.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
```

This method is used in 64 bit Windows only and requires the MASM assembler. In Linux, the image base is available as `__executable_start`, but image-relative addresses are not supported in the ELF file format. The Mach-O format allows addresses relative to an arbitrary reference point, including the image base, which is available as `__mh_execute_header`.

The third 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 3.11d.
```

```
; Load address of array into base register
; Works in all 64-bit systems. NASM syntax

default rel           ; Use RIP relative addressing when possible

SECTION .bss
A: resd 100           ; Define static array A
B: resd 100           ; Define static array B

SECTION .text

lea  rbx, [A]         ; Load 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 non-negative and less than 2^{32}. This method can use any address in the data segment as a reference point and calculate other addresses relative to this reference point.

If an array is more than 2^{31} bytes away from the instruction pointer, then we have to load the full 64 bit address into a base register. For example, we can replace `lea rbx,[A]` with `mov rbx, qword A` in example 3.11d.

Position-independent code in 64-bit mode

Position-independent code is easy to make in 64-bit mode. Static data can be accessed with rip-relative addressing. Static arrays can be accessed as in example 3.11d.

The pointer tables of switch statements can be made relative to an arbitrary reference point. It is convenient to use the table itself as the reference point:

```
; Example 3.12. switch with relative pointers, 64 bit, NASM syntax

SECTION .data
jumptable: dd case1-jumptable, case2-jumptable, case3-jumptable

SECTION .text
default rel                             ; use relative addresses

global funcb:function
funcb:  ; This function implements a switch statement
        ; The first function parameter is ecx in Windows, edi in Linux
        mov     eax, ecx                ; function parameter
        cmp     eax, 3
        jnb     case_default            ; index out of range
        lea     rdx, [jumptable]        ; address of table
        movsxd  rax, dword [rdx+rax*4]  ; read table entry
; The jump addresses are relative to jumptable, get absolute address:
        add     rax, rdx
        jmp     rax                     ; jump to desired case

case1:  ...
        ret
case2:  ...
        ret
```

```
case3:  ...
        ret
case_default:
        ...
        ret
```

This method can be useful for reducing the size of long pointer tables because it uses 32-bit relative pointers rather than 64-bit absolute pointers.

The MASM assembler cannot generate the relative tables in example 3.12 unless the jump table is placed in the code segment. It is preferred to place the jump table in the data segment for optimal caching and code prefetching, and this can be done with the NASM or Gnu assembler.

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 3.12 below.

2. Opcode (1-3 bytes)
   This is the instruction code. It can have these forms:
   Single byte: XX
   Two bytes: 0F XX
   Three bytes: 0F 38 XX or 0F 3A XX
   Three bytes opcodes of the form 0F 38 XX always have a mod-reg-r/m byte and no displacement. Three bytes opcodes of the form 0F 3A XX always have a mod-reg-r/m byte and 1 byte displacement.

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 (Intel NetBurst only)	3Eh	3Eh	3Eh

Predict branch not taken (Intel NetBurst only)	2Eh	2Eh	2Eh
Preserve bounds register on jump (MPX)	F2h	F2h	F2h
VEX prefix, 2 bytes	C5h	C5h	C5h
VEX prefix, 3 bytes	C4h	C4h	C4h
XOP prefix, 3 bytes (AMD only)	8Fh	8Fh	8Fh
EVEX prefix, 4 bytes (AVX-512)	62h	62h	62h
MVEX prefix, 4 bytes (Intel Knights Corner only)	n.a.	62h	62h

Table 3.12. 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 work only on Intel NetBurst (Pentium 4) 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 codes 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 prefixes and the multi-byte prefixes (VEX, XOP, EVEX, MVEX) which must come after any other prefixes.

The AVX instruction set uses 2- and 3-byte prefixes called VEX prefixes. The VEX prefixes include bits to replace all 66, F2, F3 and REX prefixes as well as the 0F, 0F 38 and 0F 3A escape bytes of multibyte opcodes.. VEX prefixes also include bits for specifying YMM registers, an extra register operand, and bits for future extensions. The EVEX and MVEX prefixes are similar to the VEX prefixes, with extra bits for supporting more registers, masked operations, and other features. No additional prefixes are allowed after a VEX, EVEX or MVEX prefix. The only prefixes allowed before a VEX, EVEX or MVEX prefix are segment prefixes and address size prefixes.

Meaningless, redundant or misplaced prefixes are ignored, except for the LOCK and VEX prefixes. 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 on some processors.

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, but it may take longer 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

	16 bit DOS, Windows	32 bit Windows, Unix	64 bit Windows	64 bit Unix
Registers that can be used freely	AX, BX, CX, DX, ES, ST(0)-ST(7)	EAX, ECX, EDX, ST(0)-ST(7), XMM0-XMM7, YMM0-YMM7, ZMM0-ZMM7	RAX, RCX, RDX, R8-R11, ST(0)-ST(7), XMM0-XMM5, YMM0-YMM5, YMM6H-YMM15H, ZMM16-ZMM31	RAX, RCX, RDX, RSI, RDI, R8-R11, ST(0)-ST(7), XMM0-XMM15, YMM0-YMM15, ZMM16-ZMM31
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, YMM0-YMM3, ZMM0-ZMM3	RDI, RSI, RDX, RCX, R8, R9, XMM0-XMM7, YMM0-YMM7, ZMM0-ZMM7
Registers used for return values	AX, DX, ST(0)	EAX, EDX, ST(0)	RAX, XMM0, YMM0, ZMM0	RAX, RDX, XMM0, XMM1, YMM0, ZMM0, ST(0), ST(1)

Table 4.1. Register usage

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 YMM registers must be cleared by VZEROUPPER before any call or return to non-VEX code.

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

Function calling conventions are described in detail in manual 5: "Calling conventions for different C++ compilers and operating systems". The most important rules are outlined here.

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

Table 4.2. Calling conventions in 32 bit mode

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 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. It is possible to specify different alignments, and this can lead to incompatibilities. 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, __m256, __m512).

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, __m256, __m512).

Calling conventions in 64 bit Linux, BSD and Mac OS X

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`, `__m256`, `__m512`).

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 16- and 32-bit Windows and 32- and 64-bit 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. You can avoid name mangling in these cases by defining a mangled function that calls an unmangled function. If the mangled function is defined as inline then the compiler will simply replace the call to the mangled function by the call to the unmangled function. For example, to define an overloaded C++ operator in assembly without name mangling:

```
class C1;
// unmangled assembly function;
extern "C" C1 cplus  (C1 const & a, C1 const & b);
// mangled C++ operator
inline C1 operator + (C1 const & a, C1 const & b) {
   // operator + replaced inline by function cplus
   return cplus(a, b);
}
```

Overloaded functions can be inlined in the same way. Class member functions can be translated to friend functions as illustrated in example 7.1b page 49.

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. 32-bit Windows
extern     _sin
global     _sinxpnx:function
align      4
_sinxpnx:
; parameter x = [esp+4]
; parameter n = [esp +12]
```

```
; return value = st(0)

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

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 4.1c. 32-bit Linux
extern     _sin
global     sinxpnx:function
align      4
sinxpnx:
; parameter x = [esp+4]
; parameter n = [esp +12]
; return value = st(0)

    fld   qword  [esp+4]     ; x
    sub   esp, 12            ; keep stack aligned by 16 before call
    fstp  qword  [esp]       ; store parameter for sin; clear st(0)
    call  _sin               ; library function for sin()
    add   esp, 12            ; clean up stack after call
    fild  dword  [esp+12]    ; n
    fmul  qword  [esp+4]     ; n*x
    fadd                     ; sin(x) + n*x
    ret                      ; return value is in st(0)
```

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 4.1b we subtracted only 8 from `ESP` because the stack is only aligned by 4 in 32-bit Windows.

```
; Example 4.1d. 64-bit Windows
extern    sin
global    sinxpnx:function
align     4
sinxpnx:
    ; parameter x = xmm0
    ; parameter n = edx
    ; return value = xmm0

    push     rbx             ; rbx must be saved
    movaps   [rsp+16],xmm6   ; save xmm6 in my shadow space
    sub      rsp, 32         ; 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
```

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 and we can rely on the stack being aligned by 16 before the call to sinxpnx. 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 so that the proper alignment is preserved. The extra 32 bytes on the stack is shadow space for the call to sin. Note that example 4.1d 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 4.1e. 64-bit Linux
extern    sin
global    sinxpnx:function
align     4

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
```

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 4.1e 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.

5 Using intrinsic functions in C++

As already mentioned, there are several different ways of combining assembly with high-level language code: using inline assembly in C++, using intrinsic functions in C++, using vector classes in C++, and making separate assembly modules. Inline assembly is described in chapter 6, page 36. Intrinsic functions are described in this chapter. Vector classes are described in manual 1: "Optimizing software in C++".

Intrinsic functions and vector classes are highly recommended because they are much easier and safer to use than assembly language syntax. The Microsoft, Intel, Gnu and Clang C++ compilers have support for 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.
- The code is compatible with Microsoft, Gnu, Clang and Intel compilers.
- The compiler takes care of register variables, register allocation and register spilling. The programmer does not 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, so that 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 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 Clang and Gnu 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.

- Some compilers are not able to optimize code containing intrinsic functions as much as it optimizes 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.

- Mixture of `__m128` and `__m256` types can cause severe delays if the programmer does not follow certain rules. Call `_mm256_zeroupper()` before any transition from modules compiled with AVX enabled to modules or library functions compiled without AVX.

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 instructions on vector 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 in Microsoft, Gnu, Clang, and Intel compilers. Most intrinsic functions have the same names in all compilers. You have to include an appropriate header file to get access to the intrinsic functions. Some intrinsic functions are not supported by all these compilers.

The intrinsic functions are listed in the help documentation for each compiler, in the appropriate header files, in docs.microsoft.com, in "Intel 64 and IA-32 Architectures Software Developer's Manual" (developer.intel.com) and in "Intel Intrinsic Guide" at https://software.intel.com/sites/landingpage/IntrinsicsGuide/

6 Using inline assembly

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 Gnu, Clang, or 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.
- You may inadvertently mix VEX and non-VEX instructions, whereby large penalties are incurred on some CPUs (see chapter 13.1).
- 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 syntax for inline assembly in Microsoft and Intel compilers for Windows is a MASM-style syntax. Unfortunately, the syntax for inline assembly is poorly documented or not documented at all in the compiler manuals. I will therefore briefly describe the syntax here.

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 much 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 in 32-bit Microsoft or Intel compiler:

```
// 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, Intel compiler
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 Windows
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 in MASM style inline assembly. 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 (`//`).

Hard-coded opcodes are made with `_emit` followed by a byte constant, where you would use `DB` in MASM. For example `_emit 0x90` is equivalent to `NOP`.

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 ?). 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.

6.2 Gnu style inline assembly

Inline assembly works quite differently on the Gnu and Clang compilers because these compilers compile via assembly rather than producing object code directly. 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, the Clang compiler, and the Intel compiler for Linux in both 32-bit and 64-bit mode.

AT&T syntax

Let us 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 did not 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 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. We 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 51).
- Good assemblers have powerful macro features that opens up metaprogramming possibilities that are absent in most compiled high-level languages (see page 103).

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 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.
- It is difficult to remember which instructions belong to which instruction set extensions. Using a wrong instruction can cause the program to crash.
- You may inadvertently mix VEX and non-VEX vector instructions. This incurs a large performance penalty on come Intel CPUs (see chapter 13.1).
- 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 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 possibly not 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 50) 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 31ff.

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 50) 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 only the necessary parts of the library. A dynamic link library (`*.dll`) is distributed as a separate file which is loaded at runtime by the executable file in Windows systems. Linux and Mac OS use a slightly different type of dynamic libraries called shared objects (`*.so`). These are described in the next section.

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#, and 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 if different versions have the same name. 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.

7.3 Shared object libraries

Linux, BSD, and Mac OS use shared objects (*.so) rather than DLLs. Shared objects are very similar to DLLs and used in the same way.

It is possible to override a symbol name in a shared object just like in a static library. This makes it different from a DLL. This feature, called symbol interposition, is rarely used and it comes at a high cost. All functions with public visibility are accessed through a procedure linkage table (PLT), and all variables and data objects with public visibility are accessed through a global offset table (GOT). Static variables with only local visibility are also accessed through the GOT in 32-bit mode. These complications make shared objects less

efficient than static link libraries. See the section "Position-independent code" in manual 1: "Optimizing software in C++" for a further discussion.

7.4 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 pure 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 less so if it uses inline assembly. 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. 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 RAM memory rather than transferred in registers.

7.5 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 does not have classes and member functions. The most logical solution is to use the mangled function name. Returning to example 6.2a and b page 39, 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

section .text
```

```
global ?Sum@MyList@@QAEHXZ:function

; Define structure corresponding to class MyList:
struc    MyList
length:  resd 1                  ; int length
buffer:  resd 100                ; int buffer[100];
endstruc

; int MyList::Sum()
; Mangled function name compatible with Microsoft compiler (32 bit):
?Sum@MyList@@QAEHXZ:
;32-bit Windows has 'this' pointer in ECX
      xor eax, eax               ; sum = 0
      xor edx, edx               ; Loop index i = 0
      cmp dword [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
```

The mangled function name `?Sum@MyList@@QAEHXZ` is compiler specific. Gnu and Clang compilers 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. The pointer will be on the stack in 32-bit mode, and in `rcx` or `rdi` in 64-bit mode.

7.6 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 cores. It is 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 a 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.

7.7 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, NMAKE syntax
mylibrary.zip: mylibrary.lib mylibrary.h
   zip $@ $?
```

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

.asm.obj
    nasm -f win32 -o $@.obj $*.asm
    # or MASM: ml /c /Cx /coff /Fo$@ $*.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 by a tab, specifies the command needed for building the target file `mylibrary.zip` from its dependents `mylibrary.lib` and `mylibrary.h`. The next two lines tell how to build the library file `mylibrary.lib` from the three object files. The line `.asm.obj` is a generic rule. It 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 for makefiles can use the following macros for specifying file names:

	Gnu make	MS nmake
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	$^ $+	$**
All dependents with a later timestamp than the current target	$?	$?

Table 7.1. Makefile macros

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

The easiest solution to these portability problems is to make the code in a high level language such as C++ and make any necessary low-level constructs with the use of intrinsic functions or inline assembly. The code can then be compiled with different compilers for the different platforms. Gnu and Clang C++ compilers can build code for

almost any platform, using the same syntax for inline assembly and intrinsic functions on all x86 platforms. Other compilers may differ.

If full assembly programming is necessary or desired, then there are various methods for overcoming the compatibility problems between different x86 platforms. These methods 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 31.

The `extern "C"` directive cannot be used for class member functions, overloaded functions and operators. This problem can be used by making an inline function with a mangled name to call an assembly function with an unmangled name:

```
// Example 8.1. Avoid name mangling of overloaded functions in C++
// Prototypes for unmangled assembly functions:
extern "C" double power_d (double x, double n);
extern "C" double power_i (double x, int n);

// Wrap these into overloaded functions:
inline double power (double x, double n) {return power_d(x, n);
inline double power (double x, int n)    {return power_i(x, n);
```

The compiler will simply replace a call to the mangled function with a call to the appropriate unmangled assembly function without any extra code. The same method can be used for class member functions, as explained on page 49.

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 4.1b page 31, we can add mangled names for multiple compilers in the following way:

```
; Example 8.2. (Example 4.1b rewritten)
; Function with multiple mangled names (32-bit mode)

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

section .text

global  _sinxpnx:function, ?sinxpnx@@YANNH@Z:function
global  _Z7sinxpnxdi:function, __Z7sinxpnxdi:function

ALIGN     4

; Make public names for each name mangling scheme:
_sinxpnx:                       ; extern "C" name
?sinxpnx@@YANNH@Z:              ; Microsoft compiler
_Z7sinxpnxdi:                   ; Gnu compiler for Linux
__Z7sinxpnxdi:                  ; Gnu compiler for Windows and Mac OS

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

    fild  dword [esp+12]        ; n
    fld   qword [esp+4]         ; x
```

```
        fmul  st1, st0              ; n*x
        fsin                        ; sin(x)
        fadd                        ; sin(x) + n*x
        ret
```

Example 8.2 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 to generate assembly output. Alternatively, compile to an object file and then disassemble. Command line versions of most compilers are available for free or as trial versions.

The Intel compiler for Windows is compatible with the Microsoft name mangling scheme. The Intel and Clang compilers for Linux are compatible with the Gnu name mangling scheme. Gnu and Clang compilers for Windows may be able to use the Microsoft name mangling scheme.

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. One solution is to use friend functions as explained on page 49. Another possibility is to make a function with multiple entries. The following example is a rewrite of example 7.1a page 48 with two entries for the two different calling conventions:

```
; Example 8.3a (Example 7.1a with two entries)
; Member function, 32-bit mode
; int MyList::Sum()

section .text

global _MyList_Sum:function, @MyList@Sum$qv:function
global _ZN6MyList3SumEv:function, __ZN6MyList3SumEv:function
global ?Sum@MyList@@QAEHXZ:function

; Define structure corresponding to class MyList:
struc    MyList
length:  resd 1                  ; int length
buffer:  resd 100                ; int buffer[100];
endstruc

_MyList_Sum:                     ; for extern "C" friend function

; Make mangled names for compilers with __cdecl convention:
_ZN6MyList3SumEv:                ; Gnu comp. for Linux
__ZN6MyList3SumEv:               ; Gnu comp. for Windows and Mac OS

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

; Make mangled names for compilers with __thiscall convention:
?Sum@MyList@@QAEHXZ:             ; Microsoft compiler
      xor eax, eax               ; sum = 0
      xor edx, edx               ; Loop index i = 0
      cmp dword [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
```

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 39. 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 8.3b
; void MyList::AttItem(int item);

section .text

global _MyList_AttItem:function, _ZN6MyList7AttItemEi:function
global __ZN6MyList7AttItemEi:function,
?AttItem@MyList@@QAEXH@Z:function

; Define structure corresponding to class MyList:
struc    MyList
length:  resd 1                     ; int length
buffer:  resd 100                   ; int buffer[100];
endstruc

; Function entries
_MyList_AttItem:                    ; for extern "C" friend function

; Make mangled names for compilers with __cdecl convention:
_ZN6MyList7AttItemEi:               ; Gnu comp. for Linux
__ZN6MyList7AttItemEi:              ; Gnu comp. for Windows and Mac OS

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

; Make mangled names for compilers with __thiscall convention:
?AttItem@MyList@@QAEXH@Z:           ; Microsoft compiler
        ; __thiscall requires stack cleanup by function:
        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

        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
```

In example 8.3b, 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 must remove parameters from the stack before the common section because the `__thiscall` convention requires stack cleanup by the function.

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 for several other programming languages.

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

The following example shows the function from example 8.2 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 8.4a (Example 8.2 with __stdcall and __cdecl entries)
    ; Function with entries for __stdcall and __cdecl (32-bit Windows):

section .text

global _sinxpnx@12:function, _sinxpnx:function

align     4
; __stdcall entry:
; extern "C" double __stdcall sinxpnx (double x, int n);
_sinxpnx@12:
    ; Get all parameters into registers
    fild  dword [esp+12]     ; n
    fld   qword [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:
    ; Get all parameters into registers
    fild  dword [esp+12]     ; n
    fld   qword [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  st1, st0           ; n*x
    fsin                     ; sin(x)
    fadd                     ; sin(x) + n*x
    ret                      ; return value is in st(0)
```

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 8.4b
```

```
; Function with versions for __stdcall and __cdecl (32-bit Windows)
section .text

; Choose function prolog according to calling convention:
%ifdef STDCALL                ; If STDCALL_ is defined
global _sinxpnx@12:function
_sinxpnx@12:                  ; extern "C" __stdcall function name
%else
global _sinxpnx:function
_sinxpnx:                     ; extern "C" __cdecl function name
%endif

; Function body common to both calling conventions:
    fild  dword [esp+12]      ; n
    fld   qword [esp+4]       ; x
    fmul  st1, st0            ; 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
%else
    ret                       ; Don't clean up stack if __cdecl
%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. 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 before any call, 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 any integer parameters.

Let us use example 4.1 page 31 once more and make an implementation that works in both 64-bit Windows and 64-bit Linux.

```
; Example 8.5a (Example 4.1d/e combined).
; Support for both 64-bit Windows and 64-bit Linux
; double sinxpnx (double x, int n) {return sin(x) + n * x;}

section .text
global _Z7sinxpnxdi, ?sinxpnx@@YANNH@Z
extern    sin

align     8

; 64-bit Linux entry:
_Z7sinxpnxdi:                ; 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:           ; Microsoft name mangling

    ; 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
```

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.

The old Borland, Digital Mars, and 16-bit Microsoft compilers use the OMF format for object files. Contemporary compilers for 32-bit Windows use the COFF format, also called PE32. Gnu and Intel compilers under 32-bit Linux prefer the ELF32 format. Gnu and Intel compilers for Mac OS X use the 32- and 64-bit Mach-O format. All compilers for 64-bit

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

The NASM assembler can produce OMF, COFF/PE32, ELF32/64, COFF/PE32+ and MachO32/64 formats. The Gnu assembler (Gas) can produce ELF32/64 and MachO32/64 formats. The MASM assembler can produce both OMF, COFF/PE32 and COFF/PE32+ format object files, but not ELF format.

It is possible to do cross-platform development if you have an assembler that supports all the object file formats you need or a suitable object file conversion utility. This is useful for making function libraries that work on multiple platforms. An object file converter and cross-platform library manager named objconv is available from www.agner.org/optimize.

The objconv utility can change function names in the object file as well as converting to a different object file format. This may help solve incompatible name mangling schemes. Repeating example 8.5 without name mangling:

```
; Example 8.5b.
; Support for both 64-bit Windows and 64-bit Unix systems.
; double sinxpnx (double x, int n) {return sin(x) + n * x;}

section .text
global Unix_sinxpnx, Win_sinxpnx
extern    sin

align     8

; 64-bit Linux entry:
Unix_sinxpnx:

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

; 64-bit Windows entry:
Win_sinxpnx:
; 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
```

This function can now be assembled and converted to multiple file formats with the following commands:

```
nasm -f win64 -o sinxpnx.obj sinxpnx.nasm
objconv -cof64 -np:Win_:   sinxpnx.obj sinxpnx_win.obj
objconv -elf64 -np:Unix_:  sinxpnx.obj sinxpnx_linux.o
objconv -mac64 -np:Unix_:_ sinxpnx.obj sinxpnx_mac.o
```

The first line assembles the code using the NASM assembler and produces a COFF object file.

The second line replaces "Win_" with nothing in the beginning of function names in the object file. The result is a COFF object file for 64-bit Windows where the Windows entry for our function is available as `extern "C" double sinxpnx(double x, int n)`. The name `Unix_sinxpnx` for the Unix entry is still unchanged in the object file, but is not used.
The third line converts the file to ELF format for 64-bit Linux and BSD, and replaces "Unix_" with nothing in the beginning of function names in the object file. This makes the Unix entry for the function available as `sinxpnx`, while the unused Windows entry is `Win_sinxpnx`.
The fourth line does the same for the MachO file format, and puts an underscore prefix on the function name, as required by Mac compilers.

Objconv can also build and convert static library files (`*.lib`, `*.a`). This makes it possible to build a multi-platform function library on a single source platform.

An example of using this method is the multi-platform function library `asmlib.zip` available from www.agner.org/optimize/. `asmlib.zip` includes a makefile (see page 50) that makes multiple versions of the same library by using the object file converter objconv.

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 make a dynamic link library instead. Delphi Pascal may have problems linking to object files - it is easier to use a DLL.

Calling assembly code from Java is quite complicated. You have to compile the code to a DLL or shared object, and use the Java Native Interface (JNI) or Java Native Access (JNA).

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 other 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 Java or 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 code 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 do not 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 153 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 fourth 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 CPUs 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.

Partial registers

Some CPUs can keep different parts of a register separate, while other CPUs 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. Some CPUs from both Intel, AMD and VIA 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 CPUs are able to split the register into parts in order to avoid the false dependence, but this has another disadvantage in case the two parts have to be joined together again. Assume, for example, that the code in example 9.1c is followed by `PUSH EAX`. On some processors, this instruction has to wait for the two parts of `EAX` to retire in order to join them together, at the cost of 5-6 clock cycles. Other processors will generate an extra µop for joining the two parts of the register together.

These problems are 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.

The flags register can cause similar problems for instructions that modify some of the flag bits and leave other bits unchanged. For example, the `INC` and `DEC` instructions leave the carry flag unchanged but modifies the zero and sign flags.

Micro-operations

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 µops 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`, not for `mov [esp],eax`.

Execution units

Out-of-order execution becomes even more efficient when the CPU can do more than one thing at the same time. Many CPUs can do two, three, or more 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. Modern CPUs have at least two integer ALU's (Arithmetic Logic Units) so that they can do two or more integer additions per clock cycle. Modern CPUs may have two floating point arithmetic units as well so that it may be possible to do two floating point operations at the same time. There may be one or two memory read units 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 µops per clock cycle is three or four on many processors. It may be 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, three, or four µops per clock cycle, depending on the CPU.

Pipelined instructions

Floating point operations typically take more than one clock cycle, but they are often pipelined so that e.g. a new floating point addition can start before the previous addition is finished. Vector instructions are using the floating point execution units even for integer instructions on most CPUs. 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, AMD and VIA CPUs" and manual 4: "Instruction tables".

Summary

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 general purpose registers, the stack pointer, the flags register, floating point registers, and vector registers. Some CPUs 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 are inefficient on some CPUs 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, especially if they are followed by an instruction that reads the flags.
- A chain of instructions where each instruction depends on the previous one cannot execute out of order. Avoid long dependency chains. (See page 64).
- Memory operands cannot be renamed.
- A memory read can execute before a preceding memory write to a different address in most cases. 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 may not be able to execute before a preceding write, but the write buffers can hold a number of pending writes, typically four or more.

- A memory read can execute before or simultaneously with another preceding read on most 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 or vector instructions, 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 72). The address of memory operands can possibly be loaded into pointer registers outside of a loop if instruction 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.

Most processors fetch instructions in aligned 16-byte or 32-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. Alternatively, make sure that there is no 16-byte boundary in the first few instructions after a critical loop entry or subroutine entry.

Instruction decoding is often a bottleneck. The organization of instructions that gives the optimal decoding is processor-specific. On AMD processors it is preferred to avoid instructions that generate more than 2 µops.

Instructions with multiple prefixes can slow down decoding. There is a maximum number of prefixes that an instruction can have without slowing down decoding, depending on the CPU. Avoid address size prefixes. Avoid operand size prefixes on instructions with an immediate operand. For example, it is preferred to replace `MOV AX,2` by `MOV EAX,2`.

Some CPUs have a µop cache or a tiny loopback buffer that helps remove the bottleneck of instruction decoding in small loops. Keep loops small so that they are likely to fit into the µop cache or loopback buffer. Avoid unnecessary loop unrolling.

Older 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.

All these details are processor-specific. See manual 3: "The microarchitecture of Intel, AMD and VIA CPUs" for details.

9.4 Instruction latency and throughput

The latency of an instruction is the number of clock cycles it takes from the time the instruction starts to execute till the result is ready. The time it takes to execute a dependency 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 the time 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 Core 2 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, AMD and VIA.

The following list shows some typical values.

Instruction	Typical latency	Typical reciprocal throughput
Integer move	1	0.25-0.5
Integer addition	1	0.25-0.5
Integer Boolean	1	0.25-1
Integer shift	1	0.33-1
Integer multiplication	3-10	1
Integer division	20-80	20-40
Floating point addition	3-6	0.5-1
Floating point multiplication	4-8	0.5-2
Floating point division	20-45	20-45
Integer vector addition	1-2	0.5-2
Integer vector multiplication	3-7	1-2
Floating point vector addition	3-5	0.5-2
Floating point vector multiplication	4-7	0.5-2
Floating point vector division	20-60	20-60
Memory read (cached)	3-4	0.5-1
Memory write (cached)	3-4	1
Jump or call	0	1-2

Table 9.1. Typical instruction latencies and throughputs

9.5 Break dependency chains

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

```
// Example 9.3a, Loop-carried dependency 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 dependency chain. A loop-carried dependency 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 dependency chains running in parallel and each dependency chain is one fourths as long as the original one. The optimal number of accumulators is the latency of the instruction (in this case floating point addition), divided by the reciprocal throughput. See page 64 for examples of assembly code for loops with multiple accumulators.

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

Some microprocessors can execute four or five instructions per clock cycle. The more instruction-level parallelism the microprocessor supports, the more important it is to avoid long dependency chains.

Dependency chains occur not only in loops but also in linear code. Such dependency 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 66). Which implementation is optimal depends on how predictable the branch is and how long the dependency 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. 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. Most modern processors will recognize an `XOR` instruction with two identical input registers as independent of the prior value for 32-bit and 64-bit general purpose registers, as well as vector registers of 128 bits or more.

You should not break a dependence by an 8-bit or 16-bit part of a register. For example `XOR AX,AX` breaks a dependence on some processors, but not all. But `XOR EAX,EAX` is sufficient for breaking the dependence on `RAX` in 64-bit mode.

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 some Intel processors. This can be removed in the following way:

```
; Example 9.4, Break dependence on flags
ror  eax, 1
xor  edx, edx    ; Remove false dependence on the flags
ror  ebx, 1
```

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:

- The fetching of code after an unconditional jump or a taken conditional jump is delayed by typically 1-3 clock cycles, depending on the microprocessor. The delay is worst if the target is near the end of a 16-bytes or 32-bytes code fetch block (i.e. before an address divisible by 16).

- The code cache becomes fragmented and less efficient when jumping around between noncontiguous subroutines.

- Microprocessors with a µop cache or trace cache are likely to store multiple instances of the same code in this 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 will cost 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 16.

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

All modern CPUs have an execution pipeline that contains stages for instruction prefetching, decoding, register renaming, µop reordering and scheduling, execution, retirement, etc. The number of stages in the pipeline range from 12 to 22, depending on the specific micro-architecture. When a branch instruction is fed into the pipeline then the CPU does not know for sure which instruction is the next one to fetch into the pipeline. It takes 12-22 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, AMD and VIA CPUs".

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 there are 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. Older processors will simply predict an indirect jump or call to go the same way as it did last time. Many newer processors are able to recognize simple repetitive patterns for indirect jumps.

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 more 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, except in tail calls as explained below. And do not use the `RET` instruction as an indirect jump. Far calls should be matched with far returns.

Make conditional jumps most often not taken

The efficiency and throughput for not-taken branches is better than for taken branches on most processors. Therefore, it is good to place the most frequent branch first:

```
; Example 9.5, Place most frequent branch first
Func1:
    cmp eax,567
    je L1
    ; frequent branch
    ret
L1: ; rare branch
    ret
```

Tail calls

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

```
; Example 9.6a, call/ret sequence (32-bit)
Func1:
    ...
    call  Func2
```

```
    ret
```

This can be changed to:

```
; Example 9.6b, call+ret replaced by jmp
Func1:
    ...
    jmp   Func2  ; tail call
```

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 9.7a, call/ret sequence (64-bit Windows or Linux)
Func1:
    sub   rsp, 8           ; Align stack by 16
    ...
    call  Func2            ; This call can be eliminated
    add   rsp, 8
    ret
```

This can be changed to:

```
; Example 9.7b, call+ret replaced by jmp with stack aligned
Func1:
    sub   rsp, 8
    ...
    add   rsp, 8           ; Restore stack pointer before jump
    jmp   Func2
```

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 9.8a, Function with jump that can be eliminated
FuncA:
        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
```

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

```
; Example 9.8b, Loop epilog copied to eliminate jump
```

```
FuncA:
        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
```

In example 9.8b, 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 9.8b, Function epilog copied to eliminate jump
FuncA:
        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
```

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.

Replacing conditional jumps with conditional moves

The most important jumps to eliminate are conditional jumps, especially if they are poorly predicted. Example:

```
// Example 9.9a. C++ branch to optimize
a = b > c ? d : e;
```

This can be implemented with either a conditional jump or a conditional move:

```
        ; Example 9.9b. Branch implemented with conditional jump
        mov     eax, [b]
        cmp     eax, [c]
        jng     L1
        mov     eax, [d]
        jmp     L2
L1:     mov     eax, [e]
L2:     mov     [a], eax

        ; Example 9.9c. Branch implemented with conditional move
        mov     eax, [b]
        cmp     eax, [c]
        mov     eax, [d]
        cmovng  eax, [e]
        mov     [a], eax
```

The advantage of a conditional move is that it avoids branch mispredictions. But it has the disadvantage that it increases the length of a dependency chain, while a predicted branch breaks the dependency chain. If the code in example 9.9c is part of a dependency chain then the `cmov` instruction adds to the length of the chain. If the same code is implemented with a conditional jump as in example 9.9b and if the branch is predicted correctly, then the result does not have to wait for `b` and `c` to be ready. It only has to wait for either `d` or `e`, whichever is chosen. This means that the dependency chain is broken by the predicted branch, while the implementation with a conditional move has to wait for both `b`, `c`, `d` and `e` to be available. If `d` and `e` are complicated expressions, then both have to be calculated when the conditional move is used, while only one of them has to be calculated if a conditional jump is used.

As a rule of thumb, we can say that a conditional jump is faster than a conditional move if the code is part of a dependency chain and the prediction rate is better than 75%. A conditional jump is also preferred if we can avoid a lengthy calculation of `d` or `e` when the other operand is chosen.

Loop-carried dependency chains are particularly sensitive to the disadvantages of conditional moves. For example, the code in example 12.14a on page 104 works more efficiently with a branch inside the loop than with a conditional move, even if the branch is poorly predicted. This is because the floating point conditional move adds to the loop-carried dependency chain and because the implementation with a conditional move has to calculate all the `power*xp` values, even when they are not used.

Another example of a loop-carried dependency chain is a binary search in a sorted list. If the items to search for are randomly distributed over the entire list then the branch prediction rate will be close to 50% and it will be faster to use conditional moves. But if the items are often close to each other so that the prediction rate will be better, then it is more efficient to use conditional jumps than conditional moves because the dependency chain is broken every time a correct branch prediction is made.

It is also possible to do conditional moves in vector registers on an element-by-element basis. See page 113ff 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.

Replacing conditional jumps with conditional set instructions

If a conditional jump is used for setting a Boolean variable to 0 or 1 then it is often more efficient to use the conditional set instruction. Example:

```
// Example 9.10a. Set a bool variable on some condition
int b, c;
bool a = b > c;

; Example 9.10b. Implementation with conditional set
mov    eax, [b]
cmp    eax, [c]
setg   al
mov    [a], al
```

The conditional set instruction writes only to 8-bit registers. If a 32-bit result is needed then set the rest of the register to zero before the compare:

```
; Example 9.10c. Implementation with conditional set, 32 bits
mov    eax, [b]
xor    ebx, ebx    ; zero register before cmp to avoid changing flags
cmp    eax, [c]
setg   bl
mov    [a], ebx
```

If there is no vacant register then use `movzx`:

```
; Example 9.10d. Implementation with conditional set, 32 bits
mov    eax, [b]
cmp    eax, [c]
setg   al
movzx  eax, al
mov    [a], eax
```

If a value of all ones is needed for true then use `neg eax`.

An implementation with conditional jumps may be faster than conditional set if the prediction rate is good and the code is part of a long dependency chain, as explained in the previous section (page 70).

Replacing conditional jumps with bit-manipulation instructions

It is sometimes possible to obtain the same effect as a branch by ingenious manipulation of bits and flags. The carry flag is particularly useful for bit manipulation tricks:

```
; Example 9.11, Set carry flag if eax is zero:
cmp  eax, 1

; Example 9.12, Set carry flag if eax is not zero:
neg  eax

; Example 9.13, Increment eax if carry flag is set:
adc  eax, 0

; Example 9.14, Copy carry flag to all bits of eax:
sbb  eax, eax

; Example 9.15, 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 9.16, 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 9.17a, 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 9.17b, 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 9.18a, Choose between two numbers
test  eax, eax
mov   edx, ecx
cmovs edx, ebx    ; = (a < 0) ? b : c
```

Conditional moves are not very efficient on older Intel processors. Alternative implementations may be faster in some cases. The following example gives the same result as example 9.18a.

```
; Example 9.18b, 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 9.18b may be faster than 9.18a on processors where conditional moves are inefficient. Example 9.18b destroys the value of `ebx`.

Whether these tricks are faster than a conditional jump depends on the prediction rate, as explained above.

10 Optimizing for size

The code cache can hold from 8 to 64 kb of code, as explained in chapter 11 page 81. If there are problems keeping the critical inner loop of the code within the code cache, the µop cache, or the loopback buffer, then you may consider reducing the size of the code. Reducing the code size can also improve the decoding of instructions. Very small loops are particularly fast because they can fit into a loopback buffer, or the decoder can reuse loaded code cache lines.

You may even want to reduce the size of the code at the cost of reduced speed if speed is not important.

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. The short form of INC and DEC is not available in 64-bit mode.

The following instructions take one byte less when they use the accumulator than when they use any other register: ADD, ADC, SUB, SBB, AND, OR, XOR, CMP, TEST with an immediate operand without sign extension. This also applies to the MOV instruction with a memory operand and no pointer register in 16 and 32 bit mode, but not in 64 bit mode. 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 (except ESP, RSP or R12) and a displacement than when they have a scaled index register, or both base pointer and index register, or ESP, RSP or R12 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 or R13 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].
```

Instructions in 64-bit mode need a REX prefix if at least one of the registers R8 - R15 or XMM8 - XMM15 are used. Instructions that use these registers are therefore one byte longer than instructions that use other registers, unless a REX prefix is needed anyway for other reasons:

```
; Example 10.5a. Instruction sizes (64 bit mode)
mov eax,[rbx]     is smaller than    mov eax,[r8].

; Example 10.5b. Instruction sizes (64 bit mode)
mov rax,[rbx]     is same size as    mov rax,[r8].
```

In example 10.5a, we can avoid a REX prefix by using register RBX instead of R8 as pointer. But in example 10.5b, we need a REX prefix anyway for the 64-bit operand size, and the instruction cannot have more than one REX prefix.

Floating point calculations can be done either with the old x87 style instructions with floating point stack registers ST(0)-ST(7) or the newer SSE style instructions with XMM registers. The x87 style instructions are more compact than the latter, for example:

```
; Example 10.6. Floating point instruction sizes
fadd  st0, st1      ; 2 bytes
addsd xmm0, xmm1    ; 4 bytes
```

The use of x87 style code may be advantageous even if it requires extra FXCH instructions. There are only small differences in execution speed between the two types of floating point instructions on current processors. However, it is likely that the x87 style instructions will be considered obsolete and will be less efficient on future processors.

Processors supporting the AVX instruction set can code XMM instructions in two different ways, with a VEX prefix or with the old prefixes. Sometimes the VEX version is shorter and sometimes the old version is shorter. However, there is a severe performance penalty to mixing XMM instructions without VEX prefix with instructions using YMM registers on most Intel processors.

The AVX-512 instruction set uses a new 4-bytes prefix called EVEX. While the EVEX prefix is one or two bytes longer then the VEX prefix, it allows a more efficient coding of memory operands with pointer and offset. Memory operands with a 4-bytes offset can sometimes be replaced by a 1-byte scaled offset when the EVEX prefix is used. Thereby the total instruction length becomes smaller.

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 10.7a, Static memory operands
mov ebx, [mem1]                 ; 6 bytes
add ebx, [mem2]                 ; 6 bytes
```

Reduce to:

```
; Example 10.7b, 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 10.8, 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`, `TEST`, `CALL` and `RET`. A `TEST` instruction with a 32-bit immediate operand can be replaced with various shorter alternatives, depending on the logic in case. Some examples:

```
; Example 10.9, Alternatives to test with 32-bit constant
test eax, 8                 ; 5 bytes
test ebx, 8                 ; 6 bytes
test al,  8                 ; 2 bytes
test bl,  8                 ; 3 bytes
and  ebx, 8                 ; 3 bytes
bt   ebx, 3                 ; 4 bytes (uses carry flag)
cmp  ebx, 8                 ; 3 bytes
```

It is not recommended to use the versions with 16-bit constants in 32-bit or 64-bit modes, such as `TEST AX,800H` because it will cause a penalty for decoding a length-changing prefix on some processors, as explained in manual 3: "The microarchitecture of Intel, AMD and VIA CPUs".

Shorter alternatives for `MOV register,constant` are often useful. Examples:

```
; Example 10.10, Loading constants into 32-bit registers
mov eax, 0                  ; 5 bytes
xor eax, eax                ; 2 bytes

mov eax, 1                  ; 5 bytes
xor 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. Instructions with 16-bit immediate data operands should be avoided in 32-bit and 64-bit mode because of the problem with decoding length-changing prefixes.

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 10.11a, 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 10.11b, 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 10.12, 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 10.13, 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 cannot replace `add rbx,4` by `add ebx,4`. Never use 32-bit registers as base or index inside the square brackets in 64-bit mode.

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

```
; Example 10.14. 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 pointers relative to the image base or to some reference point. 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 10.15a. Jump-table in 64-bit mode
section .data
JumpTable DQ Label1, Label2, Label3, ..

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

Implementation with image-relative pointers is available in Windows only:

```
; Example 10.15b. Image-relative jump-table in 64-bit Windows
; This works only with MASM assembler

section .data
JumpTable DD imagerel(Label1),imagerel(Label2),imagerel(Label3),..
extrn __ImageBase:byte

section .text
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
```

Another possibility is to use the jump table itself as a reference point. This method is commonly used by compilers for the Mac OS operating system, but it may be used in other 64-bit systems as well. Note that not all assemblers can make a relative reference from the data section to the code section. This example works with the NASM assembler.

```
; Example 10.15c. Self-relative jump-table in 64-bit Mac

section .data
JumpTable: DD Label1-JumpTable, Label2-JumpTable, Label3-JumpTable

section .text
default rel
mov eax, [n]                ; Index
lea rdx, [JumpTable]        ; Table and reference point
movsxd rax, [rdx + rax*4]   ; Load address relative to table
add rax, rdx                ; Add table base to address
jmp rax                     ; Jump to computed address
```

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

```
; Example 10.15d. 32-bit absolute jump table in 64-bit Linux
; Requires that addresses < 2^31
section .data
JumpTable DD Label1, Label2, Label3 ; 32-bit addresses

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

In example 10.15d, 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 10.15d can be used if there is certainty that the image base plus the program size is less than 2^{31}. This will work in application programs in Linux and BSD and in some cases in Windows, but not in Mac OS X (see page 23).

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

```
; Example 10.15d. 16-bit offsets to a reference point
section .data
; 16 bit addresses relative to Label1:
JumpTable: DW 0, Label2-Label1, Label3-Label1

section .text
default rel
mov eax, [n]                ; Index
lea rdx, [JumpTable]        ; Address of table (RIP-relative)
movsx rax, word [rdx+rax*2] ; Sign-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 10.15d uses `Label1` as a reference point. It works only if all labels are within the interval `Label1` $\pm 2^{15}$. The table contains the 16-bit offsets which are sign-extended and added to the reference point.

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 relative address, a 32-bit absolute address, or an 8-bit or 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 multiple pointers. This is typically the case in large switch statements and in linked lists.

10.6 Making instructions longer for the sake of alignment

There are situations where it can be advantageous to reverse the advice of the previous paragraphs in order to make instructions longer. Most important is the case where a loop entry needs to be aligned (see p. 85). 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. 25). The same instructions can be coded in the general form with a mod-reg-r/m byte. Examples:

```
; Example 10.16. 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 10.17. 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 a 32-bit immediate operand:

```
; Example 10.18. Making instructions longer
add  ebx,1         ; 3 bytes. Uses sign-extended 8-bit operand
DB   0x81, 0xC3
DD   1             ; 6 bytes. add ebx,1 with 32-bit operand
```

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 10.19. Making instructions longer
mov  eax,[ebx]     ; 2 bytes

; mov  eax,[ebx+00]; 3 bytes
DB   0x8B, 0x43, 0x00 ; Add 1-byte displacement

; mov  eax,[ebx+00000000]; 6 bytes
DB   0x8B, 0x83, 0, 0, 0, 0 ; Add 4-bytes displacement
```

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. 25). 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 10.20. Making instructions longer
mov  eax, [ebx]          ; Length = 2 bytes
DB   0x8B, 0x04, 0x23    ; Same with SIB byte. Length = 3 bytes
DB   0x8BH, 0x44H, 0x23, 0 ; 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 10.21. 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

Pentium 4 used 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 10.22. 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, and no instruction can have 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 x87 floating point 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 take extra time to decode on many processors. 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 10.23. Wrong way of making instructions longer
L1:  mov  ecx,1000
     DB   ox3E          ; 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.

Processors supporting the AVX instruction set use VEX prefixes, which are 2 or 3 bytes long. A 2-bytes VEX prefix can always be replaced by a 3-bytes VEX prefix. A VEX prefix can be preceded by segment prefixes but not by any other prefixes. No other prefix is allowed after the VEX prefix. Most instructions using XMM or YMM registers can have a VEX prefix. Do not mix YMM vector instructions with VEX prefix and XMM instructions without VEX prefix (see chapter 13.1). A few newer instructions on general purpose registers also use VEX prefix.

The AVX512 instruction set uses a 4-bytes prefix names EVEX. Instructions may be made longer by replacing a VEX prefix by an EVEX prefix when AVX512VL is supported.

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

10.7 Using multi-byte NOPs for alignment

The multi-byte `NOP` instruction has the opcode 0F 1F + a dummy memory operand. The length of the multi-byte `NOP` instruction can be adjusted by optionally adding 1 or 4 bytes of displacement and a SIB byte to the dummy memory operand and by adding one or more 66H prefixes. An excessive number of prefixes can cause delay on older microprocessors, but at least two prefixes is acceptable on most processors. NOPs of any length up to 10 bytes can be constructed in this way with no more than two prefixes. If the processor can handle multiple prefixes without penalty then the length can be up to 15 bytes.

The multi-byte `NOP` is more efficient than the commonly used pseudo-NOPs such as `MOV EAX,EAX` or `LEA RAX,[RAX+0]`. The multi-byte NOP instruction is supported on all Intel P6 family processors and later, as well as AMD Athlon, K7 and later, i.e. almost all processors that support conditional moves.

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. 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. The figures keep changing thanks to the fast technological development.

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

More advice 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, AMD and VIA CPUs" and in Intel's and AMD's software optimization manuals.

11.1 How caching works

A cache is a means of temporary storage that is closer to the microprocessor than the main memory. Data and code that is used often, or 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 Intel Skylake processor, for example, can contain 32 kb of data. It is organized as 512 lines of 64 bytes each. The cache is 8-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 eight possible lines. The line length in this example is 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 8 lines, there will be 512 / 8 = 64 different sets. The next six bits, i.e. bits 6 - 11, of a memory address will therefore select between these 64 sets. The remaining bits can have any value. The conclusion of this mathematical exercise is that if bits 6 - 11 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^{12} = 4096 bytes apart. No more than eight such addresses can be cached at the same time.

The cache sizes, cache line sizes, and set associativity on different microprocessors are described in manual 3: "The microarchitecture of Intel, AMD and VIA CPUs". The performance penalty for level-1 cache line contention can be quite considerable on older microprocessors, but on newer processors we are losing 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 cache lines are always aligned to physical addresses divisible by the cache line size (typically 64 bytes). 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. The level-2 cache is usually shared between code and data.

11.2 Trace cache

The old Intel processors with the NetBurst microarchitecture (Pentium 4) had a trace cache instead of a code cache. The trace cache stores the code after it has been translated to micro-operations (µops) while a normal code cache stores the raw code without translation. The trace cache removes the bottleneck of instruction decoding and attempts to store the code in the order in which it is executed rather than the order in which it occurs in memory. The main disadvantage of a trace cache is that the code takes more space in a trace cache than in a code cache. Later Intel processors do not have a trace cache, but possibly a µop cache.

11.3 µop cache

Intel microprocessors since Sandy Bridge, and AMD processors since Zen 1, have a traditional code cache before the decoders and a µop cache after the decoders. The µop cache is a big advantage because instruction decoding is often a bottleneck. The maximum capacity of the µop cache is 1536 µops in Intel Skylake and later processors, 2048 µops in AMD Zen 1, and 4096 µops in AMD Zen 2 - 3. The µop cache is such a critical resource that the programmer should economize its use and make sure the critical part of the code fits into the µop cache. Avoid unnecessary loop unrolling.

Later Intel processors also have a small loopback buffer which can store the last 30 - 60 µops. A tiny loop that fits into the loopback buffer will run particularly fast if there are no other bottlenecks.

Older AMD processors have instruction boundaries marked in the code cache. This relieves the critical bottleneck of instruction length decoding and can therefore be seen as an alternative to a µop cache. I do not know why this technique is rarely used in Intel processors.

11.4 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
32 (ymmword)	32
64 (zmmword)	64

Table 11.1. Preferred data alignment

The following example illustrates alignment of static data.

```
; Example 11.1, alignment of static data
section .data align=16
A:     DQ  0, 0          ; A is aligned by 16
B:     times  32 DB 0
C:     DD  0
D:     DW  0
ALIGN 16                 ; E must be aligned by 16
E:     DQ  0, 0

section .text
       default rel
       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.

Some microprocessors have a penalty of several clock cycles when accessing misaligned data that cross a cache line boundary.

Most XMM instructions without VEX prefix that read or write 16-byte memory operands require that the operand is aligned by 16. Instructions that accept unaligned 16-byte operands can be quite inefficient on older processors. However, this restriction is largely relieved with the AVX and later instruction sets. AVX instructions do not require alignment of memory operands, except for the explicitly aligned instructions. Processors that support the AVX instruction set generally handle misaligned memory operands very efficiently.

Aligning data stored on the stack can be done by rounding down the value of the stack pointer. The old value of the stack pointer must of course be restored before returning. A function with aligned local data typically look like this:

```
; Example 11.2a, Explicit alignment of stack (64-bit Windows)
FuncWithAlign:
        push    rbp                ; Prolog code
        mov     rbp, rsp           ; Save value of stack pointer
        sub     rsp, LocalSpace    ; Allocate space for local data
        and     rsp, -32           ; Align RSP by 32
        mov     eax, [rbp+8]       ; Function parameter = array
        vmovdqu ymm0,[rax]         ; Load from unaligned array
        vmovdqa [rsp],ymm0         ; Store in aligned space
        call    SomeOtherFunction  ; Call some other function
        ...
        mov     rsp, rbp           ; Epilog code. Restore rsp
        pop     rbp                ; Restore rbp
        ret
```

This function uses RBP to address function parameters, and RSP to address aligned local data. RSP is rounded down to the nearest value divisible by 32 simply by AND'ing it with -32. You can align the stack by any power of 2 by AND'ing the stack pointer with the negative value.

All 64-bit operating systems, and some 32-bit operating systems (Mac OS, optional in 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 11.2b, Propagate stack alignment (32-bit Linux)
FuncWithAlign:
        sub     rsp, 24             ; Allocate space for local data
        mov     rax, [rsp+32]       ; Function parameter = array
        movdqu  xmm0,[rax]          ; Load from unaligned array
        movdqa  [rsp],xmm0          ; Store in aligned space
        call    SomeOtherFunction   ; This call must be aligned
        ...
        retFuncWithAlign ENDP
```

In example 11.2b 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 8 is subtracted from the stack pointer. We have to subtract another 8 from the stack pointer before it is aligned by 16 again. The 8 is not enough for the local variable that needs 16 bytes so we have to subtract 24 to keep the stack pointer aligned by 16. 8 for the return address + 24 = 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 16 or 32 from RSP to keep it aligned by 16. Example 11.2b 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.

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

```
// Example 11.3a, 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 11.3b, 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.

See page 150 for how to move unaligned blocks of data efficiently.

11.5 Alignment of code

Most microprocessors fetch code in aligned 16-byte or 32-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. 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. Some assemblers are using pseudo-NOPs such as `MOV RAX,RAX` and `LEA RBX,[RBX+0]` as fillers. This has the disadvantage that it has a false dependence on the register, and it uses execution resources. It is better to use the multi-byte `NOP` instruction which can be adjusted to the desired length.

An alternative 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 78 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 11.4, Aligning loop entry. NASM assembler
section .text

align 16
; calculate where to place innerfunction in order to align innerloop:
times -(innerloop - innerfunction) % 16 nop

innerfunction:                ; This address will be adjusted
    mov  rax, [rsp+8]         ; array
    mov  ecx, 0x10000         ; size of array
    add  rax, rcx             ; end of array
    xor  edx, edx             ; sum
    neg  rcx                  ; negative index
innerloop:                    ; This loop entry will be aligned by 16
    add  edx, [rax+rcx]       ; calculate sum of array elements
    inc  rcx
    jnz  innerloop
    ret
```

This code will adjust the start address of the function to such a value that the loop inside the function is aligned by 16. This works with the NASM assembler, but other assemblers may not be able to do the necessary calculations with forward references.

The cost of misaligning `innerfunction` is negligible compared to the gain by aligning `innerloop` because the latter label is jumped to 0x4000 times as often.

11.6 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 11.5a. Loading double constant from data segment
section .data
C1  DQ  SomeConstant

section .text
movsd xmm0, C1
```

This can be changed to:

```
; Example 11.5b. Loading double constant from register (64-bit mode)
section .text
mov rax, SomeConstant
movq xmm0, rax      ; Some assemblers use 'movd' for this instruction
```

See page 124 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 C++ 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.

11.7 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 linked, 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) makes code caching less efficient. Dynamic link libraries are typically loaded at round memory addresses. This can cause cache contentions because the distances between multiple DLL's are divisible by high powers of 2.

11.8 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 non-temporal 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. As a rule of thumb, it can be recommended to use non-temporal writes only when writing a memory block that is bigger than half the size of the largest-level cache.

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

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 91ff 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:
```

It may be unwise to 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 on some older Intel processors and may cause false dependences on other processors.

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 `jecxz` and `loop` instructions 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
    section .text
    default rel
        mov  ecx, n         ; Load n
        test ecx, ecx       ; Test n
        jng  LoopEnd        ; Skip if n <= 0
        xor  eax, eax       ; i = 0
        lea  rsi, [Array]   ; Address of an array
    LoopTop:
        ; Loop body: Add 1 to all elements in Array:
        add  dword [rsi+4*rax], 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 `rsi` and the index in `rax`. 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
section .text
default rel
    mov  ecx, n             ; Load n
    lea  rsi, [Array]       ; Address of array
    lea  rsi, [rsi+4*rcx]   ; Point to end of array
    neg  rcx                ; i = -n
    jnl  LoopEnd            ; Skip if (-n) >= 0
LoopTop:
    ; Loop body: Add 1 to all elements in Array:
    add  dword [rsi+4*rcx], 1
    add  rcx, 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
section .text
default rel
    mov  ecx, n             ; Load n
    shl  ecx, 2             ; n * 4
    jng  LoopEnd            ; Skip if (4*n) <= 0
    lea  rsi, [Array]       ; Address of array
    add  rsi, rcx           ; Point to end of array
    neg  rcx                ; i = -4*n
LoopTop:
    ; Loop body: Add 1 to all elements in Array:
    add  dword [rsi+rcx], 1
    add  rcx, 4             ; i += 4
    js   LoopTop            ; Loop back if i < 0
LoopEnd:
```

There is no significant 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. 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 often 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`. There is a slight accumulation of rounding errors here, though, because the value `0.01` cannot be represented accurately. The assembly implementation looks like this:

```
; Example 12.5c. Assembly loop to make sine table
section .data
align 8
M0_01   dq   0.01           ; Define constant 0.01

section .bss
```

```
table resq  100              ; Define table

extern sin                   ; sin function

section .text
default rel
    push  rbx                ; rbx is saved across call to sin
    push  rdi                ; rdi is saved across call to sin
    sub   rsp, 8             ; make space for temporary variable x
    xor   ebx, ebx           ; i = 0
    xorpd xmm0, xmm0         ; x = 0.
    movsd [rsp], xmm0        ; xmm0 is not saved across call to sin
    lea   rdi, [table]       ; address of table
LoopTop:
    call  sin                ; xmm0 = sin(x)
    movsd [rdi+8*rbx], xmm0  ; table[i] = sin(x)
    movsd xmm0, [rsp]        ; x
    addsd xmm0, [M0_01]      ; x += 0.01
    movsd [rsp], xmm0        ; store x
    add   rbx, 1
    cmp   rbx, 100           ; i < n
    jb    LoopTop            ; Loop
    add   rsp, 8             ; free allocated space
    pop   rdi
    pop   rbx
    ret
```

There is no need to optimize the loop overhead in this case because the speed is limited by the floating point calculations. You may store x in `XMM6` rather than in memory in 64-bit Windows because `XMM6-XMM15` are saved across function calls in 64-bit Windows, but not in 32-bit Windows and not in Linux and Mac OS. The longer vector registers `YMM6` and `ZMM6` are not saved across function calls.

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 dependency 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 does not change inside the loop should be moved out of the loop.

The same applies to if-else branches with a condition that does not 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 dependency chains
- Instruction fetching
- Instruction decoding
- Instruction retirement
- Execution port throughput
- Execution unit throughput
- Suboptimal reordering and scheduling of µops
- Branch mispredictions
- Floating point exceptions and subnormal operands

If one particular bottleneck is limiting the performance then it does not 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, AMD and VIA CPUs" 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. Remember that a solution found by experimentation is CPU-specific and not certain to be optimal on CPUs 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 63.

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 that generate more than two µops, such as `LOOP`, `JECXZ`, `LODS`, `STOS`, etc.

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 dependency chain.

12.6 Distribute µops evenly between execution units

Manual 4: "Instruction tables" contains tables of how many µops each instruction generates and which execution port each µop goes to. This information is CPU-specific, of course. It is

necessary to calculate how many µops the loop generates in total and how many of these µops 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 µops divided by the retirement rate. The retirement rate is at least 4 µops per clock cycle for Intel Core2 and later 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 µop per clock cycle on most Intel processors. The load on a particular execution port is calculated as the number of µops 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. AMD processors do not have execution ports, but they have three or four pipelines with similar throughputs.

There may be more than one execution unit on each execution port on Intel processors. 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 through more than one execution port; and (3) on AMD processors that have no execution ports.

The load on a particular execution unit is calculated as the total number of µops 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 in vector loops

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   equ 100                 ; Define constant n (even and positive)
    mov ecx, n * 8          ; Load n * sizeof(double)
    xor eax, eax            ; i = 0
    lea rsi, [X]            ; X must be aligned by 16
    lea rdi, [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, [rsi+rax]  ; X[i], X[i+1]
    mulpd  xmm1, xmm2       ; X[i] * DA, X[i+1] * DA
    movapd xmm0, [rdi+rax]  ; Y[i], Y[i+1]
    subpd  xmm0, xmm1       ; Y[i]-X[i]*DA, Y[i+1]-X[i+1]*DA
    movapd [rdi+rax], 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 us 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, AMD and VIA CPUs".

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

Instruction	**µops fused**	**execution ports**					**execution units**	
		port 0/1	port 0/1/5/6	port 6	port 2/3/7	port 4	FADD	FMUL
`movapd xmm1,[rsi+rax]`	1				1			
`mulpd xmm1, xmm2`	1	1						1
`movapd xmm0,[rdi+rax]`	1				1			
`subpd xmm0, xmm1`	1	1					1	
`movapd [rdi+rax],xmm0`	1				1	1		
`add eax, 16`	1		1					
`cmp eax, ecx`	1		1					
`jl L1`	1			1				
Total	8	2	2	1	3	1	1	1

The total number of fused µops going to all the ports is 8. The overall throughput is four µops per clock cycle. This gives a minimum execution time of 2 clock cycles per iteration.

The arithmetic ports 0, 1, 5, and 6 receive 5 µops per iteration to distribute between 4 ports. This gives a limit of 1.25 clock cycle per iteration.

The address calculation and input ports 2,3, and 7 receive 3 µops per iteration, which gives a limit of 1 clock per iteration. The write port, 4, gets one µop.

The floating point unit FADD/FMUL receives 2 µops. A throughput of 2 µops per clock gives a limit of 2 clock per iteration.

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.

The conclusion is that the limiting bottleneck is the decoders or the µop cache which have a limit of 4 fused µops per clock cycle.

There is a dependency chain in the loop. The latencies are: 2 for memory read, 4 for multiplication, 4 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 dependency chain is `add eax,16` which has a latency of only 1.

The 2 clocks per iteration is a theoretical minimum. The measured execution time will often be longer because of suboptimal out-or-order scheduling, suboptimal distribution of µops between the execution ports, and cache contentions. The execution time will of course be much longer if there are or cache misses.

It is possible to reduce the number of instructions in the loop by using a negative index from the end of the arrays. The can improve the calculation speed because it reduces the number of µops:

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

    section .text
    default rel
        mov    eax, n * 8        ; Size = n * sizeof(double)
        lea    rsi, [X]          ; Address of array X (aligned)
        lea    rdi, [Y]          ; Address of array Y (aligned)
        add    rsi, rax          ; point to end of array X
        add    rdi, rax          ; point to end of array Y
        neg    rax               ; negative index
        movsd  xmm2, [DA]        ; Load DA
        xorpd  xmm2, [SignBit]   ; Change sign
        shufpd xmm2, xmm2, 0     ; Get -DA into both qwords of xmm2

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

This removes two instructions and two µops from the loop so that the theoretical minimum is 1.5 clock cycles per iteration.

12.8 Same example with FMA3

The Intel Haswell and later processors support the 3-operand form of fused multiply-and-add instructions called FMA3. AMD Piledriver supports both FMA3 and FMA4.

```
    ; Example 12.6d. Loop of DAXPY with FMA3, using xmm registers
section .text
default rel
    mov    eax, n * 8        ; Size = n * sizeof(double)
    lea    rsi, [X]          ; Address of array X (aligned)
    lea    rdi, [Y]          ; Address of array Y (aligned)
    add    rsi, rax          ; point to end of array X
    add    rdi, rax          ; point to end of array Y
    neg    rax               ; negative index
    vmovddup xmm2, [DA]      ; Load DA x 2
L1: vmovapd xmm1, [rdi+rax]  ; Y[i]
    vfnmadd231pd xmm1, xmm2, [rsi+rax] ; Y[i]-X[i]*DA
    vmovapd [rdi+rax], xmm1  ; Store
    add rax, 16              ; Add size of two elements to index
    jl L1                    ; Loop back
```

This loop takes 1 - 2 clock cycles per iteration on Intel processors. It may help to align the loop entry as explained in example 11.4.

12.9 Same example with AVX512

You can double the throughput by using YMM registers using the AVX instruction set, or you can quadruple the throughput by using ZMM registers with the AVX512 instruction set:

```
    ; Example 12.6e. Loop of DAXPY with AVX512, using zmm registers
```

```
section .text
default rel
n   equ    128               ; n must be divisible by 8
    mov    eax, n * 8        ; Size = n * sizeof(double)
    lea    rsi, [X]          ; Address of array X (aligned)
    lea    rdi, [Y]          ; Address of array Y (aligned)
    add    rsi, rax          ; point to end of array X
    add    rdi, rax          ; point to end of array Y
    neg    rax               ; negative index
    vbroadcastsd zmm2, [DA]  ; Load DA x 8

L1: vmovupd zmm1, [rdi+rax]  ; Y[i]
    vfnmadd231pd zmm1, zmm2, [rsi+rax] ; Y[i]-X[i]*DA
    vmovupd [rdi+rax], zmm1  ; Store
    add rax, 64              ; Add size of 8 elements to index
    jl L1                    ; Loop back
```

Example 12.6e does not assume that the arrays are aligned by 64. Therefore, `VMOVAPD` has been changed to `VMOVUPD`. The execution time for this loop has been measured to 1.5 clock cycles on Skylake. The bottleneck is not execution units here, but cache effects and perhaps the branch instruction. It is assumed that the array count n is divisible by 8 because we are doing 8 operations per clock cycle. See next section for discussion of what to do when n is not divisible by the vector size.

12.10 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 8 in order to use vectors of eight double-precision numbers. If we had used single-precision numbers then we would have rolled out the loop by 16 and used vectors of 16 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. However, the branch misprediction penalty is likely to be hidden by the fact that the loop counter is calculated way ahead of the loop body in an out-of-order processor.

- Improve caching. If the loop suffers from many data cache misses or cache contentions then it may be advantageous to schedule memory reads and writes in the way that is optimal for a specific processor. This is rarely needed on modern processors. 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 two.

- Break loop-carried dependency chain. A loop-carried dependency 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 on page 64.

- 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.

- Complete unrolling. A loop is completely unrolled when $r = n$, where n is a known constant. 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 103 for examples.

There are also disadvantages to loop unrolling. 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:

- Modern microprocessors can execute four or more independent instructions per clock cycle. The loop overhead typically consists of a counter, a compare, and a conditional jump. These instructions can execute simultaneously with the loop body so that the loop overhead adds no extra clock cycles to the execution time. The microprocessor may even execute multiple iterations of the loop simultaneously. Loop unrolling is unlikely to increase the performance significantly in this case.

- 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.

- Some processors have a loopback buffer to increase the speed of very small loops. The loopback buffer is limited to 20 - 50 instructions or 64 bytes of code, depending on the processor. Loop unrolling is likely to decrease performance if it exceeds the size of the loopback buffer. Processor-specific details are provided in manual 3: "The microarchitecture of Intel, AMD and VIA CPUs".

- Many modern processors have a µop cache of limited size (see chapter 11.3). This µop cache is so valuable that its use should be economized. Unrolled loops take up more space in the µop cache. Note that the µop cache misses are not detected when the loop is tested in isolation.

- A loop that has been unrolled to improve data caching on a specific microprocessor may not be optimal when running on another processor with a different cache structure.

- 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, as explained below.

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

There is a particular 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.

It can be quite tricky to get the extra calculations right when the repeat count is not divisible by the unroll factor. This gets particularly tricky if we are using a negative index as in example 12.6c, d, and e. The following example shows the DAXPY algorithm again, this time unrolled by 4, using AVX2. In this example n is a variable which may or may not be divisible by 4.

```
; Example 12.7. Unrolled Loop of DAXPY, single precision.

section .text
default rel

    mov     eax, [n]                    ; Array size
    shl     eax, 3                      ; Array size in bytes
    sub     rax, 32                     ; Skip any remainder
    lea     rsi, [X]                    ; Address of array X
    lea     rdi, [Y]                    ; Address of array Y
    add     rsi, rax                    ; point to end of array X - 4*8
    add     rdi, rax                    ; point to end of array Y - 4*8
    neg     rax                         ; negative index
    vbroadcastsd ymm2, [DA]             ; Load DA x 4
    jg      L2                          ; Skip main loop if n < 4

L1: ; main loop doing all elements except remainder
    vmovupd ymm1, [rdi+rax]             ; Y[i]
    vfnmadd231pd ymm1, ymm2, [rsi+rax] ; Y[i]-X[i]*DA
    vmovupd [rdi+rax], ymm1             ; Store
    add     rax, 32                     ; Four elements per iteration
    jl      L1                          ; Loop back

L2: ; Check for any remaining elements
    sub     rax, 32                     ; = -remainder
    jns     L4                          ; Skip if remainder = 0

L3: ; Extra loop for remaining (n % 4) iterations
    vmovsd xmm1, [rdi+rax+32]           ; Y[i]
    vfnmadd231sd xmm1,xmm2,[rsi+rax+32]; Y[i]-X[i]*DA
    vmovsd [rdi+rax+32], xmm1           ; Store
    add     rax, 8                      ; One element per iteration
    jl      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 subnormal 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.

To summarize: Loop unrolling has advantages and disadvantages. Do not unroll a loop unless the advantages are clearly outweighing the disadvantages. It is a common pitfall to measure the performance of an unrolled loop in isolation without considering the contention for code cache or µop cache with other parts of the program. Some compilers are unrolling loops excessively without considering the downsides. This may be a remnant from a time when loop overhead costs were higher.

12.11 Vector loops using mask registers (AVX512)

The AVX512 instruction set provides masked operations. This feature is useful for masking off the excess vector elements when the loop count is not divisible by the vector size:

```
    ; Example 12.8. Vectorized DAXPY loop using mask registers
    section .data
    align      64
    countdown  dd 7,6,5,4,3,2,1,0
    eight      dd 8

    section .text
    default rel

        lea    rsi, [X]                     ; point to beginning of X
        lea    rdi, [Y]                     ; point to beginning of Y
        mov    edx, [n]                     ; number of elements, n
        vmovd  xmm0, edx
        vpbroadcastd ymm0, xmm0             ; broadcast n
        vpaddd ymm0, ymm0, [countdown]      ; counts = n + countdown
        vpbroadcastd ymm1, [eight]          ; broadcast 8
        vbroadcastsd zmm2, [DA]             ; broadcast DA
        xor     eax, eax                    ; loop counter i = 0
    L1: ; Loop rolled out by 8, unused elements masked out
        vpcmpud k1, ymm0, ymm1, 5           ; mask k1 = (counts >= 8)
        vpsubd  ymm0, ymm0, ymm1            ; counts -= 8
        vmovupd zmm3 {k1}{z}, [rdi+rax*8]   ; load Y[i]
        vfnmadd231pd zmm3 {k1}, zmm2, [rsi+rax*8] ; Y[i]-DA*X[I]
        vmovupd [rdi+rax*8] {k1}, zmm3      ; masked Y[i]
        add     rax, 8                      ; 8 elements per iteration
        cmp     rax, rdx                    ; while i < n
        jb      L1
```

Here, the mask in the `k1` register has a 1-bit for all valid elements, and 0 for excess elements beyond n. The number of remaining elements are counted down in `YMM0`.

The load and store instructions are masked by `k1` to avoid loading and storing anything beyond the `n` elements of `Y`. The masked load instruction has the zeroing option `{z}` to avoid a false dependence on the previous value of the vector register `ZMM3`. It is good to mask the calculation instructions as well, using the same mask. Masking the calculations saves power and avoids exceptions and penalties for possible subnormal values, etc. There is no cost to adding a mask to a 512-bit vector instruction.

The countdown instructions that are used for generating the mask must have the same number of elements per vector as the calculations, but not necessarily the same number of bits per element. We are counting down n in `YMM0` with eight 32-bit unsigned integers, while the DAXPY calculations are using eight 64-bit double precision floats in `ZMM3`.

The loop in example 12.9 is dominated by vector instructions. It is likely that vector execution units will be a bottleneck here. It may be better to use integer instructions instead of vector instructions for making the mask if the execution units for vector addition is a bottleneck. This is shown in the next example:

```
    ; Example 12.10. Vectorized DAXPY, mask calculated with integer instr.
    ; Note: This version works only for n < 256!

section .text
default rel
```

```
    lea     rsi, [X]                            ; point to beginning of X
    lea     rdi, [Y]                            ; point to beginning of Y
    vbroadcastsd zmm2, [DA]                     ; broadcast DA
    mov     edx, [n]                            ; number of elements, n
    mov     ecx, -1                             ; fill with all 1's
    xor     eax, eax                            ; loop counter i = 0
L1: ; Loop rolled out by 8, unused elements masked out
    bzhi    ecx, ecx, edx                       ; zero bit positions > edx
    kmovw   k1, ecx                             ; copy bits to mask register
    vmovupd zmm3 {k1}{z}, [rdi+rax*8]           ; load Y[i]
    vfnmadd231pd zmm3 {k1}, zmm2, [rsi+rax*8] ; Y[i]-DA*X[I]
    vmovupd [rdi+rax*8] {k1}, zmm3              ; store Y[i]
    add     rax, 8                              ; 8 elements per iteration
    sub     edx, 8                              ; count down n
    ja      L1                                  ; repeat if n positive
```

Example 12.10 counts down the number of remaining elements in `edx` and uses `bzhi` to clear the remaining bits in `ecx` in the last iteration if `edx` < 16. The `bzhi` instruction belongs to the BMI2 instruction set, which is supported on all known processors with AVX512. Note that this works only for n < 256 because `bzhi` reads only the lower 8 bits of `edx`. We are using the instruction `kmovw` rather than `kmovb` because the latter instruction requires instruction set AVX512DQ.

If the iteration count is high and the instructions for calculating the mask are slowing down the loop execution then it is better to have a main loop without masking and only do the operations that require masking after the main loop:

```
; Example 12.11. Vectorized DAXPY, masking only after main loop
section .text
default rel

    lea     rsi, [X]                ; point to beginning of X
    lea     rdi, [Y]                ; point to beginning of Y
    vbroadcastsd zmm2, [DA]         ; broadcast DA
    mov     edx, [n]                ; number of elements
    and     edx, -8                 ; round down n to a multiple of 8
    lea     rsi, [rsi+rdx*8]        ; point to the end of X
    lea     rdi, [rdi+rdx*8]        ; point to the end of Y
    neg     rdx                     ; use negative index from the end
L1: ; Main loop rolled out by 8
    vmovupd zmm3, [rdi+rdx*8]       ; load Y[i]
    vfnmadd231pd zmm3, zmm2, [rsi+rdx*8] ; Y[i]-DA*X[I]
    vmovupd [rdi+rdx*8], zmm3       ; store Y[i]
    add     rdx, 8                  ; next 8 elements
    jnz     L1                      ; count up to zero
    mov     edx, [n]                ; calculate remaining elements
    and     edx, 7                  ; n modulo 8
    ; Omit the next line if branch L2 is poorly predictable
    jz      L2                      ; optionally skip if zero
    mov     eax, -1                 ; all 1's
    bzhi    eax, eax, edx           ; zero bit positions > edx
    kmovw   k1, eax                 ; copy bits to mask register
    ; rsi and rdi are pointing to where the main loop ended.
    ; Now do the last 0-7 elements, using mask k1
    vmovupd zmm3 {k1}{z}, [rdi] ; load Y[i]
    vfnmadd231pd zmm3 {k1}, zmm2, [rsi] ; Y[i]-DA*X[I]
    vmovupd [rdi] {k1}, zmm3        ; store Y[i]
L2:
```

Example 12.11 contains the loop body twice. The first body is repeated n/8 times in the L1 loop. The second body calculates the remaining n modulo 8 elements when n is not divisible by 8. It is not necessary to skip the last body with `jz L2` if this branch is poorly predictable.

Example 12.11 has only 5 instructions in the loop body. This will be the fastest solution if the loop count is high. Example 12.8 and 12.10 are useful if the loop body is so large that we do not want to include it twice, or if the loop count is small, or if the bottleneck is memory access or a loop-carried dependency chain. Example 12.10 can only be used if n < 256.

12.12 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 81.

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.

Modern 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 microprocessor is able to run two threads in each CPU core. 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.

12.13 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 dependency chains (see page 64) and distributing µops evenly between the different execution units or execution ports (see page 92).
- Use vector instructions. See chapter 13 page 105.
- Use multiple threads. See chapter 14 page 131.

Loop-carried dependency chains can be broken by using multiple accumulators, as explained on page 64. The maximum useful number of accumulators is the latency of the most critical instruction in the dependency chain divided by the reciprocal throughput for that instruction. For example, if the latency of floating point addition is 4 clock cycles and the

reciprocal throughput is 1, then the maximum useful number of accumulators is 4. A lower number of accumulators than the maximum may be sufficient to make sure the loop-carried dependency chain is not a limiting factor.

```
// Example 12.12a, Loop-carried dependency chain.
// C code without optimization
// (Same as example 9.3a page 64)
double list[100], sum = 0.0;
for (int i = 0; i < 100; i++) sum += list[i];
```

Example 12.12b shows a loop that adds a hundred numbers, using three AVX2 vector registers as accumulators.

```
; Example 12.12b, Three vector accumulators
section .text
default rel

    ; Add 100 numbers using three accumulators with 4 numbers each
    lea     rsi,  [list]          ; Pointer to list
    vmovapd ymm0, [rsi]           ; load first 4 elements
    vmovapd ymm1, [rsi+32]        ; load next 4 elements
    vmovapd ymm2, [rsi+64]        ; load next 4 elements
    add     rsi, 96               ; increment pointer
    mov     ecx, 7                ; repeat loop 7 times
L1:
    vaddpd  ymm0, ymm0, [rsi]     ; add 4 elements
    vaddpd  ymm1, ymm1, [rsi+32]  ; add 4 elements
    vaddpd  ymm2, ymm2, [rsi+64]  ; add 4 elements
    add     rsi, 96
    dec     ecx
    jnz     L1                    ; loop
    vaddpd  ymm0, ymm0, [rsi]     ; add last 4 elements
    vaddpd  ymm1, ymm1, ymm2      ; join two accumulators
    vaddpd  ymm0, ymm0, ymm1      ; join last two accumulators

    ; calculate sum of the four elements in ymm0:
    vextractf128 xmm1, ymm0, 1    ; get upper half of ymm0 into xmm1
    vaddpd  xmm0, xmm1, xmm1      ; join four values into two
    ; avoid the slow vhaddpd instruction
    vunpckhpd xmm1, xmm0, xmm0    ; get upper value
    vaddsd  xmm0, xmm1            ; join last two values
    ; The sum is now in xmm0
```

In example 12.12b, I have loaded the three vector registers with the first 3*4 values from `list`. The loop is doing 12 additions per iteration. We need to do four more additions after the loop because 100 is not divisible by 12. The three accumulators containing four partial sums each are added together to make one vector of four values after the loop. Finally, these four values are added together to get the final sum.

The loop in example 12.12b has four loop-carried dependency chains running in parallel: `ymm0`, `ymm1`, `ymm2`, and `ecx`. The speed is limited by the latency of the `vaddpd` instruction, which is longer than the latency of the `dec ecx` instruction. It may be possible to improve the speed further by using more accumulators, but other effects are likely to slow down execution as we reach the limiting throughput of the floating point adder. The optimal number of accumulators is often less than the theoretical maximum. With only 7 iterations of the loop in this example, it is probably not worth the extra code to add more accumulators. There is no need to use negative indexes in this loop because the loop overhead is not a limiting bottleneck.

The situation becomes more tricky if you want to use long double precision in order to reduce the accumulation of rounding errors. Long double precision is available only with the

old x87 style floating point registers. These registers are organized as a rolling stack. We have to use `fxch` instructions to get the desired register to the top of the stack.

```
; Example 12.12c, Four accumulators with long double precision
section .text
default rel

    ; Add 100 numbers using four long double precision accumulators
    lea   rsi,  [list]       ; Pointer to list
    fld   qword [rsi]        ; accum1 = list[0]
    fld   qword [rsi+8]      ; accum2 = list[1]
    fld   qword [rsi+16]     ; accum3 = list[2]
    fld   qword [rsi+24]     ; accum4 = list[3]
    fxch  st3                ; Get accum1 to top
    mov   eax, 32            ; Index to list[4]
    mov   ecx, 32            ; Repeat count
L1:
    fadd  qword [rsi+rax]    ; Add list[i]
    fxch  st1                ; Swap accumulators
    fadd  qword [rsi+rax+8]  ; Add list[i+1]
    fxch  st2                ; Swap accumulators
    fadd  qword [rsi+rax+16] ; Add list[i+2]
    fxch  st3                ; Swap accumulators
    add   eax, 24            ; i += 3
    dec   ecx
    jnz   L1                 ; Loop

    faddp st1, st0           ; Add two accumulators together
    fxch  st1                ; Swap accumulators
    faddp st2, st0           ; Add the two other accumulators
    faddp st1, st0           ; Add these sums
    ; The final sum is now in st0
```

In example 12.12c, we have loaded the four accumulators with the first four values from `list`. The funny thing about using x87 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.

12.14 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 NASM syntax includes a macro language, which can be quite useful. Other assemblers have similar capabilities, but the syntax is different.

If, for example, we need a list of square numbers, then the C++ code may look like this:

```
// Example 12.13a. 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 NASM language:

```
; Example 12.13b. Macro loop to produce data
```

```
section .data
squares:            ; label at start of array
%assign i    0      ; temporary loop counter
%rep    10          ; repeat 10 times
        DD   i*i    ; define one array element
%assign i    i+1    ; increment loop counter
%endrep             ; end of rep 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 `%assign i i+1` never make it into the final code, and hence take no time to execute. In fact, example 12.13b generates no executable code, only data. The macro preprocessor will translate the above code to:

```
; Example 12.13c. Resuls of macro loop expansion
squares:         ; 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 12.14a. 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 12.14b.
; INTPOWER macro calculates the integer power N of a double
; precision  floating point value X, using SSE2 instruction set.
; INTPOWER has three parameters:
; X: xmm register used as input. This register is not preserved
; Y: xmm register used for the result. Must be different from X
; N: A positive integer constant
; The result will be Y = pow(X,N)

%macro  INTPOWER 3
; local temporary variables:
; IPI: Used for shifting N
; YUSED: Remember if Y contains data
   %assign IPI %3             ; IPI = N
   %assign YUSED 0            ; remember if Y contains valid data
   %rep 32                    ; maximum repeat count is 32
      %if IPI & 1             ; test bit 0 of IPI
         %if YUSED            ; if Y already contains data
            mulsd %2, %1      ; multiply Y with a power of X
         %else                ; if this is first time Y is used:
```

```
            movsd %2, %1      ; copy data to Y
            %assign YUSED 1   ; remember that Y now contains data
         %endif
      %endif
      %assign IPI (IPI >> 1)  ; shift right IPI one place
      %if IPI == 0            ; stop when IPI = 0
         %exitrep             ; exit REP 32 loop prematurely
      %else
         mulsd %1, %1         ; square X
      %endif
   %endrep                    ; end of REP 32 loop
%endmacro                     ; end of INTPOWER macro definition
; if AVX instruction set:
; change mulsd %2, %1 to vmulsd %2, %2, %1
; change mulsd %1, %1 to vmulsd %1, %1, %1
; change movsd %2, %1 to vmovsd %2, %1
```

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 12.14c. Macro invocation
INTPOWER xmm0, xmm1, 12
```

This will be expanded to:

```
; Example 12.14d. 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

There are physical and technological limits to the maximum clock frequency of microprocessors. Therefore, 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 a thousand different SIMD instructions available. These instructions are listed in "Intel 64 and IA-32 Architectures Software Developer's Manual" and in "AMD64 Architecture Programmer's Manual".

Multiple data can be packed into 64-bit MMX registers, 128-bit XMM registers, 256-bit YMM registers, or 512-bit ZMM registers in the following ways:

data type	data per pack	register size	instruction set
8 bit integer	8	64 bit (MMX)	MMX
16 bit integer	4	64 bit (MMX)	MMX
32 bit integer	2	64 bit (MMX)	MMX
64 bit integer	1	64 bit (MMX)	SSE2
32 bit float	2	64 bit (MMX)	3DNow (obsolete)
8 bit integer	16	128 bit (XMM)	SSE2
16 bit integer	8	128 bit (XMM)	SSE2
32 bit integer	4	128 bit (XMM)	SSE2
64 bit integer	2	128 bit (XMM)	SSE2
32 bit float	4	128 bit (XMM)	SSE
64 bit float	2	128 bit (XMM)	SSE2
8 bit integer	32	256 bit (YMM)	AVX2
16 bit integer	16	256 bit (YMM)	AVX2
32 bit integer	8	256 bit (YMM)	AVX2
64 bit integer	4	256 bit (YMM)	AVX2
32 bit float	8	256 bit (YMM)	AVX
64 bit float	4	256 bit (YMM)	AVX
8 bit integer	64	512 bit (ZMM)	AVX-512BW
16 bit integer	32	512 bit (ZMM)	AVX-512BW
32 bit integer	16	512 bit (ZMM)	AVX-512
64 bit integer	8	512 bit (ZMM)	AVX-512
32 bit float	16	512 bit (ZMM)	AVX-512
64 bit float	8	512 bit (ZMM)	AVX-512

Table 13.1. Vector types

Whether the different instruction sets are supported on a particular microprocessor can be determined with the `CPUID` instruction, as explained on page 132. The 64-bit MMX registers cannot be used together with the x87 style floating point registers. The vector registers can only be used if supported by the operating system. See page 133 for how to check if the use of vector 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 vector instructions, you have to consider whether the resulting code will be faster than the simple instructions without vectors. Vector code is sometimes unsing more instructions on trivial things such as converting and moving data into the right positions in the registers, and emulating branches with conditional moves, than on the actual calculations. Example 13.9 below is an example of this. Vector instructions are relatively slow on older processors, but many newer processors can do a vector calculation just as fast as a scalar (single) calculation in many cases.

For floating point calculations, it is preferred to use vector registers rather than the old x87 registers, even if there are no opportunities for handling data in parallel. The vector registers are handled in a more straightforward way than the old x87 register stack, and some CPUs have poor performance for x87 instructions.

Memory operands for XMM instructions without VEX prefix have to be aligned by 16. Memory operands for YMM instructions are preferably aligned by 32 and ZMM by 64, but this is not necessary. See page 85 for how to align data in memory.

Most common arithmetic and logical operations can be performed in vector registers. The following example illustrates the addition of two arrays:

```
; Example 13.1a. Adding two arrays using vectors
; float a[128], b[128], c[128];
; for (int i = 0; i < 128; i++) a[i] = b[i] + c[i];
; a, b and c are preferably aligned by 64
      xor     ecx, ecx          ; Loop counter i = 0
L:    vmovups zmm0, [b+rcx]     ; Load 16 elements from b
      vaddps  zmm0,zmm0,[c+rcx]; Add  16 elements from c
      vmovups [a+rcx], zmm0     ; Store 16 results in a
      add     ecx, 64           ; 16 elements * 4 bytes = 64
      cmp     ecx, 512          ; 128 elements * 4 bytes = 512
      jb      L                 ; Loop
```

There are no instructions for integer division. Integer division by a constant divisor can be implemented as multiplication and shift, using the method described on page 137 or the functions in the vector class library at github.com/vectorclass or the assembly library at www.agner.org/optimize/asmlib.zip.

13.1 Using AVX instruction set and YMM or ZMM registers

The 128-bit XMM registers are extended to 256-bit YMM registers and 512-bit ZMM registers when the various AVX instruction set extensions are available. Further extensions to 1024 bits and perhaps 2048 bits are possible in the future, but no such plans have been published yet (2020).

There are 16 YMM registers or 32 ZMM registers in 64-bit mode. There are only 8 XMM/YMM/ZMM registers in 32-bit mode.

Most AVX instructions allow three operands: one destination and two source operands. This has the advantage that no input register is overwritten by the result.

All the XMM instructions have two versions in the AVX instruction set. A legacy version which leaves the upper half (bit 128-511) of the target ZMM register unchanged, and a VEX-prefix version which sets the upper bits to zero in order to make the result independent of the previous value of the upper bits of the target register.

The VEX-prefix version has a V prefix to the name and in most cases three operands:

```
; Example 13.2. Legacy and VEX versions of the same instruction
addps  xmm1,xmm2       ; xmm1 = xmm1 + xmm2
vaddps xmm1,xmm2,xmm3  ; xmm1 = xmm2 + xmm3
```

13.2 Mixing VEX and SSE code

There is a problem if a program contains a mixture of instructions with different vector sizes, such as SSE instructions with XMM registers and VEX instructions with YMM or ZMM registers.

Mixing YMM or ZMM instructions with legacy 128-bit instructions without VEX prefix will cause some Intel processors to switch the register file between different states which will cost many clock cycles.

The following applies to Intel Sandy Bridge, Ivy Bridge, Haswell, and Broadwell processors. These processors have YMM registers but not ZMM registers.

For the sake of compatibility with legacy SSE code, the register set has three different states on Sandy Bridge, Ivy Bridge, Haswell, and Broadwell processors:

A. (Clean state). The upper bits of all YMM registers are unused and known to be zero.

B. (Modified state). Bits 128-255 of at least one YMM register are used and contain data.

C. (Saved state). All YMM registers are split in two. The lower half is used by legacy SSE instructions which leave the upper part unchanged. All the upper-part halves are stored in a scratchpad. The two parts of each register will be joined together again if needed by a transition to state B.

Two instructions are available for the sake of fast transition between these states. VZEROALL which sets all YMM registers to zero, and VZEROUPPER which sets the upper part of YMM0-YMM15 or ZMM0-ZMM15 registers to zero. Both instructions leave the processor in state A. The state transitions can be illustrated by the following state transition table:

Current state → **Instruction ↓**	**A**	**B**	**C**
VZEROALL / UPPER	A	A	A
XMM	A	C	C
VEX XMM	A	B	B
VEX YMM	B	B	B

Table 13.2. YMM state transitions

State A is the neutral initial state. State B is needed when the full YMM registers are used. State C is needed for making legacy XMM code fast and free of false dependences when called from state B. A transition from state B to C is costly because all YMM registers must be split in two halves which are stored separately. A transition from state C to B is equally costly because all the registers have to be merged together again. A transition from state C to A is also costly because the scratchpad containing the upper part of the registers does not support out-of-order access and must be protected from speculative execution in possibly mispredicted branches. The transitions B → C, C → B and C → A are very time consuming on these Intel processors because they have to wait for all registers to retire. Transitions A → B and B → A are fast, taking at most one clock cycle. C should be regarded as an undesired state, and the transition A → C is not possible. The undesired transitions take approximately 70 clock cycles on Intel Sandy Bridge, Ivy Bridge, Haswell, and Broadwell processors according to my measurements.

The following examples illustrate the state transitions:

```
; Example 13.3a. Transition between YMM states
vaddps      ymm0, ymm1, ymm2    ; State B
addss       xmm3, xmm4          ; State C
vmulps      ymm0, ymm0, ymm5    ; State B
```

Example 13.3a has two expensive state transitions, from B to C, and back to state B. The state transition can be avoided by replacing the legacy ADDSS instruction by the VEX-coded version VADDSS:

```
; Example 13.3b. Transition between YMM states avoided
vaddps      ymm0, ymm1, ymm2    ; State B
vaddss      xmm3, xmm3, xmm4    ; State B
vmulps      ymm0, ymm0, ymm5    ; State B
```

This method cannot be used when calling a library function that uses XMM instructions. The solution here is to save any used YMM registers and go to state A:

```
; Example 13.3c. Transition to state A
vaddps      ymm0, ymm1, ymm2    ; State B
vmovaps     [mem], ymm0         ; Save ymm0
```

```
vzeroupper                          ; State A
call        XMM_Function            ; Legacy function
vmovaps     ymm0, [mem]             ; Restore ymm0
vmulps      ymm0, ymm0, ymm5        ; State B
vzeroupper                          ; Go to state A before returning
ret
...
XMM_Function proc near
addss       xmm3, xmm4              ; State A
ret
```

Intel Skylake and later processors behave differently. They have state A and B, but not state C. This avoids the expensive transitions to and from state C, but gives another problem instead.

The Skylake processor is unable to split a vector register in two. If a non-VEX instruction writes to an XMM register while in state B, then the instruction will have to wait for the previous value of this register because it may have to combine the lower part from the non-VEX instruction with the upper part from a preceding VEX instruction. For example:

```
; Example 13.4. Mixing VEX and SSE instructions on Skylake
vaddps    zmm0,  zmm1, zmm2     ; State B
mulps     xmm3,  xmm4
movaps    [mem], xmm3           ; Save xmm3
movps     xmm3,  xmm5           ; Must wait for mulps
```

The state does not distinguish between individual registers. The processor treats all registers as having a dirty upper part when in state B. Therefore, the last instruction must wait for the result of the preceding multiplication in order to join the two parts of register zmm3.

The AMD Ryzen processor splits all 256-bit instructions into two 128-bit µops so that these problems never occur.

Guidelines for mixing VEX and non-VEX code.

Mixtures of VEX and non-VEX code occur frequently when functions libraries are used. For example, a program compiled for the AVX512 instruction set may call a library function compiled for SSE2.

The following guidelines should be followed in order to get optimal performance in programs that mix SSE and VEX code.

- A function that uses YMM or ZMM instructions should issue a `VZEROALL` or `VZEROUPPER` before returning if there is a possibility that it might return to SSE code.

- A function that uses YMM or ZMM instructions should save any used YMM or ZMM register and issue a `VZEROALL` or `VZEROUPPER` before calling any function that might contain SSE code.

- A function that has a CPU dispatcher for choosing YMM or ZMM code if available and XMM code otherwise, should issue a `VZEROALL` or `VZEROUPPER` before leaving the YMM or ZMM part.

This recommendation allows any function to use the largest vector registers available, but a function that uses YMM or ZMM registers must leave the registers in state A before calling any function with unknown VEX status and before returning to any function with unknown VEX status.

Obviously, this does not apply to functions that use YMM or ZMM registers for parameter transfer or return. A function that uses YMM or ZMM registers for parameter transfer can assume state B on entry. A function that uses a YMM or ZMM register for the return value can only be in state B on return. State C should always be avoided.

The VZEROUPPER instruction is faster than VZEROALL on most processors. Therefore, it is recommended to use VZEROUPPER rather than VZEROALL unless you want a complete initialization. VZEROALL cannot be used in 64-bit Windows because the ABI specifies that registers XMM6 - XMM15 have callee-save status. In other words, the calling function can assume that register XMM6 - XMM15, but not the upper parts of the YMM or ZMM registers, are unchanged after return. No vector registers have callee-save status in 32-bit Windows or in any Unix system (Linux, BSD, Mac). Therefore it is OK to use VZEROALL in e.g. Linux. Obviously, VZEROALL cannot be used if any XMM register contains a function parameter or return value. VZEROUPPER must be used in these cases.

Unfortunately, the VZEROUPPER and VZEROALL instructions are expensive on the Knights Landing processor (the first processor to support AVX512). It is not recommended to use these instructions on the Knights Landing. The question on when to use VZEROUPPER and when not to use it is discussed at https://software.intel.com/en-us/forums/intel-isa-extensions/topic/704023.

An alternative solution is to use only ZMM16-ZMM31. These registers are not accessible to SSE code and they do not affect the state. This may be the optimal solution on processors that support AVX512, but it cannot be used when YMM0 or ZMM0 etc. are needed for function parameters or function return value. Note that VZEROUPPER and VZEROALL affect only ZMM0-ZMM15, not ZMM16-ZMM31.

Mixing VEX and non-VEX instructions is a common error that is very easy to make and very difficult to detect. The code will still work, but with reduced performance on most Intel processors. It is strongly recommended that you double-check your code for any mixing of VEX and non-VEX instructions.

Warm up time

Many processors are able to turn off the upper bits of the 256 bit or 512 bit execution units in order to save power when these units are not used.

It takes typically 10-20 µs to power up this upper part after an idle period. The throughput of 256-bit vector instructions is much lower during this warm-up period because the processor uses the lower 128-bit units twice to execute a 256-bit operation. It is possible to make the 256-bit units or 512-bit units warm up in advance by executing a dummy 256 or 512-bit instruction at a suitable time before the larger unit is needed. The upper part of the 256 or 512-bit units will be turned off again after approximately1 ms of no large vector instructions. This phenomenon is described in manual 3: "The microarchitecture of Intel, AMD and VIA CPUs".

Operating system support

Code that uses YMM or ZMM registers or VEX coded instructions can only run in an operating system that supports this register size because the operating system must save the YMM or ZMM registers on task switches.

The following operating system versions support AVX and YMM registers: Windows 7, Windows server 2008 SP2, Linux kernel version 2.6.30 and later.

The following operating system versions support AVX512 and ZMM registers: Windows 10 and Linux kernel version 3.15.

YMM/ZMM and system code

A situation where transitions between state B and C must take place is when YMM code is interrupted and the interrupt handler contains legacy XMM code that saves the XMM registers but not the full YMM registers. In fact, state C was invented exactly for the sake of preserving the upper part of the YMM registers in this situation.

It is very important to follow certain rules when writing device drivers and other system code that might be called from interrupt handlers. If any vector register is modified by a VEX instruction in system code then it is necessary to save the entire register state with `XSAVE` first and restore it with `XRESTOR` before returning. It is not sufficient to save the individual YMM registers because future processors, which may extend the YMM registers further to 512-bit ZMM registers or still larger, will zero-extend the YMM registers to ZMM when executing YMM instructions and thereby destroy the highest part of the ZMM registers. `XSAVE` / `XRESTOR` is the only way of saving these registers that is compatible with future extensions beyond 256 or 512 bits. Future extensions will not use the complicated method of having two versions of every vector instruction.

If a device driver does not use `XSAVE` / `XRESTOR` then there is a risk that it might inadvertently use VEX code even if the programmer did not intend this. A compiler that is not intended for system code may insert implicit calls to library functions such as `memset` and `memcpy`. These functions typically have their own CPU dispatcher which may select the largest register size available. It is therefore necessary to use a compiler and a function library that are intended for making system code.

These rules are described in more detail in manual 5: "Calling conventions".

Using non-destructive three-operand instructions

The AVX instruction set which defines the YMM instructions also defines an alternative encoding of all existing XMM instructions by replacing existing prefixes and escape codes with the new VEX prefix. The VEX prefix has the further advantage that it defines an extra register operand. Almost all XMM instructions that previously had two operands now have three operands when the VEX prefix is used.

The two-operand version of an instruction typically uses the same register for the destination and for one of the source operands:

```
; Example 13.5a. Two-operand instruction
movsd  xmm0, xmm1          ; copy xmm1 to avoid overwriting the value
addsd  xmm0, xmm2          ; xmm0 = xmm0 + xmm2
```

This has the disadvantage that the result overwrites the value of one of the source operands. The move instruction in example 13.5a can be avoided when the three-operand version of the addition is used:

```
; Example 13.5b. Three-operand instruction
vaddsd xmm0, xmm1, xmm2    ; xmm0 = xmm1 + xmm2
```

Here none of the source operands are destroyed because the result can be stored in a different destination register. This can be useful for avoiding register-to-register moves. The `addsd` and `vaddsd` instructions in example 13.5a and 13.5b have exactly the same length. Therefore there is no penalty for using the three-operand version. The instructions with names ending in `ps` (packed single precision) are one byte shorter in the two-operand version than the three-operand VEX version if the destination register is not `xmm8` - `xmm15`. The three-operand version is shorter than the two-operand version in a few cases. In most cases the two- and three-operand versions have the same length.

It is possible to mix two-operand and three-operand instructions in the same code as long as the register set is in state A. But if the register set happens to be in state C for whatever reason then the mixing of XMM instructions with and without VEX will cause a costly state change every time the instruction type changes. It is therefore better to use VEX versions only or non-VEX only. If YMM or ZMM registers are used (state B) then you should use only the VEX-prefix version for all XMM instructions until the VEX-section of code is ended with a `VZEROALL` or `VZEROUPPER`.

The 64-bit MMX instructions and most general purpose register instructions do not have three operand versions. There is no penalty for mixing MMX and VEX instructions. Only a few general purpose register instructions also have three operands.

Unaligned memory access

All VEX coded vector instructions with a memory operand allow unaligned memory access, except for the explicitly aligned instructions `VMOVAPS`, `VMOVAPD`, `VMOVDQA`, `VMOVNTPS`, `VMOVNTPD`, `VMVNTDQ`. Therefore, it is possible to store YMM and ZMM operands on the stack without keeping the stack aligned by 32.

Compiler support

The AVX instruction sets are supported by the Microsoft, Intel, Gnu, and Clang compilers.

The compilers will use the VEX prefix version for all XMM instructions, including intrinsic functions, if compiling for an AVX instruction set. It is the responsibility of the programmer to issue a `VZEROUPPER` instruction before any transition from a module compiled with AVX to a module or library compiled without AVX.

Fused multiply-and-add instructions

A fused multiply-and-add (FMA) instruction can make a floating point multiplication followed by a floating point addition or subtraction in the same time that it otherwise takes to make only a multiplication. The FMA operation has the form:

d = a * b + c

There are two different variants of FMA instructions, called FMA3 and FMA4. FMA4 instructions can use four different registers for the operands a, b, c and d. FMA3 instructions have only three operands, where the destination d must use the same register as one of the input operands a, b or c. Intel originally designed the FMA4 instruction set, but currently supports only FMA3. The latest AMD processors support both FMA3 and FMA4 (See en.wikipedia.org/wiki/FMA_instruction_set#History).

Examples

Example 12.8 page 99 illustrates the use of the AVX512 instruction set for a DAXPY calculation, using FMA3 instructions.

13.3 Using AVX512 instruction set and ZMM registers

The vector registers are extended to 512 bits with the AVX512 instruction set. The number of vector registers is increased to 32 registers in 64-bit mode, while there are only 8 vector registers in 32-bit mode.

There are 8 new mask registers named k0 - k7. The mask registers k1 - k7 can be used for conditional operations on each vector element as explained on page 116. Conditional operations are also useful for array loops when the array size is not divisible by the size of the vector registers. This is explained on page 99.

There are several additional extensions to AVX512. All processors with AVX512 have some of these extensions, but no processor so far has them all (writing in 2020). The known and planned extensions to AVX512 are the following:

- AVX512F. Foundation. All AVX512 processors have this. Includes operations on 32-bit and 64-bit integers, float and double in 512-bit vectors, including masked operations.
- AVX512VL. Includes the same operations on 128-bit and 256-bit vectors, including masked operations and 32 vector registers.
- AVX512BW. Operations on 8-bit and 16-bit integers in 512-bit vectors.
- AVX512DQ. Multiplication and conversion instructions with 64-bit integers. Various other instructions on float and double.
- AVX512ER. Fast reciprocal, reciprocal square root, and exponential function. Precise on float; approximate on double.
- AVX512CD. Conflict detection. Find duplicate elements in a vector.
- AVX512PF. Prefetch instructions with gather/scatter logic.
- AVX512VBMI. Permutation and shift with 8-bit granularity.
- AVX512VBMI2. Compress and expand with 8-bit and 16-bit granularity.
- AVX512IFMA. Fused multiply-and-add on 52-bit integers.
- AVX512_4VNNIW. Iterated dot product on 16-bit integers.
- AVX512_4FMAPS. Iterated fused multiply-and-add, single precision.

13.4 Conditional moves in xmm and ymm registers

Consider this C++ code which finds the biggest values in four pairs of values:

```
// Example 13.6a. 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 13.6b. 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. There are vector instructions for finding the absolute value of integers. The absolute value of floating point vector elements is calculated by AND'ing out the sign bit, as shown in example 13.17 page 125. 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. The most 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 13.7a. 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 method for doing conditional moves depends on the instruction set. The first SSE instruction set is implementing conditional moves 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 13.7b. Conditional move with SSE2 instruction set
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 13.7b) and the vectors that are selected (`a` and `b` in example 13.7b) need not be the same type. For example, `x` and `y` could be integers. But they should have the same number of bits per element. 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 13.9b 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 13.7b. If SSE4.1 is supported then we can use the `BLENDVPS` instruction instead:

```
; Example 13.7c. Conditional move with SSE4.1 instruction set
movaps   xmm0, [y]       ; Load y vector
cmpltps  xmm0, [x]       ; Compare with x. xmm0 = mask for y < x
movaps   xmm1, [a]       ; Load a vector
blendvps xmm1, [b], xmm0 ; Blend a and b
movaps   [c], xmm0       ; c = x > y ? a : b
```

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 13.8a. 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 registers:

```
; Example 13.8b. Conditional swapping in XMM registers, SSE
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 shifting the desired bit into the most significant position for use with blend instructions. If the AND / ANDN method is used then copy the most significant bit into all bit positions using the arithmetic shift right instruction `psrad`.

The shift method is illustrated in the next example where vector elements are raised to different integer powers. We are using the method in example 12.14a page 104 for calculating powers.

```
// Example 13.9a. Raise vector elements to different integer powers
double x[2], y[2];  unsigned int n[2];
for (int j = 0; j < 2; j++) {
   y[j] = pow(x[j],n[j]);
}

// This can be optimized to
double x[2], xp[2], y[2];
unsigned int n[2], i[2];
for (int j = 0; j < 2; j++) {
   xp[j] = x[j];
   y[j] = 1.0;
   for (i[j] = n[j]; i[j] != 0; i[j] >>= 1) {
      if (i[j] & 1) y[j] *= xp[j];
      xp[j] *= xp[j];
   }
}
```

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 13.9b. Raise vector to powers using integer mask, SSE4.1
section .data
align 16

one  DQ  1.0, 1.0            ; Make constant 1.0
X    DQ  2.0, 3.0            ; x[0], x[1]
Y    DQ  0, 0                ; y[0], y[1]
N    DD  4, 5                ; n[0], n[1]


section .text
default rel

; xmm0 = selector for conditional move
; xmm1 = xp
; xmm2 = i   (i0 and i1 each stored twice as DWORD integers)
; xmm3 = y
; xmm4 = (i & 1) ? xp : 1.0

    movq      xmm2, [N]      ; Load n0, n1
    punpckldq xmm2, xmm2     ; Get each value twice: n0, n0, n1, n1
    movapd    xmm1, [X]      ; Load x0, x1
    movapd    xmm3, [one]    ; y 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    xmm0, xmm2      ; Copy i
    pslld     xmm0, 31        ; Get least significant bit of i into
                              ; most significant bit
    movapd    xmm4, [one]     ; 1.0
    blendvpd  xmm4, xmm1, xmm0 ; (i & 1) ? xp : 1.0
    psrld     xmm2, 1         ; i >>= 1
    mulpd     xmm3, xmm4      ; y *= (i & 1) ? xp : 1.0
    mulpd     xmm1, xmm1      ; xp = xp * xp
    sub       ecx,  1         ; Loop counter
    jns       L1              ; Repeat ecx+1 times
    movapd    [Y],  xmm3      ; 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.

Conditional moves in general purpose registers using `CMOVcc` and floating point registers using `FCMOVcc` are no faster than in XMM registers.

13.5 Conditional moves with AVX512

The AVX512 instruction set supports masked operations controlled by a mask register. A masked vector instruction will do its operation only on those vector elements for which the corresponding bit in the mask register is 1. We can rewrite example 13.7 to illustrate this:

```
// Example 13.10a. Branch in loop using AVX512
float a[16], b[16], c[16], x[16], y[16];
for (int i = 0; i < 16; i++) {
   c[i] = x[i] > y[i] ? a[i] : b[i];
}
```

The implementation of this with AVX512 is quite efficient:

```
; Example 13.10b. Conditional move with AVX512
section .text
default rel

vmovaps  zmm1, [Y]        ; Load vector y
vcmpps k1, zmm1, [X], 1   ; Compare with x. k1 = mask for y < x
vmovaps  zmm0, [b]        ; Load vector b
vmovaps  zmm0{k1}, [a]    ; Load vector a for elements with mask bit 1
vmovaps  [c], zmm0        ; c = x > y ? a : b
```

Most vector instructions can be masked under AVX512, for example:

```
// Example 13.11a. Conditional addition using AVX512
float a[16], b[16];
for (int i = 0; i < 16; i++) {
   if (a[i] < 0) {
      a[i] += b[i];
   }
}
```

This can be implemented with a masked addition:

```
; Example 13.11b. Conditional add with AVX512
section .text
default rel

vmovaps  zmm1, [a]           ; Load vector a
vpxord   zmm0, zmm0, zmm0    ; Make zero
```

```
vcmpps k1, zmm1, zmm0, 1      ; Compare with zero. k1 = mask for a < 0
vaddps   zmm1{k1}, zmm1, [b] ; Masked addition
vmovaps  [a], zmm1            ; Store result
```

Masked instructions can also be used on the smaller XMM and YMM registers if the instruction set extension AVX512VL is supported. The `vpxord` instruction may be replaced by `vxorps` if instruction set AVX512DQ is supported.

In the above examples, the vector elements for which the mask bit is 0 are unchanged. It is possible to set the disabled elements to zero, rather than making them unchanged, by specifying the `{z}` option:

```
; Example 13.12. Masking and zeroing
section .text
default rel

mov     eax, 000FH          ; make constant
kmovw   k1, eax             ; copy constant to mask register
vmovaps zmm1{k1}, [a]       ; conditional move
vmovaps zmm2{k1}{z}, [b]    ; conditional move with zeroing
```

In this example, the first four elements of array `a` are loaded into `zmm1` and the remaining elements of `zmm1` are unchanged. The first four elements of array `b` are loaded into `zmm2` and the remaining elements of `zmm2` are set to zero. The zeroing option is useful because it removes the dependence of `zmm2` on the previous value of `zmm2`. It is recommended to use the zeroing option whenever it is appropriate because this makes out-or-order execution of masked instructions more efficient.

The masking comes for free in the sense that the latencies and throughputs of most instructions are the same with and without masking, regardless of the value of the mask. On the other hand, you may be wasting time on calculations where the mask is all zeroes. It may be advantageous to skip time-consuming conditional calculations if the mask is all zeroes. For example:

```
// Example 13.13a. Skip calculations if mask is all zeroes
float a[16], b[16];
for (int i = 0; i < 16; i++) {
   if (a[i] >= 0) {
      b[i] = sqrt(a[i]);
   }
}
```

If it often happens that all elements of the array `a` are negative then we may skip the time-consuming square root:

```
; Example 13.13b. Skip calculations if mask is all zeroes
section .text
default rel

vmovaps  zmm1, [a]          ; Load vector a
vpxord   zmm0, zmm0, zmm0   ; Make zero
vcmpps   k1, zmm1, zmm0, 1 ; Compare with zero. k1 = mask for a < 0
kortestw k1, k1             ; test bits of k1
jz       L1                 ; jump if all zero
vmovaps  zmm2, [b]          ; Load vector b
vsqrtps  zmm2{k1}, zmm1     ; square root of non-negative elements
vmovaps  [b], zmm2          ; Store modified b
L1:
```

A masked instruction can often replace two instructions because it does two things. Even compare instructions that have a mask register as output can be masked. Compare and test

instructions with a mask register as output and an additional mask applied always have an implicit zeroing option so that the output mask is zero, rather than unchanged, for bits where the input mask is zero. The result is a logical AND combination of the comparison result and the input mask. This means that AND combinations can be implemented more efficiently than OR combinations. The following example illustrates this:

```
// Example 13.14a. Use of masks
float a[16], b[16], c[16];
for (int i = 0; i < 16; i++) {
   if (a[i] > 0 || b[i] > 0) {
      c[i] = a[i] + b[i];
   }
   else {
      c[i] = a[i] - b[i];
   }
}
```

We can convert the OR (`||`) to AND (`&&`) in this example by inverting all inputs and outputs according to the formula: `a || b = !(!a && !b)`. This changes the code to:

```
// Example 13.14b. Use of masks, convert OR to AND
float a[16], b[16], c[16];
for (int i = 0; i < 16; i++) {
   if (a[i] <= 0 && b[i] <= 0) {
      c[i] = a[i] - b[i];
   }
   else {
      c[i] = a[i] + b[i];
   }
}
```

This is useful because the AND operation can be optimized by masking the second compare instruction with the result of the first one:

```
; Example 13.14c. Efficient use of masks
section .text
default rel

vmovups  zmm1, [a]             ; Load vector a
vmovups  zmm2, [b]             ; Load vector b
vpxord   zmm0, zmm0, zmm0      ; Make zero
vcmpps   k1, zmm1, zmm0, 2     ; k1 = mask for a <= 0
vcmpps   k2{k1}, zmm2, zmm0, 2 ; k2 = mask for a <= 0 && b <= 0
vaddps   zmm0, zmm1, zmm2      ; a + b
vsubps   zmm0{k2}, zmm1, zmm2  ; a - b if mask
vmovups  [c], zmm0             ; Store result
```

13.6 Using vector instructions with other types of data than they are intended for

Most XMM, YMM and ZMM instructions are 'typed' in the sense that they are intended for a particular type of data. For example, it does not 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 useful if an equivalent instruction does not exist for the type of data you have, or if an instruction for another type of data is more efficient.

All vector instructions that move, shuffle, blend, 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 subnormal 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 possible 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 performance penalty for using the wrong type of instructions on some processors. This is because the processor may have different data buses or different execution units for integer and floating point data. Moving data between the integer and floating point units can take one or more clock cycles depending on the processor, as listed in table 13.3.

Processor	Bypass delay, clock cycles
Intel Core 2 and earlier	1
Intel Nehalem	2
Intel Sandy Bridge and later	0-1
Intel Atom	0
AMD	2
VIA Nano	2-3

Table 13.3. Data bypass delays between integer and floating point execution units

On some Intel processors, a few floating point instructions are executed in the integer units. This includes XMM move instructions, Boolean, and some shuffle and pack instructions on Intel Core 2. These instructions have a bypass delay when mixed with instructions that use the floating point unit. On most other processors, the execution unit used is in accordance with the instruction name, e.g. `MOVAPS XMM1,XMM2` uses the floating point unit, `MOVDQA XMM1,XMM2` uses the integer unit.

Instructions that read or write memory use a separate unit. The bypass delay from the memory unit to the floating point unit may be longer than to the integer unit on some processors, but it does not depend on the type of the instruction. Thus, there is no difference in latency between `MOVAPS XMM0,[MEM]` and `MOVDQA XMM0,[MEM]` on current processors, but it cannot be ruled out that there will be a difference on future processors.

More details about the execution units of the different processors can be found in manual 3 "The microarchitecture of Intel, AMD and VIA CPUs". Manual 4: "Instruction tables" has lists of all instructions, indicating which execution units they use.

Using an instruction of a wrong type can be advantageous in cases where there is no bypass delay and in cases where throughput is more important than latency. 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 bypass delay occurs in some 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

An efficient way of setting a vector register to zero is `PXOR XMM0,XMM0`. Most processors recognize this instruction as being independent of the previous value of `XMM0`, while not all processors recognize the same for `XORPS` and `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 more different execution units than the floating point equivalents (`ANDPS`, etc.) on some AMD and Intel processors.

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 does not exist for the type of data you have.

There are many useful instructions for data shuffling and blending 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 bypass delay, if any, may be less than the cost of alternative solutions. The data shuffling instructions are listed in the next paragraph.

13.7 Permuting 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 reduces the advantage of using vector operations. It is possible to use a permutation 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 permutation are listed below.

Permute data within each 128-bit lane

Instruction	Block size, bits	Description	Instruction set
`PSHUFD`	32	General permute	SSE2
`PSHUFLW`	16	Permutes low half of register only	SSE2
`PSHUFHW`	16	Permutes high half of register only	SSE2
`SHUFPS`	32	Permute	SSE
`SHUFPD`	64	Permute	SSE2
`PSLLDQ`	8	Shifts to a different position and sets the original position to zero	SSE2
`PSHUFB`	8	General permute	Suppl. SSE3
`PALIGNR`	8	Rotate vector of 8 or 16 bytes	Suppl. SSE3
`PINSRB`	8	Insert byte into vector	SSE4.1
`PINSRW`	16	Insert word into vector	SSE
`PINSRD`	32	Insert dword into vector	SSE4.1
`PINSRQ`	64	Insert qword into vector	SSE4.1
`INSERTPS`	32	Insert dword into vector	SSE4.1
`VPERMILPS`	32	Permute with variable selector	AVX
`VPERMILPD`	64	Permute with variable selector	AVX
`VPPERM`	8	Permute with variable selector	AMD XOP

Table 13.4. Permute instructions

Permute data, crossing 128-bit lanes

Instruction	Block size, bits	Description	Instruction set
VPERMW	16	Permute with variable selector	AVX512BW
VPERMD	32	Permute with variable selector	AVX2
VPERMQ	64	Permute with variable selector	AVX2
VPERMPD	32	Permute with variable selector	AVX2
VPERMPD	64	Permute with variable selector	AVX2
VPERMI/T2B	8	Permute from two sources	AVX512VBMI
VPERMI/T2W	16	Permute from two sources	AVX512BW
VPERMI/T2D	32	Permute from two sources	AVX512
VPERMI/T2Q	64	Permute from two sources	AVX512
VPERMI/T2PS	32	Permute from two sources	AVX512
VPERMI/T2PD	64	Permute from two sources	AVX512
VPERM2I128	128	Permute from two sources	AVX2
VPERM2F128	128	Permute from two sources	AVX

Table 13.5. Full permute instructions

Combining data from two different sources

Instruction	Block size, bits	Description	Instruction set
SHUFPS	32	Lower 2 dwords from any position of destination higher 2 dwords from any position of source	SSE
SHUFPD	64	Low qword from any position of destination high qword from any position of source	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
PINSRB	8	Insert byte into vector	SSE4.1
PINSRW	16	Insert word into vector	SSE
PINSRD	32	Insert dword into vector	SSE4.1
INSERTPS	32	Insert dword into vector	SSE4.1
PINSRQ	64	Insert qword into vector	SSE4.1
VINSERTF128	128	Insert xmmword into vector	AVX
PALIGNR	8	Double shift (analogous to SHRD)	Suppl. SSE3
PBLENDW	16	Blend from two different sources	SSE4.1
BLENDPS	32	Blend from two different sources	SSE4.1
BLENDPD	64	Blend from two different sources	SSE4.1
PBLENDVB	8	Multiplexer	SSE4.1
BLENDVPS	32	Multiplexer	SSE4.1
BLENDVPD	64	Multiplexer	SSE4.1
VPCMOV	1	Multiplexer	AMD XOP
VPBLENDD	32	Multiplexer	AVX2
VPBLENDMB	8	Multiplexer with mask as selector	AVX512BW
VPBLENDMW	16	Multiplexer with mask as selector	AVX512BW
VPBLENDMD	32	Multiplexer with mask as selector	AVX512
VPBLENDMQ	64	Multiplexer with mask as selector	AVX512
VBLENDMPS	32	Blend with mask as selector	AVX512
VBLENDMPD	64	Blend with mask as selector	AVX512

Table 13.6. Combine data

Broadcasting data to all elements of a register

Instruction	Block size, bits	Description	Instruc-tion 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
VBROADCASTSS	32	broadcast dword from memory	AVX
VBROADCASTSS	32	broadcast dword from register	AVX2
VBROADCASTSD	64	broadcast qword from memory	AVX
VBROADCASTSD	64	broadcast qword from register	AVX2
VBROADCASTF128	128	broadcast 16 bytes from memory	AVX
VPBROADCASTB	8	broadcast byte from register or memory	AVX2
VPBROADCASTW	16	broadcast word from register or memory	AVX2
VPBROADCASTD	32	broadcast dword from register or memory	AVX2
VPBROADCASTQ	64	broadcast qword from register or memory	AVX2
VBROADCASTF32X2	64	broadcast qword from register or memory	AVX512DQ
VBROADCASTF32X4	128	broadcast 16 bytes from memory	AVX512
VPBROADCASTMB2Q	8	broadcast mask register to vector	AVX512CD
VPBROADCASTMW2D	16	broadcast mask register to vector	AVX512CD

Table 13.7. Move and broadcast data

In addition to these instructions, many AVX512 instructions have an option to use a broadcast memory operand.

Merge data from different memory locations into one vector (gather)

Instruction	Block size, bits	Description	Instruc-tion set
VPGATHERDD	32	gather dwords with dword indices	AVX2
VPGATHERQD	32	gather dwords with qword indices	AVX2
VPGATHERDQ	64	gather qwords with dword indices	AVX2
VPGATHERQQ	64	gather qwords with qword indices	AVX2
VGATHERDPS	32	gather dwords with dword indices	AVX2
VGATHERQPS	32	gather dwords with qword indices	AVX2
VGATHERDPD	64	gather qwords with dword indices	AVX2
VGATHERQPD	64	gather qwords with qword indices	AVX2

Table 13.8. Gather instructions

Vectorized table lookup

Permute instructions can be used for table lookup if the entire table can be contained in one or a few vector registers. If the table is too big then you have to use the slower gather instructions for table lookup.

Horizontal addition

The following examples show how to add all elements of a vector

```
; Example 13.15a. Add 16 elements in vector of 8-bit unsigned integers
; (SSE2)
movaps     xmm1,  [source]  ; Source vector, 16 8-bit integers
pxor       xmm0,  xmm0      ; 0
psadbw     xmm1,  xmm0      ; Sum of 8 differences
pshufd     xmm0,  xmm1, 0EH ; Get bit 64-127 from xmm1
paddd      xmm0,  xmm1      ; Sum
movd       [sum], xmm0      ; Store sum

; Example 13.15b. Add eight elements in vector of 16-bit integers
; using horizontal add instruction(SSSE3)
movaps     xmm0,  [source]   ; Source vector, 8 16-bit integers
phaddw     xmm0,  xmm0
phaddw     xmm0,  xmm0
phaddw     xmm0,  xmm0
movq       [sum], xmm0      ; Store sum

; The phaddw instruction is slow. This alternative may be faster:
; Example 13.15c. Add eight elements in vector of 16-bit integers
; avoiding horizontal add instruction(SSE2)
movaps     xmm0,  [source]   ; Source vector, 8 16-bit integers
punpckhqdq xmm1,  xmm0       ; Get elements 4, 5, 6, 7
paddw      xmm0,  xmm1       ; Add four and four elements
pshufd     xmm1,  xmm0, 1    ; Get element 2, 3
paddw      xmm0,  xmm1       ; Add two and two elements
pshuflw    xmm1,  xmm0, 1    ; Get element 1
paddw      xmm0,  xmm1       ; Add one and one elements
movq       [sum], xmm0       ; Store sum
```

```
; Example 13.15d. Add eight elements in vector of floats (AVX)
vmovaps    ymm0, [source]    ; Source vector, 8 32-bit floats
vextractf128 xmm1, ymm0, 1   ; Get upper half
vaddps     xmm0,  xmm0, xmm1 ; Add
vhaddps    xmm0,  xmm0, xmm0
vhaddps    xmm0,  xmm0, xmm0
vmovsd     [sum], xmm0       ; Store sum

; The vhaddps instruction is slow. This alternative may be faster:
; Example 13.15e. Add eight elements in vector of 32-bit floats
; avoiding horizontal add instruction (AVX)
vmovaps      ymm0, [a]           ; Source vector, 8 32-bit floats
vextractf128 xmm1, ymm0, 1       ; Get element 4, 5, 6, 7
vaddps       xmm0, xmm0, xmm1    ; Add four elements
vmovhlps     xmm1, xmm0, xmm0    ; Get element 2, 3
vaddps       xmm0, xmm0, xmm1    ; Add two elements
vshufps      xmm1, xmm0, xmm0, 1 ; Get element 1
vaddss       xmm0, xmm0, xmm1    ; Add one element
vmovsd       [b],  xmm0          ; Store sum
```

13.8 Generating constants

There is no instruction for moving a constant into a vector register. The default way of putting a constant into a vector 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 of 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 vector 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 13.9 below shows how to make various integer constants in XMM registers. The same value is generated in all elements 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` `pabsb xmm0,xmm0`	`pcmpeqw xmm0,xmm0` `psrlw xmm0,15`	`pcmpeqd xmm0,xmm0` `psrld xmm0,31`	`pcmpeqw xmm0,xmm0` `psrlq xmm0,63`

2	`pcmpeqw xmm0,xmm0`
	`pabsb xmm0,xmm0`
	`paddb 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`
	`pabsb xmm0,xmm0`
	`psllw xmm0,2`
		`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 paddb 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 13.9. Generate integer vector constants

Table 13.10 below shows how to make various floating point constants in XMM registers. The same value is generated in one or all elements in the vector:

Making floating point constants in XMM registers

Value	scalar single	scalar double	vector single	vector double
0.0	xorps xmm0,xmm0	xorps xmm0,xmm0	xorps xmm0,xmm0	xorps 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 (32 bit mode)	mov eax, value
	movd xmm0,eax
		mov eax, value>>32
		movd xmm0,eax
		psllq xmm0,32
			mov eax, value
			movd xmm0,eax
			shufps xmm0,xmm0,0
				mov eax, value>>32
				movd xmm0,eax
				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 13.10. 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 vector of doubles in xmm0:

```
; Example 13.16. 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 vector of doubles in xmm0:

```
; Example 13.17. 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 13.10 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 13.18a. 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 13.18b. 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.

13.9 Accessing unaligned data and partial vectors

All data that are read or written with vector registers should preferably be aligned by the vector size. See page 82 for how to align data.

Older processors have a penalty for accessing unaligned data, while modern processors handle unaligned data almost as fast as aligned data.

There are situations where alignment is not possible, for example if a library function receives a pointer to an array and it is unknown whether the array is aligned or not.

The following methods can be used for reading unaligned vectors:

Using unaligned read instructions

The instructions `MOVDQU`, `MOVUPS`, `MOVUPD` and `LDDQU` are all able to read unaligned vectors. The unaligned read instructions are relatively slow on older Intel processors and on Intel Atom, but fast on Nehalem and later Intel processors as well as on AMD and VIA processors.

```
; Example 13.19. Unaligned vector read
; esi contains pointer to unaligned array
movdqu  xmm0, [esi]              ; Read vector unaligned
```

On contemporary processors, there is no penalty for using the unaligned instruction `MOVDQU` rather than the aligned `MOVDQA` if the data are in fact aligned. Therefore, it is convenient to use `MOVDQU` if you are not sure whether the data are aligned or not.

Using VEX-prefixed instructions

Instructions with VEX prefix allow unaligned memory operands while the same instructions without VEX prefix will generate an exception if the operand is not properly aligned:

```
; Example 13.20. VEX versus non-VEX access to unaligned data
movups xmm1, [esi]    ; unaligned operand must be read first
addps xmm0, xmm1
```

```
; VEX prefix allows unaligned operand
vaddps xmm0, [esi]    ; unaligned operand allowed here
```

You should never mix VEX and non-VEX code if the YMM or ZMM registers are used. It is OK to mix AVX and AVX-512 instructions, though. See chapter 13.1.

Partially overlapping reads

Make the first read from the unaligned address and the next read from the nearest following 16-bytes boundary. The first two reads will therefore possibly overlap:

```
; Example 13.21. First unaligned read overlaps next aligned read
; esi contains pointer to unaligned array
movdqu  xmm1, [esi]             ; Read vector unaligned
add     esi,  10H
and     esi,  -10H              ; = nearest following 16B boundary
movdqa  xmm2, [esi]             ; Read next vector aligned
```

Here, the data in `xmm1` and `xmm2` will be partially overlapping if `esi` is not divisible by 16. This, of course, only works if the following algorithm allows the redundant data. The last vector read can also overlap with the last-but-one if the end of the array is not aligned.

Reading from the nearest preceding 16-bytes boundary

It is possible to start reading from the nearest preceding 16-bytes boundary of an unaligned array. This will put irrelevant data into part of the vector register, and these data must be ignored. Likewise, the last read may go past the end of the array until the next 16-bytes boundary:

```
; Example 13.22. Reading from nearest preceding 16-bytes boundary
; esi contains pointer to unaligned array
mov     eax, esi                ; Copy pointer
and     esi, -10H               ; Round down to value divisible by 16
and     eax, 0FH                ; Array is misaligned by this value
movdqa  xmm1, [esi]             ; Read from preceding 16B boundary
movdqa  xmm2, [esi+10H]         ; Read next block
```

In the above example, `xmm1` contains `eax` bytes of junk followed by (16-`eax`) useful bytes. `xmm2` contains the remaining `eax` bytes of the vector followed by (16-`eax`) bytes which are either junk or belonging to the next vector. The value in `eax` should then be used for masking out or ignoring the part of the register that contains junk data.

Here we are taking advantage of the fact that vector sizes, cache line sizes and memory page sizes are always powers of 2. The cache line boundaries and memory page boundaries will therefore coincide with vector boundaries. While we are reading some irrelevant data with this method, we will never load any unnecessary cache line, because cache lines are always aligned by some multiple of 16. There is therefore no cost to reading the irrelevant data. And, more importantly, we will never get a read fault for reading from non-existing memory addresses, because data memory is always allocated in pages of 4096 (= 2^{12}) bytes or more so that the aligned vector read can never cross a memory page boundary.

An example using this method is shown in the `strlenSSE2.asm` example in the appendix www.agner.org/optimize/asmexamples.zip.

Combine two unaligned vector parts into one aligned vector

The above method can be extended by combining the valid parts of two registers into one full register. If `xmm1` in example 13.22 is shifted right by the unalignment value (`eax`) and `xmm2` is shifted left by (16-`eax`) then the two registers can be combined into one vector containing only valid data.

Unfortunately, there is no instruction to shift a whole vector register right or left by a variable count (analogous to `shr eax,cl`) so we have to use a permutation instruction. The next example uses a byte permutation instruction, `PSHUFB`, with appropriate mask values for shifting a vector register right or left:

```
; Example 13.23. Combining two unaligned parts into one vector.
; (Supplementary-SSE3 instruction set required)

; This example takes the squareroot of n floats in an unaligned
; array src and stores the result in an aligned array dest.
; C++ code:
; const int n = 100;
; float * src;
; float dest[n];
; for (int i=0; i<n; i++) dest[i] = sqrt(src[i]);

; Define masks for using PSHUFB instruction as shift instruction:
; The 16 bytes from SMask[16+a] will shift right a bytes
; The 16 bytes from SMask[16-a] will shift left  a bytes
section .data
align 64
SMask:
  DB -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
  DB  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15
  DB -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1

section .text
default rel

    mov     esi,  [src]             ; Unaligned pointer src
    lea     edi,  [dest]            ; Aligned array dest
    mov     eax,  esi
    and     eax,  0FH               ; Get misalignment, a
    movdqu  xmm4, [SMask+10H+eax]   ; Mask for shift right by a
    movdqu  xmm5, [SMask+eax]       ; Mask for shift left by 16-a
    and     esi,  -10H              ; Nearest preceding 16B boundary
    xor     ecx,  ecx               ; Loop counter i = 0

L:  ; Loop
    movdqa  xmm1, [esi+ecx]         ; Read from preceding boundary
    movdqa  xmm2, [esi+ecx+10H]     ; Read next block
    pshufb  xmm1, xmm4              ; shift right by a
    pshufb  xmm2, xmm5              ; shift left by 16-a
    por     xmm1, xmm2              ; combine blocks
    sqrtps  xmm1, xmm1              ; compute four squareroots
    movaps  [edi+ecx], xmm1         ; Save result aligned
    add     ecx, 10H                ; Loop to next four values
    cmp     ecx, 400                ; 4*n
    jb      L                       ; Loop
```

Align `SMask` by 64 if possible to avoid misaligned reads across a cache line boundary.

This method gets easier if the value of the misalignment (`eax`) is a known constant. The number of instructions can be reduced by using the `PALIGNR` instruction when the shift count is constant.

Now we have discussed several methods for reading unaligned vectors. We also have to discuss how to write vectors to unaligned addresses. Some of the methods are virtually the same.

Using unaligned write instructions

The instructions `movdqu`, `movups`, and `movupd` are all able to write unaligned vectors. The unaligned write instructions are relatively slow on older Intel processors, but fast on Nehalem and later Intel processors as well as contemporary AMD and VIA processors.

```
; Example 13.24. Unaligned vector write
; edi contains pointer to unaligned array
movdqu  [edi], xmm0             ; Write vector unaligned
```

Partially overlapping writes

Make the first write to the unaligned address and the next write to the nearest following 16-bytes boundary. The first two writes will therefore possibly overlap:

```
; Example 13.25. First unaligned write overlaps next aligned write
; edi contains pointer to unaligned array
movdqu  [edi], xmm1             ; Write vector unaligned
add     edi,  10H
and     edi,  0FH               ; = nearest following 16B boundary
movdqa  [edi], xmm2             ; Write next vector aligned
```

Here, the data from `xmm1` and `xmm2` will be partially overlapping if `edi` is not divisible by 16. This, of course, only works if the algorithm can generate the overlapping data. The last vector write can also overlap with the last-but-one if the end of the array is not aligned. See the `memset` example in the appendix www.agner.org/optimize/asmexamples.zip.

Writing the beginning and the end separately

Use non-vector instructions for writing from the beginning of an unaligned array until the first 16-bytes boundary, and again from the last 16-bytes boundary to the end of the array. See the `memcpy` example in the appendix www.agner.org/optimize/asmexamples.zip.

Using masked write

The instruction `VPMASKMOVD` (AVX2) and `VMASKMOVPS` (AVX), etc. can be used for writing to the first part of an unaligned array up to the first 16-bytes boundary as well as the last part after the last 16-bytes boundary.

The non-VEX versions of the masked move instructions, such as `MASKMOVDQU` are extremely slow because they are bypassing the cache and writing to main memory (a so-called non-temporal write). The VEX versions are faster than the non-VEX versions on Intel processors, but not an AMD Bulldozer and Piledriver processors. These instructions should definitely be avoided.

The masked move versions in AVX512 are fast and useful.

13.10 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 appropriate 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 13.26. Byte vector addition 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.

It is also possible to search for a specific byte. This C code illustrates the principle by checking if a 32-bit integer contains at least one byte of zero:

```
// Example 13.27. Return nonzero if dword contains null byte
inline int dword_has_nullbyte(int w) {
   return ((w - 0x01010101) & ~w & 0x80808080);}
```

The output is zero if all four bytes of `w` are nonzero. The output will have 0x80 in the position of the first byte of zero. If there are more than one bytes of zero then the subsequent bytes are not necessarily indicated. (This method was invented by Alan Mycroft and published in 1987. I published the same method in 1996 in the first version of this manual unaware that Mycroft had made the same invention before me).

This principle can be used for finding the length of a zero-terminated string by searching for the first byte of zero. It is faster than using `REPNE SCASB`:

```
; Example 13.28, optimized strlen procedure (32-bit):
_strlen PROC    NEAR
; extern "C" int strlen (const char * s);

        push    ebx                     ; ebx must be saved
        mov     ecx, [esp+8]            ; get pointer to string
        mov     eax, ecx                ; copy pointer
        and     ecx, 3                  ; lower 2 bits, check alignment
        jz      L2                      ; s is aligned by 4. Go to loop
        and     eax, -4                 ; align pointer by 4
        mov     ebx, [eax]              ; read from preceding boundary
        shl     ecx, 3                  ; *8 = displacement in bits
        mov     edx, -1
        shl     edx, cl                 ; make byte mask
        not     edx                     ; mask = 0FFH for false bytes
        or      ebx, edx                ; mask out false bytes

        ; check first four bytes for zero
        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
        jnz     L3                      ; zero-byte found

        ; Main loop, read 4 bytes aligned
L1:     add     eax, 4                  ; increment pointer
L2:     mov     ebx, [eax]              ; read 4 bytes of string
        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

L3:     bsf     ecx, ecx                ; find right-most 1-bit
        shr     ecx, 3                  ; divide by 8 = byte index
        sub     eax, [esp+8]            ; subtract start address
        add     eax, ecx                ; add index to byte
        pop     ebx                     ; restore ebx
        ret                             ; return value in eax
_strlen ENDP
```

The alignment check makes sure that we are only reading from addresses aligned by 4. The function may read both before the beginning of the string and after the end, but since all reads are aligned, we will not cross any cache line boundary or page boundary unnecessarily. Most importantly, we will not get any page fault for reading beyond allocated memory because page boundaries are always aligned by 2^{12} or more.

If the SSE2 instruction set is available then it may be faster to use XMM instructions with the same alignment trick. Example 13.28 as well as a similar function using XMM registers are provided in the appendix at www.agner.org/optimize/asmexamples.zip. and in my assembly library at www.agner.org/optimize/asmlib.zip

Other common functions that search for a specific byte, such as `strcpy`, `strchr`, `memchr` can use the same trick. To search for a byte different from zero, just XOR with the desired byte as illustrated in this C code:

```
// Example 13.29. Return nonzero if byte b contained in dword w
inline int dword_has_byte(int w, unsigned char b) {
   w ^= b * 0x01010101;
   return ((w - 0x01010101) & ~w & 0x80808080);}
```

Note if searching backwards (e.g. `strrchr`) that the above method will indicate the position of the *first* occurrence of `b` in `w` in case there is more than one occurrence.

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 cores. The way to make use of multiple CPU cores is to divide the computing job between multiple threads. The optimal number of threads is usually equal to the number of CPU cores. The workload should ideally be evenly divided 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 to starting and stopping threads and to synchronizing the threads. Communication between threads can be quite costly, although these costs are reduced on newer processors. The computing job should preferably be divided into threads at the highest possible level. If the outermost loop can be parallelized, then it might be divided into one loop for each thread, each doing its share of the whole job.

Thread-local storage should preferably use the stack. Static thread-local memory is inefficient and should be avoided.

14.1 Simultaneous multithreading

Most modern microprocessors have multiple CPU cores so that they can run multiple threads simultaneously. Furthermore, many Intel and AMD processors can run two threads

in each core. The Knight's Corner and Knight's Landing processors can even run four threads per core. "Hyperthreading" is Intel's term for simultaneous multithreading.

Two or more threads running in the same core will always compete for the same resources, such as cache, instruction decoder, and execution units. If any of the shared resources are limiting factors for the performance of a single thread then there is no advantage in running two threads in each core. On the contrary, each thread may run at less than half speed because of cache evictions and other resource conflicts. But if a large fraction of the time goes to cache misses, branch misprediction, or long dependency chains, then each thread will run at more than half the single-thread speed. In this case there is an advantage to using simultaneous multithreading, but the performance is not doubled. A thread that shares the resources of the core with another thread will always run slower than a thread that runs alone in the core.

The higher the capacity of a CPU core, the higher the advantage of simultaneous multithreading. The Knight's Landing processor has a quite low capacity so that it is rarely advantageous to run two threads – and much less four threads – in this processor. Simultaneous multithreading is more useful on the older Knight's Corner processor because it has no out-of-order processing capability.

It may be necessary to do experiments in order to determine whether it is advantageous to use simultaneous multithreading or not in a particular application.

A particular disadvantage of simultaneous multithreading is that a low priority thread may steal resources from a high priority thread running in the same CPU core. Current operating systems are not always handling this problem optimally.

See manual 1: "Optimizing software in C++" for more details on multithreading and simultaneous multithreading.

15 CPU dispatching

We often want to take advantage of the advanced instruction sets such as AVX2 and AVX512. There is a problem if we want the same program to be able to run on microprocessors without these instruction sets. The solution is to have multiple versions of the critical part of the program, each optimized for a different instruction set. The appropriate version of the code is selected at run time after checking the capabilities of the CPU it is running on. This is called automatic CPU dispatching.

Manual 1 "Optimizing software in C++" chapter 13 has important advices on CPU dispatching.

CPU dispatching can be implemented with branches or with a function pointer, as shown in the following example.

```
Example 15.1. Function with CPU dispatching

section .data
MyFunctionPoint: DQ MyFunctionDispatch  ; Function pointer

section .text
default rel
extern InstructionSet

MyFunction:
    ; Jump through pointer. The code pointer initially points to
    ; MyFunctionDispatch. MyFunctionDispatch changes the pointer
```

```
    ; so that it points to the appropriate version of MyFunction.
    ; The next time MyFunction is called, it jumps directly to
    ; the right version of the function
    jmp     [MyFunctionPoint]

; Code for each version:

MyFunctionAVX512:
    ; AVX512 version of MyFunction
    ...
    ret

MyFunctionAVX2:
    ; AVX2 version of MyFunction
    ...
    ret

MyFunctionSSE2:
    ; Generic/SSE2 version of MyFunction
    ; (All microprocessors with x64 also support SSE2.
    ; If the program is running in 32-bit mode then you may need a
    ; 80386 compatible generic version)
    ...
    ret


MyFunctionDispatch:
    ; Detect which instruction set is supported.
    ; Function InstructionSet is in asmlib
    push   rcx                  ; Save any registers that may be used
                                ; for parameters to MyFunction
    push   rdx
    call   InstructionSet       ; eax indicates CPU instruction set
    lea    rdx, [MyFunctionSSE2]
    cmp    eax, 13              ; eax >= 13 if AVX2
    jb     DispEnd
    lea    rdx, [MyFunctionAVX2]
    cmp    eax, 16              ; eax >= 16 if AVX512BW/DQ/VL
    jb     DispEnd
    lea    rdx, [MyFunctionAVX512]

DispEnd:
    ; Save pointer to appropriate version of MyFunction
    mov    [MyFunctionPoint], rdx
    mov    rax, rdx             ; address of function
    pop    rdx                  ; restore registers
    pop    rcx
    jmp    rax                  ; Jump to this version of the function
```

The function `InstructionSet` detects which instruction set is supported. This function is provided in the library that can be downloaded from /www.agner.org/optimize/asmlib.zip. Most operating systems also have functions for this purpose.

15.1 Checking for operating system support for XMM, YMM, and ZMM registers

Unfortunately, the information that can be obtained from the `CPUID` instruction is not sufficient for determining whether it is possible to use vector 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 vector 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 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.

It is easier to check for operating support for YMM and ZMM registers. Execute CPUID with eax = 1. Check that bit 27 and 28 in ecx are both 1 (OSXSAVE and AVX feature flags). If so, then execute XGETBV with ecx = 0 to get the XFEATURE_ENABLED_MASK. Check that

bit 1 and 2 in `eax` are both set (XMM and YMM state support). If so, then it is safe to use YMM registers. ZMM registers are enabled if bit 5, 6, and 7 are also all set.

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.

"Intel 64 and IA-32 Architectures Software Developer's Manual", 2018.

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

"Intel Advanced Vector Extensions Programming Reference", 2008, 2010.

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 rax, [rbx+8*rcx-1000]
```

is much faster than

```
; Example 16.1b
mov  rax, rcx
shl  rax, 3
add  rax, rbx
sub  rax, 1000
```

A typical use of `LEA` is as a three-register addition: `lea rax,[rbx+rcx]`. 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 rax,[rbx*2]` is actually coded as `lea rax,[rbx*2+00000000H]` with an immediate displacement of 4 bytes. The size of this instruction can be reduced by writing `lea rax,[rbx+rbx]`. 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  rax, [rbx*4]       ; 8 bytes
lea  rax, [rcx+rbx*4]   ; 4 bytes
```

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 26). 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 76. Use this version when the result is sure to be less than 2^{32}. The upper half of RBX and RCX are ignored in this case. 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}.

LEA is slower than addition on some processors. The more complex forms of LEA with scale factor and offset are slower than the simple form on many processors. See manual 4: "Instruction tables" for details on each processor.

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 the address size prefix 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

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 µop on some CPUs. It can cause a partial flags stalls on some older Intel processors if a subsequent instruction reads the carry flag or all the flag bits. On all processors, 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 or when no penalty is expected.

16.3 XCHG (all processors)

The `XCHG register,[memory]` instruction is dangerous. This instruction always has an implicit LOCK prefix which forces synchronization with other processors or cores. This instruction is therefore very time consuming, and should always be avoided unless the lock is intended.

The XCHG instruction with register operands may be useful when optimizing for size as explained on page 73.

16.4 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.

16.5 Bit test (all processors)

BT, BTC, BTR, and BTS instructions should preferably be replaced by instructions like TEST, AND, OR, XOR, or shifts on older processors. Bit tests with a memory operand should be avoided on Intel processors. BTC, BTR, and BTS use 2 µops on AMD processors. Bit test instructions are useful when optimizing for size.

16.6 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 of the oldest 64-bit Intel processors.

16.7 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,[RAX+4*RAX]` in 64-bit mode or `LEA EAX,[EAX+4*EAX]` in 32-bit mode.

16.8 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 eax by 2^N
shr  eax, N
```

Dividing a signed integer by 2^N:

```
; Example 16.4. Divide signed integer eax by 2^N
cdq
and  edx, (1 shl N) - 1    ; (Or:  shr edx,32-N)
add  eax, edx
sar  eax, N
```

Obviously, the unsigned version is preferred if the dividend is certain to be non-negative.

Integer division by a constant (all processors)

A floating point number can be divided by a constant by multiplying with the reciprocal. If we want to do the same with integers, we have to scale the reciprocal by 2^n and then shift the product to the right by n. There are various algorithms for finding a suitable value of n and compensating for rounding errors. The algorithm described below was invented by Terje Mathisen, Norway, and not published elsewhere. Other methods can be found in the book Hacker's Delight, by Henry S. Warren, Addison-Wesley 2003, and the paper: T. Granlund and P. L. Montgomery: Division by Invariant Integers Using Multiplication, Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation.

The following algorithm will give you the correct result for unsigned integer division with truncation, i.e. the same result as the `DIV` instruction:

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 eax 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 eax, 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 $2^{r-b}-1$, 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 multiplication instruction with shift and add instructions as explained on page 137 if multiplication is slow.

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 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
section .data
align 16
RECIPDIV times 8 DW 0CCCDH ; Vector of reciprocal divisor

section .text
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 it is advantageous to use the same method as above. The division is done only once while the multiplication and shift operations are done for each dividend. A branch-free variant of the algorithm is provided in the paper by Granlund and Montgomery cited above.

This method is implemented in the asmlib function library and in the C++ vector class library for signed and unsigned integers as well as for vector registers.

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 x87 style 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 do not 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

The following instructions are calculating the approximate reciprocals of single precision floating point numbers with various precisions.

instruction	precision	instruction set
rcpss, rcpps	12 bits	SSE
vrcp14ss, vrcp14ps	14 bits	AVX512
vrcp28ss, vrcp28ps	28 bits	AVX512ER

Table 16.1. Approximate reciprocal instructions

You can make approximate divisions by using one of the fast reciprocal instructions on the divisor and then multiply with the dividend. The precision can be increased from 12 to 23 bits by using the principle of Newton-Raphson iteration:

```
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 by the dividend.

```
; Example 16.6, 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
```

It is possible to increase the precision further by repeating the Newton-Raphson formula with double precision, but this is not very advantageous.

16.9 String instructions (all processors)

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

`REP MOVSD` and `REP STOSD` are quite fast if the repeat count is not too small. The largest word size (`DWORD` in 32-bit mode, `QWORD` in 64-bit mode) is preferred. Both source and destination should be aligned by the word size or better. In many cases, however, it is faster to use vector registers. Moving data in the largest available registers is faster than `REP MOVSD` and `REP STOSD` in most cases, especially on older processors. See page 150 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. This may cause cache bank conflicts on some old processors. The easiest way to avoid cache bank conflicts is to align both source and destination by 8.

On many processors, REP MOVS and REP STOS can perform quite fast by moving 16 bytes or an entire cache line at a time. This happens only when certain conditions are met. Depending on the processor, the conditions for fast string instructions are, typically, that the count must be high, both source and destination must be aligned, the direction must be forward, the distance between source and destination must be at least the cache line size, and the memory type for both source and destination must be either write-back or write-combining (you can normally assume the latter condition is met).

Under these conditions, the speed is as high as you can obtain with vector register moves.

While the string instructions can be quite convenient, it must be emphasized that other solutions are faster in many cases. If the above conditions for fast move are not met, then there is a lot to gain by using other methods. See page 150 for alternatives to REP MOVS.

REP LOADS, REP SCAS, and REP CMPS take more time per iteration than simple loops. See page 130 for alternatives to REPNE SCASB.

16.10 Vectorized string instructions (processors with SSE4.2)

The SSE4.2 instruction set contains four very powerful instructions for string search and compare operations: PCMPESTRI, PCMPISTRI, PCMPESTRM, PCMPISTRM. There are two vector operands each containing a string and an immediate operand determining what operation to do on these strings. These four instructions can all do each of the following kinds of operations, differing only in the input and output operands:

- Find characters from a set: Finds which of the bytes in the second vector operand belong to the set defined by the bytes in the first vector operand, comparing all 256 possible combinations in one operation.
- Find characters in a range: Finds which of the bytes in the second vector operand are within the range defined by the first vector operand.
- Compare strings: Determine if the two strings are identical.
- Substring search: Finds all occurrences of a substring defined by the first vector operand in the second vector operand.

The lengths of the two strings can be specified explicitly (PCMPESTRx) or implicitly with a terminating zero (PCMPISTRx). The latter is faster on current Intel processors because it has fewer input operands. Any terminating zeroes and bytes beyond the ends of the strings are ignored. The output can be a bit mask or byte mask (PCMPxSTRM) or an index (PCMPxSTRI). Further output is provided in the following flags. Carry flag: result is nonzero. Sign flag: first string ends. Zero flag: second string ends. Overflow flag: first bit of result.

These instructions are very efficient for various string parsing and processing tasks because they do large and complicated tasks in a single operation. However, the current implementations are slower than the best alternatives for simple tasks such as strlen and strcopy.

Examples can be found in my function library at www.agner.org/optimize/asmlib.zip.

16.11 WAIT instruction (all processors)

The WAIT instruction (also known as FWAIT) has three functions:

A. The old 8087 coprocessor 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 16.7. 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 [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 under DOS 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 coprocessors but not on later processors.

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 x87 floating point instructions will also generate an interrupt if a previous x87 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 x87 instruction in your program to be sure to catch all exceptions.

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.

It is sometimes faster to use integer instructions for comparing floating point values, as described on page 148 and 150.

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.

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 16.8.
    fistp qword [TEMP]
    fild  qword [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.

The conversion instructions such as CVTSS2SI and CVTTSS2SI. should be used instead on processors with SSE.

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:

Single precision, for $|N| < 2^7-1$:

```
; Example 16.9a. 2 to the power of i, single precision
    mov     eax, [N]
    shl     eax, 23
    add     eax, 3f800000h
    movd    xmm0, eax
```

Single precision, for $|N| < 2^{10}-1$:

```
; Example 16.9b. 2 to the power of i, double precision
    mov     eax, [N]
    shl     eax, 20
    add     eax, 3ff00000h
    movd    xmm0, eax
    psllq   xmm0, 32
```

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 do not have to test if Y contains a valid number if you know that the argument is valid.

When optimizing for speed, it is usually faster to use a library function with SSE2.

16.17 FSQRT, SQRTSS

The following instructions are available for calculation of reciprocal square roots:

instruction	precision	instruction set
rsqrtss, rsqrtps	12 bits, float	SSE
vrsqrt14ss, vrsqrt14ps	14 bits, float	AVX512
vrsqrt14sd, vrsqrt14pd	14 bits, double	AVX512
vrsqrt28ss, vrsqrt28ps	28 bits, float	AVX512ER
vrsqrt28sd, vrsqrt28pd	28 bits, double	AVX512ER

Table 16.2. Approximate reciprocal square root instructions

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

```
sqrt(x) = x * rsqrt(x)
```

The reciprocal square root instructions give the reciprocal square root with a precision of at least 12 bits. You can improve the precision to 23 bits by using the Newton-Raphson iteration formula:

```
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 `x` to get the square root.

16.18 FLDCW

Many processors 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 without SSE2, 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.

A better solution is to compile for the SSE or SSE2 instruction set and use truncation instructions such as `CVTTSS2SI`.

16.19 MASKMOV instructions

The masked memory write instructions `MASKMOVQ` and `MASKMOVDQU` are extremely slow because they are bypassing the cache and writing to main memory (a so-called non-temporal write).

The VEX-coded alternatives `VPMASKMOVD`, `VPMASKMOVQ`, `VMASKMOVPS`, `VMASKMOVPD` are writing to the cache, and therefore much faster on Intel processors. The VEX versions are still extremely slow on AMD Bulldozer and Piledriver processors.

These instructions should be avoided at all costs. The AVX512 instruction set provides masked writes, which are much faster. See also chapter 13.9 for alternative methods.

17 Special topics

17.1 XMM versus floating point registers

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 the old x87 floating point registers. The decision of whether to use the x87 floating point registers `ST(0)` - `ST(7)` or XMM registers depends on the following factors.

Advantages of using x87 registers:

- Compatible with old processors without SSE or SSE2.
- Compatible with old operating systems without XMM support.
- Supports long double precision.
- Intermediate results are calculated with long double precision.
- Precision conversions are free in the sense that they require no extra instructions and take no extra time. You may 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 faster than library functions using XMM registers.
- Conversions to and from decimal numbers can use the `FBLD` and `FBSTP` instructions when optimizing for size, but these instructions are slow.
- 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, YMM, or ZMM registers:

- Can do multiple 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/YMM or 32 ZMM registers, but only 8 ST() registers.
- ST() registers cannot be used in device drivers in 64-bit Windows.

- The instruction set for ST() registers is no longer developed. The instructions will probably still be supported for many years for the sake of backwards compatibility, but the instructions may work less efficiently in future processors.

17.2 MMX versus XMM registers

Integer vector instructions can use either the 64-bit MMX registers or the 128-bit XMM registers in processors with SSE2.

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 is doubled in XMM registers as compared to MMX registers, and increased further in YMM and ZMM registers.
- MMX registers cannot be used together with ST() registers.
- A series of MMX instructions must end with EMMS.
- 64-bit systems have 16 or 32 XMM registers, but only 8 MMX registers.
- MMX registers cannot be used in device drivers in 64-bit Windows.
- The instruction set for MMX registers is no longer developed and is going out of use. The MMX registers will probably still be supported in many years for reason of backwards compatibility.

17.3 XMM versus YMM and ZMM registers

The 128-bit XMM registers are extended to 256 or 512 bits in the YMM and ZMM registers when the newer instruction sets are available. See page 107 for details. Advantages of using the AVX instruction set and YMM or ZMM registers:

- More elements per vector
- Non-destructive 3-operand version of all XMM, YMM, and ZMM instructions
- Masked instructions are available with AVX512

Advantages of using XMM registers:

- Compatible with older processors
- There is a penalty for switching between VEX instructions and XMM instructions without VEX prefix, see page 108. The programmer may inadvertently mix VEX and non-VEX instructions.
- YMM and ZMM registers cannot be used in device drivers without saving everything with `XSAVE` / `XRESTOR`., see page 111.

17.4 Freeing floating point registers

You have to free all used x87 style floating point registers before exiting a subroutine, except for any register used for the result.

The fastest way of freeing one register is `FSTP ST`. To free two registers you may use either `FCOMPP` or twice `FSTP ST`, whichever fits best into the decoding sequence or port load.

It is not recommended to use `FFREE`.

17.5 Transitions between floating point and MMX instructions

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. There is no such penalty on processors with out-of-order execution.

17.6 Converting from floating point to integer

All conversions between x87 style floating point registers and integer registers must go via a memory location:

```
; Example 17.1.
fistp dword [TEMP]
mov eax, [TEMP]
```

On older 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 144.

You can avoid all these problems by using `XMM` registers instead of floating point registers and use the `CVT..` instructions to avoid the memory intermediate on processors with SSE2.

17.7 Using integer instructions for floating point operations

It is often advantageous to use integer instructions for doing simple floating point operations, because integer instructions are generally faster than floating point instructions. The most obvious example is moving data. For example

```
; Example 17.2a. Moving floating point data
fld qword [esi]
```

```
fstp qword [edi]
```

can be replaced by:

```
; Example 17.2b
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 124.

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 17.3a. Testing floating point value for zero
fld    dword [ebx]
ftst
fnstsw ax
and    ah, 40h
jnz    IsZero
```

can be replaced by

```
; Example 17.3b. 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 subnormal numbers can be ruled out, then you can be certain that the value is zero if the exponent bits are all zero. Example:

```
; Example 17.3c. Testing double value for zero
fld    qword [ebx]
ftst
fnstsw ax
and    ah, 40h
jnz    IsZero
```

can be replaced by

```
; Example 17.3d. 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 17.4. 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 17.5. 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 17.6. 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 17.7. 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 17.8. 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 17.9. Convert bits to value between 0 and 1
section .data
one     dq   1.0
x       dq   0
n       dd   0

section .text
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 17.10a. Compare single precision float numbers
fld    dword [a]
fcomp  dword [b]
fnstsw ax
and    ah, 1
jnz    ASmallerThanB
```

can be replaced by:

```
; Example 17.10b. 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 32-bit integer instructions.

17.8 Moving blocks of data

There are several ways of moving large blocks of data. The most common methods are:

1. `REP MOVS` instruction.

2. If data are aligned: Read and write in a loop with the largest available register size.

3. If size is constant: inline move instructions.

4. If data are misaligned: First move as many bytes as required to make the destination aligned. Then read unaligned and write aligned in a loop with the largest available register size.

5. If data are misaligned: Read aligned, shift to compensate for misalignment and write aligned.

6. If the data size is too big for caching, use non-temporal writes to bypass the cache. Shift to compensate for misalignment, if necessary.

Which of these methods is fastest depends on the microprocessor, the data size and the alignment. If a large fraction of the CPU time is used for moving data, then it is important to select the fastest method. I will therefore discuss the advantages and disadvantages of each method here.

The `REP MOVS` instruction (1) is a simple solution which is useful when optimizing for code size rather than for speed. This instruction is implemented as microcode in the CPU. The microcode implementation may actually use one of the other methods internally. In some cases it is well optimized, in other cases not. Usually, the `REP MOVS` instruction has a large overhead for choosing and setting up the right method. Therefore, it is not optimal for small blocks of data. For large blocks of data, it may be quite efficient when certain conditions for alignment etc. are met. These conditions depend on the specific CPU (see page 141). On Intel Nehalem and later processors, this is sometimes as fast as the other methods when the memory block is large.

Aligned moves with the largest available registers (2) is an efficient method on all processors. Therefore, it is worthwhile to align big blocks of data and, if necessary, pad the data in the end to make the size of each data block a multiple of the register size.

Inlined move instructions (3) is often the fastest method if the size of the data block is known at compile time, and the alignment is known. Part of the block may be moved in smaller registers to fit the specific alignment and data size. The loop of move instructions may be fully rolled out if the size is very small.

Unaligned read and aligned write (4) is fast on most newer processors. Typically, older processors have slow unaligned reads while newer processors have a better cache interface that optimizes unaligned read. The unaligned read method is reasonably fast on all newer processors, and on many processors it is the fastest method.

The aligned read - shift - aligned write method (5) is the fastest method for moving unaligned data on older processors. On processors with SSE2, use `PSRLDQ`/`PSLLDQ`/`POR`. On processors with SSSE3, use `PALIGNR` (except on Bobcat, where `PALIGNR` is particularly slow). It is necessary to have 16 different branches for the 16 possible shift counts. This can be avoided on AMD processors with XOP instructions, by using the `VPPERM` instruction, but the unaligned read method (4) may be at least as fast on these processors.

Using non-temporal writes (6) to bypass the cache can be advantageous when moving very large blocks of data. This is not always faster than the other methods, but it saves the cache for other purposes. As a rule of thumb, it can be good to use non-temporal writes when moving data blocks bigger than half the size of the last-level cache.

As you can see, it can be very difficult to choose the optimal method in a given situation. The best advice I can give for a universal `memcpy` function, based on my testing, is as follows:

- On Intel Wolfdale and earlier, Intel Atom, AMD K8 and earlier, and VIA Nano, use the aligned read - shift - aligned write method (5).
- On Intel Nehalem and later, method (4) is up to 33% faster than method (5).
- On AMD K10 and later and Bobcat, use the unaligned read - aligned write method (4).
- The non-temporal write method (6) can be used for data blocks bigger than half the size of the largest-level cache.

Another consideration is what size of registers to use for moving data. The size of vector registers has been increased from 64 bits to 128, 256, and 512 bits with various instruction set extensions. Historically, the first processors to support a new register size have often had limited bandwidth for operations on the larger registers. Some or all of the execution units, physical registers, data buses, read ports and write ports have had the size of the previous version, so that an operation on the largest register size is split internally into two operations of half the size. The first processors with SSE would split a 128-bit read into two 64-bit reads; and the first processors with AVX would split a 256-bit read into two 128-bit

reads. Therefore, there is no increase in bandwidth by using the largest registers on these processors. In some cases, e.g. AMD Bulldozer, there is significant decrease in performance by using the 256-bit registers. In these cases, it is not advantageous to use a new register size for moving data until the second generation of processors that support it. The use of 512-bit registers is advantageous on the first Intel processors to support 512-bit registers.

There is a further complication when copying blocks of memory, namely *false memory dependence*. Whenever the CPU reads a piece of data from memory, it checks if it overlaps with any preceding write. If this is the case, then the read must wait for the preceding write to finish or it must do a store forwarding. Unfortunately, the CPU may not be able to check the full physical address, but only the offset within the memory page, i.e. the lower 12 bits of the address. If the lower 12 bits of the read address overlaps with the lower 12 bits of any unfinished preceding write then the read is delayed. This happens when the source address relative to the destination address is a little less than a multiple of 4 kbytes. This false memory dependence can slow down memory copying by a factor of 3 - 4 in the worst case. The problem can be avoided by copying backwards, from the end to the beginning of the memory block, in the case where there is a false memory dependence. This is of course only allowed when no part of the source and the destination memory blocks truly overlaps.

The asmlib library includes optimized functions for moving data, and automatically selecting the optimal method for the processor it is running on.

Prefetching

It is sometimes advantageous to prefetch data items before they are needed, and some manuals recommend this. However, modern processors have automatic prefetching. The processor hardware is constructed so that it can predict which data addresses will be needed next, and prefetch it automatically, when data items are accessed in a linear manner, both forwards and backwards. Explicit prefetching is rarely needed. The optimal prefetch strategy is processor specific. Therefore, it is easier to rely on automatic prefetching. Explicit prefetching is particularly bad on Intel Ivy Bridge and AMD Jaguar processors.

Moving data on future processors

We must bear in mind that the code that is written today will run on the processors of tomorrow. Therefore, it is important to optimize for future processors as explained in the section "CPU dispatch strategies" in Manual 1: "Optimizing software in C++". Therefore, I will discuss here which method is likely to be fastest on future processors. First, we can predict that the AVX512 (and later) instruction sets are likely to be supported on all future high-end Intel x86 processors. Smaller low power processors with 128 or 256-bit execution units can implement a 512-bit instruction as two 256-bit µops or four 128-bit µops. All processors with AVX will have reasonably fast unaligned read operations because the AVX instructions have few restrictions on alignment. As the trend goes towards having two read ports, the memory read operations are unlikely to be a bottleneck, even if unaligned. Therefore, we can expect that the unaligned read method (method 4) will be fast on future processors.

Many modern processors have optimized the `REP MOVS` instruction (method 1) to use the largest available register size and the fastest method, at least in simple cases. But there are still cases where the `REP MOVS` method is slow, for example for certain misalignment cases and false memory dependence. However, the `REP MOVS` instruction has the advantage that it will probably use the largest available register size on processors in a more distant future with registers bigger than 512 bits. As instructions with the expected future register sizes cannot yet be coded and tested, the `REP MOVS` instruction is the only way we can write code today that will take advantage of future extensions to the register size. Therefore, it may be useful to use the `REP MOVS` instruction for favorable cases of large aligned memory blocks.

It would be useful if CPU vendors provided information in the CPUID instruction to help select the optimal code or, alternatively, implemented the best method as microcode in the `REP MOVS` instruction.

Examples of the abovementioned methods can be found in the function library at www.agner.org/optimize/asmlib.zip. See manual 1: "Optimizing software in C++" for a discussion of automatic CPU dispatching, and for a comparison of available function libraries.

Processor-specific advice on improving memory access can be found in Intel's "Intel 64 and IA-32 Architectures Optimization Reference Manual" and AMD's "Software Optimization Guide for AMD Family xx Processors".

17.9 Self-modifying code

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 and 64-bit Windows call `VirtualProtect` and `FlushInstructionCache`. The trick of putting the code in a data segment does not work in newer systems that use the execute disable bit.

Self-modifying code is not considered good programming practice. It should be used only 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 that 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. A more 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.

Many microprocessors are able to change the clock frequency in the CPU core in response to changing workloads. The clock frequency is increased when the workload is high, and decreased when the workload is low, in order to save power. Intel processors have a "core clock counter" which counts at the actual core clock frequency. The core clock counter gives more consistent and reproducible results. This is useful when comparing alternative implementations of a piece of code, while the time stamp counter gives a more realistic

measure of how long time the code actually takes to execute. A device driver is needed to enable the core clock counter because this requires privileged access. Once the core clock counter is activated, it can be read without privileged mode. A test program that can enable the core clock counter is mentioned below.

You have to insert `XOR EAX,EAX` / `CPUID` before and after each read of the counters in order to prevent it from executing in parallel with anything else on processors with out-of-order execution. `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.

A serious problem when counting clock cycles 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 and 2. These programs are available at www.agner.org/optimize/testp.zip.

A further complication occurs on processors with multiple cores if a thread can jump from one core to another. The time stamp counters on different cores are not necessarily synchronized. This is not a problem when testing small pieces of code if the above precautions are taken to minimize interrupts. But it can be a problem when measuring longer time intervals. You may need to lock the process to a single CPU core, for example with the function `SetProcessAffinityMask` in Windows. This is discussed in the document "Game Timing and Multicore Processors", Microsoft 2005 http://msdn.microsoft.com/en-us/library/ee417693.aspx. The `RDTSCP` instruction allows you to detect if the task has jumped to a different CPU core.

You will soon observe when 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.

Alignment effects on some old Intel processors can make time measurements difficult. 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 point will be aligned differently and the instruction fetch blocks will be different. If instruction fetch and decoding

is a bottleneck, then the change in the alignment may make the code several clock cycles faster or slower. AMD processors may also have alignment effects on tiny loops.

Most x86 processors also have a set of so-called performance monitor counters. These counters can count events such as cache misses, misalignments, branch mispredictions, etc. They 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 Software Developer's Manual, 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 Linux and Windows. These test program support the different kinds of performance monitor counters in most Intel, AMD, and VIA 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 CodeXL.

18.2 The pitfalls of unit-testing

If you want to find out which version of a function performs best, then it is not sufficient to measure clock cycles in a small test program that calls the function many times. Such a test is unlikely to give a realistic measure of cache misses because the test program may use less memory than the cache size. For example, an isolated test may show that it is advantageous to roll out a loop, while a test where the function is inserted in the final program shows a large amount of cache misses when the loop is rolled out.

Therefore, it is important not only to count clock cycles when evaluating the performance of a function, but also to consider how much space it uses in the code cache, data cache, and branch target buffer.

See the section named "The pitfalls of unit-testing" in manual 1: "Optimizing software in C++" for further discussion of this problem.

19 Literature

The present manual is part of a series available from www.agner.org/optimize as mentioned in the introduction on page 4. See manual 3: "The microarchitecture of Intel, AMD, and VIA CPUs" of a list of relevant literature on specific processors.

A lot of other sources also have useful information. Some useful resources are listed at www.agner.org/optimize.

Some useful books:

R. C. Detmer: Introduction to 80x86 Assembly Language and Computer Architecture, 2'nd ed. Jones & Bartlett, 2006. Jones & Bartlett, 2006.
Good introduction to assembly programming

J. L. Hennessy and D. A. Patterson: Computer Architecture: A Quantitative Approach, 3'rd ed. 2002.
Good textbook on computer architecture and microarchitecture

John R. Levine: Linkers and Loaders. Morgan Kaufmann, 2000.

Explains how linkers and loaders work

Henry S. Warren, Jr.: "Hacker's Delight". Addison-Wesley, 2003.
Contains many bit manipulation tricks

20 Copyright notice