

X86 Assembly/Print Version

= Introduction =^[1]

Introduction

Why Learn Assembly?

Assembly is among some of the oldest tools in a computer-programmer's toolbox. Nowadays though, entire software projects can be written without ever looking at a single line of assembly code. So this pops up the question: why learn assembly? Assembly language is one of the closest forms of communication that humans can engage in with a computer. With assembly, the programmer can precisely track the flow of data and execution in a program in a mostly human-readable form. Once a program has been compiled, it is difficult (and at times, nearly impossible) to reverse-engineer the code into its original form. As a result, if you wish to examine a program that is already compiled but would rather not stare at hexadecimal or binary, you will need to examine it in assembly language. Since debuggers will frequently only show program code in assembly language, this provides one of many benefits for learning the language.

Assembly language is also the preferred tool, if not the only tool, for implementing some low-level tasks, such as bootloaders and low-level kernel components. Code written in assembly has less overhead than code written in high-level languages, so assembly code frequently will run much faster than equivalent programs written in other languages. Also, code that is written in a high-level language can be compiled into assembly and "hand optimized" to squeeze every last bit of speed out of it. As hardware manufacturers such as Intel and AMD add new features and new instructions to their processors, often times the only way to access those features is to use assembly routines. That is, at least until the major compiler vendors add support for those features.

Developing a program in assembly can be a very time consuming process, however. While it might not be a good idea to write new projects in assembly language, it is certainly valuable to know a little bit about it.

Who is This Book For?

This book will serve as an introduction to assembly language and a good resource for people who already know about the topic, but need some more information on x86 system architecture. It will also describe some of the more advanced uses of x86 assembly language. All readers are encouraged to read (and contribute to) this book, although prior knowledge of programming fundamentals would definitely be beneficial.

How is This Book Organized?

The first section will discuss the x86 family of chips and introduce the basic instruction set. The second section will explain the differences between the syntax of different assemblers. The third section will go over some of the additional instruction sets available, including the floating point, MMX, and SSE operations.

The fourth section will cover some advanced topics in x86 assembly, including some low-level programming tasks such as writing bootloaders. There are many tasks that cannot be easily implemented in a higher-level language such as C or C++. For example, enabling and disabling interrupts, enabling protected mode, accessing the Control Registers, creating a Global Descriptor Table, and other tasks all need to be handled in assembly. The fourth section will also deal with interfacing assembly language with C and other high-level languages. Once a function is written in assembly (a function to enable protected mode, for instance), we can interface that function to a larger, C-based (or even C++ based) kernel. The fifth section will discuss the standard x86 chipset, cover the basic x86 computer architecture, and generally deal with the hardware side of things.

The current layout of the book is designed to give readers as much information as they need without going overboard. Readers who want to learn assembly language on a given assembler only need to read the first section and the chapter in the second section that directly relates to their assembler. Programmers looking to implement the MMX or SSE instructions for different algorithms only really need to read section 3. Programmers looking to implement bootloaders, kernels, or other low-level tasks, can read section 4. People who really want to get to the nitty-gritty of the x86 hardware design can continue reading on through section 5.

Basic FAQ

This page is going to serve as a basic FAQ for people who are new to assembly language programming.

How Does the Computer Read/Understand Assembly?

The computer doesn't really "read" or "understand" anything *per se*, since a computer has no awareness nor consciousness, but that's beside the point. The fact is that the computer cannot read the assembly language that you write. Your assembler will convert the assembly language into a form of binary information called "machine code" that your computer uses to perform its operations. If you don't assemble the code, it's complete gibberish to the computer.

That said, assembly is important because each assembly instruction usually relates to just a single machine code, and it is possible for "mere mortals" to do this task directly with nothing but a blank sheet of paper, a pencil, and an assembly instruction reference book. Indeed, in the early days of computers this was a common task and even required in some instances "hand assembling" machine instructions for some basic computer programs. A classical example of this was done by [Steve Wozniak](#), when he hand assembled the entire Integer BASIC interpreter into 6502 machine code for use on his initial [Apple I](#) computer. It should be noted, however, that such tasks done for commercially distributed software are so rare that they deserve special mention from that fact alone. Very, very few programmers have actually done this for more than a few instructions, and even then only for a classroom assignment.

Is it the Same On Windows/DOS/Linux?

The answers to this question are yes and no. The basic x86 machine code is dependent only on the processor. The x86 versions of Windows and Linux are obviously built on the x86 machine code. There are a few differences between Linux and Windows programming in x86 Assembly:

1. On a Linux computer, the most popular assemblers are the GAS assembler, which uses the AT&T syntax for writing code, and the Netwide Assembler, also known as NASM, which uses a syntax similar to MASM.
2. On a Windows computer, the most popular assembler is MASM, which uses the Intel syntax but also, a lot of Windows users use NASM.
3. The available software interrupts, and their functions, are different on Windows and Linux.
4. The available code libraries are different on Windows and Linux.

Using the same assembler, the basic assembly code written on each Operating System is basically the same, except you interact with Windows differently than you interact with Linux.

Which Assembler is Best?

The short answer is that none of the assemblers is better than any other; it's a matter of personal preference.

The long answer is that different assemblers have different capabilities, drawbacks, etc. If you only know GAS syntax, then you will probably want to use GAS. If you know Intel syntax and are working on a Windows machine, you might want to use MASM. If you don't like some of the quirks or complexities of MASM and GAS, you might want to try FASM or NASM. We will cover the differences between the different assemblers in section 2.

Do I Need to Know Assembly?

You don't *need* to know assembly for most computer tasks, but it can definitely be useful. Learning assembly is not about learning a new programming language. If you are going to start a new programming project (unless that project is a bootloader, a device driver, or a kernel), then you will probably want to avoid assembly like the plague. An exception to

this could be if you absolutely need to squeeze the last bits of performance out of a congested inner loop and your compiler is producing suboptimal code. Keep in mind, though, that premature optimization is the root of all evil, although some computing-intense realtime tasks can only be optimized sufficiently if optimization techniques are understood and planned for from the start.

However, learning assembly gives you a particular insight into how your computer works on the inside. When you program in a higher-level language like C or Ada, all your code will eventually need to be converted into machine code instructions so your computer can execute them. Understanding the limits of exactly what the processor can do, at the most basic level, will also help when programming a higher-level language.

How Should I Format my Code?

Most assemblers require that assembly code instructions each appear on their own line and are separated by a carriage return. Most assemblers also allow for whitespace to appear between instructions, operands, etc. Exactly how you format code is up to you, although there are some common ways:

One way keeps everything lined up:

```
Label1:
mov ax, bx
add ax, bx
jmp Label3
Label2:
mov ax, cx
...
```

Another way keeps all the labels in one column and all the instructions in another column:

```
Label1: mov ax, bx
        add ax, bx
        jmp Label3
Label2: mov ax, cx
...
```

Another way puts labels on their own lines and indents instructions slightly:

```
Label1:
    mov ax, bx
    add ax, bx
    jmp Label3
Label2:
    mov ax, cx
...
```

Yet another way separates labels and instructions into separate columns AND keeps labels on their own lines:

```
Label1:      mov ax, bx
              add ax, bx
              jmp Label3
Label2:      mov ax, cx
...
```

So there are different ways to do it, but there are some general rules that assembly programmers generally follow:

1. Make your labels obvious, so other programmers can see where they are.
2. More structure (indents) will make your code easier to read.
3. Use comments to explain what you are doing. The meaning of a piece of assembly code can often not be immediately clear.

X86 Family

The term "x86" can refer both to an instruction set architecture and to microprocessors which implement it. The name x86 is derived from the fact that many of Intel's early processors had names ending in "86".

The x86 instruction set architecture originated at Intel and has evolved over time by the addition of new instructions as well as the expansion to 64-bits. As of 2009, x86 primarily refers to IA-32 (Intel Architecture, 32-bit) and/or x86-64, the extension to 64-bit computing.

Versions of the x86 instruction set architecture have been implemented by Intel, AMD and several other vendors, with each vendor having its own family of x86 processors.

Intel x86 Microprocessors

8086/8087 (1978)

The 8086 was the original x86 microprocessor, with the 8087 as its floating-point coprocessor. The 8086 was Intel's first 16-bit microprocessor with a 20-bit address bus, thus enabling it to address up to 1 MiB, although the architecture of the original IBM PC imposed a limit of 640 KiB of RAM, with the remainder reserved for ROM and memory-mapped expansion cards, such as video memory. This limitation is still present in modern CPUs, since they all support the backward-compatible "Real Mode" and boot into it.

8088 (1979)

After the development of the 8086, Intel also created the lower-cost 8088. The 8088 was similar to the 8086, but with an 8-bit data bus instead of a 16-bit bus. The address bus was left untouched.

80186/80187 (1982)

The 186 was the second Intel chip in the family; the 80187 was its floating point coprocessor. Except for the addition of some new instructions, optimization of some old ones, and an increase in the clock speed, this processor was identical to the 8086.

80286/80287 (1982)

The 286 was the third model in the family; the 80287 was its floating point coprocessor. The 286 introduced the "Protected Mode" mode of operation, in addition to the "Real Mode" that the earlier models used. All subsequent x86 chips can also be made to run in real mode or in protected mode. Switching back from protected mode to real mode was initially not supported, but found to be possible (although relatively slow) by resetting the CPU, then continuing in real mode. Although the processor featured an address bus with 24 lines (24 bits, thus enabling to address up to 16 MiB), these could only be used in protected mode. In real mode, the processor was still limited to the 20-bits address bus.

80386 (1985)

The 386 was the fourth model in the family. It was the first Intel microprocessor with a 32-bit word. The 386DX model was the original 386 chip, and the 386SX model was an economy model that used the same instruction set, but which only had a 16-bit data bus. Both featured a 32-bits address bus, thus getting rid of the segmented addressing methods used in the previous models and enabling a "flat" memory model, where one register can hold an entire address, instead of relying on two 16-bit registers to create a 20-bit/24-bit address. The flat memory layout was only supported in protected mode. Also, contrary to the 286, it featured an "unreal mode" in which protected-mode software could switch to perform real-mode operations (although this backward compatibility was not complete, as the physical memory was still protected). The 386EX model is still used today in embedded systems,

80486 (1989)

The 486 was the fifth model in the family. It had an integrated floating point unit for the first time in x86 history. Early model 80486 DX chips were found to have defective FPUs. They were physically modified to disconnect the FPU portion of the chip and sold as the 486SX (486-SX15, 486-SX20, and 486-SX25). A 487 "math coprocessor" was available to 486SX users and was essentially a 486DX with a working FPU and an extra pin added. The arrival of the 486DX-50 processor saw the widespread introduction of fan assisted heat-sinks being used to keep the processors from overheating.

Pentium (1993)

Intel called it the "Pentium" because they couldn't trademark the code number "80586". The original Pentium was a faster chip than the 486 with a few other enhancements; later models also integrated the MMX instruction set.

Pentium Pro (1995)

The Pentium Pro was the sixth-generation architecture microprocessor, originally intended to replace the original Pentium in a full range of applications, but later reduced to a more narrow role as a server and high-end desktop chip.

Pentium II (1997)

The Pentium II was based on a modified version of the P6 core first used for the Pentium Pro, but with improved 16-bit performance and the addition of the MMX SIMD instruction set, which had already been introduced on the Pentium MMX.

Pentium III (1999)

Initial versions of the Pentium III were very similar to the earlier Pentium II, the most notable difference being the addition of SSE instructions.

Pentium 4 (2000)

The Pentium 4 had a new 7th generation "NetBurst" architecture. Pentium 4 chips also introduced the notions "Hyper-Threading", and "Multi-Core" chips.

Core (2006)

The architecture of the Core processors was actually an even more advanced version of the 6th generation architecture dating back to the 1995 Pentium Pro. The limitations of the NetBurst architecture, especially in mobile applications, were too great to justify creation of more NetBurst processors. The Core processors were designed to operate more efficiently with a lower clock speed. All Core branded processors had two processing cores; the Core Solos had one core disabled, while the Core Duos used both processors.

Core 2 (2006)

An upgraded, 64-bit version of the Core architecture. All desktop versions are multi-core.

i Series (2008)

The successor to Core 2 processors, with the i7 line featuring Hyper-Threading.

Celeron (first model 1998)

The Celeron chip is actually a large number of different chip designs, depending on price. Celeron chips are the economy line of chips, and are frequently cheaper than the Pentium chips—even if the Celeron model in question is based off a Pentium architecture.

Xeon (first model 1998)

The Xeon processors are modern Intel processors made for servers, which have a much larger cache (measured in MiB in comparison to other chips' KiB-sized cache) than the Pentium microprocessors.

AMD x86 Compatible Microprocessors

Athlon

Athlon is the brand name applied to a series of different x86 processors designed and manufactured by AMD. The original Athlon, or Athlon Classic, was the first seventh-generation x86 processor and, in a first, retained the initial performance lead it had over Intel's competing processors for a significant period of time.

Turion

Turion 64 is the brand name AMD applies to its 64-bit low-power (mobile) processors. Turion 64 processors (but not Turion 64 X2 processors) are compatible with AMD's Socket 754 and are equipped with 512 or 1024 KiB of L2 cache, a 64-bit single channel on-die memory controller, and an 800 MHz HyperTransport bus.

Duron

The AMD Duron was an x86-compatible computer processor manufactured by AMD. It was released as a low-cost alternative to AMD's own Athlon processor and the Pentium III and Celeron processor lines from rival Intel.

Sempron

Sempron is, as of 2006, AMD's entry-level desktop CPU, replacing the Duron processor and competing against Intel's Celeron D processor.

Opteron

The AMD Opteron is the first eighth-generation x86 processor (K8 core), and the first of AMD's AMD64 (x86-64) processors. It is intended to compete in the server market, particularly in the same segment as the Intel Xeon processor.

X86 Architecture

x86 Architecture

The x86 architecture has 8 General-Purpose Registers (GPR), 6 Segment Registers, 1 Flags Register and an Instruction Pointer. 64-bit x86 has additional registers.

General-Purpose Registers (GPR) - 16-bit naming conventions

The 8 GPRs are:

1. Accumulator register (AX). Used in arithmetic operations
2. Counter register (CX). Used in shift/rotate instructions and loops.
3. Data register (DX). Used in arithmetic operations and I/O operations.
4. Base register (BX). Used as a pointer to data (located in segment register DS, when in segmented mode).

5. Stack Pointer register (SP). Pointer to the top of the stack.
6. Stack Base Pointer register (BP). Used to point to the base of the stack.
7. Source Index register (SI). Used as a pointer to a source in stream operations.
8. Destination Index register (DI). Used as a pointer to a destination in stream operations.

The order in which they are listed here is for a reason: it is the same order that is used in a push-to-stack operation, which will be covered later.

All registers can be accessed in 16-bit and 32-bit modes. In 16-bit mode, the register is identified by its two-letter abbreviation from the list above. In 32-bit mode, this two-letter abbreviation is prefixed with an 'E' (*extended*). For example, 'EAX' is the accumulator register as a 32-bit value.

Similarly, in the 64-bit version, the 'E' is replaced with an 'R' (*register*), so the 64-bit version of 'EAX' is called 'RAX'.

It is also possible to address the first four registers (AX, CX, DX and BX) in their size of 16-bit as two 8-bit halves. The least significant byte (LSB), or low half, is identified by replacing the 'X' with an 'L'. The most significant byte (MSB), or high half, uses an 'H' instead. For example, CL is the LSB of the counter register, whereas CH is its MSB.

In total, this gives us five ways to access the accumulator, counter, data and base registers: 64-bit, 32-bit, 16-bit, 8-bit LSB, and 8-bit MSB. The other four are accessed in only four ways: 64-bit, 32-bit, 16-bit, and 8-bit. The following table summarises this:

identifiers to access registers and parts thereof

Register	Accumulator		Counter		Data		Base		Stack Pointer		Stack Base Pointer		Source		Destination	
64-bit	RAX		RCX		RDX		RBX		RSP		RBP		RSI		RDI	
32-bit		EAX		ECX		EDX		EBX		ESP		EBP		ESI		EDI
16-bit		AX		CX		DX		BX		SP		BP		SI		DI
8-bit		AH	AL	CH	CL	DH	DL	BH	BL	SPL		BPL		SIL		DIL

Segment Registers

The 6 Segment Registers are:

- Stack Segment (SS). Pointer to the stack.
- Code Segment (CS). Pointer to the code.
- Data Segment (DS). Pointer to the data.
- Extra Segment (ES). Pointer to extra data ('E' stands for 'Extra').
- F Segment (FS). Pointer to more extra data ('F' comes after 'E').
- G Segment (GS). Pointer to still more extra data ('G' comes after 'F').

Most applications on most modern operating systems (like FreeBSD, Linux or Microsoft Windows) use a memory model that points nearly all segment registers to the same place (and uses paging instead), effectively disabling their use. Typically the use of FS or GS is an exception to this rule, instead being used to point at thread-specific data.

EFLAGS Register

The EFLAGS is a 32-bit register used as a collection of bits representing Boolean values to store the results of operations and the state of the processor.

The names of these bits are:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

The bits named 0 and 1 are reserved bits and shouldn't be modified.

The different use of these flags are:

0. CF : Carry Flag. Set if the last arithmetic operation carried (addition) or borrowed (subtraction) a bit beyond the size of the register. This is then checked when the operation is followed with an add-with-carry or subtract-with-borrow to deal with values too large for just one register to contain.
2. PF : Parity Flag. Set if the number of set bits in the least significant byte is a multiple of 2.
4. AF : Adjust Flag. Carry of Binary Code Decimal (BCD) numbers arithmetic operations.
6. ZF : Zero Flag. Set if the result of an operation is Zero (0).
7. SF : Sign Flag. Set if the result of an operation is negative.
8. TF : Trap Flag. Set if step by step debugging.
9. IF : Interruption Flag. Set if interrupts are enabled.
10. DF : Direction Flag. Stream direction. If set, string operations will decrement their pointer rather than incrementing it, reading memory backwards.
11. OF : Overflow Flag. Set if signed arithmetic operations result in a value too large for the register to contain.
- 12-13. IOPL : I/O Privilege Level field (2 bits). I/O Privilege Level of the current process.
14. NT : Nested Task flag. Controls chaining of interrupts. Set if the current process is linked to the next process.
16. RF : Resume Flag. Response to debug exceptions.
17. VM : Virtual-8086 Mode. Set if in 8086 compatibility mode.
18. AC : Alignment Check. Set if alignment checking of memory references is done.
19. VIF : Virtual Interrupt Flag. Virtual image of IF.
20. VIP : Virtual Interrupt Pending flag. Set if an interrupt is pending.
21. ID : Identification Flag. Support for CPUID instruction if can be set.

Instruction Pointer

The EIP register contains the address of the **next** instruction to be executed if no branching is done.

EIP can only be read through the stack after a `call` instruction.

Memory

The x86 architecture is little-endian, meaning that multi-byte values are written least significant byte first. (This refers only to the ordering of the bytes, not to the bits.)

So the 32 bit value $B3B2B1B0_{16}$ on an x86 would be represented in memory as:

Little endian representation

B0	B1	B2	B3
----	----	----	----

For example, the 32 bits double word $0x1BA583D4$ (the **0x** denotes hexadecimal) would be written in memory as:

Little endian example

D4	83	A5	1B
----	----	----	----

This will be seen as `0xD4 0x83 0xA5 0x1B` when doing a memory dump.

Two's Complement Representation

Two's complement is the standard way of representing negative integers in binary. The sign is changed by inverting all of the bits and adding one.

Two's

complement example

Start:	0001
Invert:	1110
Add One:	1111

0001 represents decimal 1

1111 represents decimal -1

Addressing modes

The addressing mode indicates the manner in which the operand is presented.

Register Addressing

(operand address R is in the address field)

```
mov ax, bx ; moves contents of register bx into ax
```

Immediate

(actual value is in the field)

```
mov ax, 1 ; moves value of 1 into register ax
```

or

```
mov ax, 010Ch ; moves value of 0x010C into register ax
```

Direct memory addressing

(operand address is in the address field)

```
.data
my_var dw 0abcdh ; my_var = 0xabcd
.code
mov ax, [my_var] ; copy my_var content into ax (ax=0xabcd)
```

Direct offset addressing

(uses arithmetics to modify address)

```
byte_table db 12, 15, 16, 22 ; table of bytes
mov al, [byte_table + 2]
mov al, byte_table[2] ; same as previous instruction
```

Register Indirect

(field points to a register that contains the operand address)

```
mov ax, [di]
```

The registers used for indirect addressing are BX, BP, SI, DI

General-purpose registers (64-bit naming conventions)

64-bit x86 adds 8 more general-purpose registers, named R8, R9, R10 and so on up to R15.

- R8–R15 are the new 64-bit registers.
- R8D–R15D are the lowermost 32 bits of each register.
- R8W–R15W are the lowermost 16 bits of each register.
- R8B–R15B are the lowermost 8 bits of each register.

As well, 64-bit x86 includes SSE2, so each 64-bit x86 CPU has at least 8 registers (named XMM0–XMM7) that are 128 bits wide, but only accessible through SSE instructions. They cannot be used for quadruple-precision (128-bit) floating-point arithmetic, but they can each hold 2 double-precision or 4 single-precision floating-point values for a SIMD parallel instruction. They can also be operated on as 128-bit integers or vectors of shorter integers. If the processor supports AVX, as newer Intel and AMD desktop CPUs do, then each of these registers is actually the lower half of a 256-bit register (named YMM0–YMM7), the whole of which can be accessed with AVX instructions for further parallelization.

Stack

The stack is a Last In First Out (LIFO) data structure; data is pushed onto it and popped off of it in the reverse order.

```
mov ax, 006Ah
mov bx, F79Ah
mov cx, 1124h

push ax ; push the value in AX onto the top of the stack, which now holds the value 0x006A.
push bx ; do the same thing to the value in BX; the stack now has 0x006A and 0xF79A.
push cx ; now the stack has 0x006A, 0xF79A, and 0x1124.

call do_stuff ; do some stuff. The function is not forced to save the registers it uses, hence us saving them.

pop cx ; pop the element on top of the stack, 0x1124, into CX; the stack now has 0x006A and 0xF79A.
pop bx ; pop the element on top of the stack, 0xF79A, into BX; the stack now has just 0x006A.
pop ax ; pop the element on top of the stack, 0x006A, into AX; the stack is now empty.
```

The Stack is usually used to pass arguments to functions or procedures and also to keep track of control flow when the `call` instruction is used. The other common use of the Stack is temporarily saving registers.

CPU Operation Modes

Real Mode

Real Mode is a holdover from the original Intel 8086. You generally won't need to know anything about it (unless you are programming for a DOS-based system or, more likely, writing a boot loader that is directly called by the BIOS).

The Intel 8086 accessed memory using 20-bit addresses. But, as the processor itself was 16-bit, Intel invented an addressing scheme that provided a way of mapping a 20-bit addressing space into 16-bit words. Today's x86 processors start in the so-called Real Mode, which is an operating mode that mimics the behavior of the 8086, with some very tiny differences, for backwards compatibility.

In Real Mode, a segment and an offset register are used together to yield a final memory address. The value in the segment register is multiplied by 16 (shifted 4 bits to the left) and the offset is added to the result. This provides a usable address space of 1 MB. However, a quirk in the addressing scheme allows access past the 1 MB limit if a segment address of 0xFFFF (the highest possible) is used; on the 8086 and 8088, all accesses to this area wrapped around to the low end of memory, but on the 80286 and later, up to 65520 bytes past the 1 MB mark can be addressed this way if the A20 address line is enabled. See: [The A20 Gate Saga](#).

One benefit shared by Real Mode segmentation and by Protected Mode Multi-Segment Memory Model is that all addresses must be given relative to another address (this is, the segment base address). A program can have its own address space and completely ignore the segment registers, and thus no pointers have to be relocated to run the program. Programs can perform *near* calls and jumps within the same segment, and data is always relative to segment base addresses (which in the Real Mode addressing scheme are computed from the values loaded in the Segment Registers).

This is what the DOS *.COM format does; the contents of the file are loaded into memory and blindly run. However, due to the fact that Real Mode segments are always 64 KB long, COM files could not be larger than that (in fact, they had to fit into 65280 bytes, since DOS used the first 256 bytes of a segment for housekeeping data); for many years this wasn't a problem.

Protected Mode

Flat Memory Model

If programming in a modern 32-bit operating system (such as Linux, Windows), you are basically programming in flat 32-bit mode. Any register can be used in addressing, and it is generally more efficient to use a full 32-bit register instead of a 16-bit register part. Additionally, segment registers are generally unused in flat mode, and it is generally a bad idea to touch them.

Multi-Segmented Memory Model

Using a 32-bit register to address memory, the program can access (almost) all of the memory in a modern computer. For earlier processors (with only 16-bit registers) the segmented memory model was used. The 'CS', 'DS', and 'ES' registers are used to point to the different *chunks* of memory. For a small program (small model) the CS=DS=ES. For larger memory models, these 'segments' can point to different locations.

Long Mode

The term "Long Mode" refers to the 64-bit mode.

Comments

Comments

When writing code, it is very helpful to use some comments explaining what is going on. A comment is a section of regular text that the assembler ignores when turning the assembly code into the machine code. In assembly comments are usually denoted with a semicolon ";", although GAS uses "#" for single line comments and "/* ... */" for multi-line comments.

Here is an example:

```
Label1:
    mov ax, bx ; move contents of bx into ax
    add ax, bx ; add the contents of bx into ax
    ...
```

Everything after the semicolon, on the same line, is ignored. Let's show another example:

```
Label1:
    mov ax, bx
    ;mov cx, ax
    ...
```

Here, the assembler never sees the second instruction "mov cx, ax", because it ignores everything after the semicolon. When someone reads the code in the future they will find the comments and hopefully try to figure out what the programmer intended.

HLA Comments

The HLA assembler also has the ability to write comments in C or C++ style, but we can't use the semicolons. This is because in HLA, the semicolons are used at the end of every instruction:

```
mov(ax, bx); //This is a C++ comment.
/*mov(cx, ax); everything between the slash-stars is commented out.
    This is a C comment*/
```

C++ comments go all the way to the end of the line, but C comments go on for many lines from the "/*" all the way until the "*/". For a better understanding of C and C++ comments in HLA, see Programming:C or the C++ Wikibooks.

16 32 and 64 Bits

When using x86 assembly, it is important to consider the differences between architectures that are 16, 32, and 64 bits. This page will talk about some of the basic differences between architectures with different bit widths.

Registers

16-bit

The registers found on the 8086 and all subsequent x86 processors are the following: AX, BX, CX, DX, SP, BP, SI, DI, CS, SS, ES, DS, IP and FLAGS. These are all 16 bits wide.

On any Windows-based system (except 64 bit versions), you can run a very handy program called "debug.exe" from a DOS shell, which is very useful for learning about 8086. If you are using DOSBox or FreeDOS, you can use "debug.exe" as provided by FreeDOS. (<http://www.ibiblio.org/pub/micro/pc-stuff/freedos/files/distributions/1.2/repos/pkg-html/debug.html>)

AX, BX, CX, DX

These general purpose registers can also be addressed as 8-bit registers. So AX = AH (high 8-bit) and AL (low 8-bit).

SI, DI

These registers are usually used as offsets into data space. By default, SI is offset from the DS data segment, DI is offset from the ES extra segment, but either or both of these can be overridden.

SP

This is the stack pointer, offset usually from the stack segment SS. Data is pushed onto the stack for temporary storage, and popped off the stack when it is needed again.

BP

The stack frame, usually treated as an offset from the stack segment SS. Parameters for subroutines are commonly pushed onto the stack when the subroutine is called, and BP is set to the value of SP when a subroutine starts. BP can then be used to find the parameters on the stack, no matter how much the stack is used in the meanwhile.

CS, DS, ES, SS

The segment pointers. These are the offset in memory of the current code segment, data segment, extra segment, and stack segment respectively.

IP

The instruction pointer. Offset from the code segment CS, this points at the instruction currently being executed.

FLAGS (F)

A number of single-bit flags that indicate (or sometimes set) the current status of the processor.

32-bit

With the chips beginning to support a 32-bit data bus, the registers were also widened to 32 bits. The names for the 32-bit registers are simply the 16-bit names with an 'E' prepended.

EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI

These are the 32-bit versions of the registers shown above.

EIP

The 32-bit version of IP. Always use this instead of IP on 32-bit systems.

EFLAGS

An expanded version of the 16-bit FLAGS register.

64-bit

The names of the 64-bit registers are the same of those of the 16-bit registers, except beginning with an 'R'.

RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI

These are the 64-bit versions of the registers shown above.

RIP

This is the full 64-bit instruction pointer and should be used instead of EIP (which will be inaccurate if the address space is larger than 4 GiB, which may happen even with 4 GiB or less of RAM).

R8–15

These are new extra registers for 64-bit. They are counted as if the registers above are registers zero through seven, inclusively, rather than one through eight.

R8–R15 can be accessed as 8-bit, 16-bit, or 32-bit registers. Using R8 as an example, the names corresponding to those widths are R8B, R8W, and R8D, respectively. 64-bit versions of x86 also allow the low byte of RSP, RBP, RSI, RDI to be accessed directly. For example, the low byte of RSP can be accessed using SPL. There is no way to directly access bits 8–15 of those registers, as AH allows for AX.

128-bit, 256-bit and 512-bit (SSE/AVX)

64-bit x86 includes SSE2 (an extension to 32-bit x86), which provides 128-bit registers for specific instructions. Most CPUs made since 2011 also have AVX, a further extension that lengthens these registers to 256 bits. Some also have AVX-512, which lengthens them to 512 bits and adds 16 more registers.

XMM0~7

SSE2 and newer.

XMM8~15

SSE3 and newer and AMD (but not Intel) SSE2.

YMM0~15

AVX. Each YMM register includes the corresponding XMM register as its lower half.

ZMM0~15

AVX-512F. Each ZMM register includes the corresponding YMM register as its lower half.

ZMM16~31

AVX-512F. 512-bit registers that are not addressable in narrower modes unless AVX-512VL is implemented.

XMM16~31

AVX-512VL. Each is the lower quarter of the corresponding ZMM register.

YMM16~31

AVX-512VL. Each is the lower half of the corresponding ZMM register.

Addressing memory

8086 and 80186

The original 8086 only had registers that were 16 bits in size, effectively allowing to store one value of the range $[0 - (2^{16} - 1)]$ (or simpler: it could address up to 65536 different bytes, or 64 kibibytes) - but the address bus (the connection to the memory controller, which receives addresses, then loads the content from the given address, and returns the data back on the data bus to the CPU) was 20 bits in size, effectively allowing to address up to 1 mebibyte of memory. That means that all registers by themselves were not large enough to make use of the entire width of the address bus, leaving 4 bits unused, scaling down the size of usable addresses by 16 bytes ($1024 \text{ KiB} / 64 \text{ KiB} = 16 \text{ bytes}$).

The problem was this: how can a 20-bit address space be referred to by the 16-bit registers? To solve this problem, the engineers of Intel came up with segment registers CS (Code Segment), DS (Data Segment), ES (Extra Segment), and SS (Stack Segment). To convert from 20-bit address, one would first divide it by 16 and place the quotient in the segment register and remainder in the offset register. This was represented as CS:IP (this means, CS is the segment and IP is the offset). Likewise, when an address is written SS:SP it means SS is the segment and SP is the offset.

This works also the reversed way. If one was, instead of convert from, to create a 20 bit address, it would be done by taking the 16-bit value of a segment register and put it on the address bus, but shifted 4 times to the left (thus effectively multiplying the register by 16), and then by adding the offset from another register untouched to the value on the bus, thus creating a full a 20-bit address.

Example

If CS = 258C and IP = 0012₁₆, then CS:IP will point to a 20 bit address equivalent to "CS × 16 + IP" which will be

$$258C \times 10_{16} + 0012_{16} = 258C0 + 0012_{16} = 258D2 \text{ (Remember: } 16 \text{ decimal} = 10_{16}\text{)}.$$

The 20-bit address is known as an absolute (or linear) address and the Segment:Offset representation (CS:IP) is known as a segmented address. This separation was necessary, as the register itself could not hold values that required more than 16 bits encoding. When programming in protected mode on a 32-bit or 64-bit processor, the registers are big enough to fill the address bus entirely, thus eliminating segmented addresses - only linear/logical addresses are generally used in this "flat addressing" mode, although the *Segment:Offset* architecture is still supported for backwards compatibility.

It is important to note that there is not a one-to-one mapping of physical addresses to segmented addresses; for any physical address, there is more than one possible segmented address. For example: consider the segmented representations B000:8000 and B200:6000. Evaluated, they both map to physical address B8000.

$$B000:8000 = B000 \times 10_{16} + 8000_{16} = B0000 + 8000_{16} = B8000, \text{ and}$$

$$B200:6000 = B200 \times 10_{16} + 6000_{16} = B2000 + 6000_{16} = B8000.$$

However, using an appropriate mapping scheme avoids this problem: such a map applies a linear transformation to the physical addresses to create precisely one segmented address for each. To reverse the translation, the map $[f(x)]$ is simply inverted.

For example, if the segment portion is equal to the physical address divided by 10₁₆ and the offset is equal to the remainder, only one segmented address will be generated. (No offset will be greater than 0F₁₆.) Physical address B8000 maps to (B8000 / 10₁₆):(B8000 mod 10₁₆) or B800:0. This segmented representation is given a special name: such addresses are said to be "normalized Addresses".

CS:IP (Code Segment: Instruction Pointer) represents the 20-bit address of the physical memory from where the next instruction for execution will be picked up. Likewise, SS:SP (Stack Segment: Stack Pointer) points to a 20-bit absolute address which will be treated as stack top (8086 uses this for pushing/popping values).

Protected Mode (80286+)

As ugly as this may seem, it was in fact a step towards the protected addressing scheme used in later chips. The 80286 had a protected mode of operation, in which all 24 of its address lines were available, allowing for addressing of up to 16 MiB of memory. In protected mode, the CS, DS, ES, and SS registers were not segments but selectors, pointing into a table that provided information about the blocks of physical memory that the program was then using. In this mode, the pointer value CS:IP = 0010:2400 is used as follows:

The CS value 0010₁₆ is an offset into the selector table, pointing at a specific selector. This selector would have a 24-bit value to indicate the start of a memory block, a 16-bit value to indicate how long the block is, and flags to specify whether the block can be written, whether it is currently physically in memory, and other information. Let's say that the memory block pointed to actually starts at the 24-bit address 164400₁₆, the actual address referred to then is 164400₁₆ + 2400₁₆ = 166800₁₆. If the selector also includes information that the block is 2400₁₆ bytes long, the reference would be to the byte immediately following that block, which would cause an exception: the operating system should not allow a program to read memory that it does not own. And if the block is marked as read-only, which code segment memory should be so that programs don't overwrite themselves, an attempt to write to that address would similarly cause an exception.

With CS and IP being expanded to 32 bits in the 386, this scheme became unnecessary; with a selector pointing at physical address 00000000₁₆, a 32-bit register could address up to 4 GiB of memory. However, selectors are still used to protect memory from rogue programs. If a program in Windows tries to read or write memory that it doesn't own, for instance, it will violate the rules set by the selectors, triggering an exception, and Windows will shut it down with the "General protection fault" message.

32-Bit Addressing

32-bit addresses can cover memory up to 4 GiB in size. This means that we don't need to use offset addresses in 32-bit processors. Instead, we use what is called the "Flat addressing" scheme, where the address in the register directly points to a physical memory location. The segment registers are used to define different segments, so that programs don't try to execute the stack section, and they don't try to perform stack operations on the data section accidentally.

The A20 Gate Saga

As was said earlier, the 8086 processor had 20 address lines (from A0 to A19), so the total memory addressable by it was 1 MiB (or 2 to the power 20). But since it had only 16 bit registers, they came up with *Segment:Offset* scheme or else using a single 16-bit register they couldn't have possibly accessed more than 64 KiB (or 2 to the power 16) of memory. So this made it possible for a program to access the whole of 1 MiB of memory.

But with this segmentation scheme also came a side effect. Not only could your code refer to the whole of 1 MiB with this scheme, but actually a little more than that. Let's see how

Let's keep in mind how we convert from a *Segment:Offset* representation to Linear 20 bit representation.

The conversion:

$$\text{Segment:Offset} = \text{Segment} \times 16 + \text{Offset}.$$

Now to see the maximum amount of memory that can be addressed, let's fill in both *Segment* and *Offset* to their maximum values and then convert that value to its 20-bit absolute physical address.

So, max value for *Segment* = FFFF_{16} , and max value for *Offset* = FFFF_{16} .

Now, let's convert FFFF:FFFF into its 20-bit linear address, bearing in mind 16_{10} is represented as 10 in hexadecimal.

So we get, $\text{FFFF:FFFF} \rightarrow \text{FFFF} \times 10_{16} + \text{FFFF} = \text{FFFF0}$ (1 MiB - 16 bytes) + FFFF (64 KiB) = $\text{FFFFFF} + \text{FFF0} = 1 \text{ MiB} + \text{FFF0 bytes}$.

- Note: **FFFFFF is hexadecimal and is equal to 1 MiB and FFF0 is equal to 64 KiB minus 16 bytes.**

Moral of the story: From Real mode a program can actually refer to (1 MiB + 64 KiB - 16) bytes of memory.

Notice the use of the word "refer" and not "access". A program can refer to this much memory but whether it can access it or not is dependent on the number of address lines actually present. So with the 8086 this was definitely not possible because when programs made references to 1 MiB plus memory, the address that was put on the address lines was actually more than 20-bits, and this resulted in wrapping around of the addresses.

For example, if a code is referring to 1 MiB, this will get wrapped around and point to location 0 in memory, likewise 1 MiB + 1 will wrap around to address 1 (or 0000:0001).

Now there were some super funky programmers around that time who manipulated this feature in their code, that the addresses get wrapped around and made their code a little faster and a few bytes shorter. Using this technique it was possible for them to access 32 KiB of top memory area (that is 32 KiB touching 1 MiB boundary) and 32 KiB memory of the bottom memory area, without actually reloading their segment registers!

Simple maths you see, if in *Segment:Offset* representation you make *Segment* constant, then since *Offset* is a 16-bit value therefore you can roam around in a 64 KiB (or 2 to the power 16) area of memory. Now if you make your segment register point to 32 KiB below the 1 MiB mark you can access 32 KiB upwards to touch 1 MiB boundary and then 32 KiB further which will ultimately get wrapped to the bottom most 32 KiB.

Now these super funky programmers overlooked the fact that processors with more address lines would be created. (Note: Bill Gates has been attributed with saying, "Who would need more than 640 KB memory?", and these programmers were probably thinking similarly.) In 1982, just 2 years after 8086, Intel released the 80286 processor with 24 address lines. Though it was theoretically backward compatible with legacy 8086 programs, since it also supported Real Mode, many 8086 programs did not function correctly because they depended on out-of-bounds addresses getting wrapped around to lower memory segments. So for the sake of compatibility IBM engineers routed the A20 address line (8086 had lines A0 -

A19) through the Keyboard controller and provided a mechanism to enable/disable the A20 compatibility mode. Now if you are wondering why the keyboard controller, the answer is that it had an unused pin. Since the 80286 would have been marketed as having complete compatibility with the 8086 (that wasn't even yet out very long), upgraded customers would be furious if the 80286 was not bug-for-bug compatible such that code designed for the 8086 would operate just as well on the 80286, but faster.

X86 Instructions

These pages will discuss, in detail, the different instructions available in the basic x86 instruction set. For ease, and to decrease the page size, the different instructions will be broken up into groups, and discussed individually.

- [Data Transfer Instructions](#)
- [Control Flow Instructions](#)
- [Arithmetic Instructions](#)
- [Logic Instructions](#)
- [Shift and Rotate Instructions](#)
- [Other Instructions](#)
- [x86 Interrupts](#)

For more info, see the [resources](#) section.

Conventions

The following template will be used for instructions that take no operands:

Instr

The following template will be used for instructions that take 1 operand:

Instr arg

The following template will be used for instructions that take 2 operands. Notice how the format of the instruction is different for different assemblers.

Instr src, dest

[GAS Syntax](#)

Instr dest, src

[Intel Syntax](#)

The following template will be used for instructions that take 3 operands. Notice how the format of the instruction is different for different assemblers.

Instr aux, src, dest

[GAS Syntax](#)

Instr dest, src, aux

[Intel Syntax](#)

Suffixes

Some instructions, especially when built for non-Windows platforms (i.e. Unix, Linux, etc.), require the use of suffixes to specify the size of the data which will be the subject of the operation. Some possible suffixes are:

- b (byte) = 8 bits.
- w (word) = 16 bits.
- l (long) = 32 bits.
- q (quad) = 64 bits.

An example of the usage with the [mov](#) instruction on a 32-bit architecture, GAS syntax:

```
movl $0x000F, %eax # Store the value F into the eax register
```

On Intel Syntax you don't have to use the suffix. Based on the register name and the used immediate value the compiler knows which data size to use.

```
MOV EAX, 0x000F
```

Data Transfer

Some of the most important and most frequently used instructions are those that move data. Without them, there would be no way for registers or memory to even have anything in them to operate on.

Data transfer instructions

Move

mov src, dest

GAS Syntax

mov dest, src

Intel Syntax

Move

The `mov` instruction copies the `src` operand into the `dest` operand.

Operands

src

- Immediate
- Register
- Memory

dest

- Register
- Memory

Modified flags

- No FLAGS are modified by this instruction

Example

```
.data
value:
    .long    2

.text
.globl _start
_start:
    movl    $6, %eax
    # %eax is now 6

    movw    %eax, value
    # value is now 6

    movl    $0, %ebx
    # %ebx is now 0

    movb    %al, %bl
    # %ebx is now 6

    movl    value, %ebx
    # %ebx is now 6
```

```

movl    $value, %esi
# %esi is now the address of value

xorl    %ebx, %ebx
# %ebx is now 0

movw    value(, %ebx, 1), %bx
# %ebx is now 6

# Linux sys_exit
movl    $1, %eax
xorl    %ebx, %ebx
int     $0x80

```

Data swap

xchg src, dest

GAS Syntax

xchg dest, src

Intel Syntax

Exchange.

The `xchg` instruction swaps the `src` operand with the `dest` operand. It's like doing three move operations: from *dest* to a temporary (another register), then from *src* to *dest*, then from the temporary to *src*, except that no register needs to be reserved for temporary storage.

If one of the operands is a memory address, then the operation has an implicit `LOCK` prefix, that is, the exchange operation is atomic. This can have a large performance penalty.

It's also worth noting that the common NOP (no op) instruction, `0x90`, is the opcode for `xchgl %eax, %eax`.

Operands

src

- Register
- Memory

dest

- Register
- Memory

However, only one operand can be in memory: the other must be a register.

Modified flags

- No FLAGS are modified by this instruction

Example

```

.data
value:  .long    2

.text
.global _start
_start:
movl    $54, %ebx
xorl    %eax, %eax

xchgl   value, %ebx
# %ebx is now 2
# value is now 54

xchgw   %ax, value
# Value is now 0
# %eax is now 54

xchgb   %al, %bl
# %ebx is now 54

```

```

# %eax is now 2

xchgw    value(%eax), %ax
# value is now 0x00020000 = 131072
# %eax is now 0

# Linux sys_exit
mov     $1, %eax
xorl    %ebx, %ebx
int     $0x80

```

cmpxchg arg2, arg1
cmpxchg arg1, arg2

GAS Syntax

Intel Syntax

Compare and exchange.

The `cmpxchg` instruction has two implicit operands `AL/AX/EAX`(depending on the size of `arg1`) and `ZF`(zero) flag. The instruction compares `arg1` to `AL/AX/EAX` and if they are equal sets `arg1` to `arg2` and sets the zero flag, otherwise it sets `AL/AX/EAX` to `arg1` and clears the zero flag. Unlike `xchg` there is not an implicit `lock` prefix and if the instruction is required to be atomic then `lock` must be prefixed.

Operands

arg1

- Register
- Memory

arg2

- Register

Modified flags

- The `ZF` flag is modified by this instruction

Example

The following example shows how to use the `cmpxchg` instruction to create a spin lock which will be used to protect the `result` variable. The last thread to grab the spin lock will get to set the final value of `result`:

```

global main

extern printf
extern pthread_create
extern pthread_exit
extern pthread_join

section .data
    align 4
    sLock:    dd 0      ; The lock, values are:
                    ; 0 unlocked
                    ; 1 locked
    tID1:     dd 0
    tID2:     dd 0
    fmtStr1:  db "In thread %d with ID: %02x", 0x0A, 0
    fmtStr2:  db "Result %d", 0x0A, 0

section .bss
    align 4
    result:   resd 1

section .text
    main:     ; Using main since we are using gcc to link

                ;
                ; Call pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                ;               void *(*start_routine) (void *), void *arg);
                ;
    push     dword 0      ; Arg Four: argument pointer
    push     thread1     ; Arg Three: Address of routine
    push     dword 0      ; Arg Two: Attributes
    push     tID1        ; Arg One: pointer to the thread ID
    call     pthread_create

```

```

push    dword 0      ; Arg Four: argument pointer
push    thread2      ; Arg Three: Address of routine
push    dword 0      ; Arg Two: Attributes
push    tID2         ; Arg One: pointer to the thread ID
call    pthread_create

        ;
        ; Call int pthread_join(pthread_t thread, void **retval) ;
        ;
push    dword 0      ; Arg Two: retval
push    dword [tID1]  ; Arg One: Thread ID to wait on
call    pthread_join
push    dword 0      ; Arg Two: retval
push    dword [tID2]  ; Arg One: Thread ID to wait on
call    pthread_join

push    dword [result]
push    dword fmtStr2
call    printf
add esp, 8           ; Pop stack 2 times 4 bytes

call    exit

thread1:
pause
push    dword [tID1]
push    dword 1
push    dword fmtStr1
call    printf
add esp, 12          ; Pop stack 3 times 4 bytes

call    spinLock

mov [result], dword 1
call    spinUnlock

push    dword 0      ; Arg one: retval
call    pthread_exit

thread2:
pause
push    dword [tID2]
push    dword 2
push    dword fmtStr1
call    printf
add esp, 12          ; Pop stack 3 times 4 bytes

call    spinLock

mov [result], dword 2
call    spinUnlock

push    dword 0      ; Arg one: retval
call    pthread_exit

spinLock:
push    ebp
mov ebp, esp
mov edx, 1           ; Value to set sLock to
spin:  mov eax, [sLock] ; Check sLock
test   eax, eax      ; If it was zero, maybe we have the lock
jnz    spin          ; If not try again
;
; Attempt atomic compare and exchange:
; if (sLock == eax):
;   sLock <- edx
;   zero flag <- 1
; else:
;   eax <- edx
;   zero flag <- 0
;
; If sLock is still zero then it will have the same value as eax and
; sLock will be set to edx which is one and therefore we acquire the
; lock. If the lock was acquired between the first test and the
; cmpxchg then eax will not be zero and we will spin again.
;
lock   cmpxchg [sLock], edx
test   eax, eax
jnz    spin
pop    ebp
ret

spinUnlock:
push    ebp
mov ebp, esp
mov    eax, 0
xchg   eax, [sLock]
pop    ebp
ret

exit:
        ;
        ; Call exit(3) syscall
        ; void exit(int status)
        ;
mov    ebx, 0        ; Arg one: the status

```

```
mov eax, 1      ; Syscall number:
int 0x80
```

In order to assemble, link and run the program we need to do the following:

```
$ nasm -felf32 -g cmpxchgSpinLock.asm
$ gcc -o cmpxchgSpinLock cmpxchgSpinLock.o -lpthread
$ ./cmpxchgSpinLock
```

Move with zero extend

movz src, dest
movzx dest, src

[GAS Syntax](#)

[Intel Syntax](#)

Move zero extend

The **movz** instruction copies the **src** operand in the **dest** operand and pads the remaining bits not provided by **src** with zeros (0).

This instruction is useful for copying a small, unsigned value to a bigger register.

Operands

src

- Register
- Memory

dest

- Register

Modified flags

- No FLAGS are modified by this instruction

Example

```
.data
byteval:
.byte 204

.text
.global _start

_start:
movzbw byteval, %ax
# %eax is now 204

movzwl %ax, %ebx
# %ebx is now 204

movzbl byteval, %esi
# %esi is now 204

# Linux sys_exit
mov $1, %eax
xorl %ebx, %ebx
int $0x80
```

Move with sign extend

movs src, dest
movsx dest, src

[GAS Syntax](#)

[Intel Syntax](#)

Move sign extend.

The `movs` instruction copies the `src` operand in the `dest` operand and pads the remaining bits not provided by `src` with the sign bit (the MSB) of `src`.

This instruction is useful for copying a signed small value to a bigger register.

Operands

src

- Register
- Memory

dest

- Register

Modified flags

- No FLAGS are modified by this instruction

Example

```
.data
byteval:
    .byte    -24 # = 0xe8

.text
    .global _start

_start:
    movsbw byteval, %ax
    # %ax is now -24 = 0xffe8

    movswl %ax, %ebx
    # %ebx is now -24 = 0xfffffe8

    movsbl byteval, %esi
    # %esi is now -24 = 0xffffffe8

# Linux sys_exit
    mov     $1, %eax
    xorl    %ebx, %ebx
    int     $0x80
```

Move String

movsb

Move byte

The `movsb` instruction copies one byte from the memory location specified in `esi` to the location specified in `edi`. If the direction flag is cleared, then `esi` and `edi` are incremented after the operation. Otherwise, if the direction flag is set, then the pointers are decremented. In that case the copy would happen in the reverse direction, starting at the highest address and moving toward lower addresses until `ecx` is zero.

Operands

None.

Modified flags

- No FLAGS are modified by this instruction

Example

```
section .text
; copy mystr into mystr2
mov esi, mystr    ; loads address of mystr into esi
mov edi, mystr2   ; loads address of mystr2 into edi
```

```

    cld                ; clear direction flag (forward)
    mov ecx,6
    rep movsb         ; copy six times

section .bss
    mystr2: resb 6

section .data
    mystr db "Hello", 0x0

```

movsw

Move word

The `movsw` instruction copies one word (two bytes) from the location specified in `esi` to the location specified in `edi`. It basically does the same thing as `movsb`, except with words instead of bytes.

Operands

None.

Modified flags

- No FLAGS are modified by this instruction

Example

```

section .code
    ; copy mystr into mystr2
    mov esi, mystr
    mov edi, mystr2
    cld
    mov ecx,4
    rep movsw
    ; mystr2 is now AaBbCca\0

section .bss
    mystr2: resb 8

section .data
    mystr db "AaBbCca", 0x0

```

Load Effective Address

lea src, dest
lea dest, src

[GAS Syntax](#)

[Intel Syntax](#)

Load Effective Address

The `lea` instruction calculates the address of the `src` operand and loads it into the `dest` operand.

Operands

src

- Immediate
- Register
- Memory

dest

- Register

Modified flags

- No FLAGS are modified by this instruction

Note Load Effective Address calculates its SRC operand in the same way as the MOV instruction does, but rather than loading the *contents* of that address into the dest operand, it loads the address itself.

LEA can be used not only for calculating addresses, but also general-purpose unsigned integer arithmetic (with the caveat and possible advantage that FLAGS are unmodified). This can be quite powerful, since the SRC operand can take up to 4 parameters: base register, index register, scalar multiplier and displacement, e.g. [eax + edx*4 - 4] (Intel syntax) or -4(%eax, %edx, 4) (GAS syntax). The scalar multiplier is limited to constant values 1, 2, 4, or 8 for byte, word, double word or quad word offsets respectively. This by itself allows for multiplication of a general register by constant values 2, 3, 4, 5, 8 and 9, as shown below (using NASM syntax):

```
lea ebx, [ebx*2]      ; Multiply ebx by 2
lea ebx, [ebx*8+ebx]  ; Multiply ebx by 9, which totals ebx*18
```

Data transfer instructions of 8086 microprocessor

General purpose byte or word transfer instructions:

- **MOV:** copy byte or word from specified source to specified destination
- **PUSH:** copy specified word to top of stack.
- **POP:** copy word from top of stack to specified location
- **PUSHA:** copy all registers to stack
- **POPA:** copy words from stack to all registers.
- **XCHG:** Exchange bytes or exchange words
- **XLAT:** translate a byte in AL using a table in memory.

These are I/O port transfer instructions:

- **IN:** copy a byte or word from specific port to accumulator
- **OUT:** copy a byte or word from accumulator to specific port

Special address transfer Instructions:

- **LEA:** load effective address of operand into specified register
- **LDS:** load DS register and other specified register from memory
- **LES:** load ES register and other specified register from memory

Flag transfer instructions:

- **LAHF:** load AH with the low byte of flag register
- **SAHF:** Stores AH register to low byte of flag register
- **PUSHF:** copy flag register to top of stack
- **POPF:** copy top of stack word to flag register

Control Flow

Almost all programming languages have the ability to change the order in which statements are evaluated, and assembly is no exception. The instruction pointer (EIP) register contains the address of the next instruction to be executed. To change the flow of control, the programmer must be able to modify the value of EIP. This is where control flow functions come in.

```
mov eip, label  ; wrong
jmp label      ; right
```

Comparison Instructions

test arg0, arg1
test arg1, arg0

[GAS Syntax](#)

[Intel Syntax](#)

Performs a bit-wise logical and on `arg0` and `arg1` the result of which we will refer to as `commonBits` and sets the ZF(zero), SF(sign) and PF (parity) flags based on `commonBits`. `CommonBits` is then discarded.

Operands

arg0

- Register
- Immediate

arg1

- AL/AX/EAX (only if `arg0` is an immediate value)
- Register
- Memory

Modified flags

- `SF` := `MostSignificantBit(commonBits)`
- `ZF` := `(commonBits = 0)`, so a set `ZF` means, `arg0` and `arg1` do not have any set bits in common
- `PF` := `BitWiseXorNor(commonBits[Max-1:0])`, so `PF` is set if and only if `commonBits[Max-1:0]` has an even number of 1 bits
- `CF` := 0
- `OF` := 0
- `AF` is undefined

Application

- passing the same register twice: `test rax, rax`
 - `SF` becomes the *sign* of `rax`, which is a simple test for non-negativity
 - `ZF` is set if `rax` is *zero*
 - `PF` is set if `rax` has an even number of set bits

cmp subtrahend, minuend
cmp minuend, subtrahend

[GAS Syntax](#)

[Intel Syntax](#)

Performs a comparison operation between `minuend` and `subtrahend`. The comparison is performed by a (signed) subtraction of `subtrahend` from `minuend`, the results of which can be called `difference`. `Difference` is then discarded. If `subtrahend` is an immediate value it will be sign extended to the length of `minuend`. The `EFLAGS` register is set in the same manner as a `Sub` instruction.

Note that the GAS/AT&T syntax can be rather confusing, as for example `cmp $0, %rax` followed by `j1` branch will branch if `%rax < 0` (and not the opposite as might be expected from the order of the operands).

Operands

minuend

- AL/AX/EAX (only if subtrahend is immediate)
- Register
- Memory

subtrahend

- Register
- Immediate
- Memory

Modified flags

- SF := MostSignificantBit(difference), so a unset SF means the difference is non-negative (minuend $\geq$ subtrahend [NB: signed comparison])
- ZF := (difference = 0)
- PF := BitWiseXorNor(difference[Max-1:0])
- CF, OF and AF

Jump Instructions

The jump instructions allow the programmer to (indirectly) set the value of the EIP register. The location passed as the argument is usually a label. The first instruction executed after the jump is the instruction immediately following the label. All of the jump instructions, with the exception of `jmp`, are **conditional jumps**, meaning that program flow is diverted only if a condition is true. These instructions are often used after a comparison instruction (see above), but since many other instructions set flags, this order is not required.

See [chapter “X86 architecture”](#), § “EFLAGS register” for more information about the flags and their meaning.

Unconditional Jumps

jmp loc

Loads EIP with the specified address (i.e. the next instruction executed will be the one specified by `jmp`).

Jump if Equal

je loc

$ZF = 1$

Loads EIP with the specified address, if operands of previous `cmp` instruction are equal. For example:

```
mov ecx, $5
mov edx, $5
cmp ecx, edx
je equal
; if it did not jump to the label equal,
; then this means ecx and edx are not equal.
equal:
; if it jumped here, then this means ecx and edx are equal
```

Jump if Not Equal

jne loc

$ZF = 0$

Loads EIP with the specified address, if operands of previous `cmp` instruction are not equal.

Jump if Greater

jg loc

$SF = OF \text{ and } ZF = 0$

Loads EIP with the specified address, if the minuend of the previous `cmp` instruction is greater than the second (performs signed comparison).

Jump if Greater or Equal

jge loc

$SF = OF \text{ or } ZF = 1$

Loads EIP with the specified address, if the minuend of the of previous `cmp` instruction is greater than or equal to the subtrahend (performs signed comparison).

Jump if Above (unsigned comparison)

ja loc

$CF = 0 \text{ and } ZF = 0$

Loads EIP with the specified address, if the minuend of the previous `cmp` instruction is greater than the subtrahend. `ja` is the same as `jg`, except that it performs an unsigned comparison.

That means, the following piece of code *always* jumps (unless `rbx` is `-1`, too), because negative one is represented as all bits set in the two's complement.

```
mov rax, -1 // rax := -1
cmp rax, rbx
ja loc
```

Interpreting all bits set (without treating any bit as the sign) has the value $2^n - 1$ (where n is the length of the register). That is the largest unsigned value a register can hold.

Jump if Above or Equal (unsigned comparison)

jae loc

$CF = 0 \text{ or } ZF = 1$

Loads EIP with the specified address, if the minuend of previous `cmp` instruction is greater than or equal to the subtrahend. `jae` is the same as `jge`, except that it performs an unsigned comparison.

Jump if Lesser

jl loc

The criterion required for a `jl` is that $SF \neq OF$. It loads EIP with the specified address, if the criterion is met. So either `SF` or `OF` can be set, but not both in order to satisfy this criterion. If we take the `sub` (which is *basically* what a `cmp` does) instruction as an example, we have:

minuend - subtrahend

With respect to `sub` and `cmp` there are several cases that fulfill this criterion:

1. `minuend < subtrahend` and the operation does not have overflow
2. `minuend > subtrahend` and the operation has an overflow

In the first case `SF` will be set but not `OF` and in second case `OF` will be set but not `SF` since the overflow will reset the most significant bit to zero and thus preventing `SF` being set. The `SF  $\neq$  OF` criterion avoids the cases where:

1. `minuend > subtrahend` and the operation does not have overflow
2. `minuend < subtrahend` and the operation has an overflow
3. `minuend = subtrahend`

In the first case neither `SF` nor `OF` are set, in the second case `OF` will be set and `SF` will be set since the overflow will reset the most significant bit to one and in the last case neither `SF` nor `OF` will be set.

Jump if Less or Equal

jle loc

`SF  $\neq$  OF` or `ZF = 1`.

Loads EIP with the specified address, if the `minuend` of previous `cmp` instruction is lesser than or equal to the `subtrahend`. See the [j1](#) section for a more detailed description of the criteria.

Jump if Below (unsigned comparison)

jb loc

`CF = 1`

Loads EIP with the specified address, if first operand of previous `CMP` instruction is lesser than the second. `jb` is the same as `j1`, except that it performs an unsigned comparison.

```
mov rax, 0 ; rax = 0
cmp rax, rbx ; rax  $\neq$  rbx
jb loc ; always jumps, unless rbx is also 0
```

Jump if Below or Equal (unsigned comparison)

jbe loc

`CF = 1` or `ZF = 1`

Loads EIP with the specified address, if `minuend` of previous `cmp` instruction is lesser than or equal to the `subtrahend`. `jbe` is the same as `jle`, except that it performs an unsigned comparison.

Jump if Overflow

jo loc

`OF = 1`

Loads EIP with the specified address, if the overflow bit is set on a previous arithmetic expression.

Jump if Not Overflow

jno loc

$OF = 0$

Loads EIP with the specified address, if the overflow bit is not set on a previous arithmetic expression.

Jump if Zero

jz loc

$ZF = 1$

Loads EIP with the specified address, if the zero bit is set from a previous arithmetic expression. **jz** is identical to **je**.

Jump if Not Zero

jnz loc

$ZF = 0$

Loads EIP with the specified address, if the zero bit is not set from a previous arithmetic expression. **jnz** is identical to **jne**.

Jump if Signed

js loc

$SF = 1$

Loads EIP with the specified address, if the sign bit is set from a previous arithmetic expression.

Jump if Not Signed

jns loc

$SF = 0$

Loads EIP with the specified address, if the sign bit is not set from a previous arithmetic expression.

Jump if counter register is zero

jcxz loc

$CX = 0$

jecxz loc

$ECX = 0$

jrcxz loc

$RCX = 0$

Loads EIP with the specified address, if the counter register is zero.

Function Calls

call proc

Pushes the address of the instruction that follows the `call` call, i.e. usually the next line in your source code, onto the top of the stack, and then jumps to the specified location. This is used mostly for subroutines.

ret [val]

Loads the next value on the stack into EIP, and then pops the specified number of bytes off the stack. If `val` is not supplied, the instruction will not pop any values off the stack after returning.

Loop Instructions

loop arg

The `loop` instruction decrements ECX and jumps to the address specified by `arg` unless decrementing ECX caused its value to become zero. For example:

```
mov ecx, 5 ; ecx = 5
head:
; the code here would be executed 5 times
loop head
```

`loop` does not set any flags.

loopcc arg

These loop instructions decrement ECX and jump to the address specified by `arg` if their condition is satisfied (that is, a specific flag is set), unless decrementing ECX caused its value to become zero.

- `loope` loop if equal
- `loopne` loop if not equal
- `loopnz` loop if not zero
- `loopz` loop if zero

That way, only testing for a non-zero ECX can be combined with testing ZF. Other flags can not be tested for, say there is no `loopnc` “loop while ECX $\neq$ 0 and CF unset”.

Enter and Leave

enter arg

`enter` creates a stack frame with the specified amount of space allocated on the stack.

leave

`leave` destroys the current stack frame, and restores the previous frame. Using Intel syntax this is equivalent to:

```
mov esp, ebp ; esp = ebp
pop ebp
```

This will set EBP and ESP to their respective value before the function prologue began therefore reversing any modification to the stack that took place during the prologue.

Other Control Instructions

hlt

Halts the processor. Execution will be resumed after processing next hardware interrupt, unless IF is cleared.

nop

No operation. This instruction doesn't do anything, but wastes (an) instruction cycle(s) in the processor.

This instruction is often *represented* as an `xchg` operation with the operands EAX and EAX (an operation without side-effects), because there is no designated opcode for doing nothing. This just as a passing remark, so that you do not get confused with *disassembled* code.

lock

Asserts `#LOCK` prefix on next instruction.

wait

Waits for the FPU to finish its last calculation.

Arithmetic

Arithmetic instructions

Arithmetic instructions take two operands: a destination and a source. The destination must be a register or a memory location. The source may be either a memory location, a register, or a constant value. Note that at least one of the two must be a register, because operations may not use a memory location as both a source and a destination.

add src, dest
add dest, src

[GAS Syntax](#)

[Intel Syntax](#)

This adds `src` to `dest`. If you are using the MASM syntax, then the result is stored in the first argument, if you are using the GAS syntax, it is stored in the second argument.

sub src, dest
sub dest, src

[GAS Syntax](#)

[Intel Syntax](#)

Like ADD, only it subtracts source from destination instead. In C: `dest -= src;`

mul arg

This multiplies `arg` by the value of corresponding byte-length in the AX register.

result registers used by mul

operand size	1 byte	2 bytes	4 bytes
other operand	AL	AX	EAX
higher part of result stored in	AH	DX	EDX
lower part of result stored in	AL	AX	EAX

In the second case, the target is not EAX for backward compatibility with code written for older processors.

imul arg

As mul, only signed. The imul instruction has the same format as mul, but also accepts two other formats like so:

imul src, dest

[GAS Syntax](#)

imul dest, src

[Intel Syntax](#)

This multiplies src by dest. If you are using the NASM syntax, then the result is stored in the first argument, if you are using the GAS syntax, it is stored in the second argument.

imul aux, src, dest

[GAS Syntax](#)

imul dest, src, aux

[Intel Syntax](#)

This multiplies src by aux and places it into dest. If you are using the NASM syntax, then the result is stored in the first argument, if you are using the GAS syntax, it is stored in the third argument.

div arg

This divides the value in the dividend register(s) by arg, see table below.

result registers for div

divisor size	1 byte	2 bytes	4 bytes
dividend	AX	DX:AX	EDX:EAX
remainder stored in	AH	DX	EDX
quotient stored in	AL	AX	EAX

The colon (:) means concatenation. With divisor size 4, this means that EDX are the bits 32-63 and EAX are bits 0-31 of the input number (with lower bit numbers being less significant, in this example).

As you typically have 32-bit input values for division, you often **need to use CDQ to sign-extend EAX into EDX just before the division.**

If quotient does not fit into quotient register, arithmetic overflow interrupt occurs. All flags are in undefined state after the operation.

idiv arg

As `div`, only signed.

neg arg

Arithmetically negates the argument (i.e. two's complement negation).

Carry Arithmetic Instructions

adc src, dest

[GAS Syntax](#)

adc dest, src

[Intel Syntax](#)

Add with carry. Adds `src` + `CF` to `dest`, storing result in `dest`. Usually follows a normal add instruction to deal with values twice as large as the size of the register. In the following example, `source` contains a 64-bit number which will be added to `destination`.

```
mov eax, [source] ; read low 32 bits
mov edx, [source+4] ; read high 32 bits
add [destination], eax ; add low 32 bits
adc [destination+4], edx ; add high 32 bits, plus carry
```

sbb src, dest

[GAS Syntax](#)

sbb dest, src

[Intel Syntax](#)

Subtract with borrow. Subtracts `src` + `CF` from `dest`, storing result in `dest`. Usually follows a normal sub instruction to deal with values twice as large as the size of the register.

Increment and Decrement

inc arg

Increments the register value in the argument by 1. Performs much faster than `add arg, 1`.

dec arg

Decrements the register value in the argument by 1. Performs much faster than `sub arg, 1`.

Pointer arithmetic

The `lea` instruction can be used for arithmetic, especially on pointers. See [chapter “data transfer”, § “load effective address”](#).

Logic

Logical instructions

The instructions on this page deal with bit-wise logical instructions. For more information about bit-wise logic, see [Digital Circuits/Logic Operations](#).

and src, dest
and dest, src

[GAS Syntax](#)

[Intel Syntax](#)

Performs a bit-wise **and** of the two operands, and stores the result in **dest**. For example:

```
movl $0x1, %edx ; edx = 1
movl $0x0, %ecx ; ecx = 0
andl %edx, %ecx ; ecx = edx & ecx
; here ecx would be 0 because 1 & 0 ⇨ 0
```

or src, dest
or dest, src

[GAS Syntax](#)

[Intel Syntax](#)

Performs a bit-wise **or** of the two operands, and stores the result in **dest**. For example:

```
movl $0x1, %edx ; edx = 1
movl $0x0, %ecx ; ecx = 0
orl %edx, %ecx ; ecx = edx | ecx
; here ecx would be 1 because 1 | 0 ⇨ 1
```

xor src, dest
xor dest, src

[GAS Syntax](#)

[Intel Syntax](#)

Performs a bit-wise **xor** of the two operands, and stores the result in **dest**. For example:

```
movl $0x1, %edx ; edx = 1
movl $0x0, %ecx ; ecx = 0
xorl %edx, %ecx ; ecx = edx ^ ecx
; here ecx would be 1 because 1 ^ 0 ⇨ 1
```

not arg

Performs a bit-wise inversion of **arg**. For example:

```
movl $0x1, %edx ; edx = 1
notl %edx ; edx = ~edx
; here edx would be 0xFFFFFFFE because a bitwise NOT 0x00000001 = 0xFFFFFFFE
```

Shift and Rotate

Logical Shift Instructions

In a **logical shift** instruction (also referred to as **unsigned shift**), the bits that slide off the end disappear (except for the last, which goes into the carry flag), and the spaces are always filled with zeros. Logical shifts are best used with unsigned numbers.

shr src, dest
shr dest, src

[GAS Syntax](#)

[Intel Syntax](#)

Logical shift **dest** to the right by **src** bits.

shl src, dest
shl dest, src

[GAS Syntax](#)

[Intel Syntax](#)

Logical shift **dest** to the left by **src** bits.

Examples ([GAS Syntax](#)):

```
movw    $ff00,%ax    # ax=1111.1111.0000.0000 (0xff00, unsigned 65280, signed -256)
shrw    $3,%ax       # ax=0001.1111.1110.0000 (0x1fe0, signed and unsigned 8160)
           # (logical shifting unsigned numbers right by 3
           #  is like integer division by 8)
shlw    $1,%ax       # ax=0011.1111.1100.0000 (0x3fc0, signed and unsigned 16320)
           # (logical shifting unsigned numbers left by 1
           #  is like multiplication by 2)
```

Arithmetic Shift Instructions

In an **arithmetic shift** (also referred to as **signed shift**), like a logical shift, the bits that slide off the end disappear (except for the last, which goes into the carry flag). But in an arithmetic shift, the spaces are filled in such a way to preserve the sign of the number being slid. For this reason, arithmetic shifts are better suited for signed numbers in two's complement format.

sar src, dest
sar dest, src

[GAS Syntax](#)

[Intel Syntax](#)

Arithmetic shift **dest** to the right by **src** bits. Spaces are filled with sign bit (to maintain sign of original value), which is the original highest bit.

sal src, dest
sal dest, src

[GAS Syntax](#)

[Intel Syntax](#)

Arithmetic shift **dest** to the left by **src** bits. The bottom bits do not affect the sign, so the bottom bits are filled with zeros. This instruction is synonymous with **SHL**.

Examples ([GAS Syntax](#)):

```
movw    $ff00,%ax    # ax=1111.1111.0000.0000 (0xff00, unsigned 65280, signed -256)
salw    $2,%ax       # ax=1111.1100.0000.0000 (0xfc00, unsigned 64512, signed -1024)
           # (arithmetic shifting left by 2 is like multiplication by 4 for
           #  negative numbers, but has an impact on positives with most
           #  significant bit set (i.e. set bits shifted out))
sarw    $5,%ax       # ax=1111.1111.1110.0000 (0xffe0, unsigned 65504, signed -32)
           # (arithmetic shifting right by 5 is like integer division by 32
           #  for negative numbers)
```

Extended Shift Instructions

The names of the *double precision* shift operations are somewhat misleading, hence they are listed as *extended* shift instructions on this page.

They are available for use with 16- and 32-bit data entities (registers/memory locations). The `src` operand is always a register, the `dest` operand can be a register or memory location, the `cnt` operand is an immediate byte value or the `CL` register. In 64-bit mode it is possible to address 64-bit data as well.

shld cnt, src, dest

[GAS Syntax](#)

shld dest, src, cnt

[Intel Syntax](#)

The operation performed by `shld` is to shift the most significant `cnt` bits out of `dest`, but instead of filling up the least significant bits with zeros, they are filled with the most significant `cnt` bits of `src`.

shrd cnt, src, dest

[GAS Syntax](#)

shrd dest, src, cnt

[Intel Syntax](#)

Likewise, the `shrd` operation shifts the least significant `cnt` bits out of `dest`, and fills up the most significant `cnt` bits with the least significant bits of the `src` operand.

Intel's nomenclature is misleading, in that the shift does not operate on double the basic operand size (i.e. specifying 32-bit operands doesn't make it a 64-bit shift): the `src` operand always remains unchanged.

Also, Intel's manual^[2] states that the results are undefined when `cnt` is greater than the operand size, but at least for 32- and 64-bit data sizes it has been observed that shift operations are performed by $(cnt \bmod n)$, with n being the data size.

Examples ([GAS Syntax](#)):

```
xorw  %ax,%ax      # ax=0000.0000.0000.0000 (0x0000)
notw  %ax           # ax=1111.1111.1111.1111 (0xffff)
movw  $0x5500,%bx   # bx=0101.0101.0000.0000
shrdw $4,%ax,%bx    # bx=1111.0101.0101.0000 (0xf550), ax is still 0xffff
shldw $8,%bx,%ax    # ax=1111.1111.1111.0101 (0xffff5), bx is still 0xf550
```

Other examples (**decimal numbers are used instead of binary number to explain the concept**)

```
# ax = 1234 5678
# bx = 8765 4321
shrd  $3, %ax, %bx    # ax = 1234 5678 bx = 6788 7654
```

```
# ax = 1234 5678
# bx = 8765 4321
shld  $3, %ax, %bx    # bx = 5432 1123 ax = 1234 5678
```

Rotate Instructions

In a rotate instruction, the bits that slide off the end of the register are fed back into the spaces.

ror src, dest

[GAS Syntax](#)

ror dest, src

[Intel Syntax](#)

Rotate `dest` to the right by `src` bits.

rol src, dest
rol dest, src

[GAS Syntax](#)

[Intel Syntax](#)

Rotate `dest` to the left by `src` bits.

Rotate With Carry Instructions

Like with shifts, the rotate can use the carry bit as the "extra" bit that it shifts through.

rcr src, dest
rcr dest, src

[GAS Syntax](#)

[Intel Syntax](#)

Rotate `dest` to the right by `src` bits with carry.

rcl src, dest
rcl dest, src

[GAS Syntax](#)

[Intel Syntax](#)

Rotate `dest` to the left by `src` bits with carry.

Number of arguments

Unless stated, these instructions can take either one or two arguments. If only one is supplied, it is assumed to be a register or memory location and the number of bits to shift/rotate is one (this may be dependent on the assembler in use, however). `shr1 $1, %eax` is equivalent to `shr1 %eax` (GAS syntax).

Notes

1. bh
2. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (<http://download.intel.com/products/processor/manual/325383.pdf>) (PDF, 6.2 MB)

Other Instructions

Stack Instructions

push arg

This instruction decrements the stack pointer and stores the data specified as the argument into the location pointed to by the stack pointer.

pop arg

This instruction loads the data stored in the location pointed to by the stack pointer into the argument specified and then increments the stack pointer. For example:

.....

```
mov eax, 5  
mov ebx, 6
```

```
push eax
```

The stack is now: [5]

```
push ebx
```

The stack is now: [6] [5]

```
pop eax
```

The topmost item (which is 6) is now stored in eax. The stack is now: [5]

```
pop ebx
```

ebx is now equal to 5. The stack is now empty.

pushf

This instruction decrements the stack pointer and then loads the location pointed to by the stack pointer with the contents of the flag register.

popf

This instruction loads the flag register with the contents of the memory location pointed to by the stack pointer and then increments the contents of the stack pointer.

pusha

This instruction pushes all the general purpose registers onto the stack in the following order: AX, CX, DX, BX, SP, BP, SI, DI. The value of SP pushed is the value before the instruction is executed. It is useful for saving state before an operation that could potentially change these registers.

popa

This instruction pops all the general purpose registers off the stack in the reverse order of PUSH.A. That is, DI, SI, BP, SP, BX, DX, CX, AX. Used to restore state after a call to PUSH.A.

pushad

This instruction works similarly to pusha, but pushes the 32-bit general purpose registers onto the stack instead of their 16-bit counterparts.

popad

This instruction works similarly to popa, but pops the 32-bit general purpose registers off of the stack instead of their 16-bit counterparts.

Flags instructions

While the flags register is used to report on results of executed instructions (overflow, carry, etc.), it also contains flags that affect the operation of the processor. These flags are set and cleared with special instructions.

Interrupt Flag

The IF flag tells a processor if it should accept hardware interrupts. It should be kept set under normal execution. In fact, in protected mode, neither of these instructions can be executed by user-level programs.

sti

Sets the interrupt flag. If set, the processor can accept interrupts from peripheral hardware.

cli

Clears the interrupt flag. Hardware interrupts cannot interrupt execution. Programs can still generate interrupts, called software interrupts, and change the flow of execution. Non-maskable interrupts (NMI) cannot be blocked using this instruction.

Direction Flag

The DF flag tells the processor which way to read data when using string instructions. That is, whether to decrement or increment the `esi` and `edi` registers after a `movs` instruction.

std

Sets the direction flag. Registers will decrement, reading backwards.

cld

Clears the direction flag. Registers will increment, reading forwards.

Carry Flag

The CF flag is often modified after arithmetic instructions, but it can be set or cleared manually as well.

stc

Sets the carry flag.

clc

Clears the carry flag.

cmc

Complements (inverts) the carry flag.

Other

sahf

Stores the content of AH register into the lower byte of the flag register.

lahf

Loads the AH register with the contents of the lower byte of the flag register.

I/O Instructions

in src, dest
in dest, src

[GAS Syntax](#)

[Intel Syntax](#)

The **IN** instruction almost always has the operands AX and DX (or EAX and EDX) associated with it. DX (src) frequently holds the port address to read, and AX (dest) receives the data from the port. In Protected Mode operating systems, the IN instruction is frequently locked, and normal users can't use it in their programs.

out src, dest
out dest, src

[GAS Syntax](#)

[Intel Syntax](#)

The **OUT** instruction is very similar to the IN instruction. OUT outputs data from a given register (src) to a given output port (dest). In protected mode, the OUT instruction is frequently locked so normal users can't use it.

System Instructions

These instructions were added with the Pentium II.

sysenter

This instruction causes the processor to enter protected system mode (supervisor mode or "kernel mode").

sysexit

This instruction causes the processor to leave protected system mode, and enter user mode.

Misc Instructions

RDTSC

RDTSC was introduced in the Pentium processor, the instruction reads the number of clock cycles since reset and returns the value in EDX:EAX. This can be used as a way of obtaining a low overhead, high resolution CPU timing. Although with modern CPU microarchitecture(multi-core, hyperthreading) and multi-CPU machines you are not guaranteed synchronized cycle counters between cores and CPUs. Also the CPU frequency may be variable due to power saving or dynamic overclocking. So the instruction may be less reliable than when it was first introduced and should be used with care when being used for performance measurements.

It is possible to use just the lower 32-bits of the result but it should be noted that on a 600 MHz processor the register would overflow every 7.16 seconds:

$$2^{32} \text{cycles} * (1\text{second}/600,000,000\text{cycles}) = 7.16\text{seconds}$$

While using the full 64-bits allows for 974.9 years between overflows:

$$2^{64} \text{cycles} * ((1\text{second}/600,000,000\text{cycles})/(86400\text{seconds in a day} * 365 \text{ days in a year})) = 974.9\text{years}$$

The following program (using [NASM](#) syntax) is an example of using RDTSC to measure the number of cycles a small block takes to execute:

```

global main

extern printf

section .data
    align 4
    a: dd 10.0
    b: dd 5.0
    c: dd 2.0
    fmtStr: db "edx:eax = %llu edx = %d eax = %d", 0x0A, 0

section .bss
    align 4
    cycleLow: resd 1
    cycleHigh: resd 1
    result: resd 1

section .text
    main:                ; Using main since we are using gcc to link

;
; op dst, src
;
xor eax, eax
cpuid
rdtsc
mov [cycleLow], eax
mov [cycleHigh], edx

;
; Do some work before measurements
;
fld dword [a]
fld dword [c]
fmulp st1
fmulp st1
fld dword [b]
fld dword [b]
fmulp st1
faddp st1
fsqrt
fstp dword [result]
;
; Done work
;

cpuid
rdtsc

;
; break points so we can examine the values
; before we alter the data in edx:eax and
; before we print out the results.
;
break1:
    sub eax, [cycleLow]
    sbb edx, [cycleHigh]
break2:
    push    eax
    push    edx
    push    edx
    push    eax
    push    dword fmtStr
    call    printf
    add esp, 20    ; Pop stack 5 times 4 bytes

;
; Call exit(3) syscall
; void exit(int status)
;
mov ebx, 0    ; Arg one: the status
mov eax, 1    ; Syscall number:
int 0x80

```

In order to assemble, link and run the program we need to do the following:

```

$ nasm -felf -g rdtsc.asm -l rdtsc.lst
$ gcc -m32 -o rdtsc rdtsc.o
$ ./rdtsc

```

X86 Interrupts

Interrupts are special routines that are defined on a per-system basis. This means that the interrupts on one system might be different from the interrupts on another system. Therefore, it is usually a bad idea to rely heavily on interrupts when you are writing code that needs to be portable.

What is an Interrupt?

In modern operating systems, the programmer often doesn't need to use interrupts. In Windows, for example, the programmer conducts business with the Win32 API. However, these API calls interface with the kernel, and the kernel will often trigger interrupts to perform different tasks. In older operating systems (specifically DOS), the programmer didn't have an API to use, and so they had to do all their work through interrupts.

Interrupt Instruction

int arg

This instruction issues the specified interrupt. For instance:

`int 0x0A`

Calls interrupt 10 (0x0A (hex) = 10 (decimal)).

Types of Interrupts

There are 3 types of interrupts: Hardware Interrupts, Software Interrupts and Exceptions.

Hardware Interrupts

Hardware interrupts are triggered by hardware devices. For instance, when you type on your keyboard, the keyboard triggers a hardware interrupt. The processor stops what it is doing, and executes the code that handles keyboard input (typically reading the key you pressed into a buffer in memory). Hardware interrupts are typically asynchronous - their occurrence is unrelated to the instructions being executed at the time they are raised.

Software Interrupts

There are also a series of software interrupts that are usually used to transfer control to a function in the operating system kernel. Software interrupts are triggered by the instruction **int**. For example, the instruction "int 14h" triggers interrupt 0x14. The processor then stops the current program, and jumps to the code to handle interrupt 14. When interrupt handling is complete, the processor returns flow to the original program.

Exceptions

Exceptions are caused by exceptional conditions in the code which is executing, for example an attempt to divide by zero or access a protected memory area. The processor will detect this problem, and transfer control to a handler to service the exception. This handler may re-execute the offending code after changing some value (for example, the zero dividend), or if this cannot be done, the program causing the exception may be terminated.

Further Reading

A great list of interrupts **for DOS and related systems** is at [Ralf Brown's Interrupt List](http://www.ctyme.com/rbrown.htm). (<http://www.ctyme.com/rbrown.htm>)

x86 Assemblers

There are a number of different assemblers available for x86 architectures. This page will list some of them, and will discuss where to get the assemblers, what they are good for, and where they are used the most.

GNU Assembler (GAS)

The GNU assembler is most common as the assembly back-end to the GCC compiler. One of the most compelling reasons to learn to program GAS (as it is frequently abbreviated) is to write inline assembly instructions (assembly code embedded in C source code), as these instructions (when compiled by the `gcc`) need to be in GAS syntax. GAS uses the AT&T syntax for writing the assembly language, which some people claim is more complicated, but other people say it is more informative.

Note: Recent versions of GCC include the `"-masm"` option which, when set to `"-masm=intel"`, allow the user to define inline assembly using the Intel syntax. The equivalent option for GAS is `"-msyntax=intel"` or using the `".intel_syntax"` directive.

Microsoft Macro Assembler (MASM)

Microsoft's Macro Assembler, MASM, has been in constant production for many many years. Many people claim that MASM isn't being supported or improved anymore, but Microsoft denies this: MASM is maintained, but is currently in a bug-fixing mode. No new features are currently being added. However, Microsoft is shipping a 64-bit version of MASM with new 64-bit compiler suites. MASM is available from Microsoft as part of Visual C++, as a download from MSDN, or as part of the Microsoft DDK. The latest available version of MASM is version 11.x (ref.: www.masm32.com).

MASM uses the Intel syntax for its instructions, which stands in stark contrast to the AT&T syntax used by the GAS assembler. Most notably, MASM instructions take their operands in reverse order from GAS. This one fact is perhaps the biggest stumbling block for people trying to transition between the two assemblers.

MASM also has a very powerful macro engine, which many programmers use to implement a high-level feel in MASM programs.

External Links

- <http://www.masmforum.com>
- <http://www.movsd.com>

JWASM

JWASM is a 16, 32 and 64-bit assembler for 80x86 platforms, based upon Open Watcom's WASM, and was created by Japheth.

While syntactically compatible with MASM, it is faster, and its sourcecode is freely available under the Sybase Open Watcom Public License, and thus it is free for both commercial and non-commercial use. Furthermore, it supports ELF, and is thus the only cross-platform assembler supporting the popular MASM syntax. JWASM is actively being developed, and is generally regarded as the unofficial successor to MASM.

External Links

- <http://www.japheth.de/JWasm.html>
- <http://sourceforge.net/projects/jwasm/>

Netwide Assembler (NASM)

The Netwide Assembler, NASM, was started as an open-source initiative to create a free, retargetable assembler for 80x86 platforms. When the NASM project was started, MASM was still being sold by microsoft (MASM is currently free), and GAS contained very little error checking capability. GAS was, after all, the backend to GCC, and GCC always feeds GAS syntax-correct code. For this reason, GAS didn't need to interface with the user much, and therefore writing code for GAS was very tough.

NASM uses a syntax which is "similar to Intel's but less complex".

The NASM users manual is found at <http://www.nasm.us/doc/> .

Features:

- Cross platform: Like Gas, this assembler runs on nearly every platform, supposedly even on PowerPC Macs (though the code generated will only run on an x86 platform)
- Open Source
- Macro language (code that writes code)

Flat Assembler (FASM)

Although it was written in assembly, it runs on several operating systems, including DOS, DexOS, Linux, Windows, and BSD. Its syntax is similar to TASM's "ideal mode" and NASM's but the macros in this assembler are done differently.

Features:

- Written in itself; and therefore its source code is an example of how to write in this assembler
- Open source
- Clean NASM-like syntax
- Very very fast
- Has macro language (code that writes code)
- Built-in IDE for DOS and Windows
- Creates binary, MZ, PE, ELF, COFF - no linker needed

External Links

- <http://flatassembler.net/>

YASM Assembler

YASM is a ground-up rewrite of NASM under the new BSD licence. YASM is designed to understand multiple syntaxes natively (NASM and GAS, currently). The primary focus of YASM is to produce "libyasm", a reusable library that can work with code at a low level, and can be easily integrated into other software projects.

External Links

- <http://www.tortall.net/projects/yasm/>

HLA

HLA is an assembler front-end created by Randall Hyde and first popularized in his book "The Art of Assembly".

HLA accepts assembly written using a high-level format, and converts the code into another format (MASM or GAS, usually). Another assembler (MASM or GAS) will then assemble the instructions into machine code.

BBC BASIC for WINDOWS (proprietary)

The proprietary BBC BASIC for Windows supports the development of 32 bit x86 assembler targeting user mode for Windows using INTEL syntax, but does not currently permit the generation of standalone EXE's (without the inclusion of a proprietary runtime and environment). Macro assembly is possible by use the BBC BASIC environment, defining macros by means of BASIC functions wrapped around the relevant code.

More information is in the [Assembler section of the manual](http://www.bbcbasic.co.uk/bbcwin/manual/bbcwina.html) (<http://www.bbcbasic.co.uk/bbcwin/manual/bbcwina.html>)

GAS Syntax

General Information

Examples in this article are created using the AT&T assembly syntax used in GNU AS. The main advantage of using this syntax is its compatibility with the GCC inline assembly syntax. However, this is not the only syntax that is used to represent x86 operations. For example, NASM uses a different syntax to represent assembly mnemonics, operands and addressing modes, as do some High-Level Assemblers. The AT&T syntax is the standard on Unix-like systems but some assemblers use the Intel syntax, or can, like GAS itself, accept both.

GAS instructions generally have the form mnemonic source, destination. For instance, the following **mov** instruction:

```
movb $0x05, %al
```

This will move the hexadecimal value 5 into the register *al*.

Operation Suffixes

GAS assembly instructions are generally suffixed with the letters "b", "s", "w", "l", "q" or "t" to determine what size operand is being manipulated.

- b = byte (8 bit).
- s = single (32-bit floating point).
- w = word (16 bit).
- l = long (32 bit integer or 64-bit floating point).
- q = quad (64 bit).
- t = ten bytes (80-bit floating point).

If the suffix is not specified, and there are no memory operands for the instruction, GAS infers the operand size from the size of the destination register operand (the final operand).

Prefixes

When referencing a register, the register needs to be prefixed with a "%". Constant numbers need to be prefixed with a "\$".

Address operand syntax

There are up to 4 parameters of an address operand that are presented in the syntax `segment:displacement(base register, index register, scale factor)`. This is equivalent to `segment:[base register + displacement + index register * scale factor]` in Intel syntax.

The base, index and displacement components can be used in any combination, and every component can be omitted; omitted components are excluded from the calculation above^{[1][2]}.

```
movl    -8(%ebp, %edx, 4), %eax    # Full example: load *(ebp + (edx * 4) - 8) into eax
movl    -4(%ebp), %eax             # Typical example: load a stack variable into eax
movl    (%ecx), %edx               # No index: copy the target of a pointer into a register
leal    8(%eax,4), %eax             # Arithmetic: multiply eax by 4 and add 8
leal    (%edx,%eax,2), %eax         # Arithmetic: multiply eax by 2 and add edx
```

Introduction

This section is written as a short introduction to GAS. GAS is part of the GNU Project (<http://www.gnu.org/>), which gives it the following nice properties:

- It is available on many operating systems.

- It interfaces nicely with the other GNU programming tools, including the GNU C compiler (gcc) and GNU linker (ld).

If you are using a computer with the Linux operating system, chances are you already have GAS installed on your system. If you are using a computer with the Windows operating system, you can install GAS and other useful programming utilities by installing [Cygwin](http://www.cygwin.com/) (<http://www.cygwin.com/>) or [Mingw](http://www.mingw.org/) (<http://www.mingw.org/>). The remainder of this introduction assumes you have installed GAS and know how to open a command-line interface and edit files.

Generating assembly

Since assembly language corresponds directly to the operations a CPU performs, a carefully written assembly routine may be able to run much faster than the same routine written in a higher-level language, such as C. On the other hand, assembly routines typically take more effort to write than the equivalent routine in C. Thus, a typical method for quickly writing a program that performs well is to first write the program in a high-level language (which is easier to write and debug), then rewrite selected routines in assembly language (which performs better). A good first step to rewriting a C routine in assembly language is to use the C compiler to automatically generate the assembly language. Not only does this give you an assembly file that compiles correctly, but it also ensures that the assembly routine does exactly what you intended it to.^[3]

We will now use the GNU C compiler to generate assembly code, for the purposes of examining the GAS assembly language syntax.

Here is the classic "Hello, world" program, written in C:

```
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

Save that in a file called "hello.c", then type at the prompt:

```
gcc -o hello_c hello.c
```

This should compile the C file and create an executable file called "hello_c". If you get an error, make sure that the contents of "hello.c" are correct.

Now you should be able to type at the prompt:

```
./hello_c
```

and the program should print "Hello, world!" to the console.

Now that we know that "hello.c" is typed in correctly and does what we want, let's generate the equivalent 32-bit x86 assembly language. Type the following at the prompt:

```
gcc -S -m32 hello.c
```

This should create a file called "hello.s" (".s" is the file extension that the GNU system gives to assembly files). On more recent 64-bit systems, the 32-bit source tree may not be included, which will cause a "bits/predefs.h fatal error"; you may replace the -m32 gcc directive with an -m64 directive to generate 64-bit assembly instead. To compile the assembly file into an executable, type:

```
gcc -o hello_asm -m32 hello.s
```

(Note that gcc calls the assembler (as) and the linker (ld) for us.) Now, if you type the following at the prompt:

```
./hello_asm
```

this program should also print "Hello, world!" to the console. Not surprisingly, it does the same thing as the compiled C file.

Let's take a look at what is inside "hello.s":

```
1      .file      "hello.c"
2      .def       __main;          .scl  2;      .type  32;      .endef
3      .text
4  LC0:
5      .ascii     "Hello, world!\n2\0"
6  .globl __main
7      .def       __main;          .scl  2;      .type  32;      .endef
8  __main:
9      pushl      %ebp
10     movl       %esp, %ebp
11     subl       $8, %esp
12     andl       $-16, %esp
13     movl       $0, %eax
14     movl       %eax, -4(%ebp)
15     movl       -4(%ebp), %eax
16     call       __alloca
17     call       __main
18     movl       $LC0, (%esp)
19     call       _printf
20     movl       $0, %eax
21     leave
22     ret
23     .def       _printf;          .scl  2;      .type  32;      .endef
```

The contents of "hello.s" may vary depending on the version of the GNU tools that are installed; this version was generated with Cygwin, using gcc version 3.3.1.

The lines beginning with periods, like `.file`, `.def`, or `.ascii` are assembler directives — commands that tell the assembler how to assemble the file. The lines beginning with some text followed by a colon, like `__main:`, are labels, or named locations in the code. The other lines are assembly instructions.

The `.file` and `.def` directives are for debugging. We can leave them out:

```
1      .text
2  LC0:
3      .ascii     "Hello, world!\n2\0"
4  .globl __main
5  __main:
6      pushl      %ebp
7      movl       %esp, %ebp
8      subl       $8, %esp
9      andl       $-16, %esp
10     movl       $0, %eax
11     movl       %eax, -4(%ebp)
12     movl       -4(%ebp), %eax
13     call       __alloca
14     call       __main
15     movl       $LC0, (%esp)
16     call       _printf
17     movl       $0, %eax
18     leave
19     ret
```

"hello.s" line-by-line

```
1      .text
```

This line declares the start of a section of code. You can name sections using this directive, which gives you fine-grained control over where in the executable the resulting machine code goes, which is useful in some cases, like for programming embedded systems. Using `.text` by itself tells the assembler that the following code goes in the default section, which is sufficient for most purposes.

```
2  LC0:
3      .ascii     "Hello, world!\n2\0"
```

This code declares a label, then places some raw ASCII text into the program, starting at the label's location. The `\n` specifies a line-feed character, while the `\0` specifies a null character at the end of the string; C routines mark the end of strings with null characters, and since we are going to call a C string routine, we need this character here. (NOTE! String in C is an array of datatype `char` (`char[]`) and does not exist in any other form, but because one would understand strings as a single entity from the majority of programming languages, it is clearer to express it this way.)

```
4  .globl _main
```

This line tells the assembler that the label `_main` is a global label, which allows other parts of the program to see it. In this case, the linker needs to be able to see the `_main` label, since the startup code with which the program is linked calls `_main` as a subroutine.

```
5  _main:
```

This line declares the `_main` label, marking the place that is called from the startup code.

```
6      pushl    %ebp
7      movl     %esp, %ebp
8      subl     $8, %esp
```

These lines save the value of EBP on the stack, then move the value of ESP into EBP, then subtract 8 from ESP. Note that `pushl` automatically decremented ESP by the appropriate length. The `l` on the end of each opcode indicates that we want to use the version of the opcode that works with *long* (32-bit) operands; usually the assembler is able to work out the correct opcode version from the operands, but just to be safe, it's a good idea to include the `l`, `w`, `b`, or other suffix. The percent signs designate register names, and the dollar sign designates a literal value. This sequence of instructions is typical at the start of a subroutine to save space on the stack for local variables; EBP is used as the base register to reference the local variables, and a value is subtracted from ESP to reserve space on the stack (since the Intel stack grows from higher memory locations to lower ones). In this case, eight bytes have been reserved on the stack. We shall see why this space is needed later.

```
9      andl     $-16, %esp
```

This code `ands` ESP with `0xFFFFF0`, aligning the stack with the next lowest 16-byte boundary. An examination of Mingw's source code reveals that this may be for SIMD instructions appearing in the `_main` routine, which operate only on aligned addresses. Since our routine doesn't contain SIMD instructions, this line is unnecessary.

```
10     movl     $0, %eax
11     movl     %eax, -4(%ebp)
12     movl     -4(%ebp), %eax
```

This code moves zero into EAX, then moves EAX into the memory location `EBP - 4`, which is in the temporary space we reserved on the stack at the beginning of the procedure. Then it moves the memory location `EBP - 4` back into EAX; clearly, this is not optimized code. Note that the parentheses indicate a memory location, while the number in front of the parentheses indicates an offset from that memory location.

```
13     call     __alloca
14     call     __main
```

These functions are part of the C library setup. Since we are calling functions in the C library, we probably need these. The exact operations they perform vary depending on the platform and the version of the GNU tools that are installed.

```
15     movl     $LC0, (%esp)
16     call     _printf
```

This code (finally!) prints our message. First, it moves the location of the ASCII string to the top of the stack. It seems that the C compiler has optimized a sequence of `popl %eax; pushl $LC0` into a single move to the top of the stack. Then, it calls the `_printf` subroutine in the C library to print the message to the console.

```
17      movl    $0, %eax
```

This line stores zero, our return value, in EAX. The C calling convention is to store return values in EAX when exiting a routine.

```
18      leave
```

This line, typically found at the end of subroutines, frees the space saved on the stack by copying EBP into ESP, then popping the saved value of EBP back to EBP.

```
19      ret
```

This line returns control to the calling procedure by popping the saved instruction pointer from the stack.

Communicating directly with the operating system

Note that we only have to call the C library setup routines if we need to call functions in the C library, like `printf()`. We could avoid calling these routines if we instead communicate directly with the operating system. The disadvantage of communicating directly with the operating system is that we lose portability; our code will be locked to a specific operating system. For instructional purposes, though, let's look at how one might do this under Windows. Here is the C source code, compilable under Mingw or Cygwin:

```
#include <windows.h>

int main(void) {
    LPSTR text = "Hello, world!\n";
    DWORD charsWritten;
    HANDLE hStdout;

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    WriteFile(hStdout, text, 14, &charsWritten, NULL);
    return 0;
}
```

Ideally, you'd want check the return codes of "GetStdHandle" and "WriteFile" to make sure they are working correctly, but this is sufficient for our purposes. Here is what the generated assembly looks like:

```
1      .file     "hello2.c"
2      .def      __main;      .scl  2;      .type  32;      .endef
3      .text
4  LC0:
5      .ascii   "Hello, world!\12\0"
6      .globl  __main
7      .def      __main;      .scl  2;      .type  32;      .endef
8  __main:
9      pushl    %ebp
10     movl     %esp, %ebp
11     subl     $4, %esp
12     andl     $-16, %esp
13     movl     $0, %eax
14     movl     %eax, -16(%ebp)
15     movl     -16(%ebp), %eax
16     call     __alloca
17     call     __main
18     movl     $LC0, -4(%ebp)
19     movl     $-11, (%esp)
20     call     _GetStdHandle@4
21     subl     $4, %esp
22     movl     %eax, -12(%ebp)
23     movl     $0, 16(%esp)
24     leal     -8(%ebp), %eax
25     movl     %eax, 12(%esp)
26     movl     $14, 8(%esp)
27     movl     -4(%ebp), %eax
28     movl     %eax, 4(%esp)
29     movl     -12(%ebp), %eax
30     movl     %eax, (%esp)
31     call     _WriteFile@20
32     subl     $20, %esp
33     movl     $0, %eax
34     leave
35     ret
```

Even though we never use the C standard library, the generated code initializes it for us. Also, there is a lot of unnecessary stack manipulation. We can simplify:

```
1      .text
2  LCO:
3      .ascii "Hello, world!\12\0"
4  .globl _main
5  _main:
6      pushl   %ebp
7      movl    %esp, %ebp
8      subl    $4, %esp
9      pushl    $-11
10     call    _GetStdHandle@4
11     pushl    $0
12     leal    -4(%ebp), %ebx
13     pushl    %ebx
14     pushl    $14
15     pushl    $LCO
16     pushl    %eax
17     call    _WriteFile@20
18     movl    $0, %eax
19     leave
20     ret
```

Analyzing line-by-line:

```
6      pushl   %ebp
7      movl    %esp, %ebp
8      subl    $4, %esp
```

We save the old EBP and reserve four bytes on the stack, since the call to WriteFile needs somewhere to store the number of characters written, which is a 4-byte value.

```
9      pushl    $-11
10     call    _GetStdHandle@4
```

We push the constant value `STD_OUTPUT_HANDLE` (-11) to the stack and call `GetStdHandle`. The returned handle value is in EAX.

```
11     pushl    $0
12     leal    -4(%ebp), %ebx
13     pushl    %ebx
14     pushl    $14
15     pushl    $LCO
16     pushl    %eax
17     call    _WriteFile@20
```

We push the parameters to `WriteFile` and call it. Note that the Windows calling convention is to push the parameters from right-to-left. The load-effective-address (`leal`) instruction adds -4 to the value of EBP, giving the location we saved on the stack for the number of characters printed, which we store in EBX and then push onto the stack. Also note that EAX still holds the return value from the `GetStdHandle` call, so we just push it directly.

```
18     movl    $0, %eax
19     leave
```

Here we set our program's return value and restore the values of EBP and ESP using the `leave` instruction.

Caveats

From The GAS manual's AT&T Syntax Bugs section (http://sourceware.org/binutils/docs/as/i386_002dBugs.html#i386_002dBugs):

The UnixWare assembler, and probably other AT&T derived ix86 Unix assemblers, generate floating point instructions with reversed source and destination registers in certain cases. Unfortunately, gcc and possibly many other programs use this reversed syntax, so we're stuck with it.

For example

```
fsub %st, %st(3)
```

results in `%st(3)` being updated to `%st - %st(3)` rather than the expected `%st(3) - %st`. This happens with all the non-commutative arithmetic floating point operations with two register operands where the source register is `%st` and the destination register is `%st(i)`.

Note that even `objdump -d -M intel` still uses reversed opcodes, so use a different disassembler to check this. See <http://bugs.debian.org/372528> for more info.

Additional GAS reading

You can read more about GAS at the GNU GAS documentation page:

<https://sourceware.org/binutils/docs/as/>

- [X86 Disassembly/Calling Conventions](#)

Quick reference

Instruction	Meaning
<code>movq %rax, %rbx</code>	<code>rbx = rax</code>
<code>movq \$123, %rax</code>	<code>rax = 123</code>
<code>movq %rsi, -16(%rbp)</code>	<code>mem[rbp-16] = rsi</code>
<code>subq \$10, %rbp</code>	<code>rbp = rbp - 10</code>
<code>cmpl %eax %ebx</code>	compare then set flags. If <code>eax == ebx</code> , zero flag is set.
<code>jmp <location></code>	unconditional jump
<code>je <location></code>	jump to <code><location></code> if equal flag is set
<code>jg, jge, jl, jle, jne, ...</code>	<code>>, >=, <, <=, !=, ...</code>

Notes

1. If segment is not specified, as almost always, it is assumed to be `ds`, unless base register is `esp` or `ebp`; in this case, the address is assumed to be relative to `ss`
2. If index register is missing, the pointless scale factor must be omitted as well.
3. This assumes that the compiler has no bugs and, more importantly, *that the code you wrote correctly implements your intent*. Note also that compilers can sometimes rearrange the sequence of low-level operations in order to optimize the code; this preserves the overall semantics of your code but means the assembly instruction flow may not match up exactly with your algorithm steps.

MASM Syntax

This page will explain x86 Programming using MASM syntax, and will also discuss how to use the macro capabilities of MASM. Other assemblers, such as [NASM](#) and [FASM](#), use syntax different from MASM, similar only in that they all use Intel syntax.

Instruction Order

MASM instructions typically have operands reversed from GAS instructions. for instance, instructions are typically written as **Instruction Destination, Source**.

The **mov** instruction, written as follows:

```
mov al, 05h
```

will move the value 5 into the al register.

Instruction Suffixes

MASM does not use instruction suffixes to differentiate between sizes (byte, word, dword, etc).

Macros

MASM is known as either the "Macro Assembler", or the "Microsoft Assembler", depending on who you talk to. But no matter where your answers are coming from, the fact is that MASM has a powerful macro engine, and a number of built-in macros available immediately.

MASM directives

MASM has a large number of directives that can control certain settings and behaviors. It has more of them compared to NASM or FASM, for example.

```
.model small
.stack 100h

.data
msg db 'Hello world!$'

.code
start:
    mov ah, 09h    ; Display the message
    lea dx, msg
    int 21h
    mov ax, 4C00h  ; Terminate the executable
    int 21h

end start
```

A Simple Template for MASM510 programming

```
;template for masm510 programming using simplified segment definition
title YOUR TITLE HERE
page 60,132
;tell the assembler to create a nice .lst file for the convenience of error pruning
.model small
;maximum of 64KB for data and code respectively
.stack 64
.data
;PUT YOUR DATA DEFINITION HERE
.code
main proc far
;This is the entry point, you can name your procedures by altering "main" according to some rules
mov ax,@DATA
;load the data segment address,"@" is the opcode for fetching the offset of "DATA","DATA" could be change according
to your previous definition for data
mov ds,ax
;assign value to ds,"mov" cannot be used for copying data directly to segment registers(cs,ds,ss,es)
;PUT YOUR CODE HERE
mov ah,4ch
int 21h
;terminate program by a normal way
main endp
;end the "main" procedure
end main
;end the entire program centering around the "main" procedure
```

HLA Syntax

HLA Syntax

HLA accepts assembly written using a high-level format, and converts the code into another format (MASM or GAS, usually).

In MASM, for instance, we could write the following code:

```
mov EAX, 0x05
```

In HLA, this code would become:

```
mov(0x05, EAX);
```

HLA uses the same order-of-operations as GAS syntax, but doesn't require any of the name decoration of GAS. Also, HLA uses the parenthesis notation to call an instruction. HLA terminates its lines with a semicolon, similar to C or Pascal.

High-Level Constructs

Some people criticize HLA because it "isn't low-level enough". This is false, because HLA can be as low-level as MASM or GAS, but it also offers the options to use some higher-level abstractions. For instance, HLA can use the following syntax to pass `eax` as an argument to the `Function1` function:

```
push(eax);  
call(Function1);
```

But HLA also allows the programmer to simplify the process, if they want:

```
Function1(eax);
```

This is called the "parenthesis notation" for calling functions.

HLA also contains a number of different loops (do-while, for, until, etc..) and control structures (if-then-else, switch-case) that the programmer can use. However, these high-level constructs come with a caveat: Using them may be simple, but they translate into MASM code instructions. It is usually faster to implement the loops by hand.

FASM Syntax

A Wikibookian has nominated this page for cleanup because:



page needs general work

You can [help make it better](https://en.wikibooks.org/w/index.php?title=X86_Assembly/Print_Version&action=edit) (https://en.wikibooks.org/w/index.php?title=X86_Assembly/Print_Version&action=edit). Please review any [relevant discussion](#).

FASM, also known as **Flat Assembler**, is an optimizing assembler for the x86 architecture. FASM is written in assembly, so it can assemble/bootstrap itself. It runs on various operating systems including DOS, Windows, Linux, and Unix. It supports the x86 and x86-64 instruction sets including SIMD extensions MMX, SSE - SSE4, and AVX.

Hexadecimal Numbers

FASM supports all popular syntaxes used to define hexadecimal numbers:

```
0xbadf00d ; C-Like Syntax  
$badf00d  ; Pascal-Like Syntax  
@badf00dh ; h Syntax, requires leading zero to be valid at assembly time
```

Labels

FASM supports several unique labeling features.

Anonymous Labels

FASM supports labels that use no identifier or label name.

- @@: represents an anonymous label. Any number of anonymous labels can be defined.
- @b refers to the closest @@ that can be found when looking backwards in source. @r and @b are equivalent.
- @f refers to the closest @@ that can be found when looking forward in source.

```
@@:
    inc eax
    push eax
    jmp @b      ; This will result in a stack fault sooner or later
    jmp @f      ; This instruction will never be hit
@@:            ; if jmp @f was ever hit, the instruction pointer would be set to this anonymous label
    invoke ExitProcess, 0 ; Winasm only
```

Local Labels

Local labels, which begin with a . (period). You can reference a local label in the context of its global label parent.

```
entry globallabel

globallabel:
    .locallabelone:
        jmp globallabel2.locallabelone
    .locallabeltwo:

globallabel2:
    .locallabelone:
    .locallabeltwo:
        jmp globallabel.locallabelone ; infinite loop
```

Operators

FASM supports several unique operators to simplify assembly code.

The \$ Operator

\$ describes the current location in an addressing space. It is used to determine the size of a block of code or data. The MASM equivalent of the \$ is equivalent is the **SIZEOF** operator.

```
mystring db "This is my string", 0
mystring.length = $ - mystring
```

The # Operator

is the symbol concatenation operator, used for combining multiple symbols into one. It can only be used inside of the body of a macro like **rept** or a custom/user-defined macro, because it will replace the name of the macro argument supplied with its value.

```
macro contrived value {
    some#value db 22
}
; ...
contrived 2

; assembles to...
some2 db 22
```

The ` Operator

` is used to obtain the name of a symbol passed to a macro, converting it to a string.

```
macro print_contrived value {
    formatter db "%s\n"
    invoke printf, formatter, `value
}
; ...
print_contrived SOMEVALUE
```

```
; assembles to...
formatter db "%s\n"
invoke printf, formatter, "SOMEVALUE"
```

Built In Macros

FASM has several useful built in macros to simplify writing assembly code.

Repetition

The **rept** directive is used to compact repetitive assembly instructions into a block. The directive begins with the word **rept**, then a number or variable specifying the number of times the assembly instructions inside of the curly braces proceeding the instruction should be repeated. The counter variable can be aliased to be used as a symbol, or as part of an instruction within the **rept** block.

```
rept 2 {
    db "Hello World!", 0Ah, 0
}

; assembles to...
db "Hello World!", 0Ah, 0
db "Hello World!", 0Ah, 0

; and...
rept 2 helloNumber {
    hello#helloNumber db "Hello World!", 0Ah, 0 ; use the symbol concatenation operator '#' to create unique labels
    hello1 and hello2
}

; assembles to...
hello1 db "Hello World!", 0Ah, 0
hello2 db "Hello World!", 0Ah, 0
```

Structures

The **struc** directive allows assembly of data into a format similar to that of a C structure with members. The definition of a **struc** makes use of local labels to define member values.

```
struc 3dpoint x, y, z
{
    .x db x,
    .y db y,
    .z db z
}

some 3dpoint 1, 2, 3

; assembles to...
some:
    .x db 1
    .y db 2
    .z db 3

; access a member through some.x, some.y, or some.z for x, y, and z respectively
```

Custom Macros

FASM supports defining custom macros as a way of assembling multiple instructions or conditional assembly as one larger instruction. They require a name and can have an optional list of arguments, separated by commas.

```
macro name arg1, arg2, ... {
    ; <macro body>
}
```

Variable Arguments

Macros can support a variable number of arguments through the square bracket syntax.

```
macro name arg1, arg2, [varargs] {
    ; <macro body>
}
```

```
}
```

Required Operands

The FASM macro syntax can require operands in a macro definition using the ***** operator after each operand.

```
; all operands required, will not assemble without
macro mov op1*, op2*, op3*
{
    mov op1, op2
    mov op2, op3
}
```

Operator Overloading

The FASM macro syntax allows for the overloading of the syntax of an instruction, or creating a new instruction. Below, the **mov** instruction has been overloaded to support a third operand. In the case that none is supplied, the regular move instruction is assembled. Otherwise, the data in **op2** is moved to **op1** and **op2** is replaced by **op3**.

```
; not all operands required, though if op1 or op2 are not supplied
; assembly should fail
; could also be defined as 'macro mov op1*, op2*, op3' to force requirement of the first two arguments
macro mov op1, op2, op3
{
    if op3 eq
        mov op1, op2
    else
        mov op1, op2
        mov op2, op3
    end if
}
```

Hello World

This is a complete example of a Win32 assembly program that prints 'Hello World!' to the console and then waits for the user to press any key before exiting the application.

```
format PE console                ; Win32 portable executable console format
entry _start                     ; _start is the program's entry point

include 'INCLUDE/WIN32A.INC'

section '.data' data readable writable    ; data definitions
hello db "Hello World!", 0
stringformat db "%s", 0ah, 0

section '.code' code readable executable  ; code

_start:
    invoke printf, stringformat, hello    ; call printf, defined in msvcrt.dll
    invoke getchar                        ; wait for any key
    invoke ExitProcess, 0                 ; exit the process

section '.imports' import data readable    ; data imports

library kernel, 'kernel32.dll', \
    msvcrt, 'msvcrt.dll'                 ; link to kernel32.dll, msvcrt.dll

import kernel, \
    ExitProcess, 'ExitProcess'           ; import ExitProcess from kernel32.dll

import msvcrt, \
    printf, 'printf', \
    getchar, '_fgetchar'                 ; import printf and getchar from msvcrt.dll
```

External Links

- FASM website (<http://flatassembler.net/>)
- FASM official manual (<http://flatassembler.net/docs.php?article=manual>)
- TAJGA FASM tutorial (<http://bos.asmhackers.net/docs/FASM%20tutorial/>)
- TAJGA FASM preprocessor tutorial (<http://bos.asmhackers.net/docs/FASM%20tutorial/preproc.html>)

NASM Syntax

The Netwide Assembler is an x86 and x86-64 assembler that uses syntax similar to Intel. It supports a variety of object file formats, including:

1. ELF32/64
2. Linux a.out
3. NetBSD/FreeBSD a.out
4. MS-DOS 16-bit/32-bit object files
5. Win32/64 object files
6. COFF
7. Mach-O 32/64
8. rdf
9. binary

NASM runs on both Unix/Linux and Windows/DOS.

NASM Syntax

The Netwide Assembler (NASM) uses a syntax "designed to be simple and easy to understand, similar to Intel's but less complex". This means that the operand order is **dest** then **src**, as opposed to the AT&T style used by the GNU Assembler. For example,

```
mov ax, 9
```

loads the number 9 into register ax.

For those using gdb with nasm, you can set gdb to use Intel-style disassembly by issuing the command:

```
set disassembly-flavor intel
```

Comments

A single semi-colon is used for comments, and functions the same as double slash in C++: the compiler ignores from the semicolon to the next newline.

Macros

NASM has powerful macro functions, similar to C's preprocessor. For example,

```
%define newline 0xA
%define func(a, b) ((a) * (b) + 2)

func (1, 22) ; expands to ((1) * (22) + 2)

%macro print 1 ; macro with one argument
    push dword %1 ; %1 means first argument
    call printf
    add esp, 4
%endmacro

print mystring ; will call printf
```

Example I/O (Linux and BSD)

To pass the kernel a simple input command on Linux, you would pass values to the following registers and then send the kernel an interrupt signal. To read in a single character from standard input (such as from a user at their keyboard), do the following:

```

; read a byte from stdin
mov eax, 3      ; 3 is recognized by the system as meaning "read"
mov ebx, 0      ; read from standard input
mov ecx, variable ; address to pass to
mov edx, 1      ; input length (one byte)
int 0x80        ; call the kernel

```

After the `int 0x80`, `eax` will contain the number of bytes read. If this number is < 0 , there was a read error of some sort.

Outputting follows a similar convention:

```

; print a byte to stdout
mov eax, 4      ; the system interprets 4 as "write"
mov ebx, 1      ; standard output (print to terminal)
mov ecx, variable ; pointer to the value being passed
mov edx, 1      ; length of output (in bytes)
int 0x80        ; call the kernel

```

BSD systems (MacOS X included) use similar system calls, but convention to execute them is different. While on Linux you pass system call arguments in different registers, on BSD systems they are pushed onto stack (except the system call number, which is put into `eax`, the same way as in Linux). BSD version of the code above:

```

; read a byte from stdin
mov eax, 3      ; sys_read system call
push dword 1    ; input length
push dword variable ; address to pass to
push dword 0    ; read from standard input
push eax
int 0x80        ; call the kernel
add esp, 16     ; move back the stack pointer

; write a byte to stdout
mov eax, 4      ; sys_write system call
push dword 1    ; output length
push dword variable ; memory address
push dword 1    ; write to standard output
push eax
int 0x80        ; call the kernel
add esp, 16     ; move back the stack pointer

; quit the program
mov eax, 1      ; sys_exit system call
push dword 0    ; program return value
push eax
int 0x80        ; call the kernel

```

Hello World (Linux)

Below we have a simple Hello world example, it lays out the basic structure of a nasm program:

```

global _start

section .data
; Align to the nearest 2 byte boundary, must be a power of two
align 2
; String, which is just a collection of bytes, 0xA is newline
str: db 'Hello, world!',0xA
strLen: equ $-str

section .bss

section .text
_start:

;
;      op      dst, src
;
; Call write(2) syscall:
;      ssize_t write(int fd, const void *buf, size_t count)
;
; Arg three: the length of the string
; Arg two: the address of the string
; Arg one: file descriptor, in this case stdout
; Syscall number, in this case the write(2) syscall:
; Interrupt 0x80

mov     edx, strLen
mov     ecx, str
mov     ebx, 1
mov     eax, 4
int     0x80

;
; Call exit(3) syscall
;      void exit(int status)
;
; Arg one: the status

mov     ebx, 0

```

```
mov     eax, 1           ; Syscall number:
int     0x80
```

In order to assemble, link and run the program we need to do the following:

```
$ nasm -f elf32 -g helloWorld.asm
$ ld -g helloWorld.o
$ ./a.out
```

Hello World (Using only Win32 system calls)

In this example we are going to rewrite the hello world example using Win32 system calls. There are several major differences:

1. The intermediate file will be a Microsoft Win32 (i386) object file
2. We will avoid using interrupts since they may not be portable and therefore we need to bring in several calls from kernel32 DLL

```
global _start

extern _GetStdHandle@4
extern _WriteConsoleA@20
extern _ExitProcess@4

section .data
str:    db 'hello, world',0x0D,0x0A
strLen: equ $-str

section .bss
numCharsWritten:    resd 1

section .text
_start:

;
; HANDLE WINAPI GetStdHandle( _In_  DWORD nStdHandle ) ;
;
push    dword -11      ; Arg1: request handle for standard output
call    _GetStdHandle@4 ; Result: in eax

;
; BOOL WINAPI WriteConsole(
;     _In_      HANDLE hConsoleOutput,
;     _In_      const VOID *lpBuffer,
;     _In_      DWORD nNumberOfCharsToWrite,
;     _Out_     LPDWORD lpNumberOfCharsWritten,
;     _Reserved_ LPVOID lpReserved ) ;
;
push    dword 0         ; Arg5: Unused so just use zero
push    numCharsWritten ; Arg4: push pointer to numCharsWritten
push    dword strLen    ; Arg3: push length of output string
push    str             ; Arg2: push pointer to output string
push    eax             ; Arg1: push handle returned from _GetStdHandle
call    _WriteConsoleA@20

;
; VOID WINAPI ExitProcess( _In_  UINT uExitCode ) ;
;
push    dword 0         ; Arg1: push exit code
call    _ExitProcess@4
```

In order to assemble, link and run the program we need to do the following. This example was run under cygwin, in a Windows command prompt the link step would be different. In this example we use the `-e` command line option when invoking `ld` to specify the entry point for program execution. Otherwise we would have to use `_WinMain@16` as the entry point rather than `_start`. One last note, `WriteConsole()` does not behave well within a cygwin console, so in order to see output the final exe should be run within a Windows command prompt:

```
$ nasm -f win32 -g helloWorldWin32.asm
$ ld -e _start helloWorldwin32.obj -lkernel32 -o helloWorldWin32.exe
```

Hello World (Using C libraries and Linking with gcc)

In this example we will rewrite Hello World to use `printf(3)` from the C library and link using `gcc`. This has the advantage that going from Linux to Windows requires minimal source code changes and a slightly different assemble and link steps. In the Windows world this has the additional benefit that the linking step will be the same in the Windows command prompt and `cygwin`. There are several major changes:

1. The "hello, world" string now becomes the format string for `printf(3)` and therefore needs to be null terminated. This also means we do not need to explicitly specify its length anymore.
2. `gcc` expects the entry point for execution to be `main`
3. Microsoft will prefix functions using the `cdecl` calling convention with a underscore. So `main` and `printf` will become `_main` and `_printf` respectively in the Windows development environment.

```
global main
extern printf

section .data
    fmtStr: db 'hello, world',0xA,0

section .text
    main:

        sub     esp, 4           ; Allocate space on the stack for one 4 byte parameter

        lea     eax, [fmtStr]
        mov     [esp], eax      ; Arg1: pointer to format string
        call    printf          ; Call printf(3):
                                ;     int printf(const char *format, ...);

        add     esp, 4           ; Pop stack once

        ret
```

In order to assemble, link and run the program we need to do the following.

```
$ nasm -felf32 helloWorldgcc.asm
$ gcc helloWorldgcc.o -o helloWorldgcc
```

The Windows version with prefixed underscores:

```
global _main
extern _printf          ; Uncomment under Windows

section .data
    fmtStr: db 'hello, world',0xA,0

section .text
    _main:

        sub     esp, 4           ; Allocate space on the stack for one 4 byte parameter

        lea     eax, [fmtStr]
        mov     [esp], eax      ; Arg1: pointer to format string
        call    _printf         ; Call printf(3):
                                ;     int printf(const char *format, ...);

        add     esp, 4           ; Pop stack once

        ret
```

In order to assemble, link and run the program we need to do the following.

```
$ nasm -fwin32 helloWorldgcc.asm
$ gcc helloWorldgcc.o -o helloWorldgcc.exe
```

Floating Point

While integers are sufficient for some applications, it is often necessary to use the floating point coprocessor to manipulate numbers with fractional parts.

x87 Coprocessor

The original x86 family members had a separate math coprocessor that handled floating point arithmetic. The original coprocessor was the 8087, and all FPU's since have been dubbed "x87" chips. Later variants integrated the floating point unit (FPU) into the microprocessor itself. Having the capability to manage floating point numbers means a few things:

1. The microprocessor must have space to store floating point numbers
2. The microprocessor must have instructions to manipulate floating point numbers

The FPU, even when it is integrated into an x86 chip, is still called the "x87" section. For instance, literature on the subject will frequently call the FPU Register Stack the "x87 Stack", and the FPU operations will frequently be called the "x87 instruction set".

FPU Register Stack

The FPU has 8 registers, st0 to st7, formed into a stack. Numbers are pushed onto the stack from memory, and are popped off the stack back to memory. FPU instructions generally will pop the first two items off the stack, act on them, and push the answer back on to the top of the stack.

Floating point numbers may generally be either 32 bits long (C "float" type), or 64 bits long (C "double" type). However, in order to reduce round-off errors, the FPU stack registers are all 80 bits wide.

Most calling conventions return floating point values in the st0 register.

Examples

The following program (using NASM syntax) calculates the square root of 123.45.

```
global _start
section .data
    val: dq 123.45 ;declare quad word (double precision)
section .bss
    res: resq 1 ;reserve 1 quad word for result
section .text
    _start:
        fld qword [val] ;load value into st0
        fsqrt ;compute square root of st0 and store in st0
        fstp qword [res] ;store st0 in [res], and pop it off the x87 stack (leaving the x87 register stack empty again)
        ;end program
```

Essentially, programs that use the FPU load values onto the stack with FLD and its variants, perform operations on these values, then store them into memory with one of the forms of FST, most commonly FSTP when you're done with x87, to clean up the x87 stack as required by most calling conventions. Because the x87 stack can only be accessed by FPU instructions – you cannot write `mov eax, st0` – it is necessary to store values to memory if you want to print them, for example.

A more complex example that evaluates the Law of Cosines:

```
;; c^2 = a^2 + b^2 - cos(C)*2*a*b
;; C is stored in ang
global _start
section .data
    a: dq 4.56 ;length of side a
    b: dq 7.89 ;length of side b
    ang: dq 1.5 ;opposite angle to side c (around 85.94 degrees)
section .bss
    c: resq 1 ;the result – length of side c
section .text
    _start:
        fld qword [a] ;load a into st0
        fmul st0, st0 ;st0 = a * a = a^2
```

```

fld    qword [b]    ;load b into st0 (pushing the a^2 result up to st1)
fmul   st0, st0     ;st0 = b * b = b^2,    st1 = a^2

faddp                      ;add and pop, leaving st0 = old_st0 + old_st1 = a^2 + b^2. (st1 is freed / empty now)

fld    qword [ang]   ;load angle into st0. (st1 = a^2 + b^2 which we'll leave alone until later)
fcos                   ;st0 = cos(ang)

fmul   qword [a]     ;st0 = cos(ang) * a
fmul   qword [b]     ;st0 = cos(ang) * a * b
fadd   st0, st0       ;st0 = cos(ang) * a * b + cos(ang) * a * b = 2(cos(ang) * a * b)

fsubp   st1, st0      ;st1 = st1 - st0 = (a^2 + b^2) - (2 * a * b * cos(ang))
                        ;and pop st0

fsqrt                      ;take square root of st0 = c

fstp    qword [c]     ;store st0 in c and pop, leaving the x87 stack empty again – and we're done!

; don't forget to make an exit system call for your OS,
; or execution will fall off the end and decode whatever garbage bytes are next.
mov     eax, 1          ; __NR_exit
xor     ebx, ebx
int     0x80            ; i386 Linux sys_exit(0)
;end program

```

Floating-Point Instruction Set

You may notice that some of the instructions below differ from another in name by just one letter: a **P** appended to the end. This suffix signifies that in addition to performing the normal operation, they also **Pop** the x87 stack after execution is complete.

Original 8087 instructions

F2XM1, FABS, FADD, FADDP, FBLD, FBSTP, FCHS, FCLEX, FCOM, FCOMP, FCOMPP, FDECSTP, FDISI, FDIV, FDIVP, FDIVR, FDIVRP, FENI, FFREE, FIADD, FICOM, FICOMP, FIDIV, FIDIVR, FILD, FIMUL, FINCSTP, FINIT, FIST, FISTP, FISUB, FISUBR, FLD, FLD1, FLDCW, FLDENV, FLDENVW, FLDL2E, FLDL2T, FLDLG2, FLDLN2, FLDPI, FLDZ, FMUL, FMULP, FNCLEX, FNDISI, FNENI, FNINIT, FNOP, FNSAVE, FNSAVEW, FNSTCW, FNSTENV, FNSTENVW, FNSTSW, FPATAN, FPREM, FPTAN, FRNDINT, FRSTOR, FRSTORW, FSAVE, FSAVEW, FSCALE, FSQRT, FST, FSTCW, FSTENV, FSTENVW, FSTP, FSTSW, FSUB, FSUBP, FSUBR, FSUBRP, FTST, FWAIT, FXAM, FXCH, FXTRACT, FYL2X, FYL2XP1

Added in specific processors

Added with 80287

FSETPM

Added with 80387

FCOS, FLDENV, FNSAVED, FNSTENV, FPREM1, FRSTORD, FSAVED, FSIN, FSINCOS, FSTENV, FUCOM, FUCOMP, FUCOMPP

Added with Pentium Pro

FCMOVB, FCMOVBE, FCMOVE, FCMOVNB, FCMOVNBE, FCMOVNE, FCMOVNU, FCMOVU, FCOMI, FCOMIP, FUCOMI, FUCOMIP, FXRSTOR, FXSAVE

Added with SSE

FXRSTOR, FXSAVE

These are also supported on later Pentium IIs which do not contain SSE support

Added with SSE3

FISTTP (x87 to integer conversion with truncation regardless of status word)

Undocumented instructions

FFREEP performs FFREE ST(i) and pop stack

Further Reading

- [X86 Disassembly/Floating Point Numbers](#)
- [Floating Point](#)

MMX

MMX is a supplemental instruction set introduced by Intel in 1996. Most of the new instructions are "single instruction, multiple data" (SIMD), meaning that single instructions work with multiple pieces of data in parallel.

MMX has a few problems, though: instructions run slightly slower than the regular arithmetic instructions, the Floating Point Unit (FPU) can't be used when the MMX registers are in use, and MMX registers use saturation arithmetic.

Saturation Arithmetic

In an 8-bit grayscale picture, 255 is the value for pure white, and 0 is the value for pure black. In a regular register (AX, BX, CX ...) if we add one to white, we get black! This is because the regular registers "roll-over" to the next value. MMX registers get around this by a technique called "Saturation Arithmetic". In saturation arithmetic, the value of the register never rolls over to 0 again. This means that in the MMX world, we have the following equations:

```
255 + 100 = 255
200 + 100 = 255
0 - 100 = 0;
99 - 100 = 0;
```

This may seem counter-intuitive at first to people who are used to their registers rolling over, but it makes sense in some situations: if we try to make white brighter, it shouldn't become black.

Single Instruction Multiple Data (SIMD) Instructions

The MMX registers are 64 bits wide, but can be broken down as follows:

```
2 32 bit values
4 16 bit values
8 8 bit values
```

The MMX registers cannot easily be used for 64 bit arithmetic. Let's say that we have 4 bytes loaded in an MMX register: 10, 25, 128, 255. We have them arranged as such:

```
MM0: | 10 | 25 | 128 | 255 |
```

And we do the following pseudo code operation:

```
MM0 + 10
```

We would get the following result:

```
MM0: | 10+10 | 25+10 | 128+10 | 255+10 | = | 20 | 35 | 138 | 255 |
```

Remember that our arithmetic "saturates" in the last box, so the value doesn't go over 255.

Using MMX, we are essentially performing 4 additions in the time it takes to perform 1 addition using the regular registers, using 4 times fewer instructions.

MMX Registers

There are 8 64-bit MMX registers. To avoid having to add new registers, they were made to overlap with the FPU stack register. This means that **the MMX instructions and the FPU instructions cannot be used simultaneously**. MMX registers are addressed directly, and do not need to be accessed by pushing and popping in the same way as the FPU registers.

```
MM7 MM6 MM5 MM4 MM3 MM2 MM1 MM0
```

These registers correspond to the same numbered FPU registers on the FPU stack.

Usually when you initiate an assembly block in your code that contains MMX instructions, the CPU automatically will disallow floating point instructions. To re-allow FPU operations you must end all MMX code with `emms`.

The following is a program for GNU AS and GCC which copies 8 bytes from one variable to another and prints the result.

Assembler portion

```
.globl copy_memory8
.type copy_memory8, @function
copy_memory8:
    pushl %ebp
    mov %esp, %ebp
    mov 8(%ebp), %eax
    movq (%eax), %mm0
    mov 12(%ebp), %eax
    movq %mm0, (%eax)
    popl %ebp
    emms
    ret
.size copy_memory8,.-copy_memory8
```

C portion

```
#include <stdio.h>

void copy_memory8(void * a, void * b);

int main () {
    long long b = 0xffffffff00000000;
    long long c = 0x00000000ffffffff;
    printf("%lld == %lld\n", b, c);
    copy_memory8(&b, &c);
    printf("%lld == %lld\n", b, c);
    return 0;
}
```

MMX Instruction Set

Several suffixes are used to indicate what data size the instruction operates on:

- **Byte** (8 bits)
- **Word** (16 bits)
- **Double word** (32 bits)
- **Quad word** (64 bits)

The signedness of the operation is also signified by the suffix: **US** for unsigned and **S** for signed.

For example, `PSUBUSB` subtracts unsigned bytes, while `PSUBSD` subtracts signed double words.

MMX defined over 40 new instructions, listed below.

EMMS, MOVD, MOVQ, PACKSSDW, PACKSSWB, PACKUSWB, PADDB, PADDD, PADDSB, PADDSW, PADDUSB, PADDUSW, PADDW, PAND, PANDN, PCMPEQB, PCMPEQD, PCMPEQW, PCMPGTB, PCMPGTD, PCMPGTW, PMADDWD, PMULHW, PMULLW, POR, PSLLD, PSLLQ, PSLLW, PSRAD, PSRAW, PSRLD, PSRLQ, PSRLW, PSUBB, PSUBD, PSUBSB, PSUBSW, PSUBUSB, PSUBUSW, PSUBW, PUNPCKHBW, PUNPCKHDQ, PUNPCKHWD, PUNPCKLBW, PUNPCKLDQ, PUNPCKLWD, PXOR

SSE

SSE stands for **Streaming SIMD Extensions**. It is essentially the floating-point equivalent of the [MMX instructions](#). The SSE registers are 128 bits, and can be used to perform operations on a variety of data sizes and types. Unlike MMX, the SSE registers do not overlap with the floating point stack.

Registers

SSE, introduced by Intel in 1999 with the Pentium III, creates eight new 128-bit registers:

```
XMM0 XMM1 XMM2 XMM3 XMM4 XMM5 XMM6 XMM7
```

Originally, an SSE register could only be used as four 32-bit single precision floating point numbers (the equivalent of a `float` in C). SSE2 expanded the capabilities of the XMM registers, so they can now be used as:

- 2 64-bit floating points (double precision)
- 2 64-bit integers
- 4 32-bit floating points (single-precision)
- 4 32-bit integers
- 8 16-bit integers
- 16 8-bit characters (bytes)

Data movement examples

The following program (using [NASM syntax](#)) performs data movements using SIMD instructions.

```
;
; nasm -felf32 -g sseMove.asm
; ld -g sseMove.o
;
global _start

section .data
    align 16
    v1: dd 1.1, 2.2, 3.3, 4.4 ; Four Single precision floats 32 bits each
    v1dp: dq 1.1, 2.2 ; Two Double precision floats 64 bits each
    v2: dd 5.5, 6.6, 7.7, 8.8
    v2s1: dd 5.5, 6.6, 7.7, -8.8
    v2s2: dd 5.5, 6.6, -7.7, -8.8
    v2s3: dd 5.5, -6.6, -7.7, -8.8
    v2s4: dd -5.5, -6.6, -7.7, -8.8
    num1: dd 1.2
    v3: dd 1.2, 2.3, 4.5, 6.7 ; No longer 16 byte aligned
    v3dp: dq 1.2, 2.3 ; No longer 16 byte aligned

section .bss
    mask1: resd 1
    mask2: resd 1
    mask3: resd 1
    mask4: resd 1

section .text
_start:

;
; op dst, src
;
;
; SSE
;
; Using movaps since vectors are 16 byte aligned
movaps xmm0, [v1] ; Move four 32-bit(single precision) floats to xmm0
movaps xmm1, [v2]
movups xmm2, [v3] ; Need to use movups since v3 is not 16 byte aligned
;movaps xmm3, [v3] ; This would seg fault if uncommented
```

```

movss    xmm3, [num1]    ; Move 32-bit float num1 to the least significant element of xmm3
movss    xmm3, [v3]     ; Move first 32-bit float of v3 to the least significant element of xmm3
movlps   xmm4, [v3]     ; Move 64-bits(two single precision floats) from memory to the lower 64-bit elements of xmm4
movhps   xmm4, [v2]     ; Move 64-bits(two single precision floats) from memory to the higher 64-bit elements of
xmm4

; Source and destination for movhlps and movlhps must be xmm registers
movhlps  xmm5, xmm4     ; Transfers the higher 64-bits of the source xmm4 to the lower 64-bits of the destination
xmm5
movlhps  xmm5, xmm4     ; Transfers the lower 64-bits of the source xmm4 to the higher 64-bits of the destination
xmm5

movaps    xmm6, [v2s1]
movmskps  eax, xmm6     ; Extract the sign bits from four 32-bits floats in xmm6 and create 4 bit mask in eax
mov [mask1], eax        ; Should be 8
movaps    xmm6, [v2s2]
movmskps  eax, xmm6     ; Extract the sign bits from four 32-bits floats in xmm6 and create 4 bit mask in eax
mov [mask2], eax        ; Should be 12
movaps    xmm6, [v2s3]
movmskps  eax, xmm6     ; Extract the sign bits from four 32-bits floats in xmm6 and create 4 bit mask in eax
mov [mask3], eax        ; Should be 14
movaps    xmm6, [v2s4]
movmskps  eax, xmm6     ; Extract the sign bits from four 32-bits floats in xmm6 and create 4 bit mask in eax
mov [mask4], eax        ; Should be 15

;
; SSE2
;
movapd    xmm6, [v1dp]   ; Move two 64-bit(double precision) floats to xmm6, using movapd since vector is 16 byte
aligned
; Next two instruction should have equivalent results to movapd xmm6, [v1dp]
movhpd    xmm6, [v1dp+8] ; Move a 64-bit(double precision) float into the higher 64-bit elements of xmm6
movlpd    xmm6, [v1dp]   ; Move a 64-bit(double precision) float into the lower 64-bit elements of xmm6
movupd    xmm6, [v3dp]   ; Move two 64-bit floats to xmm6, using movupd since vector is not 16 byte aligned

```

Arithmetic example using packed singles

The following program (using NASM syntax) performs a few SIMD operations on some numbers.

```

global _start

section .data
v1: dd 1.1, 2.2, 3.3, 4.4 ;first set of 4 numbers
v2: dd 5.5, 6.6, 7.7, 8.8 ;second set

section .bss
v3: resd 4 ;result

section .text
_start:

movups    xmm0, [v1]     ;load v1 into xmm0
movups    xmm1, [v2]     ;load v2 into xmm1

addps     xmm0, xmm1     ;add the 4 numbers in xmm1 (from v2) to the 4 numbers in xmm0 (from v1), store in xmm0. for
the first float the result will be 5.5+1.1=6.6
mulps     xmm0, xmm1     ;multiply the four numbers in xmm1 (from v2, unchanged) with the results from the previous
calculation (in xmm0), store in xmm0. for the first float the result will be 5.5*6.6=36.3
subps     xmm0, xmm1     ;subtract the four numbers in v2 (in xmm1, still unchanged) from result from previous
calculation (in xmm1). for the first float, the result will be 36.3-5.5=30.8

movups    [v3], xmm0     ;store v1 in v3

;end program
ret

```

The result values should be:

```

30.800    51.480    77.000    107.360

```

Using the GNU toolchain, you can debug and single-step like this:

```

% nasm -felf32 -g ssedemo.asm
% ld -g ssedemo.o
% gdb -q ./a.out
Reading symbols from a.out...done.
(gdb) break _start
Breakpoint 1 at 0x8048080
(gdb) r
Starting program: a.out

Breakpoint 1, 0x8048080 in _start ()
(gdb) disass
Dump of assembler code for function _start:

```

```

=> 0x08048080 <+0>: movups 0x80490a0,%xmm0
0x08048087 <+7>: movups 0x80490b0,%xmm1
0x0804808e <+14>: addps %xmm1,%xmm0
0x08048091 <+17>: mulps %xmm1,%xmm0
0x08048094 <+20>: subps %xmm1,%xmm0
0x08048097 <+23>: movups %xmm0,0x80490c0
End of assembler dump.
(gdb) stepi
0x08048087 in _start ()
(gdb)
0x0804808e in _start ()
(gdb) p %xmm0
$1 = {v4_float = {1.10000002, 2.20000005, 3.29999995, 4.4000001}, v2_double = {3.6000008549541236,
921.60022034645078}, v16_int8 = {-51, -52, -116, 63,
-51, -52, 12, 64, 51, 51, 83, 64, -51, -52, -116, 64}, v8_int16 = {-13107, 16268, -13107, 16396, 13107, 16467,
-13107, 16524}, v4_int32 = {1066192077,
1074580685, 1079194419, 1082969293}, v2_int64 = {4615288900054469837, 4651317697086436147}, uint128 =
0x408cccd40533333400cccd3f8cccd}
(gdb) x/4f &v1
0x80490a0 <v1>: 1.10000002 2.20000005 3.29999995 4.4000001
(gdb) stepi
0x08048091 in _start ()
(gdb) p %xmm0
$2 = {v4_float = {6.5999999, 8.80000019, 11, 13.2000008}, v2_double = {235929.65665283203, 5033169.0185546875},
v16_int8 = {51, 51, -45, 64, -51, -52, 12,
65, 0, 0, 48, 65, 52, 51, 83, 65}, v8_int16 = {13107, 16595, -13107, 16652, 0, 16688, 13108, 16723}, v4_int32 =
{1087583027, 1091357901, 1093664768,
1095971636}, v2_int64 = {4687346494113788723, 4707162335057281024}, uint128 =
0x4153333441300000410cccd40d33333}
(gdb)

```

Debugger commands explained

break

In this case, sets a breakpoint at a given label

stepi

Steps one instruction forward in the program

p

short for **print**, prints a given register or variable. Registers are prefixed by \$ in GDB.

x

short for **examine**, examines a given memory address. The "/4f" means "4 floats" (floats in GDB are 32-bits). You can use c for chars, x for hexadecimal and any other number instead of 4 of course. The "&" takes the address of v1, as in C.

Shuffling example using shufps

shufps IMM8, arg1, arg2

GAS Syntax

shufps arg2, arg1, IMM8

Intel Syntax

shufps can be used to shuffle packed single-precision floats. The instruction takes three parameters, **arg1** an xmm register, **arg2** an xmm or a 128-bit memory location and **IMM8** an 8-bit immediate control byte. shufps will take two elements each from **arg1** and **arg2**, copying the elements to **arg2**. The lower two elements will come from **arg1** and the higher two elements from **arg2**.

IMM8 control byte description

IMM8 control byte is split into four group of bit fields that control the output into **arg2** as follows:

1. IMM8[1:0] specifies which element of **arg1** ends up in the least significant element of **arg2**:

IMM8[1:0]	Description
00b	Copy to the least significant element
01b	Copy to the second element
10b	Copy to the third element
11b	Copy to the most significant element

2. IMM8[3:2] specifies which element of **arg1** ends up in the second element of **arg2**:

IMM8[3:2]	Description
00b	Copy to the least significant element
01b	Copy to the second element
10b	Copy to the third element
11b	Copy to the most significant element

3. IMM8[5:4] specifies which element of arg2 ends up in the third element of arg2:

IMM8[5:4]	Description
00b	Copy to the least significant element
01b	Copy to the second element
10b	Copy to the third element
11b	Copy to the most significant element

4. IMM8[7:6] specifies which element of arg2 ends up in the most significant element of arg2:

IMM8[7:6]	Description
00b	Copy to the least significant element
01b	Copy to the second element
10b	Copy to the third element
11b	Copy to the most significant element

IMM8 Example

Consider the byte 0x1B:

Byte value	0x1B							
Nibble value	0x1				0xB			
2-bit integer (decimal) value	0		1		2		3	
Bit value	0	0	0	1	1	0	1	1
Bit number (0 being LSB)	7	6	5	4	3	2	1	0

The 2-bit values shown above are used to determine which elements are copied to arg2. Bits 7-4 are "indexes" into arg2, and bits 3-0 are "indexes" into the arg1.

- Since bits 7-6 are 0, the least significant element of arg2 is copied to the most significant elements of arg2, bits 127-96.
- Since bits 5-4 are 1, the second element of arg2 is copied to third element of arg2, bits 95-64.
- Since bits 3-2 are 2, the third element of arg1 is copied to the second element of arg2, bits 63-32.
- Since bits 0-1 are 3, the fourth element of arg1 is copied to the least significant elements of arg2, bits (31-0).

Note that since the first and second arguments are equal in the following example, the mask 0x1B will effectively reverse the order of the floats in the XMM register, since the 2-bit integers are 0, 1, 2, 3. Had it been 3, 2, 1, 0 (0xE4) it would be a no-op. Had it been 0, 0, 0, 0 (0x00) it would be a broadcast of the least significant 32 bits.

Example

```
.data
.align 16
v1: .float 1.1, 2.2, 3.3, 4.4
v2: .float 5.5, 6.6, 7.7, 8.8
v3: .float 0, 0, 0, 0

.text
.global _start
_start:
    movaps v1,%xmm0      # load v1 into xmm0 to xmm6
    movaps v1,%xmm1      # using movaps since v1 is 16 byte aligned
```

```

movaps    v1,%xmm2
movaps    v1,%xmm3
movaps    v1,%xmm4
movaps    v1,%xmm5
movaps    v1,%xmm6

shufps    $0x1b, %xmm0, %xmm0 # reverse order of the 4 floats
shufps    $0x00, %xmm1, %xmm1 # Broadcast least significant element to all elements
shufps    $0x55, %xmm2, %xmm2 # Broadcast second element to all elements
shufps    $0xAA, %xmm3, %xmm3 # Broadcast third element to all elements
shufps    $0xFF, %xmm4, %xmm4 # Broadcast most significant element to all elements
shufps    $0x39, %xmm5, %xmm5 # Rotate elements right
shufps    $0x93, %xmm6, %xmm6 # Rotate elements left

movups    %xmm0, v3          #store v1 in v3
ret

```

Using GAS to build an ELF executable

```

as -g shufps.S -o shufps.o
ld -g shufps.o

```

Text Processing Instructions

SSE 4.2 adds four string text processing instructions PCMPISTRI, PCMPISTRM, PCMPESTRI and PCMPESTRM. These instructions take three parameters, `arg1` an xmm register, `arg2` an xmm or a 128-bit memory location and `IMM8` an 8-bit immediate control byte. These instructions will perform arithmetic comparison between the packed contents of `arg1` and `arg2`. `IMM8` specifies the format of the input/output as well as the operation of two intermediate stages of processing. The results of stage 1 and stage 2 of intermediate processing will be referred to as `IntRes1` and `IntRes2` respectively. These instructions also provide additional information about the result through overload use of the arithmetic flags(AF, CF, OF, PF, SF and ZF).

The instructions proceed in multiple steps:

1. `arg1` and `arg2` are compared
2. An aggregation operation is applied to the result of the comparison with the result flowing into `IntRes1`
3. An optional negation is performed with the result flowing into `IntRes2`
4. An output in the form of an index(in ECX) or a mask(in XMM0) is produced

IMM8 control byte description

IMM8 control byte is split into four group of bit fields that control the following settings:

1. `IMM8[1:0]` specifies the format of the 128-bit source data(`arg1` and `arg2`):

IMM8[1:0]	Description
00b	unsigned bytes(16 packed unsigned bytes)
01b	unsigned words(8 packed unsigned words)
10b	signed bytes(16 packed signed bytes)
11b	signed words(8 packed signed words)

2. `IMM8[3:2]` specifies the aggregation operation whose result will be placed in intermediate result 1, which we will refer to as `IntRes1`. The size of `IntRes1` will depend on the format of the source data, 16-bit for packed bytes and 8-bit for packed words:

IMM8[3:2]	Description
00b	Equal Any , arg1 is a character set, arg2 is the string to search in. IntRes1[i] is set to 1 if arg2[i] is in the set represented by arg1: <pre> arg1 = "aeiou" arg2 = "Example string 1" IntRes1 = 0010001000010000 </pre>
01b	Ranges , arg1 is a set of character ranges i.e. "09az" means all characters from 0 to 9 and from a to z., arg2 is the string to search over. IntRes1[i] is set to 1 if arg[i] is in any of the ranges represented by arg1: <pre> arg1 = "09az" arg2 = "Testing 1 2 3, T" IntRes1 = 0111111010101000 </pre>
10b	Equal Each , arg1 is string one and arg2 is string two. IntRes1[i] is set to 1 if arg1[i] == arg2[i]: <pre> arg1 = "The quick brown " arg2 = "The quack green " IntRes1 = 1111110111010011 </pre>
11b	Equal Ordered , arg1 is a substring string to search for, arg2 is the string to search within. IntRes1[i] is set to 1 if the substring arg1 can be found at position arg2[i]: <pre> arg1 = "he" arg2 = ", he helped her " IntRes1 = 0010010000001000 </pre>

3. IMM8[5:4] specifies the polarity or the processing of IntRes1, into intermediate result 2, which will be referred to as IntRes2:

IMM8[5:4]	Description
00b	Positive Polarity IntRes2 = IntRes1
01b	Negative Polarity IntRes2 = -1 XOR IntRes1
10b	Masked Positive IntRes2 = IntRes1
11b	Masked Negative IntRes2 = IntRes1 if reg/mem[i] is invalid else ~IntRes1

4. IMM8[6] specifies the output selection, or how IntRes2 will be processed into the output. For PCMPSTR and PCMPISTR, the output is an index into the data currently referenced by arg2:

IMM8[6]	Description
0b	Least Significant Index ECX contains the least significant set bit in IntRes2
1b	Most Significant Index ECX contains the most significant set bit in IntRes2

5. For PCMPSTRM and PCMPISTRM, the output is a mask reflecting all the set bits in IntRes2:

IMM8[6]	Description
0b	Least Significant Index Bit Mask, the least significant bits of XMM0 contain the IntRes2 16(8) bit mask. XMM0 is zero extended to 128-bits.
1b	Most Significant Index Byte/Word Mask, XMM0 contains IntRes2 expanded into byte/word mask

6. IMM8[7] should be set to zero since it has no designed meaning.

The Four Instructions

pcmpistri IMM8, arg2, arg1

GAS Syntax

pcmpistri arg1, arg2, IMM8

Intel Syntax

PCMPISTRI, Packed Compare Implicit Length Strings, Return Index. Compares strings of implicit length and generates index in ECX.

Operands

arg1

- XMM Register
- Memory

arg2

- XMM Register

IMM8

- 8-bit Immediate value

Modified flags

1. CF is reset if IntRes2 is zero, set otherwise
2. ZF is set if a null terminating character is found in *arg2*, reset otherwise
3. SF is set if a null terminating character is found in *arg1*, reset otherwise
4. OF is set to IntRes2[0]
5. AF is reset
6. PF is reset

Example

```
;
; nasm -felf32 -g sse4_2StrPcmpistri.asm -l sse4_2StrPcmpistri.lst
; gcc -o sse4_2StrPcmpistri sse4_2StrPcmpistri.o
;
global main

extern printf
extern strlen
extern strcmp

section .data
    align 4
    ;
    ; Fill buf1 with a repeating pattern of ABCD
    ;
    buf1:        times 10 dd 0x44434241
    s1:          db "This is a string", 0
    s2:          db "This is a string slightly different string", 0
    s3:          db "This is a str", 0
    fmtStr1:     db "String: %s len: %d", 0x0A, 0
    fmtStr1b:    db "strlen(3): String: %s len: %d", 0x0A, 0
    fmtStr2:     db "s1: %s= and s2: %s= compare: %d", 0x0A, 0
    fmtStr2b:    db "strcmp(3): s1: %s= and s2: %s= compare: %d", 0x0A, 0

;
; Functions will follow the cdecl call convention
;
section .text
    main:        ; Using main since we are using gcc to link

    sub esp, -16 ; 16 byte align the stack
    sub esp, 16  ; space for four 4 byte parameters

    ;
    ; Null terminate buf1, make it proper C string, length is now 39
    ;
    mov [buf1+39], byte 0x00

    lea eax, [buf1]
    mov [esp], eax ; Arg1: pointer of string to calculate the length of
    mov ebx, eax   ; Save pointer in ebx since we will use it again
    call strlenSSE42
    mov edx, eax   ; Copy length of arg1 into edx

    mov [esp+8], edx ; Arg3: length of string
    mov [esp+4], ebx ; Arg2: pointer to string
    lea eax, [fmtStr1]
    mov [esp], eax ; Arg1: pointer to format string
    call printf    ; Call printf(3):
                  ; int printf(const char *format, ...);

    lea eax, [buf1]
```

```

mov [esp], eax ; Arg1: pointer of string to calculate the length of
mov ebx, eax ; Save pointer in ebx since we will use it again
call strlen ; Call strlen(3):
; size_t strlen(const char *s);
mov edx, eax ; Copy length of arg1 into edx

mov [esp+8], edx ; Arg3: length of string
mov [esp+4], ebx ; Arg2: pointer to string
lea eax, [fmtStr1b]
mov [esp], eax ; Arg1: pointer to format string
call printf ; Call printf(3):
; int printf(const char *format, ...);

lea eax, [s2]
mov [esp+4], eax ; Arg2: pointer to second string to compare
lea eax, [s1]
mov [esp], eax ; Arg1: pointer to first string to compare
call strcmpSSE42

mov [esp+12], eax ; Arg4: result from strcmpSSE42
lea eax, [s2]
mov [esp+8], eax ; Arg3: pointer to second string
lea eax, [s1]
mov [esp+4], eax ; Arg2: pointer to first string
lea eax, [fmtStr2]
mov [esp], eax ; Arg1: pointer to format string
call printf

lea eax, [s2]
mov [esp+4], eax ; Arg2: pointer to second string to compare
lea eax, [s1]
mov [esp], eax ; Arg1: pointer to first string to compare
call strcmp ; Call strcmp(3):
; int strcmp(const char *s1, const char *s2);

mov [esp+12], eax ; Arg4: result from strcmpSSE42
lea eax, [s2]
mov [esp+8], eax ; Arg3: pointer to second string
lea eax, [s1]
mov [esp+4], eax ; Arg2: pointer to first string
lea eax, [fmtStr2b]
mov [esp], eax ; Arg1: pointer to format string
call printf

lea eax, [s3]
mov [esp+4], eax ; Arg2: pointer to second string to compare
lea eax, [s1]
mov [esp], eax ; Arg1: pointer to first string to compare
call strcmpSSE42

mov [esp+12], eax ; Arg4: result from strcmpSSE42
lea eax, [s3]
mov [esp+8], eax ; Arg3: pointer to second string
lea eax, [s1]
mov [esp+4], eax ; Arg2: pointer to first string
lea eax, [fmtStr2]
mov [esp], eax ; Arg1: pointer to format string
call printf

lea eax, [s3]
mov [esp+4], eax ; Arg2: pointer to second string to compare
lea eax, [s1]
mov [esp], eax ; Arg1: pointer to first string to compare
call strcmp ; Call strcmp(3):
; int strcmp(const char *s1, const char *s2);

mov [esp+12], eax ; Arg4: result from strcmpSSE42
lea eax, [s3]
mov [esp+8], eax ; Arg3: pointer to second string
lea eax, [s1]
mov [esp+4], eax ; Arg2: pointer to first string
lea eax, [fmtStr2b]
mov [esp], eax ; Arg1: pointer to format string
call printf

call exit

;
; size_t strlen(const char *s);
;
;
strlenSSE42:
push ebp
mov ebp, esp

mov edx, [ebp+8] ; Arg1: copy s(pointer to string) to edx
;
; We are looking for null terminating char, so set xmm0 to zero
;
pxor xmm0, xmm0
mov eax, -16 ; Avoid extra jump in main loop

strlenLoop:
add eax, 16
;
; IMM8[1:0] = 00b
; Src data is unsigned bytes(16 packed unsigned bytes)

```

```

; IMM8[3:2] = 10b
; We are using Equal Each aggregation
; IMM8[5:4] = 00b
; Positive Polarity, IntRes2 = IntRes1
; IMM8[6] = 0b
; ECX contains the least significant set bit in IntRes2
;
pcmpistri xmm0,[edx+eax], 0001000b
;
; Loop while ZF != 0, which means none of bytes pointed to by edx+eax
; are zero.
;
jnz strlenLoop

;
; ecx will contain the offset from edx+eax where the first null
; terminating character was found.
;
add eax, ecx
pop ebp
ret

;
; int strcmp(const char *s1, const char *s2);
;
strcmpSSE42:
push ebp
mov ebp, esp

mov eax, [ebp+8] ; Arg1: copy s1(pointer to string) to eax
mov edx, [ebp+12] ; Arg2: copy s2(pointer to string) to edx
;
; Subtract s2(edx) from s1(eax). This admittedly looks odd, but we
; can now use edx to index into s1 and s2. As we adjust edx to move
; forward into s2, we can then add edx to eax and this will give us
; the comparable offset into s1 i.e. if we take edx + 16 then:
;
;     edx      = edx + 16      = edx + 16
;     eax+edx = eax -edx + edx + 16 = eax + 16
;
; therefore edx points to s2 + 16 and eax + edx points to s1 + 16.
; We thus only need one index, convoluted but effective.
;
sub eax, edx
sub edx, 16 ; Avoid extra jump in main loop

strcmpLoop:
add edx, 16
movdqu xmm0, [edx]
;
; IMM8[1:0] = 00b
; Src data is unsigned bytes(16 packed unsigned bytes)
; IMM8[3:2] = 10b
; We are using Equal Each aggregation
; IMM8[5:4] = 01b
; Negative Polarity, IntRes2 = -1 XOR IntRes1
; IMM8[6] = 0b
; ECX contains the least significant set bit in IntRes2
;
pcmpistri xmm0, [edx+eax], 0011000b
;
; Loop while ZF=0 and CF=0:
;
; 1) We find a null in s1(edx+eax) ZF=1
; 2) We find a char that does not match CF=1
;
ja strcmpLoop

;
; Jump if CF=1, we found a mismatched char
;
jc strcmpDiff

;
; We terminated loop due to a null character i.e. CF=0 and ZF=1
;
xor eax, eax ; They are equal so return zero
jmp exitStrcmp

strcmpDiff:
add eax, edx ; Set offset into s1 to match s2
;
; ecx is offset from current poition where two strings do not match,
; so copy the respective non-matching byte into eax and edx and fill
; in remaining bits w/ zero.
;
movzx eax, byte[eax+ecx]
movzx edx, byte[edx+ecx]
;
; If s1 is less than s2 return integer less than zero, otherwise return
; integer greater than zero.
;
sub eax, edx

exitStrcmp:
pop ebp
ret

```

```

exit:
        ;
        ; Call exit(3) syscall
        ; void exit(int status)
        ;
mov ebx, 0      ; Arg one: the status
mov eax, 1      ; Syscall number:
int     0x80

```

Expected output:

```

String: ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD len: 39
strlen(3): String: ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD len: 39
s1: =This is a string= and s2: =This is a string slightly different string= compare: -32
strcmp(3): s1: =This is a string= and s2: =This is a string slightly different string= compare: -32
s1: =This is a string= and s2: =This is a str= compare: 105
strcmp(3): s1: =This is a string= and s2: =This is a str= compare: 105

```

pcmpistrm IMM8, arg2, arg1
pcmpistrm arg1, arg2, IMM8

GAS Syntax

Intel Syntax

PCMPISTRM, Packed Compare Implicit Length Strings, Return Mask. Compares strings of implicit length and generates a mask stored in XMM0.

Operands

arg1

- XMM Register

arg2

- XMM Register
- Memory

IMM8

- 8-bit Immediate value

Modified flags

1. CF is reset if IntRes2 is zero, set otherwise
2. ZF is set if a null terminating character is found in arg2, reset otherwise
3. SF is set if a null terminating character is found in arg2, reset otherwise
4. OF is set to IntRes2[0]
5. AF is reset
6. PF is reset

pcmpestri IMM8, arg2, arg1
pcmpestri arg1, arg2, IMM8

GAS Syntax

Intel Syntax

PCMPESTRI, Packed Compare Explicit Length Strings, Return Index. Compares strings of explicit length and generates index in ECX.

Operands

arg1

- XMM Register

arg2

- XMM Register
- Memory

IMM8

- 8-bit Immediate value

Implicit Operands

- *EAX* holds the length of *arg1*
- *EDX* holds the length of *arg2*

Modified flags

1. CF is reset if *IntRes2* is zero, set otherwise
2. ZF is set if *EDX* is < 16(for bytes) or 8(for words), reset otherwise
3. SF is set if *EAX* is < 16(for bytes) or 8(for words), reset otherwise
4. OF is set to *IntRes2*[0]
5. AF is reset
6. PF is reset

pcmpestrm IMM8, *arg2*, *arg1*

GAS Syntax

pcmpestrm *arg1*, *arg2*, IMM8

Intel Syntax

PCMPESTRM, Packed Compare Explicit Length Strings, Return Mask. Compares strings of explicit length and generates a mask stored in XMM0.

Operands

arg1

- XMM Register

arg2

- XMM Register
- Memory

IMM8

- 8-bit Immediate value

Implicit Operands

- *EAX* holds the length of *arg1*
- *EDX* holds the length of *arg2*

Modified flags

1. CF is reset if *IntRes2* is zero, set otherwise
2. ZF is set if *EDX* is < 16(for bytes) or 8(for words), reset otherwise
3. SF is set if *EAX* is < 16(for bytes) or 8(for words), reset otherwise
4. OF is set to *IntRes2*[0]
5. AF is reset

6. PF is reset

SSE Instruction Set

There are literally hundreds of SSE instructions, some of which are capable of much more than simple SIMD arithmetic. For more in-depth references take a look at the [resources](#) chapter of this book.

You may notice that many floating point SSE instructions end with something like PS or SD. These suffixes differentiate between different versions of the operation. The first letter describes whether the instruction should be **P**acked or **S**calar. Packed operations are applied to every member of the register, while scalar operations are applied to only the first value. For example, in pseudo-code, a packed add would be executed as:

```
v1[0] = v1[0] + v2[0]
v1[1] = v1[1] + v2[1]
v1[2] = v1[2] + v2[2]
v1[3] = v1[3] + v2[3]
```

While a scalar add would only be:

$$v1[0] = v1[0] + v2[0]$$

The second letter refers to the data size: either **Single** or **Double**. This simply tells the processor whether to use the register as four 32-bit floats or two 64-bit doubles, respectively.

SSE: Added with Pentium III

Floating-point Instructions:

ADDPs, ADDSS, CMPPS, CMPSS, COMISS, CVTPI2PS, CVTPS2PI, CVTSI2SS, CVTSS2SI, CVTTPS2PI, CVTTSS2SI, DIVPS, DIVSS, LDMXCSR, MAXPS, MAXSS, MINPS, MINSS, MOVAPS, MOVHLPs, MOVHPS, MOVLHPS, MOVLPS, MOVMSKPS, MOVNTPS, MOVSS, MOVUPS, MULPS, MULSS, RCPPS, RCPSS, RSQRTPS, RSQRTSS, SHUFPS, SQRTPS, SQRTSS, STMXCSR, SUBPS, SUBSS, UCOMISS, UNPCKHPS, UNPCKLPS

Integer Instructions:

ANDNPS, ANDPS, ORPS, PAVGB, PAVGW, PEXTRW, PINSRW, PMAXSW, PMAXUB, PMINSW, PMINUB, PMOVMASKB, PMULHUW, PSADBW, PSHUFW, XORPS

SSE2: Added with Pentium 4

Floating-point Instructions:

ADDPD, ADDSD, ANDNPD, ANDPD, CMPPD, CMPSD*, COMISD, CVTDQ2PD, CVTDQ2PS, CVTPD2DQ, CVTPD2PI, CVTPD2PS, CVTPI2PD, CVTPS2DQ, CVTPS2PD, CVTSD2SI, CVTSD2SS, CVTSI2SD, CVTSS2SD, CVTTPD2DQ, CVTTPD2PI, CVTTPS2DQ, CVTTSD2SI, DIVPD, DIVSD, MAXPD, MAXSD, MINPD, MINSD, MOVAPD, MOVHPD, MOVLPD, MOVMSKPD, MOVSD*, MOVUPD, MULPD, MULSD, ORPD, SHUFPD, SQRTPD, SQRTSD, SUBPD, SUBSD, UCOMISD, UNPCKHPD, UNPCKLPD, XORPD

* CMPSD and MOVSD have the same name as the string instruction mnemonics CMPSD (CMPS) and MOVSD (MOVS); however, the former refer to scalar double-precision floating-points whereas the latter refer to doubleword strings.

Integer Instructions:

MOVDQ2Q, MOVDQA, MOVDQU, MOVQ2DQ, PADDQ, PSUBQ, PMULUDQ, PSHUFW, PSHUFLW, PSHUFD, PSLLDQ, PSRLDQ, PUNPCKHQDQ, PUNPCKLQDQ

SSE3: Added with later Pentium 4

ADDSDUP, ADDSUBPD, ADDSUBPS, HADDDPD, HADDPS, HSUBPD, HSUBPS, MOVDDUP, MOVSHDUP, MOVSLDUP

SSSE3: Added with Xeon 5100 and early Core 2

PSIGNW, PSIGND, PSIGNB, PSHUFB, PMULHSW, PMADDUBSW, PHSUBW, PHSUBSW, PHSUBD, PHADDW, PHADDSW, PHADDD, PALIGNR, PABSW, PABSD, PABSB

SSE4

SSE4.1: Added with later Core 2

MPSADBW, PHMINPOSUW, PMULLD, PMULDQ, DPPS, DPPD, BLENDPS, BLENDPD, BLENDVPS, BLENDVPD, PBLENDVB, PBLENDW, PMINSB, PMAXS, PMINUW, PMAXUW, PMINUD, PMAXUD, PMINS, PMAXS, ROUNDPS, ROUNDSS, ROUNDPD, ROUNDSD, INSERTPS, PINSRB, PINSRD, PINSRQ, EXTRACTPS, PEXTRB, PEXTRW, PEXTRD, PEXTRQ, PMOVSXBW, PMOVZXBW, PMOVSXBD, PMOVZXBD, PMOVSXBQ, PMOVZXBQ, PMOVSXWD, PMOVZXWD, PMOVSXWQ, PMOVZXWQ, PMOVSXDQ, PMOVZXDQ, PTEST, PCMPEQQ, PACKUS, MOVNTDQA

SSE4a: Added with Phenom

LZCNT, POPCNT, EXTRQ, INSERTQ, MOVNTSD, MOVNTSS

SSE4.2: Added with Nehalem

CRC32, PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM, PCMPGTQ

3D Now

This section of the x86 Assembly book is a stub. You can help by expanding this section.

3DNow! is AMD's extension of the MMX instruction set (K6-2 and more recent) for with floating-point instruction. The instruction set never gained much popularity, and AMD announced on August 2010 that support for 3DNow! will be dropped in future AMD processors, except for two instructions.

Advanced x86

These "Advanced x86" chapters all cover specialized topics that might not be of interest to the average assembly programmer. However, these chapters should be of some interest to people who would like to work on low-level programming tasks, such as boot loaders, device drivers, and Operating System kernels. A programmer does not need to read the following chapters to say they "know assembly", although they may be interesting. The topics covered in this section are:

- High-Level Languages
- Machine Language Conversion
- Protected Mode
- Global Descriptor Table
- Advanced Interrupts
- Bootloaders

High-Level Languages

Very few projects are written entirely in assembly. It's often used for accessing processor-specific features, optimizing critical sections of code, and very low-level work, but for many applications, it can be simpler and easier to implement the basic control flow and data manipulation routines in a higher level language, like C. For this reason, it is often necessary to interface between assembly language and other languages.

Compilers

The first compilers were simply text translators that converted a high-level language into assembly language. The assembly language code was then fed into an assembler, to create the final machine code output. The GCC compiler still performs this sequence (code is compiled into assembly, and fed to the AS assembler). However, many modern compilers will skip the assembly language and create the machine code directly.

Assembly language code has the benefit that it is in one-to-one correspondence with the underlying machine code. Each machine instruction is mapped directly to a single Assembly instruction. Because of this, even when a compiler directly creates the machine code, it is still possible to interface that code with an assembly language program. The important part is knowing exactly how the language implements its data structures, control structures, and functions. The method in which function calls are implemented by a high-level language compiler is called a **calling convention**.

The calling convention is a contract between the function and caller of the function and specifies several parameters:

1. How the arguments are passed to the function, and in what order? Are they pushed onto the stack, or are they passed in via the registers?
2. How are return values passed back to the caller? This is usually via registers or on the stack.
3. What processor states are volatile (available for modification)? Volatile registers are available for modification by the function. The caller is responsible for saving the state of those registers if needed. **Non-volatile** registers are guaranteed to be preserved by the function. The called function is responsible for saving the state of those registers and restoring those registers on exit.
4. The function **prologue** and **epilogue**, which sets up the registers and stack for use within the function and then restores the stack and registers before exiting.

C Calling Conventions

CDECL

For C compilers, the CDECL calling convention is the de facto standard. It varies by compiler, but the programmer can specify that a function be implemented using CDECL usually by pre-appending the function declaration with a keyword, for example `__cdecl` in Visual studio:

```
int __cdecl func()
```

in gcc it would be `__attribute__((__cdecl))`:

```
int __attribute__((__cdecl)) func()
```

CDECL calling convention specifies a number of different requirements:

1. Function arguments are passed on the stack, in **right-to-left** order.
2. Function result is stored in EAX/AX/AL
3. Floating point return values will be returned in ST0
4. The function name is pre-appended with an underscore.
5. The arguments are popped from the stack by the caller itself.
6. 8-bit and 16-bit integer arguments are promoted to 32-bit arguments.
7. The volatile registers are: EAX, ECX, EDX, ST0 - ST7, ES and GS
8. The non-volatile registers are: EBX, EBP, ESP, EDI, ESI, CS and DS
9. The function will exit with a RET instruction.

10. The function is supposed to return values types of class or structure via a reference in EAX/AX. The space is supposed to be allocated by the function, which unable to use the stack or heap is left with fixed address in static non-constant storage. This is inherently not thread safe. Many compilers will break the calling convention:

1. GCC has the calling code allocate space and passes a pointer to this space via a hidden parameter on the stack. The called function writes the return value to this address.
2. Visual C++ will:
 1. Pass POD return values 32 bits or smaller in the EAX register.
 2. Pass POD return values 33-64 bits in size via the EAX:EDX registers
 3. For non-POD return values or values larger than 64-bits, the calling code will allocate space and passes a pointer to this space via a hidden parameter on the stack. The called function writes the return value to this address.

CDECL functions are capable of accepting variable argument lists. Below is example using cdecl calling convention:

```
global main
extern printf

section .data
    align 4
    a: dd 1
    b: dd 2
    c: dd 3
    fmtStr: db "Result: %d", 0x0A, 0

section .bss
    align 4

section .text

; int func( int a, int b, int c )
; {
;     return a + b + c ;
; }
;
func:
    push    ebp            ; Save ebp on the stack
    mov     ebp, esp       ; Replace ebp with esp since we will be using
                           ; ebp as the base pointer for the functions
                           ; stack.
                           ;
                           ; The arguments start at ebp+8 since calling the
                           ; the function places eip on the stack and the
                           ; function places ebp on the stack as part of
                           ; the preamble.
                           ;
    mov     eax, [ebp+8]    ; mov a into eax
    mov     edx, [ebp+12]   ; add b to eax
    lea     eax, [eax+edx]  ; Using lea for arithmetic adding a + b into eax
    add     eax, [ebp+16]   ; add c to eax
    pop     ebp            ; restore ebp
    ret     4              ; Returning, eax contains result

;
; Using main since we are using gcc to link
;
main:
;
; Set up for call to func(int a, int b, int c)
;
; Push variables in right to left order
;
    push    dword [c]
    push    dword [b]
    push    dword [a]
    call    func
    add     esp, 12        ; Pop stack 3 times 4 bytes
    push    eax
    push    dword fmtStr
    call    printf
    add     esp, 8        ; Pop stack 2 times 4 bytes

;
; Alternative to using push for function call setup, this is the method
; used by gcc
;
    sub     esp, 12        ; Create space on stack for three 4 byte variables
    mov     ecx, [b]
    mov     eax, [a]
    mov     [esp+8], dword 4
    mov     [esp+4], ecx
```

```

mov [esp], eax
call func
;push eax
;push dword fmtStr
mov [esp+4], eax
lea eax, [fmtStr]
mov [esp], eax
call printf

;
; Call exit(3) syscall
; void exit(int status)
;
mov ebx, 0 ; Arg one: the status
mov eax, 1 ; Syscall number:
int 0x80

```

In order to assemble, link and run the program we need to do the following:

```

nasm -felf32 -g cdecl.asm
gcc -o cdecl cdecl.o
./cdecl

```

STDCALL

STDCALL is the calling convention that is used when interfacing with the Win32 API on Microsoft Windows systems. STDCALL was created by Microsoft, and therefore isn't always supported by non-Microsoft compilers. It varies by compiler but, the programmer can specify that a function be implemented using STDCALL usually by pre-appending the function declaration with a keyword, for example `__stdcall` in Visual studio:

```
int __stdcall func()
```

in gcc it would be `__attribute__((__stdcall__))`:

```
int __attribute__((__stdcall__)) func()
```

STDCALL has the following requirements:

1. Function arguments are passed on the stack in **right-to-left** order.
2. Function result is stored in EAX/AX/AL
3. Floating point return values will be returned in ST0
4. 64-bits integers and 32/16 bit pointers will be returned via the EAX:EDX registers.
5. 8-bit and 16-bit integer arguments are promoted to 32-bit arguments.
6. Function name is prefixed with an underscore
7. Function name is suffixed with an "@" sign, followed by the number of bytes of arguments being passed to it.
8. The arguments are popped from the stack by the callee (the called function).
9. The volatile registers are: EAX, ECX, EDX, and ST0 - ST7
10. The non-volatile registers are: EBX, EBP, ESP, EDI, ESI, CS, DS, ES, FS and GS
11. The function will exit with a RET n instruction, the called function will pop n additional bytes off the stack when it returns.
12. POD return values 32 bits or smaller will be returned in the EAX register.
13. POD return values 33-64 bits in size will be returned via the EAX:EDX registers.
14. Non-POD return values or values larger than 64-bits, the calling code will allocate space and passes a pointer to this space via a hidden parameter on the stack. The called function writes the return value to this address.

STDCALL functions are not capable of accepting variable argument lists.

For example, the following function declaration in C:

```
_stdcall void MyFunction(int, int, short);
```

would be accessed in assembly using the following function label:

```
_MyFunction@12
```

Remember, on a 32 bit machine, passing a 16 bit argument on the stack (C "short") takes up a full 32 bits of space.

FASTCALL

FASTCALL functions can frequently be specified with the `__fastcall` keyword in many compilers. FASTCALL functions pass the first two arguments to the function in registers, so that the time-consuming stack operations can be avoided. FASTCALL has the following requirements:

1. The first 32-bit (or smaller) argument is passed in ECX/CX/CL (see [1] (<http://msdn.microsoft.com/en-us/library/6xa169sk.aspx>))
2. The second 32-bit (or smaller) argument is passed in EDX/DX/DI
3. The remaining function arguments (if any) are passed on the stack in right-to-left order
4. The function result is returned in EAX/AX/AL
5. The function name is prefixed with an "@" symbol
6. The function name is suffixed with an "@" symbol, followed by the size of passed arguments, in bytes.

C++ Calling Conventions (THISCALL)

The C++ THISCALL calling convention is the standard calling convention for C++. In THISCALL, the function is called almost identically to the CDECL convention, but the **this** pointer (the pointer to the current class) must be passed.

The way that the **this** pointer is passed is compiler-dependent. Microsoft Visual C++ passes it in ECX. GCC passes it as if it were the first parameter of the function. (i.e. between the return address and the first formal parameter.)

Ada Calling Conventions

Pascal Calling Conventions

The Pascal convention is essentially identical to cdecl, differing only in that:

1. The parameters are pushed left to right (logical western-world reading order)
2. The routine being called must clean the stack before returning

Additionally, each parameter on the 32-bit stack must use all four bytes of the DWORD, regardless of the actual size of the datum.

This is the main calling method used by Windows API routines, as it is slightly more efficient with regard to memory usage, stack access and calling speed.

Note: the Pascal convention is NOT the same as the Borland Pascal convention, which is a form of fastcall, using registers (eax, edx, ecx) to pass the first three parameters, and also known as Register Convention.

Fortran Calling Conventions

Inline Assembly

C/C++

This Borland C++ example splits `byte_data` into two bytes in `buf`, the first containing high 4 bits and low 4 bits in the second.

```
void ByteToHalfByte(BYTE *buf, int pos, BYTE byte_data)
{
```

```

asm
{
    mov al, byte_data
    mov ah, al
    shr al, 04h
    and ah, 0Fh
    mov ecx, buf
    mov edx, pos
    mov [ecx+edx], al
    mov [ecx+edx+1], ah
}
}

```

Pascal

The FreePascal Compiler (FPC) and GNU Pascal Compiler (GPC) allow asm-blocks. While GPC only accepts AT&T-syntax, FPC can work with both, and allows a direct pass-through to the assembler. The following two examples are written to work with FPC (regarding compiler directives).

```

program asmDemo(input, output, stderr);

// The $asmMode directive informs the compiler
// which syntax is used in asm-blocks.
// Alternatives are 'att' (AT&T syntax) and 'direct'.
{$asmMode intel}

var
    n, m: longint;
begin
    n := 42;
    m := -7;
    writeln('n = ', n, ' ; m = ', m);

    // instead of declaring another temporary variable
    // and writing "tmp := n; n := m; m := tmp;"
    asm
        mov rax, n // rax := n
        // xchg can only operate at most on one memory address
        xchg rax, m // swaps values in rax and at m
        mov n, rax // n := rax (holding the former m value)
        // an array of strings after the asm-block closing 'end'
        // tells the compiler which registers have changed
        // (you don't wanna mess with the compiler's notion
        // which registers mean what)
        end ['rax'];

    writeln('n = ', n, ' ; m = ', m);
end.

```

In FreePascal you can also write whole functions in assembly language. Also note, that if you use labels, you have to declare them beforehand (FPC requirement):

```

// the 'assembler' modifier allows us
// to implement the whole function in assembly language
function iterativeSquare(const n: longint): qword; assembler;
// you have to familiarize the compiler with symbols
// which are meant to be jump targets
{$goto on}
label
    iterativeSquare_iterate, iterativeSquare_done;
// note, the 'asm'-keyword instead of 'begin'
{$asmMode intel}
asm
    // ecx is used as counter by loop instruction
    mov ecx, n // ecx := n
    mov rax, 0 // rax := 0
    mov r8, 1 // r8 := 1

    cmp ecx, rax // ecx = rax [n = 0]
    je iterativeSquare_done // n = 0

    // ensure ecx is positive
    // so we'll run against zero while decrementing
    jg iterativeSquare_iterate // if n > 0 then goto iterate
    neg ecx // ecx := ecx * -1

    // n^2 = sum over first abs(n) odd integers
iterativeSquare_iterate:
    add rax, r8 // rax := rax + r8
    inc r8 // inc(r8) twice
    inc r8 // to get next odd integer
    loop iterativeSquare_iterate // dec(ecx)
    // if ecx <> 0 then goto iterate

iterativeSquare_done:
    // the @result macro represents the functions return value

```

```

mov @result, rax // result := rax
// note, a list of modified registers (here ['rax', 'ecx', 'r8'])
// is ignored for pure assembler routines
end;

```

Further Reading

For an in depth discussion as to how high-level programming constructs are translated into assembly language, see [Reverse Engineering](#).

- [Subject:C programming language](#)
- [Subject:C++ programming language](#)
- [x86 Disassembly/Calling Conventions](#)
- [x86 Disassembly/Calling Convention Examples](#)

Machine Language Conversion

Relationship to Machine Code

X86 assembly instructions have a one-to-one relationship with the underlying machine instructions. This means that essentially we can convert assembly instructions into machine instructions with a look-up table. This page will talk about some of the conversions from assembly language to machine language.

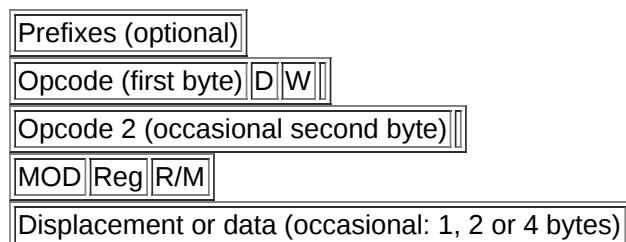
CISC and RISC

The x86 architecture is a **complex instruction set computer** (CISC) architecture. Amongst other things, this means that the instructions for the x86 architecture are of varying lengths. This can make the processes of assembly, disassembly and instruction decoding more complicated, because the instruction length needs to be calculated for each instruction.

x86 instructions can be anywhere between 1 and 15 bytes long. The length is defined separately for each instruction, depending on the available modes of operation of the instruction, the number of required operands and more.

8086 instruction format (16 bit)

This is the general instruction form for the 8086 sequentially in main memory:



Prefixes

Optional prefixes which change the operation of the instruction

D

(1 bit) Direction. 1 = Register is Destination, 0 = Register is source.

W

(1 bit) Operation size. 1 = Word, 0 = byte.

Opcode

the opcode is a 6 bit quantity that determines what instruction family the code is

MOD (Mod)

(2 bits) Register mode.

Reg

(3 bits) Register. Each register has an identifier.

R/M (r/m)

(3 bits) Register/Memory operand

Not all instructions have W or D bits; in some cases, the width of the operation is either irrelevant or implicit, and for other operations the data direction is irrelevant.

Notice that Intel instruction format is little-endian, which means that the lowest-significance bytes are closest to absolute address 0. Thus, words are stored low-byte first; the value 1234H is stored in memory as 34H 12H. By convention, most-significant bits are always shown to the left within the byte, so 34H would be 00110100B.

After the initial 2 bytes, each instruction can have many additional addressing/immediate data bytes.

Mod / Reg / R/M tables

Mod	Displacement
00	If r/m is 110, Displacement (16 bits) is address; otherwise, no displacement
01	Eight-bit displacement, sign-extended to 16 bits
10	16-bit displacement (example: MOV [BX + SI]+ displacement,al)
11	r/m is treated as a second "reg" field

Reg	W = 0	W = 1	double word
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

r/m	Operand address
000	(BX) + (SI) + displacement (0, 1 or 2 bytes long)
001	(BX) + (DI) + displacement (0, 1 or 2 bytes long)
010	(BP) + (SI) + displacement (0, 1 or 2 bytes long)
011	(BP) + (DI) + displacement (0, 1 or 2 bytes long)
100	(SI) + displacement (0, 1 or 2 bytes long)
101	(DI) + displacement (0, 1 or 2 bytes long)
110	(BP) + displacement unless mod = 00 (see mod table)
111	(BX) + displacement (0, 1 or 2 bytes long)

Note the special meaning of MOD 00, r/m 110. Normally, this would be expected to be the operand [BP]. However, instead the 16-bit displacement is treated as the absolute address. To encode the value [BP], you would use mod = 01, r/m = 110, 8-bit displacement = 0.

Example: Absolute addressing

Let's translate the following instruction into machine code:

```
XOR CL, [12H]
```

Note that this is XORing CL with the contents of address 12H – the square brackets are a common indirection indicator. The opcode for XOR is "001100dw". D is 1 because the CL register is the destination. W is 0 because we have a byte of data. Our first byte therefore is "00110010".

Now, we know that the code for CL is 001. Reg thus has the value 001. The address is specified as a simple displacement, so the MOD value is 00 and the R/M is 110. Byte 2 is thus (00 001 110b).

Byte 3 and 4 contain the effective address, low-order byte first, 0012H as 12H 00H, or (00010010b) (00000000b)

All together,

```
XOR CL, [12H] = 00110010 00001110 00010010 00000000 = 32H 0EH 12H 00H
```

Example: Immediate operand

Now, if we were to want to use an immediate operand, as follows:

```
XOR CL, 12H
```

In this case, because there are no square brackets, 12H is immediate: it is the number we are going to XOR against. The opcode for an immediate XOR is 1000000w; in this case, we are using a byte, so w is 0. So our first byte is (10000000b).

The second byte, for an immediate operation, takes the form "mod 110 r/m". Since the destination is a register, mod is 11, making the r/m field a register value. We already know that the register value for CL is 001, so our second byte is (11 110 001b).

The third byte (and fourth byte, if this were a word operation) are the immediate data. As it is a byte, there is only one byte of data, 12H = (00010010b).

All together, then:

```
XOR CL, 12H = 10000000 11110001 00010010 = 80 F1 12
```

x86 Instructions (32/64 bit)

The 32-bit instructions are encoded in a very similar way to the 16-bit instructions, except (by default) they act upon dword quantities rather than words. Also, they support a much more flexible memory addressing format, which is made possible by the addition of an SIB "scale-index-base" byte, which follows the ModR/M byte.

Continuing the previous absolute addressing example, we take this input:

```
XOR CL, [12H]
```

...and we arrive at the 32-bit machine code like so:

Beginning with the opcode byte first, it remains the same, 32H. Consulting the Intel IA-32 manual, Volume 2C, Chapter 5, "XOR"--we see this opcode defines that a) it requires 2 operands, b) the operands have a direction, and the first operand is the destination, c) the first operand is a register of 8-bits width, d) the second operand is also 8-bit but can be either a register or memory address, and e) the destination register CL will be overridden to contain the result of the operation. This fits our case above, because the first operand is CL ("L" meaning lower 8-bits of the "C" register), and the second operand is a reference to the value stored in memory at 12H (a direct/absolute pointer or address reference). It doesn't look like we need any prefix bytes to get the operand sizes we want.

Now we know we need a ModR/M byte, because the opcode requires it; a) it requires more than zero operands, and b) they are not defined within the opcode or any prefix, and c) there is no Immediate operand. So again we consult the Intel manual, Volume 2A, Chapter 2, Section 2.1.5 "Addressing-Mode Encoding of ModR/M and SIB Bytes", Table 2-2 "32-Bit Addressing Forms with the ModR/M Byte". We know the first operand is going to be our destination register, CL, so we see that maps to REG=001b. Next we look for an Effective Address formula which matches our second operand, which is a displacement with no register (and therefore no segment, base, scale, or index). The nearest match is going to be disp32, but reading the table is tricky because of the footnotes. Basically our formula is not in that table, the one we want requires a SIB byte noted as [--][--], which tells us we need to specify Mod=00b, R/M=100b to enable the SIB byte. Our second byte is therefore 00001100b or 0CH.

We know the SIB byte, if it is used, always follows the ModR/M byte, so we continue to the next Table 2-3 "32-Bit Addressing Forms with the SIB Byte" in the Intel manual, and look for the combination of Scale, Index, and Base values which will give us the disp32 formula we need. Notice there is a footnote [*], this basically tells us to specify Scale=00b, Index=100b, Base=101b which means disp32 with no index, no scale, and no base. So our third byte is now 25H.

We know the Displacement byte, if used, always follows the ModR/M and SIB byte, so here we simply specify our 32-bit unsigned integer value in little-endian, meaning our next four bytes are 12000000H.

Finally, we have our machine code:

```
XOR CL, [12H] = 00110010 00001100 00100101 00010010 00000000 00000000 00000000 = 32 0C 25 12 00 00 00
```

This instruction works in both 32-bit Protected mode and 64-bit Long mode.

Protected Mode

This page is going to discuss the differences between real mode and protected mode operations in the x86 processors. It will also discuss how to enter protected mode, and how to exit protected mode. Modern Operating Systems (Windows, Unix, Linux, BSD, etc...) all operate in protected mode, so most assembly language programmers won't need this information. However, this information will be particularly useful to people who are trying to program kernels or bootloaders.

Real Mode Operation

When an x86 processor is powered up or reset, it is in real mode. In real mode, the x86 processor essentially acts like a very fast 8086. Only the base instruction set of the processor can be used. Real mode memory address space is limited to 1MiB of addressable memory, and each memory segment is limited to 64KiB. Real Mode is provided essentially for backwards-compatibility with 8086 and 80186 programs.

Protected Mode Operation

In protected mode operation, the x86 can address 4 GB of address space. This may map directly onto the physical RAM (in which case, if there is less than 4 GB of RAM, some address space is unused), or paging may be used to arbitrarily translate between virtual addresses and physical addresses. In Protected mode, the segments in memory can be assigned protection, and attempts to violate this protection cause a "General Protection" exception.

Protected mode in the 386, amongst other things, is controlled by the **Control Registers**, which are labelled CR0, CR2, CR3, and CR4.

Protected mode in the 286 is controlled by the **Machine Status Word**.

Long Mode

Long mode was introduced by AMD with the advent of the Athlon64 processor. Long mode allows the microprocessor to access 64-bit memory space, and access 64-bit long registers. Many 16 and 32-bit instructions do not work (or work correctly) in Long Mode. x86-64 processors in Real mode act exactly the like 16 bit chips, and x86-64 chips in protected mode act exactly like 32-bit processors. To unlock the 64-bit capabilities of the chip, the chip must be switched into Long Mode.

Entering Protected Mode

The lowest 5 bits of the control register CR0 contain 5 flags that determine how the system is going to function. This status register has 1 flag that we are particularly interested in: the "Protected Mode Enable" flag (PE). Here are the general steps to entering protected mode:

1. Create a Valid GDT (Global Descriptor Table)

2. Create a 6 byte pseudo-descriptor to point to the GDT
3.
 1. If paging is going to be used, load CR3 with a valid page table, PDBR, or PML4.
 2. If PAE (Physical Address Extension) is going to be used, set CR4.PAE = 1.
 3. If switching to long mode, set IA32_EFER.LME = 1.
4. Disable Interrupts (CLI).
5. Load an IDT pseudo-descriptor that has a null limit (this prevents the real mode IDT from being used in protected mode)
6. Set the PE bit (and the PG bit if paging is going to be enabled) of the MSW or CR0 register
7. Execute a far jump (in case of switching to long mode, even if the destination code segment is a 64-bit code segment, the offset must not exceed 32-bit since the far jump instruction is executed in compatibility mode)
8. Load data segment registers with valid selector(s) to prevent GP exceptions when interrupts happen
9. Load SS:(E)SP with a valid stack
10. Load an IDT pseudo-descriptor that points to the IDT
11. Enable Interrupts.

Following sections will talk more about these steps.

Entering Long Mode

To enter Long Mode on a 64-bit x86 processor (x86-64):

1. If paging is enabled, disable paging.
2. If CR4.PAE is not already set, set it.
3. Set IA32_EFER.LME = 1.
4. Load CR3 with a valid PML4 table.
5. Enable paging.
6. At this point you will be in compatibility mode. A far jump may be executed to switch to long mode. However, the offset must not exceed 32-bit.

Using the CR Registers

Many bits of the CR registers only influence behavior in protected mode.

CR0

The CR0 32-bit register has 6 bits that are of interest to us. The low 5 bits of the CR0 register, and the highest bit. Here is a representation of CR0:

```
CR0: | PG | ---RESERVED--- | NE | ET | TS | EM | MP | PE |
```

PE

Bit 0. The Protected Environment flag. This flag puts the system into protected mode when set.

MP

Bit 1. The Monitor Coprocessor flag. This flag controls the operation of the "WAIT" instruction.

EM

Bit 2. The Emulate flag. When this flag is set, coprocessor instructions will generate an exception.

TS

Bit 3. The Task Switched flag. This flag is set automatically when the processor switches to a new task.

ET

Bit 4. The Extension Type flag. ET (also called "R") tells us which type of coprocessor is installed. If ET = 0, an 80287 is installed. if ET = 1, an 80387 is installed.

NE

Bit 5. New exceptions. If this flag is clear, FPU exceptions arrive as interrupts. If set, as exceptions.

PG

Bit 31. The Paging flag. When this flag is set, memory paging is enabled. We will talk more about that in a second.

CR2

CR2 contains a value called the **Page Fault Linear Address (PFLA)**. When a page fault occurs, the address that access was attempted on is stored in CR2.

CR3

The upper 20 bits of CR3 are called the **Page Directory Base Register (PDBR)**. The PDBR holds the physical address of the page directory.

CR4

CR4 contains several flags controlling advanced features of the processor.

Paging

Paging is a special job that microprocessors can perform to make the available amount of memory in a system appear larger and more dynamic than it actually is. In a paging system, a certain amount of space may be laid aside on the hard drive (or on any secondary storage) called the **swap file** or **swap partition**. The **virtual memory** of the system is everything a program can access like memory, and includes physical RAM and the swap space.

The total virtual memory is broken down into chunks or **pages** of memory, each usually being 4096 bytes (although this number can be different on different systems). These pages can then be moved around throughout the virtual memory, and all pointers inside those pages will be automatically directed to point to the new locations by referencing them to a global paging directory that the microprocessor maintains. The pointer to the current paging directory is stored in the CR3 register.

A **page fault** occurs when the system attempts to read from a page that is marked as "not present" in the paging directory/table, when the system attempts to write data beyond the boundaries of a currently available page, or when any number of other errors occur in the paging system. When a page fault occurs, the accessed memory address is stored in the CR2 register.

Other Modes

In addition to real, protected, and long modes, there are other modes that x86 processors can enter, for different uses :

- **Virtual 8086 Mode:** This is a mode in which application software that was written to run in real mode is executed under the supervision of a protected-mode, multi-tasking OS.
- **System Management Mode:** This mode enables the processor to perform system tasks, like power management, without disrupting the operating system or other software.

Global Descriptor Table

The Global Descriptor Table (GDT) is a table in memory that defines the processor's memory segments. The GDT sets the behavior of the segment registers and helps to ensure that protected mode operates smoothly.

GDTR

The GDT is pointed to by a special register in the x86 chip, the **GDTR Register**, or simply the GDTR. The GDTR is 48 bits long. The lower 16 bits tell the size of the GDT, and the upper 32 bits tell the location of the GDT in memory. Here is a layout of the GDTR:

`| LIMIT | ---- BASE ---- |`

LIMIT is the size of the GDT, and BASE is the starting address. LIMIT is 1 less than the length of the table, so if LIMIT has the value 15, then the GDT is 16 bytes long.

To load the GDTR, the instruction **LGDT** is used:

```
lgdt [gdtr]
```

Where `gdtr` is a pointer to 6 bytes of memory containing the desired GDTR value. Note that to complete the process of loading a new GDT, the segment registers need to be reloaded. The **CS** register must be loaded using a far jump:

```
flush_gdt:
    lgdt [gdtr]
    jmp 0x08:complete_flush

complete_flush:
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
    ret
```

GDT

The GDT table contains a number of entries called **Segment Descriptors**. Each is 8 bytes long and contains information on the starting point of the segment, the length of the segment, and the access rights of the segment.

The following NASM-syntax code represents a single GDT entry:

```
struc gdt_entry_struct
    limit_low:    resb 2
    base_low:     resb 2
    base_middle:  resb 1
    access:       resb 1
    granularity:  resb 1
    base_high:    resb 1
endstruc
```

LDT

Each separate program will receive, from the operating system, a number of different memory segments for use. The characteristics of each local memory segment are stored in a data structure called the **Local Descriptor Table** (LDT). The GDT contains pointers to each LDT.

Advanced Interrupts

In the chapter on [Interrupts](#), we mentioned the fact that there are such a thing as software interrupts, and they can be installed by the system. This page will go more in-depth about that process, and will talk about how ISRs are installed, how the system finds the ISR, and how the processor actually performs an interrupt.

Interrupt Service Routines

The actual code that is invoked when an interrupt occurs is called the **Interrupt Service Routine** (ISR). When an exception occurs, a program invokes an interrupt, or the hardware raises an interrupt, the processor uses one of several methods (to be discussed) to transfer control to the ISR, whilst allowing the ISR to safely return control to whatever it interrupted after execution is complete. At minimum, **FLAGS** and **CS:IP** are saved and the ISR's **CS:IP** loaded; however, some mechanisms cause a full task switch to occur before the ISR begins (and another task switch when it ends).

The Interrupt Vector Table

In the original 8086 processor (and all x86 processors in Real Mode), the **Interrupt Vector Table** controlled the flow into an ISR. The IVT started at memory address 0x00, and could go as high as 0x3FF, for a maximum number of 256 ISRs (ranging from interrupt 0 to 255). Each entry in the IVT contained 2 words of data: A value for IP and a value for CS (in

that order). For example, let's say that we have the following interrupt:

```
int 14h
```

When we trigger the interrupt, the processor goes to the 20th location in the IVT (14h = 20). Since each table entry is 4 bytes (2 bytes IP, 2 bytes CS), the microprocessor goes to location [4*14H]=[50H]. At location 50H is the new IP value, and at location 52H is the new CS value. Hardware and software interrupts are all stored in the IVT, so installing a new ISR is as easy as writing a function pointer into the IVT. In newer x86 models, the IVT was replaced with the Interrupt Descriptor Table.

When interrupts occur in real mode, the FLAGS register is pushed onto the stack, followed by CS, then IP. The **iret** instruction restores CS:IP and FLAGS, allowing the interrupted program to continue unaffected. For hardware interrupts, all other registers (including the general-purpose registers) *must* be explicitly preserved (e.g. if an interrupt routine makes use of AX, it should push AX when it begins and pop AX when it ends). It is good practice for software interrupts to preserve all registers except those containing return values. More importantly, any registers that *are* modified must be documented.

The Interrupt Descriptor Table

Since the 286 (but extended on the 386), interrupts may be managed by a table in memory called the **Interrupt Descriptor Table** (IDT). The IDT only comes into play when the processor is in protected mode. Much like the IVT, the IDT contains a listing of pointers to the ISR routine; however, there are now three ways to invoke ISRs:

- **Task Gates:** These cause a task switch, allowing the ISR to run in its own context (with its own LDT, etc.). Note that IRET may still be used to return from the ISR, since the processor sets a bit in the ISR's task segment that causes IRET to perform a task switch to return to the previous task.
- **Interrupt Gates:** These are similar to the original interrupt mechanism, placing EFLAGS, CS and EIP on the stack. The ISR may be located in a segment of equal or higher privilege to the currently executing segment, but not of lower privilege (higher privileges are *numerically lower*, with level 0 being the highest privilege).
- **Trap Gates:** These are identical to interrupt gates, except they do not clear the interrupt flag.

The following NASM structure represents an IDT entry:

```
struc idt_entry_struct
    base_low: resb 2
    sel:      resb 2
    always0:  resb 1
    flags:    resb 1
    base_high: resb 2
endstruc
```

Field	Interrupt Gate	Trap Gate	Task Gate
base_low	Low word of entry address of ISR		Unused
sel	Segment selector of ISR		TSS descriptor
always0	Bits 5, 6, and 7 should be 0. Bits 0-4 are unused and can be left as zero.		Unused, can be left as zero.
flags	Low 5 bits are (MSB first): 01110, bits 5 and 6 form the DPL, bit 7 is the Present bit.	Low 5 bits are (MSB first): 01111, bits 5 and 6 form the DPL, bit 7 is the Present bit.	Low 5 bits are (MSB first): 00101, bits 5 and 6 form the DPL, bit 7 is the Present bit.
base_high	High word of entry address of ISR		Unused

where:

- DPL is the Descriptor Privilege Level (0 to 3, with 0 being highest privilege)
- The Present bit indicates whether the segment is present in RAM. If this bit is 0, a **Segment Not Present** fault (Exception 11) will ensue if the interrupt is triggered.

These ISRs are usually installed and managed by the operating system. Only tasks with sufficient privilege to modify the IDT's contents may directly install ISRs.

The ISR itself must be placed in appropriate segments (and, if using task gates, the appropriate TSS must be set up), particularly so that the privilege is never lower than that of executing code. ISRs for unpredictable interrupts (such as hardware interrupts) should be placed in privilege level 0 (which is the highest privilege), so that this rule is not violated while a privilege-0 task is running.

Note that ISRs, particularly hardware-triggered ones, should *always* be present in memory unless there is a good reason for them not to be. Most hardware interrupts need to be dealt with promptly, and swapping causes significant delay. Also, some hardware ISRs (such as the hard disk ISR) might be *required* during the swapping process. Since hardware-triggered ISRs interrupt processes at unpredictable times, device driver programmers are encouraged to keep ISRs very short. Often an ISR simply organises for a kernel task to do the necessary work; this kernel task will be run at the next suitable opportunity. As a result of this, hardware-triggered ISRs are generally very small and little is gained by swapping them to the disk.

However, it may be desirable to set the present bit to 0, even though the ISR actually is present in RAM. The OS can use the Segment Not Present handler for some other function, for instance to monitor interrupt calls.

IDT Register

The x86 contains a register whose job is to keep track of the IDT. This register is called the **IDT Register**, or simply "IDTR". the IDT register is 48 bits long. The lower 16 bits are called the LIMIT section of the IDTR, and the upper 32 bits are called the BASE section of the IDTR:

```
|LIMIT|----BASE----
```

The BASE is the base address of the IDT in memory. The IDT can be located anywhere in memory, so the BASE needs to point to it. The LIMIT field contains the current length of the IDT.

To load the IDTR, the instruction **LIDT** is used:

```
lidt [idtr]
```

To store the IDTR, the instruction **SIDT** is used:

```
sub esp,6  
sidt [esp] ;store the idtr to the stack
```

Interrupt Instructions

int arg

calls the specified interrupt

into 0x04

calls interrupt 4 if the overflow flag is set

iret

returns from an interrupt service routine (ISR).

Default ISR

A good programming practice is to provide a default ISR that can be used as placeholder for unused interrupts. This is to prevent execution of random code if an unrecognized interrupt is raised. The default ISR can be as simple as a single **iret** instruction.

Note, however, that under DOS (which is in real mode), certain IVT entries contain pointers to important, but not necessarily executable, locations. For instance, entry 0x1D is a far pointer to a video initialisation parameter table for video controllers, entry 0x1F is a pointer to the graphical character bitmap table.

Disabling Interrupts

Sometimes it is important that a routine is not interrupted unexpectedly. For this reason, the x86 allows hardware interrupts to be disabled if necessary. This means the processor will ignore any interrupt signal it receives from the interrupt controller. Usually the controller will simply keep waiting until the processor accepts the interrupt signal, so the interrupts are delayed rather than rejected.

The x86 has an *interrupt flag* (IF) in the FLAGS register. When this flag is set to 0, hardware interrupts are disabled, otherwise they are enabled. The command **cli** sets this flag to 0, and **sti** sets it to 1. Instructions that load values into the FLAGS register (such as **popf** and **iret**) may also modify this flag.

Note that this flag does not affect the **int** instruction or processor exceptions; only hardware-generated interrupts. Also note that in protected mode, code running with less privilege than IOPL will generate an exception if it uses **cli** or **sti**. This means that the operating system can disallow "user" programs from disabling interrupts and thus gaining control of the system.

Interrupts are automatically disabled when an interrupt handler begins; this ensures the handler will not be interrupted (unless it issues **sti**). Software such as device drivers might require precise timing and for this reason should not be interrupted. This can also help avoid problems if the same interrupt occurs twice in a short space of time. Note that the **iret** instruction restores the state of FLAGS before the interrupt handler began, thus allowing further interrupts to occur after the interrupt handler is complete.

Interrupts should also be disabled when performing certain system tasks, such as when entering protected mode. This consists of performing several steps, and if the processor tried to invoke an interrupt handler before this process was complete, it would be in danger of causing an exception, executing invalid code, trashing memory, or causing some other problem.

Bootloaders

When a computer is turned on, there is some beeping, and some flashing lights, and then a loading screen appears. And then magically, the operating system loads into memory. The question is then raised, how does the operating system load up? What gets the ball rolling? The answer is **bootloaders**.

What is a Bootloader?

Bootloaders are small pieces of software that play a role in getting an operating system loaded and ready for execution when a computer is turned on. The way this happens varies between different computer designs (early computers required a person to manually set the computer up whenever it was turned on), and often there are several stages in the process of boot loading.

It's crucial to understand that the term "bootloader" is simply a classification of software (and sometimes a blurry one). To the processor, a bootloader is just another piece of code that it blindly executes. There are many different kinds of boot loaders. Some are small, others are large; some follow very simple rules while others show fancy screens and give the user a selection to choose from.

On IBM PC compatibles, the first program to load is the Basic Input/Output System (BIOS). The BIOS performs many tests and initialisations, and if everything is OK, the BIOS's boot loader begins. Its purpose is to load another boot loader! It selects a disk (or some other storage media) from which it loads a secondary boot loader.

In some cases, *this* boot loader loads enough of an operating system to start running it. In other cases, it loads yet another boot loader from somewhere else. This often happens when multiple operating systems are installed on a single computer; each OS may have its own specific bootloader, with a "central" bootloader that loads one of the specific ones according to the user's selection.

Most bootloaders are written exclusively in assembly language (or even machine code), because they need to be compact, they don't have access to OS routines (such as memory allocation) that other languages might require, they need to follow some unusual requirements, and they make frequent use of low-level features. However some bootloaders, particularly those that have many features and allow user input, are quite heavyweight. These are often written in a combination of assembly and C. The GRand Unified Bootloader (<http://www.gnu.org/software/grub/>) (GRUB) is an example of such.

Some boot loaders are highly OS-specific, while others are less so - certainly the BIOS boot loader is not OS-specific. The MS-DOS boot loader (which was placed on *all* MS-DOS formatted floppy disks) simply checks if the files **IO.SYS** and **MSDOS.SYS** exist; if they are not present it displays the error "Non-System disk or disk error" otherwise it loads and begins execution of **IO.SYS**.

The final stage boot loader may be expected (by the OS) to prepare the computer in some way, for instance by placing the processor in protected mode and programming the interrupt controller. While it would be possible to do these things inside the OS's initialisation procedure, moving them into the bootloader can simplify the OS design. Some operating systems require their bootloader to set up a small, basic GDT (Global Descriptor Table) and enter protected mode, in order to remove the need for the OS to have any 16-bit code. However, the OS might replace this with its own sophisticated GDT soon after.

The Bootsector

The first 512 bytes of a disk are known as the **bootsector** or **Master Boot Record**. The boot sector is an area of the disk reserved for booting purposes. If the bootsector of a disk contains a valid boot sector (the last word of the sector must contain the signature 0xAA55), then the disk is treated by the BIOS as bootable.

The Boot Process

When switched on or reset, an x86 processor begins executing the instructions it finds at address FFFF:0000 (at this stage it is operating in **Real Mode**) (Intel Software Developer's Manual Volume 3 Chapter 9 contradicts this information: Execution starts at the physical address 0xFFFFFFF0, among other things). In IBM PC compatible processors, this address is mapped to a ROM chip that contains the computer's Basic Input/Output System (BIOS) code. The BIOS is responsible for many tests and initialisations; for instance the BIOS may perform a memory test, initialise the interrupt controller and system timer, and test that these devices are working.

Eventually the actual boot loading begins. First the BIOS searches for and initialises available storage media (such as floppy drives, hard disks, CD drives), then it decides which of these it will attempt to boot from. It checks each device for availability (e.g. ensuring a floppy drive contains a disk), then the 0xAA55 signature, in some predefined order (often the order is configurable using the BIOS setup tool). It loads the first sector of the first bootable device it comes across into RAM, and initiates execution.

Ideally, this will be another boot loader, and it will continue the job, making a few preparations, then passing control to something else.

While BIOSes remain compatible with 20-year-old software, they have also become more sophisticated over time. Early BIOSes could not boot from CD drives, but now CD and even DVD booting are standard BIOS features. Booting from USB storage devices is also possible, and some systems can boot from over the network. To achieve such advanced functioning, BIOSes sometimes enter protected mode and the like, but then return to real mode in order to be compatible with legacy boot loaders. This creates a chicken-and-egg problem: bootloaders are written to work with the ubiquitous BIOS, and BIOSes are written to support all those bootloaders, preventing much in the way of new boot loading features.

However, a new bootstrap technology, the UEFI, is beginning to gain momentum. It is much more sophisticated and will not be discussed in this article.

Note also that other computer systems - even some that use x86 processors - may boot in different ways. Indeed, some embedded systems whose software is compact enough to be stored on ROM chips may not need bootloaders at all.

Technical Details

A bootloader runs under certain conditions that the programmer must appreciate in order to make a successful bootloader. The following pertains to bootloaders initiated by the PC BIOS:

1. The first sector of a drive contains its boot loader.
2. One sector is 512 bytes — the last two bytes of which *must* be 0xAA55 (i.e. 0x55 followed by 0xAA), or else the BIOS will treat the drive as unbootable.
3. If everything is in order, said first sector will be placed at RAM address 0000:7C00, and the BIOS's role is over as it transfers control to 0000:7C00 (that is, it Jumps to that address).
4. The DL register will contain the drive number that is being booted from, useful if you want to read more data from elsewhere on the drive.
5. The BIOS leaves behind a lot of code, both to handle hardware interrupts (such as a keypress) and to provide services to the bootloader and OS (such as keyboard input, disk read, and writing to the screen). You must understand the purpose of the Interrupt Vector Table (IVT), and be careful not to interfere with the parts of the BIOS that you depend on. Most operating systems replace the BIOS code with their own code, but the boot loader can't use anything but its own code and what the BIOS provides. Useful BIOS services include `int 10h` (for displaying text/graphics), `int 13h` (disk functions) and `int 16h` (keyboard input).
6. This means that any code or data that the boot loader needs must either be included in the first sector (be careful not to accidentally execute data) or manually loaded from another sector of the disk to somewhere in RAM. Because the OS is not running yet, most of the RAM will be unused. However, you must take care not to interfere with the RAM that is required by the BIOS interrupt handlers and services mentioned above.
7. The OS code itself (or the next bootloader) will need to be loaded into RAM as well.
8. The BIOS places the stack pointer 512 bytes beyond the end of the boot sector, meaning that the stack cannot exceed 512 bytes. It may be necessary to move the stack to a larger area.
9. There are some conventions that need to be respected if the disk is to be readable under mainstream operating systems. For instance you may wish to include a BIOS Parameter Block on a floppy disk to render the disk readable under most PC operating systems.

Most assemblers will have a command or directive similar to `ORG 7C00h` that informs the assembler that the code will be loaded starting at offset 7C00h. The assembler will take this into account when calculating instruction and data addresses. If you leave this out, the assembler assumes the code is loaded at address 0 and this must be compensated for manually in the code.

Usually, the bootloader will load the kernel into memory, and then jump to the kernel. The kernel will then be able to reclaim the memory used by the bootloader (because it has already performed its job). However it is possible to include OS code within the boot sector and keep it resident after the OS begins.

Here is a simple bootloader demo designed for NASM:

```
org 7C00h

jmp short Start ;Jump over the data (the 'short' keyword makes the jmp instruction smaller)

Msg: db "Hello World! "
EndMsg:

Start: mov bx, 000Fh ;Page 0, colour attribute 15 (white) for the int 10 calls below
      mov cx, 1      ;We will want to write 1 character
      xor dx, dx      ;Start at top left corner
      mov ds, dx      ;Ensure ds = 0 (to let us load the message)
      cld             ;Ensure direction flag is cleared (for LODSB)

Print: mov si, Msg    ;Loads the address of the first byte of the message, 7C02h in this case
      ;PC BIOS Interrupt 10 Subfunction 2 - Set cursor position
      ;AH = 2
      ;BH = page, DH = row, DL = column
Char:  mov ah, 2
      int 10h
      lodsb           ;Load a byte of the message into AL.
      ;Remember that DS is 0 and SI holds the
      ;offset of one of the bytes of the message.

      ;PC BIOS Interrupt 10 Subfunction 9 - Write character and colour
      ;AH = 9
      ;BH = page, AL = character, BL = attribute, CX = character count
      mov ah, 9
      int 10h
```

```

    inc dl          ;Advance cursor

    cmp dl, 80      ;Wrap around edge of screen if necessary
    jne Skip
    xor dl, dl
    inc dh

    cmp dh, 25      ;Wrap around bottom of screen if necessary
    jne Skip
    xor dh, dh

Skip:  cmp si, EndMsg ;If we're not at end of message,
       jne Char      ;continue loading characters
       jmp Print      ;otherwise restart from the beginning of the message

times 0200h - 2 - ($ - $$) db 0      ;Zerofill up to 510 bytes

    dw 0AA55h        ;Boot Sector signature

;OPTIONAL:
;To zerofill up to the size of a standard 1.44MB, 3.5" floppy disk
;times 1474560 - ($ - $$) db 0

```

To compile the above file, suppose it is called 'floppy.asm', you can use following command:

```
nasm -f bin -o floppy.img floppy.asm
```

While strictly speaking this is not a bootloader, it is bootable, and demonstrates several things:

- How to include and access data in the boot sector
- How to skip over included data (this is required for a BIOS Parameter Block)
- How to place the 0xAA55 signature at the end of the sector (NASM will issue an error if there is too much code to fit in a sector)
- The use of BIOS interrupts

On Linux, you can issue a command like

```
cat floppy.img > /dev/fd0
```

to write the image to the floppy disk (the image may be smaller than the size of the disk in which case only as much information as is in the image will be written to the disk). A more sophisticated option is to use the dd utility:

```
dd if=floppy.img of=/dev/fd0
```

Under Windows you can use software such as RAWRITE.

Hard disks

Hard disks usually add an extra layer to this process, since they may be partitioned. The first sector of a hard disk is known as the Master Boot Record (MBR). Conventionally, the partition information for a hard disk is included at the end of the MBR, just before the 0xAA55 signature.

The role of the BIOS is no different to before: to read the first sector of the disk (that is, the MBR) into RAM, and transfer execution to the first byte of this sector. The BIOS is oblivious to partitioning schemes - all it checks for is the presence of the 0xAA55 signature.

While this means that one can use the MBR in any way one would like (for instance, omit or extend the partition table) this is seldom done. Despite the fact that the partition table design is very old and limited - it is limited to four partitions - virtually all operating systems for IBM PC compatibles assume that the MBR will be formatted like this. Therefore to break with convention is to render your disk inoperable except to operating systems specifically designed to use it.

In practice, the MBR usually contains a boot loader whose purpose is to load another boot loader - to be found at the start of one of the partitions. This is often a very simple program which finds the first partition marked *Active*, loads its first sector into RAM, and commences its execution. Since by convention the new boot loader is also loaded to address 7C00h,

the old loader may need to relocate all or part of itself to a different location before doing this. Also, ES:SI is expected to contain the address in RAM of the partition table, and DL the boot drive number. Breaking such conventions may render a bootloader incompatible with other bootloaders.

However, many boot managers (software that enables the user to select a partition, and sometimes even kernel, to boot from) use custom MBR code which loads the remainder of the boot manager code from somewhere on disk, then provides the user with options on how to continue the bootstrap process. It is also possible for the boot manager to reside within a partition, in which case it must first be loaded by another boot loader.

Most boot managers support chain loading (that is, starting another boot loader via the usual first-sector-of-partition-to-address-7C00 process) and this is often used for systems such as DOS and Windows. However, some boot managers (notably GRUB) support the loading of a user-selected kernel image. This can be used with systems such as GNU/Linux and Solaris, allowing more flexibility in starting the system. The mechanism may differ somewhat from that of chain loading.

Clearly, the partition table presents a chicken-and-egg problem that is placing unreasonable limitations on partitioning schemes. One solution gaining momentum is the GUID Partition Table; it uses a dummy MBR partition table so that legacy operating systems will not interfere with the GPT, while newer operating systems can take advantage of the many improvements offered by the system.

GNU GRUB

The GRand Unified Bootloader (<http://www.gnu.org/software/grub/>) supports the flexible *multiboot* (http://www.gnu.org/software/grub/manual/multiboot/html_node/index.html) boot protocol. This protocol aims to simplify the boot process by providing a single, flexible protocol for booting a variety of operating systems. Many free operating systems can be booted using multiboot.

GRUB is extremely powerful and is practically a small operating system. It can read various file systems and thus lets you specify a kernel image *by filename* as well as separate module files that the kernel may make use of. Command-line arguments can be passed to the kernel as well - this is a nice way of starting an OS in maintenance mode, or "safe mode", or with VGA graphics, and so on. GRUB can provide a menu for the user to select from as well as allowing custom loading parameters to be entered.

Obviously this functionality cannot possibly be provided in 512 bytes of code. This is why GRUB is split into two or three "stages":

- Stage 1 - this is a 512-byte block that has the location of stage 1.5 or stage 2 hardcoded into it. It loads the next stage.
- Stage 1.5 - an optional stage which understands the filesystem (e.g. FAT32 or ext3) where stage 2 resides. It will find out where stage 2 is located and load it. This stage is quite small and is located in a fixed area, often just after Stage 1.
- Stage 2 - this is a much larger image that has all the GRUB functionality.

Note that Stage 1 may be installed to the Master Boot Record of a hard disk, or may be installed in one of the partitions and chainloaded by another boot loader.

Windows can not be loaded using multiboot, but the Windows bootloader (like those of other non-multiboot operating systems) can be chainloaded from GRUB, which isn't quite as good, but does let you boot such systems.

Example of a Boot Loader – Linux Kernel v0.01

```
SYSSIZE=0x8000
```

```
boot.s
```

```
boot.s is loaded at 0x7c00 by the bios-startup routines, and moves itself  
out of the way to address 0x90000, and jumps there.
```

```
It then loads the system at 0x10000, using BIOS interrupts. Thereafter  
it disables all interrupts, moves the system down to 0x0000, changes  
to protected mode, and calls the start of system. System then must  
RE-initialize the protected mode in it's own tables, and enable  
interrupts as needed.
```

NOTE! currently system is at most 8*65536 bytes long. This should be no problem, even in the future. I want to keep it simple. This 512 kB kernel size should be enough - in fact more would mean we'd have to move not just these start-up routines, but also do something about the cache-memory (block IO devices). The area left over in the lower 640 kB is meant for these. No other memory is assumed to be "physical", i.e. all memory over 1Mb is demand-paging. All addresses under 1Mb are guaranteed to match their physical addresses.

NOTE1 above is no longer valid in it's entirety. cache-memory is allocated above the 1Mb mark as well as below. Otherwise it is mainly correct.

NOTE 2! The boot disk type must be set at compile-time, by setting the following equ. Having the boot-up procedure hunt for the right disk type is severe brain-damage. The loader has been made as simple as possible (had to, to get it in 512 bytes with the code to move to protected mode), and continuous read errors will result in a unbreakable loop. Reboot by hand. It loads pretty fast by getting whole sectors at a time whenever possible.

| 1.44Mb disks:

sectors = 18

| 1.2Mb disks:

sectors = 15

| 720kB disks:

sectors = 9

.globl begtext, begdata, begbss, endtext, enddata, endbss

.text

begtext:

.data

begdata:

.bss

begbss:

.text

BOOTSEG = 0x07c0

INITSEG = 0x9000

SYSSEG = 0x1000 | system loaded at 0x10000 (65536).

ENDSEG = SYSSEG + SYSSIZE

entry start

start:

mov ax,#BOOTSEG

mov ds,ax

mov ax,#INITSEG

mov es,ax

mov cx,#256

sub si,si

sub di,di

rep

movw

jmp go,INITSEG

go: mov ax,cs

mov ds,ax

mov es,ax

mov ss,ax

mov sp,#0x400 | arbitrary value >>512

mov ah,#0x03 | read cursor pos

xor bh,bh

int 0x10

mov cx,#24

mov bx,#0x0007 | page 0, attribute 7 (normal)

mov bp,#msg1

mov ax,#0x1301 | write string, move cursor

int 0x10

| ok, we've written the message, now

| we want to load the system (at 0x10000)

mov ax,#SYSSEG

mov es,ax | segment of 0x010000

call read_it

call kill_motor

| if the read went well we get current cursor position ans save it for posterity.

mov ah,#0x03 | read cursor pos

xor bh,bh

int 0x10 | save it in known place, con_init fetches

mov [510],dx | it from 0x90510.

| now we want to move to protected mode ...

cli | no interrupts allowed !

| first we move the system to it's rightful place

mov ax,#0x0000

cld | 'direction'=0, movs moves forward

do_move:

mov es,ax | destination segment

add ax,#0x1000

```

cmp ax,#0x9000
jz end_move
mov ds,ax      | source segment
sub di,di
sub si,si
mov cx,#0x8000
rep
movsw
j do_move

```

| then we load the segment descriptors

end_move:

```

mov ax,cs      | right, forgot this at first. didn't work :-)
mov ds,ax
lidt idt_48    | load idt with 0,0
lgdt gdt_48    | load gdt with whatever appropriate

```

| that was painless, now we enable A20

```

call empty_8042
mov al,#0xD1    | command write
out #0x64,al
call empty_8042
mov al,#0xDF    | A20 on
out #0x60,al
call empty_8042

```

| well, that went ok, I hope. Now we have to reprogram the interrupts :-(
 | we put them right after the intel-reserved hardware interrupts, at
 | int 0x20-0x2F. There they won't mess up anything. Sadly IBM really
 | messed this up with the original PC, and they haven't been able to
 | rectify it afterwards. Thus the BIOS puts interrupts at 0x08-0x0f,
 | which is used for the internal hardware interrupts as well. We just
 | have to reprogram the 8259's, and it isn't fun.

```

mov al,#0x11    | initialization sequence
out #0x20,al    | send it to 8259A-1
.word 0x00eb,0x00eb | jmp $+2, jmp $+2
out #0xA0,al    | and to 8259A-2
.word 0x00eb,0x00eb
mov al,#0x20    | start of hardware int's (0x20)
out #0x21,al
.word 0x00eb,0x00eb
mov al,#0x28    | start of hardware int's 2 (0x28)
out #0xA1,al
.word 0x00eb,0x00eb
mov al,#0x04    | 8259-1 is master
out #0x21,al
.word 0x00eb,0x00eb
mov al,#0x02    | 8259-2 is slave
out #0xA1,al
.word 0x00eb,0x00eb
mov al,#0x01    | 8086 mode for both
out #0x21,al
.word 0x00eb,0x00eb
out #0xA1,al
.word 0x00eb,0x00eb
mov al,#0xFF    | mask off all interrupts for now
out #0x21,al
.word 0x00eb,0x00eb
out #0xA1,al

```

| well, that certainly wasn't fun :-(. Hopefully it works, and we don't
 | need no steenking BIOS anyway (except for the initial loading :-).
 | The BIOS-routine wants lots of unnecessary data, and it's less
 | "interesting" anyway. This is how REAL programmers do it.

| Well, now's the time to actually move into protected mode. To make
 | things as simple as possible, we do no register set-up or anything,
 | we let the gnu-compiled 32-bit programs do that. We just jump to
 | absolute address 0x00000, in 32-bit protected mode.

```

mov ax,#0x0001 | protected mode (PE) bit
lmsw ax        | This is it!
jmp 0,8        | jmp offset 0 of segment 8 (cs)

```

| This routine checks that the keyboard command queue is empty
 | No timeout is used - if this hangs there is something wrong with
 | the machine, and we probably couldn't proceed anyway.

empty_8042:

```

.word 0x00eb,0x00eb
in al,#0x64    | 8042 status port
test al,#2     | is input buffer full?
jnz empty_8042 | yes - loop
ret

```

| This routine loads the system at address 0x10000, making sure
 | no 64kB boundaries are crossed. We try to load it as fast as
 | possible, loading whole tracks whenever we can.

in: es - starting address segment (normally 0x1000)

| This routine has to be recompiled to fit another drive type,
 | just change the "sectors" variable at the start of the file
 | (originally 18, for a 1.44Mb drive)

```

|
sread: .word 1          | sectors read of current track
head:  .word 0          | current head
track:  .word 0          | current track
read_it:
    mov ax,es
    test ax,#0x0fff
die:   jne die           | es must be at 64kB boundary
    xor bx,bx           | bx is starting address within segment
rp_read:
    mov ax,es
    cmp ax,#ENDSEG      | have we loaded all yet?
    jb ok1_read
    ret
ok1_read:
    mov ax,#sectors
    sub ax,sread
    mov cx,ax
    shl cx,#9
    add cx,bx
    jnc ok2_read
    je ok2_read
    xor ax,ax
    sub ax,bx
    shr ax,#9
ok2_read:
    call read_track
    mov cx,ax
    add ax,sread
    cmp ax,#sectors
    jne ok3_read
    mov ax,#1
    sub ax,head
    jne ok4_read
    inc track
ok4_read:
    mov head,ax
    xor ax,ax
ok3_read:
    mov sread,ax
    shl cx,#9
    add bx,cx
    jnc rp_read
    mov ax,es
    add ax,#0x1000
    mov es,ax
    xor bx,bx
    jmp rp_read

read_track:
    push ax
    push bx
    push cx
    push dx
    mov dx,track
    mov cx,sread
    inc cx
    mov ch,d1
    mov dx,head
    mov dh,d1
    mov dl,#0
    and dx,#0x0100
    mov ah,#2
    int 0x13
    jc bad_rt
    pop dx
    pop cx
    pop bx
    pop ax
    ret
bad_rt: mov ax,#0
    mov dx,#0
    int 0x13
    pop dx
    pop cx
    pop bx
    pop ax
    jmp read_track

/*
 * This procedure turns off the floppy drive motor, so
 * that we enter the kernel in a known state, and
 * don't have to worry about it later.
 */
kill_motor:
    push dx
    mov dx,#0x3f2
    mov al,#0
    outb
    pop dx
    ret

gdt:
    .word 0,0,0,0      | dummy
    .word 0x07FF       | 8Mb - limit=2047 (2048*4096=8Mb)
    .word 0x0000       | base address=0

```

```

.word 0x9A00 | code read/exec
.word 0x00C0 | granularity=4096, 386

.word 0x07FF | 8Mb - limit=2047 (2048*4096=8Mb)
.word 0x0000 | base address=0
.word 0x9200 | data read/write
.word 0x00C0 | granularity=4096, 386

idt_48:
.word 0 | idt limit=0
.word 0,0 | idt base=0L

gdt_48:
.word 0x800 | gdt limit=2048, 256 GDT entries
.word gdt,0x9 | gdt base = 0X9xxxx

msg1:
.byte 13,10
.ascii "Loading system ..."
.byte 13,10,13,10

.text
endtext:
.data
enddata:
.bss
endbss:

```

Testing the Bootloader

Perhaps the easiest way to test a bootloader is inside a virtual machine, like VirtualBox or VMware.^[1]

Sometimes it is useful if the bootloader supports the GDB remote debug protocol.^[2]

Further Reading

1. "How to develop your own Boot Loader" (http://www.codeproject.com/KB/tips/boot-loader.aspx?fid=1541607&df=90&mpp=25&noise=3&sort=Position&view=Quick&fr=1#_Toc231383187) by Alex Kolesnyk 2009
 2. "RedBoot Debug and Bootstrap Firmware" (<http://www.ecoscentric.com/ecos/redboot.shtml>)
- [Embedded Systems/Bootloaders and Bootsectors](#) describes bootloaders for a variety of embedded systems. (Most embedded systems do not have a x86 processor).

x86 Chipset

Chipset

The original IBM computer was based around the 8088 microprocessor, although the 8088 alone was not enough to handle all the complex tasks required by the system. A number of other chips were developed to support the microprocessor unit (MPU), and many of these other chips survive, in one way or another, to this day. The chapters in this section will talk about some of the additional chips in the standard x86 chipset, including the DMA chip, the interrupt controller, and the Timer.

This section currently only contains pages about the programmable peripheral chips, although eventually it could also contain pages about the non-programmable components of the x86 architecture, such as the RAM, the Northbridge, etc.

Many of the components discussed in these chapters have been integrated onto larger die through the years. The DMA and PIC controllers, for instance, are both usually integrated into the Southbridge ASIC. If the PCI Express standard becomes widespread, many of these same functions could be integrated into the PCI Express controller, instead of into the traditional Northbridge/Southbridge chips.

The chips covered in this section are:

- [Direct Memory Access](#)
- [Programmable Interrupt Controller](#)
- [Programmable Interrupt Timer](#)
- [Programmable Parallel Interface](#)

Direct Memory Access

Direct Memory Access

The **Direct Memory Access** chip (DMA) was an important part of the original IBM PC and has become an essential component of modern computer systems. DMA allows other computer components to access the main memory directly, without the processor having to manage the data flow. This is important because in many systems, the processor is a data-flow bottleneck, and it would slow down the system considerably to have the MPU have to handle every memory transaction.

The original DMA chip was known as the 8237-A chip, although modern variants may be one of many different models.

DMA Operation

The DMA chip can be used to move large blocks of data between two memory locations, or it can be used to move blocks of data from a peripheral device to memory. For instance, DMA is used frequently to move data between the PCI bus to the expansion cards, and it is also used to manage data transmissions between primary memory (RAM) and the secondary memory (HDD). While the DMA is operational, it has control over the memory bus, and the MPU may not access the bus for any reason. The MPU may continue operating on the instructions that are stored in its caches, but once the caches are empty, or once a memory access instruction is encountered, the MPU must wait for the DMA operation to complete. The DMA can manage memory operations much more quickly than the MPU can, so the wait times are usually not a large speed problem.

DMA Channels

The DMA chip has up to 8 DMA channels, and one of these channels can be used to cascade a second DMA chip for a total of 14 channels available. Each channel can be programmed to read from a specific source, to write to a specific source, etc. Because of this, the DMA has a number of dedicated I/O addresses available, for writing to the necessary control registers. The DMA uses addresses 0x0000-0x000F for standard control registers, and 0x0080-0x0083 for page registers.

Programmable Interrupt Controller

The original IBM PC contained a chip known as the **Programmable Interrupt Controller** to handle the incoming interrupt requests from the system, and to send them in an orderly fashion to the MPU for processing. The original interrupt controller was the 8259A chip, although modern computers will have a more recent variant. The most common replacement is the APIC (Advanced Programmable Interrupt Controller) which is essentially an extended version of the old PIC chip to maintain backwards compatibility. This page will cover the 8259A.

Function

The function of the 8259A is actually relatively simple. Each PIC has 8 input lines, called Interrupt Requests (IRQ), numbered from 0 to 7. When one of these lines goes high, the PIC alerts the CPU and sends the appropriate interrupt number. This number is calculated by adding the IRQ number (0 to 7) to an internal "vector offset" value. The CPU uses this value to execute an appropriate Interrupt Service Routine. (For further information, see Advanced Interrupts).

Of course, it's not quite as simple as that, because each system has two PICS, a "master" and a "slave". So when the slave raises an interrupt, it's actually sent to the master, which sends that to the CPU. In this way, interrupts cascade and a processor can have 16 IRQ lines. Of these 16, one is needed for the two PIC chips to interface with each other, so the number of available IRQs is decreased to 15.



Path of an interrupt, from hardware to CPU

While `cli` and `sti` can be used to disable and enable *all* hardware interrupts, it's sometimes desirable to selectively disable interrupts from certain devices. For this purpose, PICs have an internal 8-bit register called the Interrupt Mask Register (IMR). The bits in this register determine which IRQs are passed on to the CPU. If an IRQ is raised but the corresponding bit in the IMR is set, it is ignored and nothing is sent to the CPU.

IRQs

Of the 15 usable IRQs, some are universally associated with one type of device:

- IRQ 0 — system timer
- IRQ 1 — keyboard controller
- IRQ 3 — serial port COM2
- IRQ 4 — serial port COM1
- IRQ 5 — line print terminal 2
- IRQ 6 — floppy controller
- IRQ 7 — line print terminal 1
- IRQ 8 — RTC timer
- IRQ 12 — mouse controller
- IRQ 13 — math co-processor
- IRQ 14 — ATA channel 1
- IRQ 15 — ATA channel 2

Programming

Each of the system's two PICs are assigned a command port and a data port:

	PIC1	PIC2
Command	0x20	0xA0
Data	0x21	0xA1

Masking

Normally, reading from the data port returns the IMR register (see above), and writing to it sets the register. We can use this to set which IRQs should be ignored. For example, to disable IRQ 6 (floppy controller) from firing:

```
in ax, 0x21
or ax, (1 << 6)
out 0x21, ax
```

In the same way, to disable IRQ 12 (mouse controller):

```
in ax, 0xA1
or ax, (1 << 4) ;IRQ 12 is actually IRQ 4 of PIC2
out 0xA1, ax
```

Remapping

Another common task, often performed during the initialization of an operating system, is remapping the PICs. That is, changing their internal vector offsets, thereby altering the interrupt numbers they send. The initial vector offset of PIC1 is 8, so it raises interrupt numbers 8 to 15. Unfortunately, some of the low 32 interrupts are used by the CPU for exceptions (divide-by-zero, page fault, etc.), causing a conflict between hardware and software interrupts. The usual solution to this is remapping the PIC1 to start at 32, and often the PIC2 right after it at 40. This requires a complete restart of the PICs, but is not actually too difficult, requiring just eight `outs`.

```
mov al, 0x11
out 0x20, al ;restart PIC1
out 0xA0, al ;restart PIC2
```

```

mov al, 0x20
out 0x21, al    ;PIC1 now starts at 32
mov al, 0x28
out 0xA1, al    ;PIC2 now starts at 40

mov al, 0x04
out 0x21, al    ;setup cascading
mov al, 0x02
out 0xA1, al

mov al, 0x01
out 0x21, al
out 0xA1, al    ;done!

```

Programmable Interrupt Timer

The **Programmable Interval Timer** (PIT) is an essential component of modern computers, especially in a multi-tasking environment. The PIT chip can be made – by setting various register values – to count up or down, at certain rates, and to trigger interrupts at certain times. The timer can be set into a cyclic mode, so that when it triggers it automatically starts counting again, or it can be set into a one-time-only countdown mode.

On newer hardware, a HPET (https://en.wikipedia.org/wiki/High_Precision_Event_Timer) (High Precision Event Timer), which is an evolution of the PIT concept, is likely to be available.

Function

The PIT contains a crystal oscillator which emits a signal 1193182 hz. This output frequency is divided by three different values to provide three output channels to the CPU. Channel 0 is used as a system timer by most operating systems. Channel 1 was used to refresh the DRAM, but is no longer used and may not even be accessible. Channel 2 is used to control the PC speaker. Of these, channel 0 is the most frequently encountered.

To make the PIT fire at a certain frequency f , you need to figure out an integer x , such that $11931820 / x = f$. This is a trivially solved problem which results in the formula:

$$x = 1193182 / f$$

How this division actually works is that each divisor is saved in an internal register. On every clock pulse, the register is decremented. Only when it reaches 0 is the clock pulse allowed to continue on to the CPU. Higher divisors result in lower frequencies, and vice versa.

Note that because the divisor is 16 bits, and a value of 0 is interpreted as 65536, there are limits on the producible frequencies:

$$\begin{aligned} \text{max} &= 1193182 / 1 = 1193182 \text{ hz} \\ \text{min} &= 1193182 / 65536 \approx 18.2065 \text{ hz} \end{aligned}$$

This final value is also the resolution of the frequency, that is, each consecutive possible frequency differs by 18.2065 hz.

Programming

The PIT is accessed via four ports, three for the three channels and one for commands:

Channel 0	0x40
Channel 1	0x41
Channel 2	0x42
Command	0x43

One commonly performed task is setting the frequency of the channel 0, the system timer. If a frequency of 100 hz is desired, we see that the necessary divisor is $1193182 / 100 = 11931$. This value must be sent to the PIT split into a high and low byte.

```
mov al, 0x36
out 0x43, al    ;tell the PIT which channel we're setting

mov ax, 11931
out 0x40, al    ;send low byte
mov al, ah
out 0x40, al    ;send high byte
```

Programmable Parallel Interface

This section of the x86 Assembly book is a stub. You can help by expanding this section.

The Original x86 PC had another peripheral chip onboard known as the 8255A **Programmable Peripheral Interface** (PPI). The 8255A, and variants (82C55A, 82B55A, etc.) controlled the communications tasks with the outside world. The PPI chips can be programmed to operate in different I/O modes.

Resources

Wikimedia Sources

- [x86 Disassembly](#)
- [Operating System Design](#)
- [Embedded Systems](#)
- [Floating Point](#)
- [C Programming](#)
- [C++ Programming](#)

Books

- Yurichev, Dennis, "An Introduction To Reverse Engineering for Beginners". Online book: http://yurichev.com/writings/RE_for_beginners-en.pdf
- Carter, Paul, "PC Assembly Tutorial". Online book. <http://www.drpaulcarter.com/pcasm/index.php>
- Hyde, Randall, "The Art of Assembly Language", No Starch Press, 2003. [ISBN 1886411972](#). <http://www.artofassembly.com>
- Triebel and Signh, "The 8088 and 8086 Microprocessors: Programming, Interfacing, Software, Hardware, and Applications", 4th Edition, Prentice Hall, 2003. [ISBN 0130930814](#)
- Jonathan Bartlett, "Programming from the Ground Up", Bartlett Publishing, July 31, 2004. [ISBN 0975283847](#). Available online at <http://download.savannah.gnu.org/releases/pgubook/>
- Tambe, Pratik, "Primitiveasm: Learn Assembly Language in 15 days!!!", 1st online Edition at <http://pratik.tambe.googlepages.com/>
- Gerber, R. and Bik, A.J.C. and Smith, K. and Tian, X., "The Software Optimization Cookbook: High-Performance Recipes for IA-32 Platforms", 2nd edition, Intel Press, 2006. [ISBN 9780976483212](#)
- Blum, R., "Professional Assembly Language", 1st edition, Wiley, 2005. [ISBN 9780764579011](#)
- [x86 Assembly Adventures video course by xorpd \(http://xorpd.net\)](#)

Web Resources

- The Intel® 64 and IA-32 Architectures Software Developer's Manuals (<http://www.intel.com/products/processor/manuals/>) cover basic architecture, instruction set, system programming and other topics. They are available in downloadable PDF and can also be ordered on CD-ROM (order form (<http://www.intel.com/products/processor/manuals/order.htm>)) and as hard copy (order information (<http://www.intel.com/design/literature.htm>)).
- There are many online instruction set references. <http://web.archive.org/20051121163920/home.comcast.net/~fbui/intel.html> includes the number of clock

cycles that each instruction takes, and <http://siyobik.info/index.php?module=x86> gives a thorough summary of each command, including pseudocode describing the operation.

- AMD's AMD64 documentation on CD-ROM (U.S. and Canada only) and downloadable PDF format - maybe not independent but complete description of AMD64 through Assembly.
<http://developer.amd.com/documentation/guides/Pages/default.aspx#manuals>
- Optimizing subroutines in assembly language: An optimization guide for x86 platforms (http://www.agner.org/optimize/optimizing_assembly.pdf)
- The microarchitecture of Intel and AMD CPU's: An optimization guide for assembly programmers and compiler makers (<http://www.agner.org/optimize/microarchitecture.pdf>)
- Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel and AMD CPU's (http://www.agner.org/optimize/instruction_tables.pdf)
- Calling conventions for different C++ compilers and operating systems (http://www.agner.org/optimize/calling_conventions.pdf)
- "8086 Microprocessor Emulator" (<http://www.emu8086.com/>) "emu8086 is the emulator ... with integrated 8086 assembler and tutorials for beginners. The emulator runs programs like the real microprocessor in step-by-step mode. It shows registers, memory, stack, variables and flags. All memory values can be investigated and edited by a double click."
- "Using the RDTSC Instruction for Performance Monitoring" (<http://www.ccsf.carleton.ca/~jamuir/rdtscpm1.pdf>)
- "SIMPLY FPU" (<http://www.website.masmforum.com/tutorials/fptute/>)
- "Paul Hsieh's x86 Assembly Language Page" (<http://www.azillionmonkeys.com/qed/asm.html>)
- "The world's leading source for technical x86 processor information" (<http://www.sandpile.org/>)
- "The Art of Picking Intel Registers" (<http://www.swansontec.com/sregisters.html>)
- x86 Assembly Adventures Open source exercises (https://github.com/xorpd/asm_prog_ex)

Other Assembly Languages

Assembly Languages	
▣ x86 Assembly	The Assembly Language used by 32-bit Intel Machines including the 386, 486, and Pentium Family.
▣ MIPS Assembly	A Common RISC Assembly Language that is both powerful and relatively easy to learn.
▣ 68000 Assembly	The Assembly language used by the Motorola 68000 series of microprocessors.
▣ PowerPC Assembly	The Assembly language used by the IBM PowerPC architecture.
▣ SPARC Assembly	The Assembly language used by SPARC Systems and mainframes.
▣ 6502 Assembly	The 6502 is a popular 8-bit microcontroller that is cheap and easy to use.
▣ TI 83 Plus Assembly	The instruction set used with the TI 83 Plus brand of programmable graphing calculators.
▣ 360 Assembly	The instruction set used with the IBM 360 / 370 / 93xx and z/System brand of Mainframe computers.

0

Licensing

[X86 Assembly/Licensing](#)

GNU Free Documentation License

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats

include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
 - I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
 - J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
 - K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
 - L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
 - M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified version.
 - N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
 - O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Retrieved from "https://en.wikibooks.org/w/index.php?title=X86_Assembly/Print_Version&oldid=3569993"

This page was last edited on 8 September 2019, at 03:34.

Text is available under the [Creative Commons Attribution-ShareAlike License](#).; additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#).