# Lecture No.3

## Lecture Outlines

2.1 General Concepts
- 2.1.1 Basic Microcomputer Design
- 2.1.2 Instruction Execution Cycle
- 2.1.3 Reading from Memory
- 2.1.4 Loading and Executing a Program

2.2 32-Bit x86 Processors
- 2.2.1 Modes of Operation
- 2.2.2 Basic Execution Environment
- 2.2.3 x86 Memory Management

## 2.1 General Concepts

This chapter describes the architecture of the x86 processor family and its host computer system from a programmer's point of view. Included in this group are all Intel IA-32 and Intel 64 processors, such as the Intel Pentium and Core-Duo, as well as the Advanced Micro Devices (AMD) processors, such as Athlon, Phenom, Opteron, and AMD64. Assembly language is a great tool for learning how a computer works, and it requires you to have a working knowledge of computer hardware. To that end, the concepts and details in this chapter will help you to understand the assembly language code you write.

We strike a balance between concepts applying to all microcomputer systems and specifics about x86 processors. You may work on various processors in the future, so we expose you to broad concepts. To avoid giving you a superficial understanding of machine architecture, we focus on specifics of the x86, which will give you a solid grounding when programming in assembly language.

> If you want to learn more about the Intel IA-32 architecture, read *the Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*. It's a free download from the Intel web site (www.intel.com).

### 2.1.1 Basic Microcomputer Design

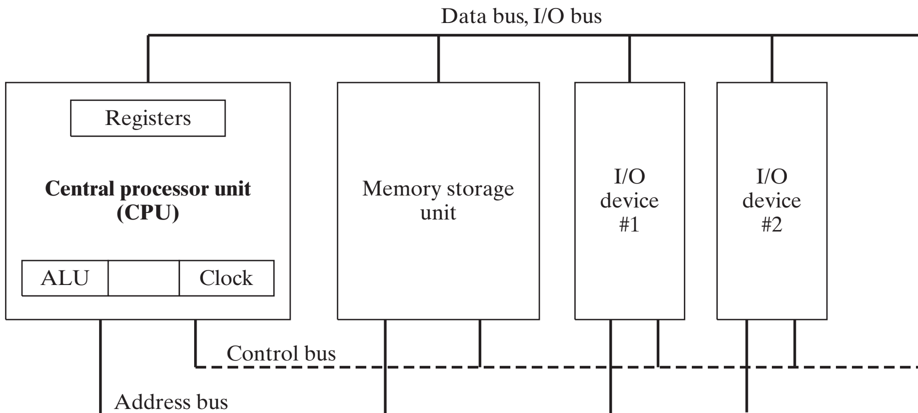
Figure 2-1 shows the basic design of a hypothetical microcomputer. The *central processor unit* (CPU), where calculations and logical operations take place, contains a limited number of storage locations named *registers*, a high-frequency clock, a control unit, and an arithmetic logic unit.

- The *clock* synchronizes the internal operations of the CPU with other system components.
- The *control unit* (CU) coordinates the sequencing of steps involved in executing machine instructions.
- The *arithmetic logic unit* (ALU) performs arithmetic operations such as addition and subtraction and logical operations such as AND, OR, and NOT.

The CPU is attached to the rest of the computer via pins attached to the CPU socket in the computer's motherboard. Most pins connect to the data bus, the control bus, and the address bus. The *memory storage unit* is where instructions and data are held while a computer program is running. The storage unit receives requests for data from the CPU, transfers data from random access memory (RAM) to the CPU, and transfers data from the CPU into memory. All processing of data takes place within the CPU, so programs residing in memory must be copied into the CPU before they can execute. Individual program instructions can be copied into the CPU one at a time, or groups of instructions can be copied together.

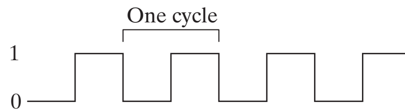
A *bus* is a group of parallel wires that transfer data from one part of the computer to another. A computer system usually contains four bus types: data, I/O, control, and address. The *data bus* transfers instructions and data between the CPU and memory. The I/O bus transfers data

Figure 2–1 Block diagram of a microcomputer.



between the CPU and the system input/output devices. The *control bus* uses binary signals to synchronize actions of all devices attached to the system bus. The *address bus* holds the addresses of instructions and data when the currently executing instruction transfers data between the CPU and memory.

*Clock* Each operation involving the CPU and the system bus is synchronized by an internal clock pulsing at a constant rate. The basic unit of time for machine instructions is a *machine cycle* (or *clock cycle*). The length of a clock cycle is the time required for one complete clock pulse. In the following figure, a clock cycle is depicted as the time between one falling edge and the next:




The duration of a clock cycle is calculated as the reciprocal of the clock's speed, which in turn is measured in oscillations per second. A clock that oscillates 1 billion times per second (1 GHz), for example, produces a clock cycle with a duration of one billionth of a second (1 nanosecond).

A machine instruction requires at least one clock cycle to execute, and a few require in excess of 50 clocks (the multiply instruction on the 8088 processor, for example). Instructions requiring memory access often have empty clock cycles called *wait states* because of the differences in the speeds of the CPU, the system bus, and memory circuits.

### 2.1.2 Instruction Execution Cycle

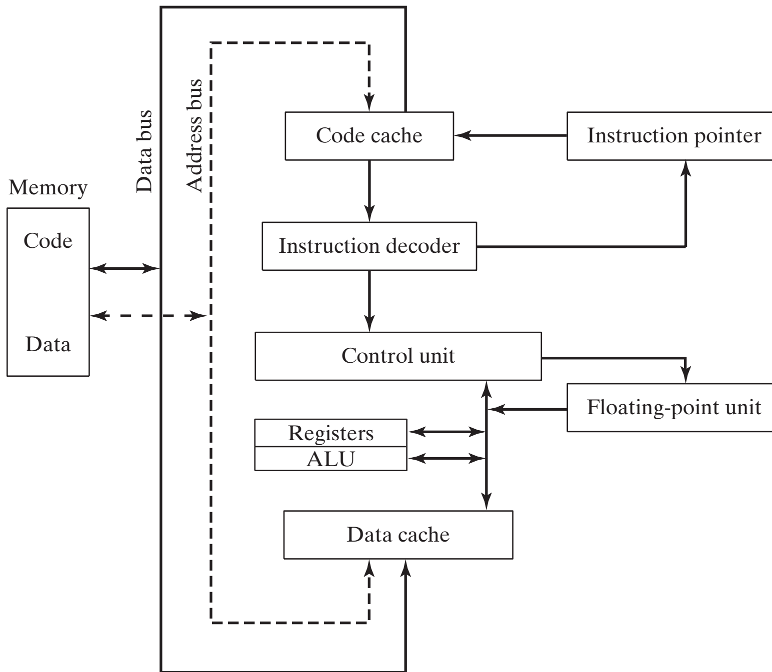
A single machine instruction does not just magically execute all at once. The CPU has to go through a predefined sequence of steps to execute a machine instruction, called the *instruction execution cycle*. Let's assume that the instruction pointer register holds the address of the instruction we want to execute. Here are the steps to execute it:

1. First, the CPU has to **fetch the instruction** from an area of memory called the *instruction queue*. Right after doing this, it increments the instruction pointer.
2. Next, the CPU **decodes** the instruction by looking at its binary bit pattern. This bit pattern might reveal that the instruction has operands (input values).
3. If operands are involved, the CPU **fetches the operands** from registers and memory. Sometimes, this involves address calculations.
4. Next, the CPU **executes** the instruction, using any operand values it fetched during the earlier step. It also updates a few status flags, such as Zero, Carry, and Overflow.
5. Finally, if an output operand was part of the instruction, the CPU **stores the result** of its execution in the operand.

We usually simplify this complicated-sounding process to three basic steps: **Fetch**, **Decode**, and **Execute**. An *operand* is a value that is either an input or an output to an operation. For example, the expression Z = X + Y has two input operands (X and Y) and a single output operand (Z).

A block diagram showing data flow within a typical CPU is shown in Figure 2-2. The diagram helps to show relationships between components that interact during the instruction execution cycle. In order to read program instructions from memory, an address is placed on the address bus. Next, the memory controller places the requested code on the data bus, making the code available inside the code cache. The instruction pointer's value determines which instruction will be executed next. The instruction is analyzed by the instruction decoder, causing the appropriate

Figure 2–2 Simplified CPU block diagram.




digital signals to be sent to the control unit, which coordinates the ALU and floating-point unit. Although the control bus is not shown in this figure, it carries signals that use the system clock to coordinate the transfer of data between the different CPU components.

### 2.1.3 Reading from Memory

As a rule, computers read memory much more slowly than they access internal registers. This is because reading a single value from memory involves four separate steps:

1. Place the address of the value you want to read on the address bus.
2. Assert (change the value of) the processor's RD *(read)* pin.
3. Wait one clock cycle for the memory chips to respond.
4. Copy the data from the data bus into the destination operand.

Each of these steps generally requires a single *clock cycle*, a measurement of time based on a

clock that ticks inside the processor at a regular rate. Computer CPUs are often described in terms of their clock speeds. A speed of *1.2 GHz*, for example, means the clock ticks, or oscillates, 1.2 billion times per second. So, 4 clock cycles go by fairly fast, considering each one lasts for only 1/1,200,000,000$^{th}$ of a second. Still, that's much slower than the CPU registers, which are usually accessed in only one clock cycle.

Fortunately, CPU designers figured out a long time ago that computer memory creates a speed bottleneck because most programs have to access variables. They came up with a clever way to reduce the amount of time spent reading and writing memory—they store the most recently used instructions and data in high-speed memory called *cache*. The idea is that a program is more likely to want to access the same memory and instructions repeatedly, so cache keeps these values where they can be accessed quickly. Also, when the CPU begins to execute a program, it can look ahead and load the next thousand instructions (for example) into cache, on the assumption that these instructions will be needed fairly soon. If there happens to be a loop in that block of code, the same instructions will be in cache. When the processor is able to find its data in cache memory, we call that a *cache hit*. On the other hand, if the CPU tries to find something in cache and it's not there, we call that a *cache miss*.

Cache memory for the x86 family comes in two types. *Level-1 cache* (or *primary cache*) is stored right on the CPU. *Level-2 cache* (or *secondary cache*) is a little bit slower, and attached to the CPU by a high-speed data bus. The two types of cache work together in an optimal way.

There's a reason why cache memory is faster than conventional RAM—it's because cache memory is constructed from a special type of memory chip called *static RAM*. It's expensive, but it does not have to be constantly refreshed in order to keep its contents. On the other hand, conventional memory, known as *dynamic RAM*, must be refreshed constantly. It's much slower, but cheaper.

### 2.1.4 Loading and Executing a Program

Before a program can run, it must be loaded into memory by a utility known as a *program loader*. After loading, the operating system must point the CPU to the program's *entry point*, which is the address at which the program is to begin execution. The following steps break this process down in more detail:

- The operating system (OS) searches for the program's filename in the current disk directory. If it cannot find the name there, it searches a predetermined list of directories (called *paths*) for the filename. If the OS fails to find the program filename, it issues an error message.
- If the program file is found, the OS retrieves basic information about the program's file from the disk directory, including the file size and its physical location on the disk drive.
- The OS determines the next available location in memory and loads the program file into memory. It allocates a block of memory to the program and enters information about the program's size and location into a table (sometimes called a *descriptor table*). Additionally, the OS may adjust the values of pointers within the program so they contain addresses of program data.

- The OS begins execution of the program's first machine instruction (its entry point). As soon as the program begins running, it is called a *process*. The OS assigns the process an identification number (*process ID*), which is used to keep track of it while running.
- The *process* runs by itself. It is the OS's job to track the execution of the process and to respond to requests for system resources. Examples of resources are memory, disk files, and input-output devices.
- When the process ends, it is removed from memory.

> ***Tip:*** If you're using any version of Microsoft Windows, press *Ctrl-Alt-Delete* and select the *Task Manager* item. The Task Manager window lets you view lists of Applications and Processes. Applications are the names of complete programs currently running, such as Windows Explorer or Microsoft Visual C++. When you click on the *Processes* tab, you see a long list of process names. Each of those processes is a small program running independently of all the others. You can continuously track the amount of CPU time and memory used by each process. In some cases, you can shut down a process by selecting its name and pressing the *Delete* key.

## 2.2 32-Bit x86 Processors

In this section, we focus on the basic architectural features of all x86 processors. This includes members of the Intel IA-32 family as well as all 32-bit AMD processors.

### 2.2.1 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:

*Protected Mode* Protected mode is the native state of the processor, 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 environment. In other words, if a program crashes or attempts to write data into the system memory area, it will not affect other programs running at the same time. A modern operating system can execute multiple separate virtual-8086 sessions at the same time.

*Real-Address Mode* Real-address mode implements the programming environment of an early Intel processor with a few extra features, such as the ability to switch into other modes. This mode is useful if a program requires direct access to system memory and hardware devices.

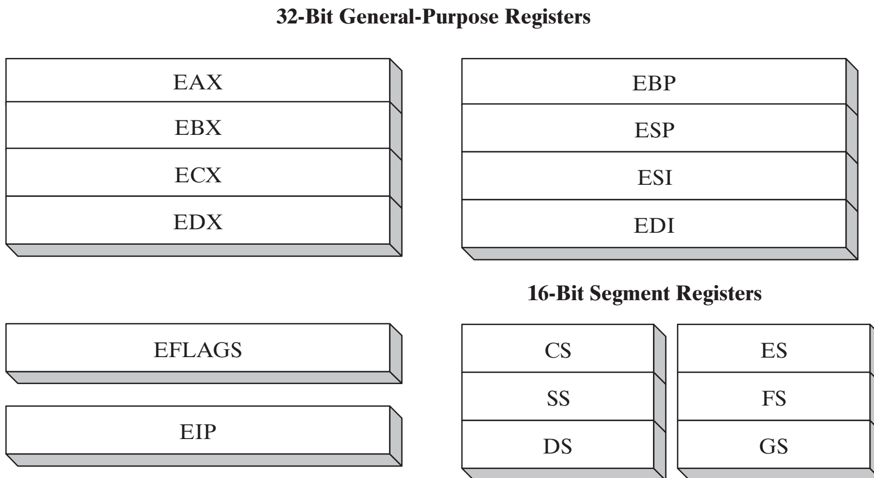
*System Management Mode* System management mode (SMM) provides an operating system with a mechanism for implementing functions such as power management and system security. These functions are usually implemented by computer manufacturers who customize the processor for a particular system setup.

### 2.2.2 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. 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.

FIGURE 2–3 Basic program execution registers.

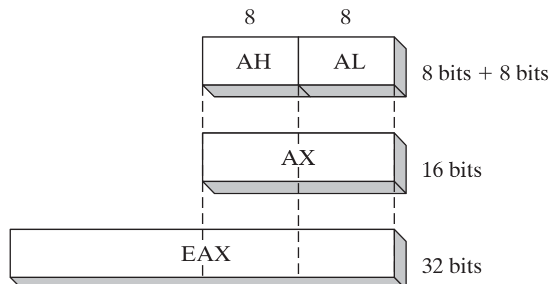



#### *Basic Program Execution Registers*

*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. Figure 2-3 shows the *basic program execution registers*. There are eight general-purpose registers, six segment registers, a processor status flags register (EFLAGS), and an instruction pointer (EIP).

*General-Purpose Registers* The *general-purpose registers* are primarily used for arithmetic and data movement. As shown in Figure 2-4, the lower 16 bits of the EAX register can be referenced by the name AX.

FIGURE 2–4 General-purpose registers.



Portions of some registers can be addressed as 8-bit values. For example, the AX register has an 8-bit upper half named AH and an 8-bit lower half named AL. The same overlapping relationship exists for the EAX, EBX, ECX, and EDX registers:

| 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 |

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* Some general-purpose registers have specialized uses:

- EAX is automatically used by multiplication and division instructions. It is often called the *extended accumulator* register.
- The CPU automatically uses ECX as a loop counter.
- ESP addresses data on the stack (a system memory structure). It is rarely used for ordinary arithmetic or data transfer. It is often called the *extended stack pointer* register.
- ESI and EDI are used by high-speed memory transfer instructions. They are sometimes called the *extended source index* and *extended destination index* registers.
- EBP is used by high-level languages to reference function parameters and local variables on the stack. It should not be used for ordinary arithmetic or data transfer except at an advanced level of programming. It is often called the *extended frame pointer* register.

*Segment Registers* In real-address mode, 16-bit segment registers indicate base addresses of preassigned memory areas named *segments*. In protected mode, segment registers hold pointers to segment descriptor tables. Some segments hold program instructions (code), others hold variables (data), and another segment named the *stack segment* holds local function variables and function parameters.

*Instruction Pointer* The EIP, or *instruction pointer*, register contains the address of the next instruction to be executed. Certain machine instructions manipulate EIP, causing the program to branch to a new location.

*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.

*Control Flags* Control flags control the CPU's operation. For example, they can cause the CPU to break after every instruction executes, interrupt when arithmetic overflow is detected, enter virtual-8086 mode, and enter protected mode.

Programs can set individual bits in the EFLAGS register to control the CPU's operation. Examples are the *Direction* and *Interrupt* flags.

*Status Flags* The status flags reflect the outcomes of arithmetic and logical operations performed by the CPU. They are the Overflow, Sign, Zero, Auxiliary Carry, Parity, and Carry flags. Their abbreviations are shown immediately after their names:

- The **Carry** flag (CF) is set when the result of an *unsigned* arithmetic operation is too large to fit into the destination.
- The **Overflow** flag (OF) is set when the result of a *signed* arithmetic operation is too large or too small to fit into the destination.
- The **Sign** flag (SF) is set when the result of an arithmetic or logical operation generates a negative result.
- The **Zero** flag (ZF) is set when the result of an arithmetic or logical operation generates a result of zero.
- The **Auxiliary Carry** flag (AC) is set when an arithmetic operation causes a carry from bit 3 to bit 4 in an 8-bit operand.
- The **Parity** flag (PF) is set if the least-significant byte in the result contains an even number of 1 bits. Otherwise, PF is clear. In general, it is used for error checking when there is a possibility that data might be altered or corrupted.

### *MMX Registers*

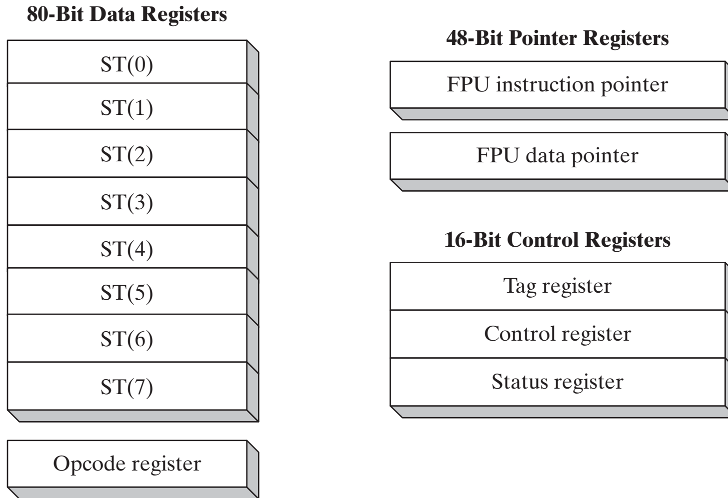
MMX technology improves the performance of Intel processors when implementing advanced multimedia and communications applications. The eight 64-bit MMX registers support special instructions called SIMD (*Single-Instruction, Multiple-Data*). As the name implies, MMX instructions operate in parallel on the data values contained in MMX registers. Although they appear to be separate registers, the MMX register names are in fact aliases to the same registers used by the floating-point unit.

### *XMM Registers*

The x86 architecture also contains eight 128-bit registers called XMM registers. They are used by streaming SIMD extensions to the instruction set.

*Floating-Point Unit* The *floating-point unit* (FPU) performs high-speed floating-point arithmetic. At one time a separate coprocessor chip was required for this. From the Intel486 onward, the FPU has been integrated into the main processor chip. There are eight floating-point data registers in the FPU, named ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), and ST(7). The remaining control and pointer registers are shown in Figure 2-5.

Figure 2–5 Floating-point unit registers.




### 2.2.3 x86 Memory Management

x86 processors manage memory according to the basic modes of operation discussed in Section 2.2.1. Protected mode is the most robust and powerful, but it does restrict application programs from directly accessing system hardware.

In ***real-address*** mode, only 1 MByte of memory can be addressed, from hexadecimal 00000 to FFFFF. The processor can run only one program at a time, but it can momentarily interrupt that program to process requests (called *interrupts*) from peripherals. Application programs are permitted to access any memory location, including addresses that are linked directly to system hardware. The MS-DOS operating system runs in real-address mode, and Windows 95 and 98 can be booted into this mode.

In ***protected*** mode, the processor can run multiple programs at the same time. It assigns each process (running program) a total of 4 GByte of memory. Each program can be assigned its own reserved memory area, and programs are prevented from accidentally accessing each other's code and data. MS-Windows and Linux run in protected mode.

In ***virtual-8086*** mode, the computer runs in protected mode and creates a virtual-8086 machine with its own 1-MByte address space that simulates an 80x86 computer running in real-address mode. Windows NT and 2000, for example, create a virtual-8086 machine when you open a *Command* window. You can run many such windows at the same time, and each is protected from the actions of the others. Some MS-DOS programs that make direct references to computer hardware will not run in this mode under Windows NT, 2000, and XP.