2

x86 Processor Architecture

This chapter focuses on the underlying hardware associated with x86 assembly language. It may be said that assembly language is the ideal software tool for communicating directly with a machine. If that is true, then assembly programmers must be intimately familiar with the processor's internal architecture and capabilities. We will discuss some of the basic operations that take place inside the processor when instructions are executed. We will talk about how programs are loaded and executed by the operating system. A sample motherboard layout will give some insight into the hardware environment of x86 systems, and the chapter ends with a discussion of how layered input/output works between application programs and operating systems. All of the topics in this chapter provide the hardware foundation for you to begin writing assembly language programs.

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

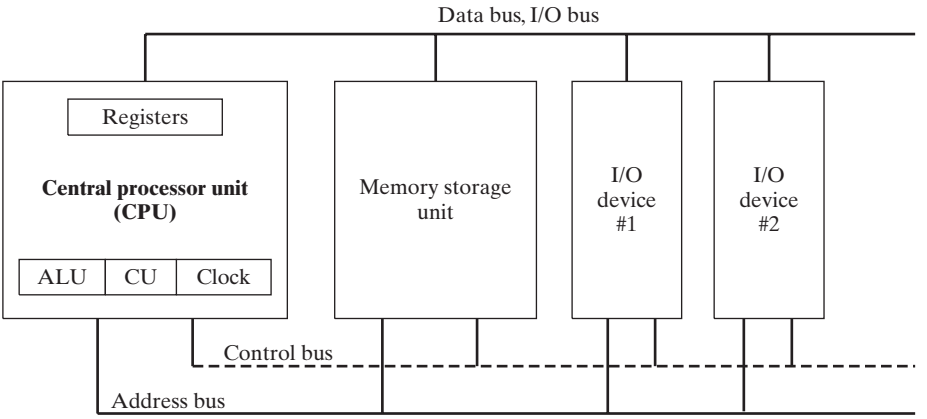
Figure 2-1 shows the basic design of a hypothetical microcomputer. The *central processor unit* (CPU), where calculations and logical operations take place, contains a limited number of storage locations named *registers*, a high-frequency clock, a control unit, and an arithmetic logic unit.

- 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.

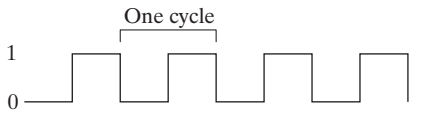
A *bus* is a group of parallel wires that transfer data from one part of the computer to another. A computer system usually contains four bus types: data, I/O, control, and address. The *data bus* transfers instructions and data between the CPU and memory. The I/O bus transfers data

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

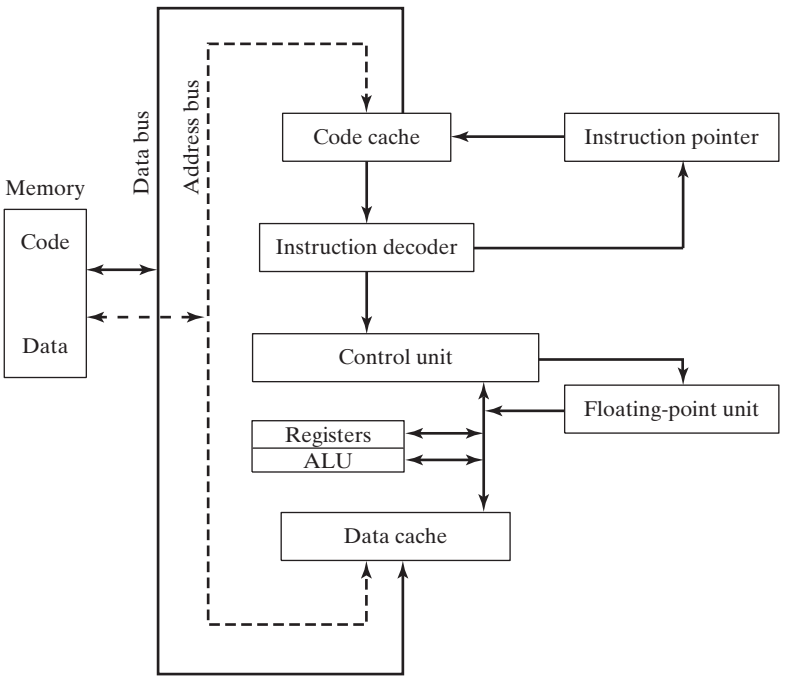
A single machine instruction does not just magically execute all at once. The CPU has to go through a predefined sequence of steps to execute a machine instruction, called the *instruction execution cycle*. Let's assume that the instruction pointer register holds the address of the instruction we want 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 1/1,200,000,000th 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.

2.1.5 Section Review

1. The central processor unit (CPU) contains registers and what other basic elements?
2. The central processor unit is connected to the rest of the computer system using what three buses?
3. Why does memory access take more machine cycles than register access?
4. What are the three basic steps in the instruction execution cycle?
5. Which two additional steps are required in the instruction execution cycle when a memory operand is used?

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.

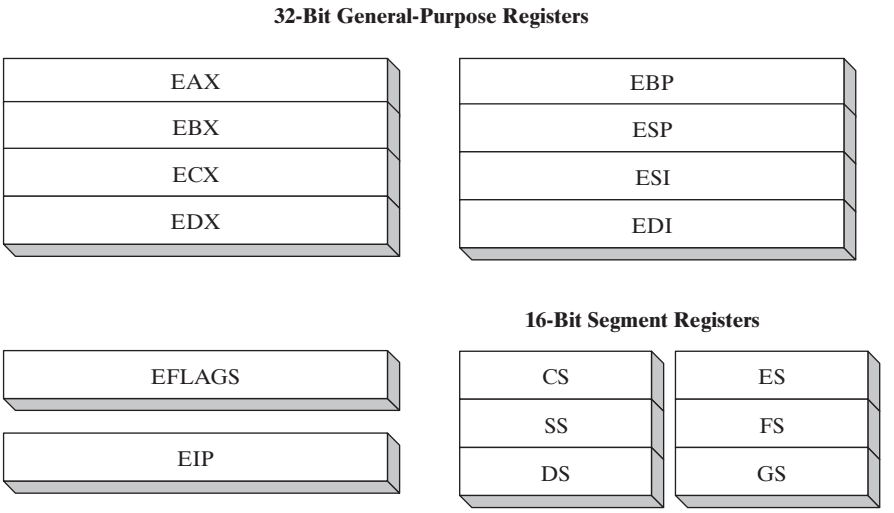
System Management Mode System management mode (SMM) provides an operating system with a mechanism for implementing functions such as power management and system security. These functions are usually implemented by computer manufacturers who customize the processor for a particular system setup.

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.

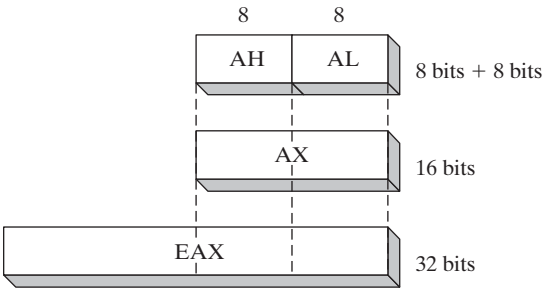


Basic Program Execution Registers

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

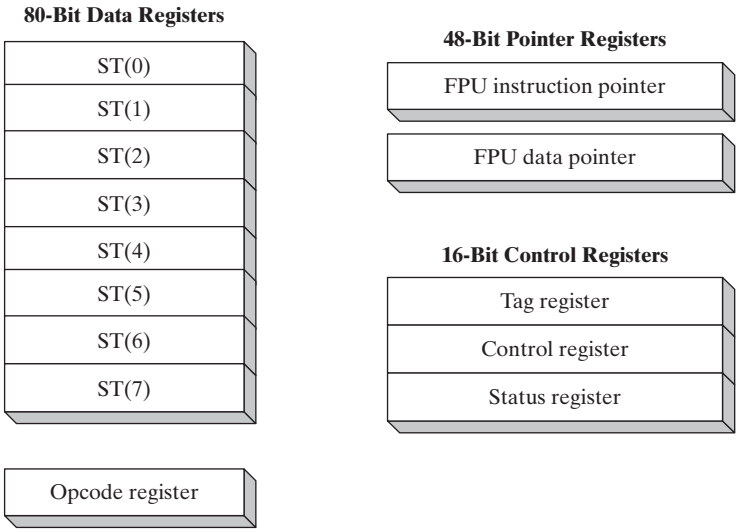
MMX technology improves the performance of Intel processors when implementing advanced multimedia and communications applications. The eight 64-bit MMX registers support special instructions called SIMD (*Single-Instruction, Multiple-Data*). As the name implies, MMX instructions operate in parallel on 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.

Chapter 11 explains many more details of both real-address mode and protected mode.

2.2.4 Section Review

1. What are the x86 processor's three basic modes of operation?
2. Name all eight 32-bit general-purpose registers.
3. Name all six segment registers.
4. What special purpose does the ECX register serve?

2.3 64-Bit x86-64 Processors

In this section, we focus on the basic architectural details of all 64-bit processors that use the x86-64 instruction set. This group the Intel 64 and AMD64 processor families. The instruction set is a 64-bit extension of the x86 instruction set we've already looked at. Here are some of the essential features:

1. It is backward-compatible with the x86 instruction set.
2. Addresses are 64 bits long, allowing for a virtual address space of size 2^{64} bytes. In current chip implementations, only the lowest 48 bits are used.
3. It can use 64-bit general-purpose registers, allowing instructions to have 64-bit integer operands.
4. It uses eight more general-purpose registers than the x86.
5. It uses a 48-bit physical address space, which supports up to 256 terabytes of RAM.

On the other hand, when running in native 64-bit mode, these processors do not support 16-bit real mode or virtual-8086 mode. (There is a *legacy mode* that still supports 16-bit programming, but it is not available in 64-bit versions of Microsoft Windows.)

> ***Note:*** Although *x86-64* refers to an instruction set, we will from this point on treat it as a processor type. For the purpose of learning assembly language, it is not necessary to consider hardware implementation differences between processors that support x86-64.

The first Intel processor to use x86-64 was the Xeon, followed by a host of other processors, including Core i5 and Core i7 processors. Examples of AMD's processors that use x86-64 are Opteron and Athlon 64.

You might also have heard of another 64-bit architecture from Intel known as *IA-64*, later renamed to *Itanium*. The IA-64 instruction set is completely different from x86 and x86-64. Itanium processors are often used for high-performance database and network servers.

2.3.1 64-Bit Operation Modes

The Intel 64 architecture introduces a new mode named *IA-32e*. Technically it contains two submodes, named *compatibility mode* and *64-bit mode*. But it's easier to refer to these as modes rather than submodes, so we will do that from now on.

Compatibility Mode

When running in compatibility mode, existing 16-bit and 32-bit applications can usually run without being recompiled. However, 16-bit Windows (Win16) and DOS applications will not run in 64-bit Microsoft Windows. Unlike earlier versions of Windows, 64-bit Windows does not have a virtual DOS machine subsystem to take advantage of the processor's ability to switch into virtual-8086 mode.

64-Bit Mode

In 64-bit mode, the processor runs applications that use the 64-bit linear address space. This is the native mode for 64-bit Microsoft Windows. This mode enables 64-bit instruction operands.

2.3.2 Basic 64-Bit Execution Environment

In 64-bit mode, addresses can theoretically be as large as 64-bits, although processors currently only support 48 bits for addresses. In terms of registers, the following are the most important differences from 32-bit processors:

- Sixteen 64-bit general purpose registers (in 32-bit mode, you have only eight general-purpose registers)
- Eight 80-bit floating-point registers
- A 64-bit status flags register named RFLAGS (only the lower 32 bits are used)
- A 64-bit instruction pointer named RIP

As you may recall, the 32-bit flags and instruction pointers are named EFLAGS and EIP. In addition, there are some specialized registers for multimedia processing we mentioned when talking about the x86 processor:

- Eight 64-bit MMX registers
- Sixteen 128-bit XMM registers (in 32-bit mode, you have only 8 of these)

General-Purpose Registers

The general-purpose registers, introduced when we described 32-bit processors, are the basic operands for instructions that do arithmetic, move data, and loop through data. The general-purpose registers can access 8-bit, 16-bit, 32-bit, or 64-bit operands (with a special prefix).

In 64-bit mode, the default operand size is 32 bits and there are eight general-purpose registers. By adding the REX (register extension) prefix to each instruction, however, the operands can be 64 bits long and a total of 16 general-purpose registers become available. You have all the

same registers as in 32-bit mode, plus eight numbered registers, R8 through R15. Table 2-1 shows which registers are available when the REX prefix is enabled.

Table 2-1 Operand Sizes in 64-Bit Mode When REX Is Enabled.

Operand Size	Available Registers
8 bits	AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L, R9L, R10L, R11L, R12L, R13L, R14L, R15L
16 bits	AX, BX, CX, DX, DI, SI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W
32 bits	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D
64 bits	RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, R15

Here are a few more details to remember:

- In 64-bit mode, a single instruction cannot access both a high-byte register, such as AH, BH, CH, and DH, and at the same time, the low byte of one of the new byte registers (such as DIL).
- The 32-bit EFLAGS register is replaced by a 64-bit RFLAGS register in 64-bit mode. The two registers share the same lower 32 bits, and the upper 32 bits of RFLAGS are not used.
- The status flags are the same in 32-bit mode and 64-bit mode.

2.4 Components of a Typical x86 Computer

Let us look at how the x86 integrates with other components by examining a typical motherboard configuration and the set of chips that surround the CPU. Then we will discuss memory, I/O ports, and common device interfaces. Finally, we will show how assembly language programs can perform I/O at different levels of access by tapping into system hardware, firmware, and by calling functions in the operating system.

2.4.1 Motherboard

The heart of a microcomputer is its *motherboard*, a flat circuit board onto which are placed the computer's CPU, supporting processors (*chipset*), main memory, input-output connectors, power supply connectors, and expansion slots. The various components are connected to each other by a *bus*, a set of wires etched directly on the motherboard. Dozens of motherboards are available on the PC market, varying in expansion capabilities, integrated components, and speed. The following components have traditionally been found on PC motherboards:

- A CPU socket. Sockets are different shapes and sizes, depending on the type of processor they support
- Memory slots (SIMM or DIMM), holding small plug-in memory boards
- BIOS (*basic input–output system*) computer chips, holding system software
- CMOS RAM, with a small circular battery to keep it powered
- Connectors for mass-storage devices such as hard drives and CD-ROMs
- USB connectors for external devices
- Keyboard and mouse ports

- PCI bus connectors for sound cards, graphics cards, data acquisition boards, and other input–output devices

The following components are optional:

- Integrated sound processor
- Parallel and serial device connectors
- Integrated network adapter
- AGP bus connector for a high-speed video card

Following are some important support processors in a typical system:

- The *Floating-Point Unit* (FPU) handles floating-point and extended integer calculations.
- The 8284/82C284 *Clock Generator*, known simply as the *clock*, oscillates at a constant speed. The clock generator synchronizes the CPU and the rest of the computer.
- The 8259A *Programmable Interrupt Controller* (PIC) handles external interrupts from hardware devices, such as the keyboard, system clock, and disk drives. These devices interrupt the CPU and make it process their requests immediately.
- The 8253 *Programmable Interval Timer/Counter* interrupts the system 18.2 times per second, updates the system date and clock, and controls the speaker. It is also responsible for constantly refreshing memory because RAM memory chips can remember their data for only a few milliseconds.
- The 8255 *Programmable Parallel Port* transfers data to and from the computer using the IEEE Parallel Port interface. This port is commonly used for printers, but it can be used with other input–output devices as well.

PCI and PCI Express Bus Architectures

The **PCI** (*Peripheral Component Interconnect*) bus provides a connecting bridge between the CPU and other system devices such as hard drives, memory, video controllers, sound cards, and network controllers. More recently, the *PCI Express* bus provides two-way serial connections between devices, memory, and the processor. It carries data in packets, similar to networks, in separate "lanes." It is widely supported by graphics controllers, and can transfer data at very high speeds.

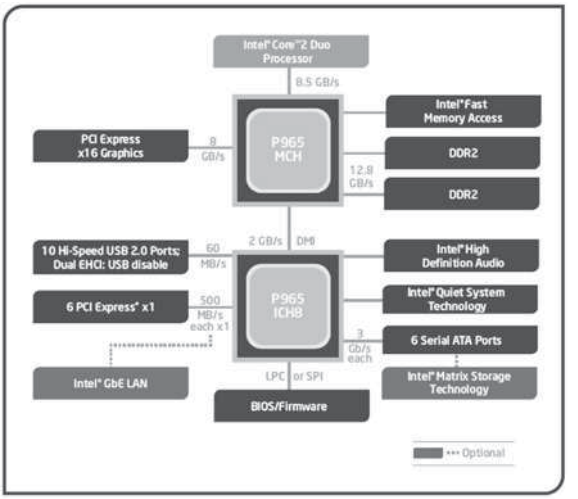
Motherboard Chipset

A *motherboard chipset* is a collection of processor chips designed to work together on a specific type of motherboard. Various chipsets have features that increase processing power, multimedia capabilities, or reduce power consumption. The *Intel P965 Express Chipset* can be used as an example. It is used in desktop PCs, with either an Intel Core 2 Duo or a Pentium D processor. Here are some of its features:

- Intel *Fast Memory Access* uses an updated Memory Controller Hub (MCH). It can access dual-channel DDR2 memory, at an 800 MHz clock speed.
- An I/O Controller Hub (Intel ICH8/R/DH) uses Intel Matrix Storage Technology (MST) to support multiple Serial ATA devices (disk drives).
- Support for multiple USB ports, multiple PCI express slots, networking, and Intel Quiet System technology.
- A high definition audio chip provides digital sound capabilities.

A diagram may be seen in Figure 2-6. Motherboard manufacturers will build products around specific chipsets. For example, the P5B-E P965 motherboard by Asus Corporation uses the P965 chipset.

FIGURE 2–6 Intel 965 express chipset block diagram.



Source: The Intel P965 Express Chipset (product brief), © 2006 by Intel Corporation, used by permission.

2.4.2 Memory

Several basic types of memory are used in Intel-based systems: read-only memory (ROM), erasable programmable read-only memory (EPROM), dynamic random-access memory (DRAM), static RAM (SRAM), video RAM (VRAM), and complimentary metal oxide semiconductor (CMOS) RAM:

- **ROM** is permanently burned into a chip and cannot be erased.
- **EPROM** can be erased slowly with ultraviolet light and reprogrammed.
- **DRAM**, commonly known as main memory, is where programs and data are kept when a program is running. It is inexpensive, but must be refreshed every millisecond to avoid losing its contents. Some systems use ECC (error checking and correcting) memory.
- **SRAM** is used primarily for expensive, high-speed cache memory. It does not have to be refreshed. CPU cache memory is comprised of SRAM.
- **VRAM** holds video data. It is dual ported, allowing one port to continuously refresh the display while another port writes data to the display.
- **CMOS RAM** on the system motherboard stores system setup information. It is refreshed by a battery, so its contents are retained when the computer's power is off.

2.4.3 Section Review

1. Describe SRAM and its most common use.
2. Describe VRAM.

3. List at least two features found in the Intel P965 Express chipset.
4. Name four types of RAM mentioned in this chapter.
5. What is the purpose of the 8259A PIC controller?

2.5 Input–Output System

Tip: Because computer games are so memory and I/O intensive, they push computer performance to the max. Programmers who excel at game programming often know a lot about video and sound hardware, and optimize their code for hardware features.

2.5.1 Levels of I/O Access

Application programs routinely read input from keyboard and disk files and write output to the screen and to files. I/O need not be accomplished by directly accessing hardware—instead, you can call functions provided by the operating system. I/O is available at different access levels, similar to the virtual machine concept shown in Chapter 1. There are three primary levels:

- **High-level language functions:** A high-level programming language such as C++ or Java contains functions to perform input–output. These functions are portable because they work on a variety of different computer systems and are not dependent on any one operating system.
- **Operating system:** Programmers can call operating system functions from a library known as the API (*application programming interface*). The operating system provides high-level operations such as writing strings to files, reading strings from the keyboard, and allocating blocks of memory.
- **BIOS:** The *basic input–output system* is a collection of low-level subroutines that communicate directly with hardware devices. The BIOS is installed by the computer's manufacturer and is tailored to fit the computer's hardware. Operating systems typically communicate with the BIOS.

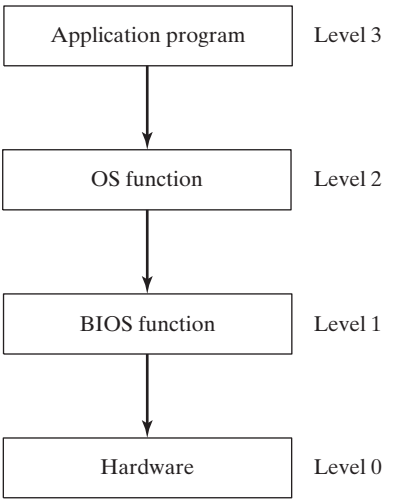
Device Drivers *Device drivers* are programs that permit the operating system to communicate directly with hardware devices and the system BIOS. For example, a device driver might receive a request from the OS to read some data; the device driver satisfies the request by executing code in the device firmware that reads data in a way that is unique to the device. Device drivers are usually installed in one of two ways: (1) before a specific hardware device is attached to a computer, or (2) after a device has been attached and identified. In the latter case, the OS recognizes the device name and signature; it then locates and installs the device driver software onto the computer.

We can put the I/O hierarchy into perspective by showing what happens when an application program displays a string of characters on the screen (Fig. 2-7). The following steps are involved:

1. A statement in the application program calls an HLL library function that writes the string to standard output.
2. The library function (Level 3) calls an operating system function, passing a string pointer.

3. The operating system function (Level 2) uses a loop to call a BIOS subroutine, passing it the ASCII code and color of each character. The operating system calls another BIOS subroutine to advance the cursor to the next position on the screen.
4. The BIOS subroutine (Level 1) receives a character, maps it to a particular system font, and sends the character to a hardware port attached to the video controller card.
5. The video controller card (Level 0) generates timed hardware signals to the video display that control the raster scanning and displaying of pixels.

FIGURE 2–7 Access levels for input–output operations.

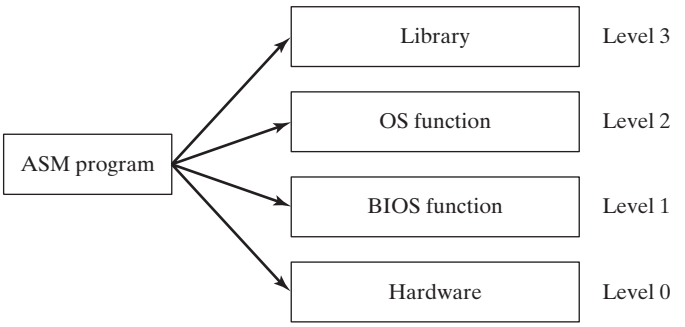


Programming at Multiple Levels Assembly language programs have power and flexibility in the area of input-output programming. They can choose from the following access levels (Figure 2-8):

- Level 3: Call library functions to perform generic text I/O and file-based I/O. We supply such a library with this book, for instance.
- Level 2: Call operating system functions to perform generic text I/O and file-based I/O. If the OS uses a graphical user interface, it has functions to display graphics in a device-independent way.
- Level 1: Call BIOS functions to control device-specific features such as color, graphics, sound, keyboard input, and low-level disk I/O.
- Level 0: Send and receive data from hardware ports, having absolute control over specific devices. This approach cannot be used with a wide variety of hardware devices, so we say that it is *not portable*. Different devices often use different hardware ports, so the program code must be customized for each specific type of device.

What are the tradeoffs? Control versus portability is the primary one. Level 2 (OS) works on any computer running the same operating system. If an I/O device lacks certain capabilities, the OS will do its best to approximate the intended result. Level 2 is not particularly fast because each I/O call must go through several layers before it executes.

Figure 2–8 Assembly language access levels.



Level 1 (BIOS) works on all systems having a standard BIOS, but will not produce the same result on all systems. For example, two computers might have video displays with different resolution capabilities. A programmer at Level 1 would have to write code to detect the user's hardware setup and adjust the output format to match. Level 1 runs faster than Level 2 because it is only one level above the hardware.

Level 0 (hardware) works with generic devices such as serial ports and with specific I/O devices produced by known manufacturers. Programs using this level must extend their coding logic to handle variations in I/O devices. Real-mode game programs are prime examples because they usually take control of the computer. Programs at this level execute as quickly as the hardware will permit.

Suppose, for example, you wanted to play a WAV file using an audio controller device. At the OS level, you would not have to know what type of device was installed, and you would not be concerned with nonstandard features the card might have. At the BIOS level, you would query the sound card (using its installed device driver software) and find out whether it belonged to a certain class of sound cards having known features. At the hardware level, you would fine tune the program for certain models of audio cards, taking advantage of each card's special features.

General-purpose operating systems rarely permit application programs to directly access system hardware, because to do so would make it nearly impossible for multiple programs to run simultaneously. Instead, hardware is accessed only by device drivers, in a carefully controlled manner. On the other hand, smaller operating systems for specialized devices often do connect directly to hardware. They do this in order to reduce the amount of memory taken up by operating system code, and they almost always run just a single program at one time. The last Microsoft operating system to allow programs to directly access hardware was MS-DOS, and it was only able to run one program at a time.

2.5.2 Section Review

1. Of the four levels of input/output in a computer system, which is the most universal and portable?
2. What characteristics distinguish BIOS-level input/output?
3. Why are device drivers necessary, given that the BIOS already has code that communicates with the computer's hardware?

4. In the example regarding displaying a string of characters, which level exists between the operating system and the video controller card?
5. Is it likely that the BIOS for a computer running MS-Windows would be different from that used by a computer running Linux?

2.6 Chapter Summary

The central processor unit (CPU) is where calculations and logic processing occur. It contains a limited number of storage locations called *registers*, a high-frequency clock to synchronize its operations, a control unit, and an arithmetic logic unit. The memory storage unit is where instructions and data are held while a computer program is running. A *bus* is a series of parallel wires that transmit data among various parts of the computer.

The execution of a single machine instruction can be divided into a sequence of individual operations called the *instruction execution cycle*. The three primary operations are fetch, decode, and execute. Each step in the instruction cycle takes at least one tick of the system clock, called a *clock cycle*. The *load and execute* sequence describes how a program is located by the operating system, loaded into memory, and executed by the operating system.

x86 processors have three basic modes of operation: *protected* mode, *real-address* mode, and *system management* mode. In addition, *virtual-8086* mode is a special case of protected mode. Intel64 processors have two basic modes of operation: *compatibility mode* and *64-bit mode*. In compatibility mode they can run 16-bit and 32-bit applications.

Registers are named locations within the CPU that can be accessed much more quickly than conventional memory. Following are brief descriptions of register types:

- The *general-purpose* registers are primarily used for arithmetic, data movement, and logical operations.
- The *segment registers* are used as base locations for preassigned memory areas called segments.
- The *instruction pointer* register contains the address of the next instruction to be executed.
- The *flags* register consists of individual binary bits that control the operation of the CPU and reflect the outcome of ALU operations.

The x86 has a floating-point unit (FPU) expressly used for the execution of high-speed floating-point instructions.

The heart of any microcomputer is its motherboard, holding the computer's CPU, supporting processors, main memory, input–output connectors, power supply connectors, and expansion slots. The PCI (Peripheral Component Interconnect) bus provides a convenient upgrade path for Pentium processors. Most motherboards contain an integrated set of several microprocessors and controllers, called a chipset. The chipset largely determines the capabilities of the computer.

Several basic types of memory are used in PCs: ROM, EPROM, Dynamic RAM (DRAM), Static RAM (SRAM), Video RAM (VRAM), and CMOS RAM.

Input–output is accomplished via different access levels, similar to the virtual machine concept. Library functions are at the highest level, and the operating system is at the next level

below. The BIOS (basic input–output system) is a collection of functions that communicate directly with hardware devices. Programs can also directly access input–output devices.

2.7 Key Terms

32-bit mode
64-bit mode
address bus
application programming interface (API)
arithmetic logic unit (ALU)
auxiliary carry flag
basic program execution registers
BIOS (basic input–output system)
bus
cache
carry flag
central processor unit (CPU)
clock
clock cycle
clock generator
code cache
control flags
control unit
data bus
data cache
device drivers
direction flag
dynamic RAM
EFLAGS register
extended destination index
extended physical addressing
extended source index
extended stack pointer
fetch-decode-execute
flags register
floating-point unit
general-purpose registers
instruction decoder
instruction execution cycle
instruction queue
instruction pointer
interrupt flag
Level-1 cache
Level-2 cache
machine cycle
memory storage unit
MMX registers
motherboard
motherboard chipset
operating system (OS)
overflow flag
parity flag
PCI (peripheral component interconnect)
PCI express
process
process ID
programmable interrupt controller (PIC)
programmable interval timer/counter
programmable parallel port
protected mode
random access memory (RAM)
read-only memory (ROM)
real-address mode
registers
segment registers
sign flag
single-instruction, multiple-data (SIMD)
static RAM
status flags
system management mode (SMM)
Task Manager
virtual-8086 mode
wait states
XMM registers
zero flag

2.8 Review Questions

1. In 32-bit mode, aside from the stack pointer (ESP), what other register points to variables on the stack?
2. Name at least four CPU status flags.
3. Which flag is set when the result of an *unsigned* arithmetic operation is too large to fit into the destination?
4. Which flag is set when the result of a *signed* arithmetic operation is either too large or too small to fit into the destination?
5. *(True/False):* When a register operand size is 32 bits and the REX prefix is used, the R8D register is available for programs to use.
6. Which flag is set when an arithmetic or logical operation generates a negative result?
7. Which part of the CPU performs floating-point arithmetic?
8. On a 32-bit processor, how many bits are contained in each floating-point data register?
9. *(True/False):* The x86-64 instruction set is backward-compatible with the x86 instruction set.
10. *(True/False):* In current 64-bit chip implementations, all 64 bits are used for addressing.
11. *(True/False):* The Itanium instruction set is completely different from the x86 instruction set.
12. *(True/False):* Static RAM is usually less expensive than dynamic RAM.
13. *(True/False):* The 64-bit RDI register is available when the REX prefix is used.
14. *(True/False):* In native 64-bit mode, you can use 16-bit real mode, but not the virtual-8086 mode.
15. *(True/False):* The x86-64 processors have 4 more general-purpose registers than the x86 processors.
16. *(True/False):* The 64-bit version of Microsoft Windows does not support virtual-8086 mode.
17. *(True/False):* DRAM can only be erased using ultraviolet light.
18. *(True/False):* In 64-bit mode, you can use up to eight floating-point registers.
19. *(True/False):* A bus is a plastic cable that is attached to the motherboard at both ends, but does not sit directly on the motherboard.
20. *(True/False):* CMOS RAM is the same as static RAM, meaning that it holds its value without any extra power or refresh cycles.
21. *(True/False):* PCI connectors are used for graphics cards and sound cards.
22. *(True/False):* The 8259A is a controller that handles external interrupts from hardware devices.
23. *(True/False):* The acronym PCI stands for *programmable component interface*.
24. *(True/False):* VRAM stands for virtual random access memory.
25. At which level(s) can an assembly language program manipulate input/output?
26. Why do game programs often send their sound output directly to the sound card's hardware ports?