x86 Processor Architecture

Contents of Lecture:

- Modes of Operation
- Basic Execution Environment

References for Lecture:

KIP R. IRVINE, Assembly Language for x86 Processors, 7th Edition, Chapter 2: x86 Processor Architecture

32-Bit x86 Processors

In this section, we focus on the basic architectural features of the x86 processors, which include both Intel IA-32 and 32-bit AMD processors.

Modes of Operation:

X86 processors have three primary modes of operation: protected mode, real-address mode, and system management mode.

A sub-mode, named *virtual-8086*, is a special case of protected mode. Here are short descriptions of each:

1. Protected Mode:

Protected mode is the native state of the processor (Windows, Linux), in which all instructions and features are available. Programs are given separate memory areas named *segments*, and the processor prevents programs from referencing memory outside their assigned segments.

- ***Virtual-8086 Mode:***

  While in protected mode, the processor can directly execute real-address mode software such as MS-DOS programs in a safe multitasking environment. In other words, if an MS-DOS program crashes or attempts to write data into the system memory area, it will not affect other programs running at the same time. Windows XP can execute multiple separate virtual-8086 sessions at the same time.

2. Real-Address Mode:

Real-address mode implements the programming environment of the Intel 8086 processor with a few extra features, such as the ability to switch into other modes. This mode is available in Windows 98, and can be used to run an MS-DOS program that requires direct access to system memory and hardware devices. Programs running in real-address mode can cause the operating system to crash (stop responding to commands).

3. System Management Mode:

System Management mode (SMM) provides an operating system with a mechanism for implementing functions such as power management and system security. These functions are usually implemented by computer manufacturers who customize the processor for a particular system setup.

Basic Execution Environment

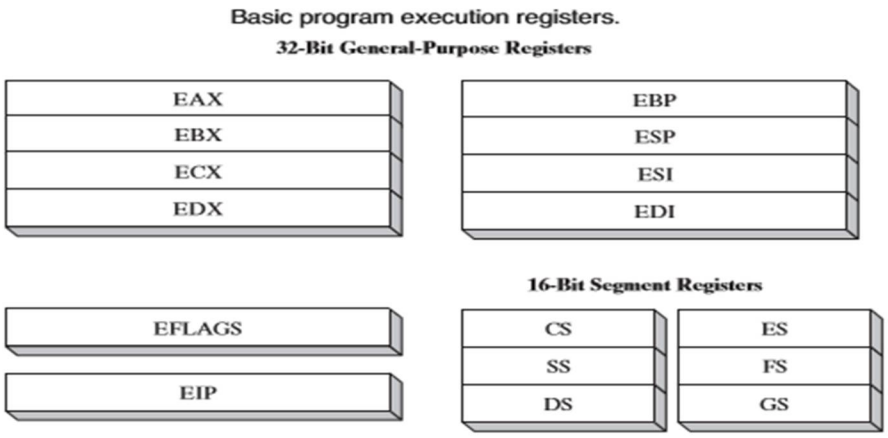
Address Space:

- In 32-bit protected mode, a task or program can address a linear address space of up to 4 GBytes. Beginning with the P6 processor, a technique called ***extended physical addressing*** allows a total of 64 GBytes of physical memory to be addressed.
  - ✓ 32-bit address (4,294,967,295)

- Real-address mode programs, on the other hand, can only address a range of 1 MByte. If the processor is in protected mode and running multiple programs in virtual-8086 mode, each program has its own 1-MByte memory area
  - ✓ 20-bit address (1,048,575)

Basic Program Execution Registers:

- Registers are high-speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory.
  - ✓ When a processing loop is optimized for speed, for example, loop counters are held in registers rather than variables.
- There are:
  - ✓ Eight general-purpose registers.
  - ✓ Six segment registers.
  - ✓ A processor status flags register (EFLAGS).
  - ✓ An instruction pointer (EIP).
- Following figure shows the basic program execution registers.



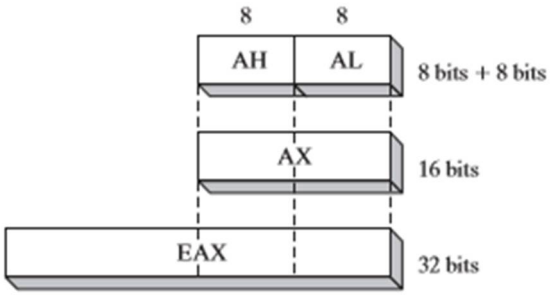
General-Purpose Registers:

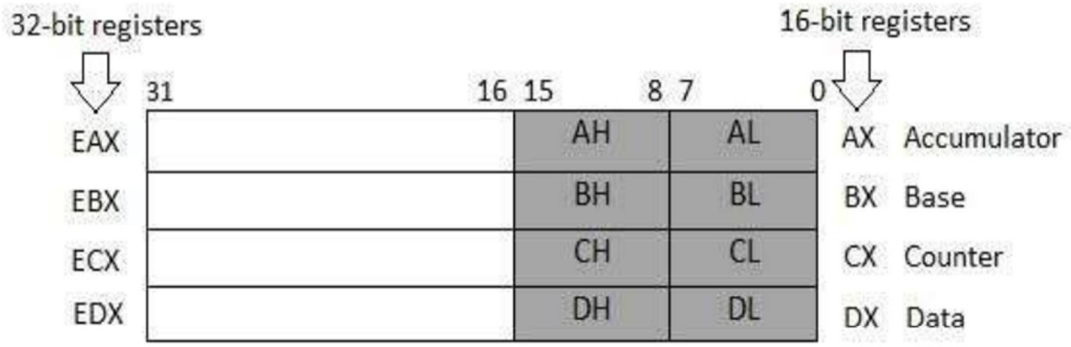
- General-Purpose Registers divide for two groups:
  - ✓ Data Registers
  - ✓ Index and Pointer Registers

Data Registers:

Four 32-bit data registers are used for arithmetic, logical and other operations. These 32-bit registers can be used in three ways:

1. As complete 32-bit data registers: EAX, EBX, ECX, EDX.
2. Lower halves of the 32-bit registers can be used as four 16-bit data registers: AX, BX, CX and DX.
3. Lower and higher halves of the above-mentioned four 16-bit registers can be used as eight 8-bit data registers: AH, AL, BH, BL, CH, CL, DH, and DL





32-Bit	16-Bit	8-Bit (High)	8-Bit (Low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

Index and Pointer Registers:

- The remaining general-purpose registers can only be accessed using 32-bit or 16-bit names, as shown in the following table:

32-Bit	16-Bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Specialized Uses of Some General-Purpose Registers:

Register name	shortcut	Register usage
Data Registers		
EAX	Extended Accumulator Register	Automatically used by multiplication and division instructions and I/O instructions.
EBX	Extended Base Register	Used in indexed addressing.
ECX	Extended Counter Register	The CPU automatically uses ECX as a loop counter
EDX	Extended Data Register	Automatically used by multiplication and division instructions and I/O instructions.

Index and Base Registers		
ESI	Extended Source Index	Used as source index for string operations
EDI	Extended Destination Index	Used as destination index for string operations
ESP	Extended Stack Pointer	• Provides the offset value within the program stack. • SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack.
EBP	Extended Base Pointer	• Mainly helps in referencing the parameter variables passed to a subroutine. • The address in SS register is combined with the offset in BP to get the location of the parameter. • BP can also be combined with DI and SI as base register for special addressing

Segment Registers:

- In real-address mode, 16-bit segment registers indicate base addresses of pre-assigned memory areas named segments.
- In protected mode, segment registers hold pointers to segment descriptor tables

Register name	shortcut	Register usage
CS	Code Segment Register	It contains all the instructions to be executed. And stores the starting address of the code segment
DS	Data segment Register	It contains data, constants and work areas. And stores the starting address of the data segment.
SS	Stack Segment Register	It contains data and return addresses of procedures or subroutines. And stores the starting address of the stack
ES, FS and GS	Extra Segment Register	Provides additional segments for storing data. And stores the starting address of the Extra segment.

Instruction Pointer:

- The EIP, or instruction pointer register:
  - ✓ Contains the offset address of the next instruction to be executed.
  - ✓ EIP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment

EFLAGS Register:

- The EFLAGS (or just Flags) register consists of individual binary bits that control the operation of the CPU or reflect the outcome of some CPU operation. Some instructions test and manipulate individual processor flags.

- A flag is **set** when it equals **1**; it is **clear** (or reset) when it equals **0**.

- Flags register is register with size 16bits or 32bits etch bit call flag and has different job. This flags determine the processor status after execute the instructions.

- 8086 processor with size 16bits determine the processor status in 9bits.
- 8086 has two types of flags:
  - Status Flags.
  - Control Flags.

- Status flags in flags register are represented in 6bits (0, 2, 4, 6, 7, 11).
- Control flags in flags register are represented in 3bits (8, 9, 10).

				Of	**Df**	**If**	**Tf**	Sf	Zf		Af		Pf		Cf

Status flags:

- This flags show the processor status after execute any command.
- Table show Status Flags and Control Flags:

Flag simple	**Name**	**Bit number**
Status Flags		
CF	Carry Flag	0
PF	Parity Flag	2
AF	Auxiliary Carry Flag	4
ZF	Zero Flag	6
SF	Sign Flag	7
OF	Overflow Flag	11
Control Flags		
TF	Trap Flag	8
IF	Interrupt Flag	9
DF	Direction Flag	10

Flags value:

- **Carry Flag:**
  - Value of this flag equal to 1 if we have a carry value from or to the Most Significant Bit (MSB), and this carry is happen in the add operation usually.

  - Carry flag also serves as a borrow flag for subtraction. In case of subtraction it is set when borrow is needed

  - Otherwise the value of flag is equal to 0.

- **Parity Flag:**
  - Value of this flag equal to 1 if Low Byte contain even number of bits contain value 1.
  - If Low Byte contain odd number of bits contain value 1 the value of flag is equal to 0.

- **Auxiliary Carry Flag:**
  - Value of this flag equal to 1 if we have a carry value from or to the fourth bit (bit-3), this flag used in the Binary Coded Decimal (BCD).
  - Otherwise the value of flag is equal to 0.

- **Zero Flag:**
  - Value of this flag equal to 1 when the result of operation equal zero.
  - Otherwise the value of flag is equal to 0.

- **Sign Flag:**
  - Value of this flag equal to 1 if the value of the Most Significant Bit (MSB) equal 1, which means the value is negative.
  - If the value of the Most Significant Bit (MSB) equal 0 the value of flag is equal to 0.

- **Overflow Flag:**
  - The Overflow flag is set by value 1 when the signed result of an operation is invalid or out of range.

  - This **flag is set if the result is out of range**. For addition this flag is set when there is a carry into the MSB and no carry out of the MSB or vice-versa. For subtraction, it is set when the MSB needs a borrow and there is no borrow from the MSB, or vice-versa.

  - When adding two integers, remember that the Overflow flag is only set when:
    - ✓ Two positive operands are added and their sum is negative.
    - ✓ Two negative operands are added and their sum is positive.

```
; Example 1
mov al,128
add al,1          ; OF = 1,   AL = ??

; Example 2
mov al,0FFh       ; OF = 1,   AL = 100h
add al,1
```

Example:

If AX contain the value ffffh and BX contain the value fffeh and execute the instruction: Add AX ,BX

```
  F F F F H
+ F F F E H
-----------
1 F F F C H
```

AX = FFFFH = 1111 1111 1111 1111
BX = FFFEH = 1111 1111 1111 1101
AX = FFFCH = 1111 1111 1111 1100

Flags value are:

CF = 1 because there is carry from the MSB.
PF = 1 because the low byte contain 14^{th} ones (even number of ones).
ZF = 0 because result is not equal zero.
SF = 1 because the MSB equal 1.
OF = 0 because result has the same sign.

Example:

If Al contain the value 80h and bl contain the value 80h and execute the instruction: Add al ,bl

```
  8 0 H
+ 8 0 H
-------
1 0 0 H
```

AL = 00H = 0000 0000

Flags value are:

CF = 1 because there is carry from the MSB.
PF = 1 because the low byte contain 0 ones (even number of ones).
ZF = 1 because result is equal zero.
SF = 0 because the MSB equal 0.
OF = 1 because result has the different sign.