X86 Assembly/X86 Architecture

Contents

x86 Architecture

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

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

The 8 GPRs are:

1. Accumulator register (AX). Used in arithmetic operations
2. Counter register (CX). Used in shift/rotate instructions and loops.
3. Data register (DX). Used in arithmetic operations and I/O operations.
4. Base register (BX). Used as a pointer to data (located in segment register DS, when in segmented mode).
5. Stack Pointer register (SP). Pointer to the top of the stack.
6. Stack Base Pointer register (BP). Used to point to the base of the stack.
7. Source Index register (SI). Used as a pointer to a source in stream operations.
8. Destination Index register (DI). Used as a pointer to a destination in stream operations.

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

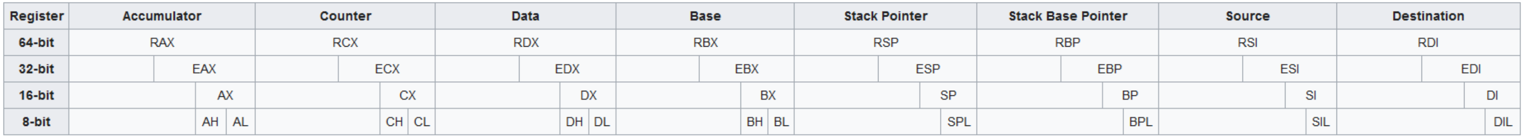
All registers can be accessed in 16-bit and 32-bit modes. In 16-bit mode, the register is identified by its two-letter abbreviation from the list above. In 32-bit mode, this two-letter abbreviation is prefixed with an 'E' (*extended*). For example, 'EAX' is the accumulator register as a 32-bit value.

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

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

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

identifiers to access registers and parts thereof



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

Segment Registers

The 6 Segment Registers are:

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

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

EFLAGS Register

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

The names of these bits are:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	NT	**IOPL**		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

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

The different use of these flags are:

0. CF : Carry Flag. Set if the last arithmetic operation carried (addition) or borrowed (subtraction) a bit beyond the size of the register. This is then checked when the operation is followed with an add-with-carry or subtract-with-borrow to deal with values too large for just one register to contain.

2. PF : Parity Flag. Set if the number of set bits in the least significant byte is a multiple of 2.

4. AF : Adjust Flag. Carry of Binary Code Decimal (BCD) numbers arithmetic operations.

6. ZF : Zero Flag. Set if the result of an operation is Zero (0).

7. SF : Sign Flag. Set if the result of an operation is negative.

8. TF : Trap Flag. Set if step by step debugging.

9. IF : Interruption Flag. Set if interrupts are enabled.

10. DF : Direction Flag. Stream direction. If set, string operations will decrement their pointer rather than incrementing it, reading memory backwards.

11. OF : Overflow Flag. Set if signed arithmetic operations result in a value too large for the register to contain.

12-13. IOPL : I/O Privilege Level field (2 bits). I/O Privilege Level of the current process.

14. NT : Nested Task flag. Controls chaining of interrupts. Set if the current process is linked to the next process.

16. RF : Resume Flag. Response to debug exceptions.

17. VM : Virtual-8086 Mode. Set if in 8086 compatibility mode.

18. AC : Alignment Check. Set if alignment checking of memory references is done.

19. VIF : Virtual Interrupt Flag. Virtual image of IF.

20. VIP : Virtual Interrupt Pending flag. Set if an interrupt is pending.

21. ID : Identification Flag. Support for CPUID instruction if can be set.

Instruction Pointer

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

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

Memory

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

So the 32 bit value B3B2B1B0$_{16}$ on an x86 would be represented in memory as:

Little endian representation

B0	B1	B2	B3

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

Little endian example

D4	83	A5	1B

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

Two's Complement Representation

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

Two's complement example

Start:	0001
Invert:	1110
Add One:	1111

0001 represents decimal 1

1111 represents decimal -1

Addressing modes

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

Register Addressing

(operand address R is in the address field)

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

Immediate

(actual value is in the field)

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

or

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

Direct memory addressing

(operand address is in the address field)

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

Direct offset addressing

(uses arithmetics to modify address)

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

Register Indirect

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

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

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

General-purpose registers (64-bit naming conventions)

Main page: X86 Assembly/16, 32, and 64 Bits

Main page: X86 Assembly/SSE

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

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

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

Stack

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

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

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

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

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

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

CPU Operation Modes

Real Mode

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

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

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

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

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

Protected Mode

Flat Memory Model

If programming in a modern 32-bit operating system (such as Linux, Windows), you are basically programming in flat 32-bit mode. Any register can be used in addressing, and it is generally more efficient to use a full 32-bit register instead of a 16-bit register part. Additionally, segment registers are generally unused in flat mode, and using them in flat mode is not considered best practice.

Multi-Segmented Memory Model

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

Long Mode

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