2

x86 Processor Architecture

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 processors, such as the Intel Pentium and Core-Duo, as well as the Advanced Micro Devices (AMD) Athlon, Phenom, and Opteron processors. 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

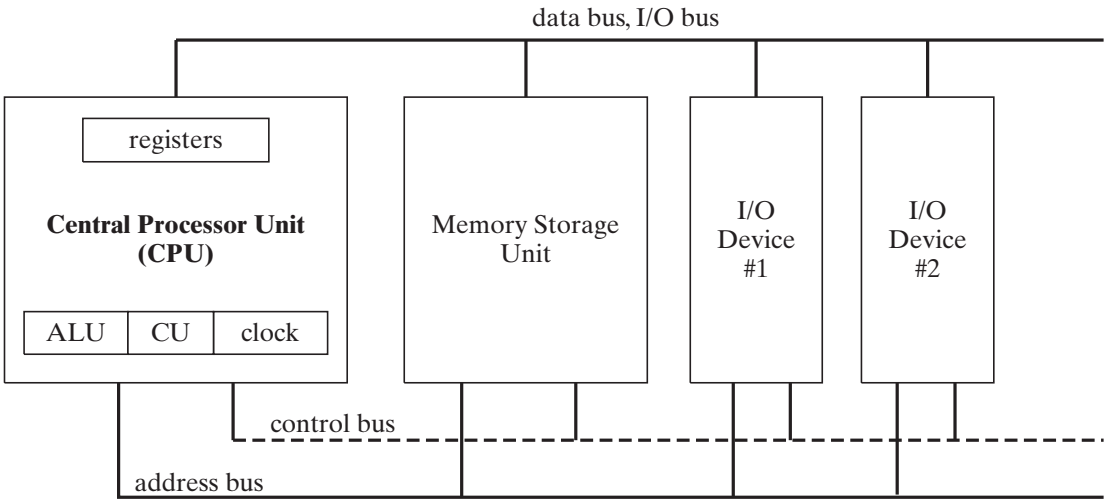
Figure 2–1 shows the basic design of a hypothetical microcomputer. The *central processor unit* (CPU), where calculations and logic operations take place, contains a limited number of storage locations named *registers*, a high-frequency clock, a control unit, and an arithmetic logic unit.

- 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.

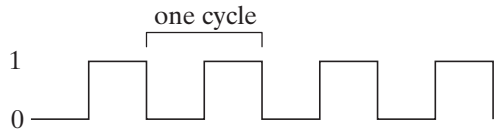
A *bus* is a group of parallel wires that transfer data from one part of the computer to another. A computer system usually contains four bus types: data, I/O, control, and address. The *data bus*

Figure 2–1 Block Diagram of a Microcomputer.



transfers instructions and data between the CPU and memory. The I/O bus transfers data 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

The execution of a single machine instruction can be divided into a sequence of individual operations called the *instruction execution cycle*. Before executing, a program is loaded into memory. The *instruction pointer* contains the address of the next instruction. The *instruction queue* holds a group of instructions about to be executed. Executing a machine instruction requires three basic steps: *fetch, decode,* and *execute.* Two more steps are required when the instruction uses a memory operand: *fetch operand* and *store output operand*. Each of the steps is described as follows:

- ***Fetch:*** The control unit fetches the next instruction from the instruction queue and increments the instruction pointer (IP). The IP is also known as the *program counter.*
- ***Decode:*** The control unit decodes the instruction's function to determine what the instruction will do. The instruction's input operands are passed to the ALU, and signals are sent to the ALU indicating the operation to be performed.
- ***Fetch operands:*** If the instruction uses an input operand located in memory, the control unit uses a *read* operation to retrieve the operand and copy it into internal registers. Internal registers are not visible to user programs.

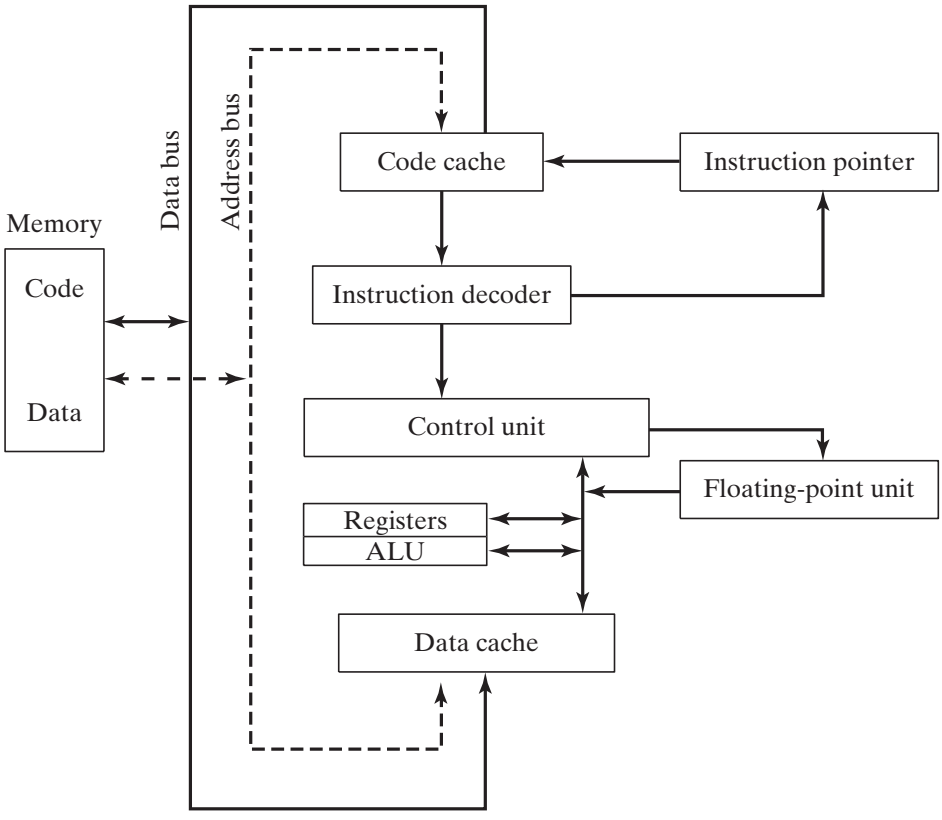
- ***Execute:*** The ALU executes the instruction using the named registers and internal registers as operands and sends the output to named registers and/or memory. The ALU updates status flags providing information about the processor state.
- ***Store output operand:*** If the output operand is in memory, the control unit uses a write operation to store the data.

The sequence of steps can be expressed neatly in pseudocode:

```
loop
     fetch next instruction
     advance the instruction pointer (IP)
     decode the instruction
     if memory operand needed, read value from memory
     execute the instruction
     if result is memory operand, write result to memory
continue loop
```

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.



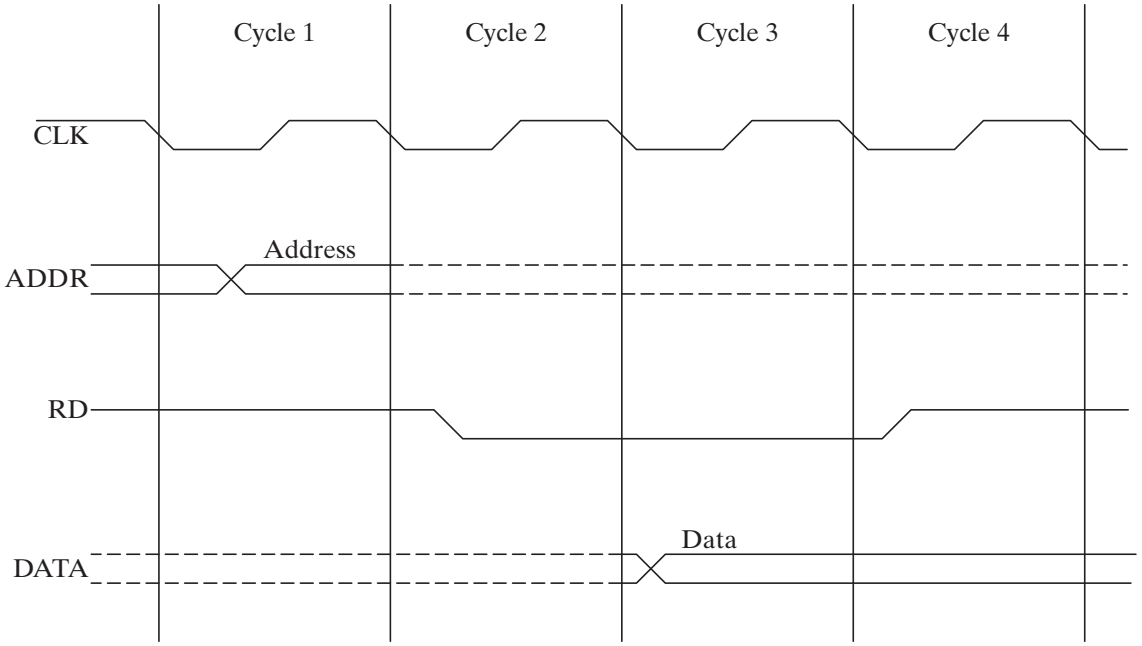
digital signals to be sent to the control unit, which coordinates the ALU and floating-point unit. Although the control bus is not shown in this figure, it carries signals that use the system clock to coordinate the transfer of data between the different CPU components.

2.1.3 Reading from Memory

Program throughput is often dependent on the speed of memory access. CPU clock speed might be several gigahertz, whereas access to memory occurs over a system bus running at a much slower speed. The CPU must wait one or more clock cycles until operands have been fetched from memory before the current instruction can complete its execution. The wasted clock cycles are called *wait states*.

Several steps are required when reading instructions or data from memory, controlled by the processor's internal clock. Figure 2–3 shows the processor clock (CLK) rising and falling at regular time intervals. In the figure, a clock cycle begins as the clock signal changes from high to low. The changes are called *trailing edges,* and they indicate the time taken by the transition between states.

Figure 2–3 Memory Read Cycle.



The following is a simplified description of what happens during each clock cycle during a memory read:

Cycle 1: The address bits of the memory operand are placed on the *address bus* (ADDR). The address lines in the diagram cross, showing that some bits equal 1 and others equal 0.

Cycle 2: The *read line* (RD) is set low (0) to notify memory that a value is to be read.

Cycle 3: The CPU waits one cycle to give memory time to respond. During this cycle, the memory controller places the operand on the *data bus* (DATA).

Cycle 4: The read line goes to 1, signaling the CPU to read the data on the data bus.

Cache Memory Because conventional memory is so much slower than the CPU, computers use high-speed *cache memory* to hold the most recently used instructions and data. The first time a program reads a block of data, it leaves a copy in the cache. If the program needs to read the same data a second time, it looks for the data in cache. A *cache hit* indicates the data is in cache; a *cache miss* indicates the data is not in cache and must be read from conventional memory. In general, cache memory has a noticeable effect on improving access to data, particularly when the cache is large.

2.1.4 How Programs Run

Load and Execute Process

The following steps describe, in sequence, what happens when a computer user runs a program at a command prompt:

- 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. 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.

If you're using any version of Microsoft Windows, press *Ctrl-Alt-Delete* and click on the *Task Manager* button. There are tabs labeled *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 30 or 40 names listed, often some you might not recognize. Each of those processes is a small program running independent of all the others. Note that each has a PID (process ID), and you can continuously track the amount of CPU time and memory it uses. Most processes run in the background. You can shut down a process somehow left running in memory by mistake. Of course, if you shut down the wrong process, your computer may stop running, and you'll have to reboot.

Multitasking

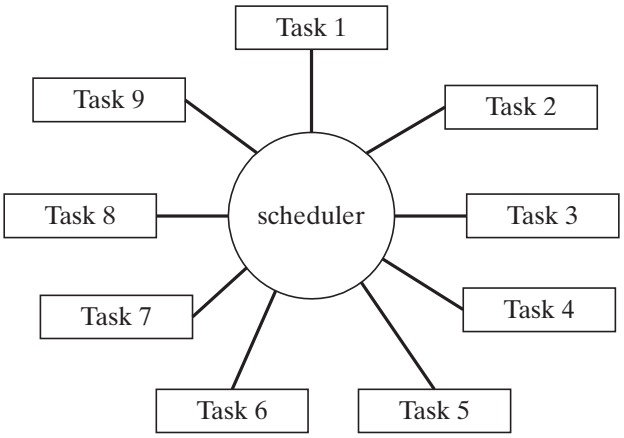
A *multitasking* operating system is able to run multiple tasks at the same time. A *task* is defined as either a program (a process) or a thread of execution. A process has its own memory area and may contain multiple threads. A thread shares its memory with other threads belonging to the same process. Game programs, for example, often use individual threads to simultaneously

control multiple graphic objects. Web browsers use separate threads to simultaneously load graphic images and respond to user input.

Most modern operating systems simultaneously execute tasks that communicate with hardware, display user interfaces, perform background file processing, and so on. A CPU can really execute only one instruction at a time, so a component of the operating system named the *scheduler* allocates a slice of CPU time (called a *time slice*) to each task. During a single time slice, the CPU executes a block of instructions, stopping when the time slice has ended.

By rapidly switching tasks, the processor creates the illusion they are running simultaneously. One type of scheduling used by the OS is called *round-robin scheduling*. In Figure 2–4, nine tasks are active. Suppose the *scheduler* arbitrarily assigned 100 milliseconds to each task, and switching between tasks consumed 8 milliseconds. One full circuit of the task list would require 972 milliseconds $(9 \times 100) + (9 \times 8)$ to complete.

Figure 2–4 Round-Robin Scheduler.



A multitasking OS runs on a processor (such as the x86) that supports *task switching*. The processor saves the state of each task before switching to a new one. A task's *state* consists of the contents of the processor registers, program counter, and status flags, along with references to the task's memory segments. A multitasking OS will usually assign varying priorities to tasks, giving them relatively larger or smaller time slices. A *preemptive* multitasking OS (such as Windows XP or Linux) permits a higher-priority task to interrupt a lower-priority one, leading to better system stability. Suppose an application program is locked in loop and has stopped responding to input. The keyboard handler (a high-priority OS task) can respond to the user's Ctrl-Alt-Del command and shut down the buggy application program.

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?

6. During which stage of the instruction execution cycle is the program counter incremented?
7. When a program runs, what information does the OS read from the filename's disk directory entry?
8. After a program has been loaded into memory, how does it begin execution?
9. Define *multitasking*.
10. What is the function of the OS scheduler?
11. When the processor switches from one task to another, what values in the first task's state must be preserved?
12. What is the duration of a single clock cycle in a 3-GHz processor?

2.2 x86 Architecture Details

In this section, we focus on the basic architectural features of the x86 processor family, which includes both Intel IA-32 and 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 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.

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).

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

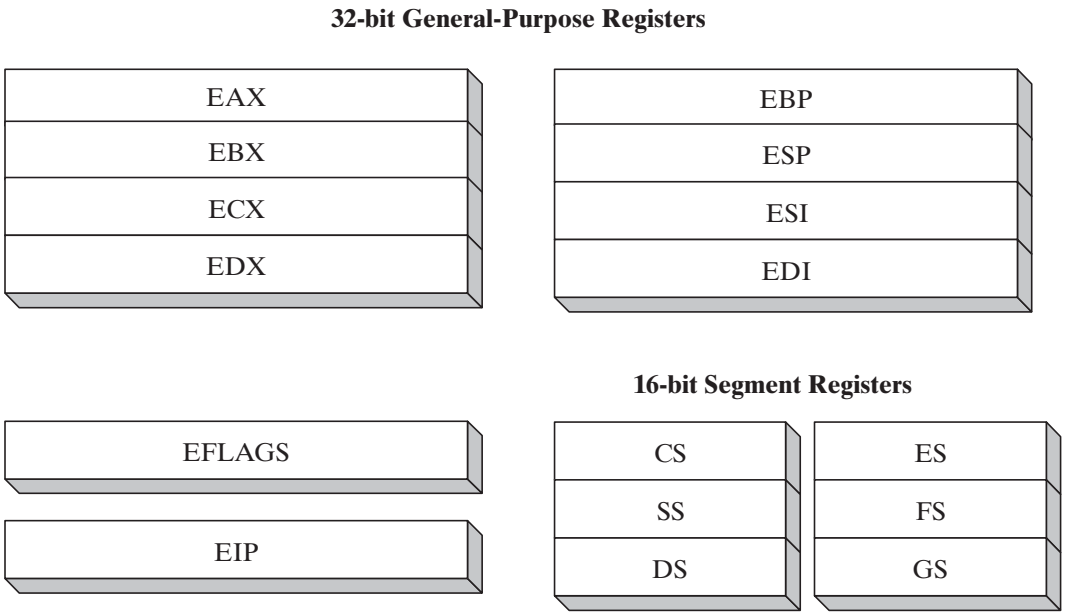
In 32-bit protected mode, a task or program can address a linear address space of up to 4 GBytes. Beginning with the P6 processor, a technique called Extended Physical Addressing allows a total of 64 GBytes of physical memory to be addressed. Real-address mode programs, on the other

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.

Basic Program Execution Registers

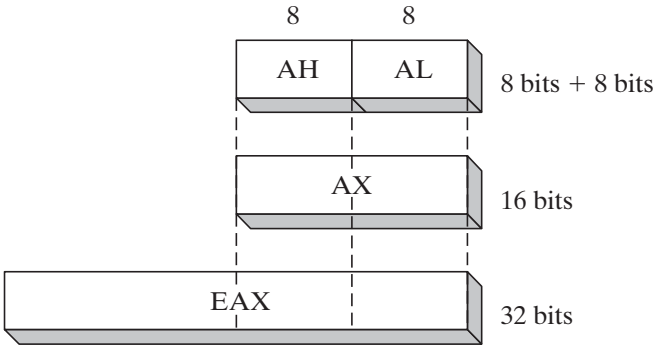
Registers are high-speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory. When a processing loop is optimized for speed, for example, loop counters are held in registers rather than variables. Figure 2–5 shows the *basic program execution registers*. There are eight general-purpose registers, six segment registers, a processor status flags register (EFLAGS), and an instruction pointer (EIP).

Figure 2–5 Basic Program Execution Registers.



General-Purpose Registers The *general-purpose registers* are primarily used for arithmetic and data movement. As shown in Figure 2–6, the lower 16 bits of the EAX register can be referenced by the name AX.

Figure 2–6 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 was added onto the Pentium processor by Intel to improve the performance of 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.

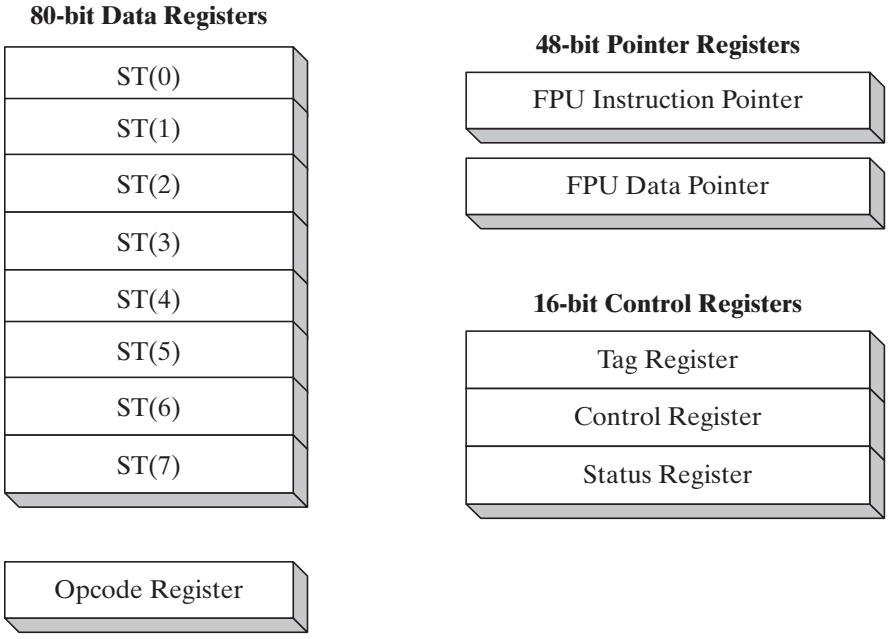
2.2.3 Floating-Point Unit

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 2–7.

2.2.4 Overview of Intel Microprocessors

Let's take a short trip down memory lane, starting when the IBM-PC was first released, when PC's had 64 KByte of RAM and no hard drives.

Figure 2–7 Floating-Point Unit Registers.



When discussing processor evolution, references to the data bus size are significant because they affect system performance. When a processor uses an 8-bit data bus to transfer a 32-bit integer to memory, for example, the integer must be broken into four parts, and four separate data transfer operations are required to complete the operation. Given that there is a significant delay (called *latency*) involved in each data transfer operation, an 8-bit data bus transfers data at one-fourth the speed of a 32-bit data bus.

Intel 8086 The Intel 8086 processor (1978) marked the beginning of the modern Intel architecture family. The primary innovations of the 8086 over earlier processors were that it had 16-bit registers and a 16-bit data bus and used a segmented memory model permitting programs to address up to 1 MByte of RAM. Greater access to memory made it possible to write complex business applications. The IBM-PC (1980) contained an Intel 8088 processor, which was identical to the 8086, except it had an 8-bit data bus that made it slightly less expensive to produce. Today, the Intel 8088 is used in low-cost microcontrollers.

Backward Compatibility. Each processor introduced into the Intel family since the 8086 has been backward-compatible with earlier processors. This approach enables older software to run (without recompilation) on newer computers without modification. Newer software eventually appeared, requiring features of more advanced processors.

Intel 80286 The Intel 80286 processor, first used in the IBM-PC/AT computer, set a new standard of speed and power. It was the first Intel processor to run in protected mode. The 80286 addresses up to 16 MByte of RAM using a 24-bit address bus.

IA-32 Processor Family (x86)

The Intel 80386 processor (1985) introduced 32-bit data registers and a 32-bit address bus and external data path. Here, we can distinguish between an internal data path, which is a bus that

moves data within the processor itself, and an external data path, which is the bus that moves data to and from memory and I/O devices. As such, it was the first member of the IA-32 family. IA-32 processors can address virtual memory larger than the computer's physical memory. Each program is assigned a 4-GByte (gigabyte) linear address space.

Intel i486 Continuing the IA-32 family, the Intel i486 processor (1989) featured an instruction set microarchitecture using pipelining techniques that permitted multiple instructions to be processed at the same time.

Pentium The Intel Pentium processor (1993) added many performance improvements, including a superscalar design with two parallel execution pipelines. Two instructions could be decoded and executed simultaneously. The Pentium used a 32-bit address bus and a 64-bit internal data path (inside the processor itself), and introduced MMX technology to the IA-32 family.

Intel64 for 64-bit Processing

Intel64 is the name given to Intel's implementation of the *x86-64 specification*, originally developed by AMD. Intel64 provides a 64-bit linear address space, although individual processors generally implement less than 64 bits. Their physical address space can be greater than 64 GBytes. Intel64 provides backward compatibility to run 32-bit programs with no performance penalty.

Intel64 was first used in the *Pentium Extreme* processor, and has continued in the Intel Xeon, Celeron D, Pentium D, Core 2, Core i7, and Atom processors, as well as newer generations of the Pentium 4. In addition to the Protected mode, Real-address mode, and System management modes of the IA-32 processor family, Intel64 processors support the *IA-32e mode*, designed for 64-bit processing.

IA-32e Mode IA-32e Mode has two sub-modes, designed to benefit users of 64-bit operating systems such as Windows Vista and Linux: Compatibility mode and 64-bit mode.

1. **Compatibility mode** permits legacy 16-bit and 32-bit applications to run without recompilation under a 64-bit operating system. Operand sizes are 16 and 32 bits, and the addressable range of memory is 4 GByte.
2. **64-bit mode** uses 64-bit addresses, 64-bit (and 32-bit) operands, a greater number of registers, and extensions to the instruction set to improve the processing of massive amounts of data. Memory segmentation is disabled, creating a flat 64-bit linear-address space.

Individual applications running at the same time can run in either Compatibility mode or 64-bit mode. But an application running in 64-bit mode cannot use the segmented or real-address modes.

Processor Families At the time of this book's publication, the following Intel processor families were currently the most widely used. To give you an idea of their relative power, some specifications are listed. These statistics become obsolete quickly, so consult the intel.com Web site for the latest information:

Intel Celeron—dual-core, 512 KByte L2 cache, up to 2.2 GHz, 800 MHz bus

Intel Pentium—dual-core, 2 MByte L2 cache, 1.6 to 2.7 GHz, 800 MHz bus

Core 2 Duo—2 processor cores, 1.8–3.33 GHz, 64 bit, 6 MByte L2 cache

Core 2 Quad—4 processor cores, up to 12 MByte L2 cache, 1333 MHz front side bus

Core i7—4 processor cores, (up to 2.93 GHz), 8 processing threads, 8 MByte smart cache, 3 channels DDR3 memory

Hyperthreading and Multi-core Processing

A *dual processor* system contains two separate physical computer processors, usually attached to the same motherboard with its own socket. The computer's operating system will schedule two separate tasks (processes or threads) to run at the same time, in parallel.

Intel *Hyper-Threading (HT)* technology allows two tasks to execute on a traditional single processor at the same time. This approach is less expensive than a dual processor system, and it makes efficient use of the processor's resources. In effect, a single physical processor is divided into two logical processors. The shared resources include cache, registers, and execution units. The Intel Xeon processor and some Pentium 4 processors use HT technology.

The term *Dual Core* refers to integrated circuit (IC) chips that contain two complete physical computer processor chips in the same IC package. Each processor has its own resources, and each has its own communication path to the computer system's front-side bus. Sometimes, dual-core processors also incorporate HT technology, causing them to appear as four logical processors, running four tasks simultaneously. Intel also offers packages containing more than two processors, called *multi core*.

CISC and RISC

The Intel 8086 processor was the first in a line of processors using a *Complex Instruction Set Computer* (CISC) design. The instruction set is large, and includes a wide variety of memory-addressing, shifting, arithmetic, data movement, and logical operations. Complex instruction sets permit compiled programs to contain a relatively small number of instructions. A major disadvantage to CISC design is that complex instructions require a relatively long time to decode and execute. An interpreter inside the CPU written in a language called *microcode* decodes and executes each machine instruction. Once Intel released the 8086, it became necessary for all subsequent Intel processors to be backward-compatible with the first one.

A completely different approach to microprocessor design is called *Reduced Instruction Set* (RISC). A RISC consists of a relatively small number of short, simple instructions that execute relatively quickly. Rather than using a microcode interpreter to decode and execute machine instructions, a RISC processor directly decodes and executes instructions using hardware. High-speed engineering and graphics workstations have been built using RISC processors for many years.

Because of the huge popularity of IBM-PC–compatible computers, Intel was able to lower the price of its processors and dominate the microprocessor market. At the same time, Intel recognized many advantages to the RISC approach and found a way to use RISC-like features, such as overlapping execution in the Pentium series. The x86 instruction set continues to expand and improve.

2.2.5 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?

5. Besides the stack pointer (ESP), what other register points to variables on the stack?
6. Name at least four CPU status flags.
7. Which flag is set when the result of an *unsigned* arithmetic operation is too large to fit into the destination?
8. Which flag is set when the result of a *signed* arithmetic operation is either too large or too small to fit into the destination?
9. Which flag is set when an arithmetic or logical operation generates a negative result?
10. Which part of the CPU performs floating-point arithmetic?
11. How many bits long are the FPU data registers?
12. Which Intel processor was the first member of the IA-32 family?
13. Which Intel processor first introduced superscalar execution?
14. Which Intel processor first used MMX technology?
15. Describe the CISC design approach.
16. Describe the RISC design approach.

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.

In Sections 2.3.1 and 2.3.2 we will explain details of both real-address mode and protected mode.

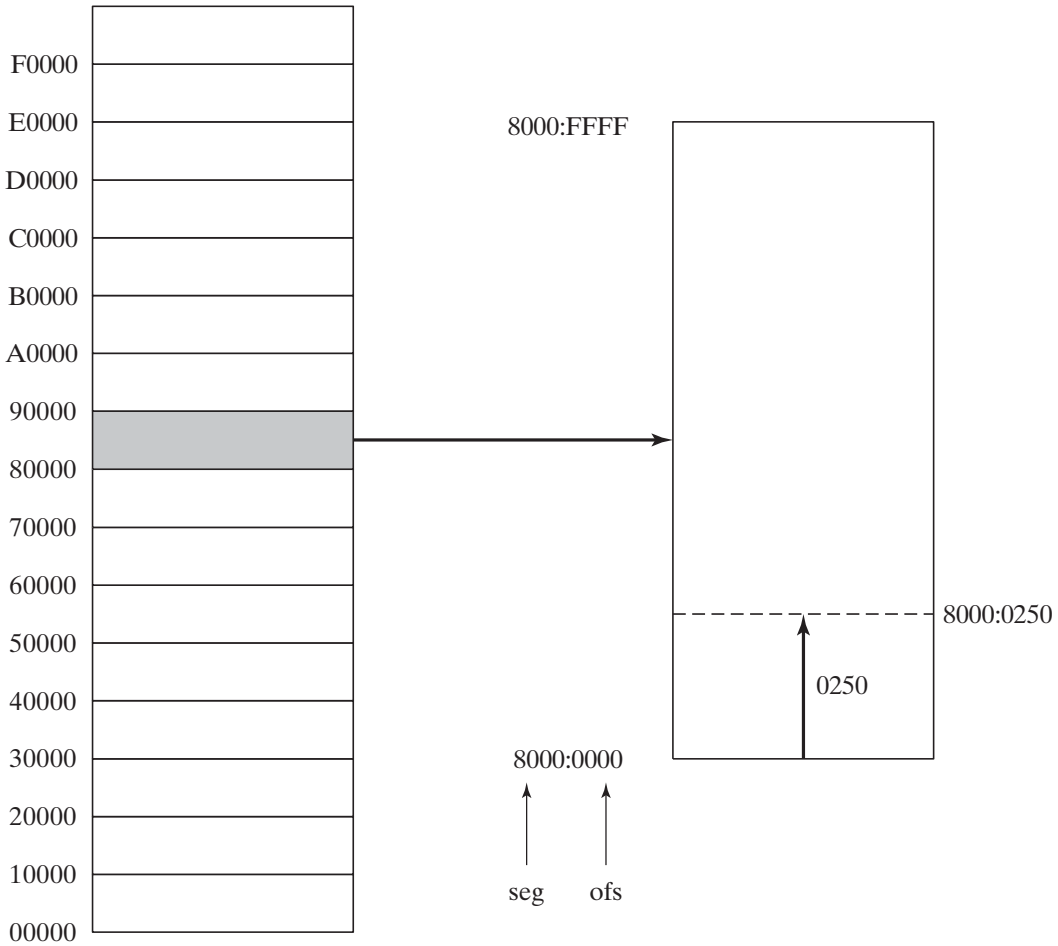
2.3.1 Real-Address Mode

In real-address mode, an x86 processor can access 1,048,576 bytes of memory (1 MByte) using 20-bit addresses in the range 0 to FFFFF hexadecimal. Intel engineers had to solve a basic

problem: The 16-bit registers in the Intel 8086 processor could not hold 20-bit addresses. They came up with a scheme known as *segmented memory*. All of memory is divided into 64-kilobyte (64-KByte) units called *segments*, shown in Figure 2–8. An analogy is a large building, in which *segments* represent the building's floors. A person can ride the elevator to a particular floor, get off, and begin following the room numbers to locate a room. The *offset* of a room can be thought of as the distance from the elevator to the room.

Again in Figure 2–8, each segment begins at an address having a zero for its last hexadecimal digit. Because the last digit is always zero, it is omitted when representing segment values. A segment value of C000, for example, refers to the segment at address C0000. The same figure shows an expansion of the segment at 80000. To reach a byte in this segment, add a 16-bit offset (0 to FFFF) to the segment's base location. The address 8000:0250, for example, represents an offset of 250 inside the segment beginning at address 80000. The linear address is 80250h.

Figure 2–8 Segmented Memory Map, Real-Address Mode.



20-Bit Linear Address Calculation An *address* refers to a single location in memory, and x86 processors permit each byte location to have a separate address. The term for this is *byte-addressable memory*. In real-address mode, the *linear* (or *absolute*) address is 20 bits, ranging

from 0 to FFFFF hexadecimal. Programs cannot use linear addresses directly, so addresses are expressed using two 16-bit integers. A *segment-offset* address includes the following:

- A 16-bit **segment** value, placed in one of the segment registers (CS, DS, ES, SS)
- A 16-bit **offset** value

The CPU automatically converts a segment-offset address to a 20-bit linear address. Suppose a variable's hexadecimal segment-offset address is 08F1:0100. The CPU multiplies the segment value by 16 (10 hexadecimal) and adds the product to the variable's offset:

```
08F1h × 10h = 08F10h          (adjusted segment value)
Adjusted Segment value:       0  8  F  1  0
Add the offset:                  0  1  0  0
Linear address:               0  9  0  1  0
```

A typical program has three segments: code, data, and stack. Three segment registers, CS, DS, and SS, contain the segments' base locations:

- CS contains the 16-bit **code** segment address
- DS contains the 16-bit **data** segment address
- SS contains the 16-bit **stack** segment address
- ES, FS, and GS can point to alternate data segments, that is, segments that supplement the default data segment

2.3.2 Protected Mode

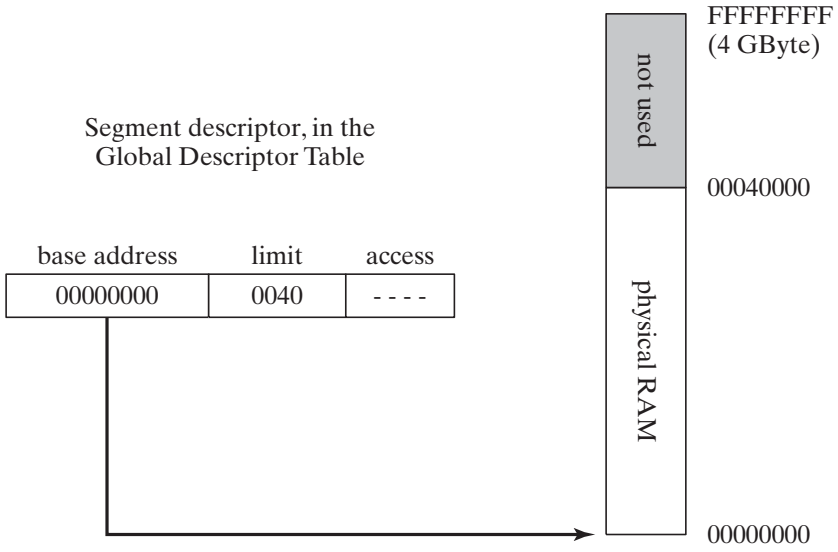
Protected mode is the more powerful "native" processor mode. When running in protected mode, a program's linear address space is 4 GBytes, using addresses 0 to FFFFFFFF hexadecimal. In the context of the Microsoft Assembler, the **flat** segmentation model is appropriate for protected mode programming. The flat model is easy to use because it requires only a single 32-bit integer to hold the address of an instruction or variable. The CPU performs address calculation and translation in the background, all of which are transparent to application programmers. Segment registers (CS, DS, SS, ES, FS, GS) point to *segment descriptor tables,* which the operating system uses to keep track of locations of individual program segments. A typical protected-mode program has three segments: code, data, and stack, using the CS, DS, and SS segment registers:

- CS references the descriptor table for the code segment
- DS references the descriptor table for the data segment
- SS references the descriptor table for the stack segment

Flat Segmentation Model

In the flat segmentation model, all segments are mapped to the entire 32-bit physical address space of the computer. At least two segments are required, one for code and one for data. Each segment is defined by a *segment descriptor*, a 64-bit integer stored in a table known as the *global descriptor table* (GDT). Figure 2–9 shows a segment descriptor whose *base address* field points to the first available location in memory (00000000). In this figure, the segment limit is 0040. The *access* field contains bits that determine how the segment can be used. All modern operating systems based on x86 architecture use the flat segmentation model.

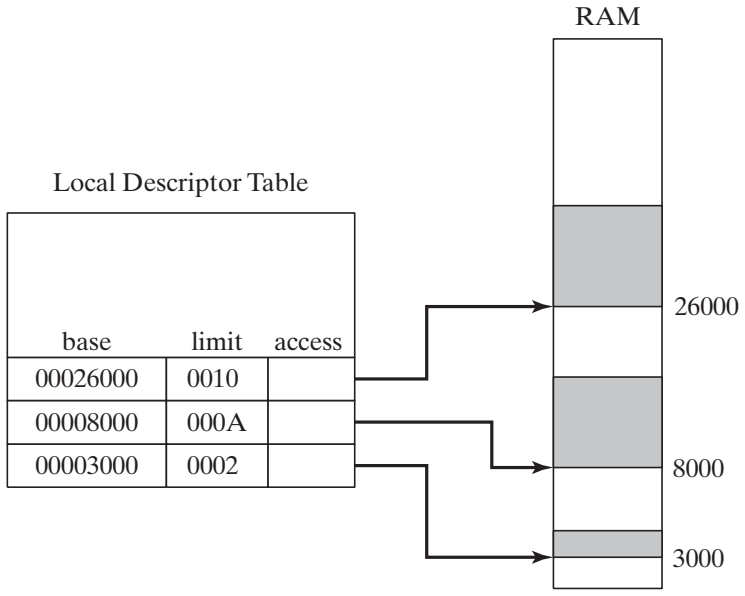
Figure 2–9 Flat Segmentation Model.



Multi-Segment Model

In the multi-segment model, each task or program is given its own table of segment descriptors, called a *local descriptor table* (LDT). Each descriptor points to a segment, which can be distinct from all segments used by other processes. Each segment has its own address space. In Figure 2–10, each entry in the LDT points to a different segment in memory. Each segment descriptor specifies the exact size of its segment. For example, the segment beginning at 3000 has size 2000 hexadecimal, which is computed as (0002 × 1000 hexadecimal). The segment beginning at 8000 has size A000 hexadecimal.

Figure 2–10 Multi-Segment Model.



Paging

x86 processors support *paging*, a feature that permits segments to be divided into 4,096-byte blocks of memory called *pages*. Paging permits the total memory used by all programs running at the

same time to be much larger than the computer's physical memory. The complete collection of pages mapped by the operating system is called *virtual memory.* Operating systems have utility programs named *virtual memory managers.*

Paging is an important solution to a vexing problem faced by software and hardware designers. A program must be loaded into main memory before it can run, but memory is expensive. Users want to be able to load numerous programs into memory and switch among them at will. Disk storage, on the other hand, is cheap and plentiful. Paging provides the illusion that memory is almost unlimited in size. Disk access is much slower than main memory access, so the more a program relies on paging, the slower it runs.

When a task is running, parts of it can be stored on disk if they are not currently in use. Parts of the task are *paged* (swapped) to disk. Other actively executing pages remain in memory. When the processor begins to execute code that has been paged out of memory it issues a *page fault*, causing the page or pages containing the required code or data to be loaded back into memory. To see how this works, find a computer with somewhat limited memory and run many large applications at the same time. You should notice a delay when switching from one program to another because the operating system must transfer paged portions of each program into memory from disk. A computer runs faster when more memory is installed because large application files and programs can be kept entirely in memory, reducing the amount of paging.

2.3.3 Section Review

1. What is the range of addressable memory in protected mode?
2. What is the range of addressable memory in real-address mode?
3. The two ways of describing an address in real-address mode are segment-offset and _____.
4. In real-address mode, convert the following hexadecimal segment-offset address to a linear address: 0950:0100.
5. In real-address mode, convert the following hexadecimal segment-offset address to a linear address: 0CD1:02E0.
6. In MASM's flat segmentation model, how many bits hold the address of an instruction or variable?
7. In protected mode, which register references the descriptor for the stack segment?
8. In protected mode, which table contains pointers to memory segments used by a single program?
9. In the flat segmentation model, which table contains pointers to at least two segments?
10. What is the main advantage to using the paging feature of x86 processors?
11. *Challenge:* Can you think of a reason why MS-DOS was not designed to support protected-mode programming?
12. *Challenge:* In real-address mode, demonstrate two segment-offset addresses that point to the same linear address.

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 about 4 GByte per second.

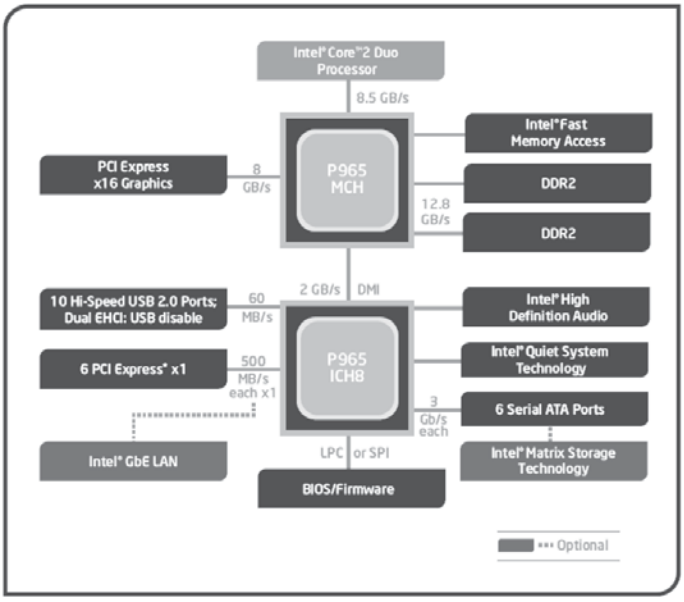
Motherboard Chipset

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 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 six Serial ATA devices (disk drives).
- Support for 10 USB ports, six PCI express slots, networking, Intel Quiet System technology.
- A high definition audio chip provides digital sound capabilities.

A diagram may be seen in Figure 2–11. Motherboard manufacturers will build products around specific chipsets. For example, the P5B-E P965 motherboard by Asus Corporation uses the P965 chipset.

Figure 2–11 Intel 965 Express Chipset Block Diagram.



Source: The Intel P965 Express Chipset (product brief),
© 2006 by Intel Corporation, used by permission.
http://www.intel.com/Assets/PDF/prodbrief/P965-prodbrief.pdf

2.4.2 Video Output

The video adapter controls the display of text and graphics. It has two components: the video controller and video display memory. All graphics and text displayed on the monitor are written into video display RAM, where it is then sent to the monitor by the video controller. The video controller is itself a special-purpose microprocessor, relieving the primary CPU of the job of controlling video hardware.

Older Cathode-ray tube (CRT) video displays used a technique called *raster scanning* to display images. A beam of electrons illuminates phosphorus dots on the screen called *pixels*. Starting at the top of the screen, the gun fires electrons from the left side to the right in a horizontal row, briefly turns off, and returns to the left side of the screen to begin a new row. *Horizontal retrace* refers to the time period when the gun is off between rows. When the last row is drawn, the gun turns off (called the *vertical retrace*) and moves to the upper left corner of the screen to start over.

A direct digital Liquid Crystal Display (LCD) panel, considered standard today, receives a digital bit stream directly from the video controller and does not require raster scanning. Digital displays generally display sharper text than analog displays.

2.4.3 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.

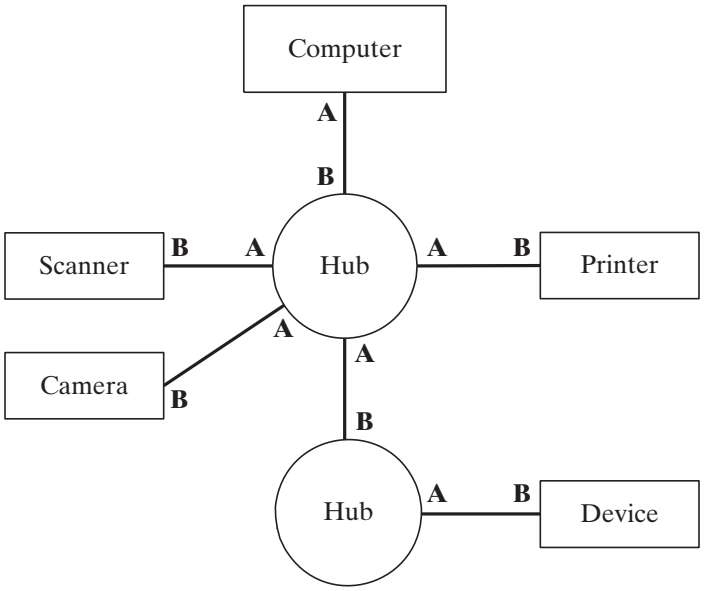
2.4.4 Input-Output Ports and Device Interfaces

Universal Serial Bus (USB) The Universal Serial Bus port provides intelligent, high-speed connection between a computer and USB-supported devices. USB Version 2.0 supports data transfer speeds of 480 megabits per second. You can connect single-function units (mice, printers) or compound devices having more than one peripheral sharing the same port. A USB hub, shown in Figure 2–12, is a compound device connected to several other devices, including other USB hubs.

When a device is attached to the computer via USB, the computer queries (enumerates) the device to get its name, device type, and the type of device driver it supports. The computer can suspend power to individual devices, putting them in a suspended state.

Parallel Port Printers have traditionally been connected to computers using *parallel ports*. The term *parallel* indicates that the bits in a data byte or word travel simultaneously from the computer to the device, each on a separate wire. Data is transferred at high speed (1 MByte per second) over short distances, usually no more than 10 feet. DOS automatically recognizes three parallel ports: LPT1, LPT2, and LPT3. Parallel ports can be *bidirectional*, allowing the computer to both send data to and receive information from a device. Although many printers now use USB connectors, parallel ports are useful for high-speed connections to laboratory instruments and custom hardware devices.

Figure 2–12 USB Hub Configuration.



ATA Host Adapters Known as *intelligent drive electronics* or *integrated drive electronics*, ATA host adapters connect computers to mass-storage devices such as hard drives and CD-ROMs. The letters ATA stand for *advanced technology attachment*, referring to the way the drive controller hardware and firmware are located on the drive itself. ATA adapters use a common interface named IDE (*integrated drive electronics*) found on all motherboards.

SATA Host Adapters SATA (*serial ATA*) host adapters have become the most common storage interface for laptop and desktop computers, replacing IDE and ATA interfaces. With only four signal lines, serial ATA uses an inexpensive high-speed cable that permits reading and writing data in both directions simultaneously.

FireWire FireWire is a high-speed external bus standard supporting data transfer speeds up to 800 MByte per second. A large number of devices can be attached to a single FireWire bus, and data can be delivered at a guaranteed rate (*isochronous data transfer*).

Serial Port An *RS-232 serial interface* sends binary bits one at a time, more slowly than parallel and USB ports, but with the ability to send over larger distances. The highest data transfer rate is 19,200 bits per second. Laboratory acquisition devices often use serial interfaces, as do

telephone modems. The 16550 UART (*Universal Asynchronous Receiver Transmitter*) chip controls serial data transfer.

Bluetooth Bluetooth is a wireless communication protocol for exchanging small amounts of data over short distances. It is commonly used with mobile devices such as cell phones and PDAs. It features low power consumption and can be implemented using low-cost microchips.

Wi-Fi Wi-Fi, sometimes known as *wireless Ethernet*, describes a certification asserting that a device can send data wirelessly to another Wi-Fi enabled device. Wi-Fi is based on industry-standard IEEE 802.11 standards. Wi-Fi devices operate at a greater speed and capacity than Bluetooth. Wi-Fi devices often communicate with each other when in range of a wireless network. For example, a wireless network can be established by a network router that has Wi-Fi capabilities. Most laptop computers sold today have built-in Wi-Fi capabilities.

2.4.5 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. Which type of RAM is used for Level 2 cache memory?
6. What advantages does a USB device offer over a standard serial or parallel device?
7. What is the purpose of the 8259A PIC controller?
8. What are the main differences between Wi-Fi and Bluetooth?

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.

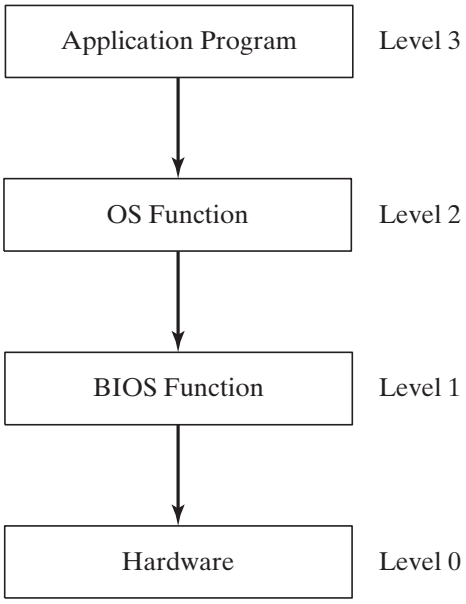
- **BIOS:** The Basic Input-Output System is a collection of low-level subroutines that communicate directly with hardware devices. The BIOS is installed by the computer's manufacturer and is tailored to fit the computer's hardware. Operating systems typically communicate with the BIOS.

Device Drivers *Device drivers* are programs that permit the operating system to communicate directly with hardware devices. 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 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 in (Figure 2–13). 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.

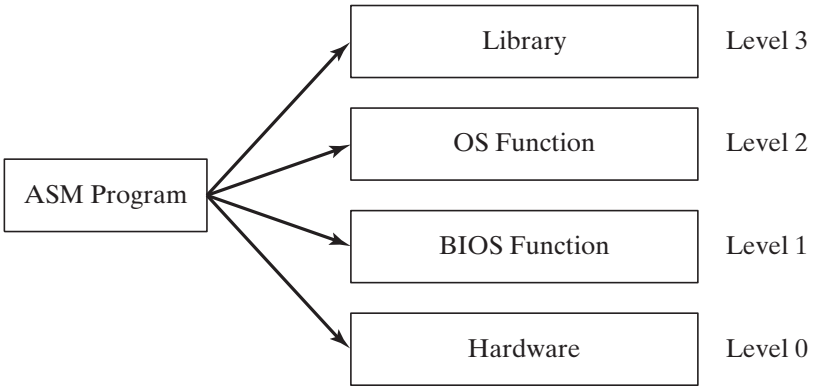
Figure 2–13 Access Levels for Input-Output Operations.



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 2–14):

- 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.

Figure 2–14 Assembly Language Access Levels.



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.

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.

Finally, not all operating systems permit user programs to directly access system hardware. Such access is reserved for the operating system itself and specialized device driver programs. This is the case with all versions of Microsoft Windows beyond Windows 95, in which vital system resources are shielded from application programs. MS-DOS, on the other hand, has no such restrictions.

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. At which level(s) can an assembly language program manipulate input/output?
6. Why do game programs often send their sound output directly to the sound card's hardware ports?
7. *Challenge:* 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 the 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.

A *multitasking* operating system can run multiple tasks at the same time. It runs on a processor that supports *task switching*, the ability to save the current task state and transfer control to a different task.

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 EIP (*instruction pointer)* register contains the address of the next instruction to be executed.
- The EFLAGS (*extended 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 Intel 8086 processor marked the beginning of the modern Intel architecture family. The Intel 80386 processor, the first of the IA-32 family, featured 32-bit registers and a 32-bit address bus and external data path. More recent processors, such as the Core 2 Duo, have multiple processor cores. They employ Intel Hyper-Threading technology to execute multiple tasks in parallel on the same processor core.

x86 processors are based on the *complex instruction set* (CISC) approach. The instruction set includes powerful ways to address data and instructions that are relatively high level complex operations. A completely different approach to microprocessor design is the *reduced instruction set* (RISC). A RISC machine language consists of a relatively small number of short, simple instructions that can be executed quickly by the processor.

In real-address mode, only 1 MByte of memory can be addressed, using hexadecimal addresses 00000 to FFFFF. 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 virtual memory. In virtual-8086 mode, the processor runs in protected mode and creates a virtual 8086 machine with its own 1-MByte address space that simulates an Intel 8086 computer running in real-address mode.

In the flat segmentation model, all segments are mapped to the entire physical address space of the computer. In the multi-segment model, each task is given its own table of segment descriptors, called a local descriptor table (LDT). x86 processors support a feature called *paging*, which permits a segment to be divided into 4096-byte blocks of memory called pages. Paging permits the total memory used by all programs running at the same time to be much larger than the computer's actual (physical) memory.

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.

The video adapter controls the display of text and graphics on IBM-compatibles. It has two components: the video controller and video display memory.

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

The Universal Serial Bus (USB) port provides an intelligent, high-speed connection between a computer and USB-supported devices. A parallel port transmits 8 or 16 data bits simultaneously from one device to another. An RS-232 serial port sends binary bits one at a time, resulting in slower speeds than the parallel and USB ports.

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.

A simple instruction set can be designed in such a way that each instruction is the same length, it carries out most operations within registers, and only reads and writes memory from a single register. The RISC (reduced instruction set architecture) is modeled along these principles.

2.7 Chapter Exercises

The following exercises require you to look in Intel's online manuals relating to the Intel64 and IA-32 processor architecture.

1. What are some of the innovative characteristics of the P6 processor architecture?
2. Locate a description of the Intel NetBurst Microarchitecture in one of the Intel64 and IA-32 processor manuals. Read about and summarize the functions of the *Front End Pipeline*.
3. Briefly explain the meaning of *out of order execution*. Why is it useful?
4. In processors that use Intel's *Hyperthreading Technology*, what components are coupled with each logical processor?
5. What is the size of the physical address space in a processor that implements the Intel64 architecture?
6. What are the Intel *Virtual Machine Extensions*? Why are virtual machines useful today?