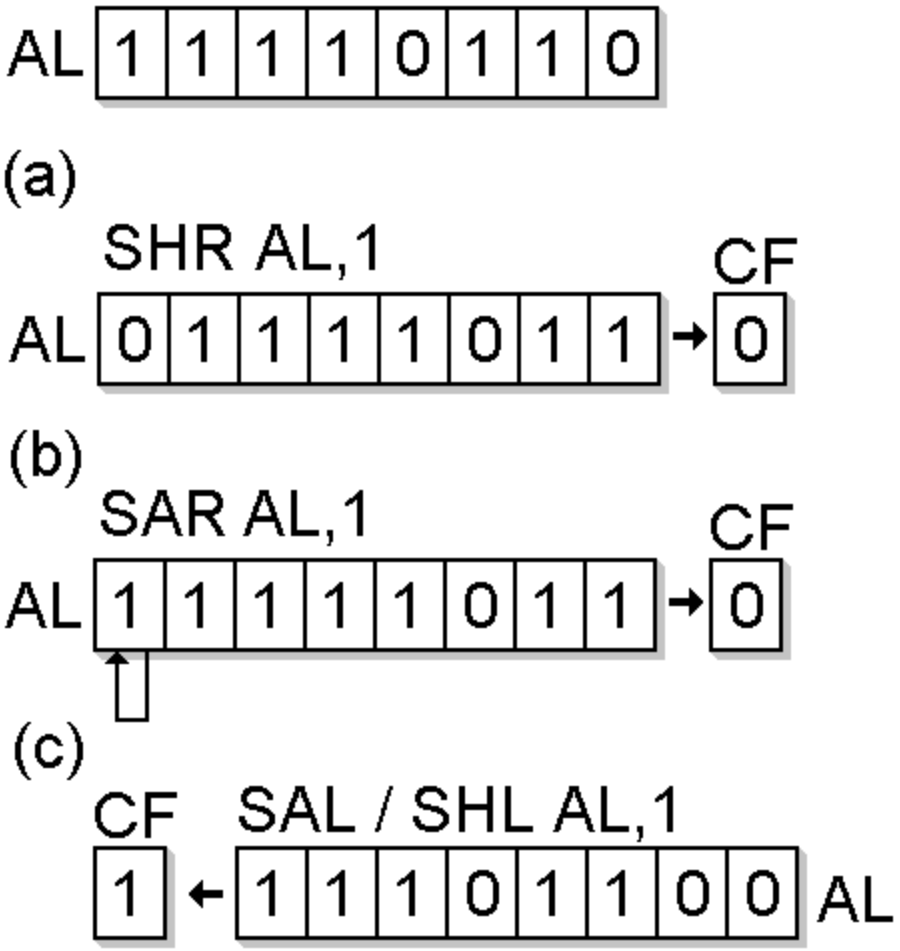
X86 Assembly

From Wikibooks, the open-content textbooks collection

Contents


AL 1 1 1 1 0 1 1 0
(a)
SHR AL,1
CF
AL 0 1 1 1 1 0 1 1 0
(b)
SAR AL,1
CF
AL 1 1 1 1 1 0 1 1 0
(c)
CF
SAL / SHL AL,1
1 1 1 1 0 1 1 0 0 AL

The Wikibook of

x86 Assembly Language

Introduction

x86 Assembly

Why Learn Assembly?

Assembly is the most primitive tool in the programmers toolbox. Entire software projects can be written without ever once looking at a single line of assembly code. So the question arises: why learn assembly? Assembly language is the closest form of communication that humans can engage in with a computer. Using assembly, the programmer can precisely track the flow of data and execution in a program. Also, another benefit to learning assembly, is that once a program has been compiled, it is difficult--if not impossible--to decompile the code. That means that if you want to examine a program that is already compiled, you will need to examine it in assembly language. Debuggers also will frequently only show the program code in assembly language. If nothing else, it can be beneficial to learn to read assembly language, if not write it.

Assembly language is also the preferred tool, if not the only tool available 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 programs written in other languages. 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 a section of code. 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 assembly language anyway.

Who is This Book For?

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

How is This Book Organized?

The first section will talk about the x86 family of chips, and will introduce the basic

instruction set. The second section will talk about the differences between the syntax of different assemblers. The third section will talk about some of the additional instruction sets available, including the Floating-Point operations, the MMX operations, and the SSE operations.

The fourth section will talk about 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, tasks such as enabling and disabling interrupts, enabling protected mode, accessing the Control Registers, creating a Global Descriptor Table, etc. all need to be handled in assembly. The fourth section will also talk about how to interface 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 deal with the standard x86 chipset, will talk about the basic x86 computer architecture, and will 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 and 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

x86 Assembly

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*, 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 noted 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 to "hand assemble" 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 the 6502 machine code for use on his initial Apple I computer. It should be noted, however, that such tasks for commercially distributed software are such rarities 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 just 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 assembler is the GAS assembler, which uses the AT&T syntax for writing code, or Netwide Assembler which is 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.
3. The list of available software interrupts, and their functions, is different on Windows and Linux.
4. The list of available code libraries is 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, etc.

Which Assembler is Best?

The short answer is that none of the assemblers are better than the others, 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 and 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 certainly is nice. 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 or 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 easily be optimized sufficiently if optimization techniques are understood and planned for from the start.

However, learning assembly gives a particular insight into how your computer works on the inside. When you program in a higher-level language like C, or Ada, or even Java and Perl, all your code will eventually need to be converted into terms of 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 will separate labels and instructions into separate columns, AND keep labels on their own lines:

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

So there are a million 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.

X86 Family

x86 Assembly

The x86 family of microprocessors is a very large family of chips with a long history. This page will talk about the specifics of each different processor in this family. x86 microprocessors are also called “IA-32” processors.

Intel x86 Microprocessors

Wikipedia has related information at ***List of Intel microprocessors***.

8086/8087 (1978)
: The 8086 was the original Intel Microprocessor, with the 8087 as its floating-point coprocessor. The 8086 was Intel's first 16-bit microprocessor.

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.

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, as opposed to the “Real Mode” that the earlier models used. All x86 chips can be made to run in real mode or in protected mode.

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 bus. 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 found to have defective FPU's 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 fanless 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 modifed 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. It is currently the fastest x86 chip on the market with respect to clock speed, capable of up to 3.8 GHz. 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.

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 megabytes in comparison to other chips kilobyte size cache) than the Pentium microprocessors.

AMD x86 Compatible Microprocessors

Wikipedia has related information at ***List of AMD 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 800MHz 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 Assembly

x86 Architecture

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

Wikipedia has related information at ***Processor register***.

General Purpose Registers (GPR)

The 8 GPRs are :

1. EAX : Accumulator register. Used in arithmetic operations.
2. ECX : Counter register. Used in shift/rotate instructions.
3. EDX : Data register. Used in arithmetic operations and I/O operations.
4. EBX : Base register. Used as a pointer to data (located in DS in segmented mode).
5. ESP : Stack Pointer register. Pointer to the top of the stack.
6. EBP : Stack Base Pointer register. Used to point to the base of the stack.
7. ESI : Source register. Used as a pointer to a source in stream operations.
8. EDI : Destination register. Used as a pointer to a destination in stream operations.

Each of the GPR are 32 bits wide and are said to be Extended Registers (thus their Exx name). Their 16 Least Significant Bits (LSBs) can be accessed using their unextended parts, namely AX, CX, DX, BX, SP, BP, SI, and DI.

The extended registers can be separated into "high" (the 16 Most Significant Bits) and "low" (the 16 Least Significant Bits) portions. Thus an extended register has the form:

[HHHHHHHHHHHHHHHHLLLLLLLLLLLLLLLL]

(Here, an H or an L denotes a single bit.) which can also be expressed as:

[HW|LW]
Where HW and LW denote "High Word" and "Low Word" respectively.

For the 4 first registers (AX, CX, DX, BX), the 8 Most Significant Bits (MSBs) and the 8 LSBs of their low word can also be accessed via AH, CH, DH, BH and AL, CL, DL, BL respectively.

AH is an abbreviation for "AX High". This term originates from the fact that the low word of the register can be decomposed into its high and low *bytes*. The CH, DH, and BH mnemonics are to be interpreted in a similar fashion.

Likewise, AL is an abbreviation for "AX Low". CL, DL, and BL are similiarily named.

Segment Registers

The 6 Segment Registers are:

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

Most applications on most modern operating systems (like 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 FS or GS is an exception to this rule, to be used to point at thread-specific data.

EFLAGS Register

The EFLAGS is a 32 bits register used as a vector to store and control 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 in of memory references are 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 to the ordering of the bytes, not bits.

So the 32 bit value B3B2B1B0 on an x86 would be represented in memory as:

Little endian representation

Byte 0	Byte 1	Byte 2	Byte 3

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

Little endian example

D4	83	A5	1B

Thus 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. A number's sign is changed by inverting all of the bits and adding one.

	0001
is inverted to:	1110
adding one nets:	1111

0001 represent decimal 1

1111 represent decimal -1

Addressing modes

Addressing modes: indicates the manner in which the operand is accessed

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, 0x010C ; moves value of 0x10C into register ax
```

Direct memory addressing
(operand address is in the address field)

```
mov ax, [102h] Actual address is DS:0 + 102h
```

Direct offset addressing
(uses arithmetics to modify address)

```
byte_tbl db 12,15,16,22,..... ;Table of bytes
mov al,byte_tbl+2
mov al,byte_tbl[2] ; same as the former
```

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

Base Displacement

```
mov ax, arr[bx] where bx is the displacement inside that array
```

Base-index

```
mov ax,[bx + di]
```

For example, if we are talking about an array, bx is the base of the address, and di is the index of the array.

Base-index with displacement

```
mov ax,[bx + di + 10]
```

Stack

The stack is a Last In First Out (LIFO) stack; 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
```

You push the value in AX onto the top of the stack, which now holds the value $006A

```
push bx
```

You do the same thing to the value in BX; the stack now has $006A and $F79A

```
push cx
```

Now the stack has $006A, $F79A, and $1124

```
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 last element pushed onto the stack into CX, $1124; the stack now has $006A and $F79A

```
pop bx
```

Pop the last element pushed onto the stack into BX, $F79A; the stack now has just $006A

```
pop ax
```

Pop the last element pushed onto the stack into AX, $006A; the stack is 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, most 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 behaviour 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 (or shifted 4 bits to the left) and the offset is added to the result. This provides a usable space of 1 MB. However, a quirk of 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 1MB 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 64KB long, COM files could not be larger than that (in fact, they had to fit into 65280 bytes, since DOS used the first 256 of a segment for housekeeping data); for many years this wasn't a problem.

Protected Mode

Flat Memory Model

If programming in a modern 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

Comments

Comments

When writing code, it is very helpful to use some comments to explain 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 ";".

Here is an example:

```
Label1:
   mov ax, bx    ;we move 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.

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

x86 Assembly

x86 assembly has a number of differences between architectures that are 16 bits, 32 bits, and 64 bits. This page will talk about some of the basic differences between architectures with different bit widths.

The 8086 Registers

All the 8086 registers were 16-bit wide. The 8086 registers are the following: AX, BX, CX, DX, BP, SP, DI, SI, CS, SS, ES, DS, IP.

Also on any Windows-based system, by entering into DOS shell you can run a very handy program called "debug.exe", very useful for learning about 8086 and is shipped along with all Windows versions.

AX, BX, CX, DX
: These registers can also be addressed as 8-bit registers. So AX = AH (high 8-bit) and AL (low 8-bit).

So the problem was this: how can a 20-bit address space be referred to by the 16-bit registers? To solve this problem, they came up with segment registers CS (Code Segment), DS (Data Segment), ES (Extra Segment), and SS (Stack Segment). To convert a 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.

Example

If CS = 0x258C and IP = 0x0012 (the "0x" prefix denotes hexadecimal notation), then CS:IP will point to a 20 bit address equivalent to "CS * 16 + IP" which will be = 0x258C * 0x10 + 0x0012 (Remember: 16 decimal = 0x10)

So CS:IP = CSx16 + IP = 0x258C*0x10 + 0x0012 = 0x258D2. The 20-bit address is known as an Absolute address and the Segment:Offset representation (CS:IP) is known as a Segmented Address.

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 = B000x10+8000 = B0000+8000 = B8000 and B200:6000 = B200x10+6000 = B2000+6000 = 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 0x10 and the offset is equal to the remainder, only one segmented address will be generated. (No offset will be greater than 0x0f.) Physical address B8000 maps to (B8000/10): (B8000%10) 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)

The A20 Gate Saga

Like said earlier also, the 8086 processor had 20 address lines (from A0 to A19), so the total memory addressable by it was 1MB (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 64Kb (or 2 to the power 16) of memory. So this made it possible for a program to access the whole of 1MB of memory.

But with segmentation scheme also came a side effect. Not only could your code refer to the whole of 1MB 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:-

```
Segment:Offset = Segment x 16 + 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 & Max value for Offset = FFFF

Now, lets convert, FFFF:FFFF into its 20-bit linear address, bearing in mind 16 is represented as 10 in hexadecimal :-

So we get, FFFF:FFFF = FFFF x 10h + FFFF = FFFF0 + FFFF = FFFF0 + (FFF0 + F) = FFFFF + FFF0 = 1MB + FFF0

- Note: **FFFFF (is hexadecimal) and is equal to 1MB** (one megabyte) and

```
FFF0 is equal to 64Kb minus 16 bytes.
```

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

Notice the use of the word "refer" and not "access". 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 1MB 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 1Mb + 1, this will get wrapped around and point to Zeroth location in memory, likewise 1MB+2 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 fewer bytes shorter. Using this technique it was possible for them to access 32kb of top memory area (that is 32kb touching 1MB boundary) and 32kb 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 64Kb (or 2 to the power 16) area of memory. Now if you make your segment register point to 32kb below 1MB mark you can access 32KB upwards to touch 1MB boundary and then 32kB further which will ultimately get wrapped to the bottom most 32kb.

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 640KB memory?", 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.

32-Bit Addressing

32-bit addresses can cover memory up to 4Gb 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.

X86 Instructions

x86 Assembly

Wikipedia has related information at ***X86 instruction listings***.

These pages are going to 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.

Wikipedia has related information at ***X86 assembly language***.

If you need more info, go to [1].

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 compilers.

Instr src, dest **GAS Syntax**

Instr dest, src **Intel syntax**

Data Transfer

x86 Assembly

Data transfer instructions

Move

mov src, dest **GAS Syntax**

mov dest, src **Intel syntax**

Move

The `mov` instruction copies the `src` operand in 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
        .global _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 2

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

        movw    value(, %ebx, 1), %bx
        # %ebx is now 0

# Linux sys_exit
        mov     $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.

Operands

src

- Register
- Memory

dest

- Register
- Memory

Modified flags

- No FLAGS are modified by this instruction

Example

```
.data

value:
       .long   2

.text
```

```
        .global _start

_start:
        movl    $54, %ebx

        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), %aw
        # value is now 0x00020000 = 131072
        # %eax is now 0

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

Move and Extend

movz src, dest **GAS Syntax**

movz dest, src **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 an unsigned small value to a bigger register.

Operands

src

- Immediate
- Register
- Memory

dest

- Register
- Memory

Modified flags

- No FLAGS are modified by this instruction

Example

```
.data

value:
       .long   34000
byteval:
       .byte   204

.text
       .global _start

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

       movzwl  %ax, value
       # value is now 204

       movzbl  byteval, %esi
       # %esi is now 204

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

movs src, dest **GAS Syntax**

movs dest, src **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` the sign of `src`.

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

Operands

src

- Immediate
- Register
- Memory

dest

- Register
- Memory

Modified flags

- No FLAGS are modified by this instruction

Example

```
.data

value:
       .long   34000
byteval:
       .byte   -204

.text
       .global _start

_start:
       movsbw  byteval, %ax
       # %eax is now -204

       movswl  %ax, value
       # value is now -204

       movsbl  byteval, %esi
       # %esi is now -204

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

Move by Data Size

movsb

Move byte

The `movsb` instruction copies one byte from the location specified in `esi` to the location specified in `edi`.

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
rep movsb

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`.

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
rep movsw
; due to endianess, the resulting mystr2 would be aAbBcC\0a

section .bss
mystr2: resb 8

section .data
mystr db "AaBbCca", 0x0
```

Control Flow

x86 Assembly

Comparison Instructions

test arg1, arg2 **GAS Syntax**

test arg1, arg2 **Intel syntax**

performs a bit-wise AND on the two operands and sets the flags, but does not store a result.

cmp arg1, arg2 **GAS Syntax**

cmp arg1, arg2 **Intel syntax**

performs a subtraction between the two operands and sets the flags, but does not store a result.

Jump Instructions

Unconditional Jumps

jmp loc

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

Jump on Equality

je loc

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 5 and 5 are not equal.
equal:
; if it jumped here, then this means 5 and 5 are equal
```

jne loc

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

Jump if Greater

jg loc

Loads EIP with the specified address, if first operand of previous CMP instruction is greater than the second (performs signed comparison).

jge loc

Loads EIP with the specified address, if first operand of previous CMP instruction is greater than or equal to the second (performs signed comparison).

ja loc

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

jae loc

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

Jump if Less

jl loc

Loads EIP with the specified address, if first operand of previous CMP instruction is less than the second (performs signed comparison).

jle loc

Loads EIP with the specified address, if first operand of previous CMP instruction is less than or equal to the second (performs signed comparison).

jb loc

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

jbe loc

Loads EIP with the specified address, if first operand of previous CMP instruction is less than or equal to the second. `jbe` is the same as `jle`, except that is performs an unsigned comparison.

Jump on Overflow

jo loc

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

Jump on Zero

jnz loc

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

jz loc

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

Function Calls

call proc

pushes the value EIP+4 onto the top of the stack, and 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 stack the specified number of times. 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
start_loop:
; the code here would be executed 5 times
loop start_loop
```

`loop` does not set any flags.

loopx arg

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

- loope
- loopne
- loopnz
- loopz

Enter and Leave

enter arg

Creates a stack frame with the specified amount of space allocated on the stack.

leave

destroys the current stack frame, and restores the previous frame

Other Control Instructions

hlt

Halts the processor

nop

"No Operation". This instruction doesnt do anything, but wastes an instruction cycle in the processor. This instruction is often translated to an **XCHG** operation with the operands **EAX** and **EAX**.

lock

asserts #LOCK

wait

waits for the CPU 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 **GAS Syntax**

add dest, src **Intel syntax**

This adds `src` to `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.

sub src, dest **GAS Syntax**

sub dest, src **Intel syntax**

Like ADD, only it subtracts source from target instead.

mul arg

This multiplies "arg" by the value of corresponding byte-length in the A register, see table below.

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.

div arg

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

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

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 + carry flag` 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.

sbb src, dest **GAS Syntax**

sbb dest, src **Intel syntax**

Subtract with borrow. Subtracts `src + carry flag` 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.

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 **GAS Syntax**

and dest, src **Intel syntax**

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

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

or src, dest **GAS Syntax**

or dest, src **Intel syntax**

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

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

xor src, dest **GAS Syntax**

xor dest, src **Intel syntax**

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

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

not arg

performs a bit-wise inversion of arg. For example:

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

Shift and Rotate

x86 Assembly

Logical Shift Instructions

In a logical shift instruction, the bits that slide off the end disappear, and the spaces are always filled with zeros. Logical shift is best used with unsigned numbers.

shr arg

Logical shifts arg to the right

shl arg

Logical shift arg to the left

Arithmetic Shift Instructions

In an arithmetic shift, the bits that "slide off the end" disappear. 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 arg

arithmetic shift to the right. spaces are filled with sign bit (to maintain sign of original value).

sal arg

arithmetic shift to the left. spaces are filled with zeros

Shift With Carry Instructions

A Logical Shift, and the bit that slides off the end goes into the carry flag.

scr arg

shift with carry to the right

scl arg

shift with carry to the left

Rotate Instructions

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

ror arg

rotate to the right

rol arg

rotate to the left

Other Instructions

x86 Assembly

Stack Instructions

push arg

This instruction decrements the stack pointer and loads 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 would be: [5]

`push ebx` the stack would be: [6] [5]

`pop eax` the topmost item (which is 6) would be stored in eax. the stack would be: [5]

`pop ebx` ebx would be equal to 5. the stack would now be 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 intruction 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.

Flags instructions

Interrupt Flag

sti

Sets the interrupt flag. Processor can accept interrupts from peripheral hardware. This flag should be kept set under normal execution.

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

std

Sets the direction flag. Normally, when using string instructions the data pointer gets incremented with each iteration. When the direction flag is set, the data pointer is decremented instead.

cld

clears the direction flag

Carry Flag

stc

sets the carry flag

clc

clears the carry flag

cmc

Complement 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 **GAS Syntax**

in dest, src **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 **GAS Syntax**

out dest, src **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.

sysexit

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

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?

Interrupts do exactly what the name suggests: they interrupt the control flow of the x86 processor. When an interrupt is triggered, the current program stops, and the processor jumps to a special program called an "Interrupt Service Routine" (ISR). Each ISR is a program in memory that handles a particular interrupt. When the ISR is finished, the microprocessor normally jumps right back to where it was in the original program (however, there are interrupts that don't do this). In the case of hardware interrupts, the program doesn't even have to know that it got interrupted: the change is seamless.

In modern operating systems, the programmer doesn't often need to use interrupts. In Windows, for example, the programmer conducts business with the Win32 API. However, these API calls will interface with the kernel, and often times the kernel will trigger interrupts to perform different tasks. However, 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 calls the specified interrupt. for instance:

```
int $0x0A
```

Will call 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 occurrance 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, may terminate the program causing the exception.

Further Reading

A great list of interrupts **for DOS and related systems** is at Ralph Brown's Interrupt List.

x86 Assemblers

x86 Assembly

Wikipedia has related information at ***Assembler***.

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)

Wikipedia has related information at ***GNU Assembler***.

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 because inline assembly instructions in the GCC compiler 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.

Microsoft Macro Assembler (MASM)

Wikipedia has related information at ***Microsoft Macro Assembler***.

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 can still be obtained from microsoft as either a download from MSDN, or as part of the Microsoft DDK. The currently available version of MASM is version 8.x.

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

Netwide Assembler (NASM)

Wikipedia has related information at ***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://nasm.sourceforge.net/doc/html/nasmdoc1.html .

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)

Wikipedia has related information at ***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
- 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/

GAS Syntax

x86 Assembly

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 accept both.

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

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

will move the 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 = short (16 bit integer) or 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 "$".

Introduction to the GNU as assembler

This section is written as a short introduction to GNU as (gas), an assembler that can assemble the x86 assembly language. gas is part of the GNU Project, which gives it the following nice properties:

- It is freely available.
- 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 or Mingw. The remainder of this introduction assumes you have installed gas and know how to open a command-line interface and edit files.

Generating assembly from C code

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.

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.exe hello.c
```

This should compile the C file and create an executable file called "hello_c.exe". 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.exe
```

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 x86 assembly language. Type the following at the prompt:

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

This should create a file called "hello.s" (".s" is the file extension that the GNU system gives to assembly files). To compile the assembly file into an executable, type:

```
gcc -o hello_asm.exe 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.exe
```

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":

```
        .file   "hello.c"
        .def    ___main;        .scl    2;      .type   32;     .endef
        .text
LC0:
        .ascii "Hello, world!\12\0"
.globl _main
        .def    _main;  .scl    2;      .type   32;     .endef
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        call    __alloca
        call    ___main
        movl    $LC0, (%esp)
        call    _printf
        movl    $0, %eax
        leave
        ret
        .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:

```
        .text
LC0:
        .ascii "Hello, world!\12\0"
.globl _main
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        call    __alloca
        call    ___main
        movl    $LC0, (%esp)
        call    _printf
        movl    $0, %eax
        leave
        ret
```

"hello.s" line-by-line

```
        .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.

```
LC0:
        .ascii "Hello, world!\12\0"
```

This code declares a label, then places some raw ASCII text into the program, starting at the label's location. The "\12" 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.

```
.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.

```
_main:
```

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

```
pushl   %ebp
movl    %esp, %ebp
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. 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.

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

This code "and"s ESP with 0xFFFFFFF0, 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.

```
movl    $0, %eax
movl    %eax, -4(%ebp)
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.

```
call    __alloca
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.

```
movl    $LC0, (%esp)
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.

```
        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.

```
        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.

```
        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:

```
        .file   "hello2.c"
        .def    ___main;        .scl    2;      .type   32;     .endef
        .text
LC0:
        .ascii "Hello, world!\12\0"
.globl _main
        .def    _main;  .scl    2;      .type   32;     .endef
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        andl    $-16, %esp
        movl    $0, %eax
        movl    %eax, -16(%ebp)
        movl    -16(%ebp), %eax
```

```
        call    __alloca
        call    ___main
        movl    $LC0, -4(%ebp)
        movl    $-11, (%esp)
        call    _GetStdHandle@4
        subl    $4, %esp
        movl    %eax, -12(%ebp)
        movl    $0, 16(%esp)
        leal    -8(%ebp), %eax
        movl    %eax, 12(%esp)
        movl    $14, 8(%esp)
        movl    -4(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    -12(%ebp), %eax
        movl    %eax, (%esp)
        call    _WriteFile@20
        subl    $20, %esp
        movl    $0, %eax
        leave
        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:

```
        .text
LC0:
        .ascii "Hello, world!\12"
.globl _main
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        pushl   $-11
        call    _GetStdHandle@4
        pushl   $0
        leal    -4(%ebp), %ebx
        pushl   %ebx
        pushl   $14
        pushl   $LC0
        pushl   %eax
        call    _WriteFile@20
        movl    $0, %eax
        leave
        ret
```

Analyzing line-by-line:

```
        pushl   %ebp
        movl    %esp, %ebp
        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.

```
        pushl   $-11
        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.

```
        pushl   $0
        leal    -4(%ebp), %ebx
```

```
pushl   %ebx
pushl   $14
pushl   $LC0
pushl   %eax
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 ("lea") 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.

```
movl    $0, %eax
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:

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:

http://sourceware.org/binutils/docs-2.17/as/index.html

- Reverse Engineering/Calling Conventions

MASM Syntax

x86 Assembly

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 usage of operands order and instruction suffixes.

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, 0x05
```

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.

HLA Syntax

HLA Syntax

HLA is an assembler front-end created by Randall Hyde. 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.

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 doesnt 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.

The Art of Assembly

HLA was first popularized in the book by Randal Hyde, named "The Art of Assembly". That book is available at most bookstores.

FASM Syntax

This book or module has been nominated for cleanup because:
page needs general work
Please edit this module to improve it. See this module's talk page for discussion.

FASM is an assembler for the IA-32 architecture. The name stands for "flat assembler". FASM itself is written in assembly language and is also available on DOS, DexOS, Linux, Windows, and MenuetOS systems. It shatters the "assembly is not portable at all" myth. FASM has some features that are advanced for assembly languages, such as macros, structures, and "virtual data". FASM contains bindings to the MSWindows GUI and OpenGL.

$FFFF 0xffff ffffh

FASM supports all popular syntaxes of hex numbers.

@@ @f @b

Anonymous labels are supported. Example:

```
@@: inc eax
    push eax
    jmp @b     ; This will result in a stack fault sooner or later
```

$

$ describes current location. Useful for determining the size of a block of code or data. Example of use:

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

Local Labels

Local Labels, which begin with a . (a period)

```
globallabel:

.locallabelone:

.locallabeltwo:

globallabel2:

.locallabelone:
```

```
.locallabeltwo:
```

You can reference local labels from their global label. For example:

```
globallabel.locallabelone
```

Macros

Macros in FASM are described in a C-like manner and are created like this:

```
macro (name) (parameters) {
  macro code.
}
```

For example, the following could be used to overload the *mov* instruction to accept three parameters in FASM:

```
   macro mov op1,op2,op3
    {
     if op3 eq
       mov   op1,op2
     else
       mov   op1,op2
       mov   op2,op3
     end if
    }
```

if op3 eq means "If the 3rd parameter (op3) equals nothing, or blank" then do a normal mov operation. Else, do the 3 way move operation.

External links

- FASM website
- FASM official manual

NASM Syntax

x86 Assembly

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

NASM Syntax

Wikipedia has related information at ***NASM***.

NASM syntax looks like:

```
mov ax, 9
```

This loads the number 9 into register ax. Notice that the instruction format is "dest, src". This follows the Intel style x86 instruction formatting, as opposed to the AT&T style used by the GNU Assembler. Note for people using gdb with nasm, you can set gdb to use Intel-style disassembly by issuing the command:

```
set disassembly-flavor intel
```

NASM Comments

A single semi-colon is used for comments, and can be used like a double slash in C/C++.

Example I/O (Linux)

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 "input"
mov    edx, 1          ; input length (one byte)
mov    ecx, variable   ; address to pass to
mov    ebx, 1          ; read from standard input
int 0x80               ; call the kernel
```

Outputting follows a similar convention:

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

Passing values to the registers in different orders won't affect the execution when the kernel is called, but deciding on a methodology can make it drastically easier to read.

Floating Point

x87 Coprocessor

The original x86 family members had a separate math coprocessor that would handle the floating point arithmetic. The original coprocessor was the 8087, and all FPUs 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

This page will talk about these 2 points in detail. The FPU, even when it is integrated into an x86 chip is still called the "x87" section, even though it is part of the x86 chip. 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, 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.

Floating-Point Instruction Set

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, FLDENVD, FNSAVED, FNSTENVD, FPREM1, FRSTORD, FSAVED, FSIN, FSINCOS, FSTENVD, FUCOM, FUCOMP, FUCOMPP

Added with Pentium Pro

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

Added with Pentium 4 supporting SSE3

as part of the SSE3 branding

FISTTP (x87 to integer conversion)

Further Reading

- Reverse Engineering/Floating Point Numbers
- Floating Point

MMX

Saturation Arithmetic

Wikipedia has related information at ***MMX***.

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 good sense: if we make white brighter, it shouldnt become black.

Single Instruction Multiple Data (SIMD) Instructions

MMX registers are 64 bits wide, but they 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, so it's a waste of time to even try. 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 in the last box, our arithmetic "saturates", and 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. The problem is that the MMX instructions run slightly slower then the regular arithmetic instructions, the FPU can't be used when the MMX register is running, and MMX registers use saturation arithmetic.

MMX Registers

There are 8 64-bit MMX registers. These registers overlay the FPU stack register. **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 to 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 here is an example of a C routine calling assembly language with MMX code (NOTE: Borland compatible C++ Example)....

```
//-----------------------------------------------------
// A simple example using MMX to copy 8 bytes of data
// From source s2 to destination s1
//-----------------------------------------------------
void __fastcall CopyMemory8(char *s1, const char *s2)
{
   __asm
   {
    push edx
    mov ecx, s2
    mov edx, s1
    movq   mm0, [ecx   ]
    movq   [edx   ], mm0
    pop edx
    emms
   }
}
```

SSE

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

Wikipedia has related information at ***Streaming SIMD Extensions***.

SSE stands for **Streaming SIMD Extensions**. SSE is essentially the floating-point equivalent of the MMX instructions. SSE registers are 128 bits, and can be used to perform operations on either two 64 bit floating point numbers (C double), or 4 32-bit floating point numbers (C float).

SSE

128-bit registers

XMM0 XMM1 XMM2 XMM3 XMM4 XMM5 XMM6 XMM7

SSE2

Same as MMX and SSE

SSE3

Same as MMX and SSE

3D Now

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

Wikipedia has related information at ***3DNow%21***.

3d Now! is AMD's extension of the MMX instruction set (K6-2 and more recent) for with floating-point instruction. This page will talk about the 3D Now! instruction set, and how it is used.

Advanced x86

x86 Assembly

The chapters in the x86 Assembly wikibook labled "Advanced x86" chapters are all specialized topics that might not be of interest to the average assembly programmer. However, these chapters will be of some interest to people who would like to work on low-level programming tasks, such as bootloaders, device drivers, and Operating System kernels. A reader does not need to read the following chapters to say they "know assembly", although they certainly are interesting.

High-Level Languages

x86 Assembly

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 has a one-to-one correlation 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**.

C Calling Conventions

CDECL

In most C compilers, the CDECL calling convention is the de facto standard. However, the programmer can specify that a function be implemented using CDECL by prepending the function declaration with the keyword **__cdecl**. Sometimes a compiler can be instructed to override cdecl as the default calling convention, and this declaration will force the compiler not to override the default setting.

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. The function name is prepended with an underscore.

CDECL functions are capable of accepting variable argument lists.

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. STDCALL functions can be declared using the **__stdcall** keyword on many compilers. 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. Function name is prepended with an underscore
4. Function name is suffixed with an "@" sign, followed by the number of bytes of arguments being passed to it.

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 EAX/AX/AL
2. The second 32-bit (or smaller) argument is passed in EDX/DX/DL
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 appended 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

Th 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++

Further Reading

For an in depth discussion as to how high-level programming constructs are translated into assembly language, see Reverse Engineering.

- C Programming
- C++
- Reverse Engineering/Calling Conventions
- Reverse Engineering/Examples/Calling Conventions

Machine Language Conversion

x86 Assembly

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.

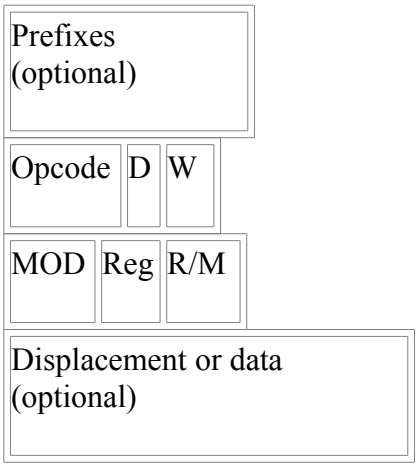
CISC and RISC

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:



Prefixes
: Optional prefixes which change the operation of the instruction

W
: (1 bit) Operation size. 1 = Word, 0 = byte.

D

(1 bit) Direction. 1 = Register is Destination, 0 = Register is source.

Opcode

the opcode is a 6 bit quantity that determines what instruction family the code is

MOD

(2 bits) Register mode.

Reg

(3 bits) Register. Each register has an identifier.

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
11	r/m is treated as a second "reg" field

Reg	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX

011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

r/m	Operand address
000	(BX) + (SI) + displacement
001	(BX) + (DI) + displacement
010	(BP) + (SI) + displacement
011	(BP) + (DI) + displacement
100	(SI) + displacement
101	(DI) + displacement
110	(BP) + displacement unless mod = 00 (see mod table)
111	(BX) + displacement

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 bytecode:

```
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 = 80H F1H 12H
```

x86-32 Instructions (32 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.

x86-64 Instructions (64 bit)

Protected Mode

x86 Assembly

This page is going to discuss the differences between real mode and protected mode operations in the x86 processors. This page 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

Wikipedia has related information at ***X86 assembly programming in real mode***.

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 to provide backwards-compatability with 8086 and 80186 programs.

Protected Mode Operation

Wikipedia has related information at ***X86 assembly programming in protected mode***.

In protected mode operation, the x86 can address 16 Mb or 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

Wikipedia has related information at ***X86 assembly programming in 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. If paging is going to be used, load CR3 with a valid page table, PDPR, or PML4.
4. If PAE (Physical Address Extension) is going to be used, set CR4.PAE = 1.
5. If switching to long mode, set IA32_EFER.LME = 1.
6. Disable Interrupts (CLI).
7. Load an IDT pseudo-descriptor that has a null limit (this prevents the real mode IDT from being used in protected mode)
8. Set the PE bit (and the PG bit if paging is going to be enabled) of the MSW or CR0 register
9. 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)
10. Load data segment registers with valid selector(s) to prevent GP exceptions when interrupts happen
11. Load SS:(E)SP with a valid stack
12. Load an IDT pseudo-descriptor that points to the IDT
13. Enable Interrupts.

Following chapters will talk more about these steps.

Entering Long Mode

To enter Long Mode on an 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 compatiblity mode. A far jump may be executed to switch to long mode. However, the offset must not exceed 32-bit.

Using the CR Registers

The CR registers may only be accessed in protected mode. For this reason, paging and task-switching can only be performed by the processor when in protected mode.

CR0

The CR0 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----|ET|TS|EM|MP|PE|
```

We recognize the PE flag as being the flag that puts the system into protected mode.

PG
: The PG flag turns on memory paging. We will talk more about that in a second.

MP
: The "Monitor Coprocessor" flag. This flag controls the operation of the "WAIT" instruction.

ET
: 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.

EM
: The Emulate Flag. When this flag is set, coprocessor instructions will generate an exception.

TS
: The Task Switched flag. This flag is set automatically when the processor switches to a new task.

CR2

CR2 contains a value called the **Page Fault Linear Address** (PFLA). When a page fault occurs, the address accessed 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 the microprocessor will perform, in order to make the available amount of memory in a system appear larger than it actually is, and be more dynamic than it actually is. In a paging system, a certain amount of space is laid aside on the harddrive (or on any secondary storage) called the **paging file** (or **swap partition**). The physical RAM, combined with this paging file are called the **virtual memory** of the system.

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 updated 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.

pages that aren't in frequent use may be moved to the paging file on the harddisk drive, to free up space in the physical RAM for pages that need to be accessed more frequently, or that require faster access. Reading and writing pages to the harddrive is a slow operation, and frequent paging may increase the strain on the disk, so in some systems with older drives, it may be a good precaution to turn the paging capabilities of the processor off. This is accomplished by toggleing the PG flag in the CR0 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 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, for instance power management related, without disrupting the operating system or other software.

Global Descriptor Table

x86 Assembly

The Global Descriptor Table (GDT) is a table in memory that defines the actions of the processor segment registers. The GDT will define the characteristics of the different segment registers, it will define the characteristics of global memory, and it helps to ensure that the protected mode operates smoothly.

GDTR

The GDT is pointed to by a special register in the x86 chip, the **GDT 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]
```

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.

Wikipedia has related information at ***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, or a program invokes an interrupt, or the hardware raises an interrupt, the processor will use 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. At least, FLAGS and CS:IP will be saved, and the ISR's CS:IP will be 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 (the same holds for 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 would go to location [4*14H]=[50H]. At location 50H would be the new IP value, and at location 52H would be the new CS value. Hardware and software interrupts would all be 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* modifed 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 routines, 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 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]
```

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

In x86, interrupts can be disabled using the **cli** command. This command takes no arguments. To enable interrupts, the programmer can use the **sti** command. Interrupts need to be disabled when performing important system tasks, because you don't want the processor to operate in an unknown state. For instance, when entering protected mode, we want to disable interrupts, because we want the processor to switch to protected mode

before anything else happens. Another thing you may want to do is load an IDT pseudo-descriptor with a null limit if for example, you are switching to real-mode to protected mode because the IDT format is different between the two modes.

Bootloaders

Wikipedia has related information at ***Bootloader***.

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 often 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.

On IBM PC compatibles, the first program to load is the Basic Input/Output System (BIOS). The BIOS performs many tests and initialisations, then the BIOS 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.

This boot loader will either load yet another boot loader somewhere else, or load enough of an Operating System to start running it. The main focus of this article will be the final stage before the OS is loaded.

Some tasks that this last boot loader may perform:

- Allocate more stack space
- Establish a GDT
- Enter Protected Mode
- Load the Kernel

Bootloaders are almost exclusively written 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 benefit from (or require) access to some low-level features. Many bootloaders will be very simple, and will only load the kernel into memory, leaving the kernel's initialisation procedure to create a GDT and enter protected mode. If the GDT is very large or complicated, the bootloader may not be physically large enough to create it.

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 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 F000:FFF0 (at this stage it is operating in **Real Mode**). In IBM PC compatibles, 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 PIC 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 remains 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 becoming 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 features in the way of boot loading.

However, a new bootstrap technology, the EFI, 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.

Specifications

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 *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. (I.e. it JMPs to that address)
4. CS, DS and ES will be set to 0000.
5. There are some conventions that need to be respected if the disk is to be readable under certain 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 (though you must also ensure the *rest* of the disk holds a valid FAT12 file system as well).
6. While standard routines installed by the BIOS are available to the bootloader, the operating system has not been loaded yet, and you cannot rely on loaders or OS memory management. Any data the boot loader needs must either be included in the first sector (be careful not to execute it!) 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 RAM that may be required by interrupts.
7. The OS code itself (or the next bootloader) will need to loaded somewhere into RAM as well.
8. The 512-byte stack allocated by the BIOS may be too small for some purposes (remember that unless interrupts are disabled, they can happen at any time). It may be necessary to create a larger stack.

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. Using this will make it easier to use procedures and data within the bootloader (you will not need to add 7C00 to all the addresses). Another option is to set some segment registers to 07C0h, so that the offsets actually start at 0 relative to those segment. Also, some bootloaders copy themselves to other locations in RAM.

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 not impossible to include OS code within the

boot sector and keep it resident after the OS begins.

Here is a simple boot sector demo designed for NASM:

```
ORG 7C00h

JMP short START ;Jump over the data (the 'short' keyword makes the JMP code smaller)

MSG:
DB "Hello World! "
ENDMSG:

START:
MOV CX, 1      ;Write 1 character
MOV BX, 000Fh  ;Colour attribute 15 (white)
XOR DX, DX     ;Start at top left corner

L1:
  MOV SI, MSG  ;Loads the address of the first byte of the message (In this case, 7C02h)
  L2:
    MOV AH, 02
    INT 10h    ;Set cursor position
    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.
    MOV AH, 9
    INT 10h    ;Write character
    INC DL     ;Advance cursor
    CMP DL, 80 ;Wrap around edge of screen
    JNE SKIP
    XOR DL, DL
    INC DH
    CMP DH, 25 ;Wrap around bottom of screen
    JNE SKIP
    XOR DH, DH
    SKIP:

        ;If we're not at end of message, continue
        ;loading characters otherwise return SI
        ;to the start of the message
    CMP SI, ENDMSG
  JNE L2
JMP L1

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 (also 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). 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 adress 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.

Example of a Boot Loader -- Linux Kernel v0.01

```
|
|       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", ie all memory
| over 1Mb is demand-paging. All addresses under 1Mb are guaranteed to match
| their physical addresses.
|
| NOTE1 abouve 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 continuos
| 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
        jmpi    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!
        jmpi    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,dl
        mov dx,head
        mov dh,dl
        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:
```

further reading

- 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--in one way or another--survive 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.

Direct Memory Access

Direct Memory Access

The **Direct Memory Access** chip (DMA) was an important part of the original IBM PC, and it has become an essential component of modern computer systems. DMA allows other computer components to access the main memory directly, without having to manage the data flow through the processor. This is an important functionality, 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 it's 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

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

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 8259-A chip, although modern computers will have a more modern variant. The most common replacement is the APIC[[2]] (Advanced Programmale Inerrupt Controller) which is essentially an extended version of the old PIC chip to maintain backwards compatibility.

Programmable Interrupt Timer

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

The **Programmable Interrupt Timer** (PIT) is an essential component of modern computers, and is an essential part of 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.

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

x86 Assembly

Wikimedia Sources

Wikipedia has related information at ***Assembly language***.

Wikipedia has related information at ***x86***.

- Wikipedia Assembler Article
- C Programming
- C++ Programming
- Operating System Design
- Embedded Systems
- x86 Disassembly
- Floating Point

Books

- 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 Edition. Presently free chapters Available online. Ebook in progress, http://pratik.tambe.ebooksupport.googlepages.com/

Web Resources

- http://developer.intel.com/design/pentiumii/manuals/243191.htm
- 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://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_4699_7980%5E875%5E4622,00.html

Other Assembly Languages

Assembly Language	
x86 Assembly	The Assembly Language used by 32-bit Intel Machines including the 386, 486, and Pentium Family.
MIPS Assembly	A Common RISC assembly set 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	This is the instruction set used with the TI 83 Plus brand of programmable graphing calculators.
360 Assembly	This is the instruction set used with the IBM 360 / 370 / 93xx and z/ System brand of Mainframe computers.
ARM	This is the instruction set used with most 32-bit embedded CPUs, including most PDAs, MP3 players, and handheld gaming units.

(edit template)

Licensing

GNU Free Documentation License

Version 1.2, November 2002

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. Secondarily, 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.

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 for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

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.